Red Hat Developer Toolset 3.x User Guide

Installing and Using Red Hat Developer Toolset

Jaromír Hradílek Matt Newsome Robert Krátký

Red Hat Developer Toolset 3.x User Guide

Installing and Using Red Hat Developer Toolset

Jaromír Hradílek
Red Hat Customer Content Services
jhradilek@redhat.com

Matt Newsome
Red Hat Software Engineering
mnewsome@redhat.com

Robert Krátký
Red Hat Customer Content Services
rkratky@redhat.com

Legal Notice

Abstract

Red Hat Developer Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. The Red Hat Developer Toolset User Guide provides an overview of this 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.

Part I. Introduction

Chapter 1. Red Hat Developer Toolset

1.1. About Red Hat Developer Toolset

Red Hat Developer Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. It 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. For detailed compatibility information, see Section 1.3, “Compatibility”.

Red Hat Developer Toolset does not replace the default system tools provided with Red Hat Enterprise Linux 6 or 7 when installed on those platforms. Instead, a parallel set of developer tools provides an alternative, newer version of those tools for optional use by developers. The default compiler and debugger, for example, remain those provided by the base Red Hat Enterprise Linux system.

What Is New in Red Hat Developer Toolset 3.1

Starting with Red Hat Developer Toolset 3.1, *Dockerfiles* are available for selected Red Hat Developer Toolset components. See Section 1.8, “Using Red Hat Developer Toolset Container Images” for more detailed information about this new feature and instructions on how to use it. See Table 1.1, “Red Hat Developer Toolset Components” for a complete list of components included in Red Hat Developer Toolset 3.1.

Table 1.1. Red Hat Developer Toolset Components

Name	Version	Description
Eclipse	4.4.2	An integrated development environment with a graphical user interface. [a]
GCC	4.9.2	A portable compiler suite with support for C, C++, and Fortran.
binutils	2.24	A collection of binary tools and other utilities to inspect and manipulate object files and binaries.
elfutils	0.161	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.
GDB	7.8.2	A command line debugger for programs written in C, C++, and Fortran.
ltrace	0.7.91	A debugging tool to display calls to dynamic libraries that a program makes. It can also monitor system calls executed by programs.
strace	4.8	A debugging tool to monitor system calls that a program uses and signals it receives.
memstomp	0.1.5	A debugging tool to identify calls to library functions with overlapping memory regions that are not allowed by various standards.
SystemTap	2.6	A tracing and probing tool to monitor the activities of the entire system without the need to instrument, recompile, install, and reboot.
Valgrind	3.10.1	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.

Name	Version	Description
OProfile	0.9.9	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	8.2.1	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.

Red Hat Developer Toolset differs from "Technology Preview" compiler releases previously supplied in Red Hat Enterprise Linux in two important respects:

1. Red Hat Developer Toolset can be used on multiple major and minor releases of Red Hat Enterprise Linux, as detailed in Section 1.3, "Compatibility".

2. Unlike Technology Preview compilers and other tools shipped in earlier Red Hat Enterprise Linux, Red Hat Developer Toolset is fully supported under Red Hat Enterprise Linux Subscription Level Agreements, is functionally complete, and is intended for production use.

Important bug fixes and security errata are issued to Red Hat Developer Toolset subscribers in a similar manner to Red Hat Enterprise Linux for two years from the release of each major version release. A new major version of Red Hat Developer Toolset is released annually, providing significant updates for existing components and adding major new components. A single minor release, issued six months after each new major version release, provides a smaller update of bug fixes, security errata, and new minor components.

Additionally, the Red Hat Enterprise Linux Application Compatibility Specification also applies to Red Hat Developer Toolset (subject to some constraints on the use of newer C++11 language features, detailed in Section B.2.1.5, "ABI Compatibility").

Important

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. Using a framework called **Software Collections**, an additional set of developer tools is installed into the **`/opt`** directory and is explicitly enabled by the user on demand using the **`scl`** utility.

1.2. Main Features

The Red Hat Developer Toolset version of the **GNU Compiler Collection** (**GCC**) provides the following features:

- Link-time optimization (LTO) has been improved in a number of ways, so that it is faster, consumes less memory, and generates smaller object files.
- Inter-procedural analysis (IPA) has been improved, and speculative devirtualization optimization has been added.
- Support for profiling code has been made more reliable.

- The Intel AVX-512 target architecture is now supported, as well as a number of new Intel microarchitectures.
- Support for C11 has been improved, and ISO C11 atomics, generic selections, and thread-local storage are now supported.
- A subset of the functionality of C++11 **auto** is now provided in GNU C through a new extension, **__auto_type**.
- The C and C++ compilers now support the OpenMP 4.0 specification.
- The **g++** compiler offers improved support for various features of the C++ standard, including generic lambdas, variable-length arrays, and digit separators.
- The C++ runtime library C++11 support has been improved and now includes experimental support for the upcoming ISO C++ standard, C++14.
- **GCC** now adds support for Cilk+, an extension to the C and C++ languages for parallel programming.

The version of the **GNU Debugger** (**GDB**) included in Red Hat Developer Toolset provides the following new features:

- Python scripting support has been enhanced in a number of ways.
- Several new commands and enhancements in **GDB/MI** have been added.
- Support for the CTF (*Common Trace Format*) has been added.
- The btrace record target has been enhanced in a number of ways.
- The remote protocol and **GDBserver**, the **GDB** remote stub, have been enhanced in a number of ways.

Additionally, the Red Hat Developer Toolset version of **binutils** provides these features:

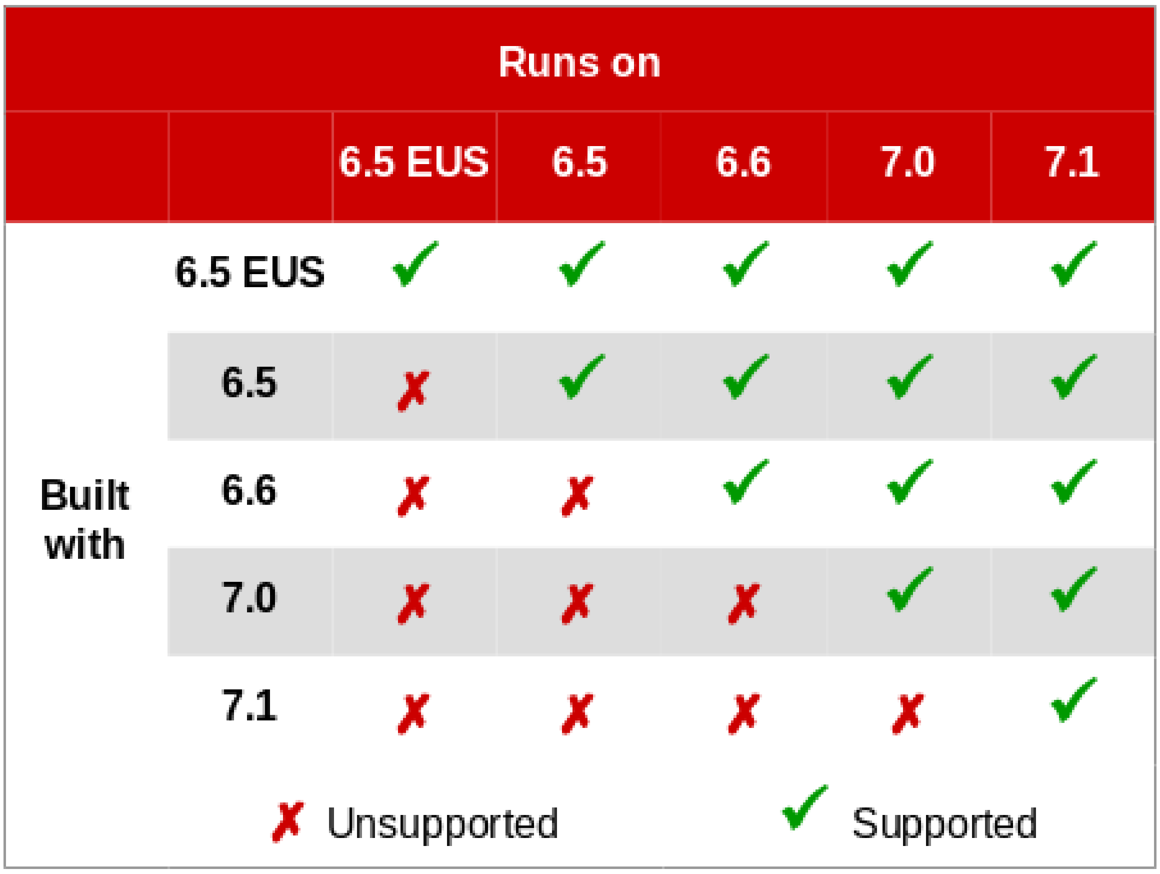
- The **objcopy** utility now supports wildcards for section names in command line options.
- The AVX-512 (512-bit Advanced Vector Extensions) are now supported.

For a full list of changes and features introduced in this release, see Appendix A, *Changes in Version 3.1*.

1.3. Compatibility

Red Hat Developer Toolset 3.1 is available for Red Hat Enterprise Linux 6 and 7 for 64-bit Intel and AMD architectures. Figure 1.1, “Red Hat Developer Toolset 3.1 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.

For ABI compatibility information, see Section B.2.1.5, “ABI Compatibility”.



		Runs on				
		6.5 EUS	6.5	6.6	7.0	7.1
Built with	6.5 EUS	✓	✓	✓	✓	✓
	6.5	✗	✓	✓	✓	✓
	6.6	✗	✗	✓	✓	✓
	7.0	✗	✗	✗	✓	✓
	7.1	✗	✗	✗	✗	✓

✗ Unsupported ✓ Supported

Figure 1.1. Red Hat Developer Toolset 3.1 Compatibility Matrix

1.4. Getting Access to Red Hat Developer Toolset

Red Hat Developer Toolset is an offering that is distributed as a part of the Red Hat Software Collections content set, which is available to customers with Red Hat Enterprise Linux 6 and 7 subscriptions listed at https://access.redhat.com/solutions/472793. Depending on the subscription management service with which you registered your Red Hat Enterprise Linux system, you can either enable Red Hat Developer Toolset by using the Red Hat Subscription Management, or by using RHN Classic.

For detailed instructions on how to enable Red Hat Software Collections (and thus gain access to Red Hat Developer Toolset) using RHN Classic or Red Hat Subscription Management, see the respective section below. For information on how to register your system with one of these subscription management services, see the Red Hat Subscription Management collection of guides.

Important

Red Hat Developer Toolset 3.1 is only compatible with Red Hat Enterprise Linux Server 6.4 and higher and Red Hat Enterprise Linux Workstation 6.5 and higher. See Figure 1.1, “Red Hat Developer Toolset 3.1 Compatibility Matrix”.

1.4.1. Using Red Hat Subscription Management

If your system is registered with Red Hat Subscription Management, complete the following steps to attach a subscription that provides access to the repository for Red Hat Software Collections (which includes Red Hat Developer Toolset), and then enable that repository:

1. Determine the pool ID of a subscription that provides Red Hat Software Collections (and thus also Red Hat Developer Toolset). To do so, type the following at a shell prompt as **root** to display a list of all subscriptions that are available for your system:

   ```
   subscription-manager list --available
   ```

   For each available subscription, this command displays its name, unique identifier, expiration date, and other details related to your subscription. The pool ID is listed on a line beginning with **Pool ID**.

   For a complete list of subscriptions that provide access to Red Hat Developer Toolset, see https://access.redhat.com/solutions/472793.

2. Attach the appropriate subscription to your system by running the following command as **root**:

   ```
   subscription-manager attach --pool=pool_id
   ```

   Replace *pool_id* with the pool ID you determined in the previous step. To verify the list of subscriptions your system has currently attached, at any time, run as **root**:

   ```
   subscription-manager list --consumed
   ```

3. Determine the exact name of the Red Hat Software Collections repository. To do so, type the following at a shell prompt as **root** to retrieve repository metadata and to display a list of available **Yum** repositories:

   ```
   subscription-manager repos --list
   ```

   The repository names depend on the specific version of Red Hat Enterprise Linux you are using and are in the following format:

   ```
   rhel-variant-rhscl-version-rpms
   rhel-variant-rhscl-version-debug-rpms
   rhel-variant-rhscl-version-source-rpms
   ```

   In addition, certain packages, such as *devtoolset-3-gcc-plugin-devel*, depend on packages that are only available in the **Optional** channel. The repository names with these packages use the following format:

   ```
   rhel-version-variant-optional-rpms
   rhel-version-variant-optional-debug-rpms
   rhel-version-variant-optional-source-rpms
   ```

   For both the regular repositories and optional repositories, replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**), and *version* with the Red Hat Enterprise Linux system version (**6-eus**, **6**, or **7**).

4. Enable the repositories from step no. 3 by running the following command as **root**:

   ```
   subscription-manager repos --enable repository
   ```

   Replace *repository* with the name of the repository to enable.

Once the subscription is attached to the system, you can install Red Hat Developer Toolset as described in Section 1.5, “Installing Red Hat Developer Toolset”. For more information on how to register your system using Red Hat Subscription Management and associate it with subscriptions, see the Red Hat Subscription Management collection of guides.

1.4.2. Using RHN Classic

If you are running Red Hat Enterprise Linux 6, and your system is registered with RHN Classic, complete the following steps to subscribe to Red Hat Software Collections (which includes Red Hat Developer Toolset):

1. Determine the exact name of the Red Hat Software Collections channel. To do so, type the following at a shell prompt as **root** to display a list of all channels that are available to you:

   ```
   rhn-channel --available-channels
   ```

   The name of the channel depends on the specific version of Red Hat Enterprise Linux you are using and is in the following format:

   ```
   rhel-x86_64-variant-version-rhscl-1
   ```

   In addition, certain packages, such as *devtoolset-3-gcc-plugin-devel*, depend on packages that are only available in the **Optional** channel. The name of this channel uses the following format:

   ```
   rhel-x86_64-variant-optional-6
   ```

   Replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

2. Subscribe the system to the channels from step no. 1 by running the following command as **root**:

   ```
   rhn-channel --add --channel=channel_name
   ```

   Replace *channel_name* with the name of the channel to enable.

3. To verify the list of channels you are subscribed to, at any time, run as **root**:

   ```
   rhn-channel --list
   ```

Once the system is subscribed, you can install Red Hat Developer Toolset as described in Section 1.5, “Installing Red Hat Developer Toolset”. For more information on how to register your system with RHN Classic, see the Red Hat Subscription Management collection of guides.

1.5. Installing Red Hat Developer Toolset

Red Hat Developer Toolset is distributed as a collection of RPM packages that can be installed, updated, uninstalled, and inspected by using the standard package management tools that are included in Red Hat Enterprise Linux. Note that a valid subscription that provides access to the Red Hat Software Collections content set is required in order to install Red Hat Developer Toolset on your system. For detailed instructions on how to associate your system with an appropriate subscription and get access to Red Hat Developer Toolset, see Section 1.4, “Getting Access to Red Hat Developer Toolset”.

Important

Before installing Red Hat Developer Toolset, install all available Red Hat Enterprise Linux updates.

1.5.1. Installing All Available Components

To install all components that are included in Red Hat Developer Toolset, install the *devtoolset-3* package by typing the following at a shell prompt as **root**:

```
yum install devtoolset-3
```

This installs the **Eclipse** development environment, all development, debugging, and performance monitoring tools, and other dependent packages to the system. Alternatively, you can choose to install only a selected package group as described in Section 1.5.2, “Installing Individual Package Groups”.

Note

Note that since Red Hat Developer Toolset 3.0, the *scl-utils* package is not a part of Red Hat Developer Toolset, which is a change from preceding versions where the **scl** utility was installed along with the Red Hat Developer Toolset software collection.

1.5.2. Installing Individual Package Groups

To make it easier to install only certain components, such as the integrated development environment or the software development toolchain, Red Hat Developer Toolset is distributed with a number of meta packages that allow you to install selected package groups as described in Table 1.2, “Red Hat Developer Toolset Meta Packages”.

Table 1.2. Red Hat Developer Toolset Meta Packages

Package Name	Description	Installed Components
devtoolset-3-ide	Integrated Development Environment	Eclipse
devtoolset-3-perftools	Performance monitoring tools	SystemTap, Valgrind, OProfile, Dyninst
devtoolset-3-toolchain	Development and debugging tools	GCC, GDB, binutils, elfutils, dwz, memstomp, strace, ltrace

To install any of these meta packages, type the following at a shell prompt as **root**:

```
yum install package_name
```

Replace *package_name* with a space-separated list of meta packages you want to install. For example, to install only the Eclipse development environment and packages that depend on it, type as **root**:

```
~]# yum install devtoolset-3-ide
```

Alternatively, you can choose to install all available components as described in Section 1.5.1, “Installing All Available Components”.

1.5.3. Installing Optional Packages

Red Hat Developer Toolset is distributed with a number of optional packages that are not installed by default. To list all Red Hat Developer Toolset packages that are available to you but not installed on your system, type the following command at a shell prompt:

```
yum list available devtoolset-3-\*
```

To install any of these optional packages, run as **root**:

```
yum install package_name
```

Replace *package_name* with a space-separated list of packages that you want to install. For example, to install the *devtoolset-3-gdb-gdbserver* and *devtoolset-3-gdb-doc* packages, type:

```
~]# yum install devtoolset-3-gdb-gdbserver devtoolset-3-gdb-doc
```

1.5.4. Installing Debugging Information

To install debugging information for any of the Red Hat Developer Toolset packages, make sure that the *yum-utils* package is installed and run the following command as **root**:

```
debuginfo-install package_name
```

For example, to install debugging information for the *devtoolset-3-dwz* package, type:

```
~]# debuginfo-install devtoolset-3-dwz
```

Note that in order to use this command, you need to have access to the repository with these packages. If your system is registered with Red Hat Subscription Management, enable the **rhel-*variant*-rhscl-*version*-debug-rpms** repository as described in Section 1.4.1, “Using Red Hat Subscription Management”. If your system is registered with RHN Classic, subscribe the system to the **rhel-x86_64-*variant*-*version*-debuginfo** channel as described in Section 1.4.2, “Using RHN Classic”. For more information on how to get access to debuginfo packages, see https://access.redhat.com/site/solutions/9907.

1.6. Updating Red Hat Developer Toolset

1.6.1. Updating to a Minor Version

When a new minor version of Red Hat Developer Toolset is available, run the following command as **root** to update your Red Hat Enterprise Linux installation:

```
yum update
```

This updates all packages on your Red Hat Enterprise Linux system, including the Red Hat Developer Toolset versions of the **Eclipse** development environment, development, debugging, and performance monitoring tools, and other dependent packages.

Important

Use of Red Hat Developer Toolset requires the removal of any earlier pre-release versions of it. Additionally, it is not possible to update to Red Hat Developer Toolset 3.1 from a pre-release version of Red Hat Developer Toolset, including beta releases. If you have previously installed any pre-release version of Red Hat Developer Toolset, uninstall it from your system as described in Section 1.7, “Uninstalling Red Hat Developer Toolset” and install the new version as documented in Section 1.5, “Installing Red Hat Developer Toolset”.

1.6.2. Updating to a Major Version

When a new major version of Red Hat Developer Toolset is available, you can install it in parallel with the previous version. For detailed instructions on how to install Red Hat Developer Toolset on your system, see Section 1.5, “Installing Red Hat Developer Toolset”.

1.7. Uninstalling Red Hat Developer Toolset

To uninstall Red Hat Developer Toolset packages from your system, type the following at a shell prompt as **root**:

```
yum remove devtoolset-3\* libasan libatomic libcilkrts libitm liblsan
libtsan libubsan
```

This removes the **GNU Compiler Collection**, **GNU Debugger**, **binutils**, and other packages that are a part of Red Hat Developer Toolset from the system.

Note

Red Hat Developer Toolset 3.1 for Red Hat Enterprise Linux 7 no longer includes the **libatomic** and **libitm** libraries, which the above command attempts to remove, because they are not required for a proper function of Red Hat Developer Toolset components on that system. Nevertheless, the above command works as expected even on Red Hat Enterprise Linux 7.

Note that the uninstallation of the tools provided by Red Hat Developer Toolset does not affect the Red Hat Enterprise Linux system versions of these tools.

1.8. Using Red Hat Developer Toolset Container Images

Starting with Red Hat Developer Toolset 3.1, *Dockerfiles* are available for selected Red Hat Developer Toolset components. Dockerfiles are text documents that contain instructions for automated building of docker-formatted container images. The resulting container images can be used to run Red Hat Developer Toolset components inside virtual software containers, thus isolating them from the host system and allowing for their rapid deployment. This section describes how to obtain Red Hat Developer Toolset Dockerfiles, how to use them to build docker-formatted container images, and how to run Red Hat Developer Toolset components using the resulting container images.

Red Hat Developer Toolset 3.1 is shipped with the following Dockerfiles:

- devtoolset-3-dyninst
- devtoolset-3-elfutils
- devtoolset-3-oprofile
- devtoolset-3-systemtap (only for Red Hat Enterprise Linux 7)
- devtoolset-3-toolchain
- devtoolset-3-valgrind
- devtoolset-3 (only for Red Hat Enterprise Linux 7)

Note

The *docker* package, which contains the **Docker** daemon, command line tool, and other necessary components for building and using docker-formatted container images, is currently only available for the Server variant of the Red Hat Enterprise Linux 7 product. Red Hat Developer Toolset Dockerfiles are distributed for Red Hat Enterprise Linux 6 as well, but the images built using them can only be deployed on Red Hat Enterprise Linux 7 Server.

1.8.1. Obtaining Dockerfiles

The Red Hat Developer Toolset Dockerfiles are provided by the *devtoolset-3-dockerfiles* package. The package contains individual Dockerfiles for building docker-formatted container images with individual components and a meta-package for building a docker-formatted container image with all the components offered. To be able to use the Dockerfiles, install this package by executing:

```
~]# yum install devtoolset-3-dockerfiles
```

Note that because some Red Hat Developer Toolset components depend on packages from the Optional channel, the provided Dockerfiles contain instructions that enable the channel automatically.

1.8.2. Building Container Images

Follow the instruction outlined at Getting Docker in RHEL 7 to set up an environment for building and using docker-formatted container images.

When you are ready to build your image, change to the directory where the Dockerfile is installed and run the **`docker build`** command as shown in the following example.

Example 1.1. Building a Container Image with a Red Hat Developer Toolset Component

To build a docker-formatted container image for deploying the **elfutils** tools in a container, follow the instructions below:

1. Make sure you have a **Docker** environment set up properly on your system by following instructions at Getting Docker in RHEL 7.
2. Install the package containing the Red Hat Developer Toolset Dockerfiles:

```
~]# yum install devtoolset-3-dockerfiles
```

3. Determine where the Dockerfile for the required component is located:

```
~]# rpm -qpl devtoolset-3-dockerfiles | grep
"elfutils/Dockerfile"
```

4. Change to the directory where the required Dockerfile is installed:

```
~]# cd /opt/rh/devtoolset-3/root/usr/share/devtoolset-3-
dockerfiles/rhel7/devtoolset-3-elfutils/
```

5. Build the container image using the **docker build** command:

```
~]# docker build -t devtoolset-3-elfutils-7 .
```

   Replace *devtoolset-3-elfutils-7* with the name you wish to assign to your resulting container image.

1.8.3. Running Red Hat Developer Toolset Tools from Container Images

To launch the docker-formatted container image you built (see Section 1.8.2, “Building Container Images”), execute the **docker run** command as follows:

```
docker run -t -i container-image /bin/bash -l
```

Substitute the *container-image* parameter with the name of the container image you chose when building it.

For example, to launch the container image built in Example 1.1, “Building a Container Image with a Red Hat Developer Toolset Component”, run the following command:

```
~]# docker run -t -i devtoolset-3-elfutils-7 /bin/bash -l
```

1.9. Additional Resources

For more information about Red Hat Developer Toolset and Red Hat Enterprise Linux, see the resources listed below.

Online Documentation

- Red Hat Subscription Management collection of guides — The Red Hat Subscription Management collection of guides provides detailed information on how to manage subscriptions on Red Hat Enterprise Linux.
- Red Hat Developer Toolset 3.1 Release Notes — The *Release Notes* for Red Hat Developer Toolset 3.1 contain more information.
- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information on the **Eclipse** IDE, libraries and runtime support, compiling and building, debugging, and profiling on these systems.

- Red Hat Enterprise Linux 6 Installation Guide and Red Hat Enterprise Linux 7 Installation Guide — The *Installation Guides* for Red Hat Enterprise Linux 6 an 7 explain how to obtain, install, and update the system.
- Red Hat Enterprise Linux 6 Deployment Guide — The *Deployment Guide* for Red Hat Enterprise Linux 6 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 6.
- Red Hat Enterprise Linux 7 System Administrator's Guide — The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.
- Get Started with Docker Formatted Container Images on Red Hat Systems — The Knowledgebase article contains a comprehensive overview of information about building and using docker-formatted container images on Red Hat Developer Toolset 7 and Red Hat Enterprise Linux Atomic.

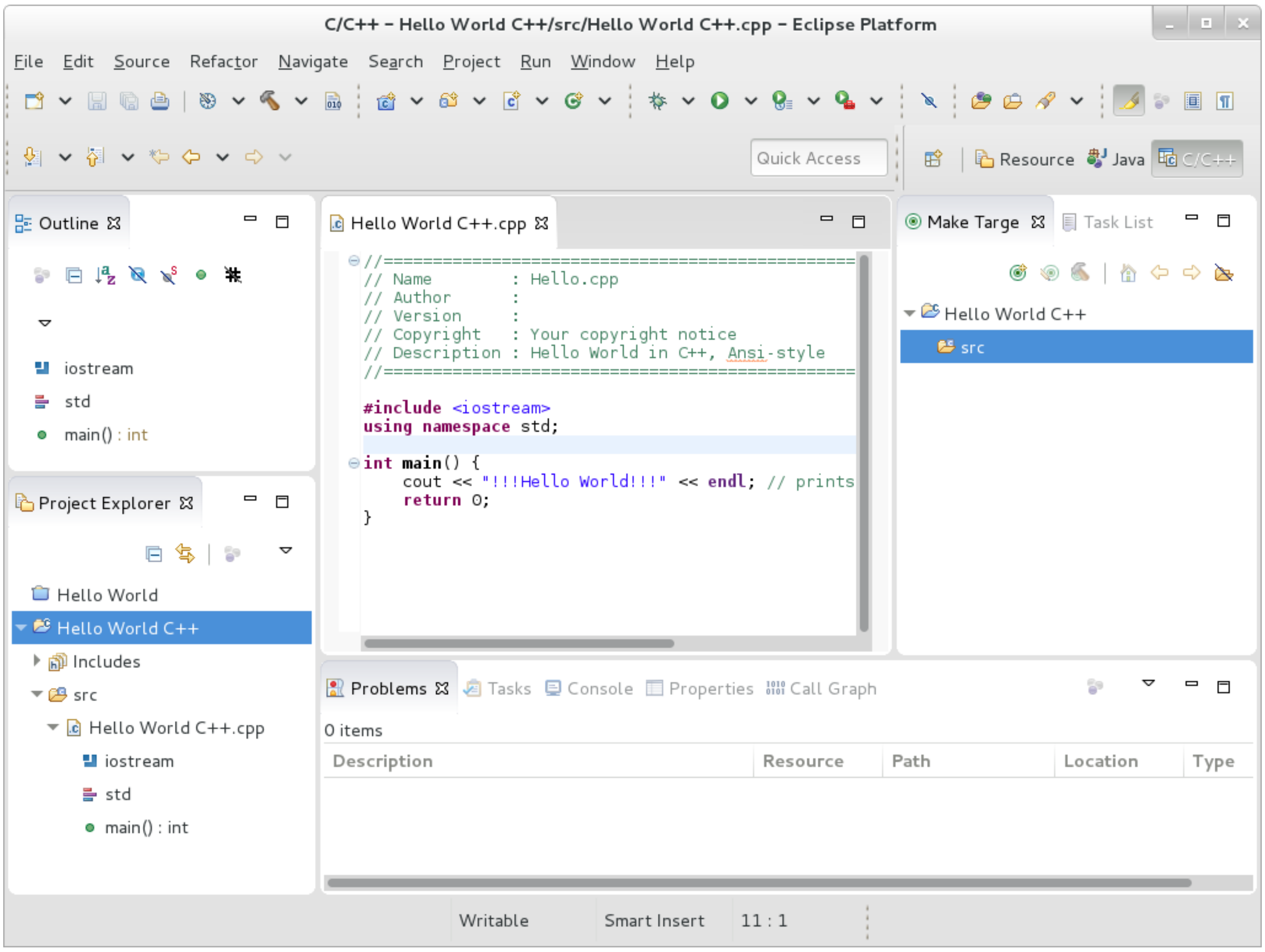
See Also

- Appendix A, *Changes in Version 3.1* provides a list of changes and improvements over the version of the GNU Compiler Collection and GNU Debugger in the previous version of Red Hat Developer Toolset.
- Appendix B, *Changes in Version 3.0* provides a comprehensive list of changes and improvements over the Red Hat Enterprise Linux system versions of the GNU Compiler Collection, GNU Debugger, and binutils, as well as information about the language, ABI, and debugging compatibility.

Part II. Integrated Development Environments

Chapter 2. Eclipse

Eclipse is a powerful development environment that provides tools for each phase of the development process. It integrates a variety of disparate tools into a unified environment to create a rich development experience, provides a fully configurable user interface, and features a pluggable architecture that allows for an extension in a variety of ways. For instance, the **Valgrind** plug-in allows programmers to perform memory profiling, otherwise performed on the command line, through the **Eclipse** user interface.



```
//===============================================
// Name        : Hello.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//===============================================

#include <iostream>
using namespace std;

int main() {
	cout << "!!!Hello World!!!" << endl; // prints
	return 0;
}
```

Figure 2.1. Sample Eclipse Session

Eclipse provides a graphical development environment alternative to traditional interaction with command line tools and as such, it is a welcome alternative to developers who do not want to use the command line interface. The traditional, mostly command line-based Linux tools suite (such as **gcc** or **gdb**) and **Eclipse** offer two distinct approaches to programming.

Red Hat Developer Toolset is distributed with **Eclipse 4.4.2**, which is based on the Eclipse Foundation's Luna release train SR2 (Service Release 2). Note that 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.

Table 2.1. Eclipse Components Included in Red Hat Developer Toolset

Package	Description
devtoolset-3-eclipse-cdt	The C/C++ Development Tooling (CDT), which provides features and plug-ins for development in C and C++.
devtoolset-3-eclipse-emf	The Eclipse Modeling Framework (EMF), which allows you to build applications based on a structured data model.
devtoolset-3-eclipse-gef	The Graphical Editing Framework (GEF), which allows you to create a rich graphical editor from an existing application model.
devtoolset-3-eclipse-rse	The Remote System Explorer (RSE) framework, which allows you to work with remote systems from Eclipse.
devtoolset-3-eclipse-jgit	JGit, a Java implementation of the **Git** revision control system.
devtoolset-3-eclipse-egit	EGit, a team provider for **Eclipse** that provides features and plug-ins for interaction with Git repositories.
devtoolset-3-eclipse-mylyn	Mylyn, a task management system for **Eclipse**.
devtoolset-3-eclipse-pde	The Plugin Development Environment for developing **Eclipse** plugins.
devtoolset-3-eclipse-remote	The Remote Services plug-in, which provides an extensible remote-services framework.
devtoolset-3-eclipse-tests	**Eclipse** tests.
devtoolset-3-eclipse-linuxtools	A meta package for Linux-specific **Eclipse** plug-ins.
devtoolset-3-eclipse-changelog [a]	The ChangeLog plug-in, which allows you to create and maintain changelog files.
devtoolset-3-eclipse-gcov[a]	The GCov plug-in, which integrates the **GCov** test coverage program with **Eclipse**.
devtoolset-3-eclipse-gprof[a]	The Gprof plug-in, which integrates the **Gprof** performance analysis utility with **Eclipse**.
devtoolset-3-eclipse-manpage[a]	The Man Page plug-in, which allows you to view manual pages in **Eclipse**.
devtoolset-3-eclipse-oprofile[a]	The OProfile plug-in, which integrates **OProfile** with **Eclipse**.
devtoolset-3-eclipse-perf[a]	The Perf plug-in, which integrates the **perf** tool with **Eclipse**.
devtoolset-3-eclipse-rpm-editor[a]	The Eclipse Spec File Editor, which allows you to maintain RPM spec files.
devtoolset-3-eclipse-systemtap[a]	The SystemTap plug-in, which integrates **SystemTap** with **Eclipse**.
devtoolset-3-eclipse-valgrind[a]	The Valgrind plug-in, which integrates **Valgrind** with **Eclipse**.

[a] This package is installed as a dependency of *devtoolset-3-eclipse-linuxtools*.

2.1. Installing Eclipse

In Red Hat Developer Toolset, the **Eclipse** development environment is provided as a collection of RPM packages and is automatically installed with the *devtoolset-3-ide* package as described in Section 1.5, “Installing Red Hat Developer Toolset”. For a list of available components, see Table 2.1, “Eclipse Components Included in Red Hat Developer Toolset”.

Note

The Red Hat Developer Toolset version of Eclipse fully supports C, C++, and Java development, but does *not* provide support for the Fortran programming language.

2.2. Using Eclipse

To start the Red Hat Developer Toolset version of **Eclipse**, either select **Applications** → **Programming** → **DTS Eclipse** from the panel, or type the following at a shell prompt:

```
scl enable devtoolset-3 'eclipse'
```

During its startup, **Eclipse** prompts you to select a *workspace*, that is, a directory in which you want to store your projects. You can either use **`~/workspace/`**, which is the default option, or click the **`Browse`** button to browse your file system and select a custom directory. Additionally, you can select the **`Use this as the default and do not ask again`** check box to prevent **Eclipse** from displaying this dialog box the next time you run this development environment. When you are done, click the **`OK`** button to confirm the selection and proceed with the startup.

2.2.1. Using the Red Hat Developer Toolset Toolchain

To use the Red Hat Developer Toolset version of **Eclipse** with support for the **GNU Compiler Collection** and **binutils** from Red Hat Developer Toolset, make sure that the *devtoolset-3-toolchain* package is installed and run the application as described in Section 2.2, "Using Eclipse". Red Hat Developer Toolset **Eclipse** uses the Red Hat Developer Toolset toolchain by default.

For detailed instructions on how to install the *devtoolset-3-toolchain* package in your system, see Section 1.5, "Installing Red Hat Developer Toolset".

Important

If you are working on a project that you previously built with the Red Hat Enterprise Linux version of the **GNU Compiler Collection**, make sure that you discard all previous build results. To do so, open the project in **Eclipse** and select **Project** → **Clean** from the menu.

2.2.2. Using the Red Hat Enterprise Linux Toolchain

To use the Red Hat Developer Toolset version of **Eclipse** with support for the toolchain distributed with Red Hat Enterprise Linux change the configuration of the project to use absolute paths to the Red Hat Enterprise Linux system versions of **`gcc`**, **`g++`**, and **`as`**.

To configure **Eclipse** to explicitly use the Red Hat Enterprise Linux system versions of the tools for the current project, complete the following steps:

1. In the C/C++ perspective, choose **Project** → **Properties** from the main menu bar to open the project properties.
2. In the menu on the left-hand side of the dialog box, click **C/C++ Build** → **Settings**.
3. Select the **`Tool Settings`** tab.
4. If you are working on a C project:

    a. select **GCC C Compiler** or **Cross GCC Compiler** and change the value of the **`Command`** field to:

    ```
    /usr/bin/gcc
    ```

b. select **GCC C Linker** or **Cross GCC Linker** and change the value of the **Command** field to:

```
/usr/bin/gcc
```

c. select **GCC Assembler** or **Cross GCC Assembler** and change the value of the **Command** field to:

```
/usr/bin/as
```

If you are working on a C++ project:

a. select **GCC C++ Compiler** or **Cross G++ Compiler** and change the value of the **Command** field to:

```
/usr/bin/g++
```

b. select **GCC C Compiler** or **Cross GCC Compiler** and change the value of the **Command** field to:

```
/usr/bin/gcc
```

c. select **GCC C++ Linker** or **Cross G++ Linker** and change the value of the **Command** field to:

```
/usr/bin/g++
```

d. select **GCC Assembler** or **Cross GCC Assembler** and change the value of the **Command** field to:

```
/usr/bin/as
```

5. Click the **OK** button to save the configuration changes.

2.3. Additional Resources

A detailed description of **Eclipse** and all its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- **Eclipse** includes a built-in **Help** system, which 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 use of this Help section is detailed in the *Red Hat Enterprise Linux Developer Guide* linked below.

Online Documentation

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information on **Eclipse**, including a description of the user interface, overview of available development toolkits, or instructions on how to use it to build RPM packages.

See Also

- Section A.1, “Changes in Eclipse” provides a comprehensive list of features and improvements over the **Eclipse** development environment included in Red Hat Enterprise Linux 6 and the previous release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 3, *GNU Compiler Collection (GCC)* provides information on how to compile programs written in C, C++, and Fortran on the command line.

Part III. Development Tools

Chapter 3. GNU Compiler Collection (GCC)

The **GNU Compiler Collection**, commonly abbreviated **GCC**, is a portable compiler suite with support for a wide selection of programming languages.

Red Hat Developer Toolset is distributed with **GCC 4.9.2**. This version is more recent than the version included in Red Hat Enterprise Linux and provides a number of bug fixes and enhancements.

3.1. GNU C Compiler

3.1.1. Installing the C Compiler

In Red Hat Developer Toolset, the GNU C compiler is provided by the *devtoolset-3-gcc* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

3.1.2. Using the C Compiler

To compile a C program on the command line, run the **gcc** compiler as follows:

```
scl enable devtoolset-3 'gcc -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line, run the following command:

```
scl enable devtoolset-3 'gcc -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file, run:

```
scl enable devtoolset-3 'gcc -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gcc** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **gcc** you are using at any point, type the following at a shell prompt:

```
which gcc
```

Red Hat Developer Toolset's **gcc** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gcc**:

```
gcc -v
```

Important

Some newer library features are statically linked into applications built with Red Hat Developer Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This adds a small additional security risk as normal Red Hat Enterprise Linux errata would not change this code. If the need for developers to rebuild their applications due to such an issue arises, Red Hat will signal this via a security erratum. Developers are strongly advised not to statically link their entire application for the same reasons.

Note

The Red Hat Developer Toolset 3.1 version of **GCC** adds support for Cilk+, an extension to the C and C++ languages for parallel programming, which can be enabled using the **-fcilkplus** option. A new runtime library, **libcilkrts**, is included in this release to support Cilk+. The **libcilkrts** library will be a part of the *gcc-libraries* package in the future Red Hat Enterprise Linux releases, but the package is not included in all supported Red Hat Enterprise Linux releases. To enable dynamic linkage of binaries and libraries built with Red Hat Developer Toolset 3.1 **GCC** using Cilk+ features on supported Red Hat Enterprise Linux releases that do not contain **libcilkrts**, install the **libcilkrts.so** shared library from Red Hat Developer Toolset 3.1 with such binaries or libraries.

Example 3.1. Compiling a C Program on the Command Line

Consider a source file named **hello.c** with the following contents:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

To compile this source code on the command line by using the **gcc** compiler from Red Hat Developer Toolset, type:

```
~]$ scl enable devtoolset-3 'gcc -o hello hello.c'
```

This creates a new binary file called **hello** in the current working directory.

3.1.3. Running a C Program

When **gcc** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and type:

```
./file_name
```

Example 3.2. Running a C Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 3.1, "Compiling a C Program on the Command Line", you can run it by typing the following at a shell prompt:

```
~]$ ./hello
Hello, World!
```

3.2. GNU C++ Compiler

3.2.1. Installing the C++ Compiler

In Red Hat Developer Toolset, the GNU C++ compiler is provided by the *devtoolset-3-gcc-c++* package and is automatically installed with the *devtoolset-3-toolchain* package as described in Section 1.5, "Installing Red Hat Developer Toolset".

3.2.2. Using the C++ Compiler

To compile a C++ program on the command line, run the **g++** compiler as follows:

```
scl enable devtoolset-3 'g++ -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the **g++** compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line, run the following command:

```
scl enable devtoolset-3 'g++ -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the **g++** compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file, run:

```
scl enable devtoolset-3 'g++ -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **g++** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **g++** you are using at any point, type the following at a shell prompt:

```
which g++
```

Red Hat Developer Toolset's **g++** executable path will begin with */opt*. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **g++**:

```
g++ -v
```

Important

Some newer library features are statically linked into applications built with Red Hat Developer Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This adds a small additional security risk as normal Red Hat Enterprise Linux errata would not change this code. If the need for developers to rebuild their applications due to such an issue arises, Red Hat will signal this via a security erratum. Developers are strongly advised not to statically link their entire application for the same reasons.

Example 3.3. Compiling a C++ Program on the Command Line

Consider a source file named **hello.cpp** with the following contents:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
  cout << "Hello, World!" << endl;
  return 0;
}
```

To compile this source code on the command line by using the **g++** compiler from Red Hat Developer Toolset, type:

```
~]$ scl enable devtoolset-3 'g++ -o hello hello.cpp'
```

This creates a new binary file called **hello** in the current working directory.

3.2.3. Running a C++ Program

When **g++** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and type:

```
./file_name
```

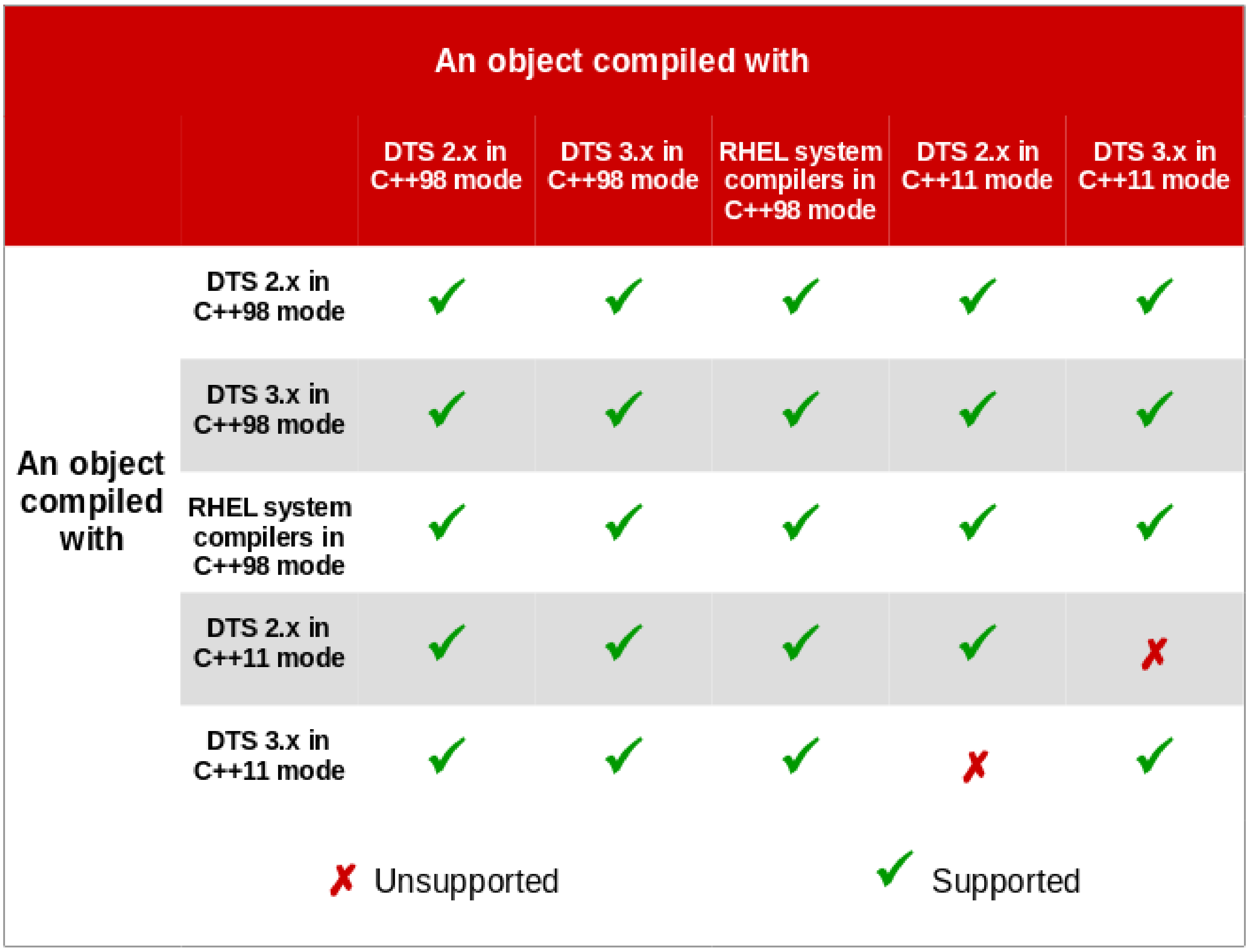
Example 3.4. Running a C++ Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 3.3, “Compiling a C++ Program on the Command Line”, you can run it by typing the following at a shell prompt:

```
~]$ ./hello
Hello, World!
```

3.2.4. C++ Compatibility

Objects compiled with Red Hat Developer Toolset 2 and Red Hat Developer Toolset 3 in C++98 mode (the default mode) are compatible with each other, and with objects compiled with the Red Hat Enterprise Linux system compilers in C++98 mode. Objects compiled in C++11 mode (that is, with **-std=c++11**) are compatible with all of the above C++98 objects, but are not guaranteed to be compatible with C++11 objects compiled with a different major version of the compiler. That is, if some C++11 objects previously compiled with Red Hat Developer Toolset 2 are rebuilt with Red Hat Developer Toolset 3, all C++11 objects should be rebuilt. For the same reasons, C++11 objects built with this release of Red Hat Developer Toolset may need to be rebuilt to be compatible with objects built by future releases of Red Hat Developer Toolset or Red Hat Enterprise Linux.



		An object compiled with				
		DTS 2.x in C++98 mode	DTS 3.x in C++98 mode	RHEL system compilers in C++98 mode	DTS 2.x in C++11 mode	DTS 3.x in C++11 mode
An object compiled with	DTS 2.x in C++98 mode	✓	✓	✓	✓	✓
	DTS 3.x in C++98 mode	✓	✓	✓	✓	✓
	RHEL system compilers in C++98 mode	✓	✓	✓	✓	✓
	DTS 2.x in C++11 mode	✓	✓	✓	✓	✗
	DTS 3.x in C++11 mode	✓	✓	✓	✗	✓

✗ Unsupported ✓ Supported

Figure 3.1. C++ Compatibility Matrix

3.3. GNU Fortran Compiler

3.3.1. Installing the Fortran Compiler

In Red Hat Developer Toolset, the GNU Fortran compiler is provided by the *devtoolset-3-gcc-gfortran* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, "Installing Red Hat Developer Toolset".

3.3.2. Using the Fortran Compiler

To compile a Fortran program on the command line, run the **gfortran** compiler as follows:

```
scl enable devtoolset-3 'gfortran -o output_file source_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line, run the following command:

```
scl enable devtoolset-3 'gfortran -o object_file -c source_file'
```

This creates an object file named *object_file*. If the **-o** option is omitted, the compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file, run:

```
scl enable devtoolset-3 'gfortran -o output_file object_file...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gfortran** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **gfortran** you are using at any point, type the following at a shell prompt:

```
which gfortran
```

Red Hat Developer Toolset's **gfortran** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gfortran**:

```
gfortran -v
```

Important

Some newer library features are statically linked into applications built with Red Hat Developer Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This adds a small additional security risk as normal Red Hat Enterprise Linux errata would not change this code. If the need for developers to rebuild their applications due to such an issue arises, Red Hat will signal this via a security erratum. Developers are strongly advised not to statically link their entire application for the same reasons.

Example 3.5. Compiling a Fortran Program on the Command Line

Consider a source file named **hello.f** with the following contents:

```
program hello
  print *, "Hello, World!"
end program hello
```

To compile this source code on the command line by using the **gfortran** compiler from Red Hat Developer Toolset, type:

```
~]$ scl enable devtoolset-3 'gfortran -o hello hello.f'
```

This creates a new binary file called **hello** in the current working directory.

3.3.3. Running a Fortran Program

When **gfortran** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and type:

```
./file_name
```

Example 3.6. Running a Fortran Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 3.5, “Compiling a Fortran Program on the Command Line”, you can run it by typing the following at a shell prompt:

```
~]$ ./hello
 Hello, World!
```

3.4. Additional Resources

A detailed description of the GNU Compiler Collections and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- **gcc**(1) — The manual page for the **gcc** compiler provides detailed information on its usage; with few exceptions, **g++** accepts the same command line options as **gcc**. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man gcc'
  ```

- **gfortran**(1) — The manual page for the **gfortran** compiler provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man gfortran'
  ```

- *C++ Standard Library Documentation* — Documentation on the C++ standard library can be optionally installed by typing the following at a shell prompt as **root**:

  ```
  yum install devtoolset-3-libstdc++-docs
  ```

  Once installed, HTML documentation is available at **/opt/rh/devtoolset-3/root/usr/share/doc/devtoolset-3-libstdc++-docs-4.9.1/html/index.html**.

Online Documentation

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide in-depth information about **GCC**.
- Using the GNU Compiler Collection — The official GCC manual provides an in-depth description of the GNU compilers and their usage.
- The GNU C++ Library — The GNU C++ library documentation provides detailed information about the GNU implementation of the standard C++ library.
- The GNU Fortran Compiler — The GNU Fortran compiler documentation provides detailed information on **gfortran**'s usage.

See Also

- Section A.2, “Changes in GCC” provides a list of improvements over the Red Hat Enterprise Linux 7.1 version of the **GNU Compiler Collection** and the version distributed in the previous release of Red Hat Developer Toolset.
- Section B.2, “Changes in GCC” provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux 7.0 and 6.6 versions of the **GNU Compiler Collection** and the version distributed in the 2.1 release of Red Hat Developer Toolset, as well as information about the language, ABI, and debugging compatibility.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *Eclipse* provides a general introduction to the **Eclipse** development environment, and describes how to use it with the tools from Red Hat Developer Toolset.
- Chapter 4, *binutils* explains how to use the **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 5, *elfutils* explains how to use **elfutils**, a collection of binary tools to inspect and manipulate ELF files.
- Chapter 6, *dwz* explains how to use **dwz** to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.
- Chapter 7, *GNU Debugger (GDB)* provides information on how to debug programs written in C, C++, and Fortran.

Chapter 4. binutils

binutils is a collection of various binary tools, such as the **GNU linker**, **GNU assembler**, and other utilities that allow you to inspect and manipulate object files and binaries. See Table 4.1, “Tools Included in binutils for Red Hat Developer Toolset” for a complete list of binary tools that are distributed with the Red Hat Developer Toolset version of **binutils**.

Red Hat Developer Toolset is distributed with **binutils 2.24**. This version is more recent than the version included in Red Hat Enterprise Linux and provides bug fixes and enhancements, including support for the AVX-512 (512-bit Advanced Vector Extensions).

Table 4.1. Tools Included in binutils for Red Hat Developer Toolset

Name	Description
addr2line	Translates addresses into file names and line numbers.
ar	Creates, modifies, and extracts files from archives.
as	The GNU assembler.
c++filt	Decodes mangled C++ symbols.
dwp	Combines DWARF object files into a single DWARF package file.
elfedit	Examines and edits ELF files.
gprof	Display profiling information.
ld	The GNU linker.
ld.bfd	An alternative to the GNU linker.
ld.gold	Another alternative to the GNU linker.
nm	Lists symbols from object files.
objcopy	Copies and translates object files.
objdump	Displays information from object files.
ranlib	Generates an index to the contents of an archive to make access to this archive faster.
readelf	Displays information about ELF files.
size	Lists section sizes of object or archive files.
strings	Displays printable character sequences in files.
strip	Discards all symbols from object files.

4.1. Installing binutils

In Red Hat Developer Toolset, **binutils** are provided by the *devtoolset-3-binutils* package and are automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

4.2. Using the GNU Assembler

To produce an object file from an assembly language program, run the **as** tool as follows:

```
scl enable devtoolset-3 'as [option...] -o object_file source_file'
```

This creates an object file named *object_file* in the current working directory.

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **as** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **as** you are using at any point, type the following at a shell prompt:

```
which as
```

Red Hat Developer Toolset's **as** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **as**:

```
as -v
```

4.3. Using the GNU Linker

To create an executable binary file or a library from object files, run the **ld** tool as follows:

```
scl enable devtoolset-3 'ld [option...] -o output_file object_file...'
```

This creates a binary file named *output_file* in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **ld** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **ld** you are using at any point, type the following at a shell prompt:

```
which ld
```

Red Hat Developer Toolset's **ld** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **ld**:

```
ld -v
```

4.4. Using Other Binary Tools

The **binutils** provide many binary tools other than a linker and an assembler. For a complete list of these tools, see Table 4.1, “Tools Included in binutils for Red Hat Developer Toolset”.

To execute any of the tools that are a part of binutils, run the command as follows:

```
scl enable devtoolset-3 'tool [option...] file_name'
```

See Table 4.1, “Tools Included in binutils for Red Hat Developer Toolset” for a list of tools that are distributed with **binutils**. For example, to use the **objdump** tool to inspect an object file, type:

```
scl enable devtoolset-3 'objdump [option...] object_file'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset binary tools as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **binutils** you are using at any point, type the following at a shell prompt:

```
which objdump
```

Red Hat Developer Toolset's **objdump** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **objdump**:

```
objdump -v
```

4.5. Additional Resources

A detailed description of **binutils** is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- as(1), ld(1), addr2line(1), ar(1), c++filt(1), dwp(1), elfedit(1), gprof(1), nm(1), objcopy(1), objdump(1), ranlib(1), readelf(1), size(1), strings(1), strip(1), — Manual pages for various **binutils** tools provide more information about their respective usage. To display a manual page for the version included in Red Hat Developer Toolset, type:

```
scl enable devtoolset-3 'man tool'
```

Online Documentation

- Documentation for binutils — The **binutils** documentation provides an in-depth description of the binary tools and their usage.

See Also

- Section B.3, "Changes in binutils" provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux 6.6 and 7.0 versions of **binutils** and the version distributed in the 2.1 release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 5, *elfutils* explains how to use **elfutils**, a collection of binary tools to inspect and manipulate ELF files.
- Chapter 3, *GNU Compiler Collection (GCC)* provides information on how to compile programs written in C, C++, and Fortran.

Chapter 5. elfutils

elfutils is a collection of various binary tools, such as **eu-objdump**, **eu-readelf**, and other utilities that allow you to inspect and manipulate ELF files. See Table 5.1, “Tools Included in elfutils for Red Hat Developer Toolset” for a complete list of binary tools that are distributed with the Red Hat Developer Toolset version of **elfutils**.

Red Hat Developer Toolset is distributed with **elfutils 0.161**. This version is more recent than the version included in Red Hat Enterprise Linux and the previous release of Red Hat Developer Toolset, and provides numerous bug fixes and enhancements.

Table 5.1. Tools Included in elfutils for Red Hat Developer Toolset

Name	Description
eu-addr2line	Translates addresses into file names and line numbers.
eu-ar	Creates, modifies, and extracts files from archives.
eu-elfcmp	Compares relevant parts of two ELF files for equality.
eu-elflint	Verifies that ELF files are compliant with the *generic ABI* (gABI) and *processor-specific supplement ABI* (psABI) specification.
eu-findtextrel	Locates the source of text relocations in files.
eu-make-debug-archive	Creates an offline archive for debugging.
eu-nm	Lists symbols from object files.
eu-objdump	Displays information from object files.
eu-ranlib	Generates an index to the contents of an archive to make access to this archive faster.
eu-readelf	Displays information about ELF files.
eu-size	Lists section sizes of object or archive files.
eu-stack	A new utility for unwinding processes and cores.
eu-strings	Displays printable character sequences in files.
eu-strip	Discards all symbols from object files.
eu-unstrip	Combines stripped files with separate symbols and debug information.

5.1. Installing elfutils

In Red Hat Developer Toolset, **elfutils** is provided by the *devtoolset-3-elfutils* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

5.2. Using elfutils

To execute any of the tools that are part of **elfutils**, run the command as follows:

```
scl enable devtoolset-3 'tool [option...] file_name'
```

See Table 5.1, “Tools Included in elfutils for Red Hat Developer Toolset” for a list of tools that are distributed with **elfutils**. For example, to use the **eu-objdump** tool to inspect an object file, type:

```
scl enable devtoolset-3 'eu-objdump [option...] object_file'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset binary tools as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **elfutils** you are using at any point, type the following at a shell prompt:

```
which eu-objdump
```

Red Hat Developer Toolset's **eu-objdump** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **eu-objdump**:

```
eu-objdump -V
```

5.3. Additional Resources

A detailed description of **elfutils** is beyond the scope of this book. For more information, see the resources listed below.

See Also

- Section A.3, "Changes in elfutils" provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux system version of **elfutils** and the version distributed in the previous release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 3, *GNU Compiler Collection (GCC)* provides information on how to compile programs written in C, C++, and Fortran.
- Chapter 4, *binutils* explains how to use the **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 6, *dwz* explains how to use **dwz** to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size.

Chapter 6. dwz

dwz is a command line tool that attempts to optimize DWARF debugging information contained in ELF shared libraries and ELF executables for size. To do so, **dwz** replaces DWARF information representation with equivalent smaller representation where possible and reduces the amount of duplication by using techniques from *Appendix E* of the *DWARF Standard*.

Red Hat Developer Toolset is distributed with **dwz 0.11**.

6.1. Installing dwz

In Red Hat Developer Toolset, the **dwz** utility is provided by the *devtoolset-3-dwz* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, "Installing Red Hat Developer Toolset".

6.2. Using dwz

To optimize DWARF debugging information in a binary file, run the **dwz** tool as follows:

```
scl enable devtoolset-3 'dwz [option...] file_name'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **dwz** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **dwz** you are using at any point, type the following at a shell prompt:

```
which dwz
```

Red Hat Developer Toolset's **dwz** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **dwz**:

```
dwz -v
```

6.3. Additional Resources

A detailed description of **dwz** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- dwz(1) — The manual page for the **dwz** utility provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man dwz'
  ```

See Also

- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 3, *GNU Compiler Collection (GCC)* provides information on how to compile programs written in C, C++, and Fortran.
- Chapter 4, *binutils* explains how to use the **binutils**, a collection of binary tools to inspect and manipulate object files and binaries.
- Chapter 5, *elfutils* explains how to use **elfutils**, a collection of binary tools to inspect and manipulate ELF files.

Part IV. Debugging Tools

Chapter 7. GNU Debugger (GDB)

The **GNU Debugger**, commonly abbreviated as **GDB**, is a command line tool that can be used to debug programs written in various programming languages. It allows you to inspect memory within the code being debugged, control the execution state of the code, detect the execution of particular sections of code, and much more.

Red Hat Developer Toolset is distributed with **GDB 7.8.2**. This version is more recent than the version included in Red Hat Enterprise Linux and the previous release of Red Hat Developer Toolset, and provides some enhancements and numerous bug fixes.

7.1. Installing the GNU Debugger

In Red Hat Developer Toolset, the **GNU Debugger** is provided by the *devtoolset-3-gdb* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

7.2. Preparing a Program for Debugging

Compiling Programs with Debugging Information

To compile a C program with debugging information that can be read by the **GNU Debugger**, make sure the **gcc** compiler is run with the **-g** option. To do so on the command line, use a command in the following form:

```
scl enable devtoolset-3 'gcc -g -o output_file input_file...'
```

Similarly, to compile a C++ program with debugging information, run:

```
scl enable devtoolset-3 'g++ -g -o output_file input_file...'
```

Example 7.1. Compiling a C Program With Debugging Information

Consider a source file named **fibonacci.c** that has the following contents:

```
#include <stdio.h>
#include <limits.h>

int main (int argc, char *argv[]) {
  unsigned long int a = 0;
  unsigned long int b = 1;
  unsigned long int sum;

  while (b < LONG_MAX) {
    printf("%ld ", b);
    sum = a + b;
    a = b;
    b = sum;
  }

  return 0;
}
```

To compile this program on the command line using **GCC** from Red Hat Developer Toolset with debugging information for the **GNU Debugger**, type:

```
~]$ scl enable devtoolset-3 'gcc -g -o fibonacci fibonacci.c'
```

This creates a new binary file called **fibonacci** in the current working directory.

Installing Debugging Information for Existing Packages

To install debugging information for a package that is already installed on the system, type the following at a shell prompt as **root**:

```
debuginfo-install package_name
```

Note that the *yum-utils* package must be installed for the **debuginfo-install** utility to be available on your system.

Example 7.2. Installing Debugging Information for the glibc Package

To install debugging information for the *glibc* package, type:

```
~]# debuginfo-install glibc
Loaded plugins: product-id, refresh-packagekit, subscription-manager
--> Running transaction check
---> Package glibc-debuginfo.x86_64 0:2.12-1.47.el6_2.5 will be
installed
...
```

7.3. Running the GNU Debugger

To run the **GNU Debugger** on a program you want to debug, type the following at a shell prompt:

```
scl enable devtoolset-3 'gdb file_name'
```

This starts the **gdb** debugger in interactive mode and displays the default prompt, **(gdb)**. To quit the debugging session and return to the shell prompt, run the following command at any time:

```
quit
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **gdb** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **gdb** you are using at any point, type the following at a shell prompt:

```
which gdb
```

Red Hat Developer Toolset's **gdb** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **gdb**:

```
gdb -v
```

Example 7.3. Running the gdb Utility on the fibonacci Binary File

Assuming that you have successfully compiled the **fibonacci** binary file as shown in Example 7.1, “Compiling a C Program With Debugging Information”, you can start debugging it with **gdb** by typing the following at a shell prompt:

```
~]$ scl enable devtoolset-3 'gdb fibonacci'
GNU gdb (GDB) Red Hat Enterprise Linux (7.4.50.20120120-43.el6)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

7.4. Listing Source Code

To view the source code of the program you are debugging, run the following command:

```
list
```

Before you start the execution of the program you are debugging, **gdb** displays the first ten lines of the source code, and any subsequent use of this command lists another ten lines. Once you start the execution, **gdb** displays the lines that are surrounding the line on which the execution stops, typically when you set a breakpoint.

You can also display the code that is surrounding a particular line. To do so, run the command in the following form:

```
list [file_name:]line_number
```

Similarly, to display the code that is surrounding the beginning of a particular function, run:

```
list [file_name:]function_name
```

Note that you can change the number of lines the **list** command displays by running the following command:

```
set listsize number
```

Example 7.4. Listing the Source Code of the fibonacci Binary File

The **fibonacci.c** file listed in Example 7.1, "Compiling a C Program With Debugging Information" has exactly 17 lines. Assuming that you have compiled it with debugging information and you want the **gdb** utility to be capable of listing the entire source code, you can run the following command to change the number of listed lines to 20:

```
(gdb) set listsize 20
```

You can now display the entire source code of the file you are debugging by running the **list** command with no additional arguments:

```
(gdb) list
1       #include <stdio.h>
2       #include <limits.h>
3
4       int main (int argc, char *argv[]) {
5         unsigned long int a = 0;
6         unsigned long int b = 1;
7         unsigned long int sum;
8
9         while (b < LONG_MAX) {
10          printf("%ld ", b);
11          sum = a + b;
12          a = b;
13          b = sum;
14        }
15
16        return 0;
17      }
```

7.5. Setting Breakpoints

Setting a New Breakpoint

To set a new breakpoint at a certain line, run the following command:

```
break [file_name:]line_number
```

You can also set a breakpoint on a certain function:

```
break [file_name:]function_name
```

Example 7.5. Setting a New Breakpoint

Assuming that you have compiled the **fibonacci.c** file listed in Example 7.1, “Compiling a C Program With Debugging Information” with debugging information, you can set a new breakpoint at line 10 by running the following command:

```
(gdb) break 10
Breakpoint 1 at 0x4004e5: file fibonacci.c, line 10.
```

Listing Breakpoints

To display a list of currently set breakpoints, run the following command:

```
info breakpoints
```

Example 7.6. Listing Breakpoints

Assuming that you have followed the instructions in Example 7.5, “Setting a New Breakpoint”, you can display the list of currently set breakpoints by running the following command:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004e5 in main at
fibonacci.c:10
```

Deleting Existing Breakpoints

To delete a breakpoint that is set at a certain line, run the following command:

```
clear line_number
```

Similarly, to delete a breakpoint that is set on a certain function, run:

```
clear function_name
```

Example 7.7. Deleting an Existing Breakpoint

Assuming that you have compiled the **fibonacci.c** file listed in Example 7.1, “Compiling a C Program With Debugging Information” with debugging information, you can set a new breakpoint at line 7 by running the following command:

```
(gdb) break 7
Breakpoint 2 at 0x4004e3: file fibonacci.c, line 7.
```

To remove this breakpoint, type:

```
(gdb) clear 7
Deleted breakpoint 2
```

7.6. Starting Execution

To start an execution of the program you are debugging, run the following command:

```
run
```

If the program accepts any command line arguments, you can provide them as arguments to the **run** command:

```
run argument…
```

The execution stops when the first breakpoint (if any) is reached, when an error occurs, or when the program terminates.

Example 7.8. Executing the fibonacci Binary File

Assuming that you have followed the instructions in Example 7.5, “Setting a New Breakpoint”, you can execute the **fibonacci** binary file by running the following command:

```
(gdb) run
Starting program: /home/john/fibonacci

Breakpoint 1, main (argc=1, argv=0x7fffffffe4d8) at fibonacci.c:10
10          printf("%ld ", b);
```

7.7. Displaying Current Values

The **gdb** utility allows you to display the value of almost anything that is relevant to the program, from a variable of any complexity to a valid expression or even a library function. However, the most common task is to display the value of a variable.

To display the current value of a certain variable, run the following command:

```
print variable_name
```

Example 7.9. Displaying the Current Values of Variables

Assuming that you have followed the instructions in Example 7.8, “Executing the fibonacci Binary File” and the execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10, you can display the current values of variables **a** and **b** as follows:

```
(gdb) print a
$1 = 0
(gdb) print b
$2 = 1
```

7.8. Continuing Execution

To resume the execution of the program you are debugging after it reached a breakpoint, run the following command:

```
continue
```

The execution stops again when another breakpoint is reached. To skip a certain number of breakpoints (typically when you are debugging a loop), you can run the **continue** command in the following form:

```
continue number
```

The **gdb** utility also allows you to stop the execution after executing a single line of code. To do so, run:

```
step
```

Finally, you can execute a certain number of lines by using the **step** command in the following form:

```
step number
```

Example 7.10. Continuing the Execution of the fibonacci Binary File

Assuming that you have followed the instructions in Example 7.8, “Executing the fibonacci Binary File”, and the execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10, you can resume the execution by running the following command:

```
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fffffffe4d8) at fibonacci.c:10
10          printf("%ld ", b);
```

The execution stops the next time the breakpoint is reached. To execute the next three lines of code, type:

```
(gdb) step 3
13          b = sum;
```

This allows you to verify the current value of the **sum** variable before it is assigned to **b**:

```
(gdb) print sum
$3 = 2
```

7.9. Additional Resources

A detailed description of the **GNU Debugger** and all its features is beyond the scope of this book. For more information, see the resources listed below.

Online Documentation

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information on the **GNU Debugger** and debugging.
- GDB Documentation — The official **GDB** documentation includes the *GDB User Manual* and other reference material.

See Also

- Section A.4, “Changes in GDB” provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux system version of the **GNU Debugger** and the version distributed in the previous release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 2, *Eclipse* provides a general introduction to the **Eclipse** development environment, and describes how to use it with the tools from Red Hat Developer Toolset.
- Chapter 3, *GNU Compiler Collection (GCC)* provides further information on how to compile programs written in C, C++, and Fortran.
- Chapter 8, *strace* documents how to use the **strace** utility to monitor system calls that a program uses and signals it receives.
- Chapter 10, *memstomp* documents how to use the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

Chapter 8. strace

strace is a diagnostic and debugging tool for the command line that can be used to trace system calls that are made and received by a running process. It records the name of each system call, its arguments, and its return value, as well as signals received by the process and other interactions with the kernel, and prints this record to standard error output or a selected file.

Red Hat Developer Toolset is distributed with **strace 4.8**.

8.1. Installing strace

In Red Hat Enterprise Linux, the **strace** utility is provided by the *devtoolset-3-strace* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, "Installing Red Hat Developer Toolset".

8.2. Using strace

To run the **strace** utility on a program you want to analyze, type the following at a shell prompt:

```
scl enable devtoolset-3 'strace program [argument...]'
```

Replace *program* with the name of the program you want to analyze, and *argument* with any command line options and arguments you want to supply to this program. Alternatively, you can run the utility on an already running process by using the **-p** command line option followed by the process ID:

```
scl enable devtoolset-3 'strace -p process_id'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **strace** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **strace** you are using at any point, type the following at a shell prompt:

```
which strace
```

Red Hat Developer Toolset's **strace** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **strace**:

```
strace -V
```

8.2.1. Redirecting Output to a File

By default, **strace** prints the name of each system call, its arguments and the return value to standard error output. To redirect this output to a file, use the **-o** command line option followed by the file name:

```
scl enable devtoolset-3 'strace -o file_name program [argument...]'
```

Replace *file_name* with the name of the file.

Example 8.1. Redirecting Output to a File

Consider a slightly modified version of the **fibonacci** file from Example 7.1, “Compiling a C Program With Debugging Information”. This executable file displays the Fibonacci sequence and optionally allows you to specify how many members of this sequence to list. To run the **strace** utility on this file and redirect the trace output to **fibonacci.log**, type:

```
~]$ scl enable devtoolset-3 'strace -o fibonacci.log ./fibonacci 20'
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

This creates a new plain-text file called **fibonacci.log** in the current working directory.

8.2.2. Tracing Selected System Calls

To trace only a selected set of system calls, run the **strace** utility with the **-e** command line option:

```
scl enable devtoolset-3 'strace -e expression program [argument...]'
```

Replace *expression* with a comma-separated list of system calls to trace or any of the keywords listed in Table 8.1, “Commonly Used Values of the -e Option”. For a detailed description of all available values, see the strace(1) manual page.

Table 8.1. Commonly Used Values of the -e Option

Value	Description
file	System calls that accept a file name as an argument.
process	System calls that are related to process management.
network	System calls that are related to networking.
signal	System calls that are related to signal management.
ipc	System calls that are related to inter-process communication (IPC).
desc	System calls that are related to file descriptors.

Example 8.2. Tracing Selected System Calls

Consider the **employee** file from Example 10.1, “Using memstomp”. To run the **strace** utility on this executable file and trace only the **mmap** and **munmap** system calls, type:

```
~]$ scl enable devtoolset-3 'strace -e mmap,munmap ./employee'
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f896c744000
mmap(NULL, 61239, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f896c735000
mmap(0x3146a00000, 3745960, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3146a00000
mmap(0x3146d89000, 20480, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x189000) = 0x3146d89000
mmap(0x3146d8e000, 18600, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3146d8e000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f896c734000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f896c733000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f896c732000
munmap(0x7f896c735000, 61239)           = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f896c743000
John,john@example.comDoe,
+++ exited with 0 +++
```

8.2.3. Displaying Time Stamps

To prefix each line of the trace with the exact time of the day in hours, minutes, and seconds, run the **strace** utility with the **-t** command line option:

```
scl enable devtoolset-3 'strace -t program [argument...]'
```

To also display milliseconds, supply the **-t** option twice:

```
scl enable devtoolset-3 'strace -tt program [argument...]'
```

To prefix each line of the trace with the time required to execute the respective system call, use the **-r** command line option:

```
scl enable devtoolset-3 'strace -r program [argument...]'
```

Example 8.3. Displaying Time Stamps

Consider an executable file named **pwd**. To run the **strace** utility on this file and include time stamps in the output, type:

```
~]$ scl enable devtoolset-3 'strace -tt pwd'
19:43:28.011815 execve("./pwd", ["./pwd"], [/* 36 vars */]) = 0
19:43:28.012128 brk(0)                  = 0xcd3000
19:43:28.012174 mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc869cb0000
19:43:28.012427 open("/etc/ld.so.cache", O_RDONLY) = 3
19:43:28.012446 fstat(3, {st_mode=S_IFREG|0644, st_size=61239, ...}) =
0
19:43:28.012464 mmap(NULL, 61239, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7fc869ca1000
19:43:28.012483 close(3)                = 0
...
19:43:28.013410 +++ exited with 0 +++
```

8.2.4. Displaying a Summary

To display a summary of how much time was required to execute each system call, how many times were these system calls executed, and how many errors were encountered during their execution, run the **strace** utility with the **-c** command line option:

```
scl enable devtoolset-3 'strace -c program [argument...]'
```

Example 8.4. Displaying a Summary

Consider an executable file named **lsblk**. To run the **strace** utility on this file and display a trace summary, type:

```
~]$ scl enable devtoolset-3 'strace -c lsblk > /dev/null'
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 80.88    0.000055           1       106        16 open
 19.12    0.000013           0       140           munmap
  0.00    0.000000           0       148           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0       258           close
  0.00    0.000000           0        37         2 stat
...
------ ----------- ----------- --------- --------- ----------------
100.00    0.000068                  1790        35 total
```

8.3. Additional Resources

A detailed description of **strace** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- strace(1) — The manual page for the **strace** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man strace'
  ```

See Also

- Section B.6, "Changes in strace" provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux 6.6 version of **strace** and the version distributed in the 2.1 release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 9, *ltrace* provides information on how to trace program library calls using the **ltrace** tool.
- Chapter 7, *GNU Debugger (GDB)* provides information on how to debug programs written in C, C++, and Fortran.

- Chapter 10, *memstomp* documents how to use the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

Chapter 9. ltrace

ltrace is a diagnostic and debugging tool for the command line that can be used to display calls that are made to shared libraries. It uses the dynamic library hooking mechanism, which prevents it from tracing calls to statically linked libraries. **ltrace** also displays return values of the library calls. The output is printed to standard error output or to a selected file.

Red Hat Developer Toolset is distributed with **ltrace 0.7.91**.

9.1. Installing ltrace

In Red Hat Enterprise Linux, the **`ltrace`** utility is provided by the *devtoolset-3-ltrace* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, "Installing Red Hat Developer Toolset".

9.2. Using ltrace

To run the **`ltrace`** utility on a program you want to analyze, type the following at a shell prompt:

```
scl enable devtoolset-3 'ltrace program [argument...]'
```

Replace *program* with the name of the program you want to analyze, and *argument* with any command line options and arguments you want to supply to this program. Alternatively, you can run the utility on an already running process by using the **`-p`** command line option followed by the process ID:

```
scl enable devtoolset-3 'ltrace -p process_id'
```

Note that you can execute any command using the **`scl`** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **`ltrace`** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **`ltrace`** you are using at any point, type the following at a shell prompt:

```
which ltrace
```

Red Hat Developer Toolset's **`ltrace`** executable path will begin with **`/opt`**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **`ltrace`**:

```
ltrace -V
```

9.2.1. Redirecting Output to a File

By default, **ltrace** prints the name of each system call, its arguments and the return value to standard error output. To redirect this output to a file, use the **-o** command line option followed by the file name:

```
scl enable devtoolset-3 'ltrace -o file_name program [argument...]'
```

Replace *file_name* with the name of the file.

Example 9.1. Redirecting Output to a File

Consider a slightly modified version of the **fibonacci** file from Example 7.1, “Compiling a C Program With Debugging Information”. This executable file displays the Fibonacci sequence and optionally allows you to specify how many members of this sequence to list. To run the **ltrace** utility on this file and redirect the trace output to **fibonacci.log**, type:

```
~]$ scl enable devtoolset-3 'ltrace -o fibonacci.log ./fibonacci 20'
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

This creates a new plain-text file called **fibonacci.log** in the current working directory.

9.2.2. Tracing Selected Library Calls

To trace only a selected set of library calls, run the **ltrace** utility with the **-e** command line option:

```
scl enable devtoolset-3 'ltrace -e expression program [argument...]'
```

Replace *expression* with a chain of rules to specify the library calls to trace. The rules can consist of patterns that identify symbol names (such as **malloc** or **free**) and patterns that identify library SONAMEs (such as **libc.so**). For example, to trace call to the **malloc** and **free** function but to omit those that are done by the **libc** library, use:

```
scl enable devtoolset-3 'ltrace -e malloc+free-@libc.so* program'
```

Example 9.2. Tracing Selected Library Calls

Consider the **ls** command. To run the **ltrace** utility on this program and trace only the **opendir**, **readdir**, and **closedir** function calls, type:

```
~]$ scl enable devtoolset-3 'ltrace -e opendir+readdir+closedir ls'
ls->opendir(".")      = { 3 }
ls->readdir({ 3 })    = { 61533, "." }
ls->readdir({ 3 })    = { 131, ".." }
ls->readdir({ 3 })    = { 67185100, "BUILDROOT" }
ls->readdir({ 3 })    = { 202390772, "SOURCES" }
ls->readdir({ 3 })    = { 60249, "SPECS" }
ls->readdir({ 3 })    = { 67130110, "BUILD" }
ls->readdir({ 3 })    = { 136599168, "RPMS" }
ls->readdir({ 3 })    = { 202383274, "SRPMS" }
```

```
ls->readdir({ 3 })    = nil
ls->closedir({ 3 })  = 0
BUILD  BUILDROOT  RPMS  SOURCES  SPECS  SRPMS
+++ exited (status 0) +++
```

For a detailed description of available filter expressions, see the ltrace(1) manual page.

9.2.3. Displaying Time Stamps

To prefix each line of the trace with the exact time of the day in hours, minutes, and seconds, run the **ltrace** utility with the **-t** command line option:

```
scl enable devtoolset-3 'ltrace -t program [argument...]'
```

To also display milliseconds, supply the **-t** option twice:

```
scl enable devtoolset-3 'ltrace -tt program [argument...]'
```

To prefix each line of the trace with the time required to execute the respective system call, use the **-r** command line option:

```
scl enable devtoolset-3 'ltrace -r program [argument...]'
```

Example 9.3. Displaying Time Stamps

Consider the **pwd** command. To run the **ltrace** utility on this program and include time stamps in the output, type:

```
~]$ scl enable devtoolset-3 'ltrace -tt pwd'
13:27:19.631371 __libc_start_main([ "pwd" ] <unfinished ...>
13:27:19.632240 getenv("POSIXLY_CORRECT")                    = nil
13:27:19.632520 strrchr("pwd", '/')                          = nil
13:27:19.632786 setlocale(LC_ALL, "")                       =
"en_US.UTF-8"
13:27:19.633220 bindtextdomain("coreutils", "/usr/share/locale") =
"/usr/share/locale"
13:27:19.633471 textdomain("coreutils")                      =
"coreutils"
...
13:27:19.637110 +++ exited (status 0) +++
```

9.2.4. Displaying a Summary

To display a summary of how much time was required to execute each system call and how many times were these system calls executed, run the **ltrace** utility with the **-c** command line option:

```
scl enable devtoolset-3 'ltrace -c program [argument...]'
```

Example 9.4. Displaying a Summary

Consider the **lsblk** command. To run the **ltrace** utility on this program and display a trace summary, type:

```
~]$ scl enable devtoolset-3 'ltrace -c lsblk > /dev/null'
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 53.60    0.261644      261644         1 __libc_start_main
  4.48    0.021848          58       374 mbrtowc
  4.41    0.021524          57       374 wcwidth
  4.39    0.021409          57       374 __ctype_get_mb_cur_max
  4.38    0.021359          57       374 iswprint
  4.06    0.019838          74       266 readdir64
  3.21    0.015652          69       224 strlen
...
------ ----------- ----------- --------- --------------------
100.00    0.488135                  3482 total
```

9.3. Additional Resources

A detailed description of **ltrace** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- ltrace(1) — The manual page for the **ltrace** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man ltrace'
  ```

Online Documentation

- ltrace for RHEL 6 and 7 — This article on the Red Hat Developer Blog offers additional in-depth information (including practical examples) on how to use **ltrace** for application debugging.

See Also

- Section B.7, "Changes in ltrace" provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux 6.6 version of **ltrace**.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 8, *strace* provides information on how to trace program system calls using the **strace** tool.
- Chapter 7, *GNU Debugger (GDB)* provides information on how to debug programs written in C, C++, and Fortran.
- Chapter 10, *memstomp* documents how to use the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

Chapter 10. memstomp

memstomp is a command line tool that can be used to identify function calls with overlapping memory regions in situations when such an overlap is not permitted by various standards. It intercepts calls to the library functions listed in Table 10.1, “Function Calls Inspected by memstomp” and for each memory overlap, it displays a detailed backtrace to help you debug the problem.

Similarly to **Valgrind**, the **memstomp** utility inspects applications without the need to recompile them. However, it is much faster than this tool and therefore serves as a convenient alternative to it.

Red Hat Developer Toolset is distributed with **memstomp 0.1.5**.

Table 10.1. Function Calls Inspected by memstomp

Function	Description
`memcpy`	Copies *n* bytes from one memory area to another and returns a pointer to the second memory area.
`memccpy`	Copies a maximum of *n* bytes from one memory area to another and stops when a certain character is found. It either returns a pointer to the byte following the last written byte, or NULL if the given character is not found.
`mempcpy`	Copies *n* bytes from one memory area to another and returns a pointer to the byte following the last written byte.
`strcpy`	Copies a string from one memory area to another and returns a pointer to the second string.
`stpcpy`	Copies a string from one memory area to another and returns a pointer to the terminating null byte of the second string.
`strncpy`	Copies a maximum of *n* characters from one string to another and returns a pointer to the second string.
`stpncpy`	Copies a maximum of *n* characters from one string to another. It either returns a pointer to the terminating null byte of the second string, or if the string is not null-terminated, a pointer to the byte following the last written byte.
`strcat`	Appends one string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
`strncat`	Appends a maximum of *n* characters from one string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
`wmemcpy`	The wide-character equivalent of the `memcpy`() function that copies *n* wide characters from one array to another and returns a pointer to the second array.
`wmempcpy`	The wide-character equivalent of the `mempcpy`() function that copies *n* wide characters from one array to another and returns a pointer to the byte following the last written wide character.
`wcscpy`	The wide-character equivalent of the `strcpy`() function that copies a wide-character string from one array to another and returns a pointer to the second array.
`wcsncpy`	The wide-character equivalent of the `strncpy`() function that copies a maximum of *n* wide characters from one array to another and returns a pointer to the second string.

Function	Description
wcscat	The wide-character equivalent of the **strcat**() function that appends one wide-character string to another while overwriting the terminating null byte of the second string and adding a new one at its end. It returns a pointer to the new string.
wcsncat	The wide-character equivalent of the **strncat**() function that appends a maximum of *n* wide characters from one array to another while overwriting the terminating null byte of the second wide-character string and adding a new one at its end. It returns a pointer to the new string.

10.1. Installing memstomp

In Red Hat Developer Toolset, the **memstomp** utility is provided by the *devtoolset-3-memstomp* package and is automatically installed with *devtoolset-3-toolchain* as described in Section 1.5, "Installing Red Hat Developer Toolset".

10.2. Using memstomp

To run the **memstomp** utility on a program you want to analyze, type the following at a shell prompt:

```
scl enable devtoolset-3 'memstomp program [argument...]'
```

To immediately terminate the analyzed program when a problem is detected, run the utility with the **--kill** (or **-k** for short) command line option:

```
scl enable devtoolset-3 'memstomp --kill program [argument...]'
```

The use of the **--kill** option is especially recommended if you are analyzing a multi-threaded program; the internal implementation of backtraces is not thread-safe and running the **memstomp** utility on a multi-threaded program without this command line option can therefore produce unreliable results.

Additionally, if you have compiled the analyzed program with the debugging information or this debugging information is available to you, you can use the **--debug-info** (or **-d**) command line option to produce a more detailed backtrace:

```
scl enable devtoolset-3 'memstomp --debug-info program [argument...]'
```

For detailed instructions on how to compile your program with the debugging information built in the binary file, see Section 7.2, "Preparing a Program for Debugging". For information on how to install debugging information for any of the Red Hat Developer Toolset packages, see Section 1.5.4, "Installing Debugging Information".

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **memstomp** as default:

```
scl enable devtoolset-3 'bash'
```

Example 10.1. Using memstomp

In the current working directory, create a source file named **employee.c** with the following contents:

```
#include <stdio.h>
#include <string.h>

#define BUFSIZE 80

int main(int argc, char *argv[]) {
  char employee[BUFSIZE] = "John,Doe,john@example.com";
  char name[BUFSIZE] = {0};
  char surname[BUFSIZE] = {0};
  char *email;
  size_t length;

  /* Extract the information: */
  memccpy(name, employee, ',', BUFSIZE);
  length = strlen(name);
  memccpy(surname, employee + length, ',', BUFSIZE);
  length += strlen(surname);
  email = employee + length;

  /* Compose the new entry: */
  strcat(employee, surname);
  strcpy(employee, name);
  strcat(employee, email);

  /* Print the result: */
  puts(employee);

  return 0;
}
```

Compile this program into a binary file named **employee** by using the following command:

```
~]$ scl enable devtoolset-3 'gcc -rdynamic -g -o employee
employee.c'
```

To identify erroneous function calls with overlapping memory regions, type:

```
~]$ scl enable devtoolset-3 'memstomp --debug-info ./employee'
memstomp: 0.1.4 successfully initialized for process employee (pid
14887).


strcat(dest=0x7fff13afc265, src=0x7fff13afc269, bytes=21) overlap for
employee(14887)
        ??:0    strcpy()
        ??:0    strcpy()
        ??:0    _Exit()
        ??:0    strcat()
        employee.c:26   main()
        ??:0    __libc_start_main()
        ??:0    _start()
John,john@example.comDoe,
```

10.3. Additional Resources

A detailed description of **`memstomp`** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- memstomp(1) — The manual page for the **`memstomp`** utility provides detailed information about its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

```
scl enable devtoolset-3 'man memstomp'
```

See Also

- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 7, *GNU Debugger (GDB)* provides information on how to debug programs written in C, C++, and Fortran.
- Chapter 8, *strace* documents how to use the strace utility to monitor system calls that a program uses and signals it receives.
- Chapter 12, *Valgrind* explains how to use **Valgrind** to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.

Part V. Performance Monitoring Tools

Chapter 11. SystemTap

SystemTap is a tracing and probing tool that allows users to monitor the activities of the entire system without needing to instrument, recompile, install, and reboot. It is programmable with a custom scripting language, which gives it expressiveness (to trace, filter, and analyze) and reach (to look into the running kernel and applications).

SystemTap can monitor various types of events, such as function calls within the kernel or applications, timers, tracepoints, performance counters, and so on. Some included example scripts produce output similar to **netstat**, **ps**, **top**, and **iostat**, others include pretty-printed function callgraph traces or tools for working around security bugs.

Red Hat Developer Toolset is distributed with **SystemTap 2.6**. This version is more recent than the version included in Red Hat Enterprise Linux and provides numerous bug fixes and enhancements.

Table 11.1. Tools Distributed with SystemTap for Red Hat Developer Toolset

Name	Description
stap	Translates probing instructions into C code, builds a kernel module, and loads it into a running Linux kernel.
stapdyn	The **Dyninst** backend for **SystemTap**.
staprun	Loads, unloads, attaches to, and detaches from kernel modules built with the **stap** utility.
stapsh	Serves as a remote shell for **SystemTap**.
stap-prep	Determines and—if possible—downloads the kernel information packages that are required to run **SystemTap**.
stap-merge	Merges per-CPU files. This script is automatically executed when the **stap** utility is executed with the **-b** command line option.
stap-report	Gathers important information about the system for the purpose of reporting a bug in **SystemTap**.
stap-server	A compile server, which listens for requests from **stap** clients.

11.1. Installing SystemTap

In Red Hat Developer Toolset, **SystemTap** is provided by the *devtoolset-3-systemtap* package and is automatically installed with *devtoolset-3-perftools* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

Note

The Red Hat Developer Toolset version of **SystemTap** is available for both Red Hat Enterprise Linux 6 and Red Hat Enterprise Linux 7, but some new features are only offered by the Red Hat Developer Toolset version of **SystemTap** for Red Hat Enterprise Linux 7. See Section A.5, “Changes in SystemTap” for details.

In order to place instrumentation into the Linux kernel, **SystemTap** may also require installation of additional packages with debugging information. To determine which packages to install, run the **stap-prep** utility as follows:

```
scl enable devtoolset-3 'stap-prep'
```

Note that if you execute this command as the `root` user, the utility automatically offers the packages for installation. For more information on how to install these packages on your system, see the *Red Hat Enterprise Linux 6 SystemTap Beginners Guide* or the *Red Hat Enterprise Linux 7 SystemTap Beginners Guide*.

11.2. Using SystemTap

To execute any of the tools that are part of **SystemTap**, type the following at a shell prompt:

```
scl enable devtoolset-3 'tool [option...]'
```

See Table 11.1, “Tools Distributed with SystemTap for Red Hat Developer Toolset” for a list of tools that are distributed with **SystemTap**. For example, to run the **stap** tool to build an instrumentation module, type:

```
scl enable devtoolset-3 'stap [option...] argument...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **SystemTap** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **SystemTap** you are using at any point, type the following at a shell prompt:

```
which stap
```

Red Hat Developer Toolset's **stap** executable path will begin with `/opt`. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **SystemTap**:

```
stap -V
```

11.3. Additional Resources

A detailed description of **SystemTap** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- stap(1) — The manual page for the **stap** command provides detailed information on its usage, as well as references to other related manual pages. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man stap'
  ```

- staprun(8) — The manual page for the **staprun** command provides detailed information on its usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

```
scl enable devtoolset-3 'man staprun'
```

- *SystemTap Tapset Reference Manual* — HTML documentation on the most common tapset definitions is located at **/opt/rh/devtoolset-3/root/usr/share/doc/devtoolset-3-systemtap-client-2.5/index.html**.

Online Documentation

- Red Hat Enterprise Linux 6 SystemTap Beginners Guide and Red Hat Enterprise Linux 7 SystemTap Beginners Guide — The *SystemTap Beginners Guides* for Red Hat Enterprise Linux 6 and 7 provide an introduction to **SystemTap** and its usage.

- Red Hat Enterprise Linux 6 SystemTap Tapset Reference and Red Hat Enterprise Linux 7 SystemTap Tapset Reference — The *SystemTap Tapset Reference* for Red Hat Enterprise Linux 6 and 7 provides further details about **SystemTap**.

- The SystemTap Documentation — The official **SystemTap** documentation provides further documentation on **SystemTap**, as well as numerous examples of **SystemTap** scripts.

See Also

- Section A.5, "Changes in SystemTap" provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux system version of **SystemTap** and the version distributed in the previous release of Red Hat Developer Toolset.

- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.

- Chapter 12, *Valgrind* explains how to use **Valgrind** to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.

- Chapter 13, *OProfile* explains how to use **OProfile** to determine which sections of code consume the greatest amount of CPU time and why.

- Chapter 14, *Dyninst* documents how to use the Dyninst library to instrument a user-space executable.

Chapter 12. Valgrind

Valgrind is an instrumentation framework that ships with a number of tools for profiling applications. It can be used to detect various memory errors and memory-management problems, such as the use of uninitialized memory or an improper allocation and freeing of memory, or to identify the use of improper arguments in system calls. For a complete list of profiling tools that are distributed with the Red Hat Developer Toolset version of **Valgrind**, see Table 12.1, "Tools Distributed with Valgrind for Red Hat Developer Toolset".

Valgrind profiles an application by rewriting it and instrumenting the rewritten binary. This allows you to profile your application without the need to recompile it, but it also makes **Valgrind** significantly slower than other profilers, especially when performing extremely detailed runs. It is therefore not suited to debugging time-specific issues, or kernel-space debugging.

Red Hat Developer Toolset is distributed with **Valgrind 3.10.1**. This version is more recent than the version included in Red Hat Enterprise Linux and provides numerous bug fixes and enhancements.

Table 12.1. Tools Distributed with Valgrind for Red Hat Developer Toolset

Name	Description
Memcheck	Detects memory management problems by intercepting system calls and checking all read and write operations.
Cachegrind	Identifies the sources of cache misses by simulating the level 1 instruction cache (I1), level 1 data cache (D1), and unified level 2 cache (L2).
Callgrind	Generates a call graph representing the function call history.
Helgrind	Detects synchronization errors in multithreaded C, C++, and Fortran programs that use POSIX threading primitives.
DRD	Detects errors in multithreaded C and C++ programs that use POSIX threading primitives or any other threading concepts that are built on top of these POSIX threading primitives.
Massif	Monitors heap and stack usage.

12.1. Installing Valgrind

In Red Hat Developer Toolset, **Valgrind** is provided by the *devtoolset-3-valgrind* package and is automatically installed with *devtoolset-3-perftools*. If you intend to use **Valgrind** to profile parallel programs that use the Message Passing Interface (MPI) protocol, also install the *devtoolset-3-valgrind-openmpi* package by typing the following at a shell prompt as **root**:

```
yum install devtoolset-3-valgrind-openmpi
```

For detailed instructions on how to install Red Hat Developer Toolset and related packages to your system, see Section 1.5, "Installing Red Hat Developer Toolset".

Note

Note that if you use **Valgrind** in combination with the **GNU Debugger**, it is recommended that you use the version of **GDB** that is included in Red Hat Developer Toolset to ensure that all features are fully supported.

12.2. Using Valgrind

12.2. Using Valgrind

To run any of the **Valgrind** tools on a program you want to profile, type the following at a shell prompt:

```
scl enable devtoolset-3 'valgrind [--tool=tool] program [argument...]'
```

See Table 12.1, “Tools Distributed with Valgrind for Red Hat Developer Toolset” for a list of tools that are distributed with **Valgrind**. The argument of the **--tool** command line option must be specified in lower case, and if this option is omitted, **Valgrind** uses **Memcheck** by default. For example, to run **Cachegrind** on a program to identify the sources of cache misses, type:

```
scl enable devtoolset-3 'valgrind --tool=cachegrind program
[argument...]'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **Valgrind** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of Valgrind you are using at any point, type the following at a shell prompt:

```
which valgrind
```

Red Hat Developer Toolset's **valgrind** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **Valgrind**:

```
valgrind --version
```

12.3. Additional Resources

A detailed description of **Valgrind** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- valgrind(1) — The manual page for the **valgrind** utility provides detailed information on how to use Valgrind. To display the manual page for the version included in Red Hat Developer Toolset, type:

```
scl enable devtoolset-3 'man valgrind'
```

- *Valgrind Documentation* — HTML documentation for **Valgrind** is located at **`/opt/rh/devtoolset-3/root/usr/share/doc/devtoolset-3-valgrind-3.9.0/html/index.html`**.

Online Documentation

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information about **Valgrind** and its **Eclipse** plug-in.

- Red Hat Enterprise Linux 6 Performance Tuning Guide Red Hat Enterprise Linux 7 Performance Tuning Guide — The *Performance Tuning Guides* for Red Hat Enterprise Linux 6 and 7 provide more detailed information about using **Valgrind** to profile applications.

See Also

- Section A.6, “Changes in Valgrind” provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux system version of **Valgrind** and the version distributed in the previous release of Red Hat Developer Toolset.

- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.

- Chapter 10, *memstomp* documents how to use the **memstomp** utility to identify calls to library functions with overlapping memory regions that are not allowed by various standards.

- Chapter 11, *SystemTap* provides an introduction to **SystemTap** and explains how to use it to monitor the activities of a running system.

- Chapter 13, *OProfile* explains how to use **OProfile** to determine which sections of code consume the greatest amount of CPU time and why.

- Chapter 14, *Dyninst* documents how to use the Dyninst library to instrument a user-space executable.

Chapter 13. OProfile

OProfile is a low overhead, system-wide profiler that uses the performance-monitoring hardware on the processor to retrieve information about the kernel and executables on the system, such as when memory is referenced, the number of level 2 cache (L2) requests, and the number of hardware interrupts received. It consists of a configuration utility, a daemon for collecting data, and a number of tools that can be used to transform the data into a human-readable form. For a complete list of tools that are distributed with the Red Hat Developer Toolset version of **OProfile**, see Table 13.1, “Tools Distributed with OProfile for Red Hat Developer Toolset”.

OProfile profiles an application without adding any instrumentation by recording the details of every nth event. This allows it to consume fewer resources than **Valgrind**, but it also causes its samples to be less precise. Unlike **Valgrind**, which only collects data for a single process and its children in user-space, **OProfile** is well suited to collect system-wide data on both user-space and kernel-space processes, and requires **root** privileges to run.

Red Hat Developer Toolset is distributed with **OProfile 0.9.9**. This version is more recent than the version included in Red Hat Enterprise Linux and provides numerous bug fixes and enhancements.

Table 13.1. Tools Distributed with OProfile for Red Hat Developer Toolset

Name	Description
oprofiled	The **OProfile** daemon that collects profiling data.
opcontrol	Starts, stops, and configures the **OProfile** daemon.
opannotate	Generates an annotated source file or assembly listing from the profiling data.
oparchive	Generates a directory containing executable, debug, and sample files.
opgprof	Generates a summary of a profiling session in a format compatible with **gprof**.
ophelp	Displays a list of available events.
opimport	Converts a sample database file from a foreign binary format to the native format.
opjitconv	Converts a just-in-time (JIT) dump file to the Executable and Linkable Format (ELF).
opreport	Generates image and symbol summaries of a profiling session.
ocount	A new tool for counting the number of times particular events occur during the duration of a monitored command.

13.1. Installing OProfile

In Red Hat Developer Toolset, **OProfile** is provided by the *devtoolset-3-oprofile* package and is automatically installed with *devtoolset-3-perftools* as described in Section 1.5, “Installing Red Hat Developer Toolset”.

13.2. Using OProfile

To run any of the tools that are distributed with **OProfile**, type the following at a shell prompt as **root**:

```
scl enable devtoolset-3 'tool [option...]'
```

See Table 13.1, “Tools Distributed with OProfile for Red Hat Developer Toolset” for a list of tools that are distributed with **OProfile**. For example, to use the **ophelp** command to list available events in the XML format, type:

```
scl enable devtoolset-3 'ophelp -X'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Red Hat Developer Toolset binaries used in preference to the Red Hat Enterprise Linux system equivalent. This allows you to run a shell session with Red Hat Developer Toolset **OProfile** as default:

```
scl enable devtoolset-3 'bash'
```

Note

To verify the version of **OProfile** you are using at any point, type the following at a shell prompt:

```
which opcontrol
```

Red Hat Developer Toolset's **opcontrol** executable path will begin with **/opt**. Alternatively, you can use the following command to confirm that the version number matches that for Red Hat Developer Toolset **OProfile**:

```
opcontrol --version
```

13.3. Additional Resources

A detailed description of **OProfile** and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- oprofile(1) — The manual page named **oprofile** provides an overview of **OProfile** and available tools. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man oprofile'
  ```

- opannotate(1), oparchive(1), opcontrol(1), opgprof(1), ophelp(1), opimport(1), opreport(1) — Manual pages for various tools distributed with **OProfile** provide more information on their respective usage. To display the manual page for the version included in Red Hat Developer Toolset, type:

  ```
  scl enable devtoolset-3 'man tool'
  ```

Online Documentation

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information on **OProfile**.

- Red Hat Enterprise Linux 6 Deployment Guide — The *Deployment Guide* for Red Hat Enterprise Linux 6 describes in detail how to install, configure, and start using OProfile on this system.
- Red Hat Enterprise Linux 7 System Administrator's Guide — The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.

See Also

- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 11, *SystemTap* provides an introduction to **SystemTap** and explains how to use it to monitor the activities of a running system.
- Chapter 12, *Valgrind* explains how to use **Valgrind** to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.
- Chapter 14, *Dyninst* documents how to use the Dyninst library to instrument a user-space executable.

Chapter 14. Dyninst

The Dyninst library provides an *application programming interface* (API) for instrumenting and working with user-space executables during their execution. It can be used to insert code into a running program, change certain subroutine calls, or even remove them from the program. It serves as a valuable debugging and performance-monitoring tool. The **Dyninst** API is also commonly used along with **SystemTap** to allow non-**root** users to instrument user-space executables.

Red Hat Developer Toolset is distributed with **Dyninst 8.2.1**.

14.1. Installing Dyninst

In Red Hat Developer Toolset, the Dyninst library is provided by the *devtoolset-3-dyninst* package and is automatically installed with *devtoolset-3-perftools* as described in Section 1.5, "Installing Red Hat Developer Toolset". In addition, it is recommended that you also install the **GNU Compiler Collection** provided by the *devtoolset-3-toolchain* package.

If you intend to write a custom instrumentation for binaries, install the relevant header files by running the following command as **root**:

```
yum install devtoolset-3-dyninst-devel
```

You can also install API documentation for this library by typing the following at a shell prompt as **root**:

```
yum install devtoolset-3-dyninst-doc
```

For a complete list of documents that are included in the *devtoolset-3-dyninst-doc* package, see Section 14.3, "Additional Resources". For detailed instructions on how to install optional packages to your system, see Section 1.5, "Installing Red Hat Developer Toolset".

14.2. Using Dyninst

14.2.1. Using Dyninst with SystemTap

To use Dyninst along with **SystemTap** to allow non-**root** users to instrument user-space executables, run the **stap** command with the **--dyninst** (or **--runtime=dyninst**) command line option. This tells **stap** to translate a **SystemTap** script into C code that uses the Dyninst library, compile this C code into a shared library, and then load the shared library and run the script. Note that when executed like this, the **stap** command also requires the **-c** or **-x** command line option to be specified.

To use the Dyninst runtime to instrument an executable file, type the following at a shell prompt:

```
scl enable devtoolset-3 "stap --dyninst -c 'command' [option...]
[argument...]"
```

Similarly, to use the Dyninst runtime to instrument a user's process, type:

```
scl enable devtoolset-3 "stap --dyninst -x process_id [option...]
[argument...]"
```

See Chapter 11, *SystemTap* for more information about the Red Hat Developer Toolset version of **SystemTap**. For a general introduction to **SystemTap** and its usage, see the *SystemTap Beginners Guide* for Red Hat Enterprise Linux 6 or the *SystemTap Beginners Guide* for Red Hat Enterprise Linux 7.

Example 14.1. Using Dyninst with SystemTap

Consider a source file named **exercise.C** that has the following contents:

```
#include <stdio.h>

void print_iteration(int value) {
  printf("Iteration number %d\n", value);
}

int main(int argc, char **argv) {
  int i;
  printf("Enter the starting number: ");
  scanf("%d", &i);
  for(; i>0; --i)
    print_iteration(i);
  return 0;
}
```

This program prompts the user to enter a starting number and then counts down to 1, calling the **print_iteration()** function for each iteration in order to print the number to the standard output. To compile this program on the command line using the **g++** compiler from Red Hat Developer Toolset, type the following at a shell prompt:

```
~]$ scl enable devtoolset-3 'g++ -g -o exercise exercise.C'
```

Now consider another source file named **count.stp** with the following contents:

```
#!/usr/bin/stap

global count = 0

probe process.function("print_iteration") {
  count++
}

probe end {
  printf("Function executed %d times.\n", count)
}
```

This **SystemTap** script prints the total number of times the **print_iteration()** function was called during the execution of a process. To run this script on the **exercise** binary file, type:

```
~]$ scl enable devtoolset-3 "stap --dyninst -c './exercise' count.stp"
Enter the starting number: 5
Iteration number 5
Iteration number 4
Iteration number 3
Iteration number 2
Iteration number 1
Function executed 5 times.
```

14.2.2. Using Dyninst as a Stand-alone Application

Before using the Dyninst library as a stand-alone application, set the value of the **DYNINSTAPI_RT_LIB** environment variable to the path to the runtime library file. You can do so by typing the following at a shell prompt:

```
export DYNINSTAPI_RT_LIB=/opt/rh/devtoolset-
3/root/usr/lib64/dyninst/libdyninstAPI_RT.so
```

This sets the **DYNINSTAPI_RT_LIB** environment variable in the current shell session.

Example 14.2, “Using Dyninst as a Stand-alone Application” illustrates how to write and build a program to monitor the execution of a user-space process. For a detailed explanation of how to use Dyninst, see the resources listed in Section 14.3, “Additional Resources”.

Example 14.2. Using Dyninst as a Stand-alone Application

Consider the **exercise.C** source file from Example 14.1, “Using Dyninst with SystemTap”: this program prompts the user to enter a starting number and then counts down to 1, calling the **print_iteration()** function for each iteration in order to print the number to standard output.

Now consider another source file named **count.C** with the following contents:

```
#include <stdio.h>
#include <fcntl.h>
#include "BPatch.h"
#include "BPatch_process.h"
#include "BPatch_function.h"
#include "BPatch_Vector.h"
#include "BPatch_thread.h"
#include "BPatch_point.h"

void usage() {
  fprintf(stderr, "Usage: count <process_id> <function>\n");
}

// Global information for counter
BPatch_variableExpr *counter = NULL;

void createCounter(BPatch_process *app, BPatch_image *appImage) {
  int zero = 0;
  counter = app->malloc(*appImage->findType("int"));
  counter->writeValue(&zero);
}

bool interceptfunc(BPatch_process *app,
                   BPatch_image *appImage,
                   char *funcName) {
  BPatch_Vector<BPatch_function *> func;
  appImage->findFunction(funcName, func);
  if(func.size() == 0) {
    fprintf(stderr, "Unable to find function to instrument()\n");
    exit (-1);
  }
  BPatch_Vector<BPatch_snippet *> incCount;
  BPatch_Vector<BPatch_point *> *points;
  points = func[0]->findPoint(BPatch_entry);
  if ((*points).size() == 0) {
    exit (-1);
  }

  BPatch_arithExpr counterPlusOne(BPatch_plus, *counter,
BPatch_constExpr(1));
  BPatch_arithExpr addCounter(BPatch_assign, *counter,
counterPlusOne);

  return app->insertSnippet(addCounter, *points);
}


void printCount(BPatch_thread *thread, BPatch_exitType) {
  int val = 0;
  counter->readValue(&val, sizeof(int));
  fprintf(stderr, "Function executed %d times.\n", val);
}

int main(int argc, char *argv[]) {
   int pid;
   BPatch bpatch;
   if (argc != 3) {
     usage();
     exit(1);
   }
   pid = atoi(argv[1]);
   BPatch_process *app = bpatch.processAttach(NULL, pid);
   if (!app) exit (-1);
   BPatch_image *appImage = app->getImage();
   createCounter(app, appImage);
   fprintf(stderr, "Finding function %s(): ", argv[2]);
   BPatch_Vector<BPatch_function*> countFuncs;
   fprintf(stderr, "OK\nInstrumenting function %s(): ", argv[2]);
   interceptfunc(app, appImage, argv[2]);
   bpatch.registerExitCallback(printCount);
   fprintf(stderr, "OK\nWaiting for process %d to exit...\n", pid);
   app->continueExecution();
   while (!app->isTerminated())
     bpatch.waitForStatusChange();
   return 0;
}
```

Note that a client application is expected to destroy all **`Bpatch`** objects before any of the Dyninst library destructors are called. Otherwise the mutator might terminate unexpectedly with a segmentation fault. To work around this problem, set the **`BPatch`** object of the mutator as a local variable in the **`main()`** function. Or, if you need to use **`BPatch`** as a global variable, manually detach all the mutatee processes before the mutator exits.

This program accepts a process ID and a function name as command line arguments and then prints the total number of times the function was called during the execution of the process. You can use the following **Makefile** to build these two files:

```
DTS      = /opt/rh/devtoolset-3/root
CXXFLAGS = -g -I$(DTS)/usr/include/dyninst
LBITS    := $(shell getconf LONG_BIT)

ifeq ($(LBITS),64)
  DYNINSTLIBS = $(DTS)/usr/lib64/dyninst
else
  DYNINSTLIBS = $(DTS)/usr/lib/dyninst
endif

.PHONY: all
all: count exercise

count: count.C
 g++ $(CXXFLAGS) count.C -I /usr/include/dyninst -c
 g++ $(CXXFLAGS) count.o -L $(DYNINSTLIBS) -ldyninstAPI -o count

exercise: exercise.C
 g++ $(CXXFLAGS) exercise.C -o exercise

.PHONY: clean
clean:
 rm -rf *~ *.o count exercise
```

To compile the two programs on the command line using the **g++** compiler from Red Hat Developer Toolset, run the **make** utility as follows:

```
~]$ scl enable devtoolset-3 make
g++ -g -I/opt/rh/devtoolset-3/root/usr/include/dyninst count.C -c
g++ -g -I/opt/rh/devtoolset-3/root/usr/include/dyninst count.o -L
/opt/rh/devtoolset-3/root/usr/lib64/dyninst -ldyninstAPI -o count
g++ -g -I/opt/rh/devtoolset-3/root/usr/include/dyninst exercise.C -o
exercise
```

This creates new binary files called **exercise** and **count** in the current working directory.

In one shell session, execute the **exercise** binary file as follows and wait for it to prompt you to enter the starting number:

```
~]$ ./exercise
Enter the starting number:
```

Do not enter this number. Instead, start another shell session and type the following at its prompt to set the **DYNINSTAPI_RT_LIB** environment variable and execute the **count** binary file:

```
~]$ export DYNINSTAPI_RT_LIB=/opt/rh/devtoolset-
3/root/usr/lib64/dyninst/libdyninstAPI_RT.so
~]$ ./count `pidof exercise` print_iteration
Finding function print_iteration(): OK
Instrumenting function print_iteration(): OK
Waiting for process 8607 to exit...
```

Now switch back to the first shell session and enter the starting number as requested by the **`exercise`** program. For example:

```
Enter the starting number: 5
Iteration number 5
Iteration number 4
Iteration number 3
Iteration number 2
Iteration number 1
```

When the **`exercise`** program terminates, the **`count`** program displays the number of times the **`print_iteration()`** function was executed:

```
Function executed 5 times.
```

14.3. Additional Resources

A detailed description of Dyninst and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

The *devtoolset-3-dyninst-doc* package installs the following documents in the **`/opt/rh/devtoolset-3/root/usr/share/doc/devtoolset-3-dyninst-doc-8.2.0/`** directory:

- *Dyninst Programmer's Guide* — A detailed description of the Dyninst API is stored in the **`DyninstAPI.pdf`** file.
- *DynC API Programmer's Guide* — An introduction to DynC API is stored in the **`dynC_API.pdf`** file.
- *ParseAPI Programmer's Guide* — An introduction to the ParseAPI is stored in the **`ParseAPI.pdf`** file.
- *PatchAPI Programmer's Guide* — An introduction to PatchAPI is stored in the **`PatchAPI.pdf`** file.
- *ProcControlAPI Programmer's Guide* — A detailed description of ProcControlAPI is stored in the **`ProcControlAPI.pdf`** file.
- *StackwalkerAPI Programmer's Guide* — A detailed description of StackwalkerAPI is stored in the **`stackwalker.pdf`** file.
- *SymtabAPI Programmer's Guide* — An introduction to SymtabAPI is stored in the **`SymtabAPI.pdf`** file.
- *InstructionAPI Reference Manual* — A detailed description of the InstructionAPI is stored in the **`InstructionAPI.pdf`** file.

For information on how to install this package on your system, see Section 14.1, “Installing Dyninst”.

Online Documentation

- Dyninst Home Page — The project home page provides links to additional documentation and related publications.

- Red Hat Enterprise Linux 6 SystemTap Beginners Guide — The *SystemTap Beginners Guide* for Red Hat Enterprise Linux 6 provides an introduction to SystemTap and its usage.
- Red Hat Enterprise Linux 7 SystemTap Beginners Guide — The *SystemTap Beginners Guide* for Red Hat Enterprise Linux 7 provides an introduction to SystemTap and its usage.
- Red Hat Enterprise Linux 6 SystemTap Tapset Reference — The *SystemTap Tapset Reference* for Red Hat Enterprise Linux 6 provides further details about SystemTap.
- Red Hat Enterprise Linux 7 SystemTap Tapset Reference — The *SystemTap Tapset Reference* for Red Hat Enterprise Linux 5 provides further details about SystemTap.

See Also

- Section A.7, “Changes in Dyninst” provides a comprehensive list of features and improvements over the Red Hat Enterprise Linux system version of **Dyninst** and the version distributed in the previous release of Red Hat Developer Toolset.
- Chapter 1, *Red Hat Developer Toolset* provides an overview of Red Hat Developer Toolset and more information on how to install it on your system.
- Chapter 11, *SystemTap* provides an introduction to **SystemTap** and explains how to use it to monitor the activities of a running system.
- Chapter 12, *Valgrind* explains how to use **Valgrind** to profile applications and detect memory errors and memory management problems, such as the use of uninitialized memory, improper allocation and freeing of memory, and the use of improper arguments in system calls.
- Chapter 13, *OProfile* explains how to use **OProfile** to determine which sections of code consume the greatest amount of CPU time and why.

Part VI. Getting Help

Chapter 15. Accessing Red Hat Product Documentation

Red Hat Product Documentation located at https://access.redhat.com/site/documentation/ serves as a central source of information. It is currently translated in 23 languages, and for each product, it provides different kinds of books from release and technical notes to installation, user, and reference guides in HTML, PDF, and EPUB formats.

Below is a brief list of documents that are directly or indirectly relevant to this book.

Red Hat Developer Toolset

- Red Hat Developer Toolset 3.1 Release Notes — The *Release Notes* for Red Hat Developer Toolset 3.1 provide more information about this product.
- Red Hat Software Collections Packaging Guide — The *Software Collections Packaging Guide* explains the concept of Software Collections and documents how to create, build, and extend them.

Red Hat Enterprise Linux

- Red Hat Enterprise Linux 6 Developer Guide and Red Hat Enterprise Linux 7 Developer Guide — The *Developer Guides* for Red Hat Enterprise Linux 6 and 7 provide more information about libraries and runtime support, compiling and building, debugging, and profiling.
- Red Hat Enterprise Linux 6 Installation Guide — The *Installation Guide* for Red Hat Enterprise Linux 6 explains how to obtain, install, and update the system.
- Red Hat Enterprise Linux 6 Installation Guide and Red Hat Enterprise Linux 7 Installation Guide — The *Installation Guides* for Red Hat Enterprise Linux 6 an 7 explain how to obtain, install, and update the system.
- Red Hat Enterprise Linux 6 Deployment Guide — The *Deployment Guide* for Red Hat Enterprise Linux 6 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 6.
- Red Hat Enterprise Linux 7 System Administrator's Guide — The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.

Chapter 16. Contacting Global Support Services

Unless you have a Self-Support subscription, when both the Red Hat Documentation website and Customer Portal fail to provide the answers to your questions, you can contact **Global Support Services** (**GSS**).

16.1. Gathering Required Information

Several items of information should be gathered before contacting GSS.

Background Information

Ensure you have the following background information at hand before calling GSS:

- Hardware type, make, and model on which the product runs
- Software version
- Latest upgrades
- Any recent changes to the system
- An explanation of the problem and the symptoms
- Any messages or significant information about the issue

Note

If you ever forget your Red Hat login information, it can be recovered at https://access.redhat.com/site/help/LoginAssistance.html.

Diagnostics

The diagnostics report for Red Hat Enterprise Linux is required as well. This report is also known as a *sosreport* and the program to create the report is provided by the *sos* package. To install the *sos* package and all its dependencies on your system, type the following at a shell prompt as **root**:

```
yum install sos
```

To generate the report, run as **root**:

```
sosreport
```

For more information, access the Knowledgebase article at https://access.redhat.com/kb/docs/DOC-3593.

Account and Contact Information

In order to help you, GSS requires your account information to customize their support, as well contact information to get back to you. When you contact GSS ensure you have your:

- Red Hat customer number or Red Hat Network (RHN) login name

- Company name
- Contact name
- Preferred method of contact (phone or email) and contact information (phone number or email address)

Issue Severity

Determining an issue's severity is important to allow the GSS team to prioritize their work. There are four levels of severity.

Severity 1 (urgent)

A problem that severely impacts your use of the software for production purposes. It halts your business operations and has no procedural workaround.

Severity 2 (high)

A problem where the software is functioning, but production is severely reduced. It causes a high impact to business operations, and no workaround exists.

Severity 3 (medium)

A problem that involves partial, non-critical loss of the use of the software. There is a medium to low impact on your business, and business continues to function by utilizing a workaround.

Severity 4 (low)

A general usage question, report of a documentation error, or a recommendation for a future product improvement.

For more information on determining the severity level of an issue, see https://access.redhat.com/support/policy/severity.

Once the issue severity has been determined, submit a service request through the Customer Portal under the **Connect** option, or at https://access.redhat.com/support/contact/technicalSupport.html. Note that you need your Red Hat login details in order to submit service requests.

If the severity is level 1 or 2, then follow up your service request with a phone call. Contact information and business hours are found at https://access.redhat.com/support/contact/technicalSupport.html.

If you have a premium subscription, then after hours support is available for Severity 1 and 2 cases.

Turn-around rates for both premium subscriptions and standard subscription can be found at https://access.redhat.com/support/offerings/production/sla.html.

16.2. Escalating an Issue

If you feel an issue is not being handled correctly or adequately, you can escalate it. There are two types of escalations:

Technical escalation

If an issue is not being resolved appropriately or if you need a more senior resource to attend to it.

Management escalation

If the issue has become more severe or you believe it requires a higher priority.

More information on escalation, including contacts, is available at https://access.redhat.com/support/policy/mgt_escalation.html.

16.3. Re-opening a Service Request

If there is more relevant information regarding a closed service request (such as the problem reoccurring), you can re-open the request via the Red Hat Customer Portal at https://access.redhat.com/support/policy/mgt_escalation.html or by calling your local support center, the details of which can be found at https://access.redhat.com/support/contact/technicalSupport.html.

Important

In order to re-open a service request, you need the original service-request number.

16.4. Additional Resources

For more information, see the resources listed below.

Online Documentation

- Getting Started — The *Getting Started* page serves as a starting point for people who purchased a Red Hat subscription and offers the *Red Hat Welcome Kit* and the *Quick Guide to Red Hat Support* for download.
- How can a RHEL Self-Support subscription be used? — A Knowledgebase article for customers with a Self-Support subscription.
- Red Hat Global Support Services and public mailing lists — A Knowledgebase article that answers frequent questions about public Red Hat mailing lists.

Appendix A. Changes in Version 3.1

The sections below document features and compatibility changes introduced in Red Hat Developer Toolset 3.1.

A.1. Changes in Eclipse

Red Hat Developer Toolset 3.1 is distributed with **Eclipse 4.4.2** and other plugins from the Luna release train SR2 (Service Release 2), which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset.

This section contains a comprehensive list of new features and compatibility changes in this release. For details on how to use these new features, refer to the built-in **Eclipse** documentation.

A.1.1. Changes Since Red Hat Developer Toolset 3.0

- The Eclipse Platform has been updated from version 4.4.0 to 4.4.2. As this is a service release, it contains over 100 bug fixes and optimizations, including:
  - The stability of SWT on GTK3 (on Red Hat Enterprise Linux 7) has been improved.
  - Support for Java 8 has been improved both in the compiler and editing capabilities.
- **Eclipse CDT** (C/C++ Development Tooling) has been updated from version 8.5 to 8.6. This release includes a number of enhancements, including the following:
  - New filtering options have been added to the **Search** view.
  - A new preference option has been added for defining the style of the **`Include Guard`** code.
  - The *autocomplete* and *content assist* functions have been improved.
  - Refactoring has been improved to properly update include statements.
  - Support has been added for **Save** actions similarly to **JDT**.
  - The **GDB** information provided during debugging has been enhanced.
  - The *remote launch* function has been enhanced to allow in-place editing of connections.
- The **Mylyn** task-management subsystem has been updated from version 3.12 to version 3.14. This new release includes the following change:
  - Wikitext now supports the Markdown language and can output the XHTML format.
- The **Eclipse Linux Tools** plug-in collection has been updated from version 3.1 to 3.2. This new minor release includes:
  - The **GCov** and **GProf** plugins have been improved to automate the setting of compiler options and provide a better vizualization of data.
  - The **LibHover** plugin has been improved to parse **Devhelp** data significantly faster.
- **EGit**, a **Git** integration plug-in for **Eclipse**, and **JGit**, a Java library implementing **Git**, have been updated from version 3.4.1 to 3.6.1. This update includes:
  - Additional options have been added to many **JGit** commands.

- Support for **Git** submodules has been added to **JGit**.
- The interactive rebase view has been improved.
- Stash actions are now exposed in the user interface.

A.2. Changes in GCC

Red Hat Developer Toolset 3.1 is distributed with **GCC 4.9.2**, which provides a number of bug fixes and a new feature over the version included in the previous version of Red Hat Developer Toolset.

A.2.1. Changes Since Red Hat Developer Toolset 3.0

The following features have been added since the release of **GCC** in Red Hat Developer Toolset 3.0:

- A new option, **`-fsanitize=`*`kernel-address`***, has been added, which enables **AddressSanitizer** for the Linux kernel.
- The **`-fsanitize-recover`** option has been improved and now allows more fine-grained selection of when it is desirable to recover and when not.

A.3. Changes in elfutils

Red Hat Developer Toolset 3.1 is distributed with **elfutils 0.161**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in the previous release of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

A.3.1. Changes Since Red Hat Developer Toolset 3.0

The following features have been added since the release of **elfutils 0.159** in Red Hat Developer Toolset 3.0:

- The following changes have been introduced in the **libdw** library:
  - The **`dwarf_cu_getdwarf`** and **`dwarf_cu_die`** functions have been added.
  - The non-existing **`DW_TAG_mutable_type`** attribute has been removed from the **`dwarf.h`** header file.
  - The handling of LZMA-compressed kernel modules (**`.ko.xz`**) is now supported.
- The **`eu-unstrip`** utility has a new option, **`-F`** or **`--force`**, for combining files with non-matching ELF headers.
- Support for the ELF v2 application binary interface on Red Hat Enterprise Linux for POWER (little endian) has been added.

A.3.2. Changes Since Red Hat Enterprise Linux 7.1

The following features have been added since the release of **elfutils 0.160** in Red Hat Enterprise Linux 7.1:

- The following new **libdw** features have been introduced:
  - The **`dwarf_peel_type`** function has been added for handling qualified types. It is also used

by **`dwarf_aggregate_size`** to provide the sizes of qualified types.

- The **`dwarf_getmacros`** function now serves both the **`.debug_macro`** and **`.debug_macinfo`** section data transparently.

- The following new interfaces are available for a more generalized inspection of macros and their parameters: **`dwarf_getmacros_off`**, **`dwarf_macro_getsrcfiles`**, **`dwarf_macro_getparamcnt`**, and **`dwarf_macro_param`**.

- The following new attributes have been added to **`dwarf.h`**: **`DW_AT_GNU_deleted`**, **`DW_AT_noreturn`**, **`DW_LANG_C11`**, **`DW_LANG_C_plus_plus_11`**, and **`DW_LANG_C_plus_plus_14`**.

A.4. Changes in GDB

Red Hat Developer Toolset 3.1 is distributed with **GDB 7.8.2**, which provides a number of bug fixes and improvements over the Red Hat Enterprise Linux system versions and the version included in the previous release of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

A.4.1. Changes Since Red Hat Developer Toolset 3.0

The following features have been added since the release of **GDB** in Red Hat Developer Toolset 3.0:

- A new command, **`add-auto-load-scripts-directory`** ***`directory`***, has been added. It adds entries to the list of directories from which automatically loaded scripts are loaded.

- The **`gdb/jit-reader.h`** header file is now installed in the include directory, so that it can be used for writing readers for parsing custom debugging information formats. See the *Custom Debug Info* section in the **GDB** manual for more information (the *devtoolset-3-gdb-doc* package needs to be installed on your system for this documentation to be locally available).

A.5. Changes in SystemTap

Red Hat Developer Toolset 3.1 is distributed with **SystemTap 2.6**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux 6.6 version and the version included in the previous release of Red Hat Developer Toolset. Below is a list of new features in this release.

A.5.1. Changes Since Red Hat Developer Toolset 3.0 and Red Red Hat Enterprise Linux 6.6

The following features have been added since the release of **SystemTap** included in Red Hat Developer Toolset 3.0 and Red Red Hat Enterprise Linux 6.6:

- The **SystemTap** front end (**`stap`**) has been improved in a number of ways.

  - A new option, **`-E`** ***`script`***, has been added, which allows for specifying additional scripts. Note that a main script still needs to be specified using the **`-e`** option or in a file.

  - DWARF process probes can now be bound to a specific process using the following syntax:

    ```
    process(PID).function("*")
    ```

  - **SystemTap** now supports probes with the following syntax:

```
process("PATH").library("PATH").plt("NAME").return
```

- A PID provided for a process probe is now expected by **SystemTap** to correspond to a running process.
- Probes are now printed in a more consistent and precise manner when the listing mode is invoked using the **-l** or **-L** options.
- Enumerated line numbers are now supported by statement probes, which allows for probing discontiguous lines. Use the following syntax:

  ```
  process.statement("example@file.c:3,5-7,9")
  ```

- **SystemTap** now supports moving a probe from the specified line to the nearest line that can be probed using the following syntax:

  ```
  statement("*@file:NUM").nearest
  ```

- The **SystemTap** runtime back end (**staprun**) has been improved in a number of ways.
  - Certain probe types, kprobes, uprobes, and **timer.*s(*NUM*)**, can now be armed and disarmed on the fly. Thus, probe overhead can be avoided when probes are not needed.
  - Non-recursive and loop-free probes now have statement counting suppressed in the generated C code. Counting can be turned back on in the unoptimized mode by using the **-u** option with **stap**.
  - *Statically Defined Tracing* (SDT) probes with operands that refer to symbols are now supported.
- Many new functions have been added to the **tapset** library and many tapsets were simplified by using the new autocast feature.

Note

Incompatibility problems with old scripts can be resolved using the backward-compatibility option, **--compatible *version***, where *version* is the version of **SystemTap** for which the script was written.

A.6. Changes in Valgrind

Red Hat Developer Toolset 3.1 is distributed with **Valgrind 3.10.1**, which provides a number of bug fixes over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

A.6.1. Changes Since Red Hat Developer Toolset 3.0

The following features have been added since the release of **Valgrind** included in Red Hat Developer Toolset 3.0:

- The **--db-attach** option is now deprecated and will be removed in the next feature release. The built-in **Valgrind gdbserver** capabilities, see the **--vgdb** options, are superior and should be used instead.

- Error messages that include stack traces can now show inline function calls if the DWARF debugging information format is available. These can also be used in suppressions.
- Error messages concerning dubious arguments (for example, to `malloc` or `calloc`) now include a stack trace and can be used in suppressions.
- The C++ demangler has been updated for better C++11 support.
- The **Valgrind gdbserver** functionality now supports the display of thread local variables and storage (`__thread`).
- Three new `monitor` commands have been added to **Valgrind**:
  - `v.info location` ***address*** to show more information about an address.
  - `v.info stats` to show various **Valgrind** core and tool statistics.
  - `v.set hostvisibility`, which allows **Valgrind gdbserver** to provide access to **Valgrind** internal host status and memory.
- A new command line option, `--vgdb-stop-at=`***event1***`,`***event2***`,...`, which allows the user to ask the **Valgrind gdbserver** to stop at the start of program execution, at the end of the program execution, and when **Valgrind** aborts on internal errors.
- In the **Valgrind memcheck** tool, the client code can now selectively disable and re-enable reporting of invalid address errors in specific ranges using new client requests.
- The **Valgrind memcheck** tool now supports more accurate checking of system-call parameters that use uninitialized fields in structures.
- In the **Valgrind memcheck** tool, it is now possible to disable the mismatched checking of memory deallocation using the `free` or `delete` functions by specifying the new `--show-mismatched-frees=no|yes` flag, with the default value being `yes`.
- In the **Valgrind helgrind** tool, more information about race conditions and locks is now shown in error messages as the new **Valgrind gdbserver** `monitor` command returns the list of locks, their location, and their status.

A.7. Changes in Dyninst

Red Hat Developer Toolset 3.1 is distributed with **Dyninst 8.2.1**, which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

A.7.1. Changes Since Red Hat Developer Toolset 3.0 and Red Hat Enterprise Linux 7.1

A number of bugs have been fixed. In addition to that, the following features have been added since the release of **Dyninst** in Red Hat Developer Toolset 3.0 and Red Hat Enterprise Linux 7.1:

- Dyninst now fails gracefully when invalid line information is discovered.
- The parsing of DWARF location lists has been improved.
- Dyninst now hides the `linux-gate.so.1` and `linux-vdso.so.1` *Virtual Dynamic Shared Objects* (VSDO) from link maps.

Appendix B. Changes in Version 3.0

The sections below document features and compatibility changes introduced in Red Hat Developer Toolset 3.0.

B.1. Changes in Eclipse

Red Hat Developer Toolset 3.0 is distributed with **Eclipse 4.4** and other plugins from the Luna release train, which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset.

This section contains a comprehensive list of new features and compatibility changes in this release. For details on how to use these new features, refer to the built-in **Eclipse** documentation.

B.1.1. Changes Since Red Hat Enterprise Linux 6.6

The following features have been added since the release of **Eclipse** in Red Hat Enterprise Linux 6.6:

- The **Eclipse C/C++ Development Toolkit** (CDT) has been updated. This update includes:
  - a greatly enhanced indexer, both in terms of capabilities and speed;
  - improved GNU Debugger integration;
  - support for GNU Debugger's pretty-printing;
  - multi-process debugging;
  - project-less debugging;
  - enhanced refactoring;
  - code-checking capabilities;
  - an improved and more polished user interface;
  - fixes related to the organization of **`include`** statements.
- **Mylyn**, a set of task and application life cycle management plug-ins, has been updated. This update includes:
  - an enhanced task-focused interface and task editing;
  - a new Jenkins/Hudson connector;
  - enhanced Bugzilla and Trac connectors;
  - EPUB authoring tools;
  - various improvements to the Bugzilla connector.
- A new **EGit** plug-in has been added. This plug-in includes an Eclipse Team provider based on JGit, a Git implementation written entirely in Java, and adds support for the Git revision control system to the Eclipse user interface by introducing the **`History`** and **`Synchronize`** views, **`Compare`** and **`Quick Diff`** menu items, and various wizards to make it easier for developers to use this plug-in.

Further updates of the **EGit** plug-in have introduced the following improvements:

- support for Mylin links in the **History** view, **`Git Staging`** view, and **`Commit`** dialog window;
- improved Gerrit integration;
- a significantly enhanced **`Git Staging`** view;
- an enhanced command line tool (**`jgit`**);
- several performance improvements.

- New **GCov** integration has been added to allow the user to visualize GCov output in both summarized form and in file editors. As well, graphing capabilities for the data have been added.
- New **GProf** integration has been added to allow the user to view profiling information such as execution time and call graphs. As well, graphing capabilities for the data have been added.
- A new **SystemTap** integration plug-in has been added. This plug-in includes:
  - an editor for the **`.stp`** files with the autocomplete feature;
  - the **Probe** view with a list of probes that are available on the system;
  - the **`Function`** view with a list of functions that are available on the system and can be used in **`.stp`** scripts.

  As well, this plug-in includes integration for running SystemTap scripts and viewing the results in a textual, tabular, and graphical manner. Note that the result sets are updated in a near-runtime way, which allows the user to use this plug-in for longer-running monitoring tasks.
- New kernel **perf** tool integration has been added. This plug-in uses the performance counters subsystem of the Linux kernel to profile applications, makes it easier to analyze the results by hyperlinking to the sources in the workspace projects, simplifies the **perf** tool configuration by selecting the counters to be used, and allows the user to run this tool remotely.
- A unified profiling launcher has been added to provide a single method to launch profiling. It allows the user to select a profiling category (**Memory**, **`Timing`**, or **`Coverage`**) and back ends for this category (such as **OProfile**, **perf**, or **GProf** for **`Timing`**).
- The C/C++ documentation plug-in has been enhanced to recognize and use **gtk-doc** generated documentation and to display it in the Eclipse Help Center.
- The **OProfile** plug-in has been enhanced to support root privilege authentication through **polkit**. This feature is configured automatically.
- The **Valgrind** plug-in now supports the **Helgrind** tool, which can be used to detect synchronization problems such as race conditions, deadlocks caused by incorrect locking order, or misuse of the POSIX **pthreads** API. When a problem is detected, the plug-in displays error markers on the corresponding lines in the source code.
- The **Eclipse Linux Tools** plug-in collection has been updated. This update includes:
  - support for the **operf** command line tool in the **OProfile** plug-in; the **operf** command is now used by default to make generic setup easier;
  - an enhanced **RPM Stubby** plug-in, which is now able to generate RPM spec files from Perl makefiles, Ruby gamespec files, and Python setup.py files;

  - an enhanced **Spec File Editor** plug-in, which is now able to download source files, prepare a buildroot, and also provides a **`Build`** menu, a corresponding toolbar, and improved support for hyperlinks in the editor;
  - an enhanced **SystemTap `.stp`** files editor, which now has improved formatting capabilities;
  - improved **SystemTap** graphing, which has been streamlined to be easier to launch and configure.

B.1.2. Changes Since Red Hat Developer Toolset 2.1

- The Eclipse Platform has been updated from version 4.3.1 to 4.4.0. As this is a feature release, it contains a number of new features, bug fixes, and optimizations, including the following:
  - SWT (Standard Widget Toolkit) gained a **GTK+3** backend, which provides for a cohesive look and feel with the default desktop user environment on Red Hat Enterprise Linux 7, thus streamlining the user experience. There is also a preliminary support for **WebKit 2.x** for an enhanced stability of the **Browser** component.
  - Split-editor mode is now supported. Pressing the **Ctrl**+**_** keyboard shortcut splits the editor horizontally, and **Ctrl**+**{** splits the editor vertically.
  - A dark user-interface theme is now available as a technology preview. Access it by selecting the **Appearance** preference page in the **General** settings.
  - Java 8 is now supported both for compiling and in generic IDE features, such as for quick fixes to make use of Java 8 features (for example, converting anonymous class creations to lambda expressions).
- **Eclipse CDT** (C/C++ Development Tooling) has been updated from version 8.2 to 8.5. The three feature releases include a number of enhancements, including the following:
  - Toolchain and Language definitions have been improved to allow for a better support for switching between language-standard versions (for example, C++11).
  - Multi-process debugging has been enhanced.
  - Dynamic **`printf`** is now supported as provided by **GDB 7.7**.
  - The **Standalone Debugger** application has been added that provides a graphical user interface for the C/C++ debugging subset of **Eclipse** plug-ins.
  - C++ attributes as defined by C++11 are now supported.
  - The auto completion and content assist features have been improved.
- The **Mylyn** task-management subsystem has been updated from version 3.9.1 to 3.12. This new release includes the following changes:
  - Task integration has been enhanced and now features better search and filtering options.
  - The saving and restoring of breakpoints in the task context is now supported.
- The **Eclipse Linux Tools** plug-in collection has been updated from version 2.2 to 3.1. This new major release includes:
  - The **GCov** plugin has been considerably improved by making it use and annotate standard **CDT** editors instead of **GCov**-specific ones.

- The **GCov** and **GProf** plugins now support an automatic changing of compiler and linker options for projects which simplifies the initial setup.
- The **Systemtap** plugin went through a major overhaul of its user interface, making it easier, faster, and more reliable for generating, editing, and displaying graphics, filtering gathered datasets, and manipulating the probe and function inventories.
- The speed of indexing the Devhelp (GTK+ and GNOME) documentation has been improved significantly.

EGit, a **Git** integration plug-in for **Eclipse**, and **JGit**, a Java library implementing **Git**, have been updated from version 3.1 to 3.4.1. This update includes:

- The command line interface has been greatly enhanced. Commands now support more parameters and new commands have been added.
- Speed has been improved by making use of the Java 7 API.
- Support has been added for performing interactive rebasing in the user interface.
- Blame annotations now expose more information and provide a cross-reference capability.

B.2. Changes in GCC

Red Hat Developer Toolset 3.0 is distributed with **GCC 4.9.1**, which provides a number of bug fixes and new features over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset.

B.2.1. Changes Since Red Hat Enterprise Linux 6.6

The following features have been added since the release of GCC in Red Hat Enterprise Linux 6.6:

B.2.1.1. Status and Features

B.2.1.1.1. 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 B.2.1.5.1, “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.

B.2.1.1.2. 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>).

B.2.1.1.3. 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 B.2.1.1.1, “C++11” and Section B.2.1.1.2, “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.

B.2.1.1.4. Architecture-specific Options

Red Hat Developer Toolset 3.0 is only available for Red Hat Enterprise Linux 6 and 7 for the 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 B.1, “Processor Optimization Options”.

Table B.1. Processor Optimization Options

Option	Description
`-march=core2` and **`-mtune=core2`**	Optimization for Intel Core 2 processors.
`-march=corei7` and **`-mtune=corei7`**	Optimization for Intel Core i3, i5, and i7 processors.
`-march=corei7-avx` and **`-mtune=corei7-avx`**	Optimization for Intel Core i3, i5, and i7 processors with AVX.

Option	Description
`-march=core-avx-i`	Optimization for the Intel processor code-named Ivy Bridge with RDRND, FSGSBASE, and F16C.
`-march=core-avx2`	Optimization for a next-generation processor from Intel with AVX2, FMA, BMI, BMI2, and LZCNT.
`-march=bdver2` and **`-mtune=bdver2`**	Optimization for AMD Opteron processors code-named Piledriver.
`-march=btver1` and **`-mtune=btver1`**	Optimization for AMD family 14 processors code-named Bobcat.
`-march=bdver1` and **`-mtune=bdver1`**	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 B.2, "Support for Processor-specific Intrinsics and Instructions".

Table B.2. 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 **`__builtin_clz*`** using the **`lzcnt`** 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 **`rdrnd`** 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 **`crc32`** intrinsics.
`-mmovbe`	Enables the use of the **`movbe`** instruction to implement **`__builtin_bswap32`** and **`__builtin_bswap64`**.
`-mxop`, **`-mfma4`**, and **`-mlwp`**	Support for the XOP, FMA4, and LWP instruction sets for the AMD Orochi processors.
`-mabm`	Enables the use of the **`popcnt`** and **`lzcnt`** instructions on AMD processors.
`-mpopcnt`	Enables the use of the **`popcnt`** 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.

B.2.1.1.5. 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.

B.2.1.1.6. 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.

B.2.1.2. 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.

B.2.1.2.1. 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.

B.2.1.2.2. 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**.

Experimental C++ Features

g++ now supports a new command line option, **-std=c++1y**.

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;
```

B.2.1.2.3. 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.

B.2.1.3. Fortran

B.2.1.3.1. 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`**.

B.2.1.3.2. 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 **`GFORTRAN_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 **`GFORTRAN_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.

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.*

B.2.1.3.3. 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).
- Support for unlimited polymorphic variables (**CLASS(*)**) has been added. Non-constant character lengths are not yet supported.

B.2.1.3.4. 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.

B.2.1.3.5. 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.

B.2.1.3.6. 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*.

B.2.1.3.7. 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.

B.2.1.3.8. 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.

GFORTRAN_TMPDIR Environment Variable

The **`GFORTRAN_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.

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.

B.2.1.3.9. 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 B.2.1.3.10, “ABI Compatibility”.

B.2.1.3.10. 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.*

B.2.1.4. 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("ssse3")`** 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 Intrinsics

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.

B.2.1.5. ABI Compatibility

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

B.2.1.5.1. 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 3.0 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 6 or 7 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 6 or 7 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 6 or 7 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.

B.2.1.5.2. 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`**.

B.2.1.6. 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.

B.2.1.7. 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 B.2.1.1.5, “Link-time Optimization”), support for the older intermodule optimization framework has been removed and the **`-combine`** command line option is not accepted anymore.

B.2.1.8. 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 paralelism. 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`**.

B.2.1.9. 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 3.0.

Note

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

B.2.1.10. 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`**.

B.2.2. Changes Since Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0

The following features have been added since the release of GCC in Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0:

B.2.2.1. General Changes

Improved Link-Time Optimization

Link-time optimization (LTO) has been improved in a number of ways. Better type merging allows for a faster operation, and less memory is required. Virtual functions are now being removed early in the LTO process, which results in smaller object files and significant improvements to link time.

Improved Inter-Procedural Analysis

Inter-procedural analysis (IPA) has been improved, especially the devirtualization optimization. Also, speculative devirtualization optimization has been added, which can be enabled using the **`-fdevirtualize-speculatively`** option.

Improved Profiling

Support for profiling code has been made more reliable.

Support for the Intel AVX-512 Architecture

GCC now supports the Intel AVX-512 target architecture, as well as a number of new Intel microarchitectures.

New -Wdate-time Warning Option

A new warning option, **`-Wdate-time`**, has been added. It triggers a warning when the **`__DATE__`**, **`__TIME__`** or **`__TIMESTAMP__`** macros are used.

New GCC ivdep Pragma

A new pragma, **`#pragma GCC ivdep`**, has been added. This pragma notifies the compiler that no loop-carried dependencies exist that would prevent a concurrent execution with SIMD (Single Instruction, Multiple Data) of consecutive iterations of the loop that follows the pragma.

Mudflap Tool Removed

The **Mudflap** runtime checking tool was removed from the **GCC** suite and is no longer supported.

B.2.2.2. C and C++ Changes

ISO C11 Support

Support for C11 has been improved:

- ISO C11 atomics (the **`_Atomic`** type specifier and qualifier and the **`<stdatomic.h>`** header) are now supported.
- ISO C11 generic selections (the **`_Generic`** keyword) are now supported.
- ISO C11 thread-local storage (the **`_Thread_local`** keyword) is now supported.

New __auto_type Extension for the C Language

A new C extension, **`__auto_type`**, has been added. It provides a subset of the functionality of C++11 **`auto`** in GNU C.

Support for OpenMP 4.0

The C and C++ compilers now support the OpenMP 4.0 specification. A new option, **`-fopenmp-simd`**, has been added that can be used to enable OpenMP SIMD (Single Instruction, Multiple Data) directives. The vectorization cost model for loops annotated with OpenMP and Cilk+ SIMD directives can be altered using the new **`-fsimd-cost-model`** option. The cost models available are **`unlimited`**, **`dynamic`**, and **`cheap`**.

Improved C++14 Support

The **`g++`** compiler offers improved support for various features of the C++ standard:

- The C++1y return-type deduction for normal functions has been updated to conform to N3638 (see the accepted proposal at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3638.html).
- C++1y lambda-capture initializers no longer trigger a warning when the **`-std=c++1y`** argument is used. Parenthesized and brace-enclosed initializers are now also accepted.
- C++1y variable-length arrays (VLA) are now supported, as well as initializers and lambda capture by reference. Note that in C++1y mode, **`g++`** complains about VLA usage not permitted by the draft standard.
- The C++1y **`[[deprecated]]`** attribute is now supported for marking classes and functions as deprecated. A message for diagnosis can be included in the attribute.
- C++1y digit separators are now supported. In order to ensure better code legibility, long numeric literals can be subdivided using the single quote character, **`'`**. For example:

  ```
  int i = 123'456'789;
  ```

- C++1y generic (polymorphic) lambdas are now supported. They serve as functional objects that increment any type. For example:

```
auto incr = [](auto x) { return x++; };
```

Improved C++11 and C++14 Support in the C++ Runtime Library

The C++ runtime library C++11 support has been improved.

- The **`<regex>`** header is now supported.
- Experimental support for the upcoming ISO C++ standard, C++14, has been added.

New Undefined-Behavior Detector

`UndefinedBehaviorSanitizer` (**ubsan**), a fast undefined-behavior detector, has been added and can be enabled using the **`-fsanitize=undefined`** command line option. Various computations will be instrumented to detect undefined behavior at run time. **`UndefinedBehaviorSanitizer`** is currently available for the C and C++ languages.

Support for Cilk+

GCC now adds support for Cilk+, an extension to the C and C++ languages for parallel programming. It can be enabled using the **`-fcilkplus`** option. The current support implements all features of the version 1.2 ABI, with the exception of **`_Cilk_for`**.

A new runtime library, **`libcilkrts`**, is included in this release to support Cilk+. The **`libcilkrts`** library will be a part of the *gcc-libraries* package in the future Red Hat Enterprise Linux releases, but the package is not included in all supported Red Hat Enterprise Linux releases. To enable dynamic linkage of binaries and libraries built with Red Hat Developer Toolset 3.1 **GCC** using Cilk+ features on supported Red Hat Enterprise Linux releases that do not contain **`libcilkrts`**, install the **`libcilkrts.so`** shared library from Red Hat Developer Toolset 3.1 with such binaries or libraries.

B.3. Changes in binutils

Red Hat Developer Toolset 3.0 is distributed with **binutils 2.24**, which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.3.1. Changes Since Red Hat Enterprise Linux 6.6

The following features have been added since the release of **binutils** in Red Hat Enterprise Linux 6.6:

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.

B.3.1.1. 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 3.0, the **`gold`** linker is not enabled by default. Users can explicitly switch between **`ld`** and **`gold`** by using the **`alternatives`** mechanism.

B.3.1.1.1. New Features

The following features have been added since the release of **binutils** included in Red Hat Enterprise Linux 6.6:

- 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.

B.3.1.1.2. Compatibility Changes

The following compatibility changes have been made since the release of **binutils** included in Red Hat Enterprise Linux 6.6:

- 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.

B.3.1.2. GNU Assembler

B.3.1.2.1. New Features

The following features have been added since the release of **binutils** included in Red Hat Enterprise Linux 6.6:

- 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.

B.3.1.3. Other Binary Tools

B.3.1.3.1. New Features

The following features have been added since the release of **binutils** included in Red Hat Enterprise Linux 6.6:

- 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.

B.3.2. Changes Since Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0

The following bugs have been fixed and features added since the release of **binutils** in Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0:

- The detection of uncompressed **.debug_str** sections has been fixed.
- The decoding of abbreviations using the **DW_FORM_ref_addr** attribute has been fixed.
- The **objcopy** utility now supports wildcards for section names in command line options.
- The BFD linker script language now supports the **ALIGN_WITH_INPUT** directive for output sections. The directive directs the linker to compute the maximum alignment of the associated input sections and use that alignment for the output section.
- The AVX-512 (512-bit Advanced Vector Extensions) are now supported.

B.4. Changes in elfutils

Red Hat Developer Toolset 3.0 is distributed with **elfutils 0.159**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in the previous release of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.4.1. Changes Since Red Hat Developer Toolset 2.1

The following features have been added since the release of **elfutils 0.157** in Red Hat Developer Toolset 2.1:

- The **libdwfl** library has been improved in a number of ways:

  - The **dwfl_core_file_report** has a new parameter, **executable**.
  - The following new functions have been added: **dwfl_module_getsymtab_first_global**, **dwfl_module_getsym_info**, and **dwfl_module_addrinfo**.
  - An unwinder has been added with a **Dwfl_Thread_Callbacks** type, two opaque types (**Dwfl_Thread** and **Dwfl_Frame**), and the following functions: **dwfl_attach_state**, **dwfl_pid**, **dwfl_thread_dwfl**, **dwfl_thread_tid**, **dwfl_frame_thread**, **dwfl_thread_state_registers**, **dwfl_thread_state_register_pc**, **dwfl_getthread_frames**, **dwfl_getthreads**, **dwfl_thread_getframes**, and **dwfl_frame_pc**.

- The **eu-addr2line** utility has a new option, **-x** or **--symbol-sections**, to show the section in which an address was found.
- A new utility, **eu-stack**, has been added that uses the new unwinder for processes and cores.
- A new backend has been added. The **unwinder** now supports the ARM 64-bit architecture, which allows for manipulation of ELF and DWARF files specific for ARM 64-bit.
- The CVE-2014-0172 has been fixed. The bug was caused by an integer overflow that led to a heap-based buffer overflow in the **libdw** library. To fix the problem, an overflow check has been added before calling **malloc** to uncompress data.

B.4.2. Changes Since Red Hat Enterprise Linux 7.0

The following features have been added since the release of **elfutils 0.158** in Red Hat Enterprise Linux 7.0:

- The **eu-stack** utility has two new options:
  - A new option, **-d** or **--debugname**, has been added for looking up DWARF debugging information names for frame addresses.
  - A new option, **-i** or **--inlines**, has been added for showing inlined frames using DWARF debugging information.
- The **libdwelf** library has been improved in a number of ways:
  - A new header file, **libdwelf.h**, has been added for **libdw.so** low-level DWARF or ELF helper functions.
  - New functions have been added: **dwelf_elf_gnu_debuglink**, **dwelf_dwarf_gnu_debugaltlink**, and **dwelf_elf_gnu_build_id**.
- The **libdw** library has been improved in a number of ways:
  - Support for DWZ multifile forms, **DW_FORM_GNU_ref_alt** and **DW_FORM_GNU_strp_alt**, is now enabled by default and is no longer experimental.
  - New functions, **dwarf_getalt** and **dwarf_setalt**, have been added for getting or setting the alternative debugging file used for the alt FORMs.
  - The **dwfl_linux_proc_find_elf** callback now finds ELF from process memory for (deleted) files if the **Dwfl** has a process state attached.
- In the **libdwfl** library, the **dwfl_build_id_find_debuginfo** and **dwfl_standard_find_debuginfo** functions now try to resolve and set an alternative debugging file.

- The **elfutils** backends have been improved in a number of ways:
  - *Call Frame Information* (CFI) unwinding has been added for the ARM architecture. It relies on the **.debug_frame** section of ELF files.
  - A mode compatible with the ARM process initial register state has been added to the ARM 64-bit architecture.
  - Native and core-unwinding support for the ARM 64-bit architecture has been added.
- All separate elfutils-robustify modifications have been included. The elfutils-robustify changes provide functional and stability fixes that are not a part of the official **elfutils** distribution.
- The CVE-2014-0172 has been fixed. The bug was caused by an integer overflow that led to a heap-based buffer overflow in the **libdw** library. To fix the problem, an overflow check has been added before calling **malloc** to uncompress data.

B.5. Changes in GDB

Red Hat Developer Toolset 3.0 is distributed with **GDB 7.8**, which provides a number of improvements and bug fixes over the Red Hat Enterprise Linux system versions and the version included in the previous release of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.5.1. Changes Since Red Hat Enterprise Linux 6.6

The following features have been added since the release of **GDB** in Red Hat Enterprise Linux 6.6:

New Features

- Support for linespecs has been improved (in particular, a more consistent handling of ambiguous linespecs, some support for labels in the program's source, and FILE:LINE support now extends to further linespec types). Breakpoints are now set on all matching locations in all inferiors and will be updated according to changes in the inferior.
- New inferior control commands **skip function** and **skip file** have been added. These commands can be uses to skip certain functions and files when stepping.
- The **info threads** command now displays the thread name as set by **prctl** or **pthread_setname_np**. In addition, new commands **thread name** and **thread find** have been added. The **thread name** command accepts a name as an argument and can be used to set the name of the current thread. The **thread find** command accepts a regular expression and allows the user to find threads that match it.
- GDB now provides support for reading and writing a new **.gdb_index** section. The command **gdb-add-index** can be used to add **.gdb_index** to a file, which allows GDB to load symbols from that file faster. Note that this feature is already present in Red Hat Enterprise Linux 6.1 and later.
- The **watch** command has been adapted to accept **-location** as an optional argument.
- Two new special values can now be used when specifying the current search path for **libthread_db**: **$sdir** represents the default system locations of shared libraries, and **$pdir** stands for the directory with the **libthread** that is used by the application.
- The **info macro** command now accepts **-all** and **--** as valid options.

- To display a function parameter's entry value (that is, the value at the time of function entry), the suffix **@entry** can be added to the parameter. GDB now displays **@entry** values in backtraces, if available.
- The **watch** command now accepts **mask *mask_value*** as an argument. This can be used to create masked watchpoints.
- The **info os** command has been changed and can now display information on several objects managed by the operating system, in particular:
  - The **info os procgroups** command lists process groups.
  - The **info os files** command lists file descriptors.
  - The **info os sockets** command lists internet-domain sockets.
  - The **info os shm** command lists shared-memory regions.
  - The **info os semaphores** command lists semaphores.
  - The **info os msg** command lists message queues.
  - The **info os modules** command lists loaded kernel modules.
- GDB now has support for *Static Defined Tracing* (SDT) probes. Currently, the only implemented back end is for SystemTap probes (the **sys/sdt.h** header file). You can set a breakpoint by using the new **-probe**, **-pstap**, or **-probe-stap** options, and inspect the probe arguments by using the new **$_probe_arg** family of convenience variables.
- The **symbol-reloading** option has been deleted.
- **gdbserver** now supports STDIO connections, for example:

```
(gdb) target remote | ssh myhost gdbserver - hello
```

- GDB is now able to print *flag* enums. In a flag enum, all enumerator values have no bits in common when pairwise AND-ed. When GDB prints a value whose type is a flag enum, GDB shows all the constants; for example, for enum **E { ONE = 1, TWO = 2}**:

```
(gdb) print (enum E) 3
        $1 = (ONE | TWO)
```

- The file name part of a linespec now matches trailing components of a source file name. For example, **break gcc/expr.c:1000** now sets a breakpoint in the **build/gcc/expr.c** file, but not in **build/libcpp/expr.c**.
- The **info proc** and **generate-core-file** commands now work on remote targets connected to **gdbserver**.
- The command **info catch** has been removed.
- The Ada-specific **catch exception** and **catch assert** commands now accept conditions at the end of the command.
- The **info static-tracepoint-marker** command now works on native targets with an in-process agent.
- GDB can now set breakpoints on inline functions.

- The **`.gdb_index`** section has been updated to include symbols for inline functions. By default, GDB now ignores older **`.gdb_index`** sections until their **`.gdb_index`** sections can be recreated. The new command **`set use-deprecated-index-sections on`** causes GDB to use any older **`.gdb_index`** sections it finds. If this option is set, the ability to set breakpoints on inline functions is lost in symbol files with older **`.gdb_index`** sections.

  The **`.gdb_index`** section has also been updated to record more information about each symbol.

- GDB now provides Ada support for GDB/MI Variable Objects.

- GDB now supports **`breakpoint always-inserted mode`** in the **`record`** target.

- **`gdbserver`** now supports evaluation of breakpoint conditions. Note that you can instruct GDB to send the breakpoint conditions in bytecode form, but **`gdbserver`** only reports the breakpoint trigger to GDB when its condition evaluates to true.

- New convenience functions **`$_memeq(buf1, buf2, length)`**, **`$_streq(str1, str2)`**, **`$_strlen(str)`**, and **`$_regex(str, regex)`** have been added.

- Target **`record`** has been renamed to **`record-full`**. Consequently, you can now use the **`record full`** command to record or replay an execution log. In addition, the following commands have been renamed:

  - The **`set record insn-number-max`** and **`show record insn-number-max`** commands have been renamed to **`set record full insn-number-max`** and **`show record full insn-number-max`**.

  - The **`set record memory-query`** and **`show record memory-query`** commands have been renamed to **`set record full memory-query`** and **`show record full memory-query`**.

  - The **`set record stop-at-limit`** and **`show record stop-at-limit`** commands have been renamed to **`set record full stop-at-limit`** and **`show record full stop-at-limit`**.

- A new record target, **`record-btrace`**, has been added. This target uses hardware support to record the control flow of a process and can be enabled by using the **`record btrace`** command. This record target does not support replaying the execution.

  Important

  The **`record-btrace`** target is only available on Intel Atom processors and requires the Linux kernel in version 2.6.32 or later.

- The **`-epoch`** command line option has been removed. This option was used by GDB mode in Epoch, a deprecated clone of the Emacs text editor.

- The **`ptype`** and **`whatis`** commands have been updated to accept an argument to control the type formatting.

- The **`info proc`** command has been updated to work on some core files.

- The **`cd`** command has been enhanced and no longer requires a directory path as its first argument. When executed with no arguments, the command now changes to the home directory.

- GDB now uses *GNU v3 ABI* as the default C++ ABI. This has been the default option for GCC since November 2000.

- The **`info tracepoints`** command has been enhanced to display **`installed on target`** or **`not installed on target`** for each non-pending location of a tracepoint.

New Remote Packets

A number of new remote packets have been added. See Table B.3, “New Remote Packets” for a complete list.

Table B.3. New Remote Packets

Remote Packet	Description
`QTBuffer:size`	Sets the size of the trace buffer. The remote stub reports support for this packet to the **`qSupported`** query.
`Qbtrace:bts`	Enables branch tracing based on *Branch Trace Store* (BTS) for the current thread. The remote stub reports support for this packet to the **`qSupported`** query.
`Qbtrace:off`	Disables branch tracing for the current thread. The remote stub reports support for this packet to the **`qSupported`** query.
`qXfer:btrace:read`	Reads the traced branches for the current thread. The remote stub reports support for this packet to the **`qSupported`** query.
`qXfer:libraries-svr4:read`'s annex	The previously unused annex of the **`qXfer:libraries-svr4:read`** packet is now used to support passing of an argument list. The remote stub reports support for this argument list to the **`qSupported`** query. The defined arguments are **`start`** and **`prev`**. These arguments are used to reduce work necessary for updating the library list and significantly speed up the process.

The **`z0`**/**`z1`** breakpoint insertion packets have been extended to carry a list of conditional expressions over to the remote stub depending on the condition evaluation mode. You can use the **`set remote conditional-breakpoints-packet`** command to control the use of this extension.

New RSP Packet

A new RSP packet has been added:

- A new RSP packet **`QProgramSignals`** can be used to specify the signals the remote stub can pass to the debugged program without GDB involvement.

Changes in the Machine Interface Interpreter (GDB/MI)

The following MI changes have been made:

- A new command **`-info-os`** has been added as the MI equivalent of **`info os`**.
- Output logs, such as **`set logging`** and related, now include MI output.
- A new async record, **`=cmd-param-changed`**, has been added. This async record reports that a command parameter has changed.
- A new async record, **`=traceframe-changed`**, has been added. This async record reports that a trace frame has been changed by using the **`tfind`** command.

- New async records **`=tsv-created`**, **`=tsv-deleted`**, and **`=tsv-modified`** have been added. These async records report that a trace state variable has been created, deleted, or modified.
- New async records **`=record-started`** and **`=record-stopped`** have been added. These async records report that a process record has been started or stopped.
- A new async record, **`=memory-changed`**, has been added. This async record reports that the memory has changed.
- When the source is requested, the **`-data-disassemble`** command now includes a new **`fullname`** field containing an absolute path to the source file name.
- The **`fullname`** filed is now always present along with the **`file`** field. This field is included even if GDB cannot find the file.
- A new optional parameter, **`COUNT`**, has been added to the **`-data-write-memory-bytes`** command. This parameter can be used to allow pattern filling of memory areas.
- The response to breakpoint commands and breakpoint async records now includes a new **`installed`** field. This field reports the current state of each non-pending tracepoint location: when the tracepoint is installed, the value of this field is **`y`**, otherwise the value is **`n`**.
- The output of the **`-trace-status`** command now includes a new **`trace-file`** field. This field is only present when examining a trace file and contains the name of this file.

New Commands

The following new commands have been added:

- New **`set use-deprecated-index-sections on|off`** and **`show use-deprecated-index-sections on|off`** commands have been added. These commands allow you to control the use of deprecated **`.gdb_index`** sections.
- New **`catch load`** and **`catch unload`** commands have been added. These commands allow you to stop execution of a debugged program when a shared library is loaded or unloaded.
- A new **`enable count`** command has been added. This command allows you to auto-disable a breakpoint after several hits.
- A new **`info vtbl`** command has been added. This command allows you to show the virtual method tables for C++ and Java objects.
- A new **`explore`** command has been added. It supports two subcommands **`explore value`** and **`explore type`**, and allows you to recursively explore values and types of expressions. Note that this command is only available with Python-enabled GDB.
- A new **`dprintf`** ***`location,format,args...`*** command has been added. This command allows you to create a dynamic **`printf`**-type breakpoint, which performs a **`printf`**-like operation and then resumes program execution.
- New **`set print symbol`** and **`show print symbol`** commands have been added. These commands allow you to control whether GDB attempts to display the symbol, if any, that corresponds to addresses it prints. This functionality is enabled by default, but you can restore the previous behavior by running the **`set print symbol off`** command.
- New **`record instruction-history`** and **`record function-call-history`** commands have been added. These commands allow you to view information about an execution log without having to replay it. The **`record instruction-history`** command displays the execution history at instruction granularity and the **`record function-call-history`** displays the

execution history at function granularity. The commands are only supported by the **record btrace** command.

- A new command, **fo**, has been added. This commands serves as a shorter variant of the **forward-search** command.
- A new command, **catch signal**, has been added. This command can be used to catch signals by their names and is similar to the **handle** command, but also allows you to attach additional conditions or commands.
- A new command, **maint info bfds**, has been added. This command can be used to list all binary files (BFDs) opened by GDB.
- Two new commands, **python-interactive [*command*]** and its shorter variant **pi [*command*]**, have been added. These commands allow you to start an interactive Python prompt or evaluate a Python command and print the results to standard output.
- A new command, **py [*command*]**, has been added. This command serves as a shorter variant of the **python [*command*]** command.
- New **enable type-printer [*name*...]** and **disable type-printer [*name*...]** commands have been added. These commands allow you to enable or disable type printers.
- New **set breakpoint condition-evaluation** and **show breakpoint condition-evaluation** commands have been added. These commands allow you to control whether breakpoint conditions are evaluated by GDB (the **host** option), or by **gdbserver** (the **target** option). The default option, **auto**, chooses the most efficient available mode.
- New **set dprintf-style gdb|call|agent** and **show dprintf-style** commands have been added. These commands allow you to control the way in which a dynamic **printf** is performed: the **gdb** option requests a GDB **printf** command, **call** causes **dprintf** to call a function in the inferior, and **agent** requests that the target agent such as **gdbserver** does the printing.
- New **set dprintf-function *expression***, **show dprintf-function**, **set dprintf-channel *expression***, and **show dprintf-channel** commands have been added. These commands allow you to set the function and optional first argument to the call when using the **call** style of dynamic **printf**.
- New **set disconnected-dprintf on|off** and **show disconnected-dprintf** commands have been added. These commands allow you to control whether agent-style dynamic **printf**s continue to be in effect after GDB disconnects.
- New **set print type methods on|off** and **show print type methods** commands have been added. These commands allow you to control whether method declarations are displayed by the **ptype** command. This functionality is enabled by default.
- New **set print type typedefs on|off** and **show print type typedefs** commands have been added. These commands allow you to control whether **typedef** definitions are displayed by the **ptype** command. This functionality is enabled by default.
- New **set filename-display basename|relative|absolute** and **show filename-display** commands have been added. These commands allow you to control the way in which file names are displayed: the **basename** option displays only the base name of a file name, **relative** displays a path relative to the compilation directory, and **absolute** displays an absolute path to the file. The default option is **relative** to preserve the previous behavior.
- New **set trace-buffer-size** and **show trace-buffer-size** commands have been added. These commands allow you to control the size of the trace buffer for a target.

- New **`set remote trace-buffer-size-packet auto|on|off`** and **`show remote trace-buffer-size-packet`** commands have been added. These commands allow you to control the use of the remote protocol **`QTBuffer:size`** packet.
- New **`set debug notification`** and **`show debug notification`** commands have been added. These commands allow you to control whether to display debugging information for asynchronous remote notification. This functionality is disabled by default.
- A new command **`info macros`** has been added. This command accepts linespec as an optional argument and can be used to display the definitions of macros at that linespec location. Note that in order to do this, the debugged program must be compiled with the **`-g3`** command line option to have macro information available in it.
- A new command **`alias`** has been added. This command can be used to create an alias of an existing command.
- New **`set print entry-values`** and **`show print entry-values`** commands have been added. The **`set print entry-values`** command accepts **`both`**, **`compact`**, **`default`**, **`if-needed`**, **`no`**, **`only`**, and **`preferred`** as valid arguments and can be used to enable printing of function arguments at function entry. The **`show print entry-values`** command can be used to determine whether this feature is enabled.
- New **`set debug entry-values`** and **`show debug entry-values`** commands have been added. The **`set debug entry-values`** command can be used to enable printing of debugging information for determining frame argument values at function entry and virtual tail call frames.
- **`!`*`command`*** has been added as an alias of **`shell`** ***`command`***.
- New **`set extended-prompt`** and **`show extended-prompt`** commands have been added. The **`set extended-prompt`** command enables support for a defined set of escape sequences that can be used to display various information. The **`show extended-prompt`** command can be used to determine whether the extended prompt is enabled.
- New **`set basenames-may-differ`** and **`show basenames-may-differ`** commands have been added. The **`set basenames-may-differ`** command enables support for source files with multiple base names. The **`show basenames-may-differ`** command can be used to determine whether this support is enabled. The default option is **`off`** to allow faster GDB operations.

New Command Line Options

The following new options have been added:

- A new command line option, **`-ix`** (or **`--init-command`**), has been added. This option acts like **`-x`** (or **`--command`**), but is executed before loading the debugged program.
- A new command line option, **`-iex`** (or **`--init-eval-command`**), has been added. This option acts like **`-ex`** (or **`--eval-command`**), but is executed before loading the debugged program.
- A new command line option, **`-nh`**, has been added. This option allows you to disable automatic loading of the **`~/.gdbinit`** file without disabling other initialization files.

C++ Language Support

The following changes have been made to the C++ language support:

- When debugging a template instantiation, parameters of the template are now put in scope.

Python Scripting Support

The following changes have been made to the Python scripting support:

- The **register_pretty_printer** function in module **gdb.printing** now takes an optional **replace** argument.
- The **maint set python print-stack on|off** command has been deprecated and will be deleted in GDB 7.5. The new command **set python print-stack none|full|message** has replaced it.
- A prompt substitution hook (**prompt_hook**) is now available to the Python API.
- A new Python module **gdb.prompt** has been added to the GDB Python modules library.
- Python commands and convenience-functions located in ***data_directory*/python/gdb/command/** and ***data_directory*/python/gdb/function/** are now automatically loaded on GDB start-up.
- Blocks now provide four new attributes: **global_block**, **static_block**, **is_static**, and **is_global**.
- The **gdb.breakpoint** function has been deprecated in favor of **gdb.breakpoints**.
- A new class **gdb.FinishBreakpoint** is provided.
- Type objects for **struct** and **union** types now allow access to the fields using standard Python dictionary (mapping) methods.
- A new event **gdb.new_objfile** has been added.
- A new function **deep_items** has been added to the **gdb.types** module.
- The function **gdb.Write** now accepts an optional keyword **stream**.
- Parameters can now be sub-classed in Python, which allows for implementation of the **get_set_doc** and **get_show_doc** functions.
- Symbols, Symbol Table, Symbol Table and Line, Object Files, Inferior, Inferior Thread, Blocks, and Block Iterator APIs now have an **is_valid** method.
- Breakpoints can now be sub-classed in Python, which allows for implementation of the **stop** function that is executed each time the inferior reaches that breakpoint.
- A new function **gdb.lookup_global_symbol** has been added. This function can be used to look up a global symbol.
- GDB values in Python are now callable if the value represents a function.
- A new module **gdb.types** has been added.
- A new module **gdb.printing** has been added.
- New commands **info pretty-printers**, **enable pretty-printer**, and **disable pretty-printer** have been added.
- A new **gdb.parameter("directories")** function call is now available.
- A new function **gdb.newest_frame** has been added. This function can be used to return the newest frame in the selected thread.
- The **gdb.InferiorThread** class now supports a new **name** attribute.

- Support for inferior events has been added. Python scripts can now add observers in order to be notified of events occurring in the process being debugged.
- GDB commands implemented in Python can now be put in the **`gdb.COMMAND_USER`** command class.
- The **`maint set python print-stack on|off`** command has been removed and replaced by **`set python print-stack`**.
- A new class **`gdb.printing.FlagEnumerationPrinter`** has been added. This class can be used to apply **`flag enum`**-style pretty-printing to enums.
- The **`gdb.lookup_symbol`** function now works correctly when there is no current frame.
- The **`gdb.Symbol`** object now has an additional attribute **`line`**. This attribute holds the line number in the source at which the symbol was defined.
- The **`gdb.Symbol`** object now has an additional attribute **`needs_frame`**, and a new method **`value`**. The **`needs_frame`** attribute indicates whether the symbol requires a frame to compute its value, and the **`value`** method computes the symbol's value.
- The **`gdb.Value`** object now has a new method **`referenced_value`**. This method can be used to dereference a pointer as well as C++ reference values.
- The **`gdb.Symtab`** object now has two new methods, **`global_block`** and **`static_block`**. These methods return the global and static blocks (as **`gdb.Block`** objects) of the underlying symbol table respectively.
- A new method **`gdb.find_pc_line`** returns the **`gdb.Symtab_and_line`** object associated with a PC value.
- The **`gdb.Symtab_and_line`** object now has an additional attribute **`last`**. This attribute holds the end of the address range occupied by the code for the current source line.
- Users can now create vectors by using the **`gdb.Type.vector()`** method.
- The **`atexit.register()`** method is now supported.
- Users can now pretty-print types by using the Python API.
- In addition to **Python 2.4** and later, GDB now also supports **Python 3**.
- A new class, **`gdb.Architecture`**, has been added. This class exposes the internal representation of the architecture in the Python API.
- A new method, **`Frame.architecture`**, has been added. This method can be used to return the **`gdb.Architecture`** object corresponding to the frame's architecture.
- Frame filters and frame decorators have been added.

Compatibility Changes

- A new command **`info auto-load`** has been added and can be used to display the status of various automatically loaded files. The **`info auto-load gdb-scripts`** command lists automatically loaded canned sequences of commands, **`info auto-load python-scripts`** displays the status of automatically loaded Python scripts, **`info auto-load local-gdbinit`** displays whether a local **`.gdbinit`** file in the current working directory is loaded, and **`info auto-load libthread-db`** displays whether the inferior-specific thread debugging shared library is loaded.

- New commands **`set auto-load`** and **`show auto-load`** have been added and can be used to control automatic loading of files:
  - The **`set auto-load gdb-scripts`** and **`show auto-load gdb-scripts`** commands control automatic loading of GDB scripts.
  - The **`set auto-load python-scripts`** and **`show auto-load python-scripts`** commands control automatic loading of Python scripts.
  - The **`set auto-load local-gdbinit`** and **`show auto-load local-gdbinit`** commands control automatic loading of **`.gdbinit`** from the current working directory.
  - The **`set auto-load libthread-db`** and **`show auto-load libthread-db`** commands control automatic loading of inferior-specific **`libthread_db`**.
  - The **`set auto-load scripts-directory`** and **`show auto-load scripts-directory`** commands control the list of directories from which to automatically load GDB and Python scripts.
  - The **`set auto-load safe-path`** and **`show auto-load safe-path`** commands control the list of directories from which it is safe to automatically load all previously mentioned items.
  - The **`set debug auto-load`** and **`show debug auto-load`** commands control displaying of debugging information for all previously mentioned items.

  The **`set auto-load off`** command can be used to disable automatic loading globally. You can also use **`show auto-load`** with no subcommand to display current settings of all previously mentioned items.
- The **`maint set python auto-load on|off`** command has been replaced with **`set auto-load python-scripts on|off`**.
- The **`maintenance print section-scripts`** command has been renamed to **`info auto-load python-scripts [`*`pattern`*`]`** and is no longer classified as a maintenance-only command.
- Support for the Guile extension language has been removed.
- The GNU Debugger has been adapted to follow GCC's rules on accessing volatile objects when reading or writing target state during expression evaluation.

B.5.2. Changes Since Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0

The following features have been added since the release of **GDB** in Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0:

Python Scripting Support

Python scripting support has been improved:

- Support for frame filters and frame decorators has been added. Frame filters are Python objects that can be used to manipulate whether frames are visible when **GDB** prints a backtrace. Frame decorators, which are sister objects to frame filters, are used to customize the printed content of each **`gdb.Frame`** in commands that execute frame filters.
- Temporary breakpoints, which are automatically deleted after they have been hit, are now supported through the optional ***`temporary`*** argument to the **`Breakpoint`** initializer.

- A representation of line tables has been added. Line tables map source lines to their executable locations in memory. The **`linetable`** function can be used to acquire line-table information for a particular symbol table.
- A new attribute, **`parent_type`**, has been added for **`gdb.Field`** objects.
- **`gdb.Field`** objects can now be used as subscripts on **`gdb.Value`** objects to access structure elements.
- A new attribute, **`name`**, has been added for **`gdb.Type`** objects.
- Valid Python operations on **`gdb.Value`** objects representing structs or classes now invoke the corresponding overloaded operators if available.
- A new feature, *Xmethods*, has been added to the Python API. Xmethods are additional methods or replacements for existing methods of a C++ class. This feature is useful for those cases where a method defined in C++ source code could be inlined or optimized out by the compiler, making it unavailable to **GDB**.

New Commands

A number of new commands have been added:

- A new command, **`catch rethrow`**, has been added. This command works like the **`catch throw`** command, but it only catches re-thrown exceptions.
- A new command, **`maint check-psymtabs`**, has been added. This command has been renamed from the old **`maint check-symtabs`**. It is used for checking the consistency of partial symbol tables.
- A new command, **`maint check-symtabs`**, has been added. It performs consistency checks on full symbol tables.
- A new command, **`maint expand-symtabs`**, has been added. It is used for expanding full symbol tables that match an optional regular expression.
- A new command, **`show configuration`**, has been added. The command displays the details of the configuration options with which **GDB** was built. This command is the equivalent of the **`--configuration`** command line option.
- A new command, **`remove-symbol-file`**, has been added. It can be used to remove a symbol file added using the **`add-symbol-file`** command. The file to remove can be identified by its file name:

  ```
  remove-symbol-file FILENAME
  ```

  Or by an address that lies within the boundaries of this symbol file in memory. In that case, use the **`-a`** option:

  ```
  remove-symbol-file -a ADDRESS
  ```

- A new command, **`info exceptions`**, has been added. It can be used to display the list of Ada exceptions defined in the program being debugged. Optionally, a regular expression can be supplied to limit the list of displayed exceptions:

  ```
  info exceptions REGEXP
  ```

If provided, only the exceptions whose names match *REGEXP* are listed.

The following new pairs of commands make it possible to change settings and show their values:

`set debug symfile [off|on]`

Controls the displaying of debugging information regarding the reading of symbol files and symbol tables within those files. The **`show debug symfile`** command shows the value of this setting.

`set print raw frame-arguments [off|on]`

Controls the displaying of frame arguments in raw mode, disregarding any defined pretty-printers. The **`show print raw frame-arguments`** command shows the value of this setting.

`set remote trace-status-packet off|on|auto`

Controls the use of the remote-protocol **`qTStatus`** packet. The packet is used to ask the remote stub whether a trace experiment is currently running. The **`show remote trace-status-packet`** command shows the value of this setting.

`set range-stepping [off|on]`

Controls whether **GDB** instructs the target to step through the corresponding range of addresses. If disabled, single-steps are always issued. The **`show range-stepping`** command shows the value of this setting. This is set to **`on`** by default.

`set startup-with-shell [off|on]`

Controls whether the shell is used to start subprocesses. If set to **`off`**, processes are started directly. The **`show startup-with-shell`** command shows the value of this setting. This is set to **`on`** by default.

`set code-cache [off|on]`

Controls whether the target memory cache is used for accesses to code segments, disregarding any configured memory regions. This improves performance of remote debugging (particularly disassembly). The **`show code-cache`** command shows the value of this setting. This is set to **`on`** by default.

`set print symbol-loading off|brief|full`

Controls whether to print informational messages when loading symbol information for a file. The default value for this option is **`full`**, but when debugging programs with large numbers of shared libraries, the amount of output becomes less useful. The **`show print symbol-loading`** command shows the value of this setting.

`maint ada set ignore-descriptive-types [on|off]`

Controls whether the debugger ignores descriptive types in Ada programs. The default value is **`off`**, that is, not to ignore the descriptive types. See the user manual for more details on descriptive types and the intended usage of this option. The **`maint ada show ignore-descriptive-types`** command shows the value of this setting.

`set auto-connect-native-target [on|off]`

Controls whether **GDB** may automatically connect to the native target when not connected to any target yet. If set to **`on`**, **GDB** attempts the **`run`**, **`attach`**, and other commands with the native target. The **`show auto-connect-native-target`** command shows the value of this setting.

`set record btrace replay-memory-access read-only|read-write`

Controls what memory accesses are allowed during replays. The **`show record btrace replay-memory-access`** command shows the value of this setting.

`maint set target-async on|off`

Controls whether **GDB** targets operate in synchronous or asynchronous mode. By default and if available, asynchronous mode is used. Set to **`off`** to simplify the debugging of problems that only occur in synchronous mode. The **`maint show target-async`** command shows the value of this setting.

`set mi-async [on|off]`

Controls whether **GDB/MI** uses asynchronous mode. By default, synchronous mode is used. Set to **`on`** to enable background execution of commands. The **`show mi-async`** command shows the value of this setting. This command supersedes the **`set target-async`** command from previous versions of **GDB**.

The following series of new commands enables displaying **GDB** resources used by each executed command:

`maint set per-command space [on|off]`

Enable or disable the displaying of the amount of memory used by each command. The **`maint show per-command space`** command shows the value of this setting.

`maint set per-command time [on|off]`

Enable or disable the displaying of the execution time of each command. The **`maint show per-command time`** command shows the value of this setting.

`maint set per-command symtab [on|off]`

Enable or disable the displaying of basic symbol-table statistics for each command. The **`maint show per-command symtab`** command shows the value of this setting.

New Command Line Options

Two new command line options have been added:

- A new command line option, **`--configuration`**, has been added. It displays the details of the configuration options with which **GDB** was built. This option is the equivalent of the **`show configuration`** interactive command.
- A new command line option, **`-D`**, has been added. This option is an alias for the **`--data-directory`** option. This option allows you to specify where **GDB** looks for its auxiliary files.

Changes in the Machine Interface Interpreter (GDB/MI)

The machine interface interpreter (GDB/MI) has been improved in a number of ways:

- All **MI** commands now accept an optional **`--language`** option. Support for this feature can be verified by using the **`-list-features`** command, which should contain the **`"language-option"`** item.
- A new command, **`-info-gdb-mi-command`**, has been added. It allows the user to determine whether a **GDB/MI** command is supported or not.

- The **`^error`** result record returned when trying to execute an undefined **GDB/MI** command now provides a variable named **`code`** whose content is the undefined-command error code. Support for this feature can be verified by using the **`-list-features`** command, which should contain the **`"undefined-command-error-code"`** item.
- The **`-trace-save`** command can now optionally save trace buffer in the Common Trace Format.
- A new command, **`-dprintf-insert`**, has been added. It sets a dynamic **`printf`** breakpoint.
- The **`-data-list-register-values`** command now accepts an optional **`--skip-unavailable`** option. When used, only available registers are displayed.
- A new command, **`-trace-frame-collected`**, has been added. It returns objects, register names, trace-state variables, memory ranges, and computed expressions that have been collected in a trace frame. There is no corresponding **GDB** command.
- The **`-stack-list-locals`**, **`-stack-list-arguments`**, and **`-stack-list-variables`** commands now accept the **`--skip-unavailable`** option. When used, only available locals or arguments are displayed.
- The **`-exec-run`** command now accepts the optional **`--start`** option. When used, the command follows the same semantics as the **`start`** command, stopping the program's execution at the start of its main subprogram. Support for this feature can be verified by using the **`-list-features`** command, which should contain the **`"exec-run-start-option"`** item.
- New commands, **`-catch-assert`** and **`-catch-exceptions`**, have been added. The commands insert catchpoints stopping the program when Ada exceptions are raised.
- A new command, **`-info-ada-exceptions`**, has been added. It provides the equivalent of the new **`info exceptions`** command in **GDB**.
- A new command, **`-gdb-set mi-async [on|off]`**, has been added. It replaces the **`-gdb-set target-async`** command, which has been deprecated and only serves as an alias to the new command for backward compatibility purposes. The **`-gdb-set mi-async`** command controls whether **MI** operates in asynchronous mode. If set to **`on`**, and provided the target supports it, **MI** commands are executed in the background, and **GDB** interprets them while the target is still running.

New Features in Remote Packets

Two remote packets have had their feature set expanded:

- The **`vCont`** packet now supports a new **`r`** ***`start,end`*** action. The action instructs the remote stub to perform one step and then continue stepping through the address range specified by the *start* (inclusive) and *end* (exclusive) parameters.
- The **`qXfer:btrace:read`** packet now supports a new annex, **`delta`**, to read the branch trace incrementally.

New Features in the GDB Remote Stub, GDBserver

GDBserver has had two new features added and one new option:

- Support for target-assisted range stepping has been added.
- A new element, **`tvar`**, has been added to the XML in the reply to the **`qXfer:traceframe-info:read`** packet. It has the **`id`** of the collected trace state variables.

- A new option, **`--debug-format=option1[,option2, ...]`**, has been added. It allows for adding additional text to each output. At present, only time stamps are supported: **`--debug-format=timestamps`**. Time stamps can also be turned on with the **`monitor set debug-format timestamps`** command in **GDB**.

General Changes

- The Fission DWP file format in version 2 is now supported. To use Fission, supply the **`-gsplit-dwarf`** option to **`gcc`** to generate split DWARF files at compile time. This option must be used in conjunction with the **`-c`** option, which disables linking, because Fission cannot be used when compiling and linking in the same step. See http://gcc.gnu.org/wiki/DebugFission for more information about the Fission project.
- Access to Intel® MPX registers is now supported.
- Support for Intel® AVX-512 registers has been added, that is, support for displaying and modifying the following Intel® AVX-512 registers: **`$zmm0`**—**`$zmm31`**, **`$k0`**—**`$k7`**, **`$xmm16`**—**`$xmm31`**, and **`$ymm16`**—**`$ymm31`**.
- A new convenience function, **`$_isvoid (`*`expression`*`)`**, has been added. The function returns **`1`** if the type of the evaluated *expression* is void. In other cases, it returns **`0`**.
- When displaying the values of registers for which the debugging information shows that they have not been saved in the frame, and there is no place to retrieve them from (callee-saved or call-clobbered registers), the **`<not saved>`** message is now consistently printed.
- A new formatter, **`z`**, has been added. When printing and examining memory, this formatter causes the value to be displayed as a hexadecimal zero padded on the left to the size of the type.
- Target-assigned range stepping with remote targets is now supported. The achieved reduction in the number of control packets sent to and from **GDB** results in an improved performance.
- **GDB** now understands the new **`tvar`** element, which has been added to the XML in the traceframe information. It has the **`id`** of the collected trace state variables.
- The **`typeid`** C++ operator is now supported.
- A new convenience variable, **`$_exception`**, has been added. It holds the exception that is being thrown or caught at an exception-related catchpoint.
- Exception-related catchpoints, such as **`catch throw`**, can now filter exceptions by type using a supplied regular expression.
- A new convenience variable, **`$_exitsignal`**, has been added. It can be used to determine the signal number with which a program terminates if it dies due to an uncaught signal. The **`$_exitcode`** is then automatically set to **`void`**.
- Commands, such as **`c&`**, **`s&`**, can now be executed in the background if the target supports them. In previous versions, asynchronous execution had to be explicitly enabled using the (now deprecated) **`set target-async on`** command.
- The btrace record target now supports the **`record goto`** command. For locations inside the execution trace, the back trace is computed based on the information stored in the execution trace.
- The btrace record target now supports limited reverse execution and replay. The target does not record data and therefore does not allow for reading the memory or registers.

- The naming of native targets has been unified, and they are now all called **`native`**. To reflect this change, the **`target child`** command has been replaced by **`target native`**. Target names feature in the output of the following commands: **`help target`**, **`info target`**, **`info files`**, and **`maint print target-stack`**.

The following commands have been modified:

- The **`unlimited`** literal value can now be used for options that interpret the **`0`** or **`-1`** values as unlimited.
- The **`set debug symtab-create`** debugging command has been changed to accept a verbosity level. **`0`** provides no debugging information, **`1`** provides basic debugging output, and values of **`2`** or greater provide a more verbose output.
- The **`compare-sections`** command now works with all targets. In previous versions, it was limited to the **`remote`** target.
- The **`target native`** command can now be used to explicitly connect to the native target, which overrides the setting of the **`set auto-connect-native-target`** command.
- The ranges given as arguments to the **`record function-call-history`** and **`record instruction-history`** commands are now inclusive.
- The **`record instruction-history`** command now starts counting instructions at one. This also affects the instruction ranges reported by the **`record function-call-history`** command when given the **`/i`** modifier.
- The **`tsave`** command now supports a new option, **`-ctf`**, to save trace data in the *Common Trace Format* (CTF).
- The **`maintenance print objfiles`** command now accepts an optional argument in the form of a regular expression.
- The commands **`set remotebaud`** and **`show remotebaud`** are no longer supported. Use **`set serial baud`** to set the baud rate for remote serial I/O and **`show serial baud`** to display the set rate.
- The **`record function-call-history`** command has been modified in a number of ways:
  - The command now supports a new modifier, **`/c`**, for indenting function names based on their call-stack depth.
  - The fields for the **`/i`** and **`/l`** modifiers have been reordered.
  - The source-line range is now prefixed with **`at`**.
  - The instruction range is now prefixed with **`inst`**. Both ranges are now printed as **`<from>, <to>`** to allow copy-and-paste to the **`record instruction-history`** and **`list`** commands.

B.6. Changes in strace

Red Hat Developer Toolset 3.0 is distributed with **strace 4.8**, which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.6.1. Changes Since Red Hat Enterprise Linux 6.6

The following features have been added since the release of **strace** in Red Hat Enterprise Linux 6.6:

- A new command line option, **`-y`**, has been added. This option can be used to print file descriptor paths.
- A new command line option, **`-P`**, has been added. This option can be used to filter system calls based on the file descriptor paths.
- A new command line option, **`-I`**, has been added. This option can be used to control how interactive **`strace`** is.
- A new command line utility, **`strace-log-merge`**, has been added. This utility can be used to merge timestamped **`strace`** output into a single file.
- The **`strace`** utility now uses optimized interfaces to extract data from the traced process for better performance.
- The **`strace`** utility now provides improved support for decoding of arguments for various system calls. In addition, a number of new system calls are supported.

B.6.2. Changes Since Red Hat Developer Toolset 2.1

The following features have been added since the release of strace in Red Hat Developer Toolset 2.1:

- When writing its output to a pipe, **`strace`** now waits for the pipe process to terminate before **`strace`** itself exits.
- A new option, **`-e trace=memory`**, has been added. It instructs **`strace`** to trace system calls related to memory mapping, allocation, deallocation, and management.
- A new option, **`-qq`**, has been added. It instructs **`strace`** to suppress messages about process exit status.
- Many bugs have been fixed and improvements added, including improvements in the decoding of system-call arguments and the addition of missing system calls.

B.7. Changes in ltrace

Red Hat Developer Toolset 3.0 is distributed with **ltrace 0.7.91**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version. Below is a comprehensive list of new features in this release.

B.7.1. Changes Since Red Hat Enterprise Linux 6.6

A number of bugs have been fixed, and many features have been added since the release of **ltrace** in Red Hat Enterprise Linux 6.6:

- A number of bugs relating to parameter passing were fixed. This includes support for structures passed in registers and fixes for displaying arguments of nested calls. The argument-fetching backends were reimplemented to address a number of issues.
- The tools for selecting what is traced have been improved to be more consistent. It is now possible to trace calls from as well as into the main binary, any shared libraries, or modules loaded using the **`dlopen`** function. It also allows for using globs and regular expressions for selecting whole sets of symbols at the same time. For example, to trace all calls to functions named **`function*`** that **`libexample*`** makes, use:

```
ltrace -e function\*@libexample\*
```

To trace all calls to functions that **libexample*** exports, use:

```
ltrace -l libexample\*
```

- A new approach to prototype discovery has been added. It allows prototype libraries to be supplied with shared libraries. When a shared library is mapped in, **ltrace** looks up the corresponding prototype library and finds its call prototypes there.
- The syntax of prototype libraries was generalized. A new concept of lens has been introduced, which separates the way data types are to be formatted from the different kinds of data types. This allows for declaring, for example, that a function expects a **char** parameter, but that parameter should be formatted as an **enum** or in hexadecimal. It is now also possible to express zero-terminated (NUL, NULL) arrays.
- Unwinder support has been added, so that the user can request a stack trace to be shown for each traced call.
- Support for tracing **IFUNC** symbols has been added. These are special symbols that resolve to one of possible implementations after the first call is made. **ltrace** now re-seats related breakpoints after an **IFUNC** call finishes. It also supports tracing the related **IRELATIVE PLT** slots.

B.8. Changes in SystemTap

Red Hat Developer Toolset 3.0 is distributed with **SystemTap 2.5**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in the previous release of Red Hat Developer Toolset. Below is a list of new features in this release.

B.8.1. Changes Since Red Hat Developer Toolset 2.1

The following features have been added since the release of **SystemTap** included in Red Hat Developer Toolset 2.1:

- The **tapset** library no longer signals certain types of errors by returning dummy values. Instead, fully-fledged **SystemTap** errors are generated that can be caught using the **try { } catch { }** construct. Alternatively, other **tapset** functions may be used to return user-specified dummy values.
- The **dyninst** backend, which can be enabled by passing the **--runtime=dyninst** or **--dyninst** parameter to **stap**, has had a number of compatibility and performance improvements added.
- The quality and detail of error messages have been greatly improved. The improvements include colorization, manual-page cross-referencing, and suggestions for correcting typing mistakes.
- The prologue analysis for placing function-entry probes has been improved.
- A ***function*****().callee(...)** type of probe for probing outgoing calls from given functions has been introduced.
- Many new functions have been added to the **tapset** library.

The following new features are only available on Red Hat Enterprise Linux 7 systems:

- SecureBoot support via a **stap-server** has been extended with key generation and module-signing.

- Support has been added for probing virtual machines from the host without a network. The probing requires a minimal runtime installed within the virtual machine, which communicates through the virtio-serial device.

- Support has been added for using **`.gnu_debugdata`** symbol tables in base executables to resolve certain probe symbols or addresses without the need for full debuginfo.

Note

Incompatibility problems with old scripts can be resolved using the backward-compatibility option, **`--compatible`** ***`version`***, where *version* is the version of **SystemTap** for which the script was written.

B.9. Changes in Valgrind

Red Hat Developer Toolset 3.0 is distributed with **Valgrind 3.9**, which provides a number of bug fixes over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.9.1. Changes Since Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 6.6

The following features have been added since the release of **Valgrind 3.8.1** included in Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 6.6:

B.9.1.1. Changes in Tools

Memcheck

- A new command line option, **`--partial-loads-ok[=`*`argument`*`]`**, has been added. The option accepts the **`yes`** or **`no`** arguments. The default is **`no`**. The option can be used to significantly reduce the number of false error reports when handling vectorised code.

- New command line options, **`--show-leak-kinds[=`*`argument`*`]`** and **`--errors-for-leak-kinds[=`*`argument`*`]`**, have been added. Both options accept a comma-separated list of the following arguments: **`definite`**, **`indirect`**, **`possible`**, **`reachable`**, as well as **`all`** or **`none`**. The default setting for both options is equivalent to passing the **`definite`** and **`possible`** arguments.

  The options allow you to exercise a finer control over the leak checker by specifying the kinds of leaks that are to be shown or suppressed in a full leak search and that are to count as errors in a full leak search. Optionally, you can also specify the kinds of leaks to suppress directly in the suppressions file using the **`match-leak-kinds:`** line. Note that the generated suppressions files now include this line by default, which makes them more specific. To get the old behavior, remove the line prior to using the suppressions file.

- A new command line option, **`--leak-check-heuristics[=`*`argument`*`]`**, has been added. The option accepts a comma-separated list of the following arguments: **`stdstring`**, **`length64`**, **`newarray`**, **`multipleinheritance`**, as well as **`all`** or **`none`**. The default is **`none`**. The option improves the way the leak checker reports on possible leaks by specifying which interior pointers are considered valid.

- A new command line option, **`--keep-stacktraces[=`*`argument`*`]`**, has been added. The option accepts one of the following arguments: **`alloc`**, **`free`**, **`alloc-and-free`**, **`alloc-then-free`**, as well as **`none`**. The default is **`alloc-then-free`**. The option controls which stack traces to keep for blocks allocated using **`malloc()`** or freed using **`free()`**. This produces more usable errors and causes **Valgrind** to consume less resources.

- Using the **`-v`** (or **`--verbose`**) option now provides more detailed information about the use of leak suppressions. For each suppression, the number of blocks and bytes it suppressed during the last leak search is now shown, as well as the file name and line number where the suppression is defined.

Helgrind

- The use of statically initialized mutexes and condition variables (such as **`PTHREAD_MUTEX_INITIALISER`**) no longer results in false errors.
- The use of the **`pthread_cond_wait()`** function that times out no longer results in false errors.

B.9.1.2. Other Changes

The space required by **Valgrind** now better corresponds with the expected capabilities of the target:

- A new command line option, **`--num-transtab-sectors=`*`number`***, has been added. The option can be used to specify the maximum number of sectors in the translation cache. The default number of sectors has been increased to 16 in order to accommodate applications that need to store large amounts of code. This improves performance because **Valgrind** does not need to re-translate and re-instrument machine code so often. The number of memory-mapped segments that can be tracked has been increased by a factor of 6.

The way **Valgrind** reads debugging information has been improved:

- **Valgrind** no longer uses excessive amounts of virtual memory when reading debuginfo from large shared objects. Instead of temporarily mapping entire objects for reading, **Valgrind** now uses only a small, fixed buffer.
- A new command line option, **`--debuginfo-server=`*`ipaddress`*`:`*`port`***, has been added. This option allows you to instruct **Valgrind** to read debuginfo from objects stored on a different machine and served by a debuginfo server, which needs to be accessible on the specified IP address and port.
- A new command line option, **`--allow-mismatched-debuginfo=`*`argument`***, has been added. The option accepts the **yes** and **no** arguments. The default is **no**. Using this option, you can instruct **Valgrind** to ignore inconsistencies between main and debuginfo objects. Use with caution, as **Valgrind** may not be stable when main and debugging objects do not match.

A new command line option, **`--merge-recursive-frame=`*`number`***, has been added. The option allows you to instruct **Valgrind** to detect and merge recursive algorithms, thus saving memory used to store the recorded stack traces. The value supplied to this option determines the complexity of the recursive calls to be merged.

A new command line option, **`--sigill-diagnostics=`*`argument`***, has been added. This option accepts the **yes** and **no** arguments. The default is **yes**. Use this option to control whether **Valgrind** displays a warning when encountering an instruction it is unable to decode or translate.

In order to be able to run applications that natively require up to 35 GB of memory, **Valgrind** can now use 64 GB of memory on 64-bit targets.

GDB Server Monitor

- A new client request, **`VALGRIND_MONITOR_COMMAND`**, has been added to **`valgrind.h`**. Use it to run **gdbserver** monitor commands from client programs.
- A new monitor command, **`v.info open_fds`**, has been added. It provides a list of open file descriptors. Note that this is only available if the **`--track-fds=yes`** option was passed on startup.
- It is now possible to pass an optional message with the **`v.info n_errs_found [`*`message`*`]`** monitor command, so that a comment can be inserted into process outputs. This helps to separate errors reported with different tests.
- A new monitor command, **`v.info exectxt`**, has been added. It displays information about stack traces (executable contexts) recorded by **Valgrind**.
- A new monitor command, **`v.do expensive_sanity_check_general`**, has been added. It runs various consistency checks. The command can be used to verify the sanity of the **Valgrind** heap.

B.10. Changes in OProfile

Red Hat Developer Toolset 3.0 is distributed with **OProfile 0.9.9**, which provides a number of bug fixes and feature enhancements over the version included in the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.10.1. Changes Since Red Hat Developer Toolset 2.1

The following features have been added since the release of **OProfile** in Red Hat Developer Toolset 2.1:

- A new command, **ocount**, has been added. It counts the number of times particular events occur during the time a monitored command is running. For example, it can be used to count the number of cache misses and instructions for an application. A relatively high number of cache misses when compared to instructions could indicate poor cache performance.
- The way the **operf** command operates has been improved, so that more accurate data is collected. The improvements are mainly related to the spawning of new processes.
- Generic AMD performance events are now provided as a fallback for newer AMD processors for which **OProfile** does not have explicit event-sets encodings.

B.11. Changes in Dyninst

Red Hat Developer Toolset 3.0 is distributed with **Dyninst 8.2**, which provides a number of bug fixes and feature enhancements over the version included in Red Hat Enterprise Linux and the previous version of Red Hat Developer Toolset. Below is a comprehensive list of new features in this release.

B.11.1. Changes Since Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0

A number of bugs have been fixed. In addition to that, the following features have been added since the release of **Dyninst** in Red Hat Developer Toolset 2.1 and Red Hat Enterprise Linux 7.0:

- The **ProcControl** library has been improved to provide callbacks whenever a syscall-monitored thread enters or exits a system call.

- The **SymtabAPI** library has been improved to support access to inline functions.
- Checks are now performed and errors returned in case the instrumentation requires traps, but the user has disabled them.
- The generation of XMM registers *save* and *restore* has been improved, and function-call snippets have been optimized as a result.
- Block-level instrumentation performance in shared code has been improved.
- Whenever possible, checks are now performed on parameter and return-value snippets.
- Symbol visibility is now properly controlled, so that only public **Dyninst** functions and variables are exposed to users.

Appendix C. Revision History

Revision 0.0-25	**Thu 04 June 2015**	**Robert Krátký**
Update to reflect RHSCL 2.0 GA.		
Revision 0.0-23	**Thu 23 Apr 2015**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.1 User Guide.		
Revision 0.0-17	**Tue 10 Mar 2015**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.1 Beta User Guide.		
Revision 0.0-16	**Thu 13 Nov 2014**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.0 User Guide with minor post-GA fixes.		
Revision 0.0-14	**Thu 30 Oct 2014**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.0 User Guide.		
Revision 0.0-10	**Tue 07 Oct 2014**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.0 Beta-2 User Guide.		
Revision 0.0-8	**Tue Sep 09 2014**	**Robert Krátký**
Release of Red Hat Developer Toolset 3.0 Beta-1 User Guide.		

Index