Oracle® Solaris Studio 12.4: IDE Quick Start Tutorial

Part No: E37083
October 2014

Contents

♦ ♦ ♦ C H A P T E R 1

Overview of the Oracle Solaris Studio IDE

This chapter describes the advantages of using Oracle Solaris Studio IDE.

Why Use the Oracle Solaris Studio IDE?

Oracle Solaris Studio offers a graphical integrated development environment (IDE) that is built on the NetBeans platform and configured with Oracle Solaris Studio tools suite directly in the environment. In the IDE, you can do the following:

- Compile and build code with **Oracle Solaris Studio C, C++, and Fortran compilers** and the `dmake` **distributed make command**.
- Debug code with `dbx` **debugger** For more information, see Chapter 7, "Debugging Your Application".
- Profile your code with profiling tools that integrate Oracle Solaris Studio analysis tools:
  - **Run Monitor tools** uses Performance Analyzer's data collection.
  - **Memory Analysis tool** uses `discover's` dynamic memory checking. For more information, see "Running Memory Access Checking on Your Project" on page 82.
  - **Static Code Checking** uses Code Analyzer's static error data collection. For more information, see "Using Static Code Error Checking" on page 35.
  - **Data Races and Deadlocks tool** uses Thread Analyzer's data collection and analysis.
- Run the IDE locally while Oracle Solaris Studio compilers and tools installed on a remote server using **remote development** For more information, see "Doing Remote Development" on page 85.
- Create a special distribution of the IDE while it's running on your remote build server with **remote desktop distribution**.

The rest of this guide shows how to use all these features in the IDE.

♦ ♦ ♦ CHAPTER 2

Creating a New Application

This chapter shows the steps for creating a new application project in the IDE, sometimes called a managed project. For more information, see "Creating an Application Project" on page 9.

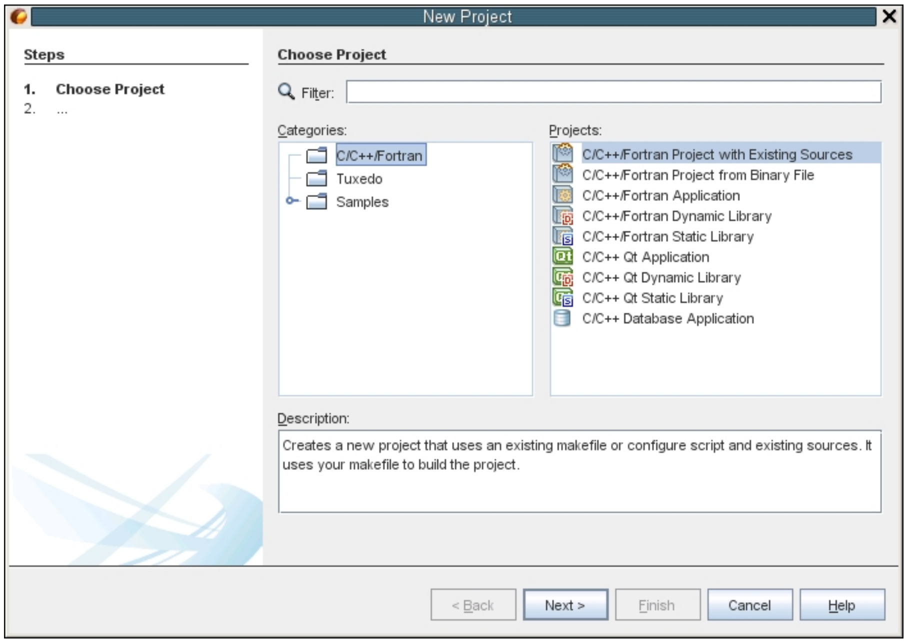
To learn about how to create a project with existing sources, see Chapter 4, "Creating a Project From Existing Sources".

This chapter also explains other tasks you can do with your project, such as the following:

- "Switching Between the Logical View and the Physical View of the Project" on page 11
- "Adding Files to Your Project" on page 12
- "Setting Properties and Configurations" on page 13
- "Building Your Project" on page 15
- "Compiling a Single File" on page 16

Creating an Application Project

1. Open the New Project wizard by choosing File > New Project (Ctrl+Shift+N).
2. In the wizard, select the C/C++/Fortran category.
3. The wizard gives you a choice of several types of new projects. Select C/C++/Fortran Application and click Next.



Creates a new project that uses an existing makefile or configure script and existing sources. It uses your makefile to build the project.

4. Create a new C/C++/Fortran Application project from the wizard using the defaults. You can choose the name of the project and the location of the project.
5. Click Finish to exit the wizard.

A project is created with several logical folders, which are displayed in the Projects window:

- Source Files
- Header Files
- Resource Files
- Test Files
- Important Files

A C,C++, or Fortran Project with Existing Sources is created with one logical folder, Important Files.

A logical folder is not a directory. It is a way for you to organize your files and does not reflect where the files are physically stored on disk. Files added to the Source Files Folder, Header Files folder, and other logical folders become part of the project and are compiled when you build the project.

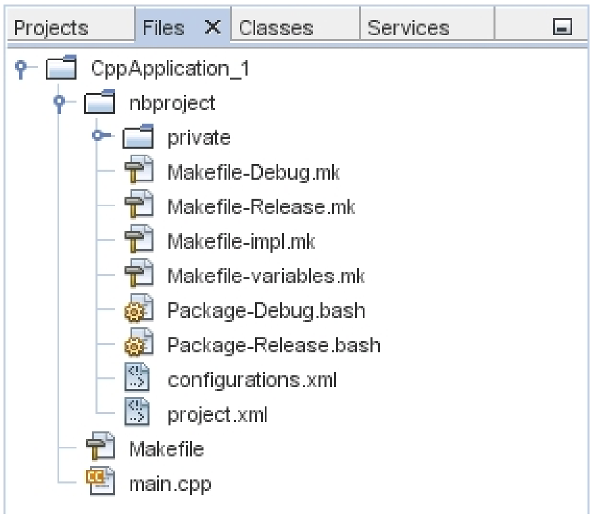
Note - Header files added to the Header Files folder must be referenced in your source files through `#include` statements in the usual way if you want them to be compiled into the program.

Files added to the Important Files folder are not part of the project and are not compiled when you build the project. These files are just for reference and are convenient when you have a project with an existing makefile.

Switching Between the Logical View and the Physical View of the Project

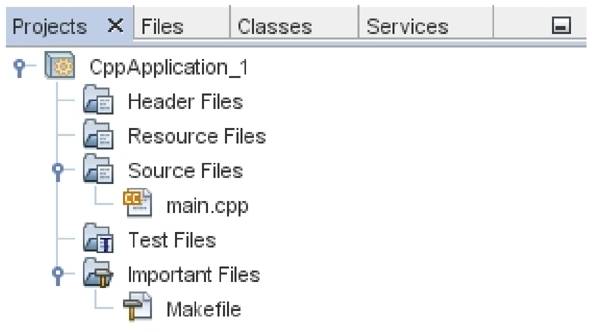
A project has both a logical and a physical view. You can switch between the logical view and the physical view of your project.

1. Select the Files tab. This window shows the physical view of your project. It displays files and folders as they are stored on disk.



```
CppApplication_1
    nbproject
        private
        Makefile-Debug.mk
        Makefile-Release.mk
        Makefile-impl.mk
        Makefile-variables.mk
        Package-Debug.bash
        Package-Release.bash
        configurations.xml
        project.xml
    Makefile
    main.cpp
```

2. Select the Projects tab. This window shows the logical view of your project.



Adding Files to Your Project

The following sections describes how to add different types of files and folders to your current project, either by creating them or introducing existing sources.

- "Creating Logical Files and Folders for Your Project" on page 12
- "Creating New Source Files for Your Project" on page 12
- "Creating Header Files for Your Project" on page 13
- "Adding Existing Files to Your Project" on page 13

Creating Logical Files and Folders for Your Project

You can create logical folders and add them to your project.

1. Right-click the project node of your `CppApplication_1` project and choose New Logical Folder. A new logical folder is added to the project.
2. Right-click the new logical folder and select Rename. Type the name you would like to give the new folder.

You can add both files and folders to an existing folder. Logical folders can be nested.

Creating New Source Files for Your Project

You can create new source files and add them to your project.

1. Right-click the Source Files folder and choose New > C Main File.

2. On the Name and Location page, `newmain` is displayed in the File Name field.
3. Click Finish.

Note - You can also remove files from the folder. In this case, you do not need the `main.cpp` file that was added by default when you created the project. To remove this file from the project, right-click on the file name and choose Remove From Project.

Creating Header Files for Your Project

You can create new header files and add them to your project.

1. Right-click the Header Files folder and choose New > C Header File.
2. On the Name and Location page, `newfile` is displayed in the File Name field.
3. Click Finish.

The `newfile.h` file is created on disk in the project directory and added to the Header Files folder.

Adding Existing Files to Your Project

You can add existing files to your project in two ways:

- Right-click the Source Files folder and choose Add Existing Item. You can point to an existing file on disk using the Select Item dialog box and add the file to the project.
- Right-click the Source Files folder and choose Add Existing Items from Folders. Use the Add Folders dialog box to add folders that contain existing files.

Do not use the New menu item to add existing items. The Name and Location panel will tell you the file already exists.

Setting Properties and Configurations

The following sections describe how to set and manage your properties for both the current project and individual files.

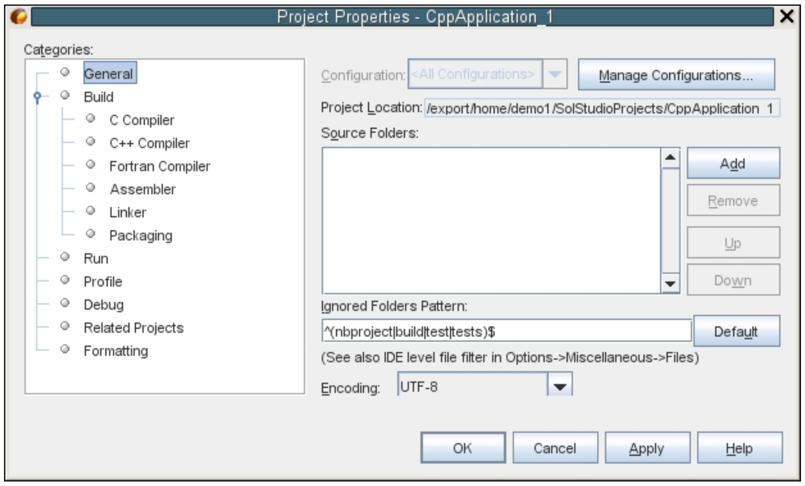
- "Setting Project Properties" on page 14
- "Managing Configurations" on page 14
- "Setting Source File Properties" on page 15

Setting Project Properties

When the project is created, it has two configurations, Debug and Release. A configuration is a collection of settings used for the project, which enables you to easily switch many property settings at once. The Debug configuration builds a version of your application that includes debug information. The Release configuration builds an optimized version.

The Project Properties dialog box contains build and configuration information for your project. To open the Project Properties dialog box:

- Right-click the project node of the Application project and choose Properties.



Configuration: <All Configurations>
Project Location: /export/home/demo1/SolStudioProjects/CppApplication_1
Source Folders:
Ignored Folders Pattern: ^(nbproject|build|test|tests)$
(See also IDE level file filter in Options->Miscellaneous->Files)
Encoding: UTF-8

You can modify the compiler settings and other configuration settings in the Project Properties dialog box by selecting a node in the left panel and modifying the properties in the right panel. Select some of the nodes and property values and notice the properties you can set. When you set General properties, you are setting them in all configurations of the project. When you set Build, Run, or Debug properties, you are setting properties in the currently selected configuration.

Managing Configurations

When a project is created, it has a Debug configuration and a Release configuration. A configuration is a collection of settings used for the project. You can easily switch many settings at once when you select a configuration. The Debug configuration builds a version of your application that includes debug information. The Release configuration builds an optimized version.

Properties changed in the Project Properties dialog box are stored in the makefile for the current configuration. You can edit the default configurations or create new ones.

To create a new configuration:

1. Click the Manage Configurations button in the Project Properties dialog box.
2. In the Configurations dialog box, select the configuration which most closely matches your desired configuration. In this case, select the Release configuration and click the Copy button. Then click Rename.
3. In the Rename dialog box, rename the configuration to PerformanceRelease. Click OK.
4. Click OK in the Configurations dialog box.
5. In the Project Properties dialog box, select the C Compiler node in the left panel. Note that the PerformanceRelease configuration is selected in the Configuration drop-down list.
6. In the property sheet in the right panel, change the Development Mode from Release to PerformanceRelease. Click OK.

You have created a new configuration that will compile the application with a different set of options.

Tip - You can also set the active configuration in the IDE's toolbar, or by right-clicking the project node and selecting Set Configuration.

Setting Source File Properties

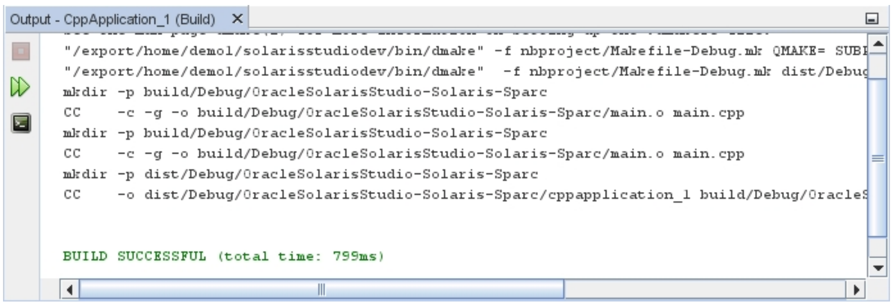
When you set the project properties for your project, the relevant properties apply to all files in the project. You can set some properties for a specific file.

1. Right-click the `newmain.c` source file and choose Properties.
2. Click the General node in the Categories panel and see that you can select a different compiler or other tool to build this file. You can also select a checkbox to exclude the file from the build of the currently selected project configuration.
3. Click the C compiler node and note that you can override the project compiler settings and other properties for this file.
4. Cancel the Project Properties dialog box.

Building Your Project

To build your project:

1. Right-click the project and choose Build. The project builds. The build output is shown in the Output window.



```
"/export/home/demol/solarisstudiodev/bin/dmake" -f nbproject/Makefile-Debug.mk QMAKE= SUBI
"/export/home/demol/solarisstudiodev/bin/dmake"  -f nbproject/Makefile-Debug.mk dist/Debug
mkdir -p build/Debug/OracleSolarisStudio-Solaris-Sparc
CC    -c -g -o build/Debug/OracleSolarisStudio-Solaris-Sparc/main.o main.cpp
mkdir -p build/Debug/OracleSolarisStudio-Solaris-Sparc
CC    -c -g -o build/Debug/OracleSolarisStudio-Solaris-Sparc/main.o main.cpp
mkdir -p dist/Debug/OracleSolarisStudio-Solaris-Sparc
CC    -o dist/Debug/OracleSolarisStudio-Solaris-Sparc/cppapplication_1 build/Debug/OracleS


BUILD SUCCESSFUL (total time: 799ms)
```

2. Switch the configuration from Debug to Performance Release in the configuration drop-down list in the main toolbar. Now the project will be built using the Performance Release configuration.
3. Right-click the project and choose Build. The project builds. The build output is shown in the Output window.

To build multiple configurations of the project at the same time, choose Run > Batch Build Main Project and select the configurations you want to build in the Batch Build dialog box.

You can build, clean, or both clean and build the project by right-clicking the project and choosing actions from the menu. The project also keeps object files and executables from different configurations separate, so you do not have to worry about mixing files from multiple configurations.

Compiling a Single File

To compile a single source file:

- Right-click on the `newmain.c` file and choose Compile File. Only this file is compiled.

♦ ♦ ♦ C H A P T E R 3

Running a Project

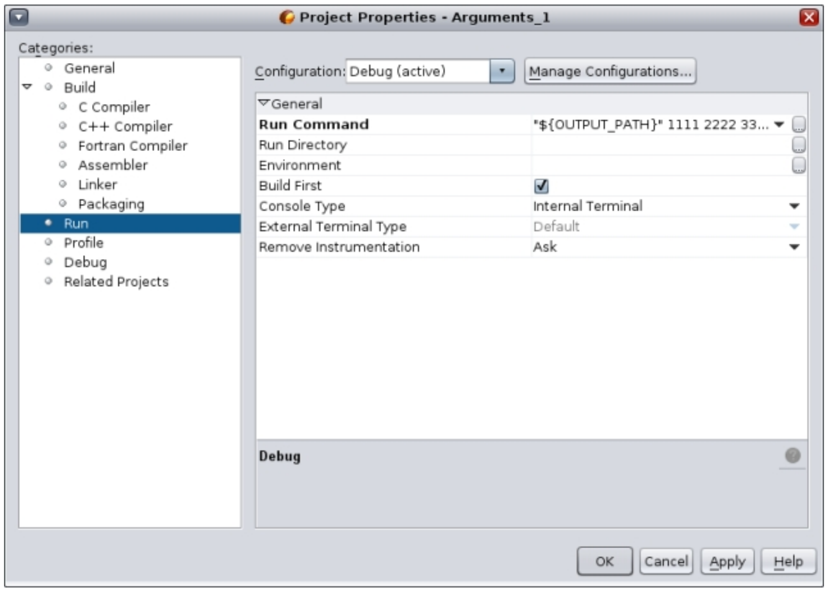
This chapter describes how to run a project in the IDE and contains the following sections:

Running a Project

The `Arguments` sample program prints command-line arguments. Before running the program, you will set some arguments in the current configuration. Then you will run the program.

To create the `Arguments_1` project, set some arguments, and run the project:

1. Choose File > New Project.
2. In the project wizard, expand the Samples category.
3. Select the C/C++ subcategory, then select the `Arguments` project. Click Next, then click Finish.
4. Right-click the `Arguments_1` project node and choose Build. The project builds.
5. Right-click the `Arguments_1` project node and choose Properties.
6. In the Project Properties dialog box, select the Run node.
7. In the Run Command text field, type `1111 2222 3333` after the output path. Click OK.



Configuration: Debug (active)

General

Run Command	"${OUTPUT_PATH}" 1111 2222 33...
Run Directory	
Environment	
Build First	☑
Console Type	Internal Terminal
External Terminal Type	Default
Remove Instrumentation	Ask

Debug

8. Choose Run > Run Main Project. The application builds and runs. Your arguments are displayed in an external window.

Launchers

You can create "launchers" so you can easily run your project from the project context menu with different arguments, for example, or launch it from a script. For more information about launchers, see “New Launchers Feature in IDE” in “What’s New in Oracle Solaris Studio 12.4 ”.

♦ ♦ ♦ CHAPTER 4

Creating a Project From Existing Sources

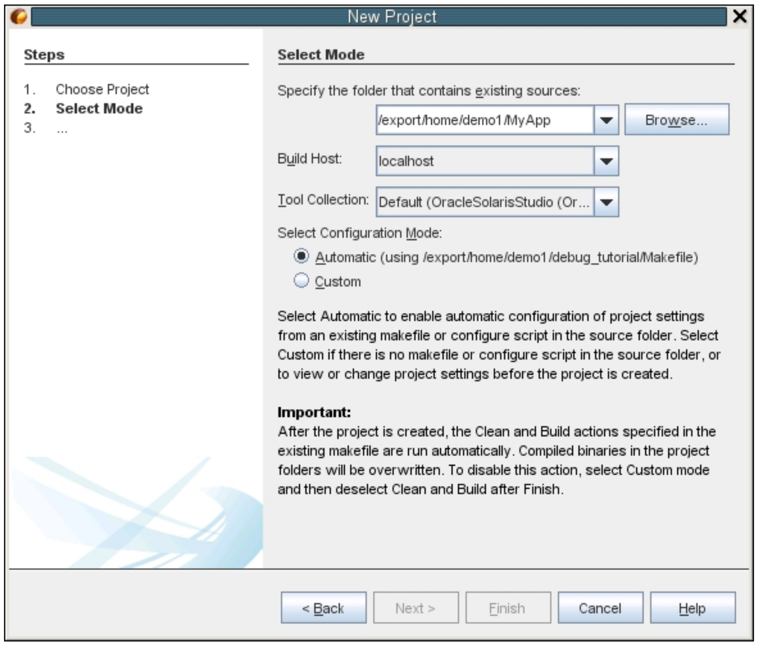
You can also create projects from existing sources that already include a makefile, or that build a makefile when you run `configure` scripts. This type of project is called an unmanaged project. For more information about projects and makefiles, see the C/C++/Fortran Projects and Makefiles page in the IDE help.

This chapter contains the following sections:

Creating a Project From Existing Sources

With a C/C++/Fortran Project From Existing Sources, the IDE relies on your existing makefile for instructions on how to compile and run your application.

1. Choose File > New Project.
2. Select the C/C++/Fortran category.
3. Select C/C++/Fortran Project From Existing Sources and click Next.
4. On the Select Mode page of the New Project wizard, click the Browse button. In the Select Project Folder dialog box, navigate to the directory where your source code is located. Click Select.



Specify the folder that contains existing sources: /export/home/demo1/MyApp

Build Host: localhost

Tool Collection: Default (OracleSolarisStudio (Or...

Select Configuration Mode:

- ● Automatic (using /export/home/demo1/debug_tutorial/Makefile)
- ○ Custom

Select Automatic to enable automatic configuration of project settings from an existing makefile or configure script in the source folder. Select Custom if there is no makefile or configure script in the source folder, or to view or change project settings before the project is created.

Important:
After the project is created, the Clean and Build actions specified in the existing makefile are run automatically. Compiled binaries in the project folders will be overwritten. To disable this action, select Custom mode and then deselect Clean and Build after Finish.

5. Use the default configuration mode, Automatic. Click Finish.
6. The project is created and opened in the Projects window, and the IDE automatically runs the Clean and Build sections specified in the existing Makefile. The project is also automatically configured for code assistance.

You have created a project that is a thin wrapper around existing code.

Building and Rebuilding Your Project

To build the project:

- Right-click the project node of the project and choose Build.

To rebuild the project:

- Right-click the project node of the project and choose Clean and Build.

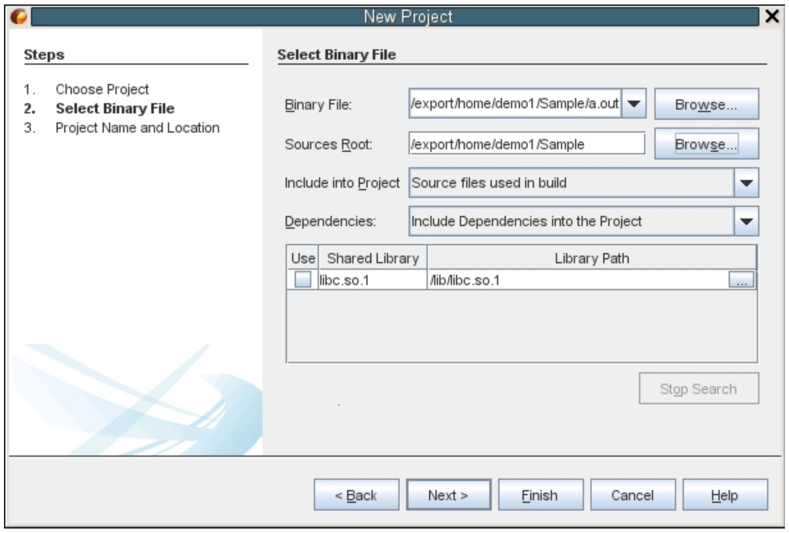
Creating a Project From a Binary File

With a C/C++/Fortran project from a binary file, you can create a project from an existing binary file in multiple ways.

Creating a Project from a Binary File With New Project Wizard

1. Choose File > New Project.
2. Select the C/C++/Fortran category.
3. Select C/C++/Fortran Project from Binary File and click Next.
4. On the Select Binary File page of the New Project wizard, click the Browse button. In the Select Binary File dialog box, navigate to the binary file from which you want to create a project.

   The root directory for the source files from which the binary was built is filled in automatically. By default, only the source files from which the binary was built are included in the project. By default, dependencies are included in the project. The shared libraries required by the project are automatically listed.



Binary File: /export/home/demo1/Sample/a.out

Sources Root: /export/home/demo1/Sample

Include into Project: Source files used in build

Dependencies: Include Dependencies into the Project

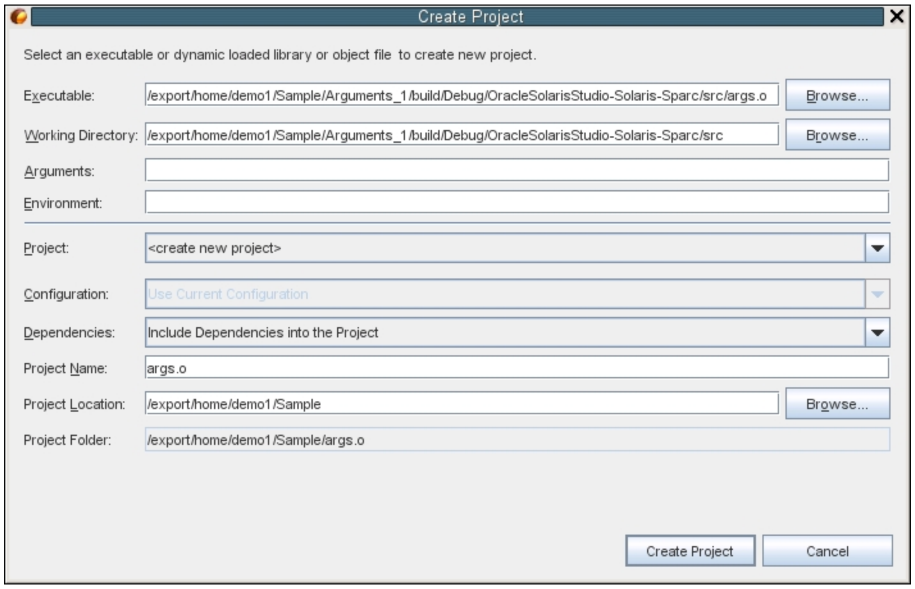
Use	Shared Library	Library Path
☐	libc.so.1	/lib/libc.so.1

5. Click Next.
6. On the Project Name and Location page, you can choose the name and location of the project. Click Finish.

Creating a Project from Binary File in Files Window

1. In the Files window, go to the Arguments_1 project and expand the build node all the way down to `args.o`.

2. Do one of the following to bring up the Create Project wizard:
   - Right-click the `args.o` file and click Create Project...
   - Drag the `args.o` file to the source editor.



Create Project

Select an executable or dynamic loaded library or object file to create new project.

Executable:	/export/home/demo1/Sample/Arguments_1/build/Debug/OracleSolarisStudio-Solaris-Sparc/src/args.o
Working Directory:	/export/home/demo1/Sample/Arguments_1/build/Debug/OracleSolarisStudio-Solaris-Sparc/src
Arguments:	
Environment:	
Project:	<create new project>
Configuration:	Use Current Configuration
Dependencies:	Include Dependencies into the Project
Project Name:	args.o
Project Location:	/export/home/demo1/Sample
Project Folder:	/export/home/demo1/Sample/args.o

   The Create Project wizard pops up, with Executable and Working Directory pre-filled out.

3. Input any arguments and environment variables you would like to use when running the project. In the Project drop-down menu, choose <create new project>.

   If you click the Create Project button on the bottom right corner, a new project called Args.o will be created in the projects tab.

4. Click cancel.

Code Assistance

Code Assistance is a set of IDE features that help you navigate and edit source code, specifically for unmanaged projects. For more information about editing and navigating, see Chapter 6, “Editing and Navigating Source Files”.

For an unmanaged project, you can specify how your code will be parsed to enable the code assistance features of the IDE. This section will discuss how to configure code assistance and code assistance cache sharing.

Configuring Code Assistance

When you create a project, you can specify the include files and macro definitions that will be used by the built-in parser to create the configuration. If the project has been built with debug information, you can have the built-in parser automatically search for the include files and macro definitions for each compiled source file.

To specify additional code assistance configuration information that will improve the accuracy of the code assistance features of the IDE for a project, use the Configure Code Assistance wizard. To start the wizard, right-click your project and choose Configure Code Assistance. To learn more about configuring Code Assistance and the Configure Code Assistance wizard, see the relevant help sections in the IDE.

Code Assistance Cache Sharing

When parsing C/C++ source code, the IDE stores parse results on a disk in the Code Assistance cache. When you open a project, the IDE examines the cache to see if it is up to date. If the cache is up to date, the IDE does not parse your project and just loads the required data for code navigation form the Code Assistance cache.

By default, the IDE creates one code assistance cache for all your projects in your user directory in the `$`*userdir*`/var/cahche` folder. The cache in your user directory cannot be copied or shared to another location.

However, if the Code Assistance cache is placed inside a project, it can be copied to another computer if the computer's operating system is identical to the operating system where the code was parsed and if the tool collection used by the project is available in the same location on the computer.

For more information about Code Assistance Cache sharing and to learn how to instruct the IDE to please the Code Assistance cache inside your project metadata, see "Code Assistance Cache Sharing" in "What's New in Oracle Solaris Studio 12.4 " and Relocating the Code Assistance Cache for Version Controlled Projects in the IDE help.

Project Properties Options for Code Assistance

The IDE now provides the following project properties to make it easier for you to use unmanaged projects in version control systems.

Transient macros	You can provide a list of macros (`-D` options) that are volatile. They depend on time, date, or specific environment. These macros environment variable values will not be stored with the project's public metadata.

User Environment Variables	You can provide a list of environment variables that the project uses to pass system-specific paths. These macros environment variable values will not be stored with the project's public metadata. For unmanaged projects, you can specify the list of environment variables to be used when storing project metadata. When the IDE stores the compiler options and an option value coincides with a variable value, a macros will be written instead.

Search File System for C/C++ Header Files

If you create an unmanaged project where the sources have not been built and does not contain any debugging information, the IDE might have trouble configuring code assistance. In this case, you can specify in the Configure Code Assistance wizard to use a special mode, Search file system for C/C++ header files. In this mode, the IDE tries to resolve failed include directives by searching the file system for headers. The wizard asks you to enter the path to search for headers. By default, the path is the project source root.

♦ ♦ ♦ CHAPTER 5

Creating an Oracle Database Project

This chapter describes how to create an Oracle Database project in the IDE and points to several references to learn more about Oracle Database projects and how to use them in the IDE. For more resources about Oracle Database Projects and the IDE, see "Oracle Database Projects" on page 93.

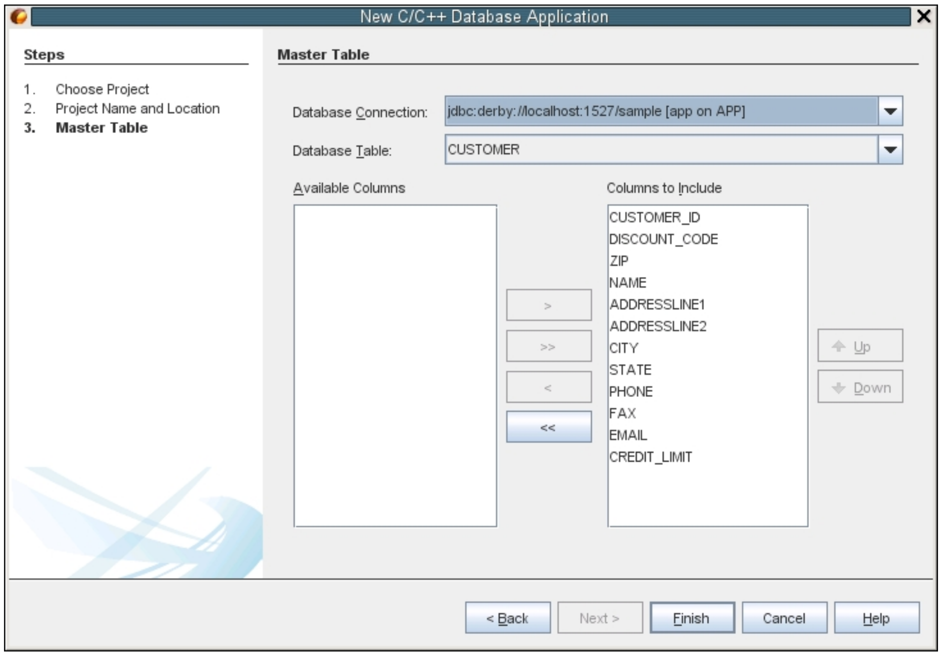
Creating an Oracle Database Project

You can create a project for an Oracle Database application. In order to do so, the Oracle Solaris Studio installation you are using must include the optional Oracle Instant Client component.

1. Choose File > New Project.
2. In the New Project dialog box, select the C/C++/Fortran category and the C/C++ Database Application project. Click Next.
3. On the Project Name and Location page, you can choose the name and location of the project. Click Next.
4. On the Master table page, select `jdbc:derby://localhost:1527/sample` from the Database Connection drop-down list. The IDE connects to the database.

   If there is no database listed under Database Connection, wait a few moments and try again. Derby is a database that is built into JDK 6 and later versions.

   If you would like to create a new database connection, see "Creating a New Database Connection" on page 26.
5. Select the master table for your project from the Database Table drop-down list.
6. Use the arrow keys between the Available Columns and Columns to Include lists to select the table columns you want to include in your project.



Database Connection: jdbc:derby://localhost:1527/sample [app on APP]

Database Table: CUSTOMER

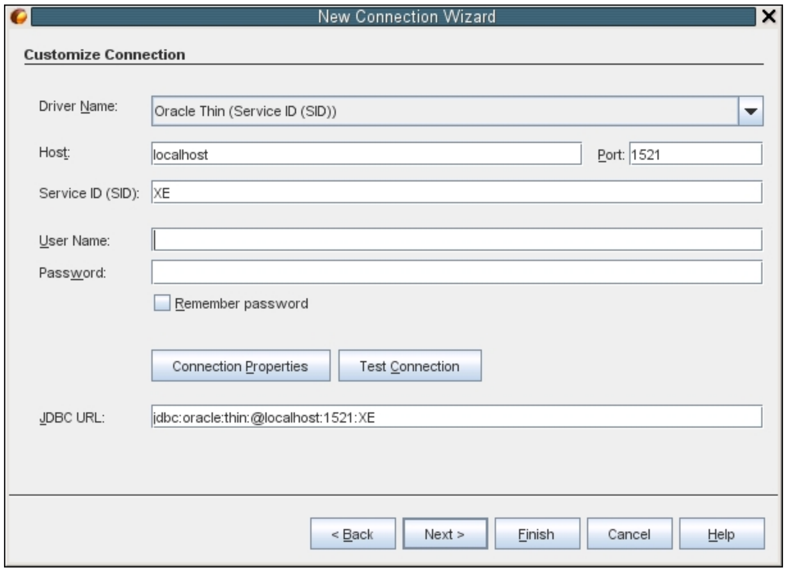
Columns to Include: CUSTOMER_ID, DISCOUNT_CODE, ZIP, NAME, ADDRESSLINE1, ADDRESSLINE2, CITY, STATE, PHONE, FAX, EMAIL, CREDIT_LIMIT

7. Click Finish.

Creating a New Database Connection

To set up a new connection to an Oracle database:

1. On the Master Table page, select New Database Connection to create a new database connection.


New Connection Wizard

Customize Connection

Driver Name: Oracle Thin (Service ID (SID))

Host: localhost Port: 1521

Service ID (SID): XE

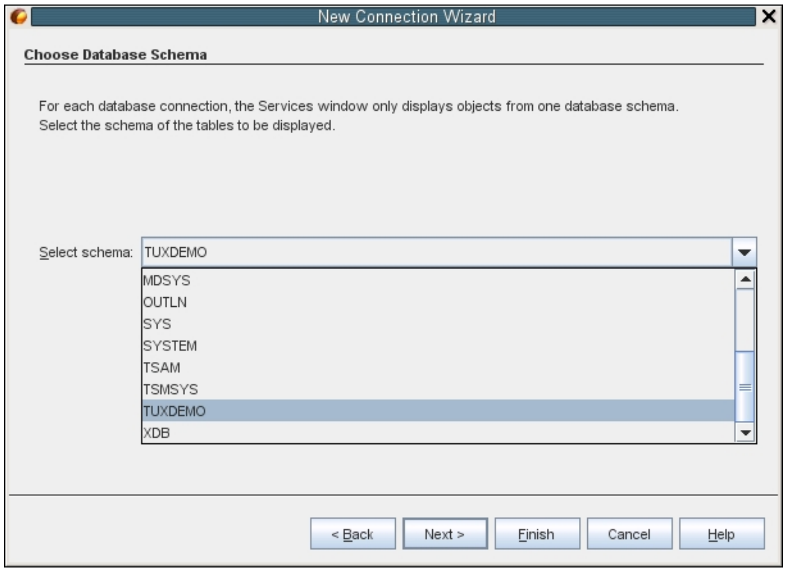
User Name:

Password:

Remember password

JDBC URL: jdbc:oracle:thin:@localhost:1521:XE

2. Enter the database address and your user name and password on the Customize Connection page of the New Connection wizard, then click Next.
3. On the Choose Database Schema page of the New Connection wizard, enter the database schema or select it from the list. Click Finish.


New Connection Wizard

Choose Database Schema

For each database connection, the Services window only displays objects from one database schema.
Select the schema of the tables to be displayed.

Select schema: TUXDEMO

MDSYS
OUTLN
SYS
SYSTEM
TSAM
TSMSYS
TUXDEMO
XDB

♦ ♦ ♦ CHAPTER 6

Editing and Navigating Source Files

The IDE enables you to edit and navigate through your source files in the editor. This chapter contains the following sections:

- "Editing Source Files" on page 29
- "Navigating Source Files" on page 41

Editing Source Files

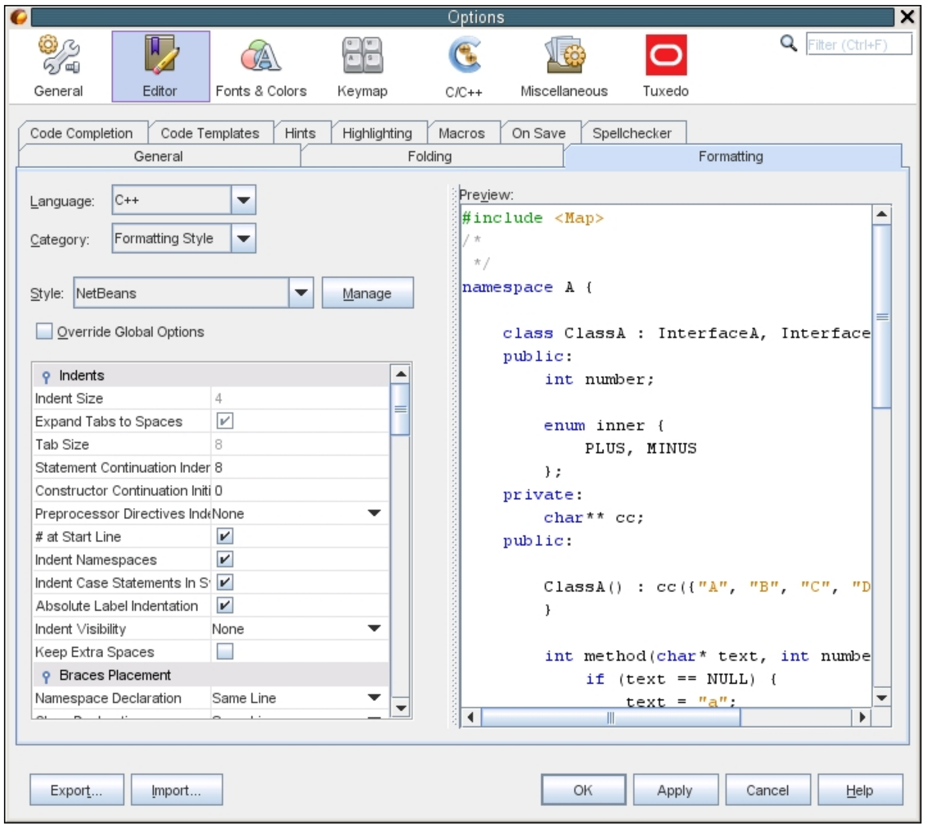
The Oracle Solaris Studio IDE provides advanced editing and code assistance features to help you in viewing and modifying your source code. To explore these features, use the `Quote` project:

1. Choose File > New Project.
2. In the project wizard, expand the Samples category and the C/C++ subcategory, then select the `Quote` project. Click Next, then click Finish.

Setting the Formatting Style

You can use the Options dialog box to configure default formatting style for your projects.

1. Choose Tools > Options.
2. Click Editor in the top pane of the dialog box.
3. Click the Formatting tab.
4. Select the language for which you want to set formatting style from the Language drop-down list.
5. Select the style you want to set from the Style drop-down list.



Language: C++

Category: Formatting Style

Style: NetBeans

☐ Override Global Options

Indents	
Indent Size	4
Expand Tabs to Spaces	✓
Tab Size	8
Statement Continuation Inder	8
Constructor Continuation Initi	0
Preprocessor Directives Ind	None
# at Start Line	✓
Indent Namespaces	✓
Indent Case Statements In S	✓
Absolute Label Indentation	✓
Indent Visibility	None
Keep Extra Spaces	☐
Braces Placement	
Namespace Declaration	Same Line

Preview:

```
#include <Map>
/*
 */
namespace A {

    class ClassA : InterfaceA, Interface
    public:
        int number;

        enum inner {
            PLUS, MINUS
        };
    private:
        char** cc;
    public:

        ClassA() : cc({"A", "B", "C", "D
        }

        int method(char* text, int numbe
            if (text == NULL) {
                text = "a";
```

6. Modify the style properties as desired.

Folding Blocks of Code in C and C++ Files

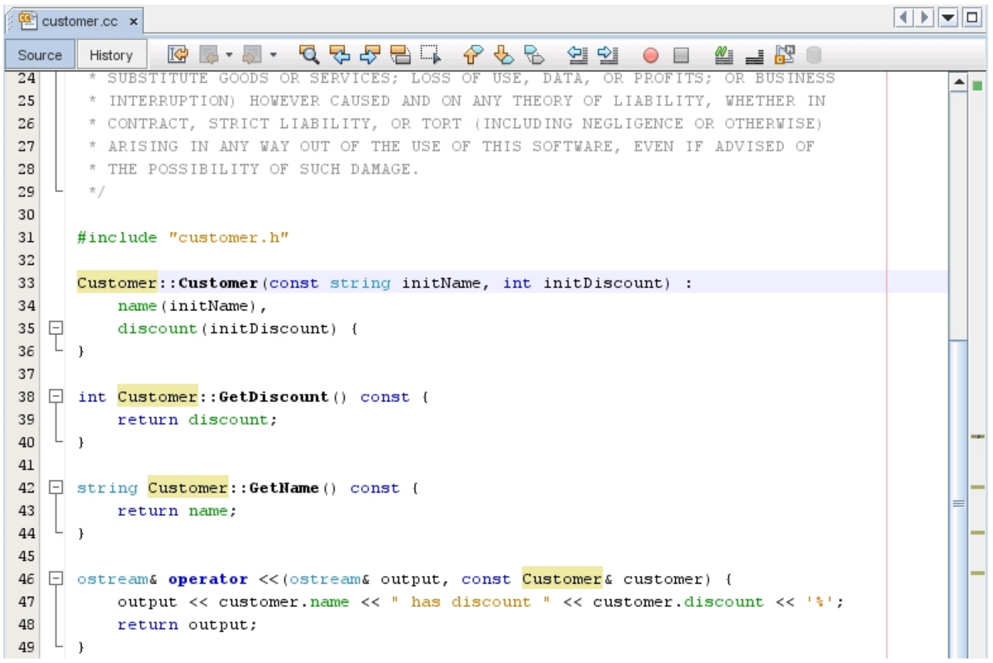
For some types of files, you can use the code folding feature to collapse blocks of code so that only the first line of the block appears in the Source Editor.

1. In the `Quote_1` application project, open the Source Files folder, then double-click the `cpu.cc` file to open it in the Source Editor.
2. Click the collapse icon (small box with minus sign) in the left margin to fold the code of one of the methods.
3. Mouse over the {...} symbol to the right of the folded block to display the code in the block.

Using Semantic Highlighting

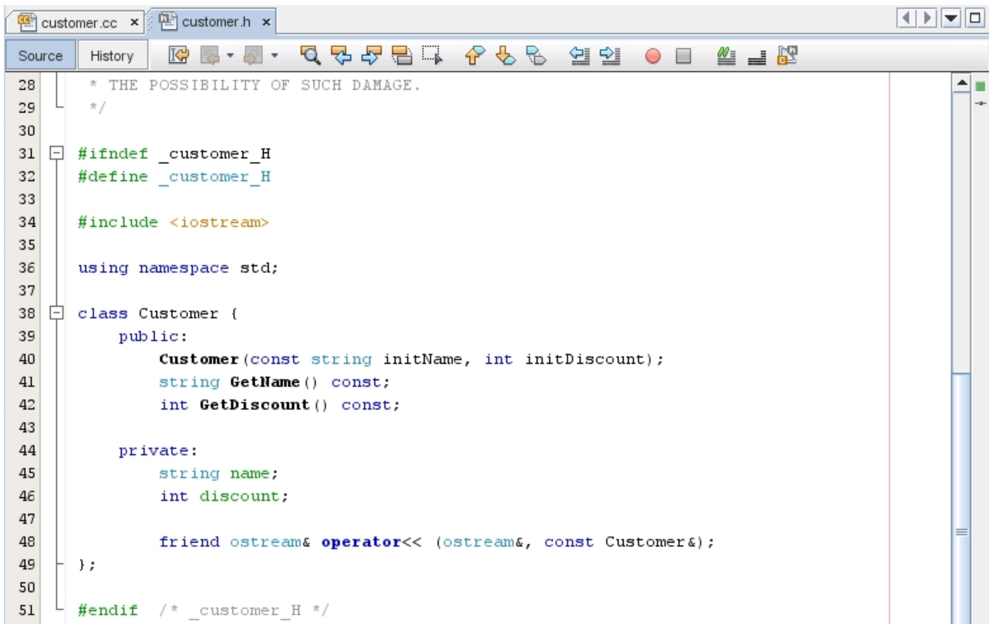
You can set an option so that when you click on a class, function, variable, or macro, all occurrences of that class, function, variable, or macro in the current file are highlighted.

1. Choose Tools > Options.
2. Click C/C++ in the top pane of the dialog box.
3. Click the Highlighting tab.
4. Make sure that all of the check boxes contain checkmarks.
5. Click OK.
6. In the `customer.cc` file of the `Quote_1` project, notice that the function names are highlighted in bold.
7. Click on an occurrence of the `Customer` class.
8. All of the occurrences of the `Customer` class in the file are highlighted with a yellow background.



```
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "customer.h"

Customer::Customer(const string initName, int initDiscount) :
    name(initName),
    discount(initDiscount) {
}

int Customer::GetDiscount() const {
    return discount;
}

string Customer::GetName() const {
    return name;
}

ostream& operator <<(ostream& output, const Customer& customer) {
    output << customer.name << " has discount " << customer.discount << '%';
    return output;
}
```

9. In the `customer.h` file, notice that class fields are highlighted in bold.

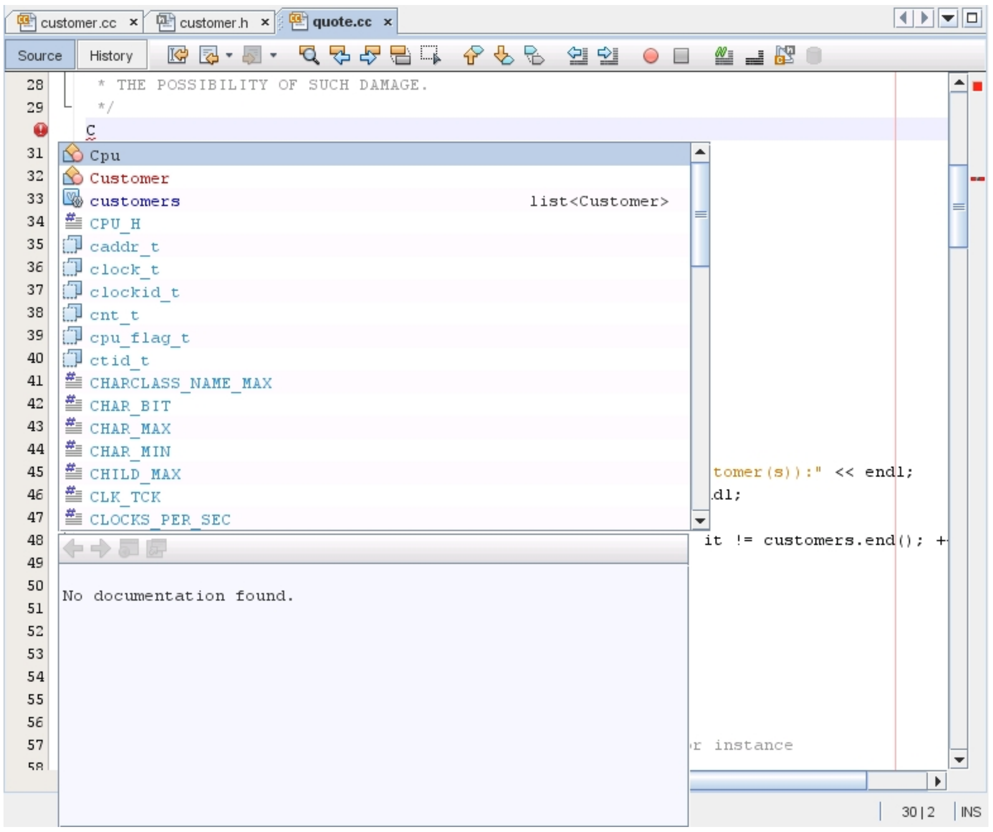


```
 * THE POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef _customer_H
#define _customer_H

#include <iostream>

using namespace std;

class Customer {
    public:
        Customer(const string initName, int initDiscount);
        string GetName() const;
        int GetDiscount() const;

    private:
        string name;
        int discount;

        friend ostream& operator<< (ostream&, const Customer&);
};

#endif  /* _customer_H */
```

Using Code Completion

The IDE has a dynamic C and C++ code completion feature that enables you to type one or more characters and then see a list of possible classes, methods, variables, and so on that can be used to complete the expression.

1. Open the `quote.cc` file in the `Quote_1` project.
2. On the first blank line of the `quote.cc` file, type a capital C and press Ctrl-Space. The code completion box displays a short list that includes the `Cpu` and `Customer` classes. A documentation window also opens and displays the message No documentation found because the project source code does not include documentation.
3. Expand the code completion list by pressing Ctrl-Space again.

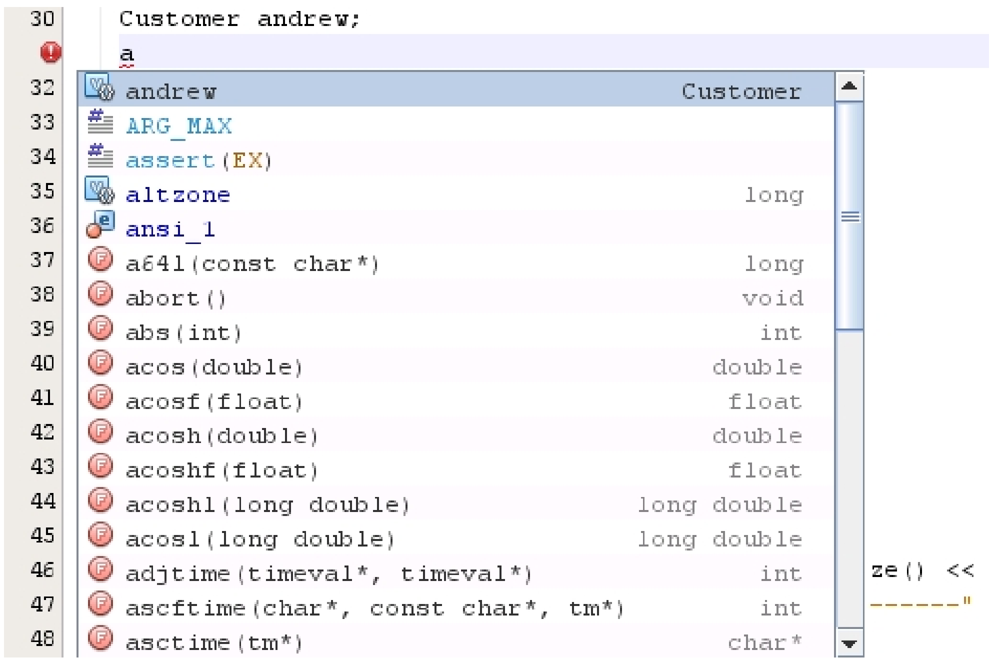


```
 * THE POSSIBILITY OF SUCH DAMAGE.
 */
C
```

```
Cpu
Customer
customers                                  list<Customer>
CPU_H
caddr_t
clock_t
clockid_t
cnt_t
cpu_flag_t
ctid_t
CHARCLASS_NAME_MAX
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHILD_MAX
CLK_TCK
CLOCKS_PER_SEC
```

```
No documentation found.
```

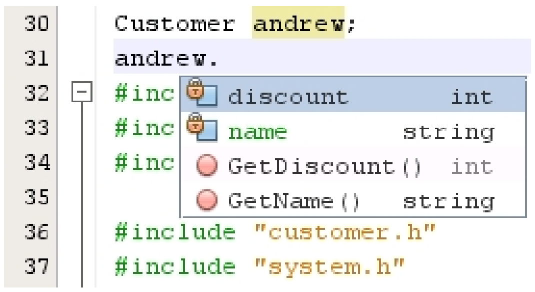
4. Select a standard library function such as `calloc()` from the list and the documentation window displays the man page for the function if the man page is accessible to the IDE.
5. Select the `Customer` class and press Enter.
6. Complete the new instance of the `Customer` class by typing `andrew;`. On the next line, type the letter `a` and press Ctrl-Space. The code completion box displays `andrew`. If you press Ctrl-Space again, the code completion box displays a list of choices starting with the letter a, such as method arguments, class fields, and global names, that are accessible from the current context.



```
Customer andrew;
a
```

```
andrew                                   Customer
ARG_MAX
assert(EX)
altzone                                  long
ansi_1
a64l(const char*)                        long
abort()                                  void
abs(int)                                 int
acos(double)                             double
acosf(float)                             float
acosh(double)                            double
acoshf(float)                            float
acoshl(long double)                      long double
acosl(long double)                       long double
adjtime(timeval*, timeval*)              int
ascftime(char*, const char*, tm*)        int
asctime(tm*)                             char*
```

7. Double-click the `andrew` option to accept it and type a period after it. You are automatically provided with a list of the public methods and fields of the `Customer` class.



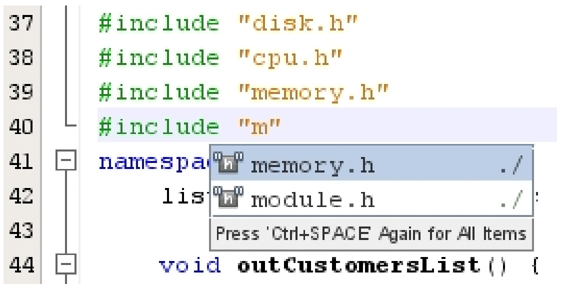
```
Customer andrew;
andrew.
#inc  discount          int
#inc  name              string
#inc  GetDiscount()     int
      GetName()         string
#include "customer.h"
#include "system.h"
```

8. Delete the code you have added.

Using Static Code Error Checking

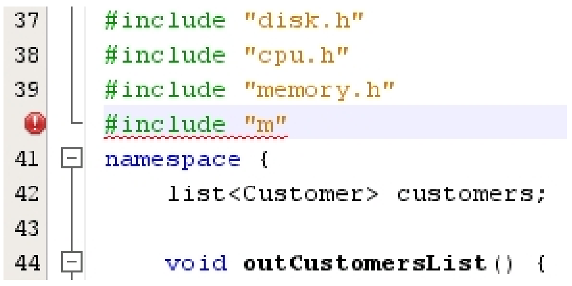
When you type code in a source or header file in the Source Editor, the editor performs static code error checking as you type and displays an error icon in the left margin when it detects an error.

1. In the `quote.cc` file of the `Quote_1` project, type `#include "m"` on line 40. A code completion box pops up, recommending two header files that begin with `m`.



```
#include "disk.h"
#include "cpu.h"
#include "memory.h"
#include "m"
namespace {
    list<Customer> customers;

    void outCustomersList() {
```

2. Click in the source editor away from your added code. Notice the error icon that appears in the margin.



```
#include "disk.h"
#include "cpu.h"
#include "memory.h"
#include "m"
namespace {
    list<Customer> customers;

    void outCustomersList() {
```

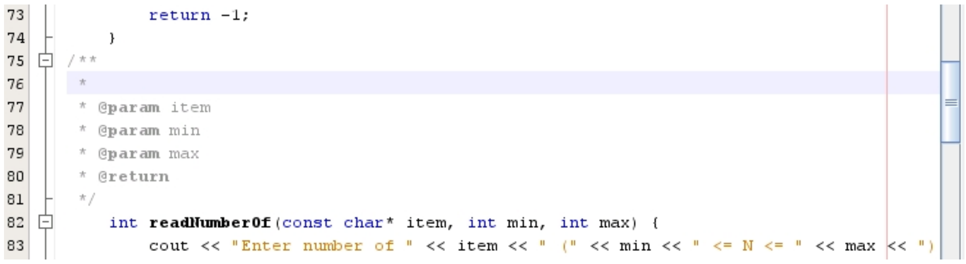
3. Backspace over the second quotation mark and complete the statement by typing `odule.h"`, and notice that error icon disappears as soon as the statement references an existing header file.
4. Delete the statement you have added.

For more information about how to choose which errors to see or to disable static code error checking, see the relevant help page in the IDE.

Adding Source Code Documentation

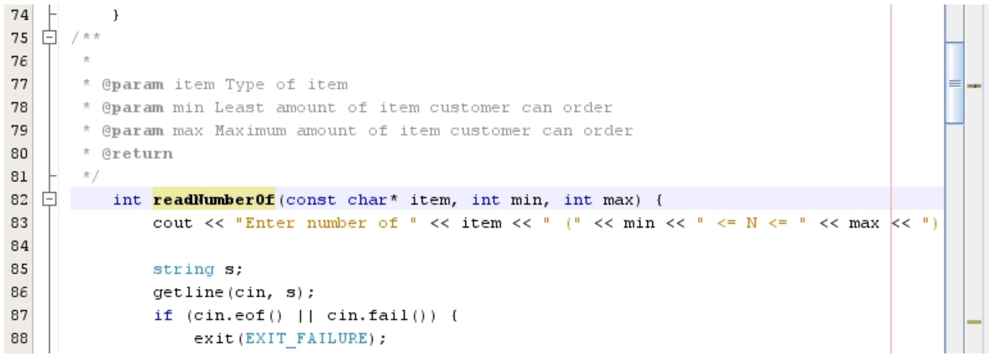
You can add comments to your code to generate documentation for your functions, classes, and methods. The IDE recognizes comments that use Doxygen syntax and automatically generates documentation. It can also automatically generate a comment block to document the function below the comment.

1. In the `quote.cc` file, place your cursor on the line above the line `int readNumberOf(const char* item, int min, int max) {`.
2. Type a forward slash and two asterisks (/**) and press Enter. The editor inserts a Doxygen-formatted comment for the `readNumberOf` class.



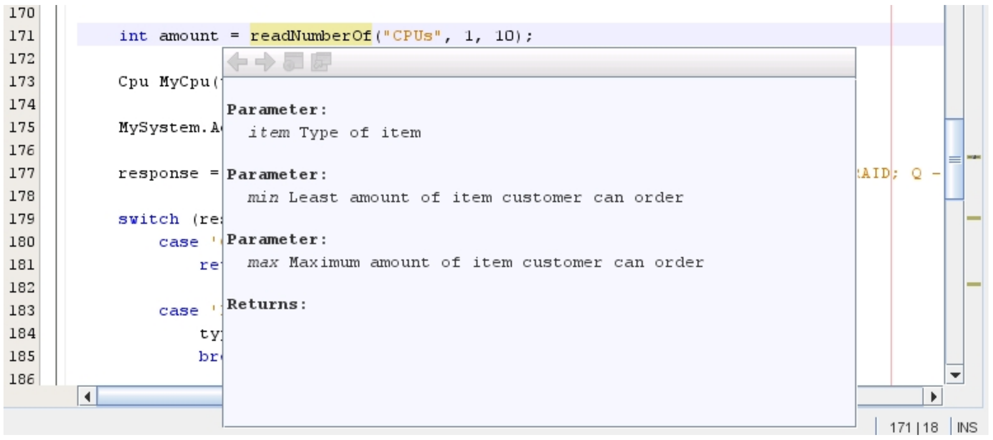
```
        return -1;
    }
/**
 *
 * @param item
 * @param min
 * @param max
 * @return
 */
    int readNumberOf(const char* item, int min, int max) {
        cout << "Enter number of " << item << " (" << min << " <= N <= " << max << ")
```

3. Add some descriptive text to each of the `@param` lines and save the file.
4. Click the `readNumberOf` class to highlight it in yellow, and click one of the occurrences marks on the right to jump to a location where the class is used.



```
    }
/**
 *
 * @param item Type of item
 * @param min Least amount of item customer can order
 * @param max Maximum amount of item customer can order
 * @return
 */
    int readNumberOf(const char* item, int min, int max) {
        cout << "Enter number of " << item << " (" << min << " <= N <= " << max << ")

        string s;
        getline(cin, s);
        if (cin.eof() || cin.fail()) {
            exit(EXIT_FAILURE);
```

5. Click the `readNumberOf` class in the line you jumped to, and press Ctrl-Shift-Space to show the documentation you just added for the parameters.



```
    int amount = readNumberOf("CPUs", 1, 10);

    Cpu MyCpu(

    MySystem.A

    response =                                                    AID; Q -

    switch (re
        case '
            re

        case '
            ty
            br
```

```
Parameter:
  item Type of item

Parameter:
  min Least amount of item customer can order

Parameter:
  max Maximum amount of item customer can order

Returns:
```

171 | 18 INS

6. Click anywhere else in the file to close the documentation window, and click on the `readNumberOf` class again.
7. Choose Source > Show documentation to open the documentation window for the class again.

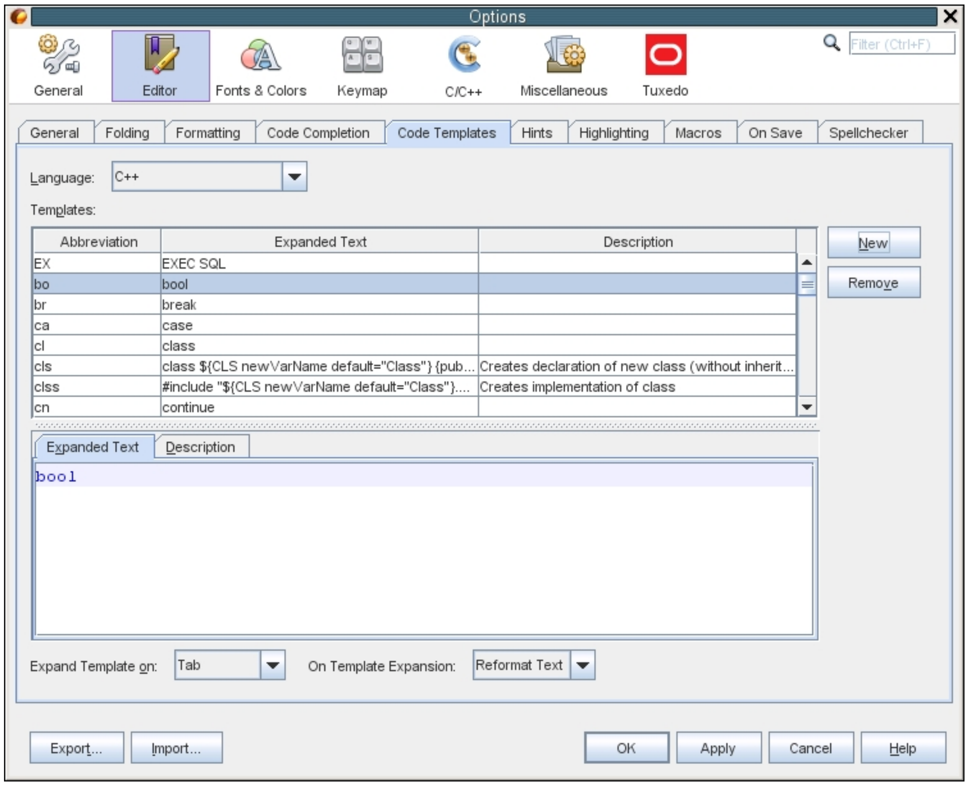
Using Code Templates

The Source Editor has a set of customizable code templates for common snippets of C, C++, and Fortran code. You can generate the full code snippet by typing its abbreviation and pressing the Tab key. For example, in the `quote.cc` file of the `Quote_1` project:

- Type `uns` followed by a tab and `uns` expands to `unsigned`.
- Type `iff` followed by a tab and `iff` expands to `if (exp) {}`.
- Type `ifs` followed by a tab and `ifs` expands to `if (exp) {} else {}`.
- Type `fori` followed by a tab and `fori` expands to `for (int i=0; i< size; i++) { Object size = array[i]; }`.

To see all of the available code templates, modify them, create your own code templates, or select a different key to expanded the templates:

1. Choose Tools > Options.
2. In the Options dialog box, select C/C++, and click the Code Templates tab.
3. Select a language from the Language drop-down list.
4. Add or remove items using the New or Remove buttons. You can also edit current templates in the Expanded Text tab or add a description of the template in the Description tab.



Language: C++

Templates:

Abbreviation	Expanded Text	Description
EX	EXEC SQL	
bo	bool	
br	break	
ca	case	
cl	class	
cls	class ${CLS newVarName default="Class"} {pub...	Creates declaration of new class (without inherit...
clss	#include "${CLS newVarName default="Class"}....	Creates implementation of class
cn	continue	

```
bool
```

Expand Template on: Tab

On Template Expansion: Reformat Text

Using Pair Completion

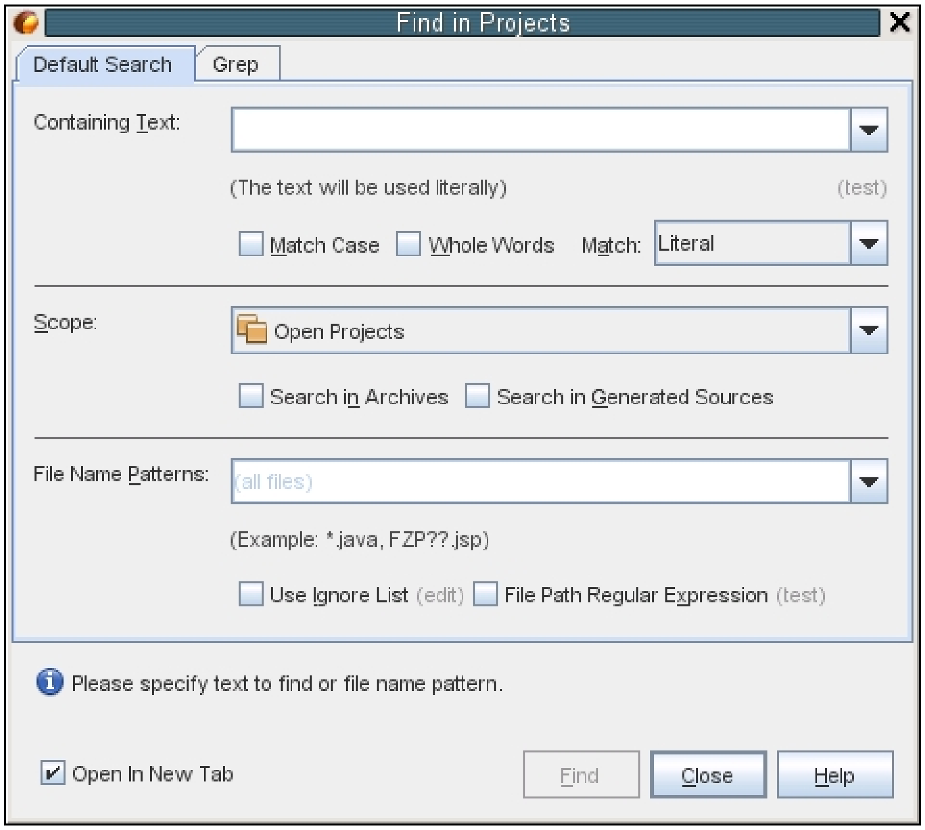
When you edit your C and C++ source file, the Source Editor does "smart" matching of pair characters such as brackets, parentheses, and quotation marks. When you type one of these characters, the Source Editor automatically inserts the closing character.

1. In the `Quote_1` project, place the cursor after the `{` on line 116 of the `module.cc` file and press Return to open a new line.
2. Type `enum state {` and press Return. The closing curly bracket and the semi-colon are added automatically and the cursor is placed between the brackets.
3. Type `invalid=0, success=1` to complete the enumeration.
4. On the line after the closing `};` of the enumeration, type `if (`. The closing parenthesis is added automatically and the cursor is placed between the parentheses.
5. Type `v==null`. Then type `i` and newline after the right parenthesis. The closing bracket is added automatically.

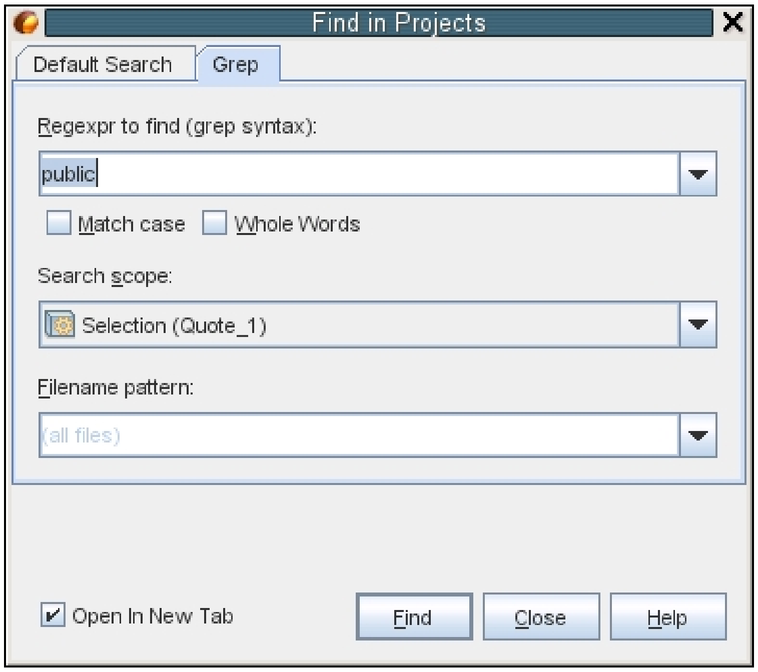
6. Delete the code you have added.

Finding Text in Project Files

You can use the Find in Projects dialog box to search projects for instances of specified text or a regular expression.

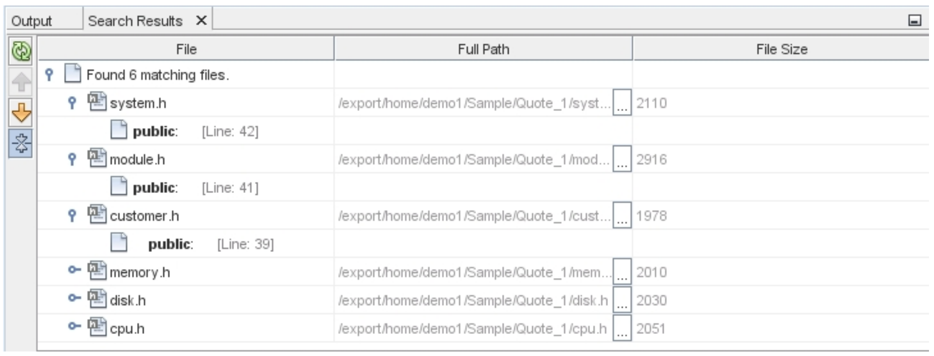
1. Open the Find in Projects Dialog box by right-clicking a project in the Projects window and choose Find or choose Edit > Find in Projects (Ctrl+Shift+F).
2. In the Find in the Projects dialog box, select the Default Search tab or the Grep tab. The Grep tab uses the `grep` utility, which provides a faster search, especially for remote projects.
3. 

4. In the Grep tab, type the text or regular expression that you want to search, specify the search scope and file name pattern, and select the check box for Open in New Tab so you can save multiple searches in separate tabs.



Regexpr to find (grep syntax):
public
Match case
Whole Words
Search scope:
Selection (Quote_1)
Filename pattern:
(all files)
Open In New Tab

5. Click Find.



File	Full Path	File Size
Found 6 matching files.		
system.h	/export/home/demo1/Sample/Quote_1/syst...	2110
public: [Line: 42]		
module.h	/export/home/demo1/Sample/Quote_1/mod...	2916
public: [Line: 41]		
customer.h	/export/home/demo1/Sample/Quote_1/cust...	1978
public: [Line: 39]		
memory.h	/export/home/demo1/Sample/Quote_1/mem...	2010
disk.h	/export/home/demo1/Sample/Quote_1/disk.h	2030
cpu.h	/export/home/demo1/Sample/Quote_1/cpu.h	2051

The Search Results tab lists the files in which the text or regular expression is found. Buttons in the left margin enable you to change your view of the search results.

6. Click the Expand/Collapse button to collage the list of files so only the files names are shown.

7. Double-click one of the items in the list and the IDE takes you to the corresponding location in the source editor.

Find and Replace

You can use the Find/Replace feature in the editor and it works entirely in the Find tool bar at the bottom of the editor window, instead of a separate dialog box for replacing. The Replace field and buttons are displayed in the tool bar under the Find field and buttons. Press Ctrl+F to activate the Find tool bar and Ctrl+H to activate the Replace feature.

Using Your Clipboard History

Once you find the text you were looking for, you might want to copy and use this text in other areas of your code.

You can view the last nine buffers of text that were copied to your desktop clipboard and select one to paste. With your cursor at the point where you want to insert text, press Ctrl+Shift+D to open a popup of the clipboard entries. Use the arrow keys to navigate through the clipboard buffers and see the full contents in the window below the list of buffers. To paste the contents of a buffer, type the number of the buffer, or press Enter when the buffer is selected. Note that this buffer contains content copied from any window on your desktop, not just the IDE.

Tip - You can copy a file path to your clipboard by hovering the mouse cursor over any file in the IDE and press Alt+Shift+L.

Navigating Source Files

The IDE provides advanced navigation features for viewing your source code. To explore these features, continue using the `Quote_1` project.

Window Management and Grouping

You can perform actions on groups of windows in addition to individual windows. Each window belongs to a group you can minimize, drag to a new location, float in a separate window, or dock back into the IDE window.

1. Minimize the Projects, Files, Classes, Classes, and Services windows in the top left by clicking the Minimize Window Group button on the right side of the group.

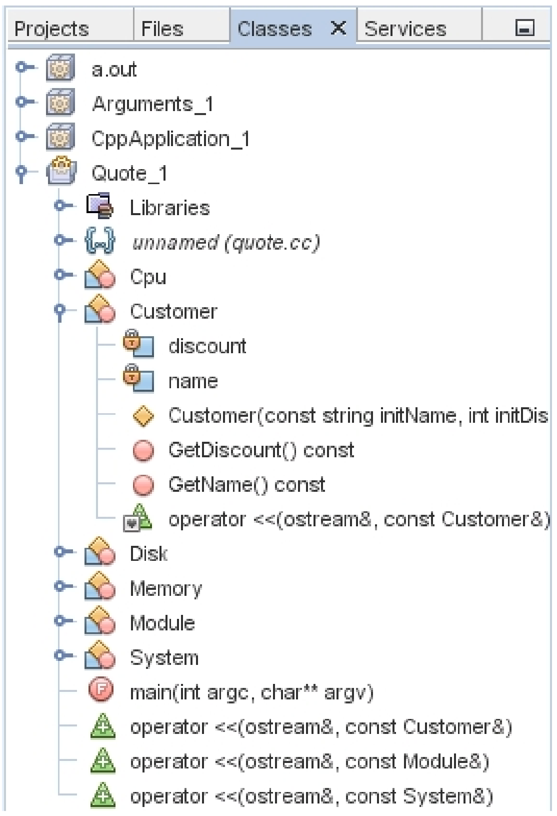
2. Right-click in the tab area of the group or choose Window > Configure Windows and select Maximize (Shift+Escape) to maximize the Window Group.

   Note that you can choose other options such as Float or Dock your window group.

Using the Classes Window

The Classes window lets you see all of the classes in your project, and the members and fields for each class.

1. Click the Classes tab to display the Classes window.
2. Expand the `Quote_1` node. All classes in the project are listed.
3. Expand the `Customer` class.



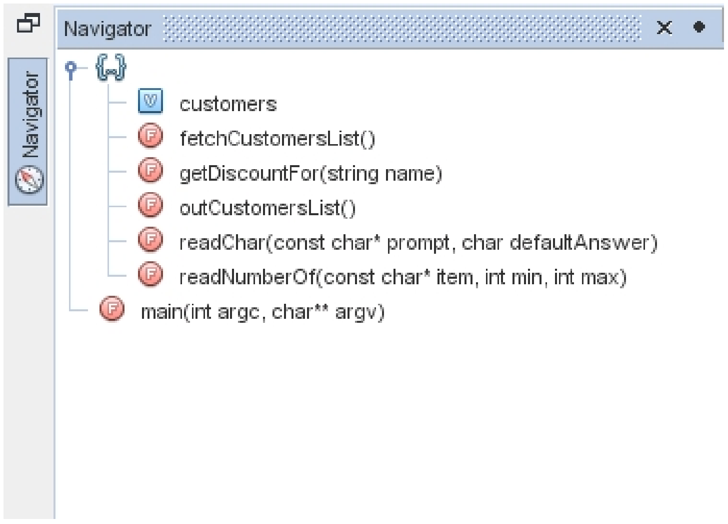
```
a.out
Arguments_1
CppApplication_1
Quote_1
  Libraries
  unnamed (quote.cc)
  Cpu
  Customer
    discount
    name
    Customer(const string initName, int initDis
    GetDiscount() const
    GetName() const
    operator <<(ostream&, const Customer&)
  Disk
  Memory
  Module
  System
  main(int argc, char** argv)
  operator <<(ostream&, const Customer&)
  operator <<(ostream&, const Module&)
  operator <<(ostream&, const System&)
```

4. Double-click on the `name` variable to open the `customer.h` header file.

Using the Navigator Window

The Navigator window provides a compact view of the file that is currently selected, and simplifies navigation between different parts of the file. If the Navigator window is not open, choose Window > Navigating > Navigator (Ctrl-7) to open it.

1. Click anywhere in the `quote.cc` file in the Editor. window.
2. A compact view of the file is displayed in the Navigator window. Click the node at the top of the window to expand the view.


Navigator
Navigator
{...}
customers
fetchCustomersList()
getDiscountFor(string name)
outCustomersList()
readChar(const char* prompt, char defaultAnswer)
readNumberOf(const char* item, int min, int max)
main(int argc, char** argv)

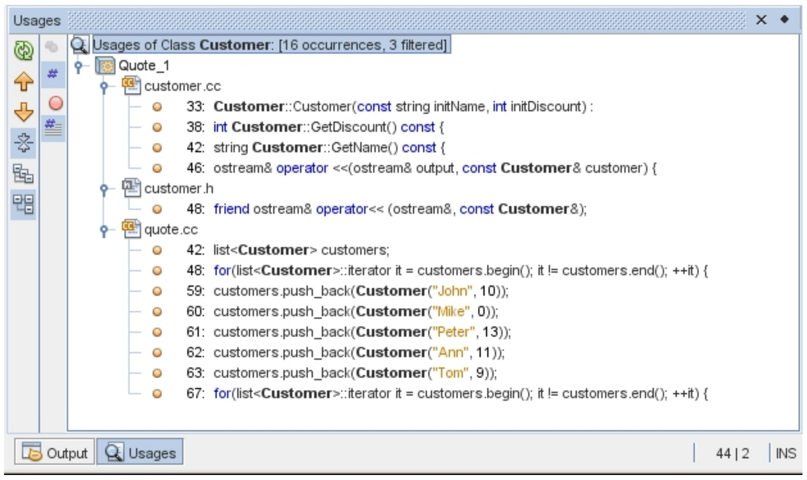
3. To navigate to an element of the file, double-click the element in the Navigator window and the cursor in the Editor window moves to that element.
4. Right-click in the Navigator window to see options for sorting the elements in the window, grouping the items, or filtering.
5. To see what the icons in the Navigator window represent, open the IDE online help by choosing Help > Help Contents. In the Help browser, click the Search tab and type `navigator icons` in the Find field.

Finding Class, Method, and Field Usages

You can use the Usages window to show you everywhere a class (structure), function, variable, macro, or file is used in your project's source code.

1. In the `customer.cc` file, right-click the `Customer` class on line 42, and choose Find Usages (Alt-F7).

2. In the Find Usages dialog box, click Find.
3. The Usages window opens and displays all of the usages of the `Customer` class in the source files of the project.



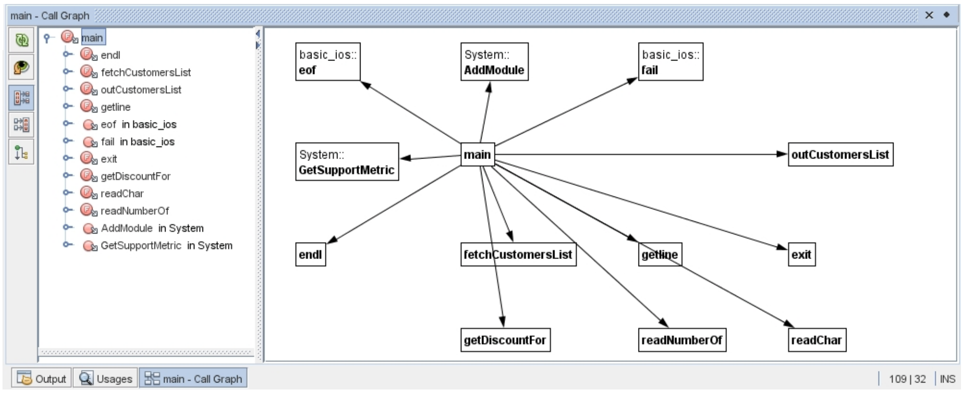
```
Usages of Class Customer: [16 occurrences, 3 filtered]
Quote_1
  customer.cc
    33: Customer::Customer(const string initName, int initDiscount) :
    38: int Customer::GetDiscount() const {
    42: string Customer::GetName() const {
    46: ostream& operator <<(ostream& output, const Customer& customer) {
  customer.h
    48: friend ostream& operator<< (ostream&, const Customer&);
  quote.cc
    42: list<Customer> customers;
    48: for(list<Customer>::iterator it = customers.begin(); it != customers.end(); ++it) {
    59: customers.push_back(Customer("John", 10));
    60: customers.push_back(Customer("Mike", 0));
    61: customers.push_back(Customer("Peter", 13));
    62: customers.push_back(Customer("Ann", 11));
    63: customers.push_back(Customer("Tom", 9));
    67: for(list<Customer>::iterator it = customers.begin(); it != customers.end(); ++it) {
```

Find Usages runs in the background so you can do other tasks while it is search a large number of files. The Usages window updates results as they are incrementally found. There is a progress indicator and an incrementing count of occurrences. You can stop the search at any point and your search results up to that point are saved. You can navigate between the search hits, change the view from logical to physical, and run Find Usages again with different settings.

Using the Call Graph

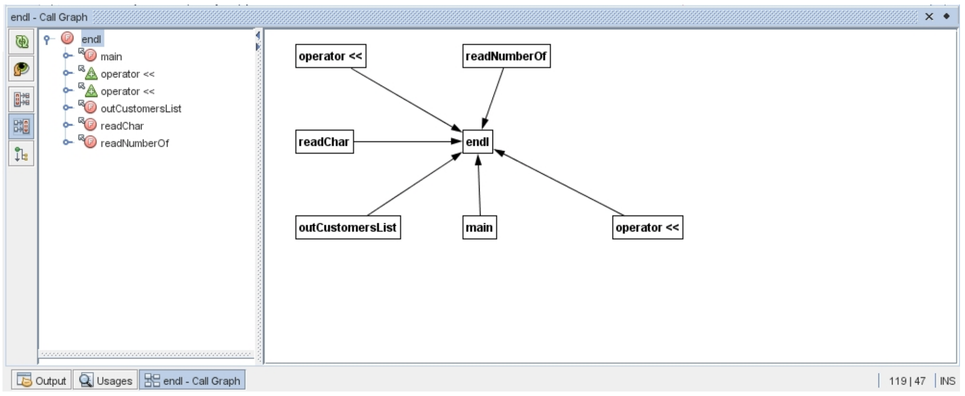
The Call Graph window displays two views of the calling relationships between functions in the classes. A tree view shows the functions called from a selected function, or the functions that call that function. A graphical view shows the calling relationships using arrows between the called and calling functions.

1. In the `quote.cc` file, right-click on the main function and choose Show Call Graph.
2. The Call Graph window opens and displays a tree view and a graphical view of all of the functions called from the `main` function.

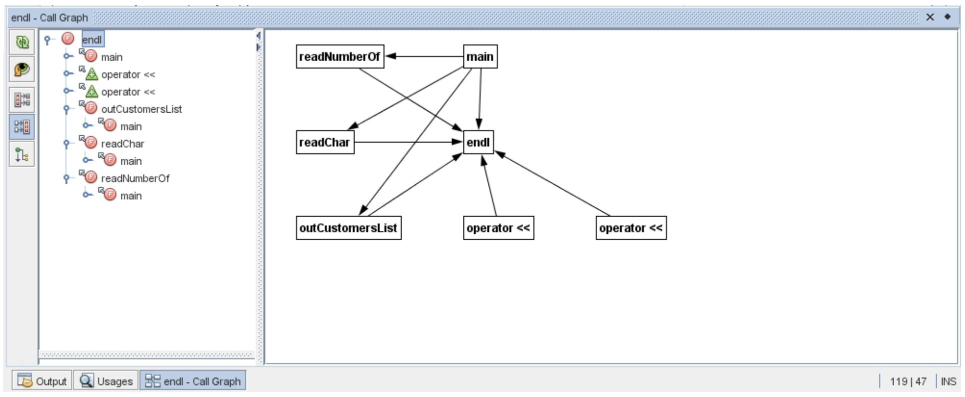


If you do not see all of the functions shown in the screen shot, click Who is Called From the Function button on the left side of the Call Graph window to show who is called from the main function.

3. Expand the `endl` node to display the functions called by that function. Notice that the graph is updated to add the functions called by `endl`.
4. Select the `endl` node and click the Bring Into Focus button on the left side of the window to focus on the `endl` function, then click the Who Calls This Function button to view all of the functions that call the `endl` function.



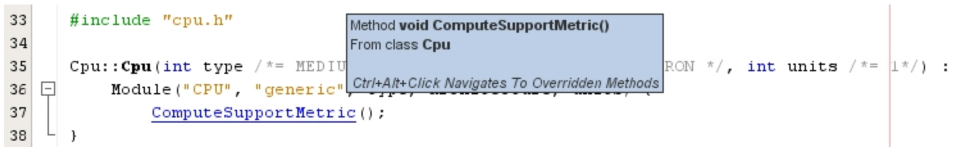
5. Expand some of the nodes in the tree to see more functions.



Using Hyperlinks

Hyperlink navigation lets you jump from the invocation of a class, method, variable, or constant to its declaration, and from its declaration to its definition. Hyperlinks also let you jump from a method that is overridden to the method that overrides it, and the reverse.

1. In the `cpu.cc` file of the `Quote_1` project, mouse over line 37 while pressing Ctrl. The `ComputeSupportMetric` function is highlighted and an annotation displays information about the function.

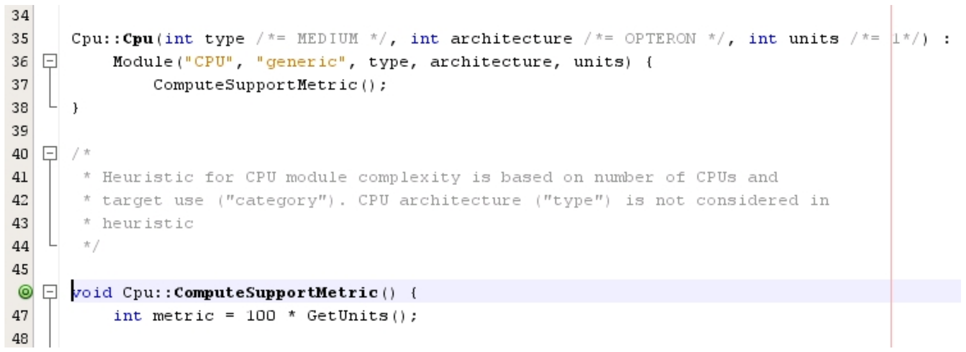


```
33  #include "cpu.h"
34
35  Cpu::Cpu(int type /*= MEDIU                              RON */, int units /*= 1*/) :
36      Module("CPU", "generic"
37          ComputeSupportMetric();
38  }
```

Method **void ComputeSupportMetric()**
From class **Cpu**

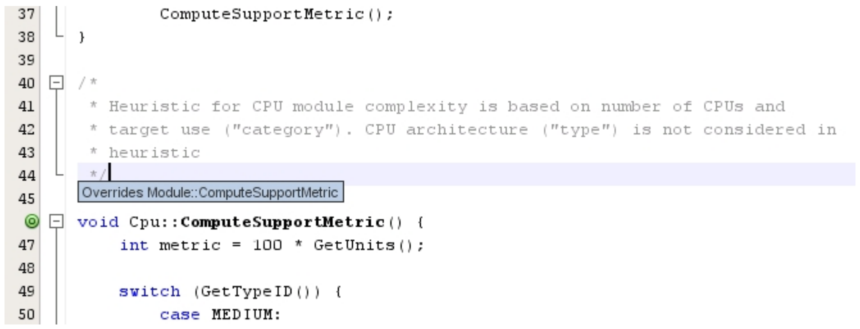
Ctrl+Alt+Click Navigates To Overridden Methods

2. Click the hyperlink and the editor jumps to the definition of the function.



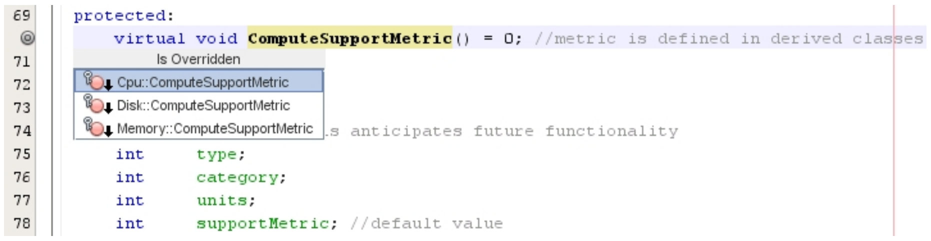
```
Cpu::Cpu(int type /*= MEDIUM */, int architecture /*= OPTERON */, int units /*= 1*/) :
    Module("CPU", "generic", type, architecture, units) {
        ComputeSupportMetric();
}

/*
 * Heuristic for CPU module complexity is based on number of CPUs and
 * target use ("category"). CPU architecture ("type") is not considered in
 * heuristic
 */

void Cpu::ComputeSupportMetric() {
    int metric = 100 * GetUnits();
```

3. Mouse over the definition while pressing Ctrl, and click the hyperlink. The editor jumps to the declaration of the function in the `cpu.h` header file.
4. Click the left arrow in the editor tool bar and the editor jumps back to the definition in `cpu.cc`.
5. Hover the mouse cursor over the green circle ◎ in the left margin and see the annotation that indicates that this method overrides another method.



```
        ComputeSupportMetric();
}

/*
 * Heuristic for CPU module complexity is based on number of CPUs and
 * target use ("category"). CPU architecture ("type") is not considered in
 * heuristic
 */
Overrides Module::ComputeSupportMetric

void Cpu::ComputeSupportMetric() {
    int metric = 100 * GetUnits();

    switch (GetTypeID()) {
        case MEDIUM:
```

6. Click the green circle to go to the overridden method and the editor jumps to the `module.h` header file, which shows a gray circle in the margin to indicate the method is overridden.
7. Click the gray circle and the editor displays a list of methods that override this method.



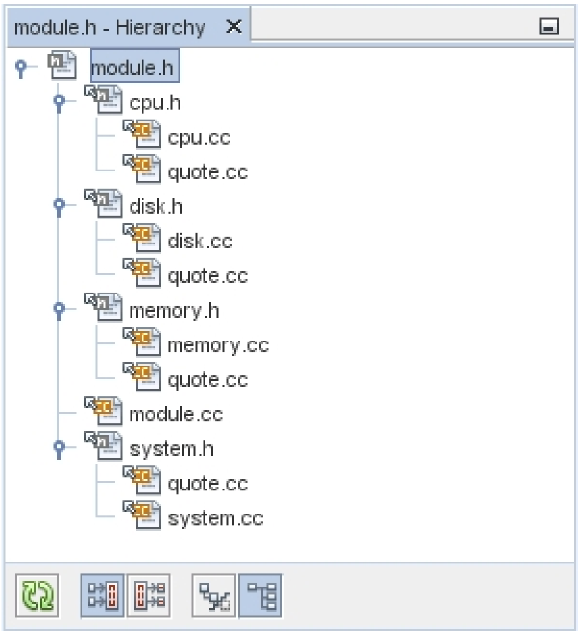
```
protected:
    virtual void ComputeSupportMetric() = 0; //metric is defined in derived classes
                   Is Overridden
    Cpu::ComputeSupportMetric
    Disk::ComputeSupportMetric
    Memory::ComputeSupportMetric   s anticipates future functionality
    int      type;
    int      category;
    int      units;
    int      supportMetric; //default value
```

8. Click the `Cpu::ComputerSupportMetric` item and the editor jumps back to the declaration of the method in the `cpu.h` header file.

Using the Include Hierarchy

The Include Hierarchy window lets you inspect all header and source files that are directly or indirectly included in a source file, or all source and header files that directly or indirectly include a header file.

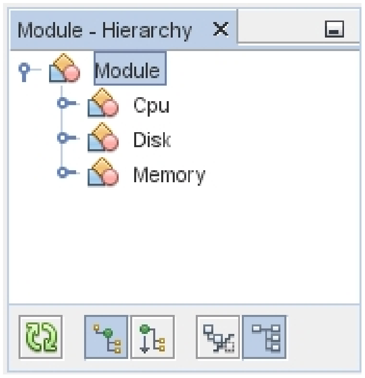
1. In the `Quote_1` project, open the `module.cc` file in the Source Editor.
2. Right-click on the `#include "module.h"` line in the file and choose Navigate > View Includes Hierarchy.
3. By default, the Hierarchy window displays a plain list of files that directly include the header file. Click the Show Tree View button at the bottom of the window . Click the Show Direct Includes Only button to display all files that include or are included. Expand the nodes in the tree view to see all of the source files that include the header file.



Using the Type Hierarchy

The Type Hierarchy window lets you inspect all subtypes or supertypes of a class.

1. In the `Quote_1` project, open the `module.h` file.
2. Right-click on the declaration of the `Module` class and choose Navigate > View Type Hierarchy.
3. The Hierarchy window displays all of the subtypes of the `Module` class.


Module - Hierarchy
Module
Cpu
Disk
Memory

♦ ♦ ♦ C H A P T E R 7

Debugging Your Application

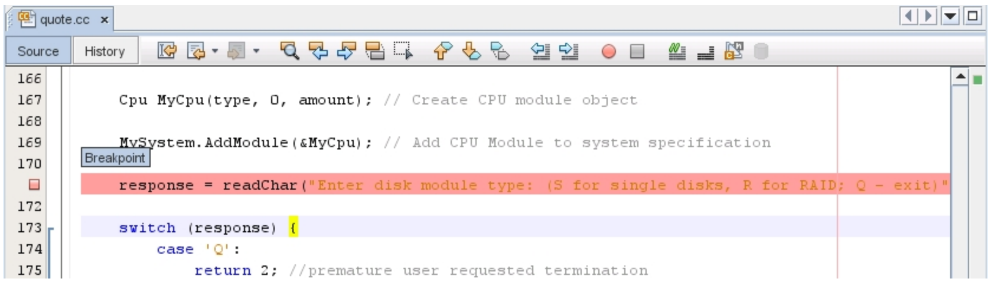
This chapter discusses how to create breakpoints for your project and how to use those breakpoints to debug your code. This chapter contains the following sections:

Creating Breakpoints

You can create and manipulate breakpoints in your code at any time. In order to run the debugger, you must have breakpoints set in your code so the debugger knows where to pause execution so you can examine the values of variables, step through lines of code, and fix errors.

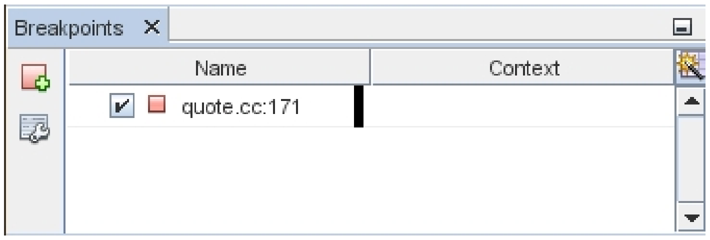
Creating and Removing a Line Breakpoint

1. In the `Quote_1` project, open the `quote.cc` file.
2. Set a line breakpoint by clicking in the left margin of the Editor window next to line 171 (`response = readChar("Enter disk module type: (S for single disks, R for RAID; Q - exit)", 'S');`). The line is highlighted in red to indicate that the breakpoint is set.



```
    Cpu MyCpu(type, 0, amount); // Create CPU module object

    MySystem.AddModule(&MyCpu); // Add CPU Module to system specification

    response = readChar("Enter disk module type: (S for single disks, R for RAID; Q - exit)"

    switch (response) {
        case 'Q':
            return 2; //premature user requested termination
```

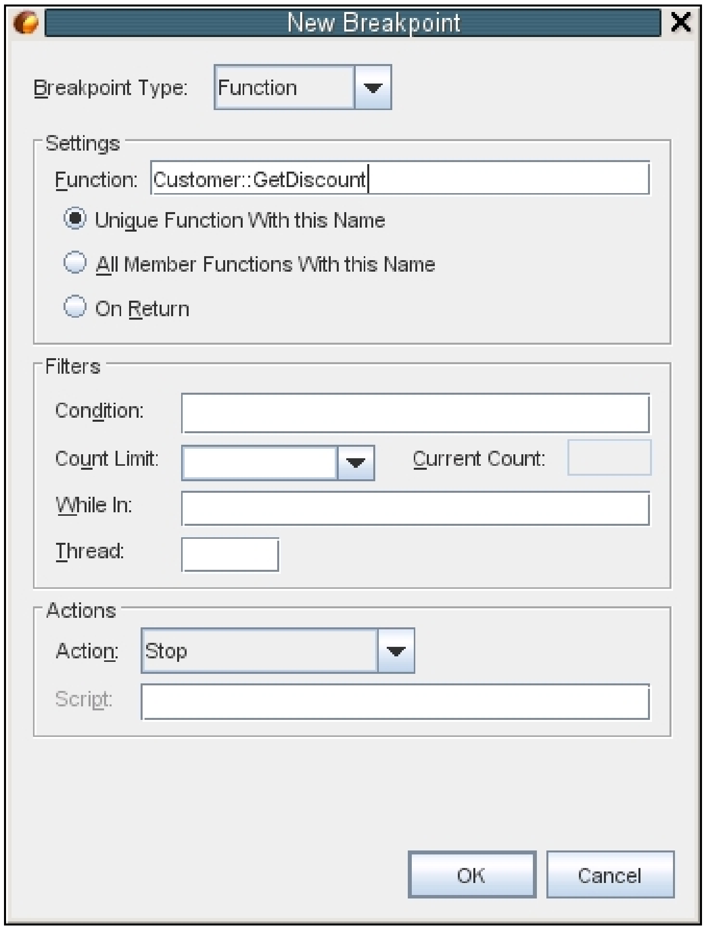
3. You could remove the breakpoint by clicking on the icon in the left margin.
4. Choose Window > Debugging > Breakpoints to open the Breakpoints window. Your line breakpoint is listed in the window.



Name	Context
✓ 🟥 quote.cc:171	

Creating a Function Breakpoint

1. Choose Debug > New Breakpoint (Ctrl+Shift+f8) to open the New Breakpoint dialog box.
2. In the Breakpoint Type drop-down list, set the type to Function.
3. Type the function name `Customer::GetDiscount` in the Function text field. Click OK.



Breakpoint Type: Function

Function: Customer::GetDiscount

- [✓ Unique Function With this Name]
- All Member Functions With this Name
- On Return

Condition:

Count Limit:

Current Count:

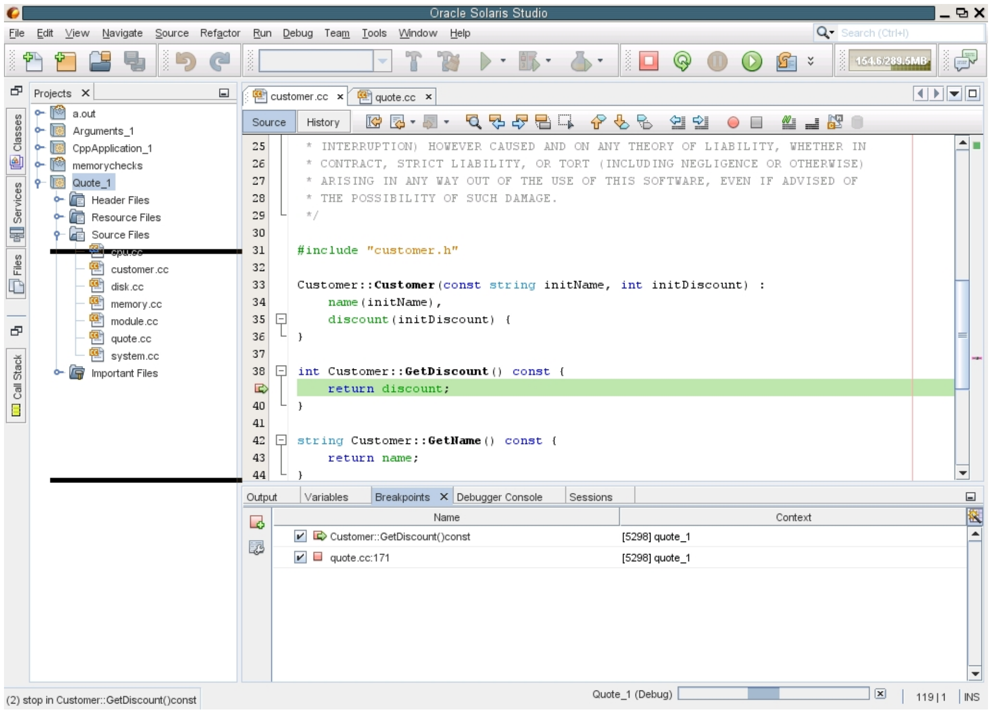
While In:

Thread:

Action: Stop

Script:

4. Your function breakpoint is set and is added to the list in the Breakpoints window.



```
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "customer.h"

Customer::Customer(const string initName, int initDiscount) :
    name(initName),
    discount(initDiscount) {
}

int Customer::GetDiscount() const {
    return discount;
}

string Customer::GetName() const {
    return name;
}
```

Name	Context
✓ Customer::GetDiscount()const	[5298] quote_1
✓ quote.cc:171	[5298] quote_1

(2) stop in Customer::GetDiscount()const

Grouping Breakpoints

You can group breakpoints using several different categories such as per file, per project, by type, by language, etc.

1. Choose Window > Debugging > Breakpoints window.
2. Click the Select Breakpoint Groups icon.
3. Select the breakpoint group.

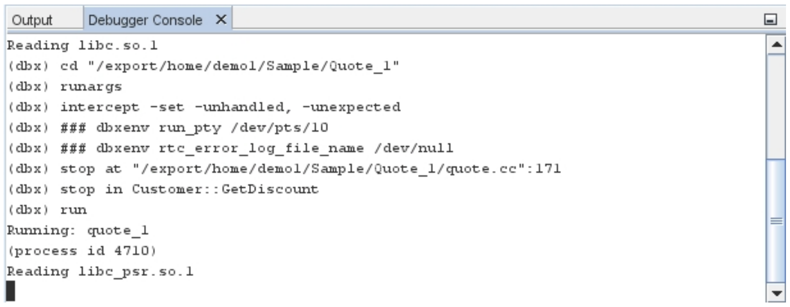
   The breakpoints are arranged according to your selection.

Debugging a Project

When you start a debugging session, the IDE starts the debugger in the project's associated tool collection (by default, the `dbx` debugger), then runs the application inside the debugger. The IDE automatically opens debugger windows and prints debugger output to the Debugger Console window.

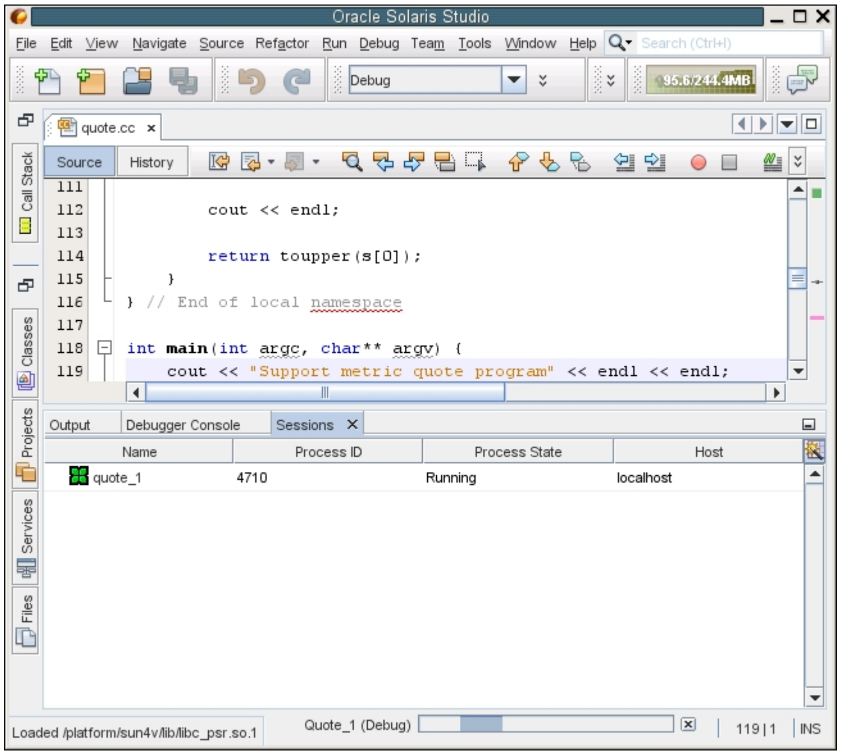
Starting a Debugging Session

1. Start a debugging session for the `Quote_1` project by right-clicking the project node and choosing Debug. The debugger starts and the application runs, and the Debugger Console windows open.



```
Reading libc.so.1
(dbx) cd "/export/home/demo1/Sample/Quote_1"
(dbx) runargs
(dbx) intercept -set -unhandled, -unexpected
(dbx) ### dbxenv run_pty /dev/pts/10
(dbx) ### dbxenv rtc_error_log_file_name /dev/null
(dbx) stop at "/export/home/demo1/Sample/Quote_1/quote.cc":171
(dbx) stop in Customer::GetDiscount
(dbx) run
Running: quote_1
(process id 4710)
Reading libc_psr.so.1
```

2. Open the Variables window by choosing Window > Debugging > Variables (Alt+Shift-1).
3. Open the Sessions window by choosing Window > Debugging > Sessions (Alt+Shift-6). The debugging session is shown in this window.



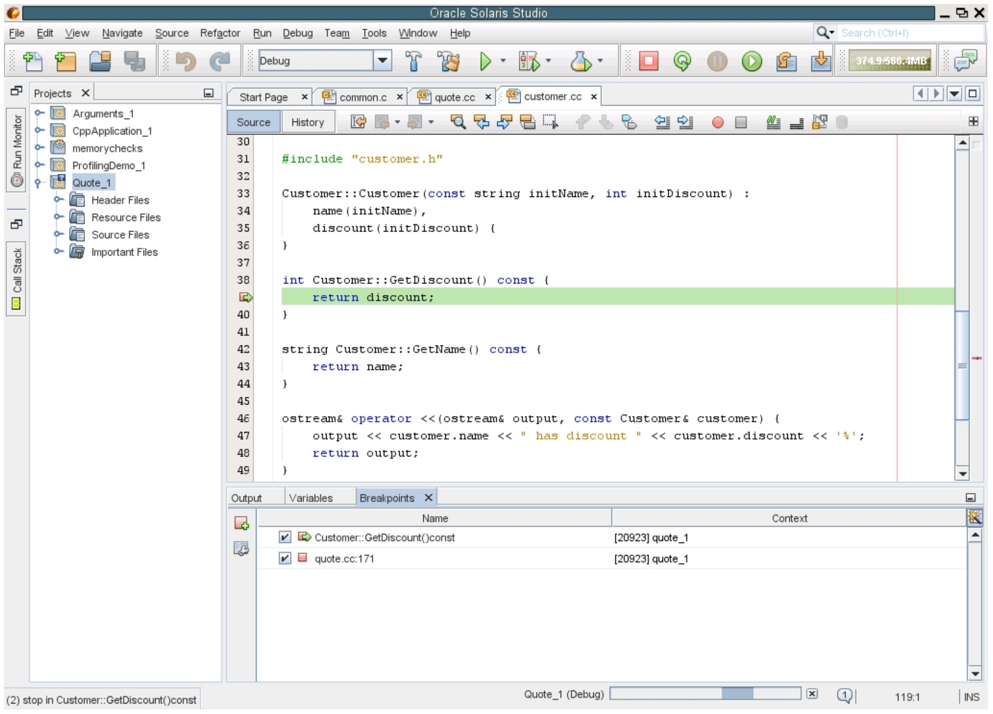
```
        cout << endl;

        return toupper(s[0]);
    }
} // End of local namespace

int main(int argc, char** argv) {
    cout << "Support metric quote program" << endl << endl;
```

Name	Process ID	Process State	Host
quote_1	4710	Running	localhost

Loaded /platform/sun4v/lib/libc_psr.so.1 Quote_1 (Debug) 119 | 1 INS

Inspecting the State of the Application

1. The `Quote_1` application prompts you for input in the Output window. Enter a customer name after the `Enter customer name:` prompt.
2. In the `customer.cc` file, the green program counter arrow appears on top of the breakpoint icon on the first line of the `GetDiscount` function.

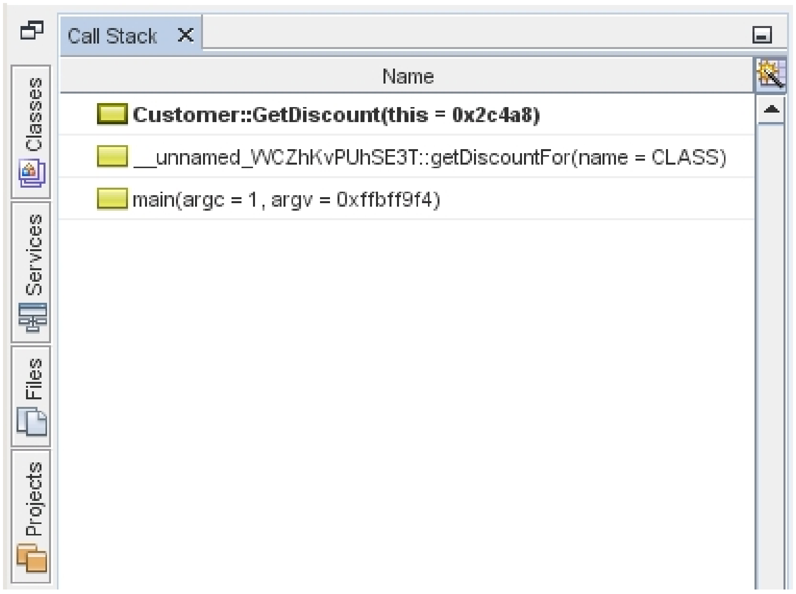


```
#include "customer.h"

Customer::Customer(const string initName, int initDiscount) :
    name(initName),
    discount(initDiscount) {
}

int Customer::GetDiscount() const {
    return discount;
}

string Customer::GetName() const {
    return name;
}

ostream& operator <<(ostream& output, const Customer& customer) {
    output << customer.name << " has discount " << customer.discount << '%';
    return output;
}
```

Name	Context
✓ Customer::GetDiscount()const	[20923] quote_1
✓ quote.cc:171	[20923] quote_1

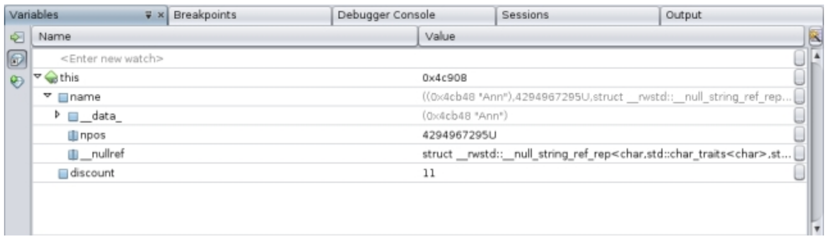
(2) stop in Customer::GetDiscount()const

3. Open the Call Stack window (Alt+Shift-3). The call stack shows three frames.



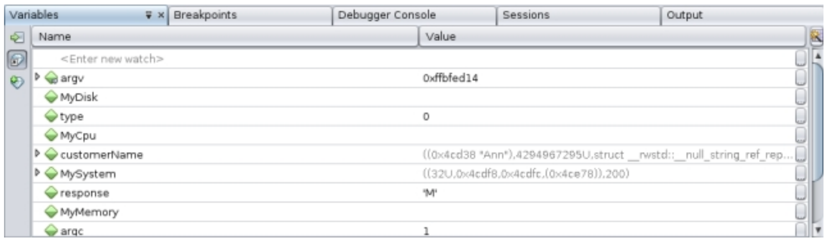
Name
Customer::GetDiscount(this = 0x2c4a8)
__unnamed_WCZhKvPUhSE3T::getDiscountFor(name = CLASS)
main(argc = 1, argv = 0xffbff9f4)

4. Click the Variables window and note that one variable is displayed. Click the nodes to expand the structure.



Name	Value
<Enter new watch>	
this	0x4c908
name	((0x4cb48 "Ann"),4294967295U,struct __rwstd::__null_string_ref_rep...
__data_	(0x4cb48 "Ann")
npos	4294967295U
__nullref	struct __rwstd::__null_string_ref_rep<char,std::char_traits<char>,st...
discount	11

5. Click the Continue button. The `GetDiscount` function is executed, printing the customer discount to the Output window. Then you are prompted for input.
6. Enter the input in response to the prompts. The program stops at the next breakpoint, the line breakpoint you set earlier. Click the Variables window and note the long list of local variables.



Name	Value
<Enter new watch>	
argv	0xffbfed14
MyDisk	
type	0
MyCpu	
customerName	((0x4cd38 "Ann"),4294967295U,struct __rwstd::__null_string_ref_rep...
MySystem	((32U,0x4cdf8,0x4cdfc,(0x4ce78)),200)
response	'M'
MyMemory	
argc	1

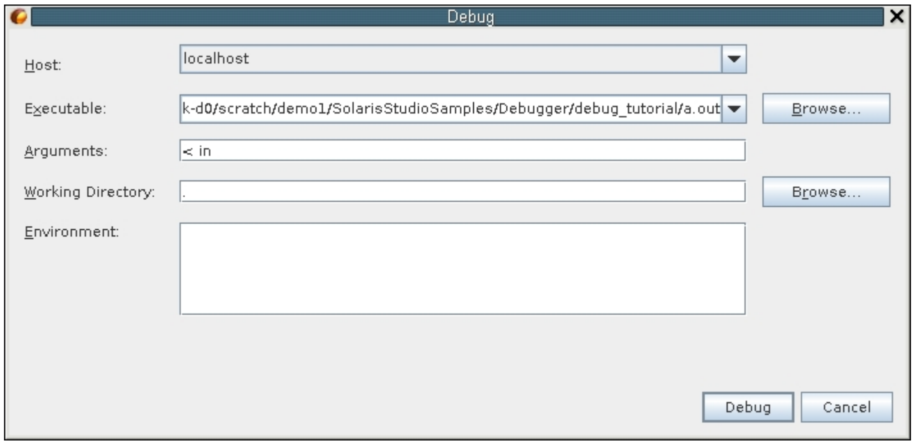
7. Look at the Call Stack window and note that there is only one frame in the stack.
8. Click Continue and continue entering input in response to the prompts in the Output window until the program is completed. When you enter the last input to the program, your debug session ends. To end the debug session before the execution of the program was complete, you could right-click the session in the Sessions window, and choose Finish.

Debugging an Executable File

You can debug an executable binary that is not in an IDE project. However, the executable should be located with the source code that was used to build it so the debugger can find debugging information. This is also called Projectless Debugging.

To debug an executable:

1. In the menu, choose Debug > Debug Executable.



Host: localhost

Executable: k-d0/scratch/demo1/SolarisStudioSamples/Debugger/debug_tutorial/a.out

Arguments: < in

Working Directory: .

Environment:

2. In the Debug dialog box, select which development host you want to use from the Host drop-down list.
3. Click the Browse button to find the executable path or type it directly into the Executable text field. If you clicked Browse to find the executable path, the Working Directory text field is auto generated.
4. Add any arguments to the Arguments text field.
5. If the Working Directory is not specified, either type the working directory path directly into the Working Directory text field or by click Browse and selecting the directory in the Working Directory Dialog box and then clicking Select.
6. Specify any environment variables by typing the settings in the Environment pane.
7. Click Debug.

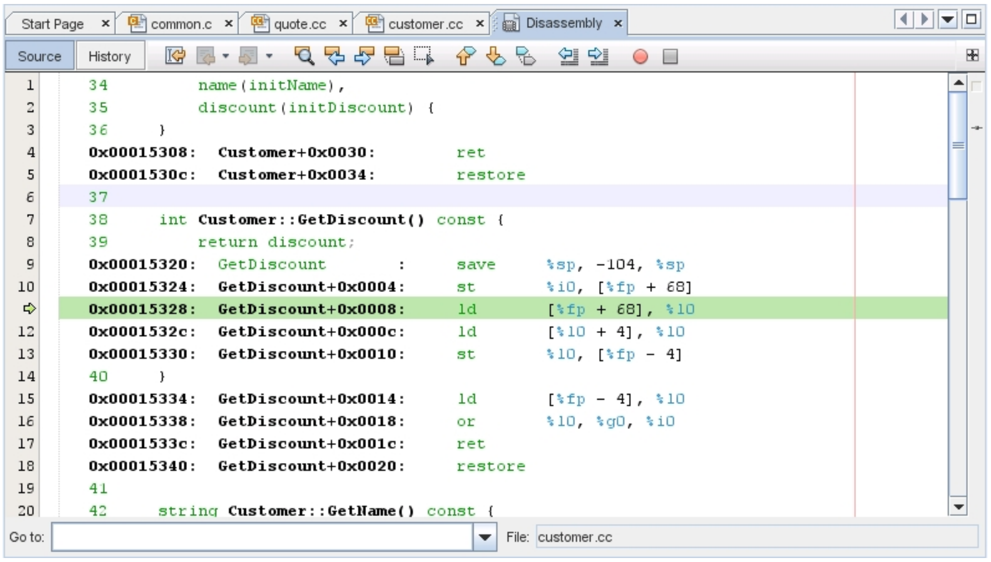
The program you selected is loaded into the IDE, but is in the Pause state. You can continue debugging by clicking Continue.

For more information about Debugging an Executable, see the corresponding IDE help page.

Debugging at the Machine-Instruction Level

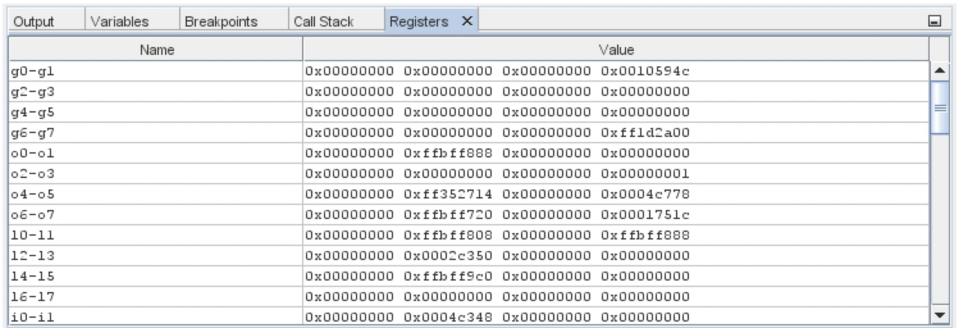
The debugger provides windows that let you debug your project at the machine-instruction level.

1. Right-click the `Quote_1` project and choose Debug.
2. In the Output window, type a customer name in response to the prompt.
3. When the program pauses at the breakpoint on the `GetDiscount` function, choose Window > Debugging > Disassembly to open the Disassembly window as in the Editor window. The green program counter arrow appears on top of the breakpoint icon at the instruction on which the program is paused.



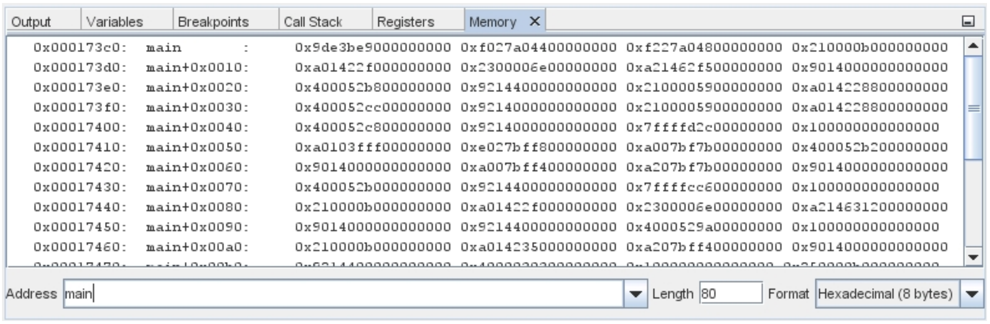
```
34          name(initName),
35          discount(initDiscount) {
36      }
0x00015308:  Customer+0x0030:        ret
0x0001530c:  Customer+0x0034:        restore
37
38      int Customer::GetDiscount() const {
39          return discount;
0x00015320:  GetDiscount        :    save      %sp, -104, %sp
0x00015324:  GetDiscount+0x0004:     st        %i0, [%fp + 68]
0x00015328:  GetDiscount+0x0008:     ld        [%fp + 68], %l0
0x0001532c:  GetDiscount+0x000c:     ld        [%l0 + 4], %l0
0x00015330:  GetDiscount+0x0010:     st        %l0, [%fp - 4]
40      }
0x00015334:  GetDiscount+0x0014:     ld        [%fp - 4], %l0
0x00015338:  GetDiscount+0x0018:     or        %l0, %g0, %i0
0x0001533c:  GetDiscount+0x001c:     ret
0x00015340:  GetDiscount+0x0020:     restore
41
42      string Customer::GetName() const {
```

4. Choose Window > Debugging > Registers to open the Registers window, which displays the contents of the registers.



Name	Value
g0-g1	0x00000000 0x00000000 0x00000000 0x0010594c
g2-g3	0x00000000 0x00000000 0x00000000 0x00000000
g4-g5	0x00000000 0x00000000 0x00000000 0x00000000
g6-g7	0x00000000 0x00000000 0x00000000 0xff1d2a00
o0-o1	0x00000000 0xffbff888 0x00000000 0x00000000
o2-o3	0x00000000 0x00000000 0x00000000 0x00000001
o4-o5	0x00000000 0xff352714 0x00000000 0x0004c778
o6-o7	0x00000000 0xffbff720 0x00000000 0x0001751c
l0-l1	0x00000000 0xffbff808 0x00000000 0xffbff888
l2-l3	0x00000000 0x0002c350 0x00000000 0x00000000
l4-l5	0x00000000 0xffbff9c0 0x00000000 0x00000000
l6-l7	0x00000000 0x00000000 0x00000000 0x00000000
i0-i1	0x00000000 0x0004c348 0x00000000 0x00000000

5. Choose Window > Debugging > Memory to open the Memory window, which displays the contents of memory currently used by your project At the bottom of the window, you can specify a memory address to browse, change the length of the memory browse, or change the format for memory information.



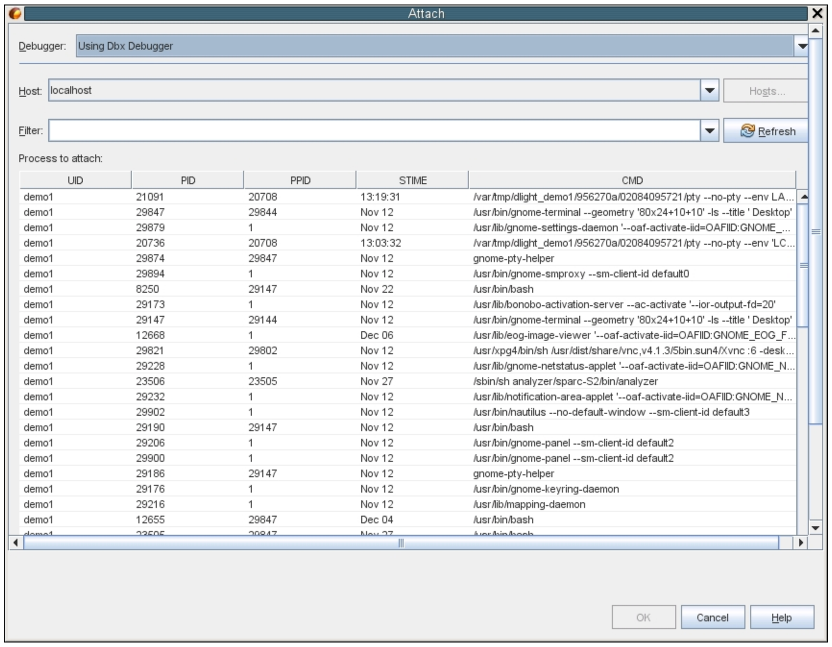
```
0x000173c0:  main       :     0x9de3be9000000000 0xf027a04400000000 0xf227a04800000000 0x210000b000000000
0x000173d0:  main+0x0010:     0xa01422f000000000 0x2300006e00000000 0xa21462f500000000 0x9014000000000000
0x000173e0:  main+0x0020:     0x400052b800000000 0x9214400000000000 0x2100005900000000 0xa014228800000000
0x000173f0:  main+0x0030:     0x400052cc00000000 0x9214000000000000 0x2100005900000000 0xa014228800000000
0x00017400:  main+0x0040:     0x400052c800000000 0x9214000000000000 0x7ffffd2c00000000 0x100000000000000
0x00017410:  main+0x0050:     0xa0103fff00000000 0xe027bff800000000 0xa007bf7b00000000 0x400052b200000000
0x00017420:  main+0x0060:     0x9014000000000000 0xa007bff400000000 0xa207bf7b00000000 0x9014000000000000
0x00017430:  main+0x0070:     0x400052b000000000 0x9214400000000000 0x7ffffcc600000000 0x100000000000000
0x00017440:  main+0x0080:     0x210000b000000000 0xa01422f000000000 0x2300006e00000000 0xa214631200000000
0x00017450:  main+0x0090:     0x9014000000000000 0x9214400000000000 0x4000529a00000000 0x100000000000000
0x00017460:  main+0x00a0:     0x210000b000000000 0xa014235000000000 0xa207bff400000000 0x9014000000000000
```

Address main
Length 80
Format Hexadecimal (8 bytes)

Debugging a Running Program by Attaching to It

If you want to debug a program that is already running, you can attach the debugger to the appropriate process.

1. Choose File > New Project.
2. In the New Project wizard, expand the Samples node and select the C/C++ category.
3. Select the Freeway Simulator project. Click Next and then click Finish.
4. Right-click the `Freeway_1` project you created and choose Run. The project builds and the `Freeway` application starts. In the `Freeway` GUI window, choose Actions > Start.
5. In the IDE, choose Debug > Attach Debugger.



Debugger: Using Dbx Debugger

Host: localhost

Filter:

Process to attach:

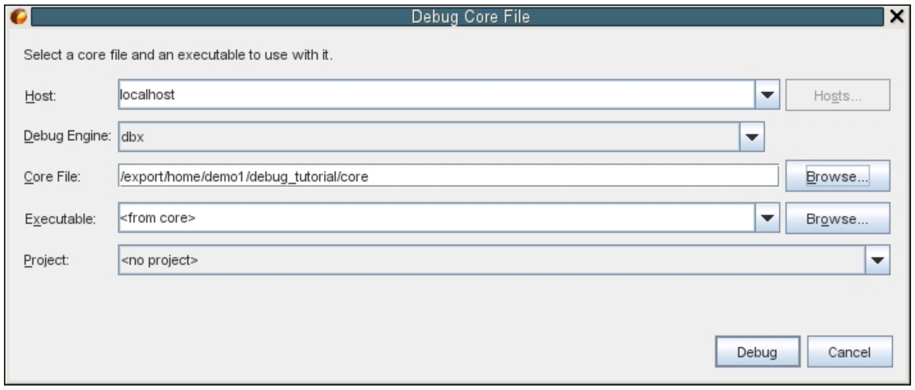
UID	PID	PPID	STIME	CMD
demo1	21091	20708	13:19:31	/var/tmp/dlight_demo1/956270a/02084095721/pty --no-pty --env LA...
demo1	29847	29844	Nov 12	/usr/bin/gnome-terminal --geometry '80x24+10+10' -ls --title ' Desktop'
demo1	29879	1	Nov 12	/usr/lib/gnome-settings-daemon '--oaf-activate-iid=OAFIID:GNOME_...
demo1	20736	20708	13:03:32	/var/tmp/dlight_demo1/956270a/02084095721/pty --no-pty --env 'LC...
demo1	29874	29847	Nov 12	gnome-pty-helper
demo1	29894	1	Nov 12	/usr/bin/gnome-smproxy --sm-client-id default0
demo1	8250	29147	Nov 22	/usr/bin/bash
demo1	29173	1	Nov 12	/usr/lib/bonobo-activation-server --ac-activate '--ior-output-fd=20'
demo1	29147	29144	Nov 12	/usr/bin/gnome-terminal --geometry '80x24+10+10' -ls --title ' Desktop'
demo1	12668	1	Dec 06	/usr/lib/eog-image-viewer '--oaf-activate-iid=OAFIID:GNOME_EOG_F...
demo1	29821	29802	Nov 12	/usr/xpg4/bin/sh /usr/dist/share/vnc,v4.1.3/5bin.sun4/Xvnc :6 -desk...
demo1	29228	1	Nov 12	/usr/lib/gnome-netstatus-applet '--oaf-activate-iid=OAFIID:GNOME_N...
demo1	23506	23505	Nov 27	/sbin/sh analyzer/sparc-S2/bin/analyzer
demo1	29232	1	Nov 12	/usr/lib/notification-area-applet '--oaf-activate-iid=OAFIID:GNOME_N...
demo1	29902	1	Nov 12	/usr/bin/nautilus --no-default-window --sm-client-id default3
demo1	29190	29147	Nov 12	/usr/bin/bash
demo1	29206	1	Nov 12	/usr/bin/gnome-panel --sm-client-id default2
demo1	29900	1	Nov 12	/usr/bin/gnome-panel --sm-client-id default2
demo1	29186	29147	Nov 12	gnome-pty-helper
demo1	29176	1	Nov 12	/usr/bin/gnome-keyring-daemon
demo1	29216	1	Nov 12	/usr/lib/mapping-daemon
demo1	12655	29847	Dec 04	/usr/bin/bash

6. In the Attach dialog box, type `Freeway` in the Filter field to filter the list of processes.
7. Select the `Freeway` process from the filtered list.
8. Click OK.
9. A debugging session is started and execution of the `Freeway` process pauses at the point where the debugger attached to it.
10. Click Continue ▶ to continue execution of `Freeway`, which is now running under control of the debugger. If you click Pause ⏸, execution of `Freeway` pauses, and you can examine variables, the call stack, and such.
11. Click Continue again and then click Finish Debugger Session ■. The debugger session ends, but the `Freeway` process continues executing. Choose File > Exit in the `Freeway` GUI to exit the application.

Debugging a Core File

If your program is crashing, you might want to debug the core file (the memory image of your program when it crashed). To load a core file into the debugger:

1. Choose Debug > Debug core file.
2. Type the full path to a core file in the Core File field or click Browse and navigate to your core file in the Select Core File dialog box.



3. If the debugger cannot associate the core file you specified with an executable, it displays an error message. If this situation occurs, type the path name of the executable in the Executable text box, or click the Browse button and use the Executable dialog box to select the executable.
4. By default, the Project text field displays either `<no project>` or the name of an existing project that exactly matches the name of the executable. If you want a new project created for the executable, select `<create new project>`.
5. Click Debug.

For a more in-depth tutorial on debugging, see the "Oracle Solaris Studio 12.4: dbxtool Tutorial ".

♦ ♦ ♦ CHAPTER 8

Monitoring a Project

The IDE provides tools for examining the behavior of your running C/C++/Fortran project, in order for you to detect runtime problems in your applications. These types of problems might not be detectable when debugging your code. The profiling tools are the following:

- CPU Usage
- Thread Usage
- Memory Usage
- Memory Access Errors

This chapter describes what these tools are and how to use them and contains the following sections:

- “Profiling a Project with the Run Monitor Profiling Tools” on page 65
- “Running Memory Access Checking on Your Project” on page 82

Profiling a Project with the Run Monitor Profiling Tools

The Run Monitor profiling information is collected using Oracle Solaris Studio performance analysis tools and the underlying operating system. The Run Monitor tools graphically show information and have buttons that you can click for more information about problem areas in your code. This section will set up a profiling demo project, enable the Profile tools in the project's properties, and examine the Run Monitor profiling tools results.

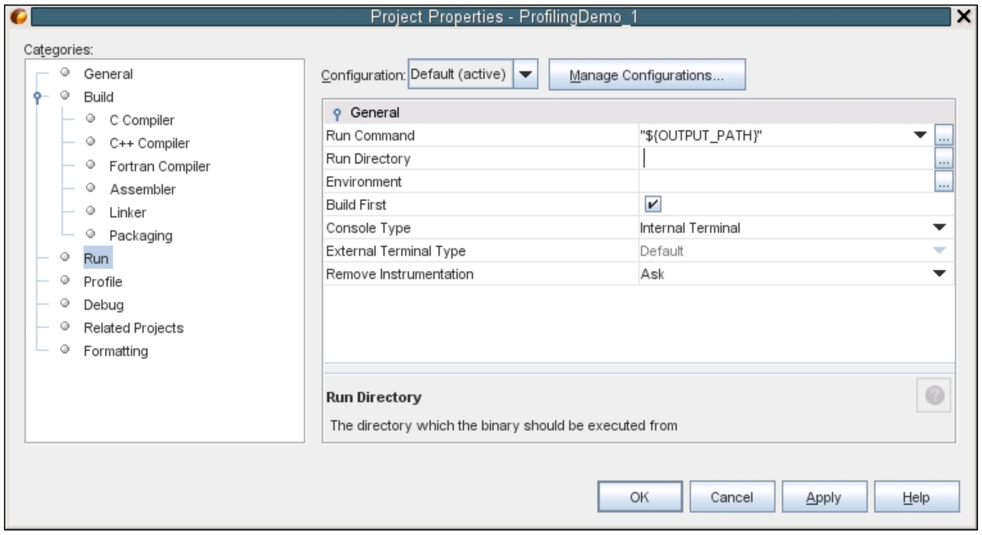
Setting Up the Profiling Project

This tutorial will use the ProfilingDemo sample project. To create this demo:

1. Choose File > New Project (Ctrl+Shift+N).
2. In the New Project wizard, expand the Samples node and select the C/C++ category.
3. Select the Profiling Demo project. Click Next and then click Finish.

To configure the project and enable profiling:

1. Right-click the ProfilingDemo_1 project node in the Projects tab, and select Properties.
2. Select the Build node in the Categories panel, and make sure the Tool Collection is set to Oracle Solaris Studio .
3. Select the Run node in the Categories panel, and make sure the Console Type is set to Internal Terminal. This enables you to see the program output in the IDE's Output window instead of an external terminal window.



Configuration: Default (active)

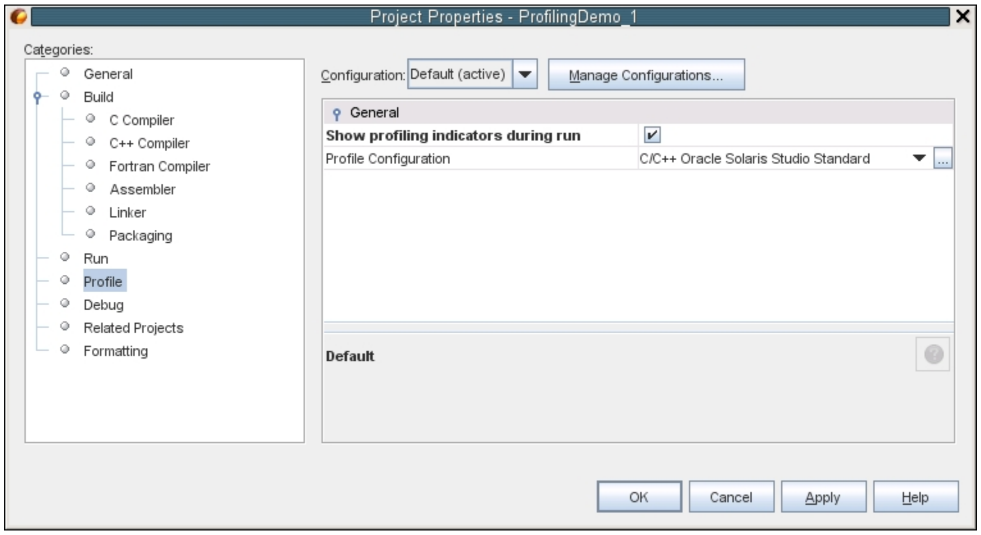
General

Run Command	"${OUTPUT_PATH}"
Run Directory	
Environment	
Build First	✓
Console Type	Internal Terminal
External Terminal Type	Default
Remove Instrumentation	Ask

Run Directory

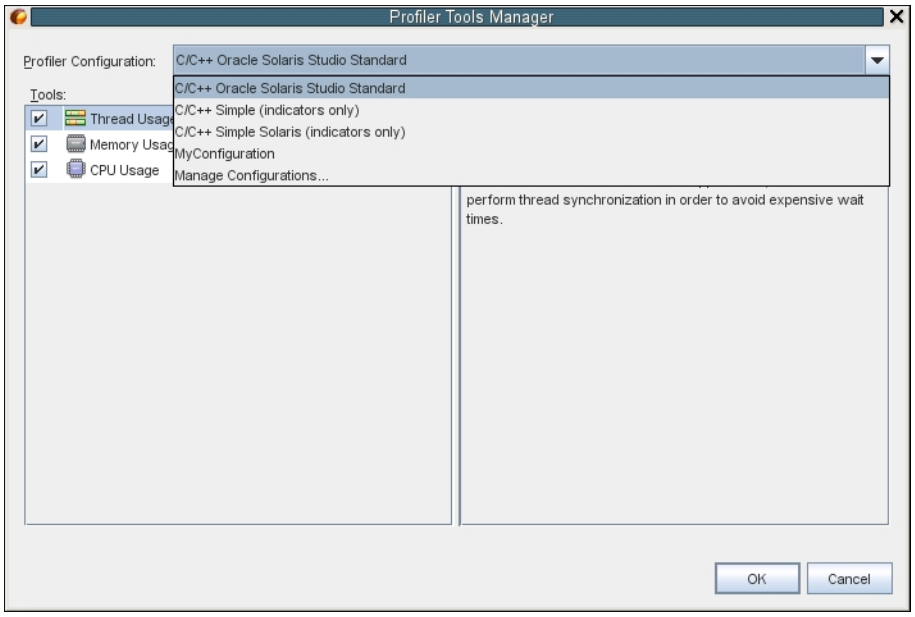
The directory which the binary should be executed from

4. Select the Profile node in the Categories panel. Notice that the Show profiling indicators during run option is checked. You can check this option in any of your projects to show the Run Monitor Tools tab.



Show profiling indicators during run ✓
Profile Configuration: C/C++ Oracle Solaris Studio Standard

5. Select C/C++ Oracle Solaris Studio Standard in Profile Configuration. Click the ... button next to the Profile Configuration list.
6. Notice the tools selected for the C/C++ Oracle Solaris Studio Standard are Thread Usage, Memory Usage, and CPU Usage.


Profiler Tools Manager
Profiler Configuration:
C/C++ Oracle Solaris Studio Standard
C/C++ Oracle Solaris Studio Standard
C/C++ Simple (indicators only)
C/C++ Simple Solaris (indicators only)
MyConfiguration
Manage Configurations...
Tools:
Thread Usage
Memory Usage
CPU Usage
OK
Cancel

perform thread synchronization in order to avoid expensive wait times.

Note - If you select CC++ Simple (indicators only) or C/C++ Simple Solaris (indicators only), then only the indicators in the Run Monitor window will display. Clicking the details buttons in each indicator window will give the following message: Detailed information is not available in the current profile configuration. To see details, select a different Profile Configuration in your project properties and run the project again.

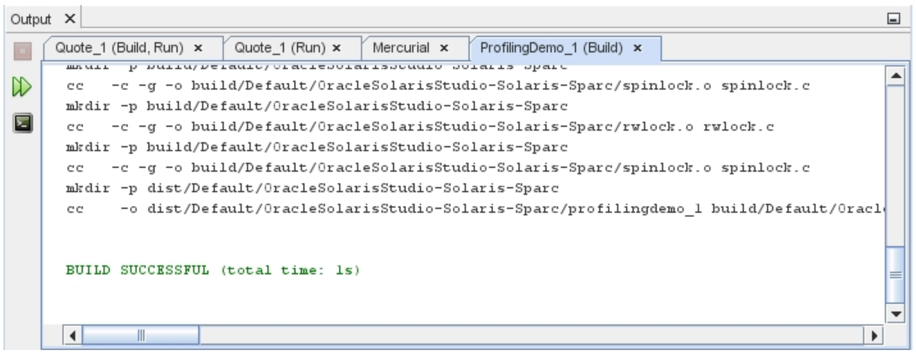
7. Click OK in the Project Properties dialog box.

Building and Running the ProfilingDemo_1 Project

To build and run ProfilingDemo_1:

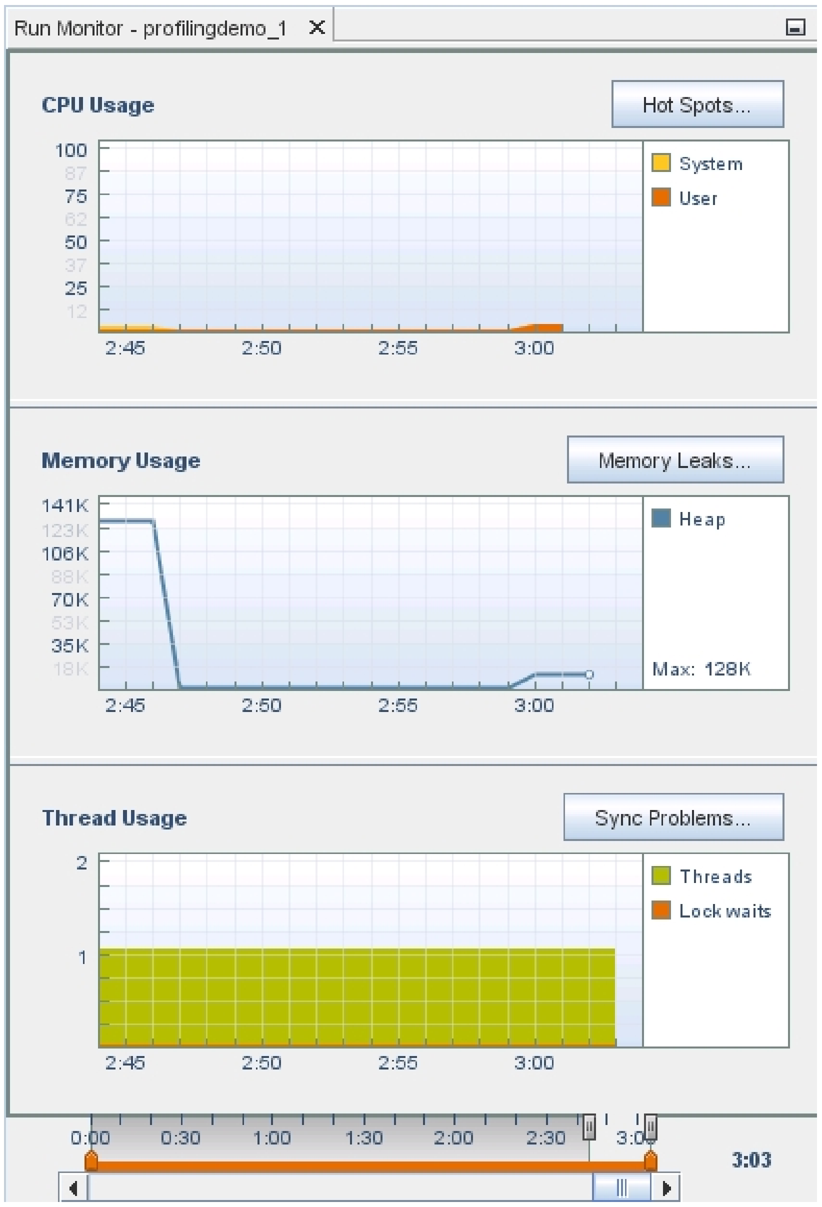
1. Right click the ProfilingDemo_1 project node and select Build.

   The Output window shows the results of the build.



```
cc   -c -g -o build/Default/OracleSolarisStudio-Solaris-Sparc/spinlock.o spinlock.c
mkdir -p build/Default/OracleSolarisStudio-Solaris-Sparc
cc   -c -g -o build/Default/OracleSolarisStudio-Solaris-Sparc/rwlock.o rwlock.c
mkdir -p build/Default/OracleSolarisStudio-Solaris-Sparc
cc   -c -g -o build/Default/OracleSolarisStudio-Solaris-Sparc/spinlock.o spinlock.c
mkdir -p dist/Default/OracleSolarisStudio-Solaris-Sparc
cc    -o dist/Default/OracleSolarisStudio-Solaris-Sparc/profilingdemo_1 build/Default/Oracl


BUILD SUCCESSFUL (total time: 1s)
```

2. Right-click the ProfilingDemo_1 project node and select Run. The Run Monitor window opens to display indicators with dynamic graphs for CPU Usage, Memory Usage, and Thread Usage.

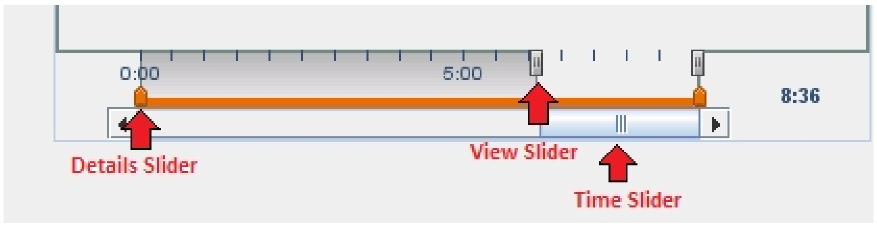


Notice in the Output window, the ProfilingDemo_1 program tells you what it is doing, so you can match it to the data that the IDE is graphically representing in the tools. For example, the program displays how much memory it is allocating, performs calculations, and then frees memory. You can see the graph reflect this activity.

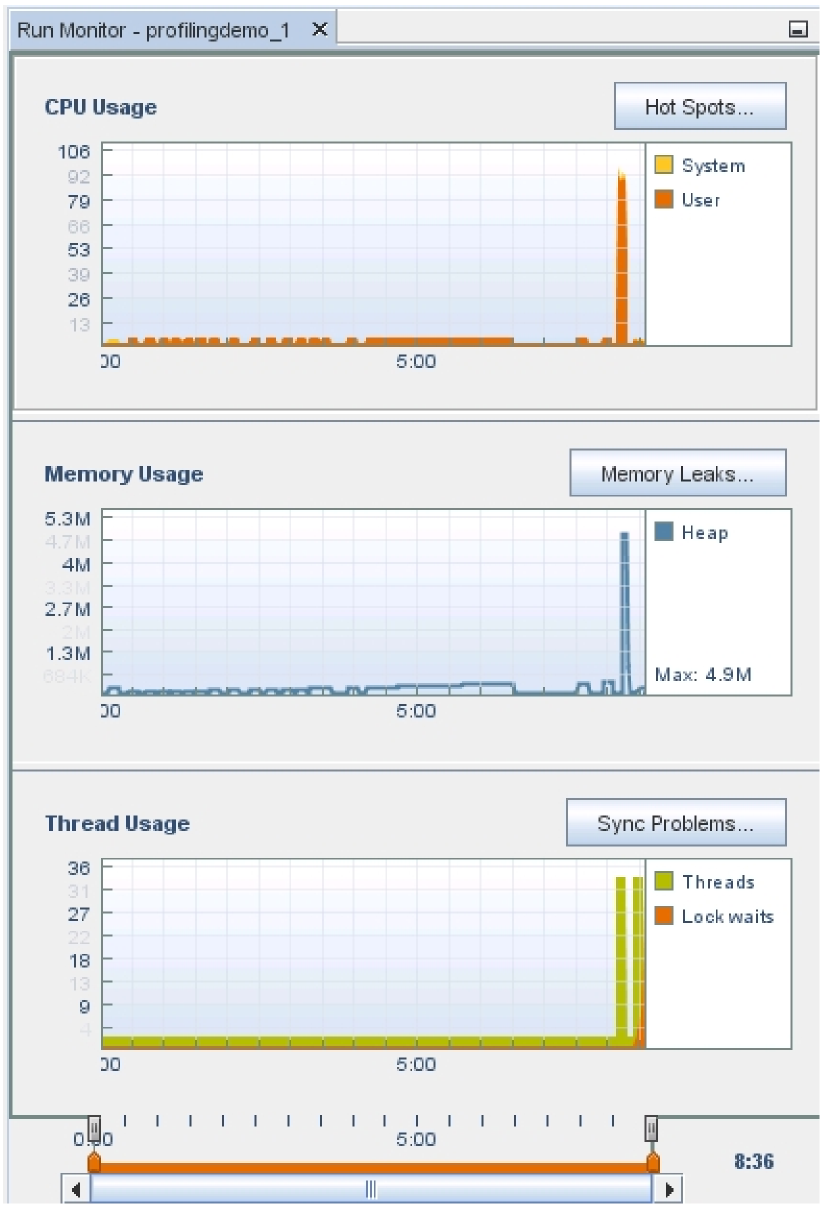
3. Press Enter each time you are requested until the program is finished.
4. Hover your mouse cursor over the indicators to see tooltips explaining what each graph represents. Each indicator includes a button for more detailed information, which will be explored in later sections.

Using the Indicator Controls

At the bottom of the Run Monitor window, there are sliders for controlling your view of the graphs: View slider, Details slider, and Time slider. The controls are also used for filtering data. The following image labels the controls.



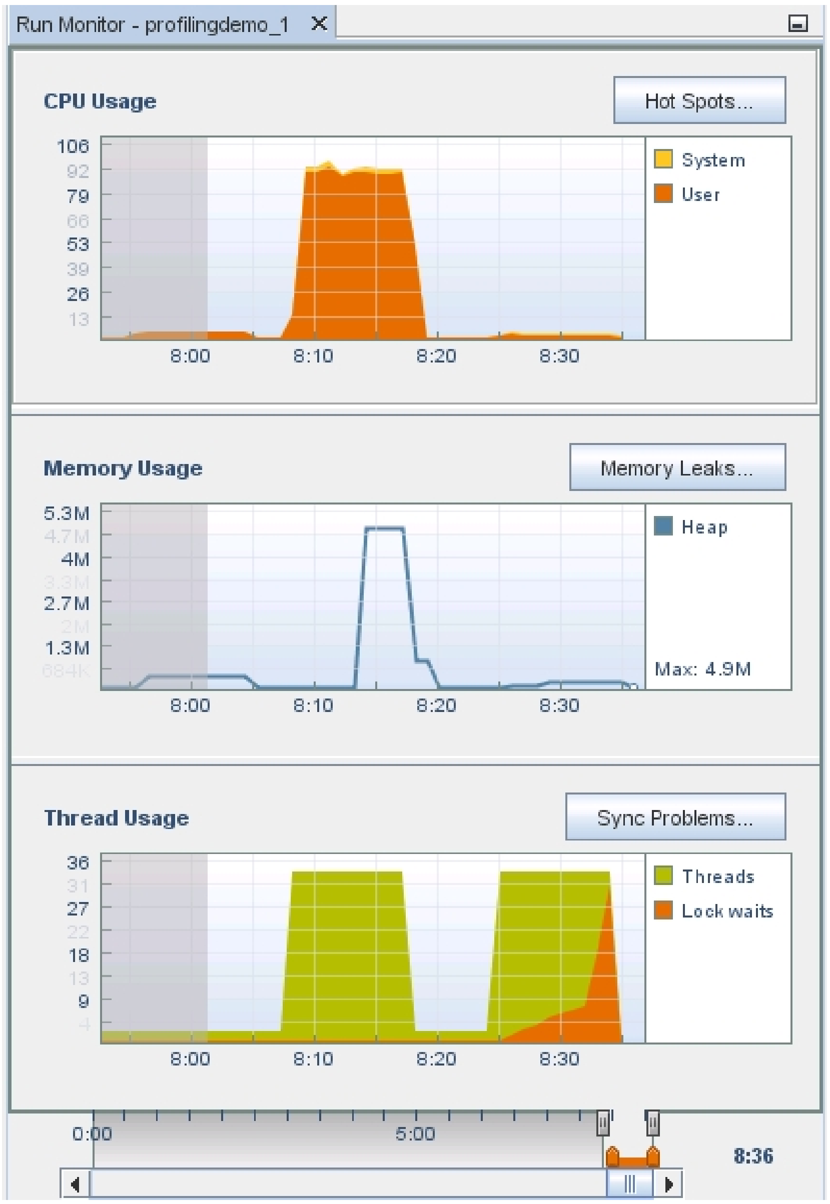
1. Place your mouse cursor over the end points of the sliders for information about the sliders.
2. Click and hold the mouse button on the Time slider, the horizontal scroll bar at the bottom. All the graphs slide in unison so you can see CPU, memory, and threads at any given time and to see the relationships between them.



Drag the time slider from left to right to see the complete run.

3. Move your mouse to the view slider, the controls that overlay the time units. The view slider controls the part of the run time displayed in the indicator.

4. Click and drag the handles of the start point of the view slider to the beginning of the run. The indicators show the entire run. The effect is similar to zooming out. Notice that the time scroll bar is maximized when you select the complete run time, since you are already looking at all the data.
5. Drag the view slider's start point to where the PTHREAD_MUTEX_DEMO starts. As you drag the handle, the indicators zoom in to focus on this area. Notice that the scroll bar can again be used to scroll the run time.
6. Place your mouse cursor over the end points of the orange-colored details slider for a description of how to use the slider. The details slider controls enable you to select a portion of the runtime for examining the detailed information.



Drag the details slider's start point handle past the view slider's start point. Notice that the indicators grey out the area in from the of the detail slider's start point. This gives a highlighting effect to the graph between the start and end points. When you click an indicator's detail button, the details tab shows the data for the area highlighted.

7. Drag the start point of the details slider back to the beginning so you can see all the data.

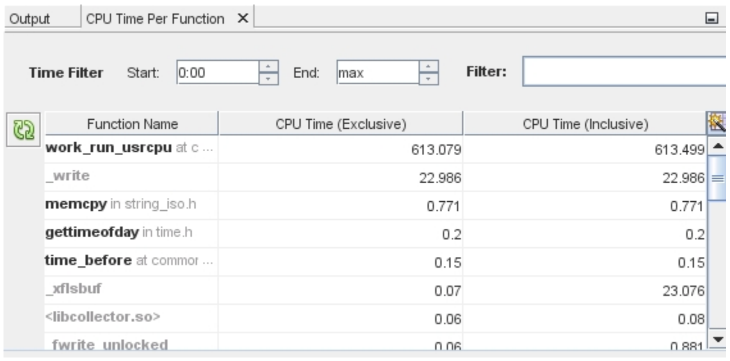
Note - You can slide either the starting handle or ending point handle for the view slider and the details slider, so you can view and get details at any time of your run.

Exploring CPU Usage

The CPU Usage graph shows the percentage of the total CPU time used by your application during its run.

1. Click the CPU Usage Hot Spots... button to display details about the CPU Time.

   The CPU Time Per Function window opens to display the functions of the program, along with the CPU time used by each function. The functions are listed in order of CPU time used, with the function that used the most time listed first. If the program is still running, the time initially displayed is the time consumed at the moment you clicked the graph.



Start: 0:00 End: max

Function Name	CPU Time (Exclusive)	CPU Time (Inclusive)
work_run_usrcpu at c ...	613.079	613.499
_write	22.986	22.986
memcpy in string_iso.h	0.771	0.771
gettimeofday in time.h	0.2	0.2
time_before at commor ...	0.15	0.15
_xflsbuf	0.07	23.076
<libcollector.so>	0.06	0.08
_fwrite_unlocked	0.06	0.881

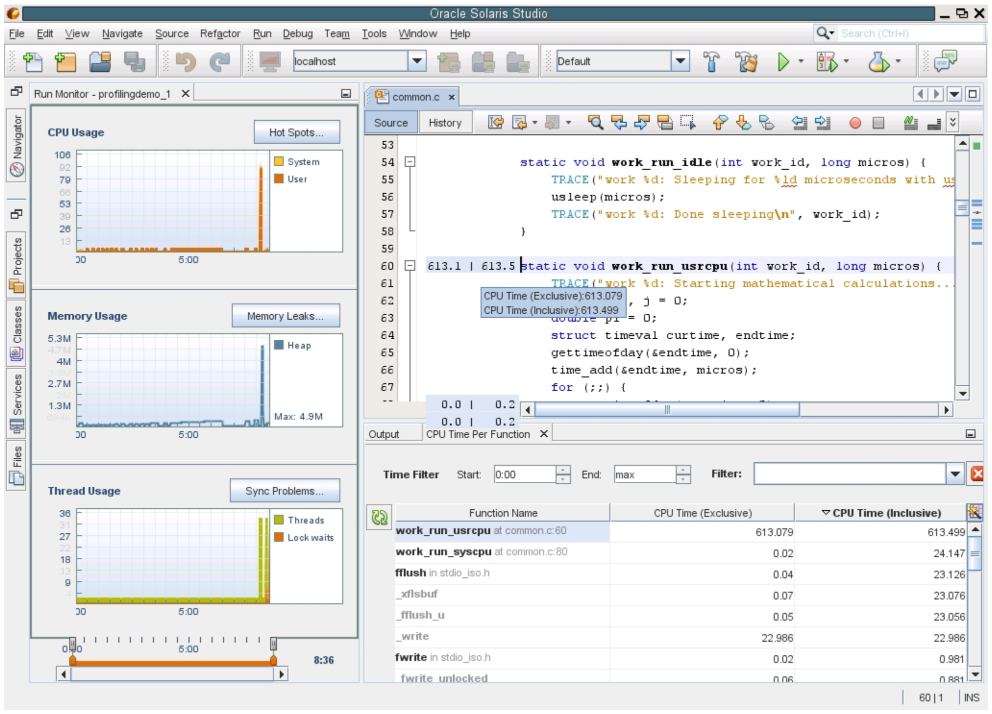
2. Click the Function Name column head to sort the functions alphabetically.

   Notice the difference between the two columns of CPU Time. CPU Time (Inclusive) shows the total CPU time spent form the time the function is entered until the time it is exited, including the time of all other functions that are called by the listed function. CPU Time (Exclusive) shows the time used only by the specific function, not including any functions that it calls.

3. Click the CPU Time (Inclusive) column head to place the most time-consuming functions back at the top.

   Notice that in our example, the `work_run_usrcpu()` function has 613.079 CPU Time (Exclusive) versus a 613.499 CPU Time (Inclusive). This means that a small amount of the CPU time was actually used by other functions that it calls, but the `work_run_usrcpu()` function itself used most of the time.

4. Some of the functions are listed in bold. You can go to the source file that calls these functions. Double-click the `work_run_usrcpu()` function.



```
53
54              static void work_run_idle(int work_id, long micros) {
55                  TRACE("work %d: Sleeping for %ld microseconds with us
56                  usleep(micros);
57                  TRACE("work %d: Done sleeping\n", work_id);
58              }
59
60  613.1 | 613.5 static void work_run_usrcpu(int work_id, long micros) {
61                  TRACE("work %d: Starting mathematical calculations..
62                          , j = 0;
63                          pi = 0;
64                  struct timeval curtime, endtime;
65                  gettimeofday(&endtime, 0);
66                  time_add(&endtime, micros);
67                  for (;;) {
      0.0 |   0.2
      0.0 |   0.2
```

CPU Time (Exclusive):613.079
CPU Time (Inclusive):613.499

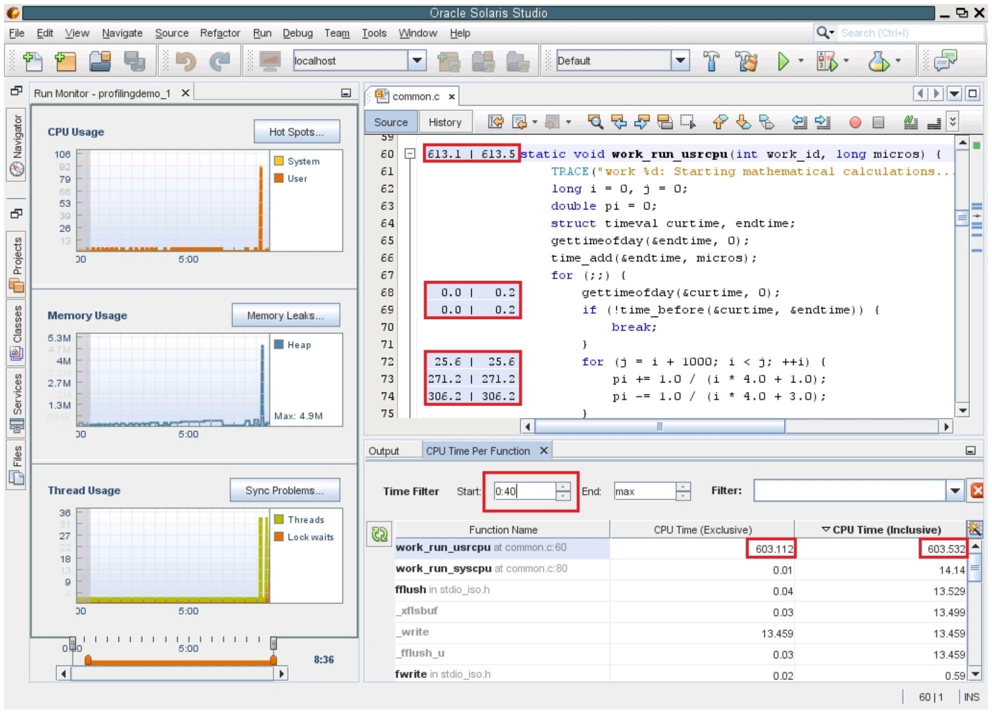
Time Filter Start: 0:00 End: max Filter:

Function Name	CPU Time (Exclusive)	▽ CPU Time (Inclusive)
work_run_usrcpu at common.c:60	613.079	613.499
work_run_syscpu at common.c:80	0.02	24.147
fflush in stdio_iso.h	0.04	23.126
_xflsbuf	0.07	23.076
_fflush_u	0.05	23.056
_write	22.986	22.986
fwrite in stdio_iso.h	0.02	0.981
fwrite_unlocked	0.06	0.881

60 | 1 INS

The `common.c` file opens, with the cursor resting on the `work_run_usrcpu()` function, line 60. Some numbers are displayed in the left margin for this line.

5. Place your cursor over the numbers in the left margin. The numbers are the same metrics for exclusive and inclusive CPU time for the functions displayed in the CPU Time Per Function window. The metrics are rounded to the nearest tenth, but the unrounded value pops up when you hover the mouse over them. Metrics for CPU-consuming lines such as the `for()` loop that does calculations within the `work_run_usrcpu()` function are also shown in the `common.c` source file.
6. In the CPU Time Per Function window, change the Time Filter start time to 0:40 by either typing the time and pressing Enter, or using the arrows to scroll through the seconds. The graphic indicators change just as they did when you moved the handles on the data filtering control. If you drag the handles, the Time Filter settings in the CPU Time Per Function window are updated to match. More importantly, the data shown for the functions in the table is updated to reflect the filter so only the CPU time used during that time period is shown.



```
613.1 | 613.5  static void work_run_usrcpu(int work_id, long micros) {
                       TRACE("work %d: Starting mathematical calculations...
                       long i = 0, j = 0;
                       double pi = 0;
                       struct timeval curtime, endtime;
                       gettimeofday(&endtime, 0);
                       time_add(&endtime, micros);
                       for (;;) {
  0.0 |   0.2              gettimeofday(&curtime, 0);
  0.0 |   0.2              if (!time_before(&curtime, &endtime)) {
                               break;
                           }
 25.6 |  25.6              for (j = i + 1000; i < j; ++i) {
271.2 | 271.2                  pi += 1.0 / (i * 4.0 + 1.0);
306.2 | 306.2                  pi -= 1.0 / (i * 4.0 + 3.0);
                           }
```

Time Filter Start: 0:40 End: max Filter:

Function Name	CPU Time (Exclusive)	CPU Time (Inclusive)
work_run_usrcpu at common.c:60	603.112	603.532
work_run_syscpu at common.c:80	0.01	14.14
fflush in stdio_iso.h	0.04	13.529
_xflsbuf	0.03	13.499
_write	13.459	13.459
_fflush_u	0.03	13.459
fwrite in stdio_iso.h	0.02	0.59

60 | 1 INS

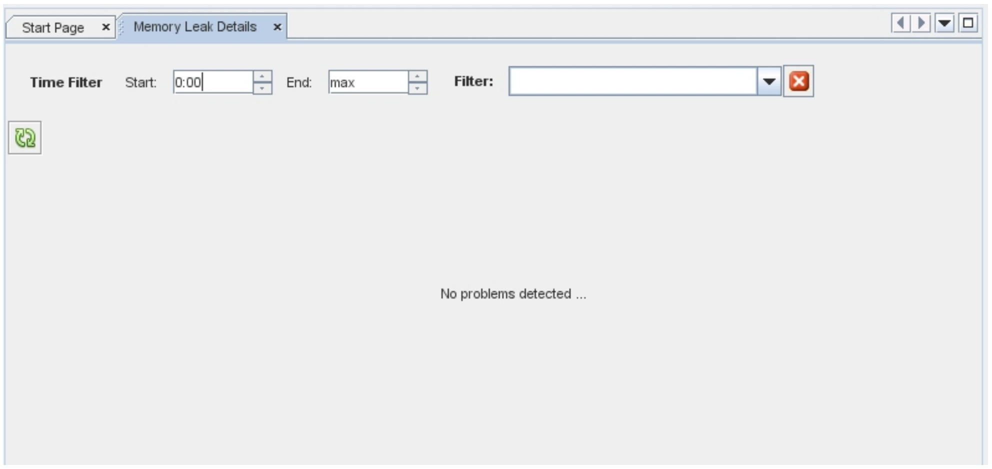
7. You can also filter for data that meet a certain metric. Right-click on the CPU Time (Exclusive) metrics for `work_run_usrcpu()`. In our example, right-click the 603.112 in the CPU Time Per Function window. Select Show only where > CPU Time (Exclusive) == 603.112. All the rows are filtered away, and only the row whose exclusive CPU time is equal to 603.112 is displayed.

Exploring Memory Usage

The Memory Usage indicator shows how your project's memory heap changes over its runtime. You can use it to identify memory leaks, which are points in your program where memory that is no longer needed fails to be released. Memory leaks can lead to increased memory consumption in your program. If a program with a memory leak runs long enough, it can eventually run out of usable memory.

1. Slide the time slider in the Run Monitor to the left and right to see how the memory heap increases and decreases over time. There should be four spikes that occur in ProfilingDemo_1. The first two occur during the Sequential Demo, the third occurs during the Parallel Demo, and the last occurs during the Pthread Mutex Demo.

2. Click the Memory Leaks button to display the Memory Leak Details window. This shows details about which functions exhibit memory leaks. Only functions that are producing memory leaks are listed in the table. If your program is running when you click the button, the leak locations shown are those that exist at the moment you clicked the button. More leaks may occur as time goes on, so you should click the Refresh button. If no memory leaks are detected by the end of the run, the Memory Leak Details tab indicates that no memory leaks were found.
3. You can change the Start and End times to filter the data, or use the orange colored details slider in the Run Monitor window, same as in the Exploring CPU Usage section. There are no memory leaks in this example.



No problems detected ...

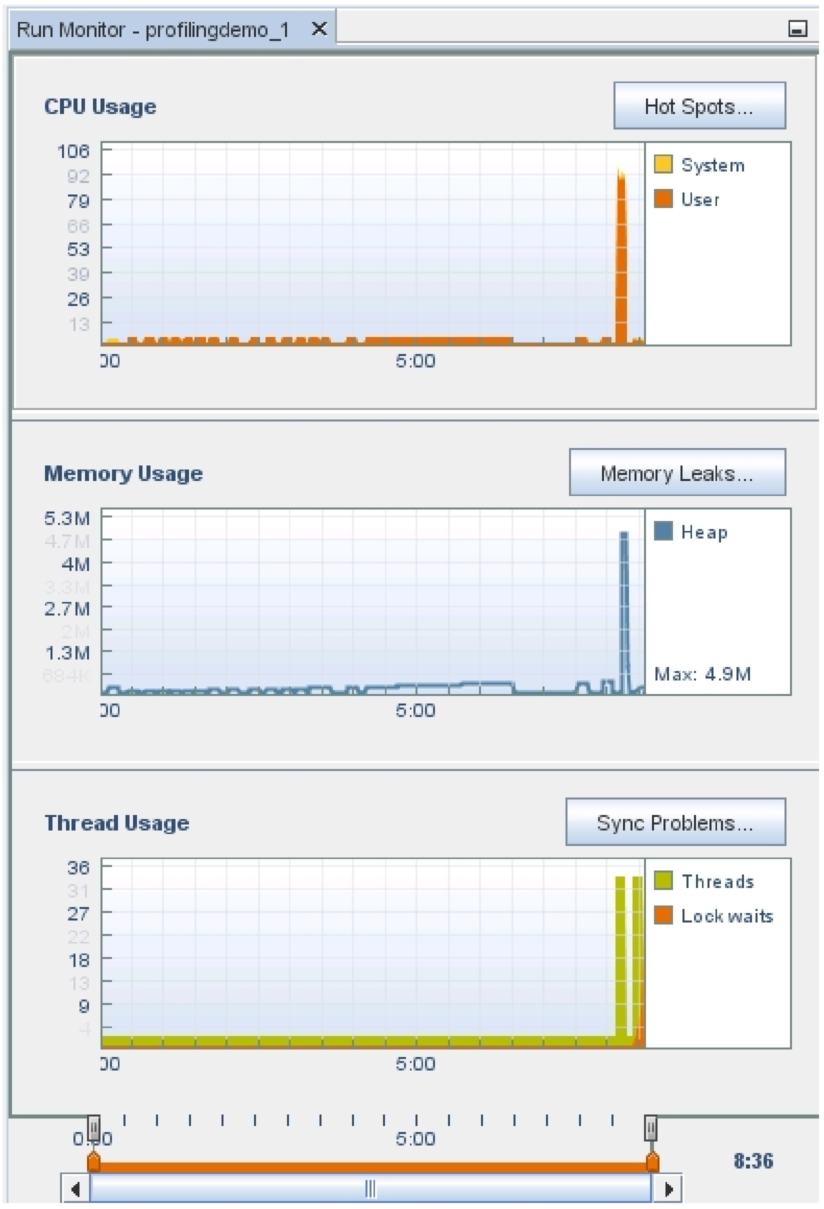
4. Double-click a function and the corresponding file opens at the line where the memory leak occurs in the function. The memory leak metrics are displayed in the left margin in the source editor.
5. Mouse over them to display the details as we did with the CPU Usage metrics.
6. Right-click the metrics in the table to filter the data in the table. This data filtering in the table is possible with all the profiling tools.

Exploring Thread Usage

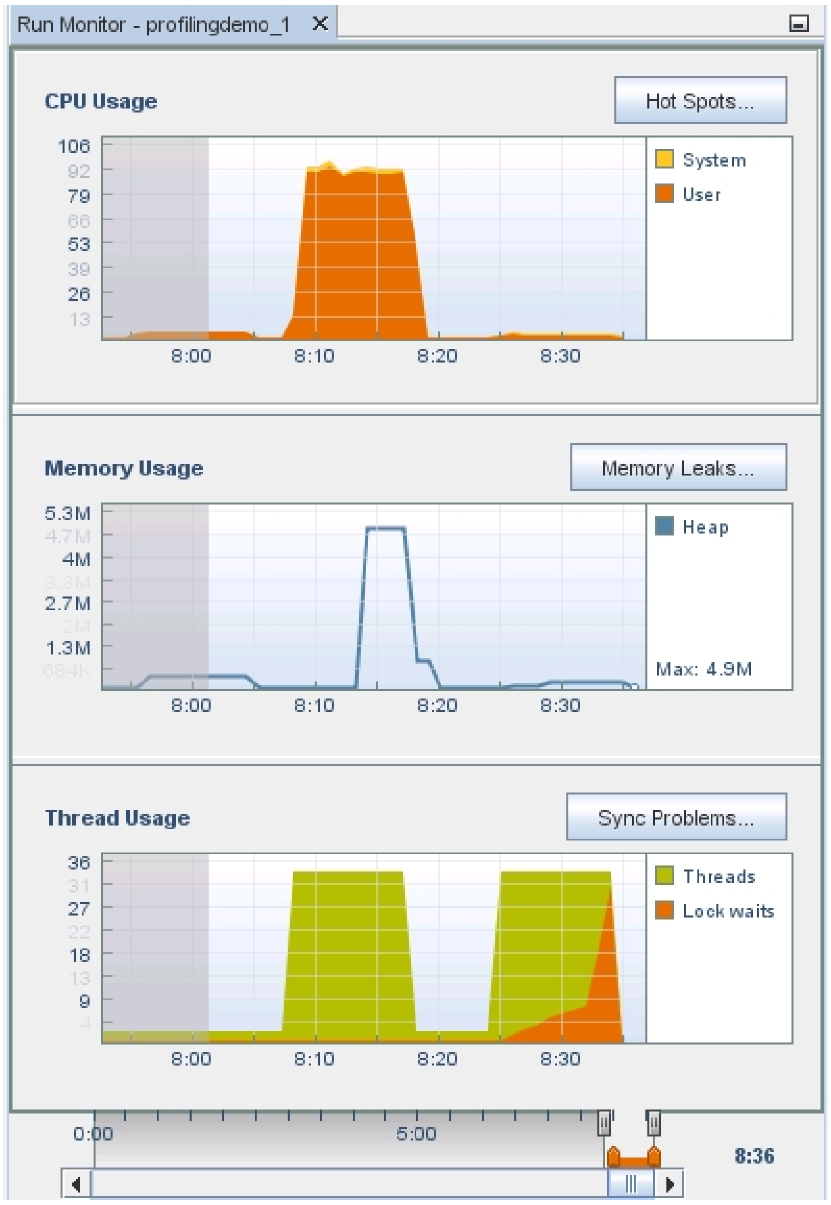
The Thread Usage indicator shows the number of threads in use by your program, and any moments where a thread has to wait to get a lock in order to proceed with its task. This data is useful for multithreaded applications, which must perform thread synchronization in order to avoid expensive wait times.

1. The Thread Usage indicator shows the number of threads running during a project's run time. Slide the time slider back to the beginning and notice the number of threads is 1 until

the Parallel Demo starts. In the following image, this is at 8:07, with a jump of 1 to 32 threads.



2. Move the view slider endpoint handle to where the Parallel Demo just starts.



Notice that in the CPU Usage and Memory Usage indicators for the same time period has the single thread performing some activity that uses CPU time and memory. This corresponds to the Sequential Demo portion, in which the main thread writes to a file and then performs some calculations sequentially. CPU and memory usage both decrease when the program waits for the user to press Enter and the number of threads remains at 1.

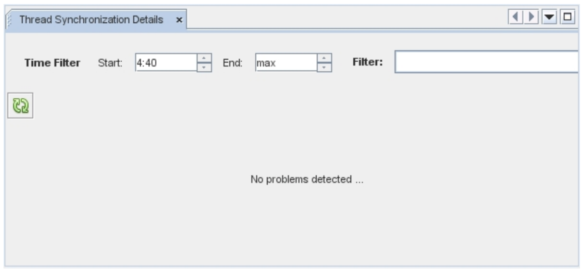
3. Slide the time slider to the right until you see the two points where the threads increase in number.

   The increase in threads at 8:07 corresponds to the Parallel Demo portion of the project running. The main threads starts additional threads to the work of writing to a file and performing calculations in parallel. Notice the increase in memory usage and CPU usage, but the two tasks are completed in much less time.

4. The number of threads returns to 1 after the Parallel Demo threads finish, and the main thread waits for you to press Enter. The thread count increases again as the Pthread Mutex Demo portion of the program runs.

   Notice that a lock waits appears, shown in orange, during the Pthread Mutex Demo portion. The Pthread Mutex Demo uses mutual exclusion locks to prevent overlapping access to certain functions by multiple threads, which causes the threads to wait to obtain a lock.

   Click the Sync Problems button to display details about thread locks in your project. The Thread Synchronization Details window opens and lists functions that had to wait to obtain a mutex lock. Also displayed are metrics for the number of milliseconds that the function spent waiting and the number of times the function had to wait for a lock. This example had no sync problems.



No problems detected ...

   If you click the Sync Problems button while the program is running, you might need to click the refresh button to update the display with the latest thread lock information.

5. Click the Wait Time column to sort the functions in order of time spent waiting.
6. Click the Lock Waits column to sort the functions by the number of times a thread was waiting in the functions.
7. Double-click a function to open the source file in the editor where the function responsible for locking a memory location. The metrics for Wait Time and Lock Waits are displayed in the left column f the source file. Place your mouse over the metrics to see the details, which matches what is shown in the Thread Synchronization Details window.
8. Right-click on the metrics and deselect Show Profiler Metrics. The metrics are no longer displayed in the source editor for any of the profiling tools. To display the metrics again, choose View in the IDE menu bar and select Show Profiler Metrics.

Running Memory Access Checking on Your Project

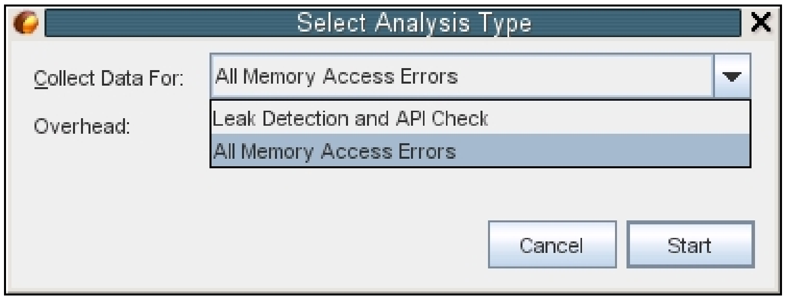
You can use the Memory Analysis Tool to find memory access errors in your project. The tool allows you to find these errors easily by pointing out exactly where each error occurs in your source code.

The Memory Analysis tool catches and reports memory access errors dynamically during program execution, so if a portion of your code is not executed at run time, errors in that portion are not reported.

1. If you have not already done so, download the sample applications zip file from the Oracle Solaris Studio 12.3 Sample Applications web page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html`, and unzip the file in a location of your choice. The `memorychecks` application is located in the `CodeAnalyzer` subdirectory of the `SolarisStudioSampleApplications` directory.
2. Create a project from existing sources using the `memorychecks` application.
3. Right-click the project and choose Properties. In the Project Properties dialog box, select the Run node, and type Customer.db after the output path in the Run Command. Click OK.
4. Run the project.
5. Build the project with instrumentation for memory analysis.

    a. Click the down arrow next to the Profile Project button and select Profile Project to find Memory Access Errors from the drop-down list.

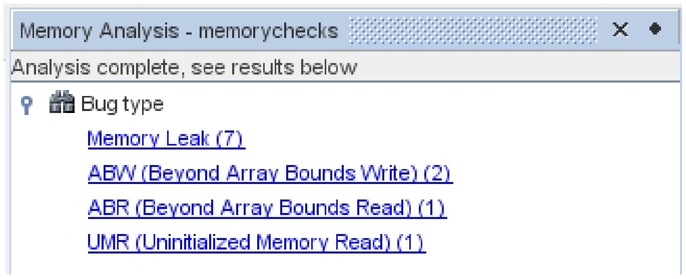
    b. In the Select Analysis Type dialog box, select All Memory Access Errors from the drop-down list.



The Overhead field displays High or Moderate to indicate the load that will be placed on the system. The performance of other programs running on your system might be affected when the overhead is high, which is the case when you are detecting both data races and deadlocks.

   c. Click Start.

6. The Run Memory Profile dialog box opens to let you know that your binary will be instrumented. Click OK.
7. The project is built and instrumented. The application starts running and the Memory Analysis window opens. When your project run is complete, the Memory Analysis window lists the memory access error types found in your project. The number of errors of each type is shown in parentheses after the error type.

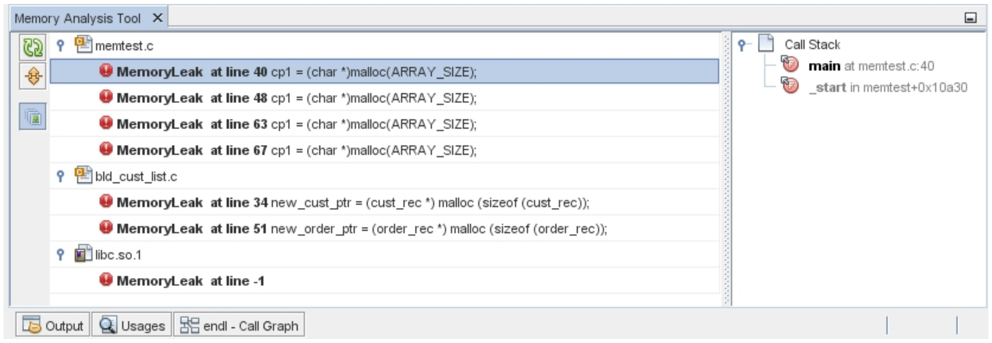


Memory Analysis - memorychecks

Analysis complete, see results below

Bug type
- Memory Leak (7)
- ABW (Beyond Array Bounds Write) (2)
- ABR (Beyond Array Bounds Read) (1)
- UMR (Uninitialized Memory Read) (1)

8. When you click on an error type, the errors of that type are displayed in the Memory Analysis Tool window.



Memory Analysis Tool

```
memtest.c
    MemoryLeak  at line 40 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 48 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 63 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 67 cp1 = (char *)malloc(ARRAY_SIZE);
bld_cust_list.c
    MemoryLeak  at line 34 new_cust_ptr = (cust_rec *) malloc (sizeof (cust_rec));
    MemoryLeak  at line 51 new_order_ptr = (order_rec *) malloc (sizeof (order_rec));
libc.so.1
    MemoryLeak  at line -1
```

```
Call Stack
    main at memtest.c:40
    _start in memtest+0x10a30
```

By default, the errors are grouped by the source file in which they were found. When you click on an error, the call stack for that error is displayed. Double-click a function call in the stack to display the associated lines in the source file.

♦ ♦ ♦ CHAPTER 9

Doing Remote Development

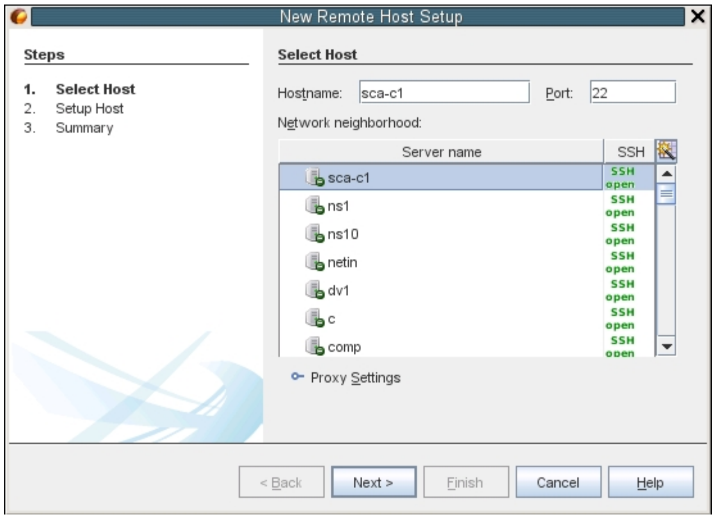
This chapter describes how to do remote development. It also describes how to use the remote development toolbar. Additionally, this chapter gives a brief description of remote desktop distribution.

Doing Remote Development

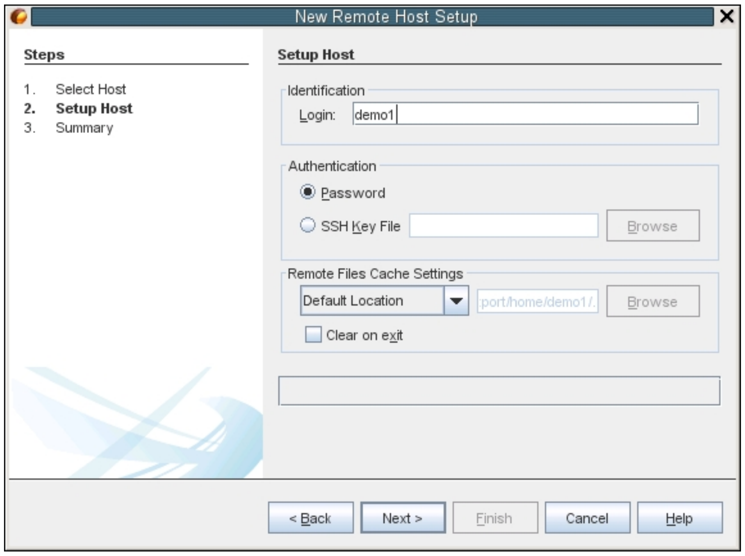
You can build, run, and debug projects on the local host (the system from which you started the IDE) or on a remote host running a UNIX® operating system. Remote development lets you run the IDE locally in your familiar desktop environment, while using the computing power and development tools on a remote server to build your projects.

You can configure remote development hosts in the Build Tools tab of the Options dialog box. To add a remote host:

1. Choose Tools > Options, and click the C/C++ category.
2. In the Build Tools tab of the Options dialog box, click Edit.
3. In the Build Hosts Manager dialog box, click Add.
4. On the Select Host page in the New Remote Build Host wizard, type the system name of the host in the Hostname field or double-click an available host in the Network neighborhood list to select it. Click Next.



5. On the Setup Host page, type your login name in the Login field and click Next.



6. The wizard prompts you for a password, connects to the host, and displays a Summary page. Click Finish.
7. After the host is added to the Build Hosts list in the Build Hosts Manager dialog box, click OK.

8. You can set properties that specify how the IDE uses the remote host in the Services window. Expand the C/C++ Build Hosts node, right-click the remote host, and choose Properties. Set the desired properties in the Host Properties dialog box.
9. To set the remote host as the default build host, right-click the host in the C/C++ Build Hosts node in the Services window, and choose Set as Default.

To develop a project on a remote host, the project must be on a shared filesystem that is visible on both the local host and the remote host. Typically such a filesystem is shared using NFS or Samba. You can define the mapping between local and remote paths to project source files when you define the remote host.

When you create a project, the default build host is selected as the build host for the project. You can change the build host for the project on the Build panel of the Project Properties dialog box. You can also specify the build host when you are debugging an executable or a core file.

To work on a project that resides on a remote host on your local host, choose File > Open Remote C/C++ Project.

Using the Remote Development Toolbar

The IDE provides a Remote Development Toolbar, for easier access to projects and files on the remote host. You can use the icons to select a remote host you have already configured and work on projects and files on the remote host the same as if they were local. This is known as full remote development.

To learn more about remote development modes, see the C/C++ Remote Development Modes help page in the IDE.

You can do the following with the remote toolbar:

- Click the Connect Status icon to connect or disconnect to the server selected in the list next to the icon.
- Click the Create Remote Project icon to create a new project on the currently connected host.
- Click the Open Remote Project icon to open a project on the currently connected host.
- Click the Open Remote File icon to open a file on the currently connected host.

Using the Terminal Window for a Build Host

You can open a terminal window in the IDE for a local or remote host. The terminal window opens as a tab in the Output area of the IDE. A remote host does not need to be set up in advance for the IDE, but must be running the SSH daemon.

To learn about how to open a terminal window in the IDE, see Opening a Terminal Window for a Build Host in the IDE help. To open a terminal window for a local host, choose Window > IDE Tools > Terminal.

You can also personalize the display options of the command line terminal window. For more information, see Options Window: Miscellaneous: Terminal in the IDE help.

Remote Desktop Distribution

The Remote Desktop Distribution feature lets you generate a zip file containing a distribution of the IDE and the Code Analyzer that will run on almost any operating system and use the Oracle Solaris Studio compilers and tools on a remote server. When you run the IDE on a desktop system, it will recognize the server on which you generated the distribution as a remote host and access the tool collection in your Oracle Solaris Studio installation.

For more information about Remote Desktop Distribution, see `How to Develop Code from a Remote Desktop with Oracle Solaris Studio (http://www.oracle.com/technetwork/articles/servers-storage-dev/howto-use-ide-desktop-1639741.html)`.

♦ ♦ ♦ CHAPTER 10

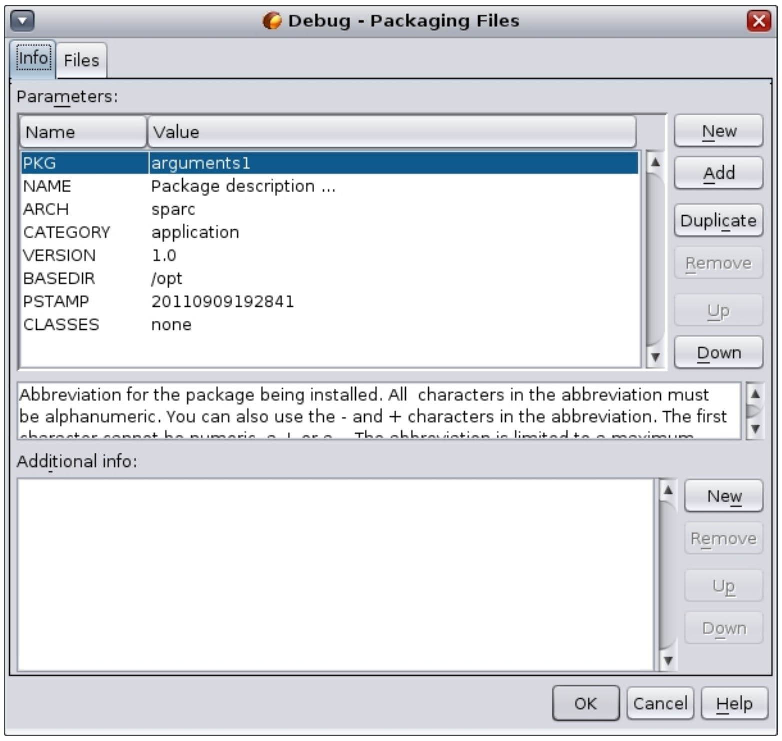
Packaging an Application

This chapter describes how to package an application after you've finished creating, compiling, and debugging your application.

Packaging an Application

You can package a completed application as a tar file, zip file, Solaris SVR4 package, RPM, or Debian package.

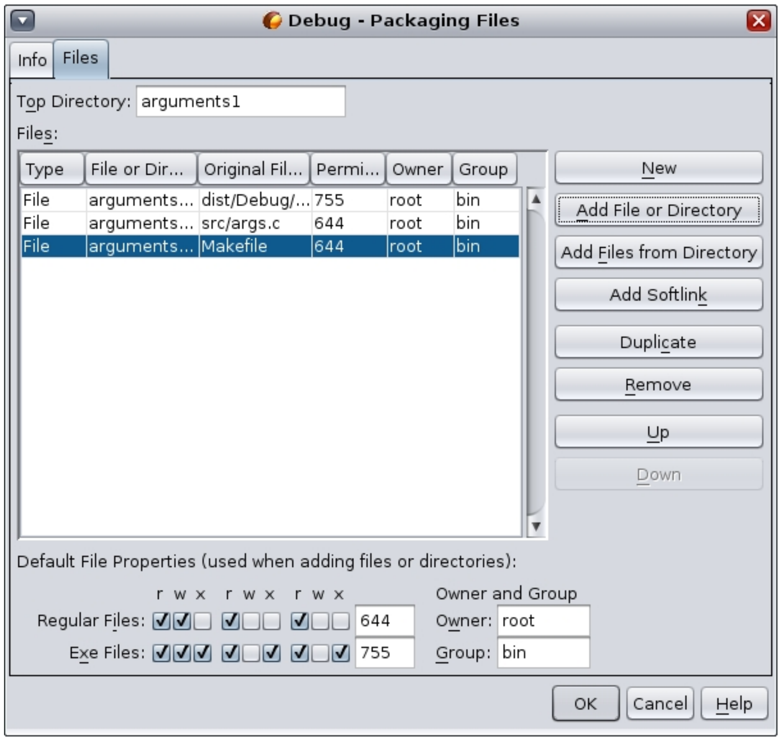
1. Right-click the `Arguments_1` project and choose Properties.
2. In the Project Properties dialog box, select the Packaging node.
3. Select the Solaris SVR4 package type from the drop-down list.
4. Change the output path if you want to use a different destination directory or filename for the package.
5. Click the Packaging Files browse button. In the Packaging Files dialog box (for an SVR4 package), modify the package parameters on the Info tab as needed.



Name	Value
PKG	arguments1
NAME	Package description ...
ARCH	sparc
CATEGORY	application
VERSION	1.0
BASEDIR	/opt
PSTAMP	20110909192841
CLASSES	none

Abbreviation for the package being installed. All characters in the abbreviation must be alphanumeric. You can also use the - and + characters in the abbreviation. The first

6. For all package types, add files to the package using the buttons on the Files tab. For each file, you can specify the path you want it to have in the package in the File or Directory Path in Package column of the Files list. Click OK when your Files list is complete.



Top Directory: arguments1

Files:

Type	File or Dir...	Original Fil...	Permi...	Owner	Group
File	arguments...	dist/Debug/...	755	root	bin
File	arguments...	src/args.c	644	root	bin
File	arguments...	Makefile	644	root	bin

Default File Properties (used when adding files or directories):

	r	w	x	r	w	x	r	w	x		Owner and Group	
Regular Files:	✓	✓		✓			✓			644	Owner:	root
Exe Files:	✓	✓	✓	✓		✓	✓		✓	755	Group:	bin

7. Turn off verbose mode if you wish by clicking the checkbox.
8. Click OK.
9. To build your package, right-click the project and choose More Build Commands > Build Package.

♦ ♦ ♦ CHAPTER 11

More Information About the IDE

This chapter points you to other resources about Oracle Solaris Studio IDE.

Learn More About the IDE

You can learn about using the Oracle Solaris Studio IDE by reading the IDE's integrated help, which you can access through the IDE's Help menu or by pressing the F1 key. Many dialog boxes also have a Help button for information about how to use the dialog box.

The tutorials on the `NetBeans IDE C/C++ Learning Trail` can also be helpful for learning how to use the Oracle Solaris Studio IDE, although there are some differences between the user interfaces and features. In particular, NetBeans documentation about debugging does not apply to Oracle Solaris Studio IDE.

Oracle Database Projects

For more information about the database explorer in the IDE, see `How to Use the Database Explorer in the Oracle Solaris Studio IDE`.

For information about Oracle Instant Client in the IDE, see `How to Use Oracle Instant Client in the Oracle Solaris Studio IDE`.

Remote Development

For information on choosing a remote development mode, see `How to Select a Remote Development Mode in Oracle Solaris Studio`.

For more general information about remote development, see the Overview of C/C++ Remote Development help page in the IDE. This page also directs you to other useful pages about remote development.

Version Control Systems

The IDE is integrated with the option to use several version control systems, like Mercurial, Subversion, CVS, and Git, enabling easier source code management for your projects. For more information, see `How to Use IDE Projects Under Version Control`.