Setting Project and File Options

This topic explains how to review and customize project-level and file-level options, including build settings, custom compiler and linker options, and other settings.

Sections include:

Reviewing and Modifying Settings

The appropriate build settings must be configured in order for C/C++test to properly test your code. Additionally, Advanced Settings (instrumentation options) can be configured as needed.

To review and modify settings:

1. Right-click the C/C++ Projects tree (a.k.a. "the project tree") node for the project or file whose settings you want to review and modify, then choose **Properties** from the shortcut menu. The Properties dialog will open.
2. Expand the **Parasoft> C++test** category in the left pane.
3. Select the category that represents the settings you want to review and/or change. Categories and available settings are described below.
   - Build Settings must be reviewed and then modified as needed. Reviewing other settings is optional.
4. Modify the options in the right pane.
5. Click **Apply**, then **OK**.

Available Settings

Build Settings

- **Options source**: Determines how options are set. Options can be set manually, or based on build system options, build data file options, Visual C++ 6.0 project options, Green Hills project options, or Eclipse Managed Make project settings.

  - **Use options from a build data file** is recommended if you used `cpptestscan` to create a build data file, as described inCreating a Project Using an Existing Build System, and if you prefer to manually manage regeneration and update the build data file.
  - **Use options from a build system** is recommended if 1) your project can be built from the command line, but you did not create a build data file or 2) if you created a build data file, but would prefer C++test to manage regeneration / update of the build data file.
  - **Use options from Microsoft Visual C++ 6.0 project** is recommended for projects that were originally developed in Microsoft Visual C++ 6.0.
  - **Use options from GHS project** is recommended for projects that were originally developed in Green Hills IDEs.
  - **Use options from the project** is recommended for Managed Make projects that were originally developed in Eclipse.
  - **Specify all options manually** is recommended otherwise.

The available controls depend on the **Options source** selected.

Using options from build system vs. options from build data file

Build system complexity and time required to perform a full project rebuild determine which options source is appropriate.

If **Use options from a build system** is enabled:

- C++test will automatically scan options from your build system
- C++test will perform full rebuild of the project each time build options have to be rescanned

If **Use options from a build data file** is enabled:

- You must manually generate or update the build data file (e.g. during your regular build process).
- You can can update the build data file while performing incremental builds.

Note

If you decide to manually manage a build data file, integrating the mechanism for creating build data files into your build process is highly recommended. This is so that members of the development team can access and use your build data file, as well as helps you seamlessly generate or update it during regular build activities.

Migrating from C++test 7.0—What about "Use options from Makefile-based project"?

In C++test 7.1 and above, **Use options from Makefile-based project** was replaced by **Use options from a build system**.

To achieve the same functionality as that provided by **Use options from Makefile-based project**, use the following settings

- Build command line: `[ make -i -B CXX=${CPPTEST_SCAN} CC=$ CPPTEST_SCAN} LD=${CPPTEST_SCAN} clean all ]`
- Build working directory: `[ ${project_loc} ]`
- Dependency file(s): `[ ${project_loc}/Makefile ]`

Managing Multiple C++test's Project Configurations

Multiple C++test Project Configurations can be used to manage tests for a range of different configurations. For example, you can configure a project for testing a host-compiler, cross-compiler, testing in Debug mode, testing in Release mode, etc.

You can also export current active properties to a file or import previously saved properties from a file:

1. From right-click your project in the project explore and click **Properties**.
2. Choose **Parasoft> C++test** and click **Export** or **Import**.
3. Browse for the properties and click **Save** or **Open**.

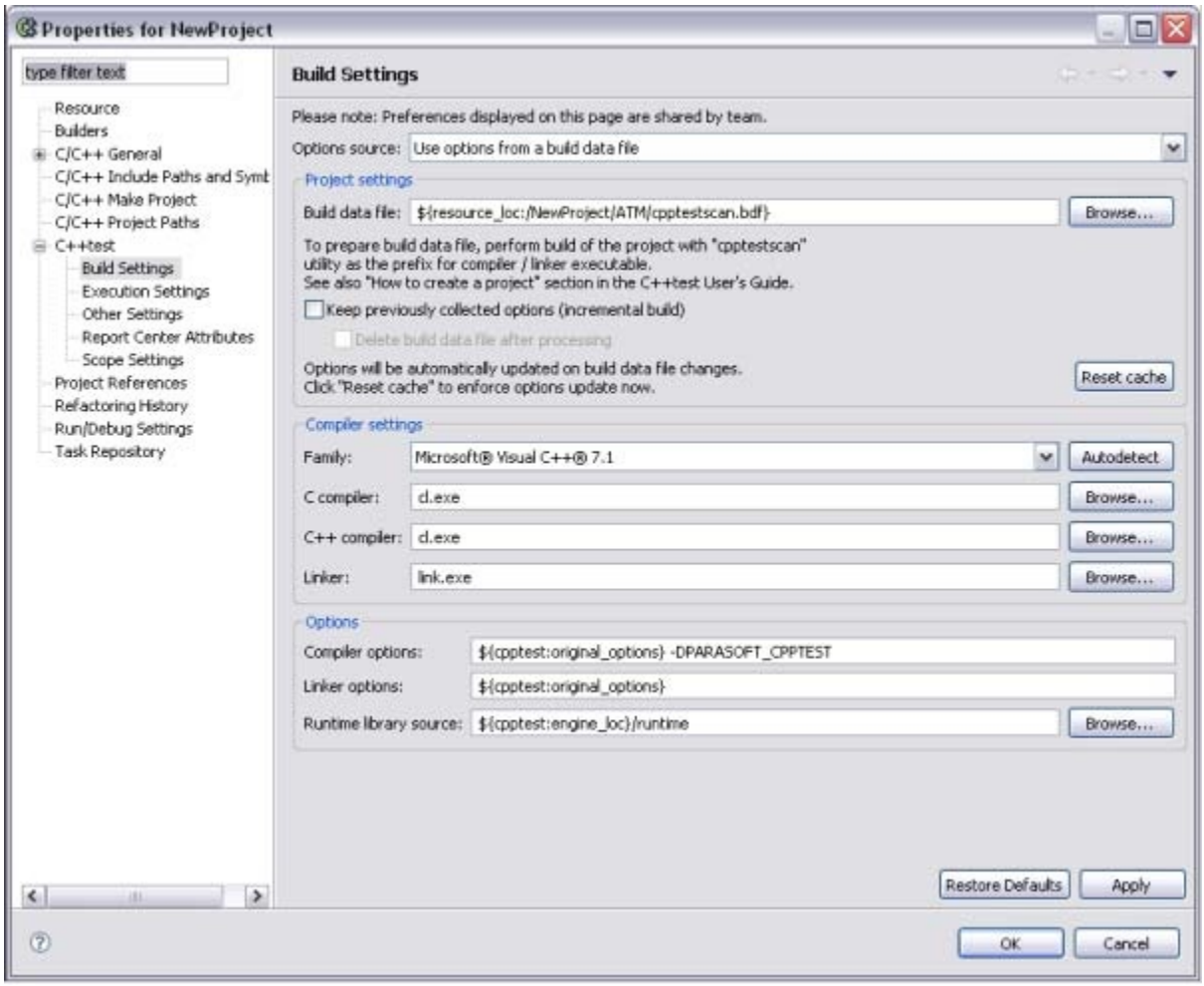
Exporting a properties creates a file that's can also be used as an argument in the `cpptestcli -localsettings` parameter. This allows you to easily switch between different project configurations when testing from the command line.

Use options from a build data file

To configure the appropriate settings:

1. In the **Build data file** field, enter or browse to the location of the build data file that was previously created (as described in Using cpptestscan or cpptesttrace to Create a Build Data File).
    - Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variable to specify the path:
        - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.
2. If you want C++test to clear previous options information each time the options are scanned, clear the **Keep previously collected options (incremental build)** check box. Otherwise, each build will be treated as an incremental build.
    - If you have an incremental build, select the **Delete build data file after processing** checkbox if you want C++test to remove the build data file after C++test scans options from it. This will ensure that build data file does not grow infinitely when subsequent builds are performed.
3. In the **Compiler settings** area, specify the compiler executable (for C and C++ sources), linker, and family.
    - If your compiler is not listed, add a custom compiler definition for it (as described in Configuring Testing with the Cross Compiler).
4. In the **Options** area, specify any test-specific C++test-specific options compiler or linker options that you want to use, and that are not already specified in the Makefile.
    - See Specifying Custom Compiler Settings and Linker Options for details.

When the options in the specified build data file change, C++test will automatically rescan the options and modify the project accordingly. If you want to force C++test to update options immediately (for example, if an external configuration file has changed), click the **Reset cache** button.



Use options from a build system

To configure the appropriate settings:

1. In the **Build command line** field, review the options that C++test will use to run the build (e.g., execute make with the specified makefile) and collect project build options from that process.
    - C++test replaces the compiler by reassigning values to the appropriate variables, as specified in the build command line.
    - The C++test options scanner is represented by the `${CPPTEST_SCAN}` variable.
2. If your build system uses compiler/linker variables that are different than the default ones provided with C++test, modify the build command line as needed.
    - C++test will scan options for all files that will be processed with the options scanner. You should provide an appropriate command line to force execution of the compiler for all source files. Otherwise, some of the files won't be tested.
    - Here is a sample GNU build command line:
      `make -i -B objects CXX=${CPPTEST_SCAN} CC=${CPPTEST_SCAN}`
      This command line will build the `objects` make target with the flags `-B` (make all targets unconditionally) and `-i` (ignore errors). The `CXX` and `CC` make variables will be substituted with `${CPPTEST_SCAN}`. This command line will allow C++test to extract options for all source files that are normally compiled to build the `objects` target.
    - If your command line shell processes $ in a special way (e.g., in bash), remember to escape the $ sign with \ (a backslash); for example, `CXX=\${CPPTEST_SCAN}`
    - C++test will support any build process (using nmake, ant, or arbitrary build scripts) as long as it can replace the compiler and linker by reassigning the appropriate variables. Consult your build engineer for appropriate options.

Important note about the -B Switch

The build command line is preset to include the -B switch option (unconditionally make all targets). Only GNU Make 3.80+ supports this option.

If you are using an earlier make, clean the build before creating a project. Keep the -k switch.

If you are using a make that does not support -B, further modification of the command line is required. The default build command line used in the project properties does not have any targets. We recommend that you modify this command line with the target(s) used to build the code you want analyzed For example:

`make -i CXX=\${CPPTEST_SCAN} ... all` (where `all` is a target).

Also, add a clean step prior to the normal build target. For example:

```
make -i CXX={CPPTEST_SCAN} ... clean all
```

This would first make the clean target, then `all` target. Although the make run from C++test will not actually build object files, the `clean` step will actually clean them.

3. In the **Build working directory** field, specify the directory in which the build process should execute.

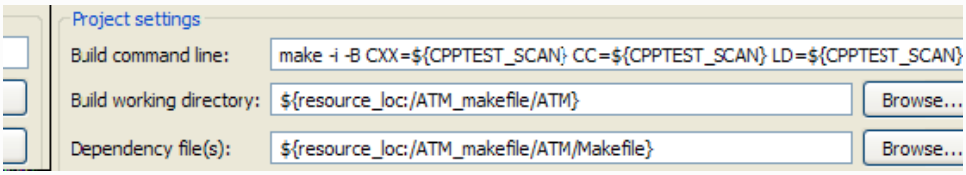
- Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variables to specify the path:
  - `${project_loc}` resolves to the absolute path to the Eclipse project location (the location that contains the `.project` file). *Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location—see the info box below for details.*
  - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.

Important note for projects with linked source folders

When configuring a Makefile-based project with a linked source folder, you cannot use `${project_loc}/linked_src_dir` as the build working directory. Why? Because there is no `linked_src_dir` subdirectory in the actual project location.

To properly configure the project—without using an absolute path to the source directory on your local system (this would not be portable across the team)—use `${resource_loc:/MyProject/linked_src_dir}`. This will resolve to the absolute path to the `linked_src_dir` directory, which is linked in to the `MyProject` project.

For example, assume that you are working on the ATM_makefile project, which contains a linked ATM folder that points to the source location. The actual project is stored in a different physical location (because you want to separate project setup files from the actual source). To configure C++test to locate the make run directory and any files referenced in relation to that, use `${resource_loc:<source>}` in the project settings. For this example, **Build working directory** is set to `${resource_loc:/ATM_Makefile/ATM}` and the **Dependency file(s)** is set to `${resource_loc:/ATM_Makefile/ATM/Makefile}`. The appropriate setup is shown in the following screen shot:



4. In the **Dependency file(s)** field, specify the Makefile(s) to be checked for changes. Options will be automatically updated if the specified file(s) changes.

- Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variables to specify the path:
  - `${project_loc}` resolves to the path to the Eclipseproject location (the location that contains the `.project` file). Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location—see the info box below for details.
  - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.

5. In the **Compiler settings** area, specify the compiler executable (for C and C++ sources), linker, and family.

- If your compiler is not listed, add a custom compiler definition for it (as described in Configuring Testing with the Cross Compiler).

6. In the **Options** area, specify any test-specific C++test-specific options compiler or linker options that you want to use, and that are not already specified in the Makefile.

- See Specifying Custom Compiler Settings and Linker Options for details.

When the options in the specified dependency file(s) change, C++test will automatically rescan the options and modify the project accordingly. If you want to force C++test to update options immediately (for example, if an external configuration file has changed), click the **Reset cache** button.

Use options from Microsoft Visual C++ 6.0 project

To configure the appropriate settings:

1. In the **Project file(.dsp)** field, enter the location of the Microsoft Visual C++ (`*.dsp`) project file.
    - Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variables to specify the path:
        - `${project_loc}` resolves to the path to the Eclipse project location (the location that contains the .project file). *Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location—see the info box above for details.*
        - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.
2. In the **Configuration field**, select the configuration that you want to use.
    - If you want to refresh the list of available configurations, click **Refresh**.
3. In the **Build variables** field, select the option that describes how you want C++test to configure `$INCLUDE`, `$LIB` and `$PATH` variables. Available options are:
    - **Use settings from environment first:** Configures C++test to append environment variables with values set up in Visual Studio.
    - **Use settings from Visual Studio first:** Configures C++test to append values set up in Visual Studio with environment settings.
    - **Use settings from environment only:** Configures C++test to use only values set up in the environment.
4. In the **Compiler settings** area, specify the compiler executable (for C and C++ sources), linker, and family.
5. In the **Options** area, specify any test-specific/C++test-specific options compiler or linker options that you want to use, and that are not already specified in the Visual Studio 6 configuration.
    - See Specifying Custom Compiler Settings and Linker Options for details.

When Visual C++ project file options change, C++test will automatically rescan the options and modify the project accordingly. If you want to force C++test to update options immediately (for example, if an external configuration file has changed), click the **Reset cache** button.

Use options from Green Hills Software (GHS) project

To configure the appropriate settings:

1. In the **Root project file (.gpj)** field, enter the location of the Green Hills (`*.gpj`) project file where you want to start options scanning.
    - Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variables to specify the path:
        - `${project_loc}` resolves to the path to the Eclipse project location (the location that contains the `.project` file). *Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location—see the info box above for details*.
        - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.
2. In the **Project** box, select a project, then click **Refresh**.
    - You can either select one of the available "testable" projects inside the GPJ project hierarchy (this may be the root itself), OR select the root node (even if it isn't testable) to use the compatibility mode, which is designed to support static analysis of "not test-able" projects.
    - See Additional Details on How C++test Uses a GHS Options Source for details on "testable" vs. "not testable" projects and about the compatibility mode.
3. In the **Compiler settings** area, specify the compiler executable (for C and C++ sources), linker, and family.
4. In the **Options** area, specify any test-specific/C++test-specific options compiler or linker options that you want to use, and that are not already specified in the project.
    - See Specifying Custom Compiler Settings and Linker Options for details.

When GHS project options change, C++test will automatically rescan the options. If you want to force C++test to update options immediately (for example, if an external configuration file has changed), click the **Reset cache** button.

Additional Details on How C++test Uses a GHS Options Source

C++test supports following kinds of GPJ projects, meaning that it treats project nodes designated with these project types as projects to scan for sources and build options:

- Project
- Subproject
- Program
- Library
- DLL
- Select One
- Shared Object
- Singleton Library
- INTEGRITY Application

C++test divides GPJ projects into two groups:

- **Testable**: Includes Program, Library and DLL. Artifacts of these project types are well known (executables and libraries), and C++test can build a Test Executable based on them. These are the only types of projects that can imported using the GHS Projects Importer.

- **Not testable**: All other supported project types can be used to introduce build options for all their sources and subprojects.

The root project is the top-level project file selected as the one from which options scanning starts.

For projects of complex structure:

- All first-level "not testable" projects in the logical projects' hierarchy (starting from the root) are treated as a kind of workspace, taking options introduced by them.
- Children of subprojects of testable projects are considered direct children of testable parents; options introduced by those subprojects are read.

Testable sources are C and C++ sources.

There are two kinds of build options introduced by GPJ projects:

- Compilation and linking options.
- Build properties (builder-only options) not passed directly to the compiler tools, but interpreted in a special way by the MULTI Builder itself.

For C++test, build options (compilation options and build properties) for a specific source file include all options collected over all parent project nodes leading to this source, together with all options of the source node itself. Moreover, linking options for a specific testable project include all options collected over all parent project nodes leading to this project, together with all options of the testable project node itself.

The following are recognized build properties, which may be specified from MULTI Build Options Editor:

- Source Directories Relative to This File" (:sourceDir)
- Source Directories Relative to Top-Level Project" (:sourceDirNonRelative)

For backwards compatibility with the GHS support available in previous C++test versions, the current version of C++test has a compatibility mode where the root project is always treated as testable, regardless of its type (as long as its type is supported). However, it may not be possible to build a test executable from a "not testable" project; two common reasons for this are:

- The scanned linking options aren't valid or sufficient.
- The underlying sources don't compose to the correct application (e.g., they contain duplicated global symbol names); this is mostly true for container-like projects holding several Program and/or Library nodes.

The compatibility mode is primarily used for running static analysis on all the testable sources found in the complete project's hierarchy.

Use options from Managed Make C/C++ project

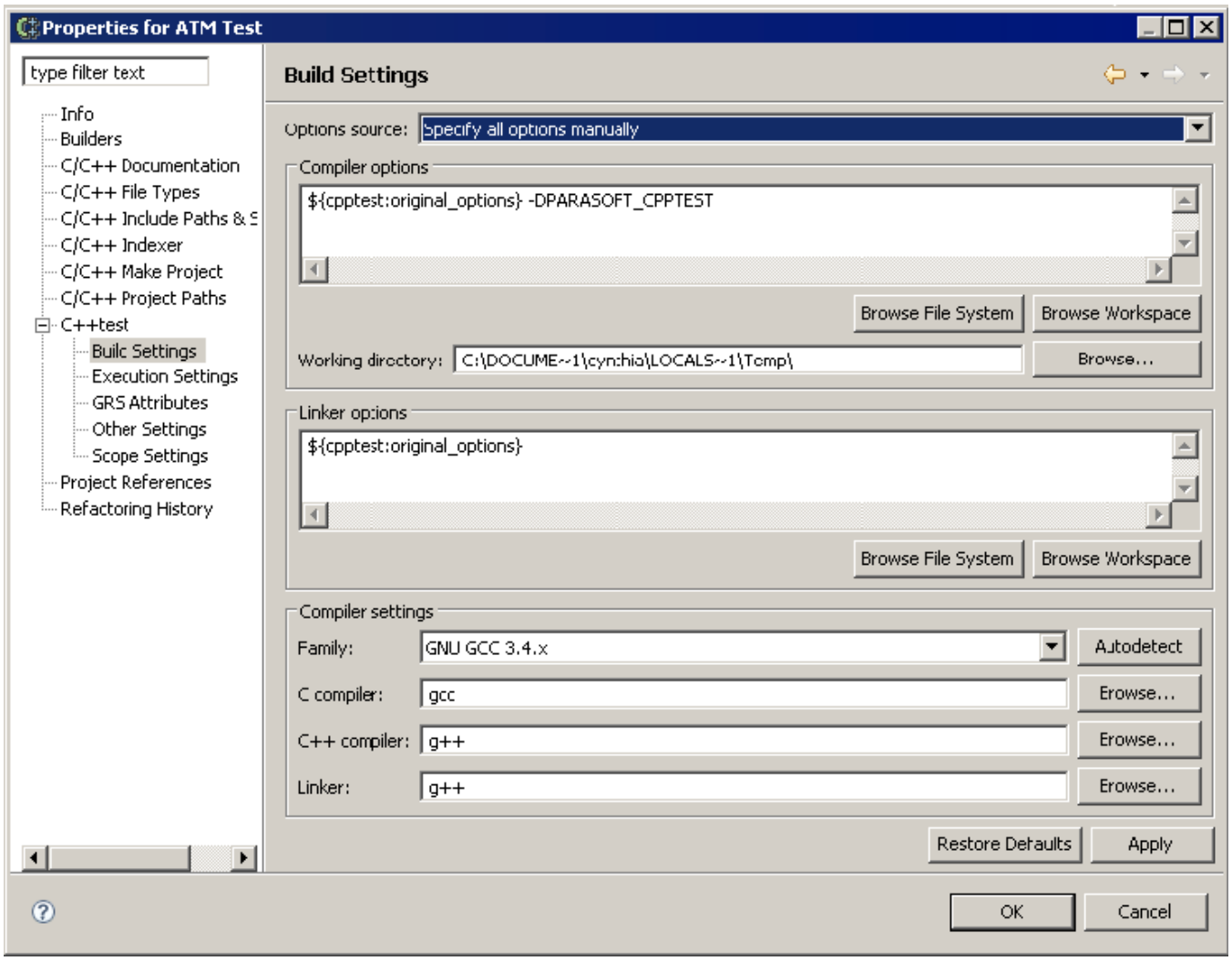
To configure the appropriate settings:

1. In the **Configurations** field, select the configuration that you want to use.
    - If you want to refresh the list of available configurations, click **Refresh**.
2. In the **Family** field, select the compiler family that your compiler belongs to.
3. In the **Use custom commands** area, specify any custom commands you want to use for the C compiler, C++ compiler, and/or linker.
4. In the **Options** area, specify any test-specific/C++test-specific options compiler or linker options that you want to use, and that are not already specified in the project (e.g., as a Make-file target or in a Visual Studio 6 configuration).
    - See Specifying Custom Compiler Settings and Linker Options for details.

Specify all options manually

To configure the appropriate settings:

1. In the **Compiler options** field, enter the compiler options you want to use.
2. In the **Working Directory** field, enter the directory where the compiler executes.
    - Using an absolute path could result in a non-portable project. We recommend that you use the following C++test variables to specify the path:
        - `${project_loc}` resolves to the path to the Eclipse project location (the location that contains the `.project` file). *Do not use this variable if your project has "linked" source folders—source files that do not actually live within your project location—see the info box above for details*.
        - `${resource_loc}` resolves to the path to the specific project resource. For instance, `${resource_loc:/MyProject/linked_src_dir/source.cpp}` will be resolved into `C:\src\source.cpp`.
3. In the **Linker options** field, enter the linker options you want to use.
    - If your compiler is not listed, add a custom compiler definition for it (as described in Configuring Testing with the Cross Compiler).
4. In the **Compiler settings** area, specify the compiler executable (for C and C++ sources), linker, and family. Compiler and linker settings should be consistent.
    - If your compiler is not listed, add a custom compiler definition for it (as described in Configuring Testing with the Cross Compiler).



Execution (File-Level Only)

Determines whether the selected source file is instrumented during test case execution.

In order to apply this setting to multiple files, select files in the project tree, then right-click the selection and choose **Parasoft> C++test> Execution Settings**. Note that the current setting will be applied uniformly to all selected files.

Other Settings

Allows you to specify the location for C++test's temporary files, as well as Advanced Options.

C++test Temporary Files

The **C++test temporary files** field controls where C++test keeps temporary data/files, such as instrumented sources, compiled objects, linked test executables, and automatically-generated header files. This setting controls `{cpptest:testware_loc}`.

Since C++test can automatically re-generate these files as needed, they do not need to be shared across team members. Thus, these files should not be saved within the project or added to source control.

The temporary data can be deleted by clicking the Clear button. We recommend that you do not delete the temporary data between test runs; keeping this data can significantly improve performance.

For optimal performance, use a location on the local hard drive.

Advanced Options

This table allows you to specify various low-level and debug options; for a list of available options, see Advanced Instrumentation Configuration Options.

Source File Encoding

Determines if multibyte encoding should be used when processing source files. By default, C++test automatically activates support for multibyte characters based on the current system encoding (**Auto** mode). In order to manually control multibyte support, set the option value to **On** or **Off**.

Note that performance may be impacted by analyzing code with support for multibyte characters enabled.

Scope Settings

Allows you to specify project files that you do not want tested. See Testing a User-Defined Set of Resources for details.