VisualAge® C++ Professional for AIX®

IBM

Debugging with the Integrated Development Environment

Version 5.0

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page v.

Edition Notice

This edition applies to Version 5.0 of IBM® VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Notices

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

References to VisualAge or VisualAge C++ in this book should be interpreted as VisualAge C++, Version 5.0.

Chapter 1. Debugging with VisualAge C++

You debug your programs in the VisualAge C++ Integrated Development Environment (IDE) in a substantially different way from the traditional edit/compile/link/debug methodology. The VisualAge IDE is designed to make the entire development process easier and more efficient. Debugging tasks are simplified in the following ways:

No Special Debugging Options

When you build your application within the IDE you do not need to specify special compiler or linker options for debugging. Because the IDE maintains a database or "codestore" of information about the functions, classes, variables, statements, and other information in your program, the IDE uses this codestore to access any required debugging information. The executable you debug can be the same executable you ship to customers. Because it does not contain debug information, it is more compact.

Batch compiled programs must still be compiled with debug information in order use the IDE debugger.

Breakpoints Can Be Set Any Time

You can set a breakpoint on any statement, function, virtual function, or class, before you start debugging. You can only set a breakpoint on a statement in functions that you build. As you add new capabilities to your program, you can set breakpoints in your new code so that the next time you debug your program, execution stops in your new code, allowing you to narrow down the scope of debugging.

Breakpoints are stored in your project in relation to your source code, not to executable addresses. If you add code to your program and rebuild it, the context of existing breakpoints is preserved. Each time you start debugging a process, the breakpoints you have set for the project are set at the appropriate addresses in the process. Once a process has started, you can disable or remove breakpoints for that process without affecting those breakpoints in other concurrent or subsequent debugging runs of the program.

Bugs Can Be Corrected from Within the Debug Pages

You can correct a bug in any source view within a debug page, and quickly rebuild your application. This causes all processes for the project that are running under debug control to terminate, but the time required to rebuild the application given a minor change is considerably less than with a traditional compiler, and from the time you enter the change to the time you start a new debug session for the application may be a matter of a few seconds or a minute, compared to much longer for the same change in a traditional environment.

For batch compiled programs, changes you make in the source code will not be picked up until the next time you compile your program.

You can debug multiple processes simultaneously

The IDE lets you start more than one process under debug control, and you can easily switch between running processes to compare, for example, how your application behaves given different inputs. The processes under debug control can also be different executables, which may help you to debug a distributed application.

You can run these multiple processes without cluttering your windowing workspace with countless windows; all debugging windows (other than the windows created by your application) are now views within the IDE, and are organized within pages under a process tab for each process you are debugging.

RELATED TASKS

"Start a Program Under Debug Control" on page 7
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Chapter 2. Prepare a program for debugging

Set Breakpoints Before Debugging

You can set most types of breakpoints in a project before you even start debugging, for example as soon as you have written the code for a function and have rebuilt the project. When you start a debug process for the project, these breakpoints affect that process. Note that an entry point breakpoint is automatically set at the entry point to the **main** function of your executable.

To set a breakpoint before debugging, do the following:

1. In the view containing the object (function, statement, class, or variable) you want to set a breakpoint on, click mouse button 2 over that object.
2. If a choice to set the appropriate breakpoint is shown on the resulting popup menu, select that choice. Otherwise:
3. Select the menu item that identifies the object; a cascading menu is displayed. If a breakpoint can be set for the object, a breakpoint or watchpoint menu choice appears. Select this.

Note that class instance breakpoints and watchpoints cannot be set before starting a debug session.

Each breakpoint is enabled when you set it, even when you have disabled the "All breakpoints" breakpoint in a Breakpoints view.

Further information on setting specific types is available in the Procedures help for each type of breakpoint.

RELATED CONCEPTS

"Chapter 1. Debugging with VisualAge C++" on page 1
"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from the Breakpoints View
"Set a Template Class Breakpoint" on page 15
"Set a Class Breakpoint" on page 16
"Set a Class Instance Breakpoint" on page 16
"Set a Function Breakpoint" on page 14
"Set a Statement Breakpoint" on page 12
"Set a Virtual Function Breakpoint" on page 15
"Set an Instruction Breakpoint" on page 17

Establish a debug environment

Specify Arguments to Your Program

You can specify arguments to a program you want to run or debug by entering those arguments in the **Arguments**entry field of the program's run specification before you start running or debugging the program. To set arguments for your program, do the following:

1. In the IDE, select the Debug page for your project.

2. Go to the Run Specifications view. If you do not see a Run Specifications view, you can set another project pane to show this view by clicking on the view name near the right hand side of the pane title bar, and selecting **Run Specifications**.
3. Select the Arguments entry field in the Run Specifications view by clicking mouse button 1 on it.
4. Enter your program arguments.

Now when you start running or debugging your program using this run specification, the arguments you entered are provided to your program.

If you want to set up several run specifications so that you can run or debug your program with different sets of arguments, click the **New** button to create a new, blank run specification for the project. If you want to duplicate an existing run specification, including its arguments, so that you can modify those arguments, select that run specification, click mouse button 2, and select **Duplicate**.

RELATED TASKS

"Start a Program Under Debug Control" on page 7

Write Programs for Debugging

You will find it easier to debug your programs if you follow these guidelines:

1. Do not do any hand-tuning for performance until you have all the logic in place and fully debugged.
2. Use meaningful names for loop counter variables and other temporary variables. This helps you quickly relate the variable names, when they appear in Stack frame or other Variables views, to the code to which they apply.
3. Use lvalues as array indices, rather than expressions that have a side effect, if you want to be able to determine what the value of an array index is at a given point. For example, statements such as a[i++]=b[j++]; make it more difficult for you to observe the effect of the statement on the affected variables and arrays. Instead, try using a compound statement:
   ```
   a[i]=b[j];
   i++;
   j++;
   ```
4. Break complex function calls into several simpler ones, as long as breaking up the call does not cause one function call's side effects to change the result of another function call. This makes it easier to step into or over the individual functions inolved, without having to switch to or open a disassembly view.

RELATED TASKS

"Debug Optimized Code" on page 5

Debugging Optimized Code

When you debug optimized code, information in debugger panes may lead you to suspect logic problems that do not actually exist. Consider the following points:

- Do not rely on panes such as stack frame panes to show the current values of variables. Numeric and char values may be kept in processor registers, as may pointers to other types of variables such as strings and class objects. In the optimized program, these values and pointers are not always written out to memory; in some cases, they may be discarded because they are not needed.

- Static or external variables can be monitored at function entry and exit points but their value as displayed by the debugger during debugging of a function may not be valid. The debugger displays the storage allocated to the static or external variable, but in optimized code, changes to a variable may not be written out to storage immediately.
- The register and storage panes are correct. Unlike panes that show actual variables, such as the stack frame, the registers and storage panes are always up-to-date as of the last time execution stopped.
- Use the Disassembly view of your program to see whether source statements whose result you were relying on have been eliminated during optimization (via dead code elimination, where code that performs no useful work is removed). You may find, for example, that an assignment to a variable in your source code does not result in any disassembly code being produced; this may indicate that the variable's value is never used after the assignment.

RELATED TASKS

"Debug Optimized Code"

Debug Optimized Code

Problems that only surface during optimization are often an indication of logic errors that are exposed by optimization; for example, using a variable that has not been initialized. If you encounter an error in your program that only occurs in the optimized version, you can usually find the cause of the error using a binary search technique to find the failing module:

1. Begin by optimizing half the modules and see if the error persists.
2. After each change in the number of optimized modules, if the error persists, optimize fewer modules; if the error goes away, optimize more modules. Eventually you will have narrowed the error down to a single module or a small number of modules.
3. Debug the failing module. If possible, turn off the instruction scheduling optimizations for that module. Look for problems such as reading from a variable before it has been written to, and pointers or array indices exceeding the bounds of storage allocated for the pointer or array.

See the Reference information below for tips on uncovering problems with optimized code you are debugging.

RELATED REFERENCES

"Debugging Optimized Code" on page 4

Chapter 3. Start or stop debugging a program

Start a Program Under Debug Control

From within your project, you can start debugging a program as follows:

1. If the project has not been built since you last made changes, rebuild the project.
2. Choose the Debug page of the project tab.
3. In the Run Specification view, select the run specification you want to debug. You can enter program arguments in the **arguments** entry field to pass to your program. You can select different environments for program to run in. You can create multiple run specifications for a project, for example, to specify different arguments to different versions.
4. Choose the environment that program will.
5. Click on the Debug button in the Run Specification view.

You can also start debugging a program by pressing CTRL+Shift+D at any time. The program will run using the most recently selected run specification. If you did not select a run specification, the program will run using the first run specification listed in the Run Specification view.

Either of these methods starts your program running in a new process under debug control. You can start multiple processes for the same program and debug them at the same time. Just return to the Debug page of the project tab, select the run specification you want from the Run Specification view, and click on Debug again. You can then switch between debugging one process and another by selecting the appropriate process tab.

If you are debugging multiple processes, the process toolbar buttons and Debug menu items that appear when you start debugging the first process apply to whichever process has its workbook section in the foreground, or to the process whose workbook section was most recently selected, if a different type of workbook section is the foreground section.

You can change which process the process toolbar and Debug menu items control by selecting a different process object in the Process Log view or the Debuggable Processes view in the Debug page.

RELATED TASKS

"Specify Arguments to Your Program" on page 3
Start Debugging an Already Running Process

Start Debugging a DLL or Shared Library

You can debug a DLL or Shared Library in the VisualAge C++ Integrated Development Environment the same way you debug executables. Follow these steps to set breakpoints in DLL or Shared Library functions so that execution stops when your DLL or Shared Library is called:

1. From the Debug page of the **Project** tab, start the program running under debug control.
2. Turn to the Modules page of the **Process** tab

3. Expand the DLL or Shared Library module you want to debug.
4. Expand the Exported Symbols section for the DLL or Shared Library.
5. For each function you want to set a breakpoint on, click mouse button 2 over the function name and choose **Set Function Breakpoint** from the popup menu.
6. If the DLL or Shared Library is not part of the current codestore, the exported symbols list does not include functions. Instead, a link symbol is shown for the mangled name of each function. To set an instruction breakpoint on the entry point to such a function, click mouse button 2 over the address shown for the function, and select **Set Instruction Breakpoint**.
7. If you do not want to stop in **main**, disable the **Entry point main** breakpoint in the Breakpoints view.
8. Run your program. Execution stops in the first DLL or Shared Library function called by your program.

RELATED TASKS

"Disable Breakpoints" on page 19
"Run a Program Under Debug Control" on page 25
"Set a Function Breakpoint" on page 14
"Set an Instruction Breakpoint" on page 17
"Start a Program Under Debug Control" on page 7

Chapter 4. Debug logic and program flow

View Disassembly Code for a Process

To view the disassembly code for the current execution point of a process under debug control:

1. Switch to the process tab for the process being debugged if you have not already done so, or start a new debug session.
2. Select the Disassembly page. By default this page includes a Disassembly view.

If you want a disassembly view from the Source page or another page that does not contain such a view, you can change the Source view on that page to a Disassembly view, and later you can change it back by selecting **Page** > **Reset Page** from the main menu. Or you can create a new pane, change the input object to the thread being debugged, and change its view to Disassembly.

RELATED CONCEPTS

"Step Commands: Overview" on page 21

RELATED TASKS

"Start a Program Under Debug Control" on page 7
"Step through Machine Language (Disassembly) Instructions" on page 24

RELATED REFERENCES

"Debugging Optimized Code" on page 4
"View Source Code for a Process" on page 10

Change Default Settings for the Disassembly View

You can change the default appearance of the Disassembly view from the View Settings page of the Workbook tab. Follow these steps:

1. Select the Workbook tab.
2. Select the View Settings page.
3. In the Views with Settings pane, select Disassembly. The righthand pane updates to show the settings you can change for the Disassembly view.
4. Change any settings here.

Changes you make to the defaults do not affect disassembly views that have already been created. For example, if you start a process under debug control, go to the Disassembly page for that process, then to the View Settings page of the Workbook tab to change the defaults for the Disassembly view, your changes do not affect the existing Disassembly page. However, if you start the process, then change the defaults, *then* view the Disassembly page for the process, the Disassembly view is created only then, and the new defaults take effect.

RELATED TASKS

"View Disassembly Code for a Process"

View Source Code for a Process

To view the source code for a thread and stack frame of a process running under debug control:

1. Switch to the process tab for the process being debugged if you have not already done so, or start a new debug session.
2. Select the Source page. By default this page includes a Source view.
3. If the thread whose source you want to see is not the current thread, select the thread from the Threads pane. If the function whose source you want to see is not on top of the stack frame, select the stack frame of the function whose source you wish to see.
4. If you want a Source view from the Disassembly page or another page that does not contain such a view, you can create a new pane containing that view, or change another pane to show that view. To do this, set the input object of the pane to the process object, then change the view to Source view.

RELATED CONCEPTS

"Source-Level Step Commands" on page 22

RELATED TASKS

"Start a Program Under Debug Control" on page 7
"Step through Source Code" on page 23

RELATED REFERENCES

"Debugging Optimized Code" on page 4
"View Disassembly Code for a Process" on page 9

Types of Breakpoints

You can set the following types of breakpoints in your program:

statement breakpoints
Breakpoints that apply to any source language statement that has executable code associated with it. These include compound statements such as **for** loop blocks. Statement breakpoints that apply to compound statements may be set at both the entry and exit points of the statement. If the compound statement is a constructor with an explicit initialization list, a breakpoint may be set on the initializer.

function breakpoints
Breakpoints that apply to the entry of a function. Execution stops on entry to the function each time it is called.

virtual function breakpoints
Breakpoints that apply to any virtual class method. When you set a virtual function breakpoint, the breakpoint applies to the selected function, and to any overrides of this function in any derived classes. Execution stops each time the method is invoked on a derived class object, whether or not the derived class redefines the virtual function. Execution also stops in methods invoked on instances of the class for which the breakpoint was set.

template class breakpoints
Breakpoints that apply to all methods of a template class. When you set a breakpoint on a template class, a function breakpoint is set for each

nonvirtual function of the template, and a virtual function breakpoint is set for each virtual function of the template. Breakpoints are also set for the functions in each instance of template.

class breakpoints
: Breakpoints that apply to all methods of a class. When you set a breakpoint on a class, a function breakpoint is set for each nonvirtual function of the class, and a virtual function breakpoint is set for each virtual function of the class. The virtual function breakpoints also apply to virtual function overrides in any derived classes.

class instance breakpoints
: Similar to class breakpoints, but a class instance breakpoint applies only to a particular instance of the class. For example, given the class Bicycle and the declaration of an instance of it, Bicycle ModelZ, if you set an instance breakpoint on the ModelZ instance, any time a Bicycle member function is called for the ModelZ object, a breakpoint is encountered. However, these member function breakpoints do not apply to other objects of class Bicycle.

instruction breakpoints
: Breakpoints that apply to a disassembly instruction. You set instruction breakpoints in a disassembly view. Execution stops when the instruction at the specified address is encountered.

Two additional breakpoints are automatically created and enabled for you. An **entry point breakpoint** applies to the start of the **main** function of your program. This breakpoint causes execution to stop in **main** when you start debugging your program. A **terminate breakpoint** applies to the termination of your program. This breakpoint causes execution to stop when the process ends (for example, through a return from **main** or a call to **exit**) but before the console window for the process has closed, if a console window exists.

When you remove, enable, or disable a breakpoint that has child breakpoints (for example, a class breakpoint, which has breakpoints set on each of its methods), the change affects the parent breakpoint as well as its children.

RELATED TASKS

"Set a Template Class Breakpoint" on page 15
"Set a Class Breakpoint" on page 16
"Set a Class Instance Breakpoint" on page 16
"Set a Function Breakpoint" on page 14
"Set an Instruction Breakpoint" on page 17
"Set a Statement Breakpoint" on page 12
"Set a Statement Entry Breakpoint" on page 13
"Set a Statement Initializer Breakpoint" on page 14
"Set a Statement Exit Breakpoint" on page 13
"Set a Virtual Function Breakpoint" on page 15
Enable, Disable, or Delete Breakpoints from a Breakpoints View

RELATED REFERENCES

Tree Structure of the Breakpoints View

Breakpoint Conditions

Every breakpoint, including watchpoints, can have a condition. This condition is an expression evaluated when the breakpoint is hit to produce a true or false result. If the expression is true or evaluation of the expression fails, execution of the program you are debugging halts. If the expression is false, the program you are debugging keeps executing.

Breakpoint conditions are inherited. For example, setting a condition on a class breakpoint sets the same condition on the all the child class instance breakpoints.

Setting conditions on a breakpoint overrides any conditions that breakpoint may have inherited.

When setting a condition on breakpoint that may have some ambiguity, like an entry poitn breakpoint, you must select a condition scope. The condition scope is the codestore that the breakpoint condition is set in. This prevents the condition from applying to the same breakpoint set in a different codestore.

There are two special variables which may be used in breakpoint condition expressions:

__hitCount
: This is the number of times the breakpoint has been encountered. This variable is always incremented **before** the condition is evaluated. **__hitCount** is not incremented if the breakpoint is disabled.

__threadId
: This is the ordinal number of the thread in which the breakpoint is being evaluated.

Note that both variables start with two (2) underscores. These variables are **integer** types.

RELATED CONCEPTS

"Types of Breakpoints" on page 10
Expressions Supported by the Debugger

RELATED TASKS

"Set Conditions on Breakpoints" on page 18
"Remove Conditions from Breakpoints" on page 20

Control breakpoints

Set a Statement Breakpoint

You can set a statement breakpoint from any view that displays source language statements, including source views and token stream views. To set a statement breakpoint, follow these steps:

1. Position the pointer over the statement you want to set a breakpoint on.
2. Click mouse button 2.
3. From the resulting popup menu, select **Set breakpoint**, if that choice appears. The statement breakpoint is now set.

4. If the **Set breakpoint** choice does not appear, select the Stmt object on the popup menu.
5. From the cascading menu that appears beside the Stmt object, select **Set statement breakpoint**.

If no **Set breakpoint** or **Set statement breakpoint** choice appears, or if an error message is displayed in the message area when you try to set the breakpoint, you may not be able to set a statement breakpoint on the statement. For example, you cannot set a statement breakpoint on a function declaration, because the function declaration is never executed.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

"Set a Statement Entry Breakpoint"
"Set a Statement Initializer Breakpoint" on page 14
"Set a Statement Exit Breakpoint"
Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Set a Statement Entry Breakpoint

You can set a statement entry breakpoint from any view that displays source language statements, including source views and token stream views.

To set a statement entry breakpoint:

1. Right-click on the part of the compound statement you want to set an entry breakpoint on.
2. Select **Set statement entry breakpoint** from the popup menu. The statement entry breakpoint is now set.

No **Set statement entry breakpoint** choice will be available if a statement entry breakpoint can not be set on the compound statement.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

"Set a Statement Initializer Breakpoint" on page 14
"Set a Statement Exit Breakpoint"
Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Set a Statement Exit Breakpoint

You can set a statement exit breakpoint from any view that displays source language statements, including source views and token stream views.

To set a statement exit breakpoint:

1. Position the pointer over the compound statement you want to set an exit breakpoint on.
2. Click mouse button 2.
3. From the resulting popup menu, select **Set statement exit breakpoint**. The statement breakpoint is now set.

No **Set statement exit breakpoint** choice will be available if a statement exit breakpoint can not be set on the compound statement.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

"Set a Statement Entry Breakpoint" on page 13
"Set a Statement Initializer Breakpoint"
Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Set a Statement Initializer Breakpoint

You can set a statement initializer breakpoint from any view that displays source language statements, including source views and token stream views. Statement initializer breakpoints only apply to compound statements containing an explicit initialization list.

To set a statement initializer breakpoint:

1. Position the pointer over the compound statement you want to set an initializer breakpoint on.
2. Click mouse button 2.
3. From the resulting popup menu, select **Set Statement Initializer Breakpoint**. The statement initializer breakpoint is now set.

No **Set initializer breakpoint** choice will be available if a statement initializer breakpoint can not be set on the compound statement.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

"Set a Statement Entry Breakpoint" on page 13
"Set a Statement Exit Breakpoint" on page 13
Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Set a Function Breakpoint

You can set a function breakpoint, so that execution stops whenever the function is called, from any view that displays function names, declarations, definitions, or statements containing function calls. Such views include declarations views, class details views, stack views, source views, and token stream views. To set a function breakpoint, follow these steps:

1. Position the pointer over the function you want to set a breakpoint on, or over the object that represents the function in that view.
2. Click mouse button 2.
3. From the resulting popup menu, select **Set function breakpoint**, if that choice appears. The function breakpoint is now set. (If the function is a class virtual function, choose **Set virtual function breakpoint**).
4. If the **Set function breakpoint** choice does not appear, select the F object on the popup menu.
5. From the cascading menu that appears beside the F object, select **Set function breakpoint**.

When you set a function breakpoint, a breakpoint marker is shown in the source view containing the function definition. The marker is not shown on declarations of the function.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View

Set a Virtual Function Breakpoint

You can set a virtual function breakpoint from any view that displays the virtual function as a declaration or definition, or in a statement, including declaration views, class details views, stack views, source views, and token stream views. When you set a virtual function breakpoint, a breakpoint is set both in the function in the class itself, and in any overrides of that function in derived classes. To set a virtual function breakpoint, follow these steps:

1. Right-click on the virtual function you want to set a breakpoint on, or the object that represents the virtual function in that view.
2. Select **Set virtual function breakpoint** from the popup menu, if that choice appears. The function breakpoint is now set.
3. If the **Set virtual function breakpoint** choice does not appear, select the F object on the popup menu. Then, select **Set virtual function breakpoint** from the casacading menu.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set a Function Breakpoint" on page 14

Set a Template Class Breakpoint

You can set a template class breakpoint from any view that displays class names, declarations, definitions, or declarations of objects of the template class, including declarations views, class details views, class hierarchy views, source views, and token stream views. When you set a template class breakpoint, a breakpoint is set for each method of the template class and each method of each instance of the template class , including constructors and the destructor. To set a template class breakpoint, follow these steps:

1. Position the pointer over thetemplate class you want to set a breakpoint on, or over the object that represents the template class in that view, and click mouse button 2.
2. Select **Set class breakpoint** from the popup menu, or from the object's cascading menu.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View

Set a Class Breakpoint

You can set a class breakpoint from any view that displays class names, declarations, definitions, or declarations of objects of the class, including declarations views, class details views, class hierarchy views, source views, and token stream views. When you set a class breakpoint, a breakpoint is set for each method of the class, including constructors and the destructor. To set a class breakpoint, follow these steps:

1. Right-click on the class you want to set a breakpoint on, or the object that represents the class in that view.
2. Select **Set class breakpoint** from the popup menu, or from the object's cascading menu.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View

Set a Class Instance Breakpoint

You can set an instance breakpoint on an instance of a class, so that execution stops each time a member function is called for that instance, from any view that displays the value of the class instance. This includes the local variables view and any details view of the instance. When you set an instance breakpoint, a breakpoint is set for each method of the instance's class, including constructors and the destructor, but the breakpoint only applies to that particular instance of the class (that is, the breakpoint only applies to the method when it is called for that particular instance).

To set an instance breakpoint from a **source view**, follow these steps:

1. Position the pointer over the class instance you want to set a breakpoint on, and click mouse button 2. Follow the object on the popup menu to the cascading menu, and select **Show Variable Value**.
2. In the resulting view of the class instance, click mouse button 2 over the address field.
3. From the resulting popup menu, select **Set Instance Breakpoint**.

If the instance is already visible in a **local variables view**, follow these steps:

1. In the local variables view, position the pointer over the ⊞ or ⊟ icon, or the equals sign (=), for the class instance .
2. From the resulting popup menu, select **Set Instance Breakpoint**.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View

Set an Instruction Breakpoint

You can set an instruction breakpoint (a breakpoint that is triggered when the instruction at a particular address is executed) from any view that displays that instruction address, including disassembly views and stack views. You can also set an instruction breakpoint on an arbitrary instruction address from any view that displays a process object.

To set an instruction breakpoint from a view that displays instruction addresses:

1. Right-click on the address of the instruction you want to set a breakpoint on. For example, in the disassembly view, right-click on an address in the first column of numbers; in a stack view, right-click on the entry point address for the module.
2. Select **Set instruction breakpoint** from the popup menu.

In a Stack view, you can set an instruction breakpoint on the point of execution address for modules for which there is only disassembly code available. If source code is available, you can only set a function breakpoint for the function in the stack frame.

In a Disassembly view, you can also set an instruction breakpoint by clicking on the small circle to the left of an instruction's address, or right-clicking on the small circle and selecting **Set instruction breakpoint** from the popup menu. The small circle changes to a large red octagon, indicating that the breakpoint has been set.

To set an instruction breakpoint on an arbitrary instruction address:

1. Right-click on the process object you want to set a breakpoint in.
2. Select **View Instruction Address** from the popup menu.
3. Enter an instruction address. A floating pane displaying the details of the instruction address appears. Addresses will be converted according the prefixes given to the address entered:
   - address starting with "0x" will be treated as a hexadecimal address
   - addresses starting with "0b" will be treated as a binary address
   - addresses starting with "0" will be treated as an octal address
   - all other addresses will be treated as a binary address
4. Expand the Disassembly section of the Details view.
5. Right-click on the address you want to set a breakpoint on.
6. Select **Set Instruction Breakpoint** from the popup menu.

RELATED CONCEPTS

"Types of Breakpoints" on page 10

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View
"Set a Function Breakpoint" on page 14

Set Conditions on Breakpoints

You can change the effect a breakpoint has on the execution of your program during debugging by setting a condition on the breakpoint.

To set a condition on a breakpoint:

1. Obtain a details view for the breakpoint you want to set a condition on.
2. Enter a C++ expression supported by the debugger in the **Condition** field.
3. If available, select the codestore from the **Condition scope** field. This prevents the condition from applying to the same breakpoint set in a different codestore.

If the condition is valid, two new lines are added to the details view of the breakpoint : **Condition Owner** and **Expression**.

Condition Owner indicates the breakpoint from which the condition is inherited. If the breakpoint does not inherit any conditions, the owner is "self".

Expression shows the expression you just entered.

If the breakpoint is one that will be triggered when you are debugging your program, a **Last Evaluation** line will appear. This field will say either "Unevaluated", "true", or "false", depending on the result of the conditions.

RELATED CONCEPTS

"Breakpoint Conditions" on page 12
Expressions Supported by the Debugger

RELATED TASKS

Enable, Disable, or Delete Breakpoints from a Breakpoints View
Set Breakpoints Within Debug Pages
"Set Breakpoints Before Debugging" on page 3
"Remove Conditions from Breakpoints" on page 20

Enable Breakpoints That Have Been Disabled

You enable breakpoints that have been disabled in the source view, the token stream view, the disassembly view, and the breakpoints view. You can enable groups of breakpoints, or *all* breakpoints, at once.

To enable an disabled breakpoint in a source view, click on the breakpoint marker. The breakpoint marker normally appears as green highlighting behind a token in a source view. Do not double-click on the breakpoint marker. Double-clicking on the breakpoint marker will delete the breakpoint.

To disable a breakpoint that is displayed in a token stream or disassembly view, click on the green octagon that represents the breakpoint.

To disable a breakpoint from the breakpoints view:

1. Expand the breakpoint tree in the breakpoints view until you see the disabled breakpoint you want to enable. Disabled breakpoints are shown with a green octagon.
2. Click on the green octagon. Its color changes to red to indicate that the breakpoint has been disabled.

If you want to enable *all* breakpoints, or a group of breakpoints (for example, all the method breakpoints that were automatically set when a class breakpoint was set), click on the green octagon for the parent breakpoint for the group in the breakpoints view, and enable that breakpoint. If that breakpoint is already enabled, disable it and then enable it.

RELATED TASKS

"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages
"Disable Breakpoints"

Disable Breakpoints

You can disable breakpoints so that execution does not stop when the breakpoint is encountered. This is useful if you want to temporarily suspend the use of a breakpoint, but you may need to use the breakpoint later. You can disable breakpoints in the source view, the token stream view, the disassembly view, and the breakpoints view.

To disable an enabled breakpoint in a source view, click on the breakpoint marker. The breakpoint marker normally appears as red highlighting behind a token in a source view. Do not double-click on the breakpoint marker. Double-clicking on the breakpoint marker will delete the breakpoint.

To disable a breakpoint that is displayed in a token stream or disassembly view, click on the red octagon that represents the breakpoint.

To disable a breakpoint from the breakpoints view:

1. Expand the breakpoint tree in the breakpoints view until you see the enabled breakpoint you want to disable. Enabled breakpoints are shown with a red octagon.
2. Click on the red octagon. Its color changes to green to indicate that the breakpoint has been disabled.

If you want to disable *all* breakpoints, or a group of breakpoints (for example, all the method breakpoints that were automatically set when a class breakpoint was set), click on the red octagon for the parent breakpoint for the group in the breakpoints view, and disable that breakpoint. If that breakpoint is already disabled, enable it and then disable it again.

RELATED TASKS

"Enable Breakpoints That Have Been Disabled" on page 18
"Remove Breakpoints"

Remove Breakpoints

Note: You may want to consider disabling a breakpoint, rather than removing it. When you disable a breakpoint it has the same effect as removing the breakpoint temporarily. You can enable it later from the Breakpoints view with less effort than it takes to set a new breakpoint at the same location.

You can remove a breakpoint from any view that displays the object the breakpoint applies to, including the source view, the token stream view, the disassembly view, and the breakpoints view. You can remove groups of breakpoints, or *all* breakpoints, at once.

To remove a breakpoint from a Source view, follow these steps:

1. Find the breakpoint marker in the source view. This marker is shown as a red or green background behind the first letter of a statement. A red marker indicates an enabled breakpoint; a green marker indicates a disabled breakpoint.
2. Double-click on the breakpoint marker. If you only click on the marker, you will only toggle the breakpoint's state between enabled and disabled.

To remove a breakpoint from a Disassembly or Token Stream view, follow these steps:

1. Find the breakpoint marker for the breakpoint you want to remove. This marker is a red or green circle to the left of the line of disassembly code or token stream information.
2. Click mouse button 2 over this marker.
3. Select the Bp object from the popup menu.
4. Select **Remove Breakpoint** from the cascading menu.

To remove a breakpoint from the Breakpoints view, follow these steps:

1. Expand the breakpoint tree in the breakpoints view until you see the breakpoint you want to remove.
2. Click mouse button 2 over the breakpoint.
3. Select Remove Breakpoint.

To remove *all* breakpoints, follow these steps:

1. Go to a Breakpoints view.
2. Select the parent "All Breakpoints" breakpoint, and delete that breakpoint.

RELATED TASKS

"Disable Breakpoints" on page 19

Remove Conditions from Breakpoints

To remove a condition from a breakpoint:

1. Obtain a details view for the breakpoint you want to remove the condition from.
2. Clear the expression in the **Condition** field.

The condition on the breakpoint will be replaced by an inherited condition or no condition if there are no conditions to inherit.

RELATED CONCEPTS

"Breakpoint Conditions" on page 12

RELATED TASKS

"Set Conditions on Breakpoints" on page 18

Step Commands: Overview

Each step command operates either at the source statement level, or at the machine (or assembly code) level. Once you know what step command you want to use, select the appropriate one from the Debug menu or by using the appropriate toolbar button on the debugger toolbar.

Here is a brief summary of available step commands:

step into
: Executes the current statement. If the current statement contains a function call, execution stops at the first statement or instruction of the first called function in the statement. Otherwise, execution stops at the statement that logically follows the current statement within the same function.

step over
: Executes the current statement. Execution stops at the statement that logically follows the current statement within the same function, or in the calling function if the statement you step over is a return statement. Function calls within the statement are stepped over and execution does not stop within them, unless the debugger encounters a breakpoint within a called function.

step debug
: Executes from the current statement up to the next statement for which debugging information is available. This statement may be in the current function, or in any function called while the statement is executed. For example, if the statement stepped over is a call to func1, where func1 calls func2 and func2 calls func3, and debug information is only available for func3, execution would stop in the first statement of func3. Use step debug instead of step into when you want to step into only those functions that have source code views available.
Step debug has a time-out feature. If a function with debug information is not found in approximately 20 seconds, a step over command is performed instead of a step debug command.

machine step into
: Executes the current machine language instruction. If the current instruction is a call, execution stops at the first instruction within the called function or routine. Otherwise, execution stops at the instruction that logically follows the current instruction.

machine step over
: Executes the current machine language instruction. Execution stops at the instruction that logically follows the current instruction. Calls within the statement are stepped over and execution does not stop within them, unless the debugger encounters a breakpoint within the called code.

return step
: Executes from the current statement or instruction, through the return from

the current function. Execution stops at the statement or instruction following the call to the function that you just returned from. If the view on return is a source view, execution may stop at the same statement that issued the original call, if that statement contains multiple function calls. You can use a combination of Step into and Step return commands to step into each function in a statement containing multiple function calls.

Note that, for all step commands except machine step into, execution stops before the command completes if an enabled breakpoint is encountered.

If a function terminates on a signal or exception when stepping through function calls, you may not always stop at the expected point.

RELATED TASKS

Step over a Function
"Step through Machine Language (Disassembly) Instructions" on page 24

RELATED REFERENCES

"Machine Step Into Command" on page 41
"Machine Step Over Command" on page 41

Source-Level Step Commands

If the statements you want to step through do not contain any function calls, the following three step commands work in exactly the same way: they execute the current statement, and execution stops at the statement that logically follows the current statement:

- "Step Into Command" on page 42
- "Step Over Command" on page 42
- "Debug Step Into Command" on page 41

The three commands work differently for statements that *do* contain function calls. See "Step over a Function" below for information on the differences.

You can use the fourth step command, "Return Step Command" on page 43, to run through the remainder of the current function. Execution stops at the source code statement that logically follows the statement that called the function you are currently in. If the function you are currently in was called as part of a complex statement (for example, the compound function call func1(func2(),obj->func3());), execution may stop within the compound statement, if only part of the statement was executed.

RELATED CONCEPTS

"Step Commands: Overview" on page 21

RELATED TASKS

Step over a Function

RELATED REFERENCES

"Step Into Command" on page 42
Step Over Command
"Debug Step Into Command" on page 41
"Return Step Command" on page 43

Jump to Location

You can use Jump to Location commands to jump over a section of code that may be causing problems, without executing that code. However, jumping over implicit function calls such as constructors or destructors may produce umpredictable results.

Consider the following code excerpt:

```
MyType a=func1();
MyType b=func2();
a+=b;
if (a.mem==True) {
   // ...
} else {
   // ...
}
```

Suppose you have noted that the addition of object b to object a is resulting in the execution of the `if` block, whereas you expected the else block to execute. You suspect that the statement a+=b; may be in error (or perhaps that the operator+= you have defined is not working as expected). To test your theory, you want to see whether skipping over the addition of b to a solves the problem. This is called jumping over, because the code you jump over does not get executed. To jump over the statement a+=b;do the following:

1. Step over the definitions and assignments for a and b.
2. Click mouse button 2 over the if statement. A popup menu appears.
3. Select the Stmt object, and, from the cascading menu, **Jump to This Statement**.

The if statement is now the next statement to be executed, but the statement a+=b; was not executed. From here you can determine whether the suspect statement was indeed causing the problem.

RELATED CONCEPTS

"Step Commands: Overview" on page 21

RELATED TASKS

"Jump over Code without Executing It" on page 26
Step over a Function

Step through, run, or halt a program

Step through Source Code

You can step through source code, one statement at a time, by using **Step commands**. (A related set of debugger commands, called **Machine step commands**, let you step through assembly language or machine code one instruction at a time).

To use a source step command, do **one** of the following:

- Click on one of the source step buttons in the process toolbar. See the References section below for information on these buttons.
- Select a source step command from the Debug title bar menu.
- Click mouse button 2 over the process tab for the process you want, and select a source step command from the popup menu.

If you have multiple processes running under debug control, source step commands issued from the process toolbar or the Debug title bar menu apply to the debuggable process whose process tab is the foreground tab, or, if a different type of tab is the foreground tab, to the process you most recently selected.

RELATED CONCEPTS

"Source-Level Step Commands" on page 22
"Step Commands: Overview" on page 21

RELATED REFERENCES

"Step Over Command" on page 42
"Step Into Command" on page 42
"Debug Step Into Command" on page 41
"Return Step Command" on page 43

RELATED TASKS

"Step through Machine Language (Disassembly) Instructions"

Step through Machine Language (Disassembly) Instructions

You can step through machine language or disassembly code, one instruction at a time, by using machine step commands. (A related set of debugger commands, source-code-based step commands, let you step through source code one statement at a time). Note that you can use machine step commands even when no disassembly view is open for the process running under debug control. When you machine step through a process, and a source view is visible, you may notice that it takes several machine step commands for the current source statement to completely execute.

To use a machine step command, do **one** of the following:

- Click on one of the machine step buttons in the process toolbar. See the References section below for information on these buttons.
- Select a machine step command from the Debug title bar menu.
- Right-click the process tab for the process you want, and select a machine step command from the popup menu.

If you have multiple processes running under debug control, machine step commands issued from the process toolbar or the Debug title bar menu apply to the debuggable process whose process tab is the foreground tab, or, if a different type of tab is the foreground tab, to the process you most recently selected.

RELATED CONCEPTS

"Step Commands: Overview" on page 21

RELATED REFERENCES

"Machine Step Over Command" on page 41
"Machine Step Into Command" on page 41

RELATED TASKS

"Step through Source Code" on page 23

Step Over or Into Functions

You can choose from among several options when debugging function calls:

- You can step over all function calls. In this case, use **Step Over** commands each time you want to step over a function call in the current statement within your program.
- You can step into all function calls. In this case, use **Step Into** commands each time you want to step into a function call in the current statement within your program. If you step into a function that does not have source code information, you must bring up a disassembly view of the code if you want to see what is being stepped through. If you inadvertently step into a function that does not have source code, issue a Step Return command to return to the calling code.
- You can step into only those functions that contain source code information. In this case, use **Step Debug** commands each time you want to step within your program. Execution stops at the next source statement encountered, whether that statement is in the same function, a called function, or a function called by the called function.

RELATED TASKS

"Run to the End of the Current Function, Then Return"
"Step through Source Code" on page 23
Step Over a Function That Has No Source Information

Run to the End of the Current Function, Then Return

You can run to the end of the current function, return to the calling function, and have execution stop automatically, by issuing a **Return Step** command. To issue this command, do **one** of the following:

- Select **Return Step** from the Debug title bar menu..
- Click on the **Return Step** button in the process toolbar.
- Click mouse button 2 over the process tab for the process you want, and select **Return Step** from the popup menu.

If you have multiple processes running under debug control, a Return step command issued from the process toolbar or the Debug title bar menu applies to the debuggable process whose process tab is the foreground tab, or, if a different type of tab is the foreground tab, to the process you most recently selected.

Execution stops at the machine instruction right after the instruction that called the current function. In a source view, execution may appear to stop at the statement following the call, or within the statement that issued the call, if the statement contains multiple function calls.

RELATED TASKS

"Step Over or Into Functions"

Run a Program Under Debug Control

To run your program, you must first have started up a debug process within which your program will run. You will probably also want to set breakpoints before you run, otherwise your program will simply run to completion or an exception.

When you are ready to run your program, you can issue a run command in any one of the following ways:

- Choose Debug->Run from the title bar menu.
- Click on the Run toolbar button
- Click mouse button 2 over the process in the Debuggable Processes view, or over the process tab, and select Run from the resulting popup menu.

RELATED TASKS

"Start a Program Under Debug Control" on page 7
"Halt Execution of a Process"

Jump over Code without Executing It

To jump over a section or block of code without executing it, you must first execute up to but excluding the start of the block, in one of the following ways:

- Select the first statement you want to jump over and issue the Run to this statement command.
- Set a breakpoint at the first statement you want to jump over, and issue the Run command.
- Step over the statements up to but excluding the first statement you want to jump over.

Once the current execution point is the first statement of the block you want to jump over, do the following:

1. Right-click on the statement you want to jump *to*. This is the first statement that logically follows the block of code you want to jump over. This statement should be in the same function as the current execution point, otherwise you may corrupt the stack of your debug process.
2. Select the Stmt object, and from its cascading menu, select **Jump to This Statement**.

RELATED CONCEPTS

"Jump to Location" on page 23

RELATED TASKS

Run to a Specific Statement or Instruction

Halt Execution of a Process

To halt execution of a process running under debug control, so that you can debug it, follow these steps:

1. If the process you want to halt is one of several running under debug control, select the process tab for the process you want to halt.
2. Select the stop button on the process toolbar, or click mouse button 2 over the process tab and select stop.
3. If the process still does not halt, it may be waiting for console input. Switch to the console window, and enter the required input. Switch back to the IDE and issue a Debug Step Into command. Execution then stops at the source code statement following the one that requested input.

You can also set breakpoints in your program so that execution stops at predetermined instruction addresses or statements, when specific functions are called, specific storage is written to, and so on.

RELATED TASKS

"Terminate a Program Under Debug Control"
"Set Breakpoints Before Debugging" on page 3
Set Breakpoints Within Debug Pages

Terminate a Program Under Debug Control

To terminate a process running under debug control, do *one* of the following:

- Select Debug->Terminate from the titlebar menu.
- Click the Terminate button in the process toolbar.
- Click mouse button 2 on the process tab for the process you want to halt, and select Terminate from the resulting popup menu. Note that if you *remove* the process tab without terminating the process, the process remains, and you can later attach a new process tab to it from the Debuggable Processes pane of the Project section of the workbook.

If you have multiple processes running under debug control, a Terminate command issued from the process toolbar or the Debug title bar menu applies to the debuggable process whose process tab is the foreground tab, or, if a different type of tab is the foreground tab, to the process you most recently selected.

If you think your program has stalled in an endless loop, and you want to debug it, do not terminate the program, halt its execution instead and use step commands to determine where the program is stalling..

RELATED TASKS

"Halt Execution of a Process" on page 26

Debug Child Processes

When you start debugging a program, you can set an option from the Run Specifications view so that a process tab opens for any child process started by the program. You can then switch between the process tab for the parent process, and those for any child processes, and view information for each process. However, you cannot use step or run commands in the parent process if the child process causes the parent process to suspend, for example when using **system**, or **_spawnl** with a MODEFLAG argument of PWAIT. Follow these steps to debug child processes:

1. In the Run Specifications view, select Debug child processes.
2. Click the Debug button.
3. Run the parent process.
4. A child process tab opens for each child process created. You can debug a child process (provided it is not suspended) by selecting its tab.

Each time the parent process starts a child process, a breakpoint is set on entry to the **main** function of the child process, or, if the **main** function cannot be found, a breakpoint is set at the entry point for the child process. If the child process is an application created by the IDE and a codestore is available for it, all breakpoints contained in the codestore are also set for it. Execution stops at the first encountered breakpoint.

If the child process was created in suspend mode, step and run commands in the thread of the parent process that created the child process have no effect until the child process terminates.

RELATED CONCEPTS

AIX Fork Handling

AIX Exec Handling

RELATED TASKS

"Start a Program Under Debug Control" on page 7

Debug Exception-Handling Code

You can debug C++ exception-handling code that you wrote.

To debug exception-handling code that you wrote (code inside a C++ **catch** block), follow these steps:

1. Start your program under debug control and turn on signal handling for user-defined exceptions.
2. Run your program. When a user-defined exception occurs, execution stops and a floating pane opens showing the process log, with the type of exception thrown as the last object in the process log.
3. Issue a Step Exception command. This causes the code leading up to your exception handler to be executed.
4. Issue a Debug Step Into command. Execution stops in the first source code statement for the **catch** block that caught the exception. You can now debug your exception-handling code.

RELATED TASKS

"Handle Signals for User-Defined Exceptions" on page 29
"Determine Causes of System Exceptions"

Determine Causes of System Exceptions

When an exception is raised in your program by operating system or runtime library functions, you can use the Step Exception command (not availabe on AIX) to step into the exception-handling code for that exception. You cannot view source for this exception-handling code, which makes debugging it difficult. It is usually easier to determine which source statement caused the exception, set a breakpoint on that statement, and run the program again under debug control to examine the statement and likely causes of the exception.

Follow these steps to determine which source statement caused an exception:

1. Disable the "All Breakpoints" breakpoint in the Breakpoints view for the project.
2. Find the Stack view in the Source page of the Process tab for the process that caused the exception.
3. In the Stack view, look down the list of stack frames to the first one containing the name of a function within your program. Set a breakpoint on this function.
4. Terminate the process, start another debug process, and run it. Make sure that you provide the same program arguments and input as before. Execution should stop when the function you set the breakpoint on is called. Use Step Over commands to move through the function, until the exception is thrown.

5. Set a breakpoint on the statement causing the exception, and remove the function breakpoint you set earlier.
6. Terminate the process, start another debug process, and run it. Execution stops at the statement that previously caused the exception. From here you can examine the context of the problem statement, to determine possible reasons for the error. For example, you may be copying a string to a null pointer or past the allocated bounds of an array.

RELATED TASKS

"Handle Signals for User-Defined Exceptions"
"Debug Exception-Handling Code" on page 28
"Set a Function Breakpoint" on page 14
"Set a Statement Breakpoint" on page 12

Handle Signals for User-Defined Exceptions

If you want to debug your own C++ exception-handling code, you can enable signal handling for user-defined exceptions, so that whenever an exception is thrown, execution stops before your code handles the exception. Follow these steps to enable signal handling for user-defined exceptions:

1. Start debugging your program. A process workbook section appears and becomes the foreground section. Execution stops at the start of the **main** function.
2. Select the Details page for the process.
3. Expand the ⊞ icon for the **signals** object. The first signal, SIGUSR1, is the signal for user-defined exceptions. If the octagon to its right is green, click on it to turn it red. This enables the signal for user-defined exceptions.

If your exception-handling code occurs in only a few places, you can also force execution to stop whenever an exception is thrown by setting a statement breakpoint at the start of each **catch** block.

RELATED TASKS

"Debug Exception-Handling Code" on page 28
"Determine Causes of System Exceptions" on page 28

Chapter 5. Debug variable contents and memory

Expression and Value Monitors

The Monitored Expressions view and the Monitored Values view can both be used to display the contents of variables. Consider the following points when choosing where to monitor the variable. You can always place the variable in both views.

- The Monitored Expressions view displays type and name information about each variable, but not the variable's address. The Monitored Values view shows type and the address of the variable, but not the variable's name.
- The Monitored Expressions view displays the actual value of the variable at all times. If the variable passes out of scope, its contents are displayed as "Not in scope". The Monitored Values view displays the storage that was allocated for the variable. If the variable passes out of scope, its contents continue to be displayed, even though the storage shown may no longer be allocated to the variable and may, over time, become allocated to another variable.
- The Monitored Expressions view keeps track of the expressions you add to it, even between builds and between different debug processes. The Monitored Values view is cleared for each debug process, because it shows physical addresses.

The Monitored Values view is useful for viewing the contents of objects allocated on the heap, particularly objects that are not easily accessible via an expression. For example, consider the case where you step into a class method. You can add the **this** pointer to the Monitored Expressions view to see its contents, but as soon as you return from the method, the expression is no longer valid, so no value is shown. But the storage that was pointed to by the **this** pointer is still valid, and you may still want to monitor the object. By adding the object to the Monitored Values view, you can keep track of the object contents even when there is no pointer name that explicitly and consistently points to it.

RELATED TASKS

"Add a Variable to the Monitored Expressions View" on page 32
"Add a Value to the Monitored Values View" on page 35
"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable" on page 32

Debug local variables

Obtain a Local Variables View

To obtain a Local Variables view, do the following:

1. Go to the Source page of the Process tab for the process running under debug control
2. In the Stack view, select the function on the stack frame whose variables you want to see. If the function has been recursively called, (it appears several times in the Stack view) be sure to select the correct stack frame for the function.
3. If a Local variables view is still not displayed, change the view type in the Stack Frame view to a Local Variables view.

The Local Variables view has two sections, and you can collapse or expand each section. The first section is for function parameters, and the second is for local variables defined within the function.

You cannot obtain a Local Variables view for a function that is not on the stack, because a stack frame is only allocated for these local variables when the function is entered, and this stack frame is freed on return.

RELATED TASKS

"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable"

Obtain a Details View of a Variable

To obtain a Details view of a variable, follow these steps:

1. From any view that displays the name of the variable, right-click on the variable name. A popup menu appears.
2. If the resulting popup menu shows a object for the variable, click on that object.
3. Select **Show Variable Value**.

RELATED TASKS

"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable"

Add a Variable to the Monitored Expressions View

To add a variable to the Monitored Expressions view, follow these steps:

1. From any view that displays the name of the variable, right-click on the variable name. A popup menu appears.
2. If the resulting popup menu shows a object for the variable, click on that object.
3. Select **Add to Expression Monitor**.

RELATED TASKS

"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable"
"Add a Value to the Monitored Values View" on page 35

RELATED CONCEPTS

"Expression and Value Monitors" on page 31

Change the Representation of a Variable

You can change the representation of a variable in any view that lets you edit the variable's contents, for example in a Local Variables view or in a Monitored Expressions view. Follow these steps to change a variable's representation in such a view:

1. Click mouse button 1 on the arrow to the right of the variable's contents.
2. Select one of the available representations from the list.

For compound variables such as class objects and arrays, you can expand the variable (object or array) to display components, and then change the representation of individual components.

You can change the representation of a class between flattened (in which the members of all inherited classes are shown, without the information for their class) and nested (in which each inherited class is shown as a separate, expandable entry within the class).

You can change the representation of a pointer to show a pointer, the object pointed to, or to show an array of objects. The representation of pointers to char and wchar can also be changed to show a string.

RELATED TASKS

"Add a Variable to the Monitored Expressions View" on page 32
"Add a Value to the Monitored Values View" on page 35
"Change the Contents of a Variable"

RELATED REFERENCES

"Appendix A. Available Representations for Variables" on page 39

Change the Contents of a Variable

To change the contents of a variable, the variable must first be visible in a view that displays the variable's contents:

- The Local Variables view
- The Monitored Expressions view
- The Monitored Values view
- A floating pane for the variable's value

If no such view is available, click mouse button 1 on the variable, select the V object for the variable from the popup menu, and select **Show Variable Value** or **Add to Expression Monitor** from the cascading menu.

Follow these steps once the variable is visible in such a view:

1. If the variable's contents are not visible, click on the ⊞ icon beside the variable's name to display its contents.
2. Click mouse button 1 over the variable's contents.
3. Edit the current contents, or delete them and enter a new value. For numeric values, the value you enter must indicate its representation according to C++ language rules. For example, precede a hexadecimal value with 0x0. For a variable of type char, enter the character in single quotes. For a character array, enter the string *without* its enclosing double quotes. For a member of an enumeration, you can enter the enumeration name.
4. Press Enter or select a different pane with the mouse to update. If the value you entered was invalid, a warning beep sounds and the original value is restored.

RELATED CONCEPTS

Expressions Supported by the Debugger

RELATED TASKS

"Add a Variable to the Monitored Expressions View" on page 32

Debug Global Variables

To debug a global variable, you must first add the global variable to a Monitored Expressions view or to a floating pane that shows the variable's runtime value.

To add a global variable to a floating pane, follow these steps:

1. Right-click on the variable.
2. Select the ☑ object representing the variable from the popup menu
3. Select **Show Variable Value** from the cascading menu.

To add a global variable to a Monitored Expressions view, follow these steps:

1. Right-click on the variable.
2. Select the ☑ object representing the variable from the popup menu.
3. Select **Add to Expression Monitor** from the cascading menu select

Once the global variable is visible in a floating pane or a Monitored Expressions view, you can edit its contents or change its representation. If the variable is a compound variable such as a class object or an array, you can also change the level of detail shown for the variable, by clicking on the ⊞ expander by the variable name and any expanders of subparts.

RELATED TASKS

"Add a Variable to the Monitored Expressions View" on page 32
"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable" on page 32

Debug program storage

View a Different Address in a Storage View

To change the address being viewed in a storage view, do *one of* the following:

- Enter a new address in a line of the storage view:
  1. Click on an address (an entry in the leftmost block of the storage view)
  2. The current address is selected. You can enter a new address, or you can use the cursor keys to deselect the current address and edit it. The new address is interpreted using the number system set for the column. For example, if you enter a hexadecimal value, but the number system indicated is decimal, the address will be considered invalid.
  3. Press Enter. If the address you enter is valid, the Storage view updates to show the new location.
- Click on the scroll keys on the left of the storage view. The single arrows scroll storage up or down one row at a time. The double arrows scroll storage up or down one screenful at a time.

You can also change the address being viewed to an address pointed to within the current view, by chasing that address.

RELATED TASKS

Chase an Address in Storage
Find Memory Allocation (Heap) Problems

Change the Representation of Storage in a Storage View

To change the representation of storage in a Storage view, do the following:

1. Click on the arrow beside the block heading for a block in the view.
2. From the expandable/collapsible list that displays, choose the desired characteristics of the representation. For example, you can change the base of the number system, and you can set the number of bytes (size of units) for the column..

RELATED TASKS

Chase an Address in Storage
"View a Different Address in a Storage View" on page 34

Change the Contents of Program Storage

To change the contents of storage for a process from a Storage view, do the following:

1. Make sure the range of storage you want to change is displayed in the view. See the related topics below for help on scrolling through storage or choosing a new storage location to view.
2. Position the pointer on the line of storage you want to change and click mouse button 1.
3. If you press the left or right cursor keys before you enter text, you can edit the current text. If you start typing without using the cursor keys, the current text is replaced by what you enter.
4. When you have made the desired change, press Enter, select another object in the view, or make a different view the foreground view.

If the storage you tried to change was writeable by the process being debugged, and the change you entered was a valid expression for the current representation in the storage view, the storage is updated. Otherwise an error message displays on the status line.

RELATED CONCEPTS

Expressions Supported by the Debugger

RELATED TASKS

"Change the Representation of Storage in a Storage View"
"Check Heap for Corruption After Each Step Command" on page 37
"View a Different Address in a Storage View" on page 34
Chase an Address in Storage
Find Memory Allocation (Heap) Problems

Add a Value to the Monitored Values View

To add a value to the Monitored Values view, you must first view the value in a view that shows its contents, for example in the Monitored Expressions view or in the Local Variables view.

Once the value's contents are visible, follow these steps:

1. Click mouse button 2 over the value.

2. Select Add to Value Monitor.

Use the Monitored Values view when you want to display the storage for a particular variable, pointer, class member, or expression, even after the expression that yielded that value has changed. Use the Monitored Expressions view when you want to view the result of an expression, and the result of the expression may change over time.

RELATED TASKS

"Change the Contents of a Variable" on page 33
"Change the Representation of a Variable" on page 32
"Add a Variable to the Monitored Expressions View" on page 32

RELATED CONCEPTS

"Expression and Value Monitors" on page 31

Debug memory allocation (heap) problems

Set Incorporation Options for Heap Debugging

If you want to debug your program's use of the heap (storage you allocate and deallocate explicitly using **new**, **delete**, and C Library functions such as **malloc** and **free**), you may want the heap to be automatically checked after each step command, breakpoint encountered, or halt command. If so, you need to set heap checking options before the module you want to debug is incorporated. These options cause heap debugging versions of the memory management functions to be used, so that heap usage can be tracked and heap usage errors found.

To set the appropriate options for incremental compiler:

1. Select the Configuration section of your project.
2. Select either the Options or the Subproject Options page of the Configuration section.
3. In the Options or Subproject Options view, expand the **C++ Compiler Options** entry if it is not already expanded.
4. Expand the **Miscellaneous Options** entry if it is not already expanded.
5. Expand the **Common C/C++ Miscellaneous options** entry.
6. Under Common C/C++ Miscellaneous Options, turn on the **Enable debug version of memory management functions** setting.
7. Reincorporate your project.

To set the appropriate options for the batch compiler, specify the heap debug option of the batch compiler.

You can also set this option by using Live Find from within the Options or Subproject Options page. To use Live Find, select the pane the view is in, then press Ctrl+F.

RELATED TASKS

"Check Heap for Corruption After Each Step Command" on page 37
Find Memory Allocation (Heap) Problems

Check Heap for Corruption After Each Step Command

You can check the heap for corruption after each step or run to command if you have built your program with heap checking enabled. The check can be performed manually or automatically.

Checking the heap may take a non-trivial amount of time. Consider your method of heap checking carefully if you are going to be stepping through your program.

To check the heap for corruption manually:

1. Select **Debug > Check the Heap** from the title bar menu. The heap status appears in the message area.

To start checking the heap for corruption automatically:

1. Select **Debug > Enable Automatic Heap Checking**. The heap status message will be updated after each step or run to command.

To stop automatic heap checking:

1. Select **Debug > Disable Automatic Heap Checking**.

RELATED TASKS

"Set Incorporation Options for Heap Debugging" on page 36
"Change the Representation of Storage in a Storage View" on page 35
Chase an Address in Storage
Find Memory Allocation (Heap) Problems
"View a Different Address in a Storage View" on page 34

Debug processor registers

Choose Which Registers Display in a Registers View

You can change the default appearance of a Registers view by expanding or collapsing branches of the Registers view tree. For example, you may not want to see floating-point registers if your program does not perform floating-point arithmetic. Expanding or collapsing branches in one Registers view does not affect other Register views.

The Registers view displays processor registers and flags in groups or branches. You can collapse a branch of the list by clicking on the minus (⊟) to the left of the topmost entry for that branch. This ⊟ then becomes a ⊞ . You can expand a collapsed branch by clicking on the ⊞ .

RELATED TASKS

"Change the Contents of Registers" on page 38
"Change the Representation of Register Contents"

Change the Representation of Register Contents

To change the representation of a register's contents, do the following:

1. Make sure the register is displayed in a Registers view. You may need to expand a branch of the Registers view tree to view the register.
2. Click mouse button 1 on the arrow to the right of the register contents.
3. Choose the desired representation from the list of available representations.

RELATED TASKS

"Change the Contents of Registers"
"Choose Which Registers Display in a Registers View" on page 37

Change the Contents of Registers

To change the contents of a register in the Registers view, do the following:

1. If the register you want to alter is not visible in the view, expand the appropriate category of registers until the register is displayed.
2. Click on the contents of the register.
3. Enter the new contents and press Enter.

The value you enter must be a valid C++ representation of a hexadecimal, decimal, octal, binary, or character value. For example, to enter a hexadecimal value, precede the value with 0x0.

RELATED TASKS

"Change the Representation of Register Contents" on page 37
"Choose Which Registers Display in a Registers View" on page 37

Appendix A. Available Representations for Variables

You can change the representation of a variable in any view that lets you edit the variable’s contents. The following list provides details on available representations for different categories of variables:

Numeric types
: You can change the number system (binary, octal, decimal, hexadecimal) used to represent numeric types, and you can change between fixed and scientific notation.

Pointers
: You can dereference the pointer or dereference the pointer as an array. In addition, the representation of pointers to char and wchar can be changed to show a string.

Classes
: You can change the representation of a class between flattened (in which the members of all inherited classes are shown, without the information for their class) and nested (in which each inherited class is shown as a separate, expandable entry within the class).

Enumerations
: You can change the representation of enumeration members to show their integer values or their names.

RELATED TASKS

“Change the Contents of a Variable” on page 33
“Change the Representation of a Variable” on page 32

Appendix B. Debugger commands

Machine Step Into Command

The Machine step into command steps through disassembly instructions in the current function, *and* into any called code. If you issue a Machine step into from a source view, and the function stepped into does not have source code available, the source view will become blank. Use a Step return command to return to the source view, or change the view to a disassembly view to see the current execution point.

RELATED CONCEPTS

“Step Commands: Overview” on page 21
“Source-Level Step Commands” on page 22

RELATED TASKS

Step Into a Function
“Step through Machine Language (Disassembly) Instructions” on page 24
“View Disassembly Code for a Process” on page 9
“View Source Code for a Process” on page 10

RELATED REFERENCES

“Step Into Command” on page 42

Machine Step Over Command

The Machine step over command steps through disassembly instructions in the current function, without stepping into any called code.

RELATED CONCEPTS

“Step Commands: Overview” on page 21
“Source-Level Step Commands” on page 22

RELATED TASKS

Step Over a Function
“Step through Machine Language (Disassembly) Instructions” on page 24
“View Disassembly Code for a Process” on page 9
“View Source Code for a Process” on page 10

RELATED REFERENCES

“Step Over Command” on page 42

Debug Step Into Command

The Debug step into command steps through statements in the current function, *and* into any called functions, *provided* such called functions have debug information available (that is, can be debugged at the source code level by VisualAge C++). After each Debug step into command, execution stops at the first point the debugger encounters that matches one of the following criteria:

- The statement that logically follows the statement from which the Debug step into command was issued
- The first debuggable statement encountered within a called function, after the Debug step into command was issued
- The first enabled breakpoint.

Use **Debug step into** to debug your own code without wasting time stepping through machine language code in modules that you did not develop.

For shared libraries, the Debug step into command will be treated as a Step over command.

RELATED CONCEPTS

"Step Commands: Overview" on page 21
"Source-Level Step Commands" on page 22

RELATED TASKS

Step Over a Function That Has No Source Information

Step Into Command

The Step Into command executes the current statement. If the current statement contains a function call, execution stops at the first statement or instruction of the first called function in the statement. Otherwise, execution stops at the statement that logically follows the current statement within the same function.

RELATED CONCEPTS

"Step Commands: Overview" on page 21
"Source-Level Step Commands" on page 22

RELATED TASKS

Step Into a Function

RELATED REFERENCES

"Machine Step Into Command" on page 41

Step Over Command

The Step over command executes the current statement. Execution stops at the statement that logically follows the current statement within the same function, or in the calling function if the statement you step over is a return statement. Function calls within the statement are stepped over and execution does not stop within them, unless the debugger encounters a breakpoint within a called function.

AIX

Stepping over a call to `exec()`results in the process stopping at the entry point of the new process image, if one is loaded.

RELATED CONCEPTS

"Step Commands: Overview" on page 21
"Source-Level Step Commands" on page 22

RELATED TASKS

Step Over a Function
Step Over a Function That Has No Source Information

RELATED REFERENCES

"Machine Step Over Command" on page 41

Return Step Command

The Return step command executes from the current statement or instruction, through the point at which the current function returns to its calling function. Execution stops at the statement or instruction following the call to the function that you just returned from. If the view on return is a source view, execution may stop at the same statement that issued the original call, if that statement contains multiple function calls. You can use a combination of Step into and Step return commands to step into each function in a statement containing multiple function calls.

RELATED CONCEPTS

"Step Commands: Overview" on page 21

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Getting Started book, or our PDF documents. Send your e-mail to: torrcf@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

http://www.ibm.com/support/ describes IBM Support Offerings on all platforms, worldwide.

http://www.ibm.com/rs6000/support/ describes support offerings on the RS/6000® platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

http://www.lotus.com/passport describes the IBM and Lotus® Passport Advantage™ contracting option.

The IBM Software Support Handbook, accessible from http://www.ibm.com/software/support, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- **United States**: 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- **Canada**: 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

- Elsewhere, please contact your local IBM office.

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred

- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere™ Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit http://www.ibm.com/software/ad/vaws-services/ or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com