

C/C++ Productivity Tools for OS/390



# Getting Started with C/C++ Productivity Tools for OS/390

*Release 10*



C/C++ Productivity Tools for OS/390



# Getting Started with C/C++ Productivity Tools for OS/390

*Release 10*

**Note**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 39.

**First Edition (September 1999)**

This edition applies to C/C++ Productivity Tools for OS/390 Release 1.0, program number 5655-B85 and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable system bibliography for current information on these products.

Order publications through your IBM representative or through the IBM branch office serving your locality.

© Copyright International Business Machines Corporation 1999. All rights reserved.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>About this book.</b> . . . . .	<b>v</b>	Stopping on a line only if a condition is true . . . . .	19
Who should read this book . . . . .	v	Viewing and modifying data members of the this pointer . . . . .	20
Conventions used in this book . . . . .	v	Debugging when only a few parts are compiled with TEST . . . . .	21
Related information . . . . .	vi	Displaying storage . . . . .	21
How to send your comments. . . . .	vi	Finding unexpected storage overwrite errors . . . . .	22
 <b>Chapter 1. Introducing the Tools</b> . . . . .	<b>1</b>	Finding uninitialized storage errors . . . . .	23
Introducing the Editor . . . . .	1	Getting a function traceback . . . . .	24
Introducing the Distributed Debugger. . . . .	2	 <b>Chapter 4. Analyzing program performance</b> . . . . .	<b>25</b>
Introducing the Performance Analyzer . . . . .	3	Analyzing a program's performance . . . . .	25
 <b>Chapter 2. Editing programs</b> . . . . .	<b>5</b>	Creating a trace file . . . . .	25
Editing source. . . . .	5	Starting the Performance Analyzer . . . . .	26
Starting the Editor . . . . .	5	Analyzing a trace file . . . . .	26
Creating a new file and entering source . . . . .	5	Searching for trace data in a diagram . . . . .	27
Marking and copying a block of code . . . . .	6	Controlling what data is shown in a diagram. . . . .	29
Finding and replacing a class name . . . . .	9	Navigating the trace file views . . . . .	30
Setting and finding quick marks . . . . .	11	 <b>Appendix. Sample Program.</b> . . . . .	<b>33</b>
Fixing compile-time errors . . . . .	12	Sample CALC Program . . . . .	33
 <b>Chapter 3. Debugging programs</b> . . . . .	<b>15</b>	 <b>Notices</b> . . . . .	<b>39</b>
Debugging a program . . . . .	15	Trademarks and service marks . . . . .	41
Starting the Distributed Debugger user interface daemon. . . . .	15		
Starting a program with Debug Tool in OS/390 UNIX. . . . .	16		
Setting a breakpoint to halt before a certain function is entered . . . . .	17		
Displaying and modifying the value of a variable . . . . .	18		



---

## About this book

*Getting Started* introduces you to C/C++ Productivity Tools for OS/390 and provides information about how to edit, debug and analyze the performance of an OS/390 C and C++ application.

---

## Who should read this book

*Getting Started* is intended for application programmers who want to maintain and develop C and C++ applications on OS/390 and want workstation development tools to enhance their existing familiar host environment. For these users, this document introduces the tools and shows how to use them.

This document does not cover installation and configuration information for C/C++ Productivity Tools workstation components or host components Performance Analyzer and Debug Tool. See the *Readme* file for installation information for C/C++ Productivity Tools workstation components. See the *Program Directory* and *Memo to Users* for information on the C/C++ Productivity Tools host components.

This document also does not cover installation and configuration information for C/C++ Productivity Tools prerequisite products OS/390 and OS/390 C/C++ with Debug Tool. See the documentation with these products for this information.

---

## Conventions used in this book

The following conventions distinguish different text styles within this book:

plain	Window titles, folder names, icon names, and method names.
monospace	Programming examples, user input at the command line prompt or into an entry field, directory paths.
<b>bold</b>	Menu choices and menu names, labels for push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combination-boxes, notebook tabs, and entry fields.
<i>italics</i>	Programming keywords and variables, and titles of documents.

---

## Related information

For information on OS/390 C/C++ related features, news and Web sites, add this Web site to your browser's bookmark list:

<http://www.ibm.com/software/ad/c390>

---

## How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other C/C++ Productivity Tools documentation, send your comments by e-mail to [torrcf@ca.ibm.com](mailto:torrcf@ca.ibm.com). Be sure to include the name of the book, the document number of the book, the version of C/C++ Productivity Tools, and, if applicable, the specific location of the information on which you are commenting (for example, a page number or a table number).



---

# Chapter 1. Introducing the Tools

---

## Introducing the Editor

C/C++ Productivity Tools provides a simple to use, yet powerful, Editor that you can use to edit host C and C++ source files that are available on a local workstation drive. The features provided by this workstation Editor include:

- Insert, delete, split and join line, and find
- Line, character, stream, and rectangular (block) selection modes
- Ability to edit and browse multiple files simultaneously
- Multiple views of the same file
- Full clipboard support for cut and paste
- Unlimited Undo and Redo actions
- Bookmark settings to allow for quick returns to specific places in the file
- History of recently opened files for quick reloading
- Capability to import or get files
- A keystroke recorder for automating repetitive editing tasks
- A compare function for graphically highlighting the differences between two selected files
- Hex editing capability
- Various Editor personalities (configurations) to support line command and function key settings including ISPF
- Completely tailorable key assignments and line commands to customize the Editor's look and feel
- Language unique and product unique features include:
  - Syntax checking for C, C++, and High Level Assembler for early identification for lexical errors
  - Parsed source showing each language construct in various colors and fonts
  - Online language-sensitive reference help for C, C++, and High Level Assembler which is a single press of a key (F1) away
  - View subsetting by function, comment lines, or flow-of-control statements

---

## Introducing the Distributed Debugger

With C/C++ Productivity Tools' Distributed Debugger, you can debug your C and C++ applications that are running in your host OS/390 environment from the workstation. Distributed Debugger works with the IBM Debug Tool to allow for the source-level debugging of your host C and C++ applications.

The Distributed Debugger's intuitive graphical user interface brings debugging to a new level of ease. With a click of your mouse, you can add and delete both simple and complex breakpoints. You can set these breakpoints so that program execution halts when a specific line number is reached, upon entry to a specific function, when a specific module is loaded, or when a particular location in storage is changed. You can display and edit the values of variables, registers, and storage. You can also monitor the call stack.

All of these actions, and more, can be accomplished from the convenience of your workstation. Distributed Debugger allows you to view storage areas in a number of useful formats. Default views are provided for OS/390 control blocks such as the Dynamic Storage Area (DSA) and Prefixed Save Area (PSA). Control blocks are represented as a tree structure with storage displayed as floats, integers and longs. This facility makes reading storage much more intuitive and errors easier to spot.

Distributed Debugger supports debugging of C and C++ applications in any of these environments:

- TSO
- Batch
- CICS
- IMS
- DB2
- OS/390 UNIX System Services (OS/390 UNIX)
- Webserver

You can debug both multi threaded and multi process applications with the choice of following either the parent or the child processes. You can debug multiple applications from the same user interface. Each new application becomes a new tab.

The efficiency of debugging is one of the more important aspects of application development. Distributed Debugger in combination with Debug Tool makes that aspect of application development and maintenance much more productive and easier than ever before.

---

## Introducing the Performance Analyzer

You can use C/C++ Productivity Tools' Performance Analyzer on your workstation to graphically display a trace of the execution of your host OS/390 C and C++ application. Using this understanding of your program, you can tune your code for increased performance.

Analyzing the performance of your application with the Performance Analyzer is a two-stage process. First, you use the host Performance Analyzer component included with C/C++ Productivity Tools to create a function-by-function trace of an execution of your host application. Then, you make the trace data available to the workstation through a file transfer facility such as FTP or NFS. You then use the Performance Analyzer to graphically display the execution trace file.

During the function tracing of your program, the host component of the Performance Analyzer collects and records all trace data including:

- The functions that each function calls to identify who calls whom
- A count of the number of times each function is called to indicate which functions are being called most frequently
- The time spent to execute each function to identify costly or time consuming functions.

You can use the Performance Analyzer to:

### **Diagnose program abends**

When performing a function trace, the Performance Analyzer provides a complete history of events leading up to the point where a program abends.

### **Trace multithreaded programs**

After tracing a multi threaded program, you can examine the individual threads to identify their function usage and compare the execution of different threads.

### **Trace multiple processes**

When your POSIX program uses the `fork()` or `spawn()` functions to create new processes, separate trace files are created for each process allowing you to view the events in the different processes.

Performance Analyzer allows you to display the information gathered in the trace file through several diagrams on your workstation. Each diagram presents a different view to give you an overall idea of how your program performs:

- The Call Nesting diagram shows the execution of your program as a series of function calls and returns. All threads can be shown at once or you can select the threads to be shown.
- The Dynamic Call Graph diagram is a two-dimensional graphical representation of your program's execution. It shows the relative importance (in terms of execution time) of program components and the call hierarchy.
- The Execution Density diagram shows trace data chronologically from top to bottom as colored horizontal lines in columns assigned to each traced function.
- The Statistics diagram is a textual report of cumulative information about your program's execution. It provides summary and detailed statistics on execution time and events for each component type: function, class and executable. In addition to time statistics, information about the number of calls is also provided.
- The Time Line diagram shows function calls and returns in chronological order along a vertical line. A function call is represented by a short horizontal line to the right, and a function return is represented by a short horizontal line to the left. The horizontal lines are connected by vertical lines whose length is proportional to the amount of time that elapsed between the respective events.

Performance Analyzer helps you gain an understanding of your program's behavior. With this knowledge, you can not only tune your code for increased performance but also accomplish the following:

- Show logic flow (useful for C++ constructors and destructors)
- Identify potential functions to make inline
- Determine which functions of a DLL are being called
- Track library calls
- Verify built-in function usage
- Track function calls among threads
- Track class interaction
- Track module interaction.

---

## Chapter 2. Editing programs

---

### Editing source

The following sections identify typical tasks you might want to perform while using the Editor with a C and C++ program and explains how to accomplish these tasks. The “Sample CALC Program” on page 33 is used to demonstrate these actions. Follow the tasks in sequence as the tasks are often related to each other.

For explanation of a pane or dialog box, click the pane or dialog box and press F1.

1. “Starting the Editor”
2. “Creating a new file and entering source”
3. “Marking and copying a block of code” on page 6
4. “Finding and replacing a class name” on page 9
5. “Setting and finding quick marks” on page 11
6. “Fixing compile-time errors” on page 12

---

### Starting the Editor

In this section, you learn how to start the Editor. You can start the Editor in any of the following ways:

- By double-clicking on its icon from a program folder
- By selecting **IBM C and C++ Productivity Tools for OS 390 > Editor** from the Windows **Start** menu
- By entering the Editor command `iedit` on a Productivity Tools command line.

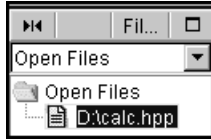
---

### Creating a new file and entering source

In this section, you learn how to create a new file for the CALC program. The first new file you create is `calc.hpp`. Once you learn how to create an empty file with the type `.hpp`, you then learn how to enter the source for this new file including how to undo typing and check each line you enter.

1. Select **File > New file** from the Editor menu bar. An untitled document appears in the Editor Pane. You need to save this untitled document with the `.hpp` extension to indicate that the file has the type `.hpp`.

2. Select **File > Save file**. Type `calc.hpp` in the **File name** field and click **Save**. The File Selector refreshes to show the file `calc.hpp` on a local drive.



3. To enter code, position the cursor on the Editor pane and begin typing. Type in the source for `calc.hpp`. When you have completed a line of code, press the Enter key. The line of code is parsed by the C/C++ parser. A blank line is inserted after the current line. The cursor moves to the first column position of this new line.
4. If you make a typing error, use the Backspace key to go back and correct the error. If you type over, for example, a variable name but decide that you want to use the first variable name that you entered, select **Undo** from the **Edit** menu. When you are done, the top of your Editing pane will look as follows:

```
/*-----File calc.hpp-----  
/*  
/* Header file for CALC.CPP PUSHPOP.CPP READTKN.CPP  
/* a simple calculator  
/*-----  
typedef enum toks {  
    T_INTEGER,  
    T_PLUS,  
    T_TIMES,  
    T_MINUS,  
    T_DIVIDE,  
    T_EQUALS,  
    T_STOP  
} Token;  
extern "C" Token read_token(char buf[]);
```

5. To save `calc.hpp`, select **File > Save file**.

---

## Marking and copying a block of code

In this section, you learn how to mark and copy a block of code. To do so, you will create another new file for the CALC program, `calc.cpp`. Instead of entering all the source for `calc.cpp`, you will avoid rekeying similar code by marking, copying and pasting multiple blocks of code from `calc.hpp` into new locations in the file `calc.cpp`.

1. Select **File > New file** from the Editor menu bar. An untitled document appears in the Editor Pane. You need to save this untitled document with the `.cpp` extension to indicate that the file has the type `.cpp`.
2. Select **File > Save file**. Type `calc.cpp` in the **File name** field and click **Save**. The File Selector refreshes to show the file `calc.cpp` on a local drive.

3. Select the file **calc.hpp** from the File Selector list to open this file. The Editor pane refreshes to show the contents of the file calc.hpp.
4. In calc.hpp, left-click at the start of the header comment block, hold the mouse button down and drag to the end of the header comment block. Release the mouse button and the header comment block is selected.

```
-----File calc.hpp-----*/
/*
 *
 * file for CALC.CPP PUSHPOP.CPP READTKN.CPP
 * calculator
 *
 */
/*
 *
 */
} toks {
/*
 *
 */
}

Token read_token(char buf[]);
tok {
```

5. Right-click and from the pop-up menu select **Copy**. Select **calc.cpp** in the list of open files.
6. Position the cursor to row 1 column 1 in this file. Right-click and select **Paste** from the pop-up menu. Change the comments to reflect the calc.cpp header comments.
7. Begin entering code for calc.cpp. Note that syntax errors are flagged as you enter the code.

```
/*-----File calc.cpp-----*/
/*
 *
 * A simple calculator that does operations on integers
 * that are pushed and popped on a stack
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "calc.cpp"
Syntax error.
```

After coding the second case statement, stop entering code.

8. Move the cursor to the top left corner of the block case T\_INTEGER, right-click and select **Selection > Select rectangle** from the pop-up menu.
9. Move the cursor to the lower right corner of this case statement. Position the cursor at the location that represents the longest statement length in the block of code. Right-click and select **Selection > Select rectangle** from the pop-up menu. The entire block between corner points is selected.

```

{
    Token tok;
    char word[100];
    char buf_out[100];
    int num;
    for(;;)
    {
        tok=read_token(word);
        switch(tok)
        {
            case T_STOP:
                break;
            case T_INTEGER:
                num = atoi(word);
                stack.push(num); /*CALC1 statement */
                break;

```

10. Right-click and select **Copy** from the pop-up menu. Position the cursor in the column directly below the case statement. Right-click and select **Paste** from the pop-up menu.
11. Repeat the previous step three more times. The result is five identical case statements.

```

        num = atoi(word);
        stack.push(num); /*CALC1 statement */
        break;
    case T_INTEGER:
        num = atoi(word);
        stack.push(num); /*CALC1 statement */
        break;
    case T_INTEGER:
        num = atoi(word);
        stack.push(num); /*CALC1 statement */
        break;
    case T_INTEGER:
        num = atoi(word);
        stack.push(num); /*CALC1 statement */
        break;
    case T_INTEGER:
        num = atoi(word);
        stack.push(num); /*CALC1 statement */
        break;

```

12. Edit the case statements to reflect the correct coding. Some changes require statements to be deleted as well as new statements to be added. To insert lines, press Enter. To delete lines, click at the start of the line and drag the mouse to the end of the line, right-click and from the pop-up menu select **Cut**. Complete the remaining code for calc.cpp.
13. To save calc.cpp, select **File > Save file**.



---

## Finding and replacing a class name

In this section, you learn how to find and replace a class name. To do so, you will create another new file for the CALC program, pushpop.cpp. As you enter the source for pushpop.cpp, you will intentionally enter IntStac as the class name for the class IntStack. This class is referenced in several files: calc.cpp, calc.hpp, and pushpop.cpp. You will use the Editor to find all occurrences of IntStac and replace them with the correct name, IntStack.

1. Select **File > New file** from the Editor menu bar. An untitled document appears in the Editor Pane. You need to save this untitled document with the .cpp extension to indicate that the file has the type .cpp.
2. Select **File > Save file**. Type pushpop.cpp in the **File name** field and click **Save**. The File Selector refreshes to show the file pushpop.cpp on a local drive.
3. Position the cursor in the Editor pane at row 1 column 1 and begin entering the code. Remember to enter Instac instead of Instack. You can copy blocks of text or code to avoid rekeying. See a previous section on marking and copying text.

```
/*-----  
/*  
/* input: num - value to push on the stack  
/* action: get a link to hold the pushed value, push link on  
/* stack  
/*-----  
void IntStac::push(int num) {  
    IntLink * ptr;  
    ptr = new IntLink;  
    ptr->set_i(num);  
    ptr->set_next(top);  
    top = ptr;  
}  
/*-----  
/*  
/* input: num - value to push on the stack  
/* action: get a link to hold the pushed value, push link on  
/* stack  
/*-----
```

4. Complete the code for pushpop.cpp.

```

while(top)
    pop();
}
IntLink::IntLink() { /* constructor leaves elements unassigned
}
IntLink::~IntLink() {
}
void IntLink::set_i(int j) {
    i = j;
}
int IntLink::get_i() {
    return i;
}
void IntLink::set_next(IntLink * p) {
    next = p;
}
IntLink * IntLink::get_next() {
    return next;
}

```

5. From the **Edit** menu, select **Find > Find/replace**.
6. Enter the name of the function to find, in this case `IntStac`, the name with which to replace it, in this case `IntStack` and click **All**. All occurrences of `IntStac` appear.

```

void IntStac::push(int num) {
int IntStac::pop() {
IntStac::Instac() {
IntStac::~IntStac() {

```

7. Click **Replace all**. All occurrences of `IntStac` are replaced with `IntStack`.
8. To check that `IntStack` is named correctly in other modules that make up the CALC program, select the other modules by holding the Shift key and left-clicking on each module name. Then select **All**.

```

class IntStack {

```

Notice that the first occurrence of `IntStack` is in `calc.hpp` and the name is correct.

9. Select **calc.cpp**. `IntStack` is present in this module.
10. Select **pushpop.cpp**. `IntStack` is present in this module as these are the occurrences that you have just changed.
11. To save `pushpop.cpp`, select **File > Save file**.

---

## Setting and finding quick marks

In this section, you learn how to set and find quick marks to quickly locate code within a file. To do so, you will create another new file for the CALC program, `readtokn.cpp`. As you enter the source for `readtokn.cpp`, you will set a quick mark to go to the top of the source file.

1. Select **File > New file** from the Editor menu bar. An untitled document appears in the Editor Pane. You need to save this untitled document with the `.cpp` extension to indicate that the file has the type `.cpp`.
2. Select **File > Save file**. Type `readtokn.cpp` in the **File name** field and click **Save**. The File Selector refreshes to show the file `readtokn.cpp` on a local drive.
3. Position the cursor in the Editor pane at row 1 column 1. Right-click and select **Edit > Set quick mark** from the pop-up menu. Setting this mark at the top of the file allows you to quickly move to this location. You can have only one quick mark in a file. Setting another quick mark replaces the previous quick mark, if one exists.
4. Begin entering code. You can copy blocks of text or code to avoid rekeying. See a previous section on marking and copying text.

```
case '+' : return T=PLUS;
case '-' : return T_MINUS;
case '*' : return T=TIMES;
case '/' : return T_DIVIDE;
case '=' : return T=EQUALS;
default:
    i=0;
    while (isdigit(c)) {
        buf[i++] = c;
        c = nextchar();
    }
    buf[i] = 0;
    if (i==0)
        return T_STOP;
    else
        return T_INTEGER;
}
```

5. Right-click on the Editor pane and select **Find > Find quick mark** from the pop-up menu. The cursor moves to the position marked by the quick mark located at the top of the file.

```

/*-----File readtokn.cpp-----
/*
/* A function to read input and tokenize it for a simple calcul
/*
/*-----
#include <ctype.h>
#include <stdio.h>
#include "calc.hpp"
/*-----File readtokn.cpp-----
/*
/* A function to read input and tokenize it for a simple calcul
/*
/*-----
static char nextchar(void)
{
/*   input   action
/*   -----
/*    2      push 2 on stack
/*   18      push 18

```

6. To save readtokn.cpp, select **File > Save file**.

---

## Fixing compile-time errors

In this section, you learn how to fix compile-time errors using the Editor on your workstation.

You send the completed source files calc.cpp, readtokn.cpp, calc.hpp and pushpop.cpp to the OS/390 system. There are many ways to transfer files from the workstation local file system to the host file system. For example, you can use FTP to send the files to the host system or you can mount your host file system as a Windows NT drive using a remote file system such as NFS. Once the host file system is mounted as a Windows NT drive, you can then copy the workstation files to this Windows NT drive.

When the files are on OS/390, you then compile these files. The compile process results in several compile-time errors for the file pushpop.cpp. You fix these errors using the workstation Editor. After you fix pushpop.cpp, you resend the file to the OS/390 system where you recompile the updated source.

1. Select **File > Open file** from the Editor menu bar.
2. Type pushpop.cpp in the **File name** field and click **Open**. The File Selector refreshes to show the file pushpop.cpp on a local drive.
3. Right-click on the Editor pane and select **Preferences** from the pop-up menu. The Editor Preferences dialog opens. Select **Controls** from the list and then check the **Line numbers** check box. Click **Apply**. Click **OK**. Line numbers appear on the Editor pane.

```

000001/*-----File pushpop.cpp-----
000002/*
000003/* Push and pop functions for a stack of integers
000004/*
000005/*-----
000006#include <stdio.h>
000007#include <stdlib.h>
000008#include "calc.hpp"
000009/*-----
000010/*
000011/* input: num - value to push on the stack
000012/* action: get a link to hold the pushed value, push
000013/* stack
000014/*-----
000015void IntStack::push(int num) {

```

4. Use the line numbers in the compiler error listing to locate each error. For example, several errors are caused by an incorrect function name for IntStack on line 37.

```

000027/*-----
000028int IntStack::pop() {
000029    IntLink * ptr;
000030    int num;
000031    ptr = top;
000032    num = ptr->get_i();
000033    top = ptr->get_next();
000034    delete ptr;
000035    return num;
000036}
000037IntStack::Instac() {
000038    top = 0;
000039}
000040IntStack::~IntStack() {
000041    while(top)

```

5. Other errors are the result of incorrect names in calc.hpp. For example, T-EQUALS and T-STOP should be T\_EQUALS and T\_STOP.

```

000001/*-----File calc.hpp-----
000002/*
000003/* Header file for CALC.CPP PUSHPOP.CPP READTKN.CPP
000004/* a simple calculator
000005/*-----
000006typedef enum toks {
000007    T_INTEGER,
000008    T_PLUS,
000009    T_TIMES,
000010    T_MINUS,
000011    T_DIVIDE,
000012    T-EQUALS,
000013    T-STOP
000014} Token;
000015extern "C" Token read_token(char buf[]);

```

6. Fix the errors. Then, send the files to the host and recompile. Repeat the above steps until all errors are fixed.



---

## Chapter 3. Debugging programs

---

### Debugging a program

The following sections identify typical tasks you might want to perform while using the debugger with a C and C++ program and explains how to accomplish these tasks. The “Sample CALC Program” on page 33 is used to demonstrate some of these actions. Follow the tasks in sequence as the tasks are often related to each other.

For explanation of a pane or dialog box, click the pane or dialog box and press F1.

1. “Starting the Distributed Debugger user interface daemon”
2. “Starting a program with Debug Tool in OS/390 UNIX” on page 16
3. “Setting a breakpoint to halt before a certain function is entered” on page 17
4. “Displaying and modifying the value of a variable” on page 18
5. “Stopping on a line only if a condition is true” on page 19
6. “Viewing and modifying data members of the this pointer” on page 20
7. “Debugging when only a few parts are compiled with TEST” on page 21
8. “Displaying storage” on page 21
9. “Finding unexpected storage overwrite errors” on page 22
10. “Finding uninitialized storage errors” on page 23
11. “Getting a function traceback” on page 24

---

### Starting the Distributed Debugger user interface daemon

In this section, you learn how to start the Distributed Debugger user interface daemon. To debug the CALC program, you need to first start the Distributed Debugger user interface in daemon mode. After you start the daemon, you then go to an OS/390 session and run your program with the appropriate runtime option to cause the Language Environment runtime library to load Debug Tool. The next section describes how to define the script to start a debug session.

The Distributed Debugger client is invoked on the workstation in a mode which causes it to wait for a TCP/IP connection from Debug Tool running on OS/390. The Distributed Debugger provides the client graphical user interface

to the debug information provided by Debug Tool. For example, to monitor a variable, the Distributed Debugger client asks Debug Tool for the value; Debug Tool responds with a value and the Distributed Debugger client displays the value in the Monitors pane.

You can start the Distributed Debugger daemon in any of the following ways:

- By double-clicking on its icon from a program folder. If you start the Distributed Debugger from an icon you must make sure the properties for the icon include the `-qdaemon` and `-quiport` options.
- By selecting **IBM C and C++ Productivity Tools for OS 390 > IBM Distributed Debugger** from the Windows **Start** menu.
- By entering the Distributed Debugger command `idebug -qdaemon -quiport=<port>` on a Productivity Tools command line.

`<port>` The port number where you want the Distributed Debugger user interface daemon to listen for Debug Tool. For the port number in this example, use 8000. This is the port number used by default in the port parameter of the TEST runtime option which causes the Language Environment to load Debug Tool. The same port number must be used by the Distributed Debugger user interface daemon and Debug Tool.

---

## Starting a program with Debug Tool in OS/390 UNIX

In this section, you learn how to start the CALC program with Debug Tool.

Complete the following steps to compile the CALC program, set up the environment for debugging with Debug Tool, and debug CALC:

1. Compile and bind your program with the `-g` debug option, for example,  
`c++ -+ -g calccpp calc.cpp readtokn.cpp pushpop.cpp`
2. Create a script file that will set up the environment for debugging and run your program. Create a file called `dbg` and do the following in the file:
3.
  - Set your STEPLIB to point to the Debug Tool SEQAMOD data set, the Language Environment runtime library, and the library which contains the application you intend to debug, if you intend to run the application from a PDS.
  - Specify the runtime TEST option. If you did not code `#pragma runopts` in your program, you need to specify the runtime TEST option through the system variable `_CEE_RUNOPTS`.

Here is a sample script you can use:



```
#DBG - SHELL SCRIPT TO DEBUG PROGRAM
export STEPLIB=EQAW.V1R2M0.SEQAMOD:\
SYSID.CEE.SCEERUN:\
SYSID.CBC.SCLBDLL:\USERID.PROJ1.LOAD:$STEPLIB
export _CEE_RUNOPTS="TEST(,,,VADTCPIP&wkst_id%portid:*)"
$*&
```

*wkst\_id*

The numeric IP address of your workstation or the TCP/IP name of your workstation.

*%portid*

The TCP/IP port number (this is optional and defaults to 8000); if specified, the value must match the port number value that was entered when the Distributed Debugger user interface daemon was started.

4. To set up your environment and debug calccpp, use the shell script dbg and execute as follows: >dbg calccpp

The application being debugged, calccpp, runs on the host and causes the Language Environment to load Debug Tool. Debug Tool will connect to the Distributed Debugger daemon running on the workstation and a Distributed Debugger program pane will display.

---

## Setting a breakpoint to halt before a certain function is entered

In this section, you learn how to set a function breakpoint. When the CALC program is run, the program stops at that function before it is entered. To set the breakpoint you need to include a C++ signature along with the function name.

**Note:** For the Distributed Debugger to be able to halt the program, the file with the called code must be compiled with the compile-time TEST option.

To set a function breakpoint from the Modules pane:

1. Click the Monitors tab. Expand the list under pushpop.cpp in the Modules pane. You see all the names of all the functions and methods defined in the compile unit. Look for the function IntStack::push(int).
2. To halt just after the function IntStack::push(int) is called, right-click on that function to open a pop-up menu.
3. Choose **Set Function Breakpoint** from the pop-up menu. The breakpoint is set.

```

Thread 1:./pushpop.cpp
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "calc.hpp"
4  ➡ void IntStack::push(int num)
5      IntLink * ptr;
6      ptr = new IntLink;
7      ptr->set_i(num);
8      ptr->set_next(top);
9      top = ptr;
10 }

```

4. Select **Debug > Run** to run the CALC program until the breakpoint is hit.

```

Thread 1:./pushpop.cpp
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "calc.hpp"
4  ➡ void IntStack::push(int num)
5      IntLink * ptr;
6      ptr = new IntLink;
7      ptr->set_i(num);
8      ptr->set_next(top);
9      top = ptr;
10 }

```

---

## Displaying and modifying the value of a variable

In this section, you learn how to display and change the value of the variable *num* in the pushpop.cpp module. To do this you use the Monitors pane.

1. Select **Debug > Step Into** from the **Debug** menu. The CALC program runs and steps into the first call of function `IntStack::push(int)`.
2. Select **Debug > Step Over** until just after `IntLink` has been allocated, which will be the statement `ptr->set_i(num);`.

```

Thread 1:./pushpop.cpp
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "calc.hpp"
4  ● void IntStack::push(int num)
5      IntLink * ptr;
6      ptr = new IntLink;
7  ➡ ptr->set_i(num);
8      ptr->set_next(top);
9      top = ptr;
10 }

```

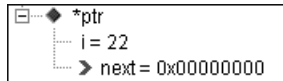
3. Highlight the variable *num* and right-click. Select **Add to Program Monitor** from the pop-up menu.
4. Click the Monitors tab to display the Monitors pane. The value for variable *num* is shown.



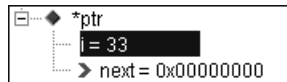
5. To modify the value of *num* to 22, double-click on the variable *num*.
  - a. Enter the value 22 for the *num* variable.



- b. Press Enter to submit the change.
6. Select **Debug > Step Over** to execute the current line.
7. Click the Monitors tab then select **Monitor Expression** from the **Monitors** menu.
8. Enter the expression *\*ptr*, click on the **Program monitor** radio button and then click **OK**.
9. Expand the *\*ptr* expression in the Monitors pane. A list of the data members of the object pointed to displays in the Monitors pane.



10. To modify the value of *i* to 33, double-click *i*.
  - a. Enter the value 33 for *i*.
  - b. Press Enter to submit the change.



Alternatively, instead of monitoring all values of *\*ptr*, you can monitor the expression *(\*ptr).i* directly. You can then modify the value for this expression by adding this expression to the **Monitor Expressions** dialog and from the Monitors pane, edit the value of *i* such that the expression on the Monitors pane reads *(\*ptr).i=33*.

---

## Stopping on a line only if a condition is true

In this section, you learn how to set a conditional breakpoint in `main()` of `calc.cpp`. Often a particular part of your program works fine for the first few thousand times, but fails under certain conditions. You do not want to set a simple line breakpoint because you do not want to stop on the line every time. Instead, you want to stop in `T_DIVIDE` only if the divisor is 0 (before the exception occurs).

1. Move the cursor to the location to set the line breakpoint, in this case the statement `stack.push(num/num2)`.
2. Select **Source > Set Line Breakpoint** from the menu bar.

3. Enter the expression `num==2` in the **Expression** field.
4. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog.
5. Select **Debug > Run** from the **Debug** menu. The CALC program runs and then stops at the line breakpoint only if `num2=0`. If the value of `num2` is not 0, the program will continue.

---

## Viewing and modifying data members of the *this* pointer

In this section, you learn how to modify the data members of the *this* pointer, and step into a class method, for example, one from class `IntLink`. To do this you set another line breakpoint in the module `pushpop.cpp` and monitor the values of the *this* pointer.

1. Set a line breakpoint in the method `IntLink::set_i` in `pushpop.cpp`. The breakpoint should be set at line 32 where `i = j`;

```
31 void IntLink::set_i(int j) {
32     i = j;
```

2. Select **Debug > Run** from the **Debug** menu. The CALC program runs and then stops at the breakpoints set in earlier sections. Continue to select **Debug > Run** until you reach the line breakpoint that you just set.

```
31 void IntLink::set_i(int j) {
32     i = j;
33 }
34 int IntLink::get_i() {
```

3. Click the Monitors tab. From the **Monitors** menu, select **Monitor Expression**.
4. Enter `*this` as the expression to be evaluated. Select the **Program monitor** radio button. Click **OK**.
5. Expand the *\*this* expression in the Monitors pane. A list of the data members of the object pointed to displays in the Monitors pane.

```
*this
├── i = 0
└── next = 0x00000000
```

6. To modify element *i*, double-click on *i*.
7. Enter the value 2001 for *i*.
8. Press Enter to submit the change.

```
*this
├── i = 2001
└── next = 0x00000000
```

Instead of monitoring all values of *\*this*, you can monitor the element *i* directly. You can then modify the value for *i* by adding *i* to the Monitor Expressions dialog and changing the value of *i* to 2001 from the Monitors pane. Similarly, if there were ambiguity (for example, if you also had an auto variable named *i*), you could monitor the expression *(\* this).i* through the Monitor Expressions dialog and, from the Monitors pane edit the value of *i* so that the expression in the Monitors pane would read *(\*this).i=2001*.

---

## Debugging when only a few parts are compiled with TEST

In this section, you learn how to debug only one part in a multiple-part program. To do this you compile that one part with debug information, and then set breakpoints in this one part to debug. For example, you only compile `pushpop.cpp` with the TEST option. All other files are compiled without this option. Then you set a breakpoint after the call to the function `IntStack::push(int)` in the file `pushpop.cpp`. The Distributed Debugger will show module display trees for only those components containing debug information.

Since the `pushpop.cpp` part displays, you can set a breakpoint after the call to the function `IntStack::push(int)` as follows:

1. In the Modules pane, expand the list under `pushpop.cpp`. You see all the names of all the functions and methods defined in the compile unit. Look for the function `IntStack::push(int)`.
2. To halt just after the function `IntStack::push(int)` is called, right-click on that function.
3. Choose **Set Function Breakpoint** from the pop-up menu. The breakpoint is set.
4. Select **Debug > Run** from the **Debug** menu. The CALC program runs and execution stops after the function call.

---

## Displaying storage

In this section, you learn how to display the content of a variable in storage. To do this you monitor the `IntLink *` variable *ptr* which can point to a location in storage. To display the storage:

1. Set a line breakpoint at line 7 where the statement reads `ptr->set_num()`; in `pushpop.cpp`.

6	<code>ptr = new IntLink;</code>
7 ●	<code>ptr-&gt;set_i(num);</code>
8	<code>ptr-&gt;set_next(top);</code>

2. Select **Debug > Run** from the **Debug** menu. The CALC program runs and then stops at the breakpoints set in earlier sections. Continue to select **Debug > Run** until you reach the line breakpoint that you just set.

7	ptr->set_i(num);
8	ptr->set_next(top);
9	top = ptr;

3. Click the Monitors tab. From the **Monitors** menu, select **Monitor Expression**.
4. Enter ptr as the expression to be evaluated. Select the **Storage monitor** radio button. Click **OK**. A new Storage Monitor pane appears as a tab.
5. Select **Debug > Step Over** to step over line 7.
6. Click the ptr monitor tab to return to the ptr Storage Monitor pane. The memory value of *num* displays in the ptr Storage Monitor pane.

---

## Finding unexpected storage overwrite errors

In this section, you learn how to set a storage change breakpoint to cause the program execution to halt when the 4 bytes of storage pointed to by the variable *abcd* is changed. During program execution, some storage might unexpectedly change its value. The Distributed Debugger allows you to find out when and where this happened. Consider this simple example where function *set\_i* changes more than the caller expects it to change.

```
struct s { int i; int j;};
struct s abcd = { 0, 0 };
/* function sets only field i */
void set_i(struct s * p, int k)
{
    p->i = k;
    p->j = k;    /* error, it unexpectedly sets field j also */
}
main() {
    set_i(&abcd,123);
}
```

1. Select the Breakpoints pane then select **Breakpoints > Set Storage Change** from the menu bar.
2. Enter the expression *&(abcd.j)* in the **Address or Expression** field. This expression evaluates to an address.
3. Enter 4 as the number of bytes to monitor in the **Bytes to Monitor** field.
4. Click **OK** to set the breakpoint and dismiss the Storage Change Breakpoint dialog.
5. Select **Debug > Run** from the **Debug** menu. The program runs until the contents of any of these 4 bytes of storage change value.

---

## Finding uninitialized storage errors

In this section, you learn how to find uninitialized heap storage. To do this you debug the program CALC, setting the run-time STORAGE option as STORAGE(FD,FB,F9). You then set a function breakpoint for the IntStack::push(int) function which is before the *ptr* variable declaration in pushpop.cpp. You then run the program to the breakpoint and then monitor the *\*ptr* expression.

The Language Environment run-time TEST and STORAGE options are coded as follows:

```
'TEST (,,,VADTCPIP&wkst_id %portid:*) STORAGE(FD,FB,F9)'
```

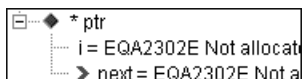
The first subparameter of STORAGE is the fill byte for storage allocated from the heap. For example, storage allocated through operator new is filled with the byte 0xFD. If you see this byte repeated throughout storage, it is likely uninitialized heap storage.

The second subparameter of STORAGE is the fill byte for storage allocated from the heap but then freed. For example, storage freed by the operator delete will be filled with the byte 0xFB. If you see this byte repeated throughout storage, it is likely storage that was allocated on the heap but has been freed.

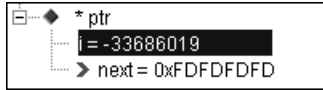
The third subparameter of STORAGE is the fill byte for auto storage variables in a new stack frame. If you see this byte repeated throughout storage, it is likely that it is uninitialized auto storage.

The values chosen here are odd and large, to maximize early problem detection. For example, if you attempt to branch to an odd address, you will get an exception immediately.

1. Set a function breakpoint for the IntStack::push(int) function in pushpop.cpp. If you have set this breakpoint from a previous section, ignore this step.
2. Select **Debug > Run** from the **Debug** menu. The CALC program runs and then stops at the breakpoints set in earlier sections. Continue to select **Debug > Run** until you reach the function breakpoint that you just set.
3. Highlight the variable *\*ptr* in pushpop.cpp you want to monitor.
4. Right-click on the highlighted variable, and select **Add to Program Monitor** from the pop-up menu. Click the Monitors tab. Expand the variable *\*ptr*.



5. Select **Debug > Step Over** to step over the current line. Continue to step over until you step to the statement `ptr-> set_i (num);`. You will see the byte fill for uninitialized heap storage.



6. To change the representation of the *i* value to hexadecimal, right click on *i* and from the pop-up menu select **Representation**.
7. On the Monitor Representation dialog, click on the **Hexadecimal** radio button. The value of *i* changes from a decimal value to a hexadecimal value.

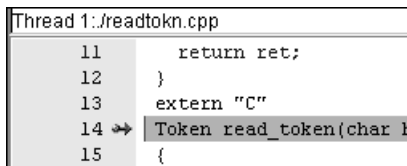


---

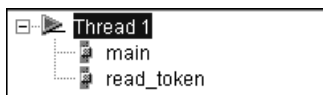
## Getting a function traceback

In this section, you learn how to use the Stacks pane to view the traceback of calling functions. Often when you get close to a programming error, you want to know how you got into that situation, especially what the traceback of calling functions is. To get this information, you use the Stacks pane to view program call stack information. Initially, the stack display trees are collapsed so only thread names and thread numbers are shown. Each stack display tree can be expanded so the names of all functions in the stack are displayed.

1. Set a line breakpoint at line 14 where the statement reads `read_token()` in `readtokn.cpp`.
2. Select **Debug > Run** from the **Debug** menu. The CALC program runs and then stops at the breakpoints set in earlier sections. Continue to select **Debug > Run** until you reach the function breakpoint that you just set.



3. Click the Stacks tab.
4. Expand the thread to view the program call stack information. The Stacks pane shows the traceback of callers.





---

## Chapter 4. Analyzing program performance

---

### Analyzing a program's performance

The following sections identify typical tasks you might want to perform while using the Performance Analyzer with a C and C++ program and explains how to accomplish these tasks. The "Sample CALC Program" on page 33 is used to demonstrate these actions. Follow the tasks in sequence as the tasks are often related to each other.

For explanation of a pane or dialog box, click the pane or dialog box and press F1.

1. "Creating a trace file"
2. "Starting the Performance Analyzer" on page 26
3. "Analyzing a trace file" on page 26
4. "Searching for trace data in a diagram" on page 27
5. "Controlling what data is shown in a diagram" on page 29
6. "Navigating the trace file views" on page 30

---

### Creating a trace file

In this section, you use the CALC program to learn how to build and trace a program in OS/390 UNIX.

To trace a program with the Performance Analyzer, it must be compiled with the correct options and then executed with the run-time option `PROFILE(ON,'string')` set in order to start the Performance Analyzer.

Complete the following steps to build your program, set up the environment for tracing with the Performance Analyzer, and perform the trace:

1. Compile and bind your program with `TEST(HOOK)` and `NOGONUMBER`, for example, `c++ -+ -o calccpp -O -Wc,"TEST(HOOK),NOGONUMBER" calc.cpp readtokn.cpp pushpop.cpp`
2. Create a script file that will set up the environment for tracing and run your program. Create a file called `trc` and do the following in the file:
  - Set the `__PROF_FILE_NAME=filename` environment variable to specify the name of the trace file to be generated.
  - Add `SCTVMOD` load module data set to the `STEPLIB` of the program if it has not been installed in the link pack area (LPA).

- Set the run-time option, PROFILE(ON,'string')
- Enter the name of the program.

Here is a sample script you can use:

```
#PA - SHELL SCRIPT TO ANALYZE PROGRAM PERFORMANCE
#CBC is determined by the location of the C++ compiler.
export STEPLIB=CBC.SCTVMOD:$STEPLIB
export _CEE_RUNOPTS="PROFILE(ON,'FUNCTION=ALL,REAL')"
export __PROF_FILE_NAME=calccpp.trc
calccpp
```

3. Run the script file to trace the program and create a trace file.

---

## Starting the Performance Analyzer

In this section, you learn how to start the Performance Analyzer.

You can start the Performance Analyzer in any of the following ways:

- By double-clicking on its icon from a program folder
- By selecting **IBM C and C++ Productivity Tools for OS 390 > Performance Analyzer** from the Windows **Start** menu
- By entering the following Performance Analyzer command on a Productivity Tools command line:

```
ianalyze [/x]
```

Where /x represents any number of Performance Analyzer invocation parameters.

---

## Analyzing a trace file

In this section, you learn how to use the Performance Analyzer on the workstation to analyze the trace file that you have just created on the host. After you create a trace file on the host, you download it as a binary file to the workstation so you can use the Performance Analyzer. For example, you can use FTP to send the files to the workstation or you can mount your host file system as a Windows NT drive using a remote file system such as NFS. Once you have downloaded the trace file, you can use the Performance Analyzer on your workstation to create diagrams to analyze your trace file.

### Analyzing time events in the Time Line diagram

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name calccpp.trc in the dialog box; or search for the trace file by clicking the **Find** button.

3. Select the **Time Line** diagram to view the data.
4. Click the **OK** button.
5. Highlight the entire time range, in this example, 0 to 307.
6. Check the status area of the Time Line diagram for the elapsed time between all events.



### Finding the functions that take the most time to execute

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the **Statistics** diagram to view the data.
4. Click the **OK** button.
5. Select **View > Details on > Functions** to see a list of functions in the Statistics diagram.
6. Locate the functions that take the most time to execute. The functions that take the most time to execute will be at the top of the list by default because the list is sorted by execution time. In this example, the `IntStack::push(int)` function takes the most time to execute (22%).

---

## Searching for trace data in a diagram

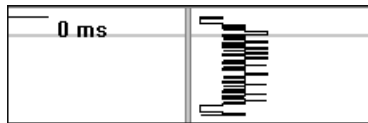
In this section, you learn how to find the call to function `IntStack::push(int)` in several Performance Analyzer diagrams.

### Finding a specific function call or return

To search for the function call `IntStack::push(int)` using the Time Line diagram:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the Time Line diagram to view the data.
4. Click the **OK** button.
5. Open the Find Function window in the Time Line diagram, select **Edit > Find > Function**.
6. Enter the function name `IntStack::push(int)` in the **Find** entry field.
7. Check the **Case sensitive** check box to enable case-sensitive searching.

8. Select **1** from the list in the **Thread** field.
9. Click the **Call or Return** radio button to search for occurrences of when the function was either called or returned.
10. Click **OK** to continue. The first occurrence of the call to the function `IntStack::push(int)` is located.
11. Select **Edit > Find next** to find the next occurrence of the function call or return.



The search for the function begins at the currently selected function. The search continues until the function is found or the end of the diagram is reached. If the function is found, it is highlighted; if it is not found, a message box to that effect appears.

### Finding trace data for the `IntStack::push(int)` function

To search for the function `IntStack::push(int)` in the Statistics diagram, complete the following steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the Statistics diagram to view the data. Click **OK**.
4. Make sure trace data for functions is shown in the diagram. Select **View > Details on > Functions** in the Statistics diagram.
5. Open the Find Function window in the Statistics diagram, by selecting **Options > Find**.
6. Enter the function name `IntStack::push(int)` in the **Find** entry field.
7. Select the **Case sensitive** check box if you want to enable case-sensitive searching.
8. Click **OK** to continue.
9. If more than one function matches your search criteria, select the desired function in the list box and click **OK**. The function is highlighted when found.

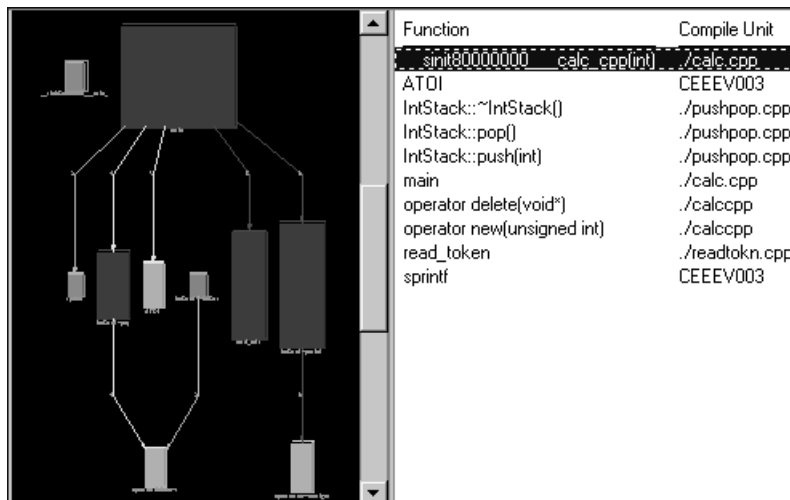
## Controlling what data is shown in a diagram

In this section, you learn how to filter the trace data. Filters allow you to temporarily reduce the amount of trace data shown in a diagram or to isolate interesting or problematic areas. There are several techniques for filtering the trace data.

### Showing trace data for a specific component type

To show trace data for a specific component type in the Statistics diagram and the Dynamic Call Graph diagram, complete the following steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the **Dynamic Call Graph** diagram and the **Statistics** diagram to view the data. Click **OK**.
4. Select **View > Details on > Functions** in the Statistics diagram or **View > Nodes of > Functions** in the Dynamic Call Graph diagram.



To filter specific functions from the Call Nesting diagram, complete these steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the **Call Nesting** diagram to view the data.
4. Select **View > Include functions**. The Include Functions window appears.

5. Click **Deselect all**.
6. Scroll the list to find the function `IntStack::~IntStack()` to include in the diagram's display.
7. Select this function.
8. Click **OK**. The selected function displays.



### Filtering nodes in the Dynamic Call Graph

To define a specific cross section of nodes that you want shown in the Dynamic Call Graph diagram, complete these steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the **Dynamic Call Graph** diagram to view the data.
4. Select **View > Filters > Nodes**. The Nodes Filter window appears.
5. Select the check boxes for the desired filter criteria such as **Execution Time** and **Number of Calls**, and fill in the corresponding values by which you want to filter the nodes.
6. Click the **And** radio button to show the nodes that meet the values for all the selected criteria. Alternatively, click the **Or** radio button to show the nodes that meet the values of at least one of the selected criteria.
7. Select one or more compile units in which you want to search for nodes that meet the filter criteria.
8. Click **OK** to apply the filters and close the Nodes Filter window.

---

## Navigating the trace file views

In this section, you learn how to correlate events between diagrams and enlarge or reduce diagram details.

### Correlating events between Call Nesting, Execution Density and Time Line diagrams

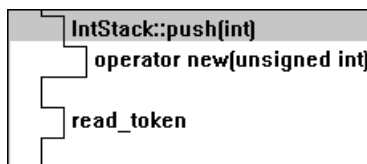
To correlate events between the Call Nesting, Execution Density, and Time Line diagrams, complete these steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the Call Nesting, Execution Density and Time Line diagrams to correlate events.
4. Highlight the event range of interest in any of the three diagrams by taking these steps:
  - a. Left-click on the first event.
  - b. Drag the pointer to the last event.
  - c. Release the mouse button.
5. Select **Options > Correlation** in the diagram where you highlighted the event range. The Correlation window appears.
6. Click **Select all** to correlate to all of the diagrams listed.
7. Click **OK**. All three diagrams correlate to show the same event range in each diagram.

### Correlating the `IntStack::push(int)` function

To correlate from the `IntStack::push(int)` function in the Statistics diagram to the instance of the call to that same function in the Call Nesting diagram that used the most time of all calls to that function, complete these steps:

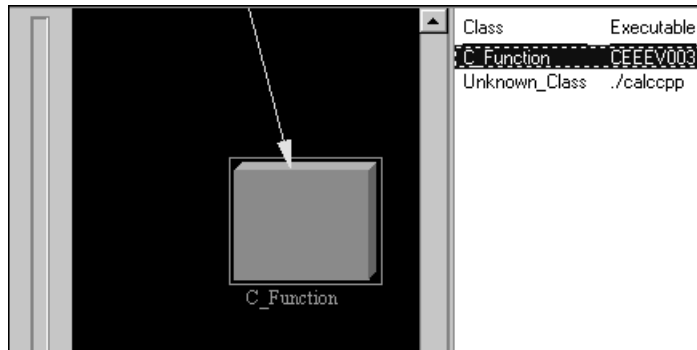
1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name `calccpp.trc` in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the Statistics and Call Nesting diagrams to view the data.
4. Open the trace file in the Statistics and Call Nesting Diagrams.
5. Select **View > Details on > Functions** in the Statistics diagram to show functions in the diagram.
6. Highlight a single function called `IntStack::push(int)` in the Statistics diagram.
7. Select **Options > Correlation** in the Statistics diagram. The Correlation window appears.
8. Click the Call Nesting diagram.
9. Click **OK**. The Call Nesting diagram shows the corresponding function.



### Enlarging and reducing diagram details

To enlarge a region of a the Time Line diagram, Execution Density diagram or Dynamic Call Graph diagram that is of most interest, follow these steps:

1. In the Performance Analyzer - Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name calccpp.trc in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select any one of the above three diagrams to view the data. For example, select the Dynamic Call Graph diagram.
4. Select **View > Zoom in**.
5. Scroll until you see the area you selected.
6. Continue alternately selecting **Zoom in** and scrolling to the selected area until the diagram is enlarged to the degree you want.
7. If you zoom in too far, select **View > Zoom out** to quickly back out one step.





---

## Appendix. Sample Program

---

### Sample CALC Program

The CALC program is a simple calculator that reads its input from a character buffer. If integers are read, they are pushed on a stack. If one of the operators (+ - \* /) is read, the top two elements are popped off the stack, the operation is performed on them, and the result is pushed on the stack. The = operator writes out the value of the top element of the stack to a buffer.

```
/* FILE CALC.HPP                                     */
/*                                                     */
/* Header file for CALC.CPP PUSHPOP.CPP READTKN.CPP    */
/* a simple calculator                                  */
/*                                                     */
typedef enum toks {
    T_INTEGER,
    T_PLUS,
    T_TIMES,
    T_MINUS,
    T_DIVIDE,
    T_EQUALS,
    T_STOP
} Token;
extern "C" Token read_token(char buf[]);
class IntLink {
private:
    int i;
    IntLink * next;
public:
    IntLink();
    ~IntLink();
    int get_i();
    void set_i(int j);
    IntLink * get_next();
    void set_next(IntLink * d);
};
class IntStack {
private:
    IntLink * top;
public:
    IntStack();
    ~IntStack();
    void push(int);
    int pop();
};
/* FILE CALC.CPP                                     */
/*                                                     */
/* A simple calculator that does operations on integers that */
/* are pushed and popped on a stack                       */
```

```

/*                                                                    */
#include <stdio.h>
#include <stdlib.h>
#include "calc.hpp"
IntStack stack;
int main()
{
    Token tok;
    char word[100];
    char buf_out[100];
    int num,num2;
    for(;;)
    {
        tok=read_token(word);
        switch(tok)
        {
            case T_STOP:
                break;
            case T_INTEGER:
                num = atoi(word);
                stack.push(num);      /* CALC1 statement */
                break;
            case T_PLUS:
                stack.push(stack.pop()+stack.pop());
                break;
            case T_MINUS:
                num = stack.pop();
                stack.push(num-stack.pop());
                break;
            case T_TIMES:
                stack.push(stack.pop()*stack.pop() );
                break;
            case T_DIVIDE:
                num = stack.pop();
                num2 = stack.pop();
                stack.push(num/num2); /* CALC2 statement */
                break;
            case T_EQUALS:
                num = stack.pop();
                sprintf(buf_out,"= %d ",num);
                stack.push(num);
                break;
        }
        if (tok==T_STOP)
            break;
    }
    return 0;
}
/* FILE: PUSHPOP.CPP                                                */
/*                                                                    */
/* Push and pop functions for a stack of integers                    */
/*                                                                    */
#include <stdio.h>
#include <stdlib.h>
#include "calc.hpp"

```

```

/*                                                                    */
/* input:  num - value to push on the stack                          */
/* action: get a link to hold the pushed value, push link on stack  */
/*                                                                    */
void IntStack::push(int num) {
    IntLink * ptr;
    ptr = new IntLink;
    ptr->set_i(num);
    ptr->set_next(top);
    top = ptr;
}
/*                                                                    */
/* return: int value popped from stack (0 if stack is empty)        */
/* action: pops top element from stack and get return value from it */
/*                                                                    */
int IntStack::pop() {
    IntLink * ptr;
    int num;
    ptr = top;
    num = ptr->get_i();
    top = ptr->get_next();
    delete ptr;
    return num;
}
IntStack::IntStack() {
    top = 0;
}
IntStack::~IntStack() {
    while(top)
        pop();
}
IntLink::IntLink() { /* constructor leaves elements unassigned */
}
IntLink::~IntLink() {
}
void IntLink::set_i(int j) {
    i = j;
}
int IntLink::get_i() {
    return i;
}
void IntLink::set_next(IntLink * p) {
    next = p;
}
IntLink * IntLink::get_next() {
    return next;
}
/*      FILE READTOKEN.CPP                                          */
/*                                                                    */
/* A function to read input and tokenize it for a simple calculator */
/*                                                                    */
#include <ctype.h>
#include <stdio.h>
#include "calc.hpp"
/*                                                                    */

```

```

/* action: get next input char, update index for next call      */
/* return: next input char                                     */
/*                                                             */
static char nextchar(void)
{
    /*    input    action
    *    ---    ---
    *    2        push 2 on stack
    *    18       push 18
    *    +        pop 2, pop 18, add, push result (20)
    *    =        output value on the top of the stack (20)
    *    5        push 5
    *    /        pop 5, pop 20, divide, push result (4)
    *    =        output value on the top of the stack (4)
    */
    char * buf_in = "2 18 + = 5 / = ";
    static int index; /* starts at 0 */
    char ret;
    ret = buf_in[index];
    ++index;
    return ret;
}
/*                                                             */
/* output: buf - null terminated token                         */
/* return: token type                                         */
/* action: reads chars through nextchar() and tokenizes them */
/*                                                             */
extern "C"
Token read_token(char buf[])
{
    int i;
    char c;
    /* skip leading white space */
    for( c=nextchar();
        isspace(c);
        c=nextchar())
        ;
    buf[0] = c;
    /* get ready to return single char e.g. "+" */
    buf[1] = 0;
    switch(c)
    {
        case '+' : return T_PLUS;
        case '-' : return T_MINUS;
        case '*' : return T_TIMES;
        case '/' : return T_DIVIDE;
        case '=' : return T_EQUALS;
        default:
            i = 0;
            while (isdigit(c)) {
                buf[i++] = c;
                c = nextchar();
            }
            buf[i] = 0;
            if (i==0)

```

```
        return T_STOP;
    else
        return T_INTEGER;
    }
}
```



---

## Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
Intellectual Property and Licensing  
IBM Corporation  
North Castle Drive, MD-NC 119  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director  
IBM Canada Ltd.  
1150 Eglinton Ave E  
Toronto, Ontario, M3C 1H7  
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.



All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

CICS	IMS
CICS/ESA	Language Environment
CICS/MVS	MVS/ESA
CICS/VSE	OS/390
DB2	S/390
IBM	

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Sun, SunLink, Solaris, SunOS, Java, all Java-based trademarks and logos, NFS, and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Other company, product, and service names may be trademarks or service marks of others.







Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

GC09-2918-00



Spine information:



Getting Started

C/C++ Productivity Tools

Release 1.0