Running and Debugging

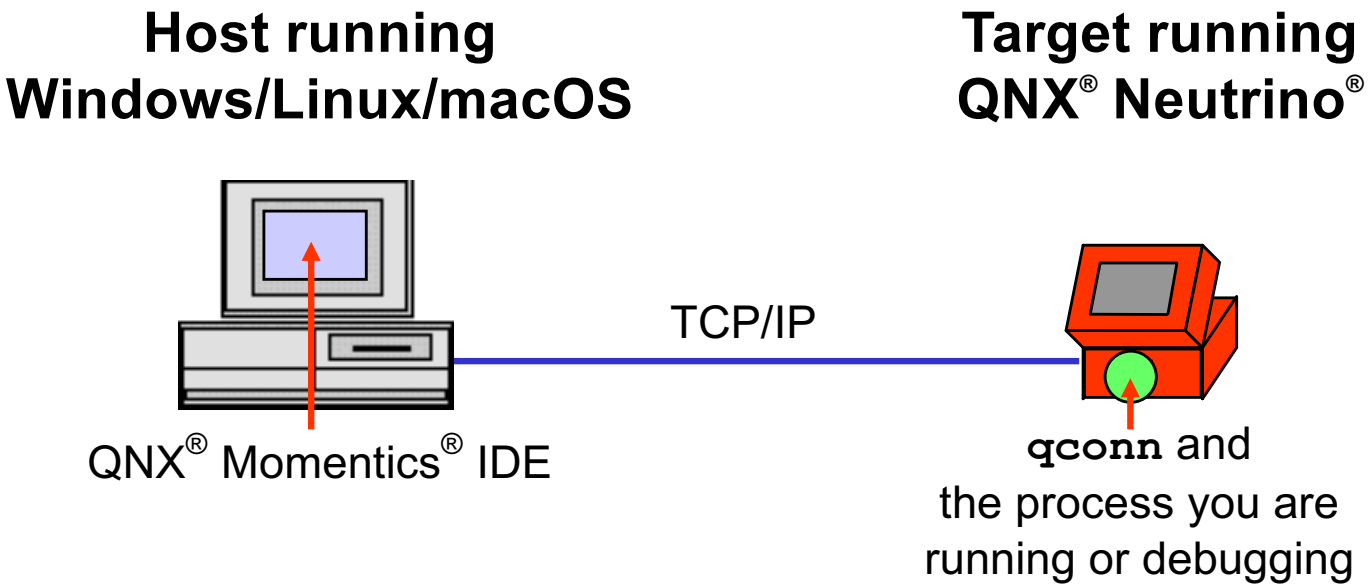
You will learn:

- some ways of running your executable
- how to start up a debugging session
  - from the start, postmortem, and attaching to an already running program
- various debugging techniques such as:
  - stepping through code as it executes
  - stopping your code at various points and when specific conditions are met
  - looking at and changing your data
  - debugging library code

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

Remote development:



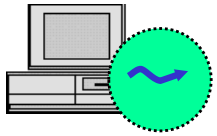
- **`qconn`** is a program on the target that must be running for the IDE to deal with the target

What is a Launch Configuration?

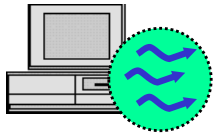
- the IDE needs to know:
  - where to run or debug your program
  - how to get your program there
  - what program to run
  - command line options
  - environment variables
  - special tools or settings
  - etc…
- a Launch Configuration is where the IDE stores this information
  - seems a lot of work up front, but
  - you only need to enter it once
  - the same one can be used for both running and debugging

Types of debugging supported:

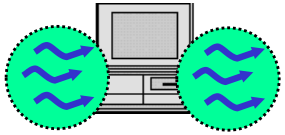
- single threaded process



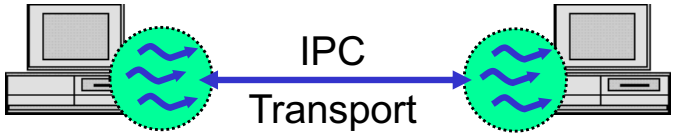
- multithreaded process



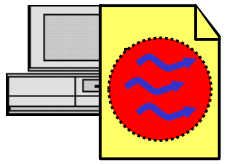
- multiprocess



- multitarget



- postmortem



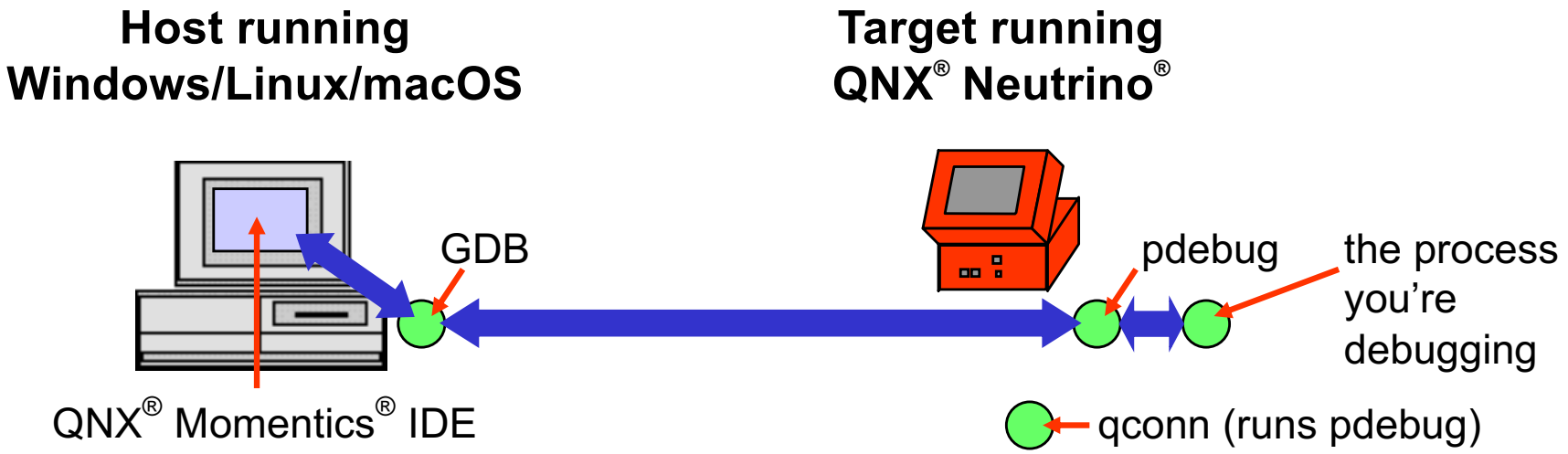
Topics:

- **Overview**
- → **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

qconn and pdebug:

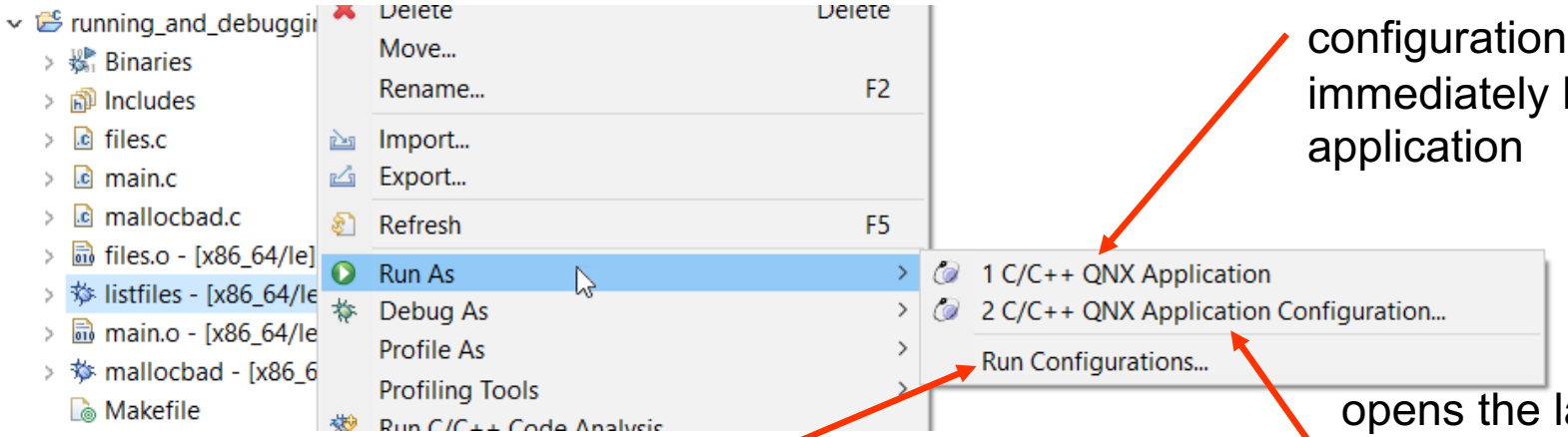
- **`qconn`** is a program that gives all kinds of information to and does all kinds of work for the IDE
  - you should have **`qconn`** running on your target
  - you must be root (user ID 0) to run qconn
- **`pdebug`** is a debug agent, it acts as the interface between the IDE and the process being debugged
  - **`pdebug`** will generally be started as needed
  - **`pdebug`** requires pseudo-terminals (ptys), i.e. **`devc-pty`** must be running, and
  - **`pdebug`** requires a shell (e.g. **`ksh`**) to be available on the target

Processes involved in debugging:



Before running or debugging you need a Launch configuration:

- setup (wizard) is similar for both
- the configuration, once created, can be used for either debugging or running
- how you start depends on whether you initially want to run or debug



creates a default launch configuration and immediately launches the application

opens the launch configuration wizard

opens the launch configuration dialog specific to this executable

Launch Configuration – Launch Types

Several Launch types available –

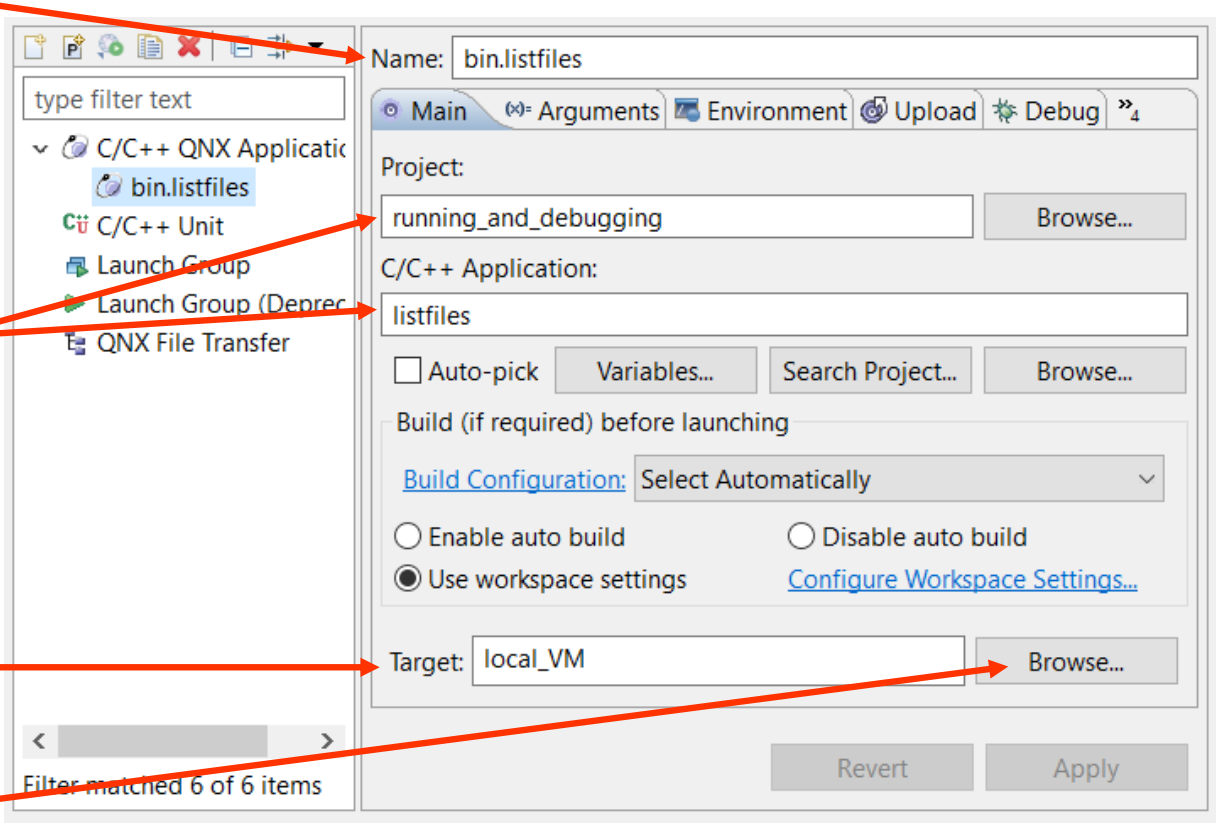
- C/C++ QNX Application
  - connect using IP network to qconn on target
  - most common type
- C/C++ QNX Local Core Dump Debugging
  - postmortem debugging using a core dump
- C/C++ QNX Attach to Remote Process
  - attaching to a running process
- C/C++ QNX Serial Debugging
    - if you only have a serial connection to the target
    - often, you’re better off using PPP in this case

The minimum you need is:

Name for the launch configuration: pick something descriptive

What program to run: specified by Project & Application

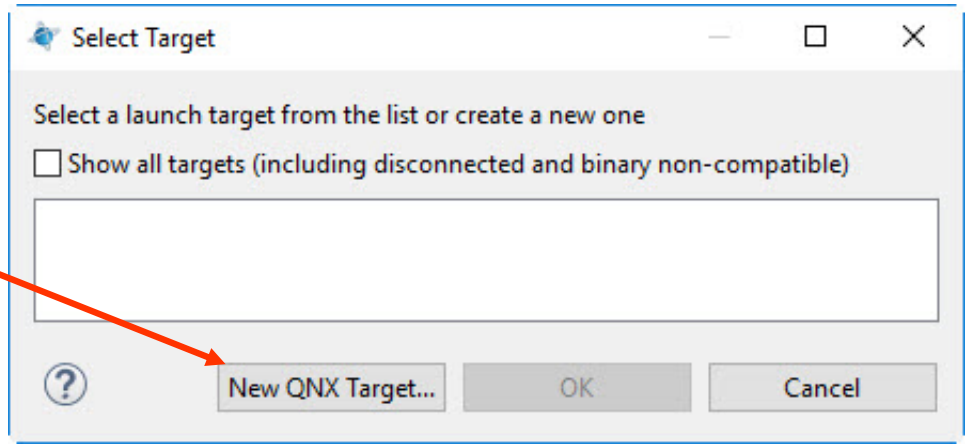
Where to run it: if you haven't created a target project, do it here.



but you may wish to set further things…

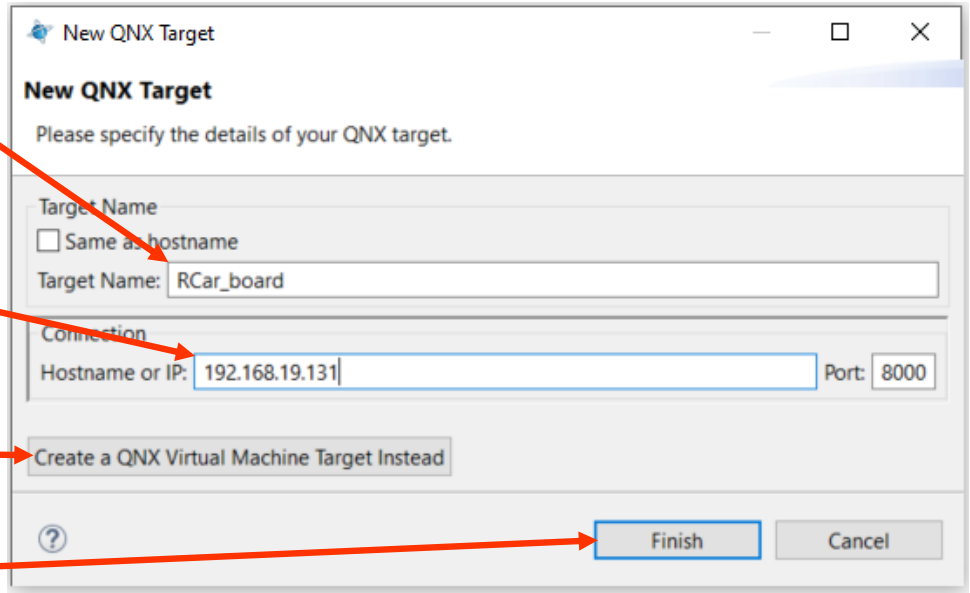
Creating a Target project:

in the Launch wizard, click Browse … then New QNX Target … in the Select Target Wizard



for an actual board or existing VM...

- fill in a name representing your target. This will be the Target project's name
- fill in your target's IP address. `qconn` uses port 8000 by default so you wouldn't normally change this



for a new virtual machine...

choose Finish and your Target project will be created

Create a new VM target and Target project

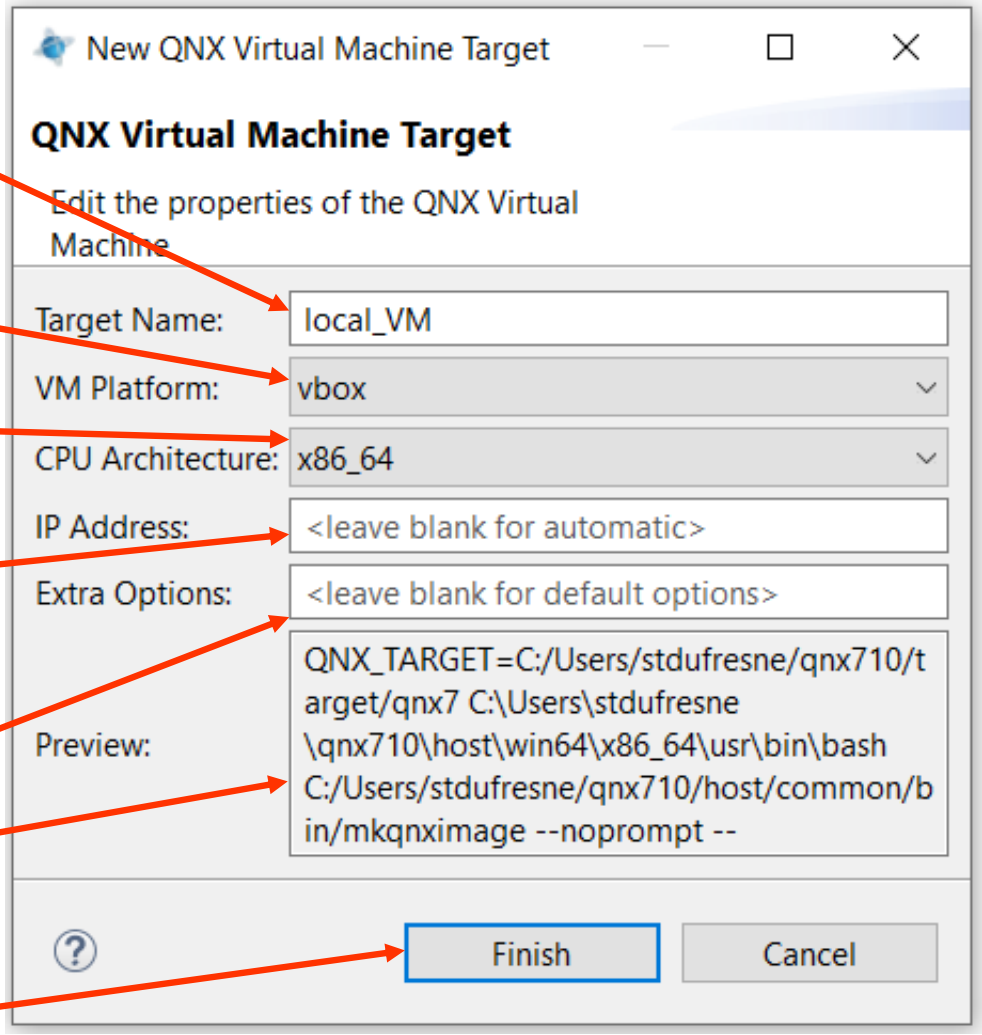
fill in a name representing your target. This will be the Target project's name

the virtual machine that you're using (e.g. vbox for Virtual Box)

the architecture running on your VM

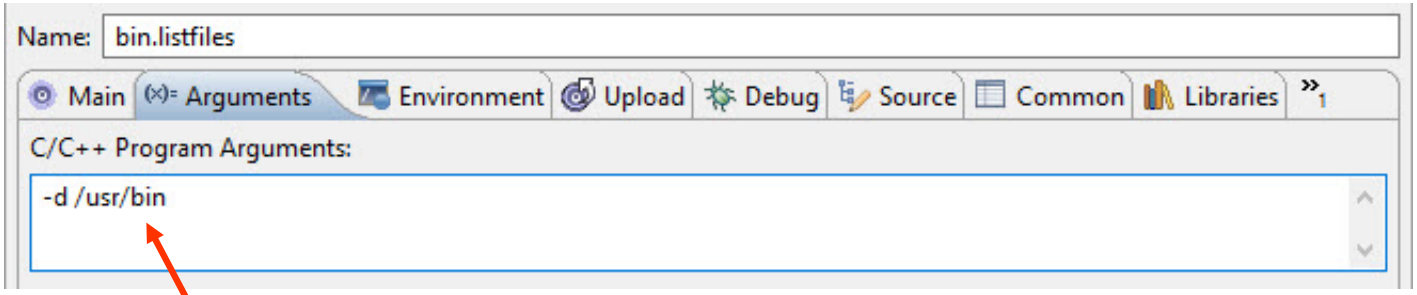
the IP address can be found automatically once/if your VM is running

put extra command line options here to be added to the **`mkqnximage`** command line used by the IDE to create and run an image for your VM.



choose Finish and your Target project will be created and **`mkqnximage`** will create and run the image

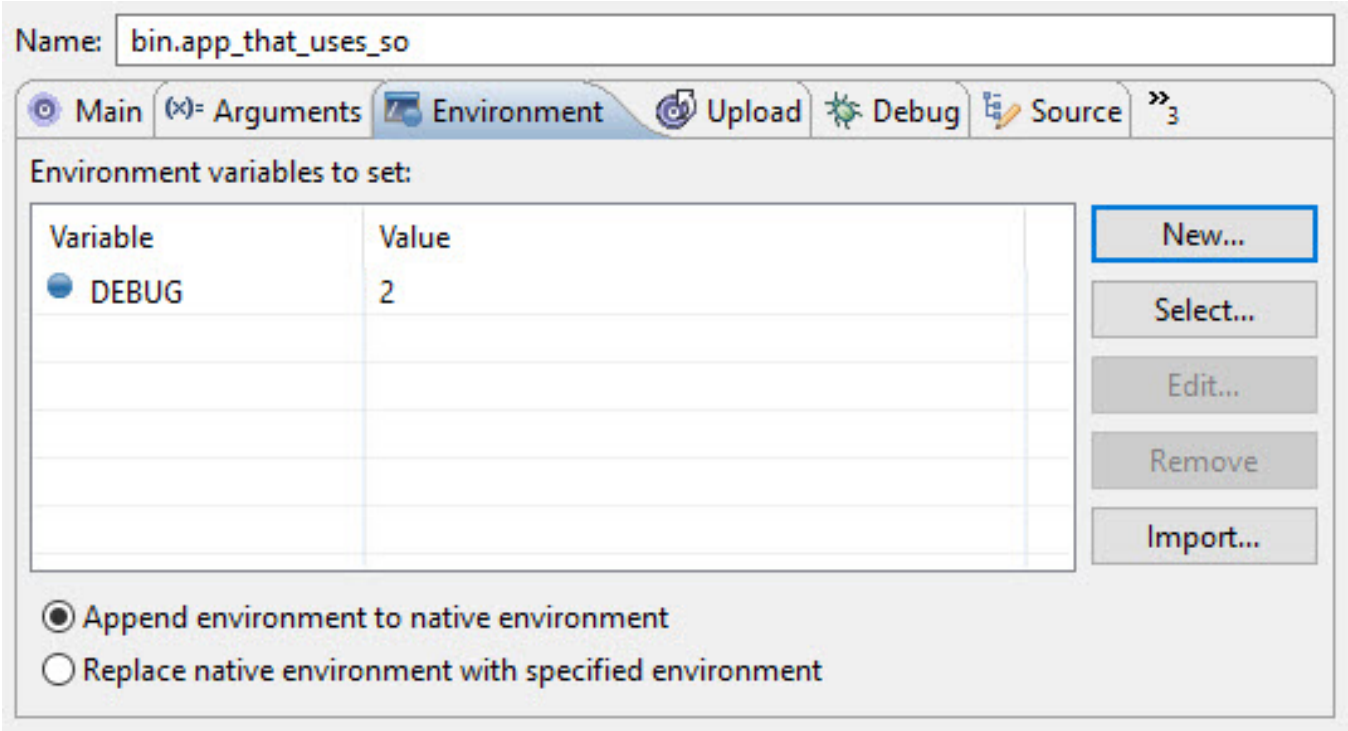
Command line arguments:



- enter command line arguments that you want for your program
  - the special string **`${string_prompt}`** will cause the IDE to prompt you for an argument on each launch
    - useful if you want something different each time you run the program

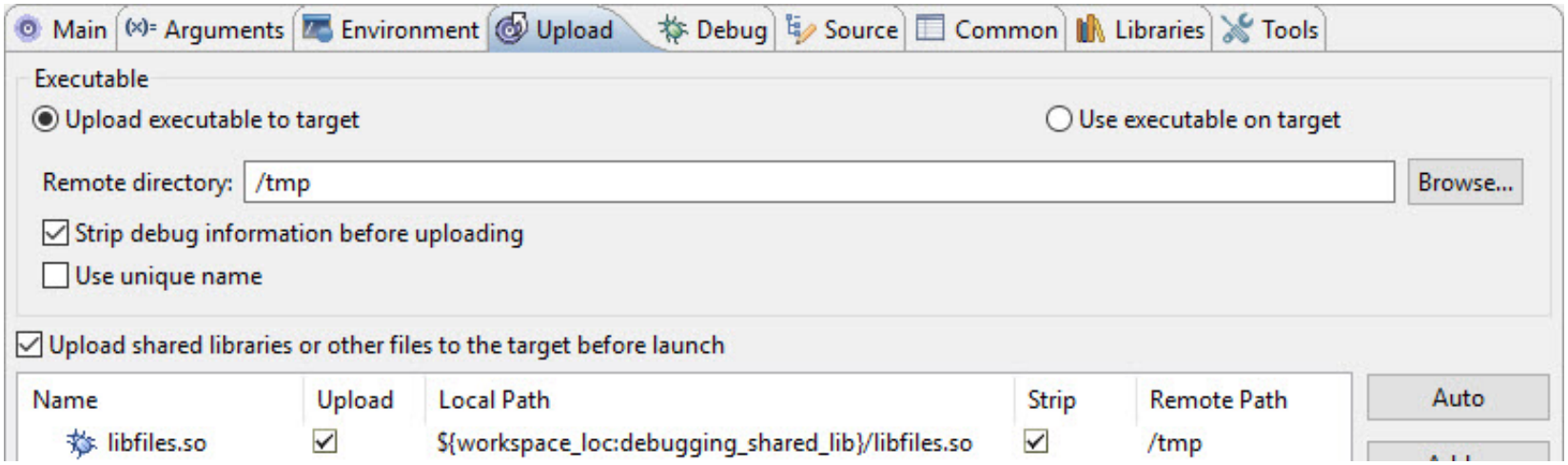
Launch Config - Environment

You can set environment variables, or completely specify the environment:



Launch Config - Upload

You can control how the executable is transferred to the target:



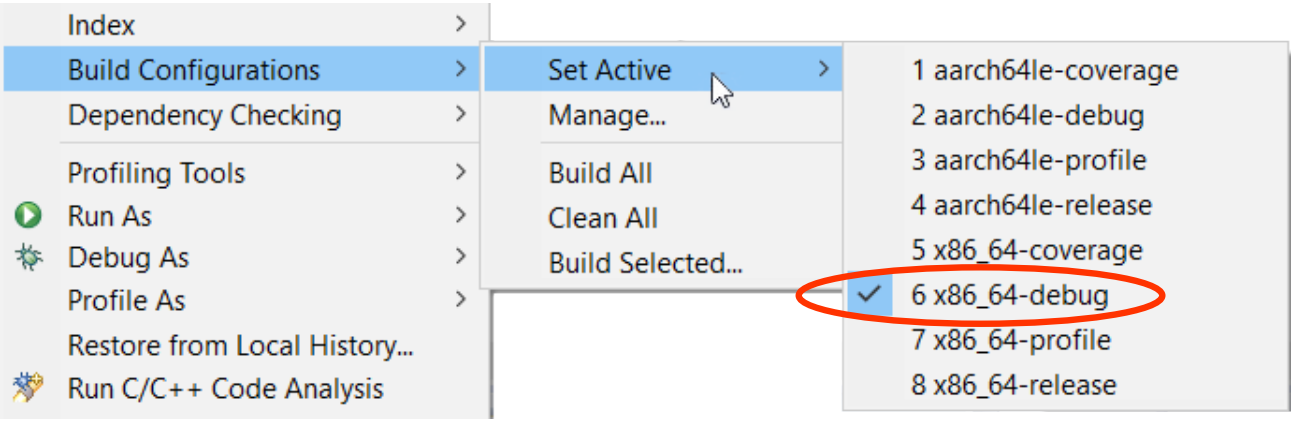
- strip debug because debug information is only needed on the host
- unique name gives the “strange” program names, but helps on shared targets
- remote directory must be writeable

Also, to debug, you must have debug information in your executable:

- to get this debug information:
  - if you create an Empty QNX Project then compile and link with the **-g** option, e.g.:

```
qcc -g -o hello hello.c
```

  - for a QNX Executable Project then Right-Click on the project:



  - or for more control, Right-Click on the project, Properties→C/C++ Build→Manage Configurations

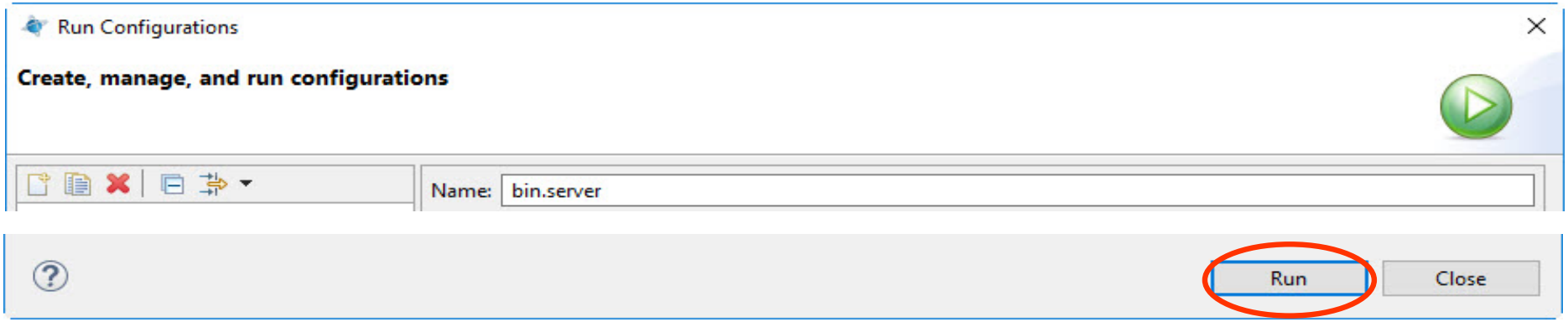
Topics:

- **Overview**
- **Setup**
- → **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

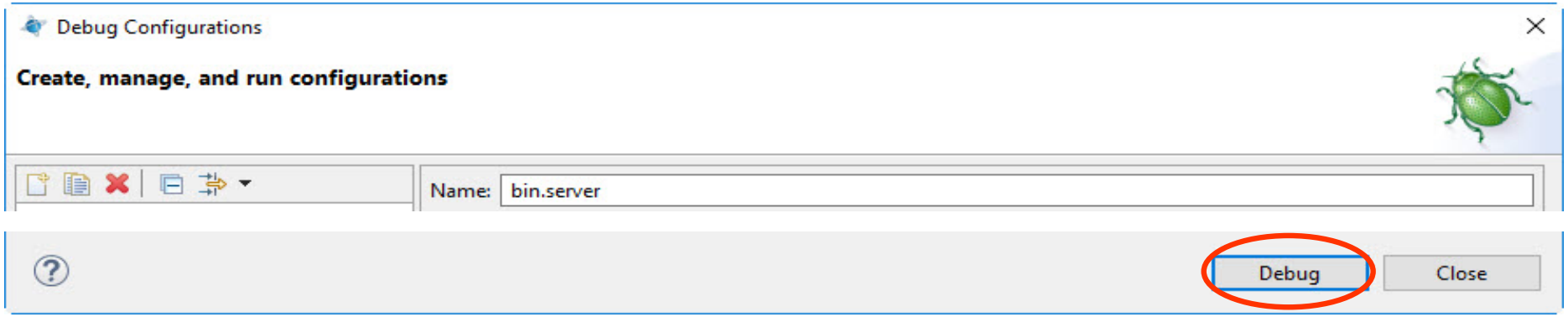
Running or Debugging – The final step

Once you're ready, click:

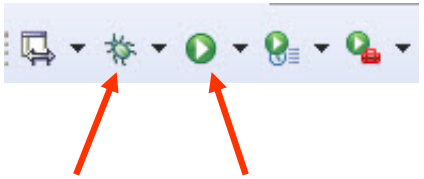
- Run to run your program:



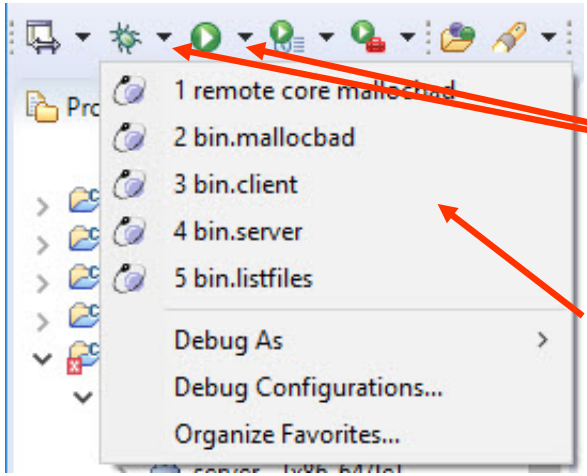
- Debug to debug your program:



If you've already created and used a launch configuration then:



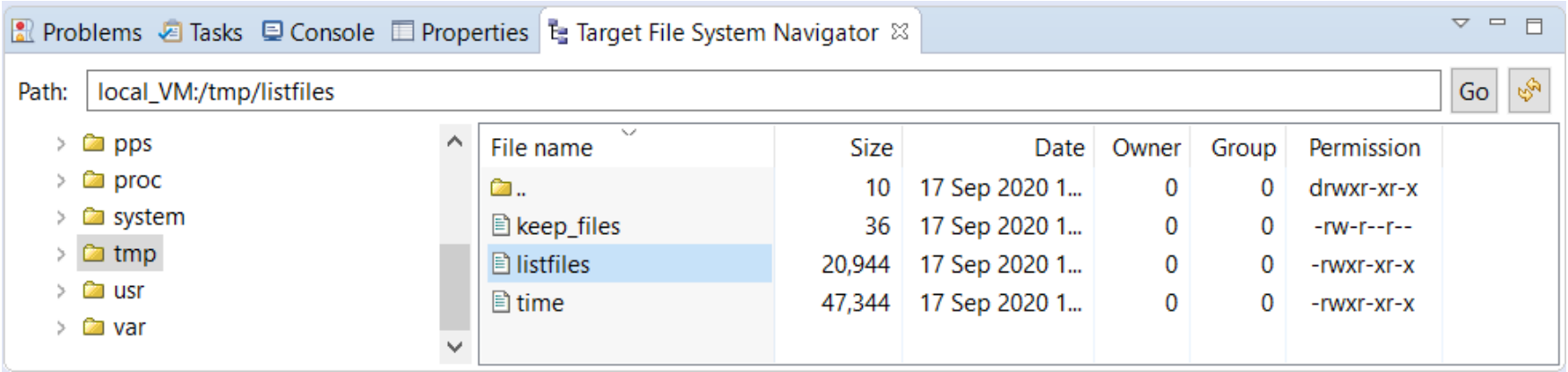
clicking on the Debug or Run buttons will debug or run the last thing you launched (ran or debugged). This is quite useful in a code-debug cycle.



click on this triangle beside the Debug or Run buttons to get:
a history of your last few launches, allowing you to easily select one

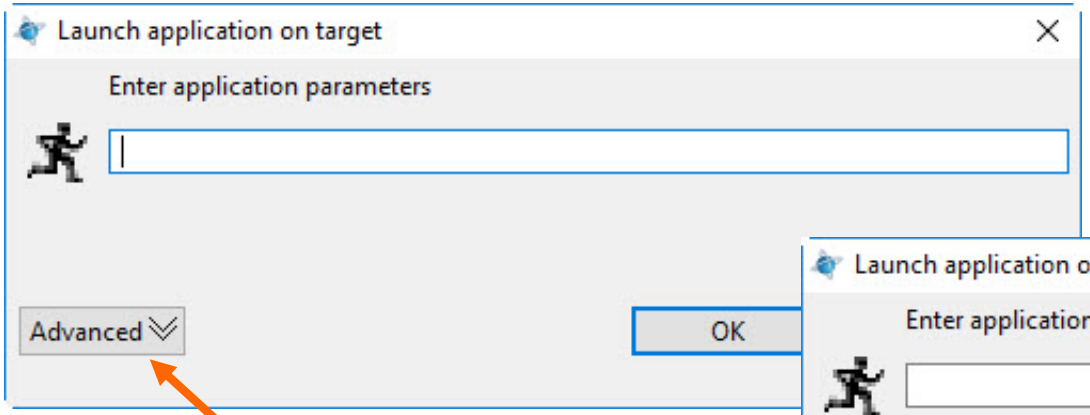
Other ways of running an executable are:

- drag and drop it from the Project Explorer view to the Target File System Navigator view

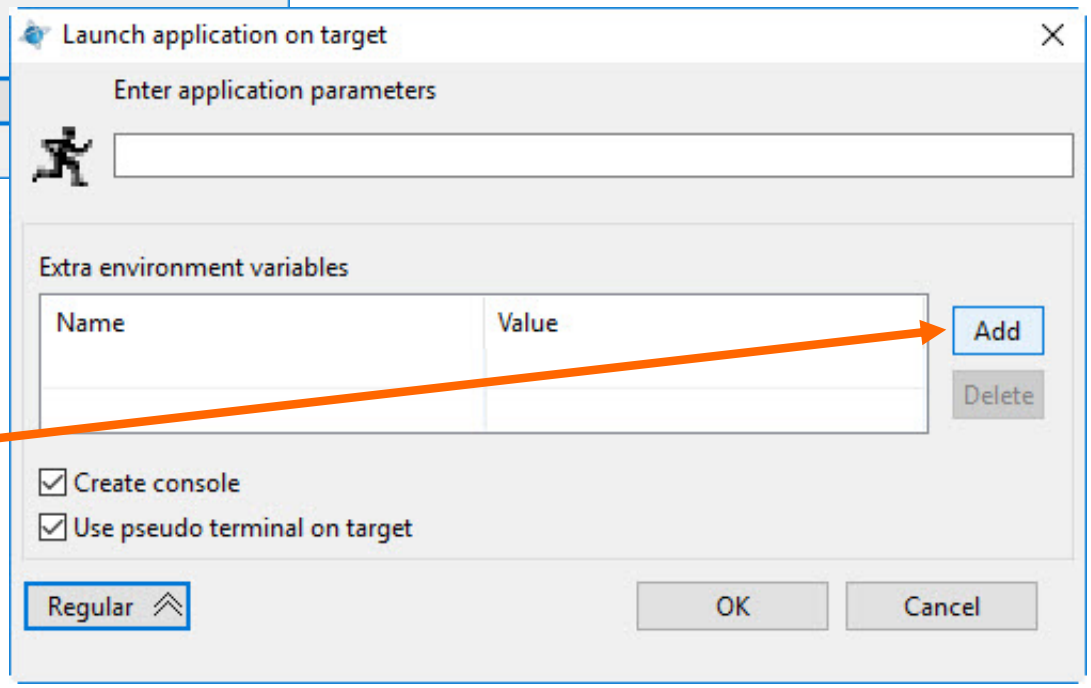


- now that it's on the target, you can:
  - run it from a ssh or telnet session
  - run it from a Terminal view (configured for serial port)
  - double-click on the executable in the Target File System Navigator view and it will run
    - can be run with or without a console…

Double-clicking an executable in the Target File System Navigator gives:



Click Advanced to get:



Click on Add to specify any extra environment variables

Running - Output

When you run a program from the IDE:

- the output goes to the Console View
- it maintains a separate input/output stream for each program that is running/being debugged
- default behavior is that:
  - most recent program that has generated output has its console displayed
  - whenever output occurs, the console view becomes active/moves to the front/appears
- this default behavior can be changed in a variety of ways

The Console view control buttons:

– details on some of the above icons:

Icon	What it does
	Terminate - kill process
	Remove Launch – cleanup from this run/debug session
	Remove All Terminated Launches – clean up all terminated launches
	Show when… – unclick these to prevent console popping up on writes to stdout, stderr
	Display Selected Console – if you have multiple programs, gives you a menu to select which programs output you want
	Pin Console – lock console to selected output stream
	Open Console – provides a new console view

EXERCISE

Running a program:

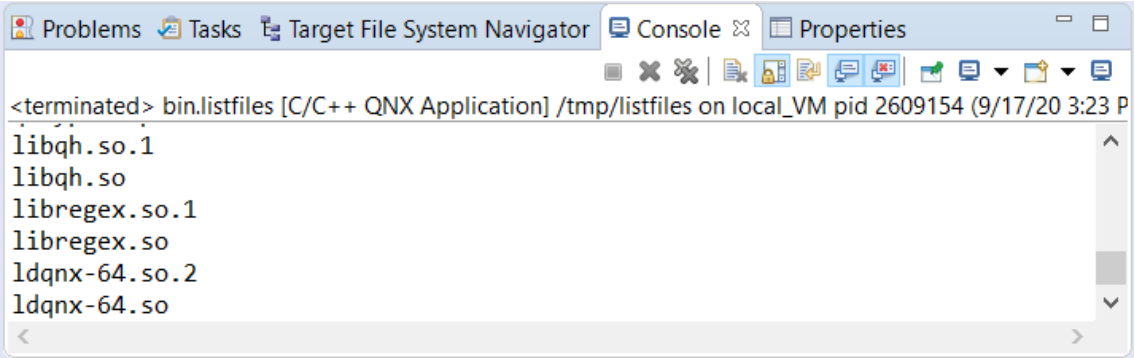
- run the **`listfiles`** program from your **`running_and_debugging`** project

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- → **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

For the remaining slides:

- we'll use the **`listfiles`** program. By default it displays the names of the files in **`/proc/boot`**



- it's in your **`running_debugging`** project
- it's an Empty QNX Project so the **`Makefile`** contains the **`-g`** option to put debugging information into the executable
- it's made from two source files:

  `main.c` - contains *main()* and setup code

  `files.c` - manages filenames

The Debug perspective:

ide-7.1-workspace - running_and_debugging/files.c - Momentics IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Debug | bin.listfiles | on: local_VM | Quick Access

Debug | Project Explorer

bin.listfiles [C/C++ QNX Application]
listfiles [29069314]
Thread #1 unnamed (STOPPED)
read_filenames() at files.c:45
main() at main.c:58 0x8048e
C:\Users\stdufresne\qnx710\host\
/tmp/listfiles on local_VM pid 290

hello.c | main.c | files.c

```
39      while (1)
40      {
41          /*
42           * read a directory entry.  'd
43           * contains 'd_name', the file
44           */
45          direntp = readdir(dirp);
46          if (direntp == NULL)
47              break;
48
49          /
50                                  of pointers
51                                  that if 'fi
52                                  oc().
53
54          tmp = realloc(filenames, (nfil
55          if (tmp == NULL)
56          {
57              *ifilenames = filenames;
```

Variables | Breakpoints | Expressions | Modules

Name	Type	Value
dirname	char *	0x8049256 "/...
ifilenames	char ***	0x8047d20
filenames	char **	0x904c090
tmp	char **	0x904c090
nfilename	int	2
dirp	DIR *	0x9053040
direntp	struct dirent *	0x90530a0

Console | Registers | Problems | Executables | Debugger Console | Memory

bin.listfiles [C/C++ QNX Application] /tmp/listfiles on local_VM pid 29069314 (9/18/20 8:05 AM)

Writable | Smart Insert | 45 : 1 : 922 | 122M of 256M

Debug view: your main control panel

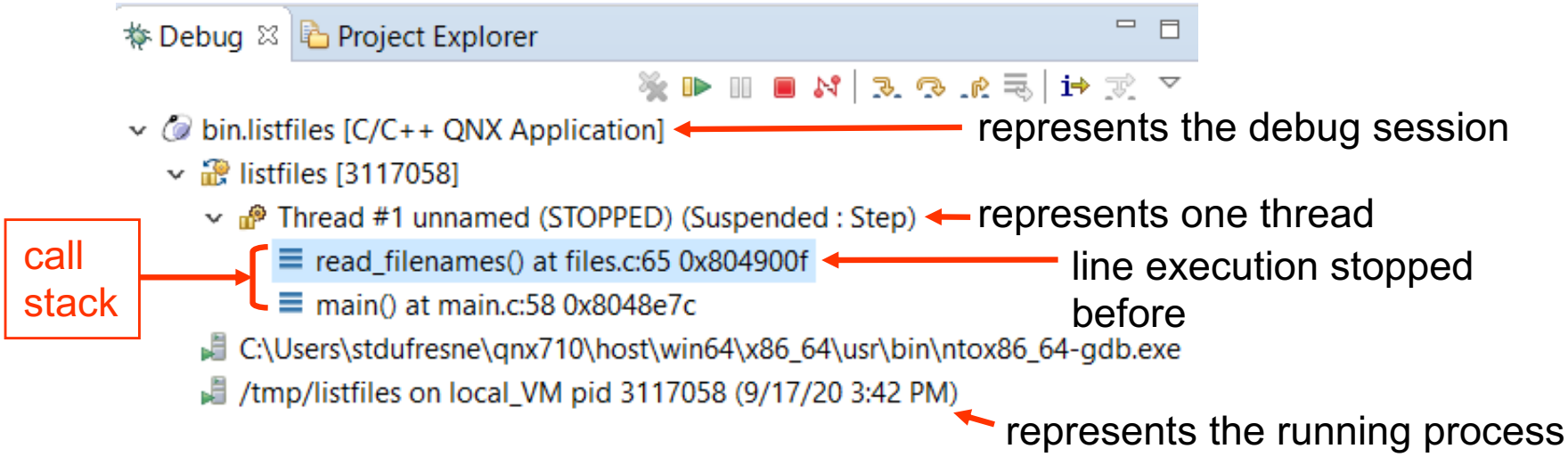
editors with your source

variables and other information

indicates next line to be executed

The Debug view:

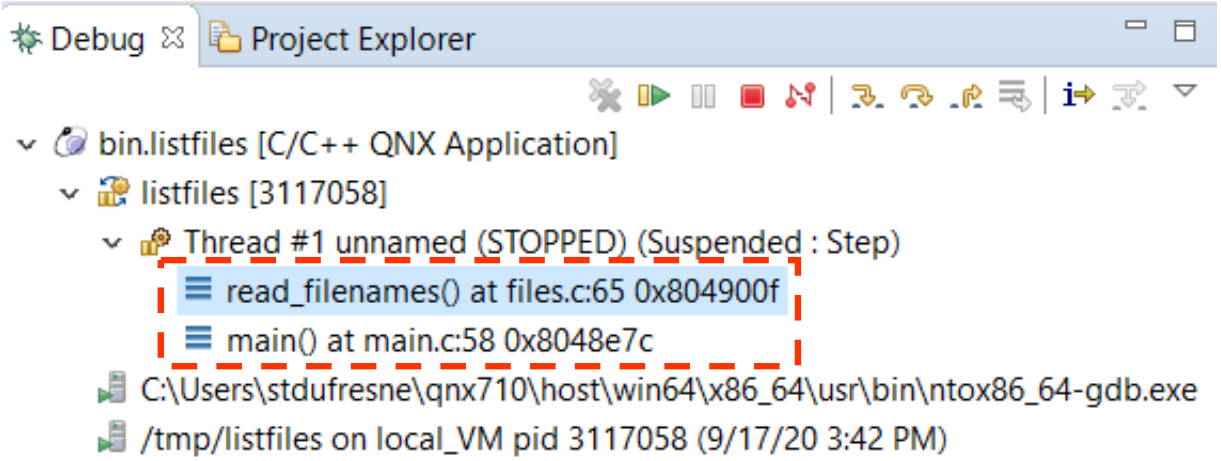
- is the main view for terminating, relaunching program
- for stepping through code
- controls what is in the Variables view, editor, ...



- what you see above will vary based on:
  - how many threads exist
  - what state the program is in (suspended, terminated, …)

The call stack:

- the Debug view shows the call stack



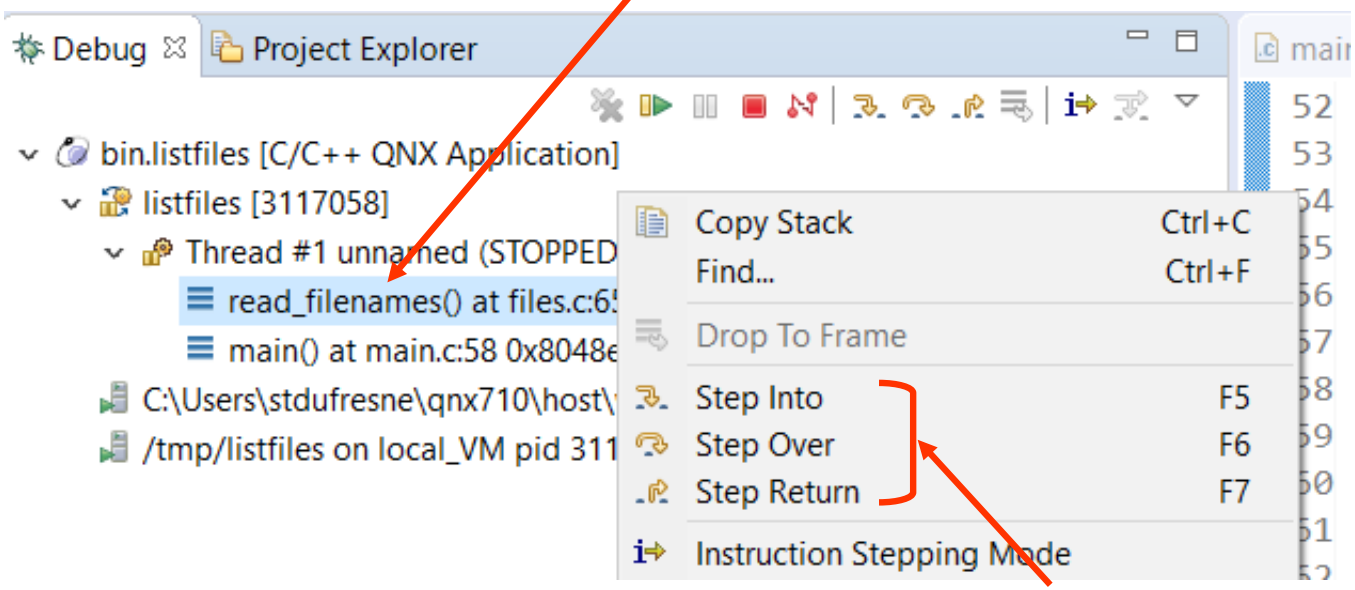
- in the above example, we can see *read_filenames()* was called from *main()*
- as you select the items in the call stack, the Variables view will change to show the variables and their values that are on the stack for that function

The Debug view menu:

- right-clicking on an item will give a menu
- the menu contents will reflect the item

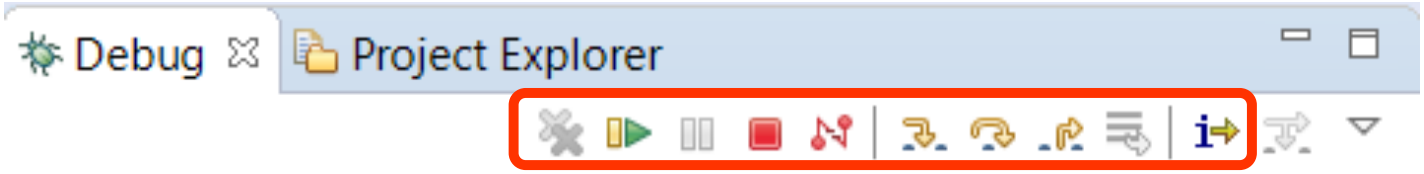
Example:

line execution stopped before



options for controlling which line to execute next

The Debug view control buttons:



– details on some of the above icons:

Icon	Hotkey	What it does
	F8	Resume - run in debugger from current point
		Suspend - regain control of running process
		Terminate - kill process
		Disconnect – detach debugger without killing process
		Remove All Terminated Launches - remove terminated processes from the Debug view
		Instruction Stepping Mode - toggle step by source line/ assembly instruction, most useful with Disassembly view

Topics:

Overview

Setup

Running or Debugging

Overview of the Debug Perspective

Debugging Techniques

- Stepping through code
- Breakpoints
- Viewing and Changing data

Debugging Library Code

Postmortem Debugging

Attaching to a Running Process

Conclusion

When execution stops:

- the line to be executed next is indicated with an arrow

hello.c | main.c | files.c

```
51          * one more.  Note that if 'filenames' is NULL, this will
52          * act like a malloc().
53          */
54         tmp = realloc(filenames, (nfilenames + 1) * sizeof(char *));
55         if (tmp == NULL)
56         {
57             *ifilenames = filenames;
58             return nfilenames;
59         }
60         filenames = tmp;
61
```

- at this point you have a number of execution options:
  - resume, terminate, step into, step over, run to return, and run to C/C++ line
- we've already seen the first two. Let's look at the others...

Stepping and running:

– details on some of the above icons:

Icon	Hotkey	What it does
	F5	Step Into - execute current line and if it's a function that you have debugging information for then make the new current line the first line in the function (i.e. step into it)
	F6	Step Over - execute current line and if it's a function, regardless of whether we have debugging information for it, don't step into it. Execute the function and make the next line the current line instead (i.e. step over the function)
	F7	Step Return – finish this function

Run to C/C++ line:

current line

desired current line

hello.c | main.c | files.c

```
61
62          /*
63           * allocate room for the filename
64           */
65          filenames[nfilenames] = malloc(str
66          strcpy(filenames[nfilenames], dire
67
68          nfilenames++; /* new number of fil
69      }
70      closedir(dirp);
71
72      *ifilenames = filenames;
73      return nfilenames;
```

- Refactor
- Declarations
- References
- Search Text
- Resource Configurations
- Step Into Selection Ctrl+F5
- Run to Line Ctrl+R
- Move to Line
- Resume at Line
- Add Watch Expression...

1 click on the line you want to run to, then right-click for the menu

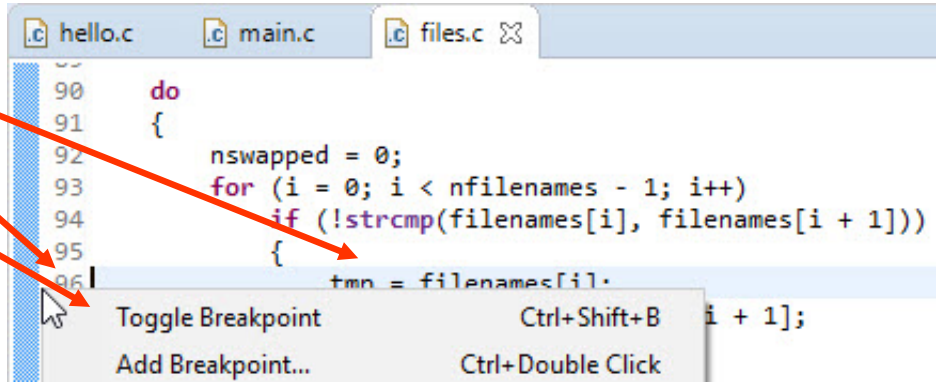
2 from the menu, choose Run To Line

Topics:

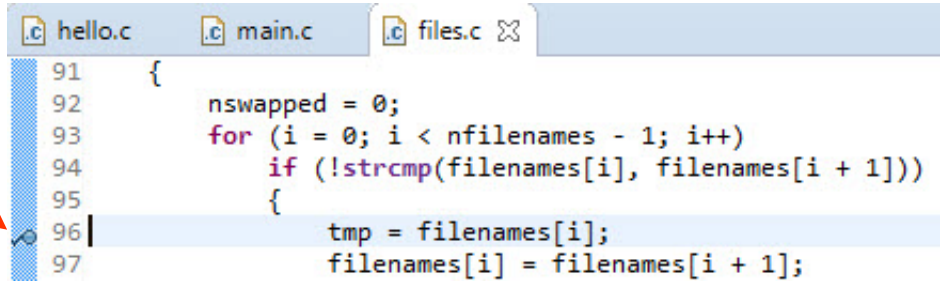
- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

A breakpoint is a place you want the program to stop and return control to you:

to set a breakpoint on this line, right-click in the grey margin here and choose Toggle Breakpoint
Or double-click in the grey margin at the line you want



the end result is this indicator telling you that a breakpoint has been set for this line

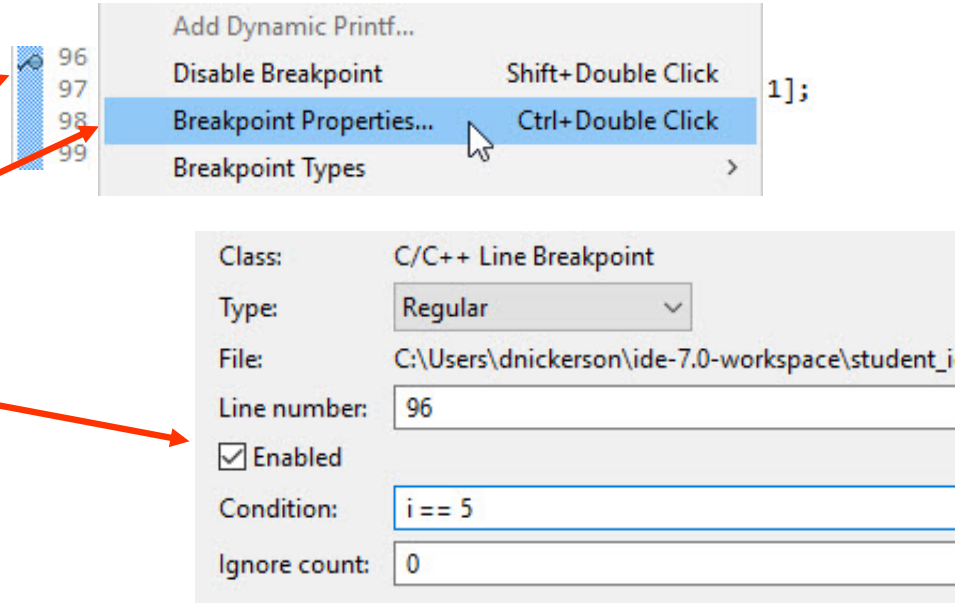


– if you resume execution from wherever it is stopped, it'll run until it either terminates or reaches the above line. If it reaches the above line then it would stop before executing the line.

Breakpoints can also be conditional:

– execution would stop at the breakpoint only if a specific condition has been met

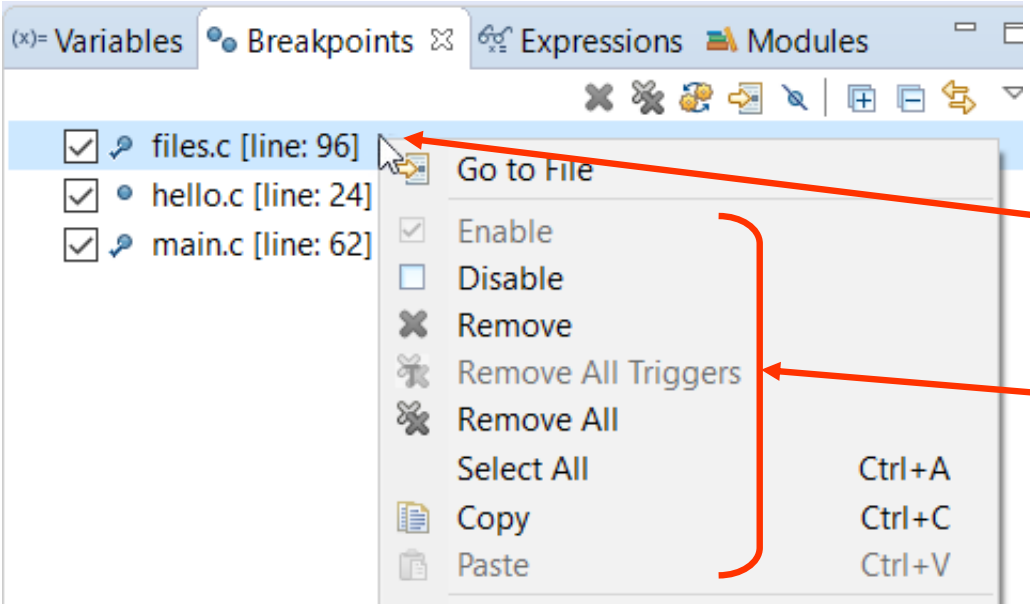
right-click on an existing breakpoint indicator and choose Breakpoint Properties…
… the Breakpoint Properties window appears



– the Condition and Ignore count are evaluated by **gdb** on the host each time the break point is hit:

- if the condition is not true, **gdb** resumes the process
- if the count has not been reached, **gdb** resumes the process

The Breakpoints view lists your breakpoints:



by right-clicking on a breakpoint you get this menu…
… with these useful options

– breakpoints live past the end of a debug session:

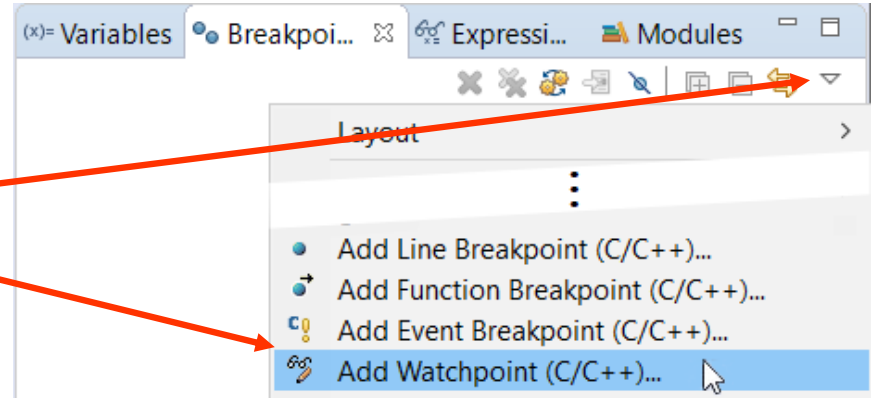
- you don't have to add them again
- the icon changes:  versus 

Watchpoints:

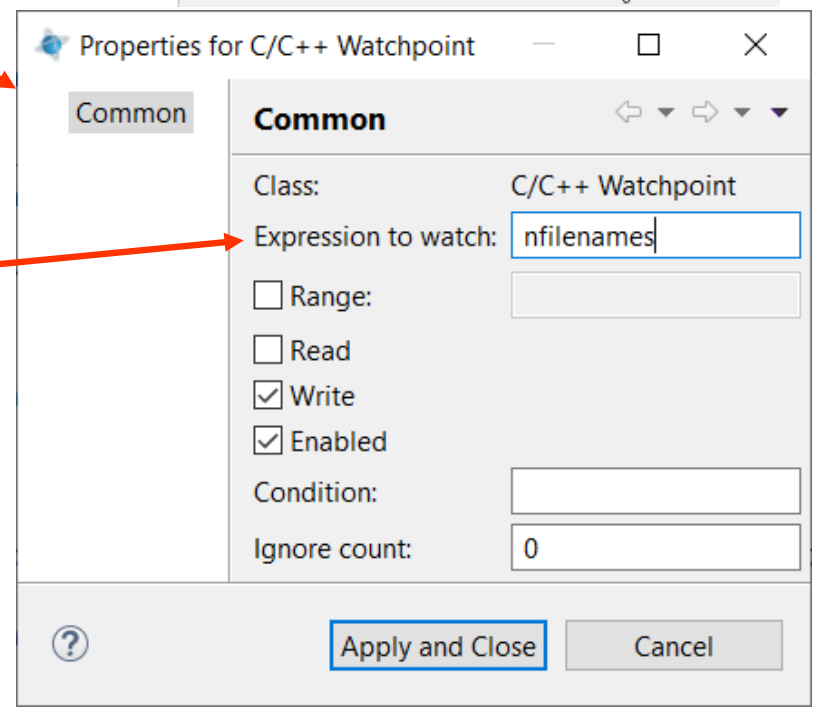
- stop when the contents of a memory address is updated
  - access watchpoints are available, but performance is usually unacceptable (done in software, not hardware)
- may not be supported on all hardware platforms
  - or may only be supported as software watchpoints
- stop after the memory has been accessed
  - not before a line is executed as a breakpoint does
- added through the breakpoints view
- can watch a:
  - global variable
  - address

Create a watchpoint in the breakpoints view:

Select “Add Watchpoint…” from the “View Menu”

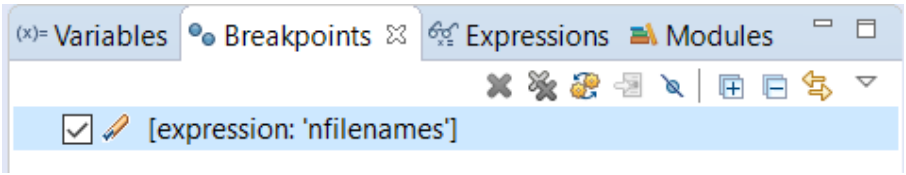


The Add Watchpoint window appears



Enter an expression referencing a global variable or an address:

The result:

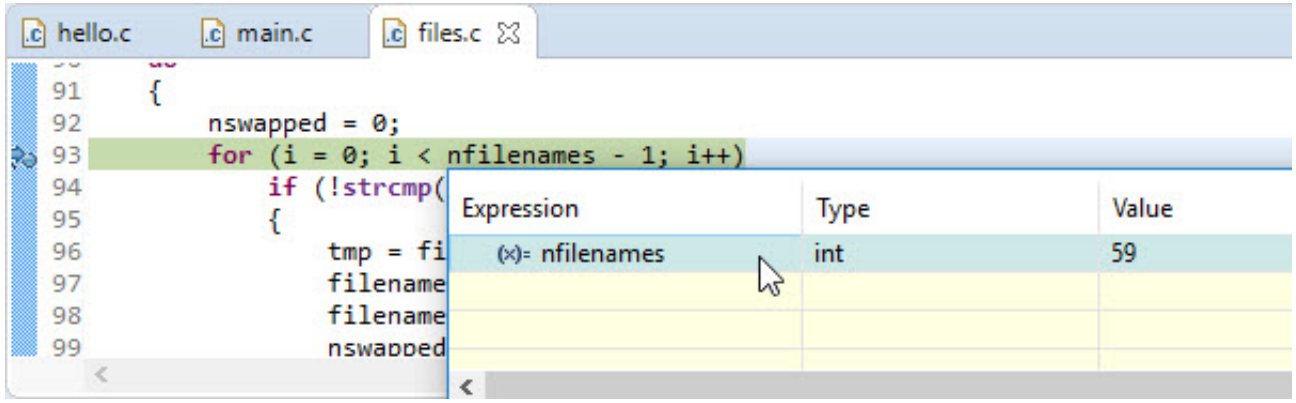


Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

Looking at variables:

- while in an editor, hold the mouse pointer steady over a variable and a balloon will pop-up



- otherwise use the:
  - Variables view for local variables
  - Expressions view for more complex expressions (which can contain variables)

Various data views are available:

- by default you get Variables, Breakpoints, Registers and Memory views:

(x)= Variables | Breakpoints | Registers

Name	Type	Value
filenames	char **	0x805ad70
*filenames	char *	0x804d088 "procnto-smp-in...
(x)= **filenames	char	112 'p'
(x)= nfilenames	int	59
tmp	char *	0x8249ec8 "\b\235$\b"
(x)= i	int	2825072

Console | Memory | Tasks | Problems | Executables | Debugger Console

Monitors

*filenames

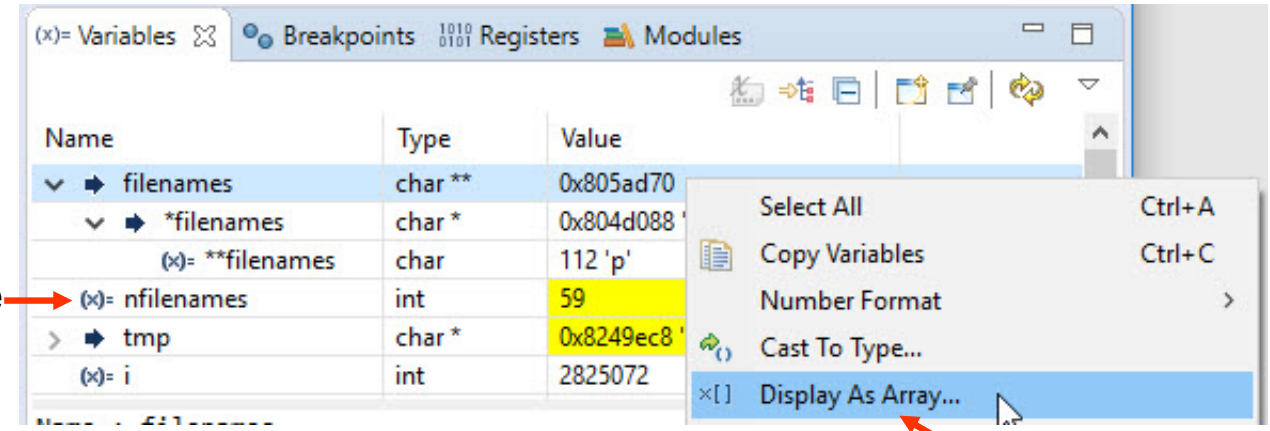
*filenames : 0x804D088 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
000000000804D080	C04C6157	0D0A0000	70726F63	6E746F2D
000000000804D090	736D702D	696E7374	72000000	00000000
000000000804D0A0	A2FFFFFF	FFFFFFFF	184C6157	0D0A0000

- another useful view to add is the Expressions view

The Variables view:

yellow means its value changed while we were stepping



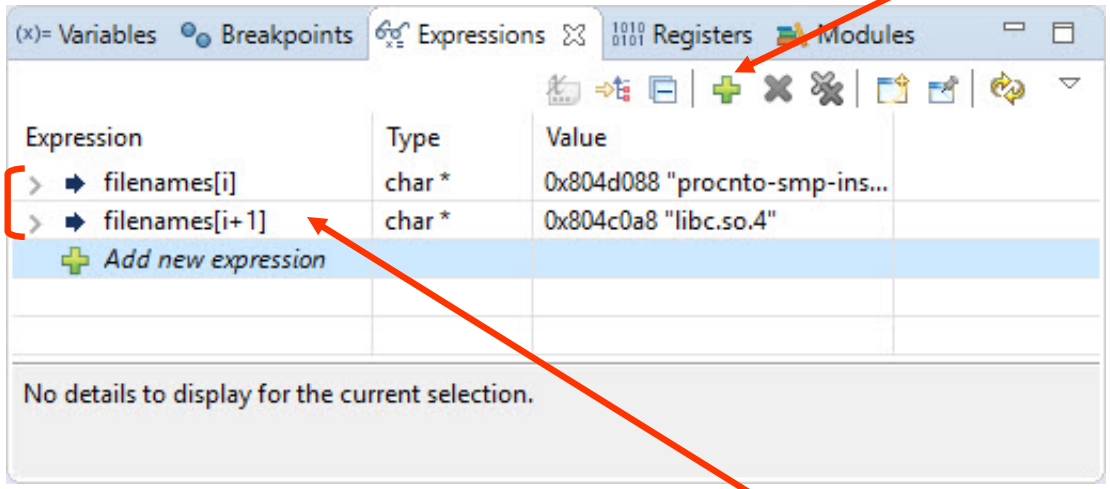
allows a pointer to be interpreted as an array

- shows local variables in current stack frame
- global variables are not displayed
- values can be changed in Value field

The Expressions view:

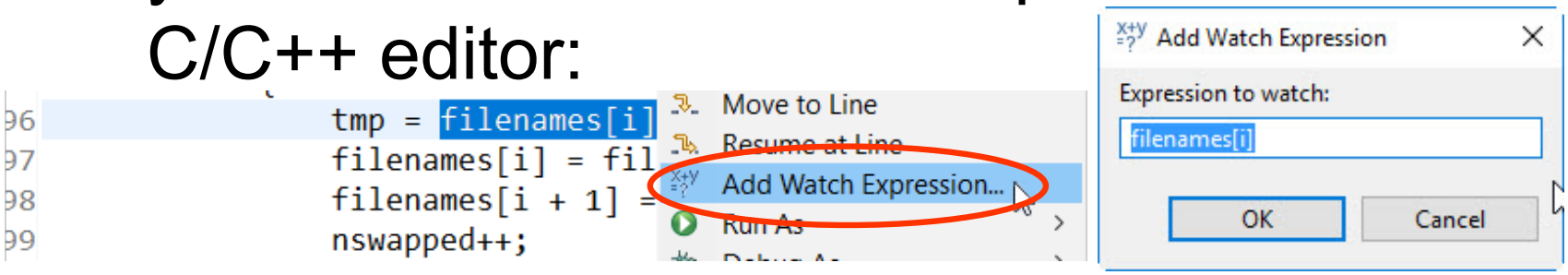
you can create new expressions

expressions will be evaluated as you step



or modify existing ones

– you can also add watch expressions from the C/C++ editor:



The Memory view:

click to enter an address or a variable that equates to an address

can track multiple memory areas, click to select each

Console | Memory | Tasks | Problems | Executables | Debugger Console

Monitors

*filenames

&nswapped

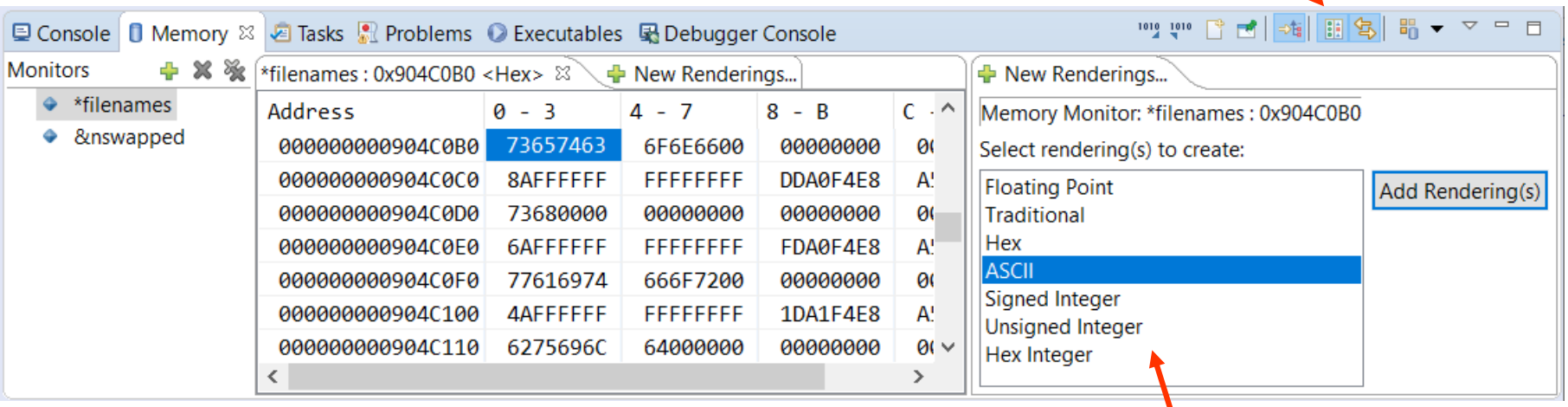
*filenames : 0x804D088 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
000000000804D080	AB80DD61	50350000	70726F63	6E746F2D
000000000804D090	736D702D	696E7374	72000000	00000000
000000000804D0A0	A2FFFFFF	FFFFFFFF	7380DD61	50350000
000000000804D0B0	6C696274	72616365	6C6F672E	736F2E31
000000000804D0C0	00C00408	00000000	7AFFFFFF	FFFFFFFF
000000000804D0D0	1B80DD61	50350000	6C696273	6C6F6732
000000000804D0E0	70617273	652E736F	2E310000	00000000
000000000804D0F0	52FFFFFF	FFFFFFFF	2380DD61	50350000
000000000804D100	6C696273	6C6F6732	70617273	652E736F
000000000804D110	00000000	00000000	2AFFFFFF	FFFFFFFF
000000000804D120	C381DD61	50350000	6C696273	6C6F6732
000000000804D130	7368696D	2E736F2E	31000000	00000000
000000000804D140	02FFFFFF	FFFFFFFF	9381DD61	50350000

you can type here to change the memory

Adding an additional memory rendering:

Click to open additional rendering pane



Select the type of rendering to create

Basic debugging:

- the **`listfiles`** program from your **`running_and_debugging`** project contains a bug

its output is supposed to be sorted alphabetically!

Console | Memory | Tasks | Problems | Executables

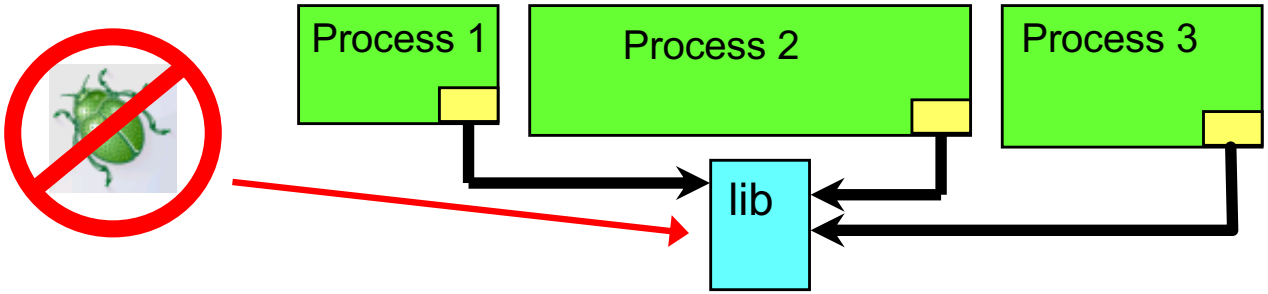
```
<terminated> bin.listfiles [C/C++ QNX Application] /tmp/listfiles on
setconf
sh
ability
waitfor
build
post_startup.sh
procnto-smp-instr
libc.so.5
mount_fs.sh
qcrypto.conf
```

- use the debugging techniques you learned to find the bug

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- → **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

You can debug code in libraries:

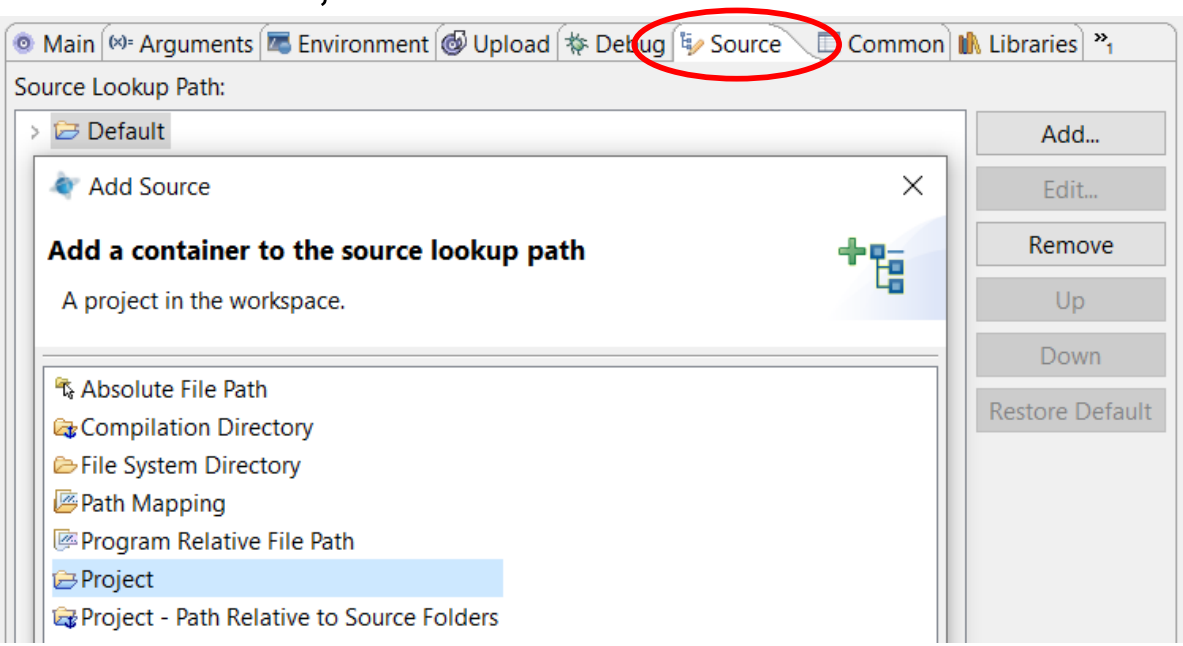


– three cases:

1. library that is linked during build (static)
2. library that is loaded/linked during initial execution of the application (shared object)
3. library that is loaded on demand using *dlopen()* (shared object/dll)

1. Debugging a library that is linked during build (static)

- create a normal debug launch
- in the ‘Source’ tab, tell IDE where the source code is:



- if you don’t do this, and try and debug into the library code
  - it might “just work”, or
  - the IDE may ask you where the source is

Use the debugger on static library code:

- build, and create a launch configuration for the project:
  - **`debugging_app_that_uses_static_lib`**
- try using the debugger to:
  - use breakpoints
  - single step
  - monitor the values of variables
- the following functions are located within the library:
  - *read_filenames()*
  - *sort_filenames()*
  - *display_filenames()*
  - *cleanup_filenames()*

Debugging shared objects (cases 2. and 3.):

2. library that is loaded/linked during initial execution of the application (shared object)
3. library that is loaded on demand using *dlopen()* (shared object/dll)

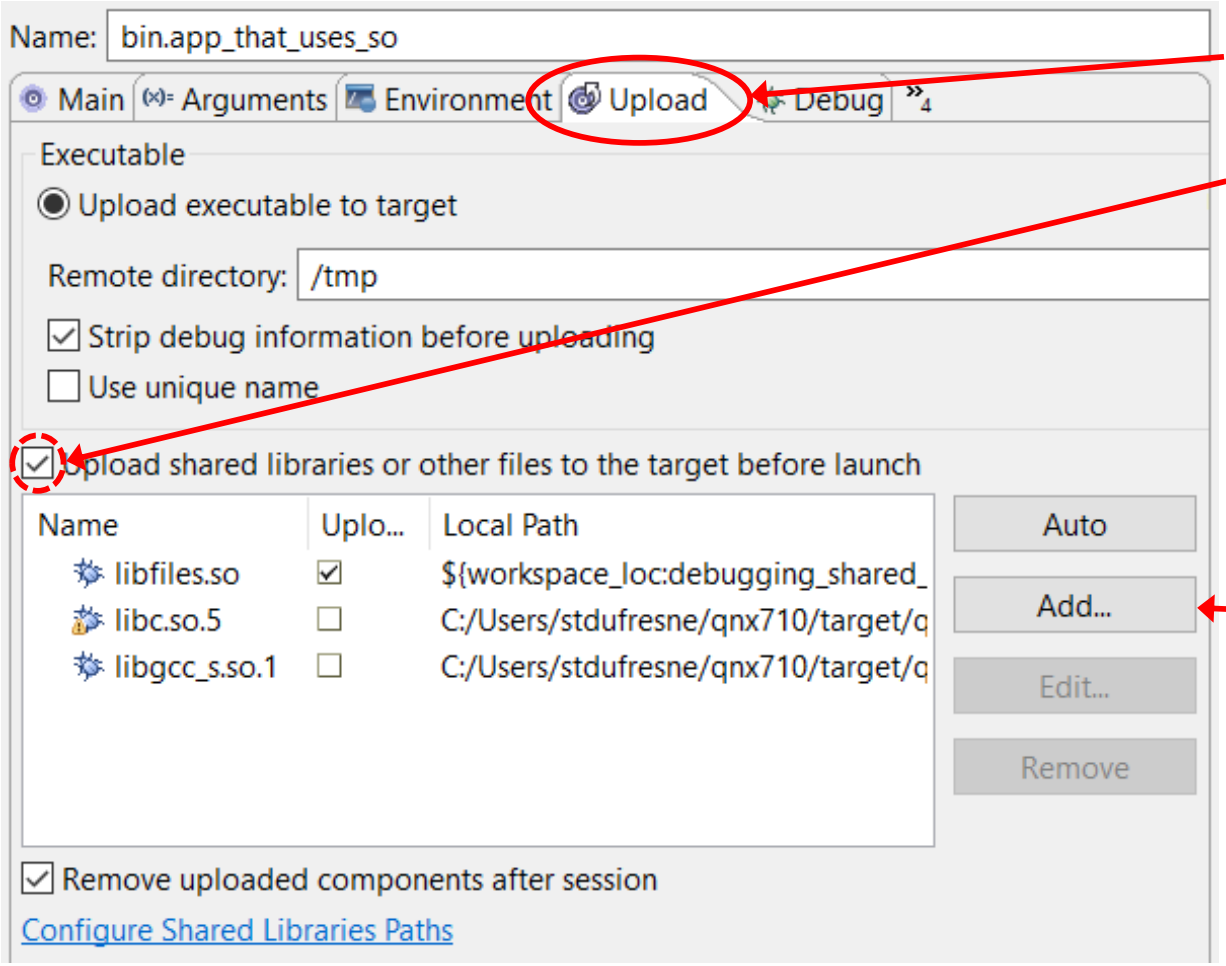
– both require the same setup

– create a normal debug launch

– follow the steps outlined on the next few pages to set up the tabs in the launch configuration

Debugging shared objects:

– create a debug launch, and…

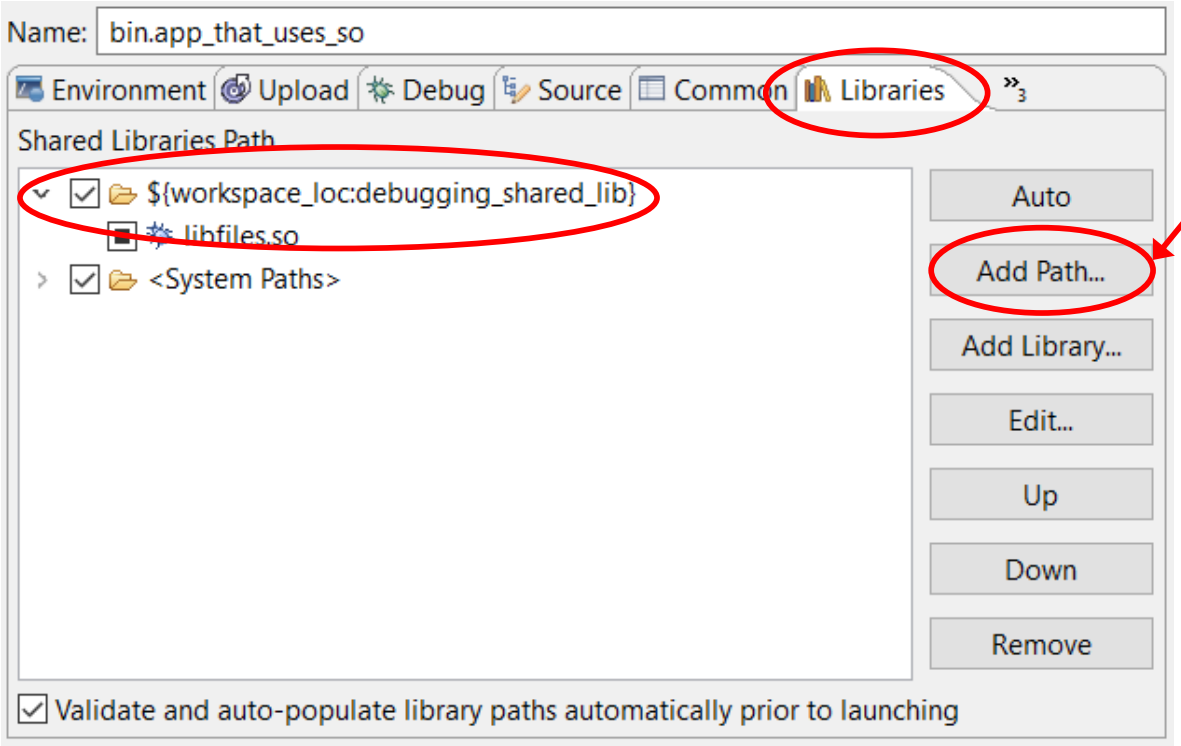


1. go to Upload tab
2. check, telling IDE to upload the library along with binary
3. if it didn't automatically find the library:

Add… lets you manually specify where to find the library

continued…

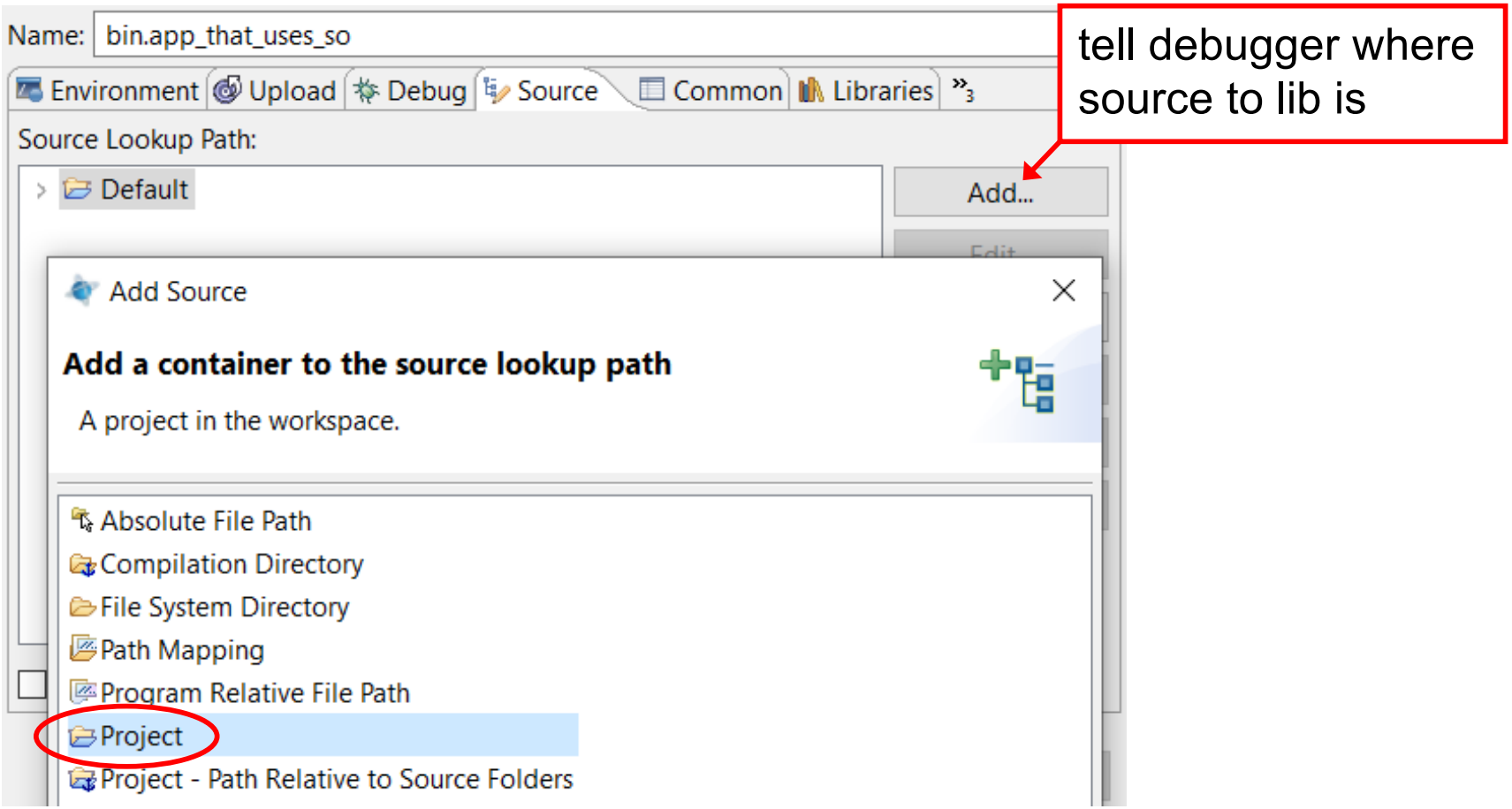
Continue setting up the launch configuration as follows (continued):



tell debugger where lib lives on host machine

continued...

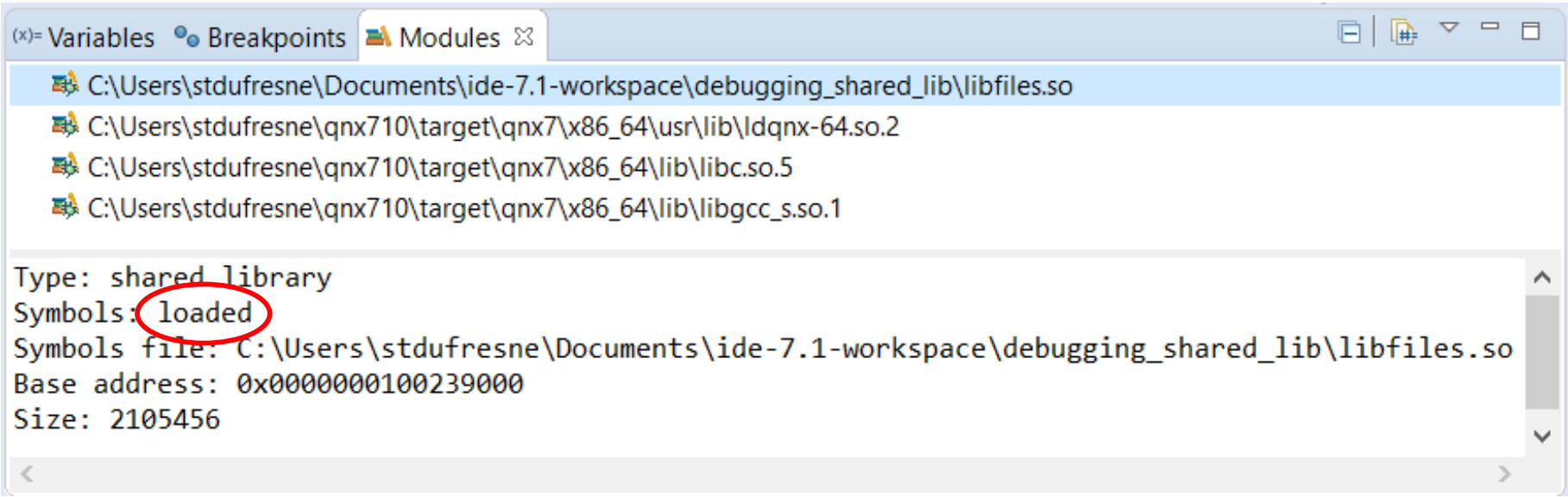
Continue setting up the launch configuration as follows (continued):



Debugging Shared Objects

When debugging a shared library:

- you can use the Modules view to see if the lib has been loaded
- if there is a problem, check upload, debugger, and source tabs in your launch, as described



EXERCISE

Use the debugger on shared object code:

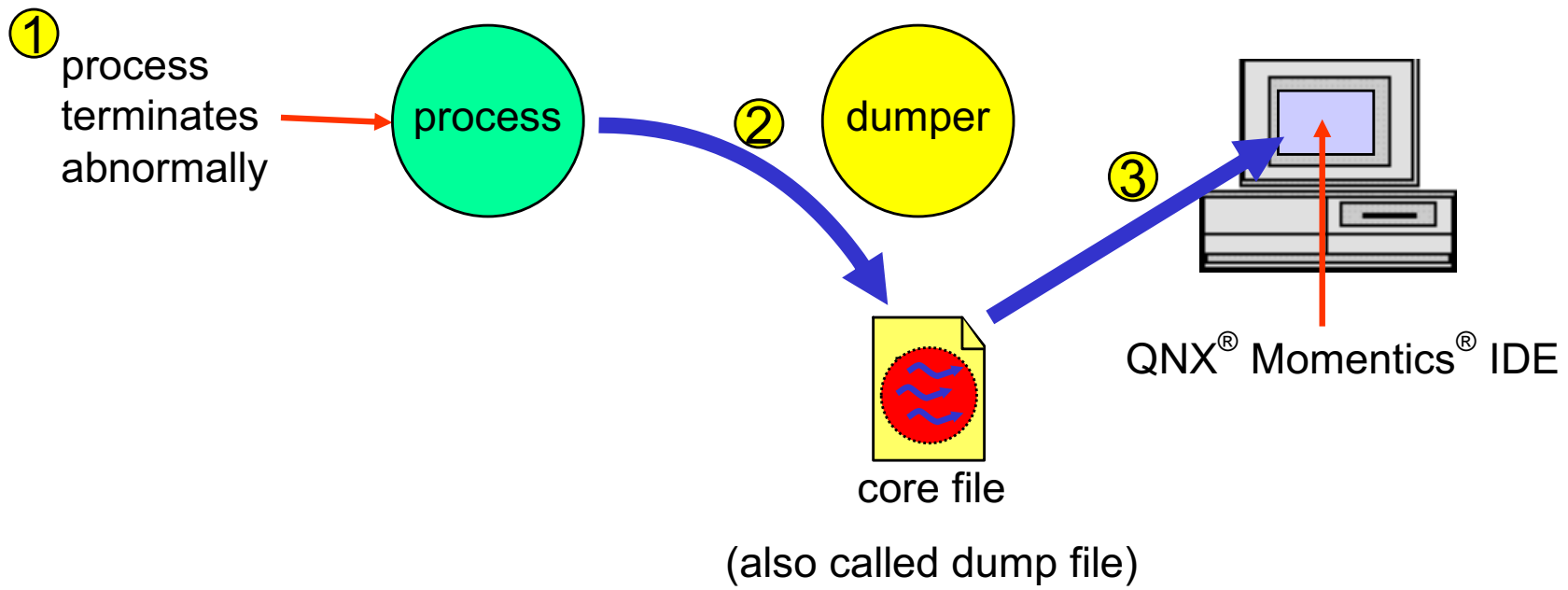
- there are 2 application projects:
  - `debugging_app_that_uses_shared_lib`
  - `debugging_app_that_uses_shared_lib_as_dll`
- they both make use of the library project:
  - `debugging_shared_lib`
- create a launch configuration to debug the library code
- try breakpoints and stepping into the library functions:
  - `read_filenames()`
  - `sort_filenames()`
  - `display_filenames()`
- pick 1 of the 2 cases, or try both, if you have time

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- → **Postmortem Debugging**
- **Attaching to a Running Process**
- **Conclusion**

Postmortem debugging is:

- a way of examining the state of a process at the time that it terminated abnormally
- it works as follows:



The abnormal termination occurs when:

– the process terminates due to receipt of one of the following signals:

SIGABRT	Program-called abort function
SIGBUS	Bus error, address alignment error
SIGEMT	Emulator Trap instruction
SIGFPE	Floating-point error or integer division by zero
SIGILL	Illegal instruction executed
SIGQUIT	Quit
SIGSEGV	Invalid memory (segment) reference
SIGSYS	Bad argument to a system call
SIGTRAP	Trace trap (not reset when caught)
SIGXCPU	Exceeded the CPU time limit
SIGXFSZ	Exceeded the file size limit

Running dumper:

- **`dumper`** must already be running when the process terminates
- examples:
  - **`dumper`** will dump core files to your home/login directory or /tmp if home/login doesn't exist

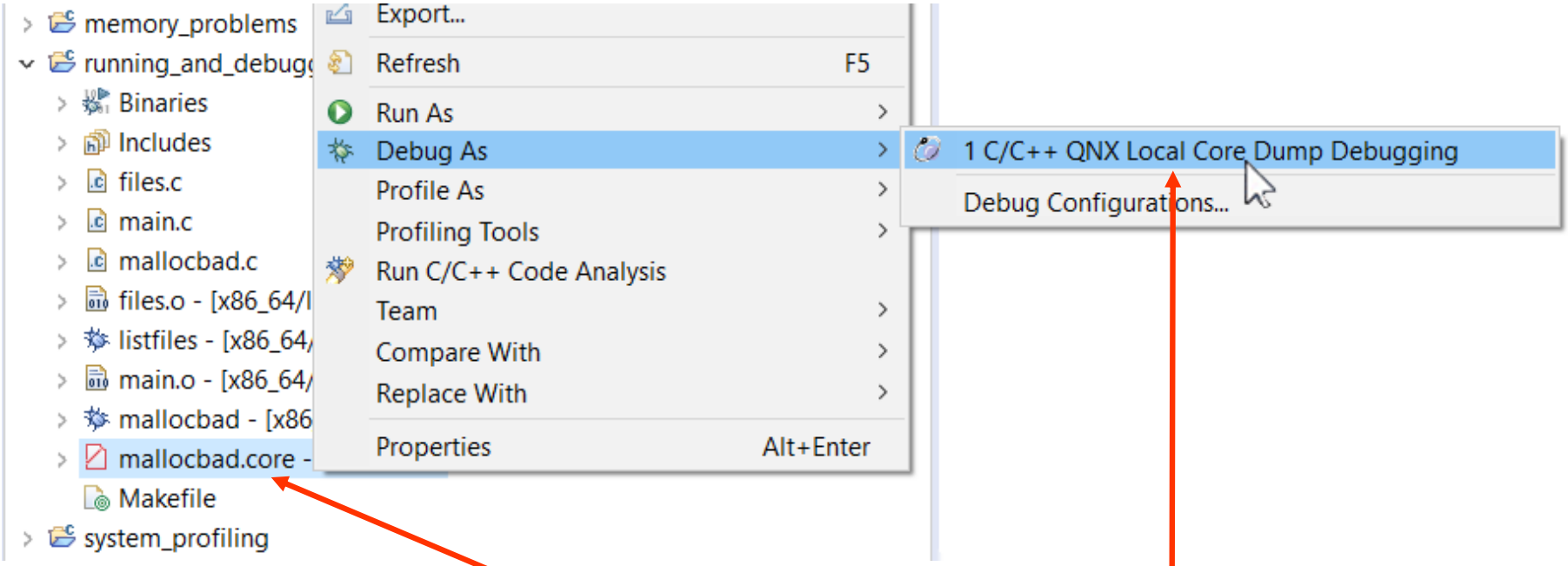
    `dumper &`
  - **`dumper`** will dump core files to **`/var/dumper/`**

    `dumper -d /var/dumper &`
- or you can dump a running process
  - **`dumper -p pid`**
  - will stop, dump, and release the process

Postmortem Debugging:

- we'll use the **`mallocbad`** executable that is in your **`running_and_debugging`** project
- run **`mallocbad`**
  - it very quickly crashes
- a core file called **`mallocbad[something].core`** will be produced on the target, to see where:
  - run **`pidin arg`** from a terminal or use the IDE's Process Information view in the System Information perspective to see if **`dumper`** has a **`-d`** command line argument
  - if not then see your home/login directory (e.g. **`/root`**)
- we'll copy the core file to the **`running_and_debugging`** project
  - we'll use the Target File System Navigator view

Next, launch the debugger with the core file:



right-click on the core file and choose
Debug As → C/C++ QNX Local Core Dump Debugging

the IDE will try to find any source and libraries and then put you directly into the debugger

And in the Debug perspective...

ide-7.1-workspace - running_and_debugging/mallocbad.c - Momentics IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Run | mallobad PMD | on: local_VM | Quick Access

Debug | Project Explorer

- mallobad PMD [C/C++ QNX Local Core Dump
  - mallocbad
    - Thread #1 unnamed (Suspended : Signa
      - strlen() at strlen.si:65 0x1002b1677
      - read_filenames() at mallocbad.c:119
      - main() at mallocbad.c:58 0x8048e1b
  - C:\Users\stdufresne\qnx710\host\win64\x8

main.c | files.c | strlen() at strlen.si:65 0x1002... | mallocbad.c

```
117  */
118
119  filenames[nfilenames] = malloc(strlen(direntp->d_name)+1);
120  strcpy(filenames[nfilenames], direntp->d_name);
121
122  nfilenames++; /* new number of filenames pointed to by the
123 hile (direntp != NULL);
124
125 sedir(dirp);
126
127 ilenames = filenames;
128 urn nfilenames;
129
```

Variables | Breakpoints

Name	Type	Valu
filenames	char **	0x90
tmp	char **	0x90
nfilenames	int	77
dirp	DIR *	0x90
direntp	struct dirent *	0x0

ble | Memory | Tasks | Problems | Executables

d PMD [C/C++ QNX Local Core Dump Debugging] C:\User 4-gdb.exe

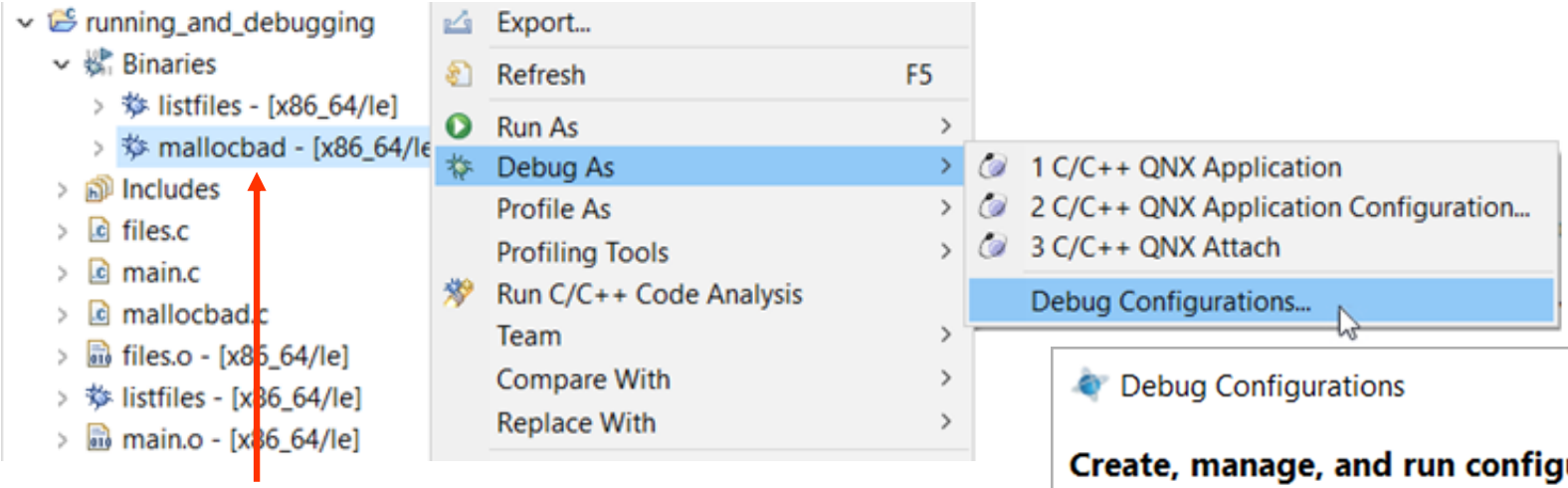
```
[New pid 9093122 tid 1]
```

select this line to see where you were in your code when you crashed

note that this address is 0, but you used it here… memory segment violation (i.e. bad pointer)!

But if the IDE couldn't find everything:

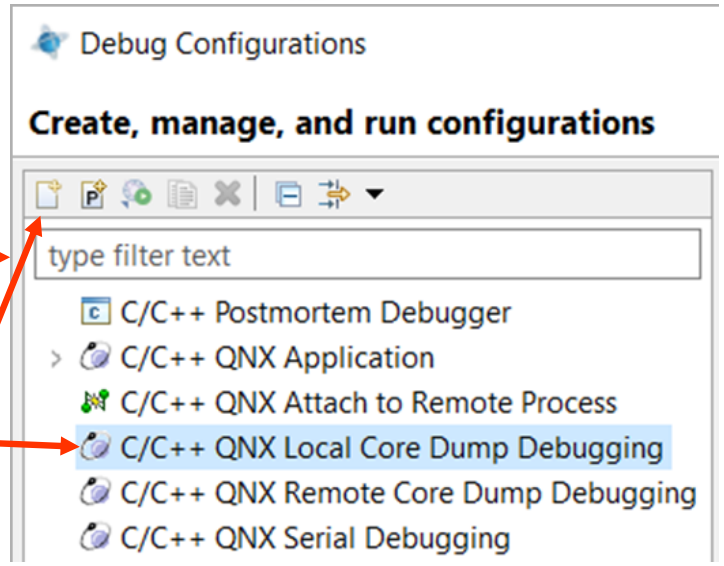
– start with the Debug Configuration wizard



right-click on the executable and bring up the Debug Configurations wizard

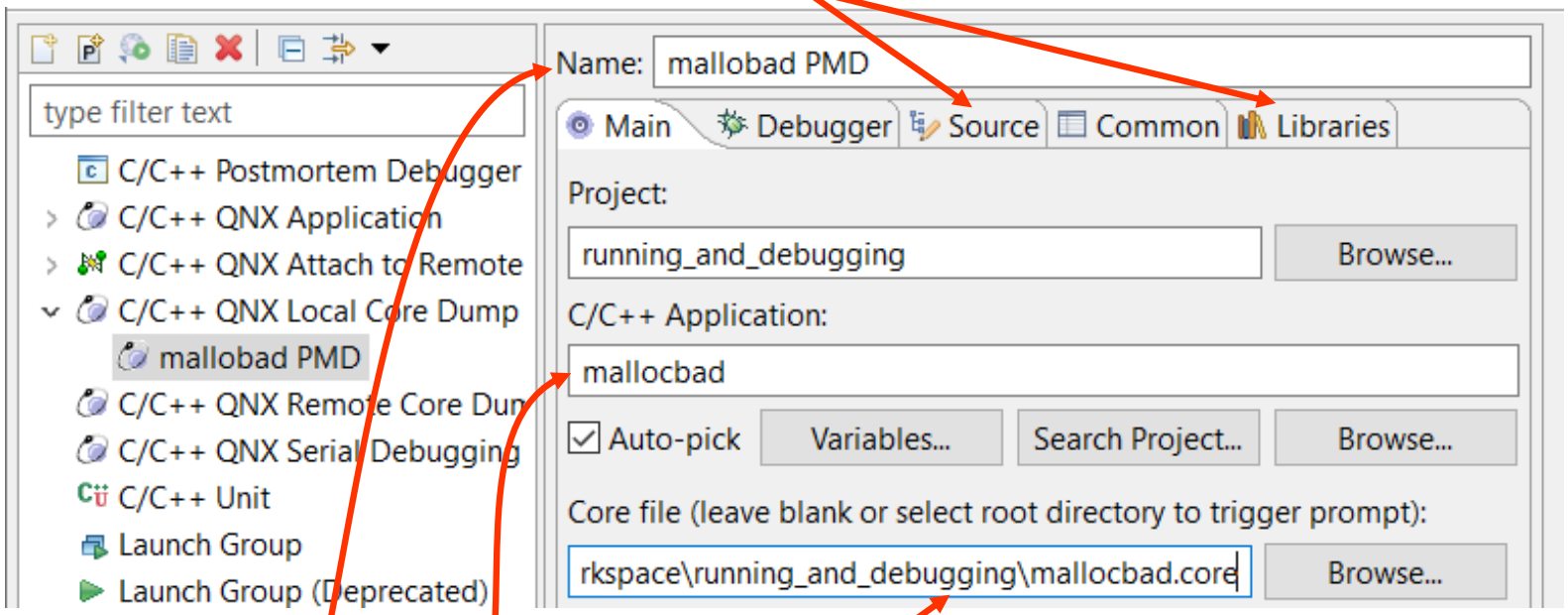
select C/C++ QNX Local Core Dump Debugging

then click the New Configuration button



Point the IDE at things:

– use the Source and Libraries tabs



give a descriptive Name

this tells the IDE which binary to reference, don't put the core file name there!

the core file goes in here

Create and load a core file:

- make sure **`dumper`** is running on your target
  - if not, run **`dumper`**
- copy a program that will crash (e.g. **`mallocbad`**) to your target
- run the program
- copy the core file back
- create a C/C++ QNX Local Core Dump Debugging launch configuration and load the core file

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- → **Attaching to a Running Process**
- **Conclusion**

Attaching to Running Process

You can:

- attach to a process that is already running and then
- debug the process

Useful for debugging a process that:

- needs to be launched in a particular place in start up order
- needs to be started with a particular environment
- may take a while to reach a broken/failed state

To do this:

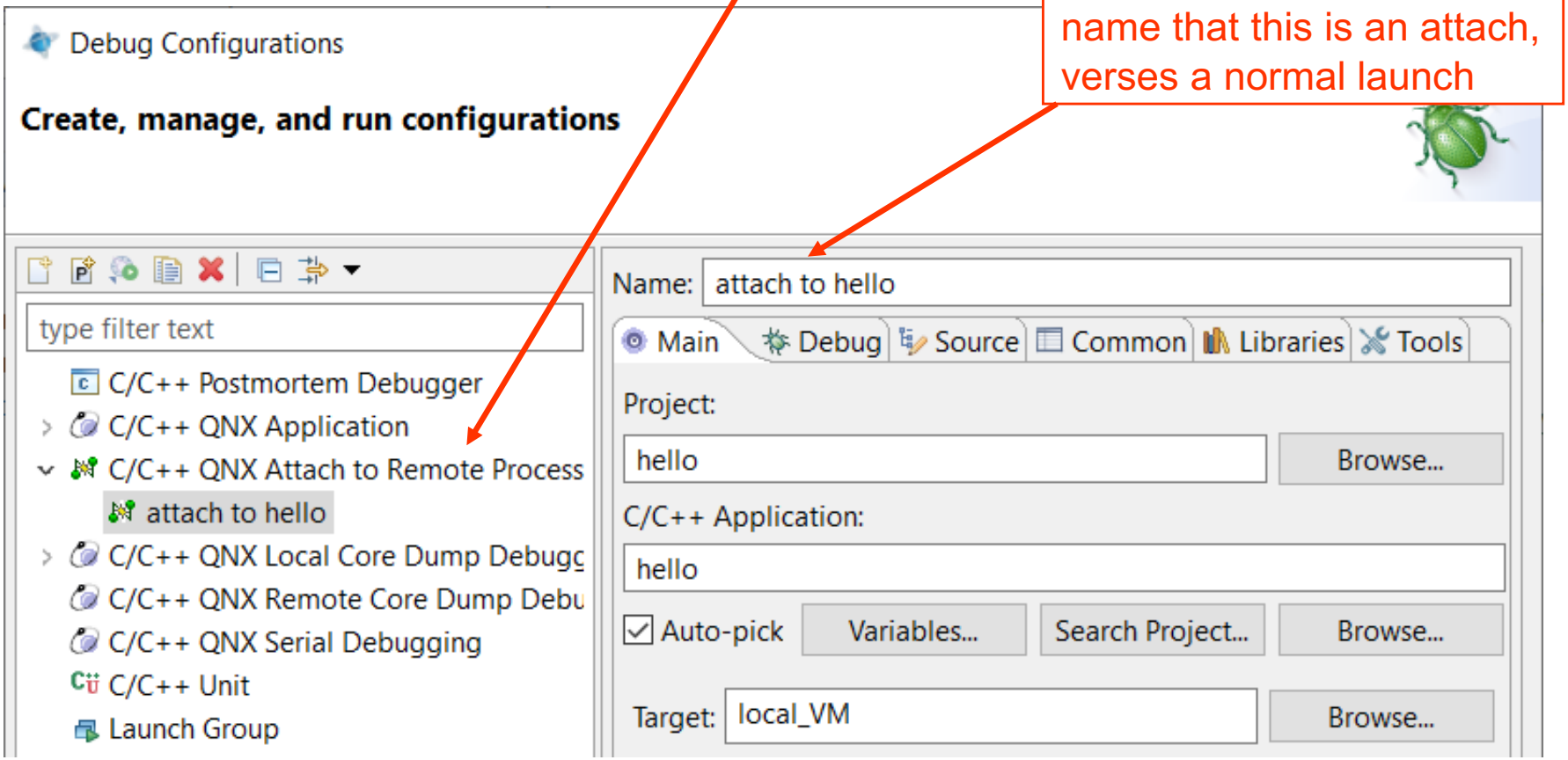
- create a Debug launch configuration...
  - right-click on the executable in the Project Explorer view, Debug As → Debug Configurations

In the Debug Configurations wizard:

– the type must be:

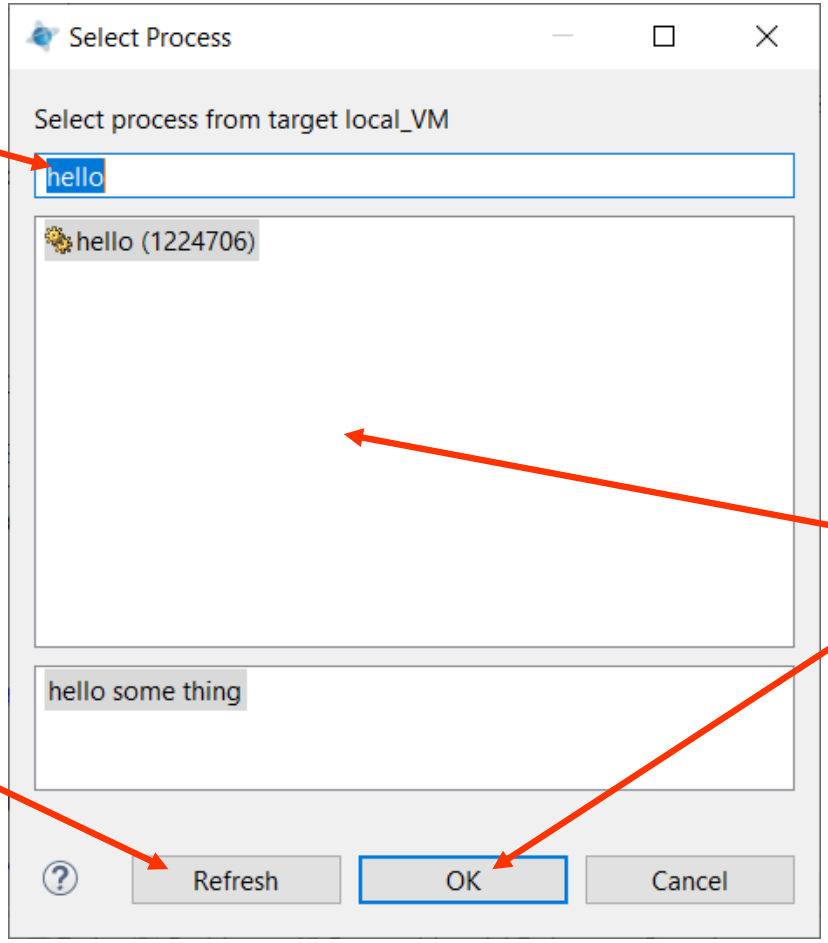
C/C++ QNX Attach to Remote Process:

indicate in the launch name that this is an attach, verses a normal launch



When you click Debug, a list of running processes will appear:

filter to help find your process



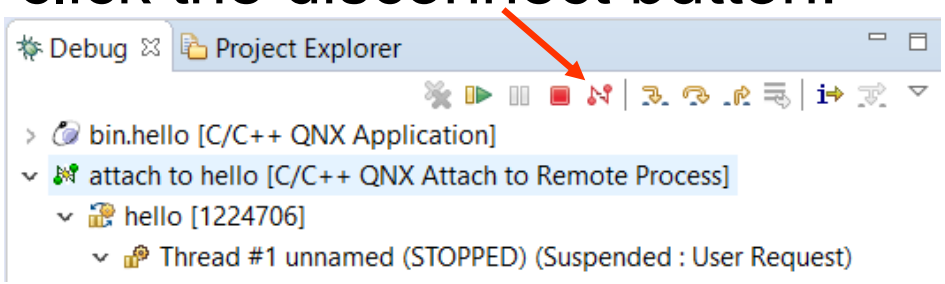
select your process, click OK, and you should now be debugging the process

if your process is not in the list, click on Refresh.

This list may simply be out of date.

You may not see source:

- the debugger will stop the process at whatever code it was executing, it could be anywhere in its code
  - you can stop the program in advance with *raise(SIGSTOP);*
- you can:
  - look at registers and memory using the Registers and Memory views
  - select code using the call stack
  - open source files, set breakpoints and resume execution. It will then stop at a breakpoint.
- do whatever you would normally do when debugging
- to release the program click the disconnect button:



Attaching to a process:

- run a program:
  - that you have in some project
  - that runs continuously
- don’t use a debug launch
  - either download and run it from the command line
  - or launch an executable (run launch)
- create a launch configuration to attach to it
- try attaching to the process, and using the debugger
- release it without killing it when done

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Debugging Library Code**
- **Postmortem Debugging**
- **Attaching to a Running Process**
- → **Conclusion**

Conclusion

You learned:

- how to run or debug a program by using a launch configuration
- the set up needed for debugging
- how to debug running and crashed programs
- basic debugging techniques:
  - stepping through your code
  - setting breakpoints
  - examining and changing the values of variables and memory
  - debugging library code