

C++ Language Fundamentals

Course Version 24.03

Lab Manual

Revision 1.0

© 1990-2024 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

- The publication may be used solely for personal, informational, and noncommercial purposes;

- The publication may not be modified in any way;

- Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and

- Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence customers in accordance with, a written agreement between Cadence and the customer.

Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Table of Contents

C++ Language Fundamentals

Module 1:	About This Course	7
Lab 1-1	Executing C++ Code Using g++ Compiler/GDB Debugger and Microsoft Visual Studio	9
	Part 1: g++ Compiler	9
	Installation.....	9
	Running the Code Using g++.....	10
	GDB Debugger	10
	Debugging the Code Using GDB.....	11
	List of Commands.....	15
	Part 2: Visual Studio	16
	Exploring the Debugger	18
	Part 3: Makefile.....	21
Module 2:	C Language Review	23
	There are no labs for this module.....	25
Module 3:	Object-Oriented Programming and C++	27
Lab 3-1	Organizing an OOP Project	29
Module 4:	C++ Basics	31
Lab 4-1	Writing a Simple C++ Program.....	33
	Part 1: Write Code.....	33
	Part 2: Execute the Steps.....	34
Module 5:	Constructors and Destructors	37
Lab 5-1	Defining Constructors and a Destructor	39
	Part 1: Review Constructors.....	39
	Part 2: Define a Counter Class	40
Module 6:	References	41
Lab 6-1	Experimenting with Reference Variables	43
	Part 1: Review References	43
	Part 2: Add a Reference Variable.....	43
Module 7:	Functions	45
Lab 7-1	Defining Class Member Functions.....	47
	Part 1: Review Class Member Functions	47
	Part 2: Define Overloaded Methods and a Static Variable.....	47
Module 8:	Type Conversion	49
Lab 8-1	Exploring Cast Operations.....	51
	Part 1: Review Cast Operations	51
	Part 2: Explore the reinterpret_cast Operator.....	51
Module 9:	Operator Overloading	53
Lab 9-1	Overloading Member Operators	55
	Part 1: Review Operator Overloading	55

	Part 2: Define an Assignment Operator	55
Module 10:	Inheritance.....	57
Lab 10-1	Deriving Subclasses.....	59
	Part 1: Review Class Inheritance.....	59
	Part 2: Define a Derived Class	59
Module 11:	Polymorphism.....	61
Lab 11-1	Defining and Using Polymorphic Classes.....	63
	Part 1: Review Polymorphism.....	63
	Part 2: Make the Counter Class Polymorphic	63
Module 12:	Constant Objects and Constant Functions	67
Lab 12-1	Experimenting with Constant and Mutable Objects.....	69
	Part 1: Review const Objects and const Functions.....	69
	Part 2: Cast “away” the “constness” of a const Reference	69
Module 13:	Templates	73
Lab 13-1	Defining a Class Template.....	75
	Part 1: Review Function and Class Templates	75
	Part 2: Define and Test a Class Template	75
Module 14:	Exceptions.....	77
Lab 14-1	Throwing and Catching Exceptions	79
	Part 1: Review Exceptions	79
	Part 2: Define and Test an Exception Class	79
Module 15:	Input and Output.....	81
Lab 15-1	Inputting and Outputting Program Data.....	83
	Part 1: Review Input and Output Operations	83
	Part 2: Define an Output Operator	83
Module 16:	Debugging	85
Lab 16-1	Debugging a Program	87
	Part 1: Review GDB and DDD	87
	Part 2: Watch Nodes Being Checked Out and In	87
Module 17:	Containers and Algorithms	91
Lab 17-1	Instantiating and Using Standard Containers	93
	Part 1: Review Containers and Algorithms.....	93
	Part 2: Utilize a Standard Container	93
Module 18:	Introduction to System-C.....	97
Lab 18-1	Implementing FIR Filter in C++ and Running Stratus Tool to Generate the RTL	99
	Part 1: Review System-C	99
	Part 2: FIR Filter Design	99
	Handshake Mechanism	100
	Part 3: Stratus Project Setup.....	101
	project.tcl Structure	102
	project.tcl File	103

project.tcl File	104
Makefile-project.tcl Relationship.....	105
Mechanisms for Controlling Synthesis	105
Part 4: Implementation.....	106
Testbench	108
System_top.....	109
Main File.....	109
Part 5: Execution Steps	111
Invoking the Tool.....	112
Opening a Project.....	113
Stratus IDE Edit Mode.....	113
Run a Behavioral Simulation	114
Run Stratus HLS from the IDE	114
Stratus IDE Analysis Mode.....	115
Analysis Window Dashboard.....	115
To Run Jobs from a Linux Shell Window.....	116
Part 6: Instructions Summary.....	116
Examining a Generated RTL Verilog	117
Module 19: Equivalence Checking C++ for Verification	119
Lab 19-1 Equivalence Checking of a Pipelined Multiplier	121
Part 1: Fixed Delay Standard Miter Model Setup	121
Part 2: Arbitrary Slice Miter Model Setup	122
Part 4: C Versus C Equivalence Checking Setup.....	122
Setup 5: RTL Versus RTL Equivalence Checking Setup for Datapath Designs	123

Module 1: About This Course

Lab 1-1 Executing C++ Code Using g++ Compiler/GDB Debugger and Microsoft Visual Studio

Objective: To execute the C++ code using g++/GDB and Microsoft Visual Studio.

In this lab, we learn how to execute the C++ code using a g++ compiler/GDB debugger and Microsoft Visual Studio. The same instructions apply to all labs.

Part 1: g++ Compiler

- ◆ g++ is a C++ compiler developed by the Free Software Foundation.
- ◆ It is a part of the GNU compiler collection.
- ◆ It converts the high-level C++ code into a lower-level language that is understood by the computer.

Installation

- ◆ Generally, g++ and GCC compilers are pre-installed on Linux systems.
- ◆ You can either install it as mentioned in the below steps or infer it from the Xcelium™ tool installation path as mentioned in the *class_setup* file.
- ◆ Check if installed or not by running the following command:
 - Which g++, for C++ compiler
 - Which GCC, for C compiler
 - Alternatively:
 - For C++ compiler : `g++`
 - For C-compiler : `gcc`
 - If installed, it gives out a fatal error: “files not found”.
 - If not installed, it shows a “command not found” message.
- ◆ To know which version is installed, execute the following command:
 - For C++ compiler : `g++ --version`
 - For C-compiler : `gcc --version`

```
[shilpav@noivl-shilpav M03CppBasics]$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-36)
```

- ◆ If not installed, do so by executing the following command, and to install from the repository, run:

About This Course

```
sudo apt update
sudo apt install gcc g++
```

- ◆ Alternatively, you can install it as a part of the development package:

```
sudo apt install build-essential gdb g++
```

It pulls up the package from the web and installs it.

- ◆ If not in the repository, download the g++ /GCC compiler from the internet and go to downloads and install.

- To check if it is installed, run the command:

```
g++ --version
```

- ◆ Once installed, you can look at the list of available options using:

```
g++ -help
```

Running the Code Using g++

1. Open a file in any text editor and save the file with a .cc or .cpp extension.
2. If the file already exists in any location, go to that location.
3. Compile the file by executing the command:

```
g++ file_name.cpp -o filename
```

this command creates a binary file named 'file_name'.

- -o option tells g++ to place its executable in the file 'file_name'.

4. Execute the 'file_name' executable by using the command:

```
./file_name
```

If -o option is not specified, by default, it puts the executable in a.out file, and you run the code by executing the command:

```
./a.out
```

GDB Debugger

- ◆ GDB stands for GNU debugger.
- ◆ It allows:
 - To set breakpoints in the code so that we can look at its state.
 - Step through the code line by line.
 - Watch/change a variable.

- Observe stack.

Debugging the Code Using GDB

1. Create a file using any text editor and save it with a .cc or .cpp extension.

Let us execute and debug the program, which has the following code. We discuss the constructs in this code in Module 5. Here, we just learn how to execute and debug any given code. This code is saved in the *test.cc* file under the *LAB01_About_This_Course* folder. You can experiment execution with this code.

```
13 #include <iostream>
14 using namespace std;
15
16 int var1 = 1, var2 = 3 ; // global namespace
17 int fnc1(){return 1;}; int fnc2(){return 3;};
18
19 namespace NS {
20
21     int var1 = 5, var3 = 7; // named namespace
22     int fnc1(); int fnc3(){return 7;};
23
24     int fnc1() {
25         int var1 = 9;          // block (function local)
26         cout << var1 << endl; // local -- prints out "9"
27         cout << NS::var1 << endl; // named -- prints out "5"
28         cout << ::var1 << endl; // global -- prints out "1"
29         cout << ::fnc1() << endl;
30         cout << var2 << endl; // global -- but not ambiguous
31         cout << fnc2() << endl;
32         cout << var3 << endl; // named -- but not ambiguous
33         cout << fnc3() << endl;
34         return 5;
35     }
36
37 }
38
39 int main() {
40     NS::fnc1();
41     return 0;
42 }
```

2. Compile the program with the -g debugging option and execute the code using the command:

```
g++ -g file_name.cpp -o file_name
```

- The `-g` option puts the debugging information into the executable file. It preserves the variable and function names intact.

3. Start the debugger by running the command **`gdb file_name`**.

- This invokes the GDB debugger. Launch it in the same path where the source code is. This opens up the GDB command prompt:

```
Reading symbols from /servers/esd_noida/shilpav/CppIntro_21_03/examples/M03CppBasics/test...done.  
(gdb) █
```

4. Once GDB is invoked, we can run the code by using the ‘run’ command. We can also provide any arguments if required. For example, run “hello”. This runs the code at once until exit, not allowing us to debug.

5. To debug, set a breakpoint at any line by executing the command below. This tells the debugger to stop before executing that line.

- `b line_number`. For example, `b 16`:

```
(gdb) b 16  
Breakpoint 1 at 0x40090a: file test.cc, line 16.  
(gdb) █
```

- Other formats:

- `break <line_number>`
- `break <file_name.cpp>:<line_number>`

- Similarly, to break with certain functions:

- `b function_name`. For example, `b func2`:

```
(gdb) b fnc2  
Breakpoint 1 at 0x40079c: file test.cc, line 17.
```

- Other formats:

- `break <function_name>`
- `break <file_name.cpp>:<function_name>`
- `break main`; to break at the beginning of the program

```
(gdb) b main  
Breakpoint 1 at 0x4008ba: file test.cc, line 40.
```

6. Run the code by executing the ‘run’ command. This stops at the breakpoint.

```
(gdb) run
Starting program: /servers/esd_noida/shilpav/CppIntro_21_03/examples/M03CppBasics/test

Breakpoint 1, main () at test.cc:40
40      NS::fnc1();
```

7. Execute 'layout next' (multiple times until you get the right layout) command if you want to see the c-code and the assembly code.

```
test.cc
40      NS::fnc1();
41      return 0;
42  }
43
44
45
46
47

B+> 0x4008ba <main()+4>      callq 0x4007ae <NS::fnc1()>
0x4008bf <main()+9>      mov     $0x0,%eax
0x4008c4 <main()+14>     pop     %rbp
0x4008c5 <main()+15>     retq
0x4008c6 <__static_initialization_and_destruction_0(int, int)> push    %rbp
0x4008c7 <__static_initialization_and_destruction_0(int, int)+1> mov     %rsp,%rbp
0x4008ca <__static_initialization_and_destruction_0(int, int)+4> sub     $0x10,%rsp
0x4008ce <__static_initialization_and_destruction_0(int, int)+8> mov     %edi,-0x4(%rbp)

child process 25837 In: main
(gdb)                                     Line: 40  PC: 0x4008ba
```

- a. Optionally, execute the 'list' command to see the code above and below instead of seeing it from the current line of execution.

You can do one of the following:

- n or next: Executes next line as a single instruction and does not step into function.

```
40      NS::fnc1();
41      return 0;
```

```
40      NS::fnc1();
41      return 0;
```

- s or step: Does not treat functions as a single line instruction; instead, gets into the function and starts executing line by line.

```
int main() {
40      NS::fnc1();
41      return 0;
42 }
```

```
int fnc1() {  
25     int var1 = 9;           // block (function local)  
26     cout << var1 << endl; // local -- prints out "9"
```

- b. Executing 's' has taken us inside the function instead of the next instruction in the immediate scope.

- C or continue : debugger will continue executing until the next breakpoint.

We can also print the variable value by executing the command 'print variable_name'.

```
$1 = 9  
(gdb)
```

We can also set the value of the variable using the this command.

```
25     int var1 = 9;           // block (function local)  
(gdb) s  
26     cout << var1 << endl; // local -- prints out "9"  
(gdb) print var1  
$1 = 9  
(gdb) set var1 = 24  
(gdb) s  
24
```

List of Commands

Command	Description
b main	Puts a breakpoint at the beginning of the program.
b	Puts a breakpoint at the current line.
b line_number	Puts a breakpoint at line line_number.
b +num_of_lines	Puts a breakpoint num_of_lines lines down from the current line.
b function_name	Puts a breakpoint at the beginning of function “function”.
d num	Deletes breakpoint number num.
layout next	To see the c and assembly source code.
info break	Lists all the breakpoints.
enable/disable bk_num	To enable or disable the breakpoint number.
clear bp	To clear the breakpoint.
r	Runs the program until a breakpoint or error.
c	Continues running the program until the next breakpoint or error.
f	Runs until the current function is finished; shows the line where we are at (frame).
s	Runs the next line of the program.
N	Runs the next N lines of the program.
print *arr@len	To print the array.
n	Like s, but it does not step into functions (does not go inside function and execute, instead prints out if any in functions whereas s goes inside function).
u N	Runs until you get N lines in front of the current line.
p var	Prints the current value of the variable "var".
whatis <variable>	To know the datatype of the variable.
watch <variable>	Watches a variable for changes.
bt	Prints a stack trace i.e., shows previous lines of code along with the current line that we are executing i.e., shows how we reached the current line (stack created when function called).
info local	Prints all variables local to the function.
u	Goes up a level in the stack.
d	Goes down a level in the stack.
q	Quits GDB.
info registers	To know the state of registers.
call variable.assign(“new_string”)	To set a string variable in GDB.

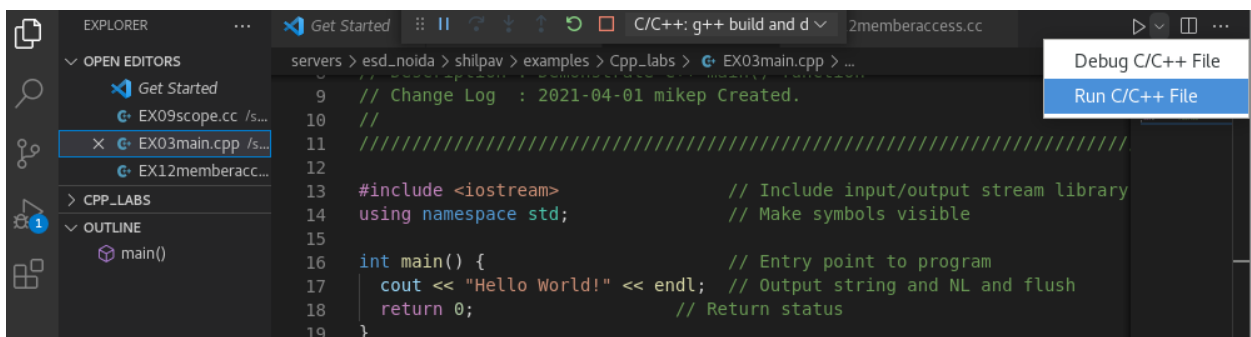
l command	Use the GDB command l or list to print the source code in the debug mode. Use l line-number to view a specific line number (or) l function to view a specific function.
bt: backtrace	Print a backtrace of all stack frames or innermost COUNT frames.
help	View help for a particular gdb topic — help TOPICNAME.
quit	Exit from the GDB debugger.

Part 2: Visual Studio

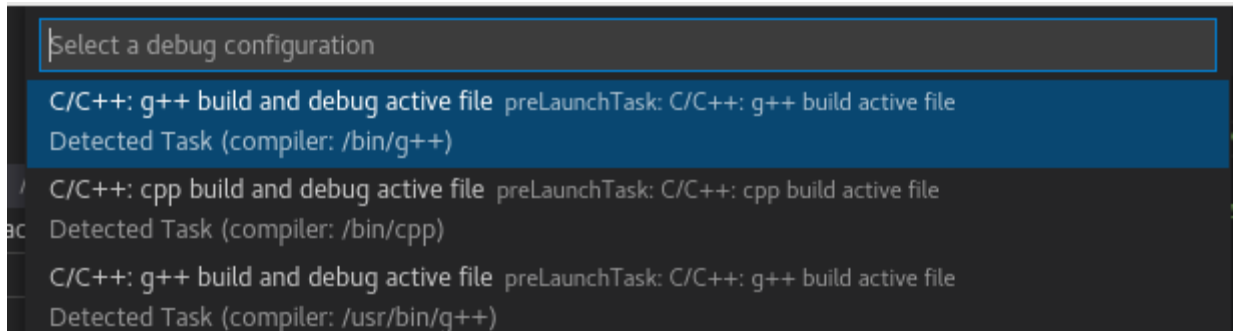
- ◆ It is a code editor developed by Microsoft.
- ◆ Visual Studio is used to edit the source code. However, we will use g++ to compile the C++ code and GDB to debug.

Important: Make sure g++ and GDB are already installed.

1. Install the Visual Studio code. Install the C++ extension by searching for it in the extensions view (ctrl+ Shift+x).
 - Within Cadence, we invoke Visual Studio in Linux by executing the command:
`/grid/common/pkgsvscode/latest/bin/code`
2. Create a folder named **Cpp_labs** to store the code projects.
 - a. Under the folder Cpp_labs, create a new file by clicking on **File – New file**. Type your code in the file and save it with a .cc or .cpp extension. Alternatively, you can also open the file stored at some other path and save it under the Cpp_labs directory.
3. Open the file that you want to execute.
4. Run the code by clicking on the **Play** icon in the top-left corner and select the **Run C/C++ File** option.



5. Select the **g++ build** and debug active from the list of compilers from your system.



Once the build succeeds, you can see your output on your integrated terminal.

```
&"warning: GDB: Failed to set controlling terminal: Operation not permitted\n"
Hello World!
[shilpav@noivl-shilpav cpp_labs]$
```

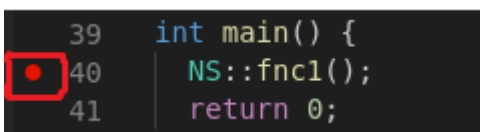
Exploring the Debugger

Let us explore some debug options.

1. Execute the program **test.cc**, which has the following code. We discuss the constructs used in the code in upcoming modules. Here, we only focus on how to execute any given C++ code using Visual Studio.

```
13  #include <iostream>
14  using namespace std;
15
16  int var1 = 1, var2 = 3 ; // global namespace
17  int fnc1(){return 1;}; int fnc2(){return 3;};
18
19  namespace NS {
20
21      int var1 = 5, var3 = 7; // named namespace
22      int fnc1(); int fnc3(){return 7;};
23
24      int fnc1() {
25          int var1 = 9;          // block (function local)
26          cout << var1 << endl; // local -- prints out "9"
27          cout << NS::var1 << endl; // named -- prints out "5"
28          cout << ::var1 << endl; // global -- prints out "1"
29          cout << ::fnc1() << endl;
30          cout << var2 << endl; // global -- but not ambiguous
31          cout << fnc2() << endl;
32          cout << var3 << endl; // named -- but not ambiguous
33          cout << fnc3() << endl;
34          return 5;
35      }
36
37  }
38
39  int main() {
40      NS::fnc1();
41      return 0;
42  }
```

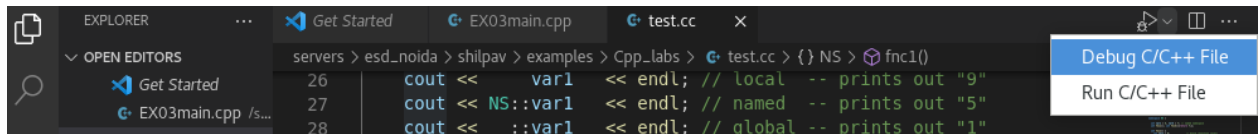
2. Set the breakpoint by clicking in the left margin or by pressing **F9** on the current line.



```
39  int main() {
40      NS::fnc1();
41      return 0;
```

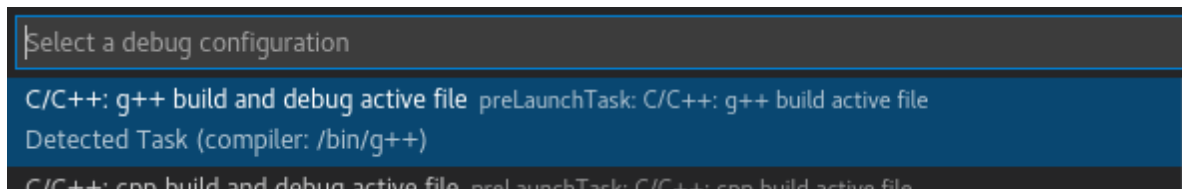
A red circle with a dot inside is positioned in the left margin next to line 40, indicating a breakpoint has been set.

- Click on the **Play** icon in the top-left corner and select **Debug C/C++ File** from the drop-down menu.

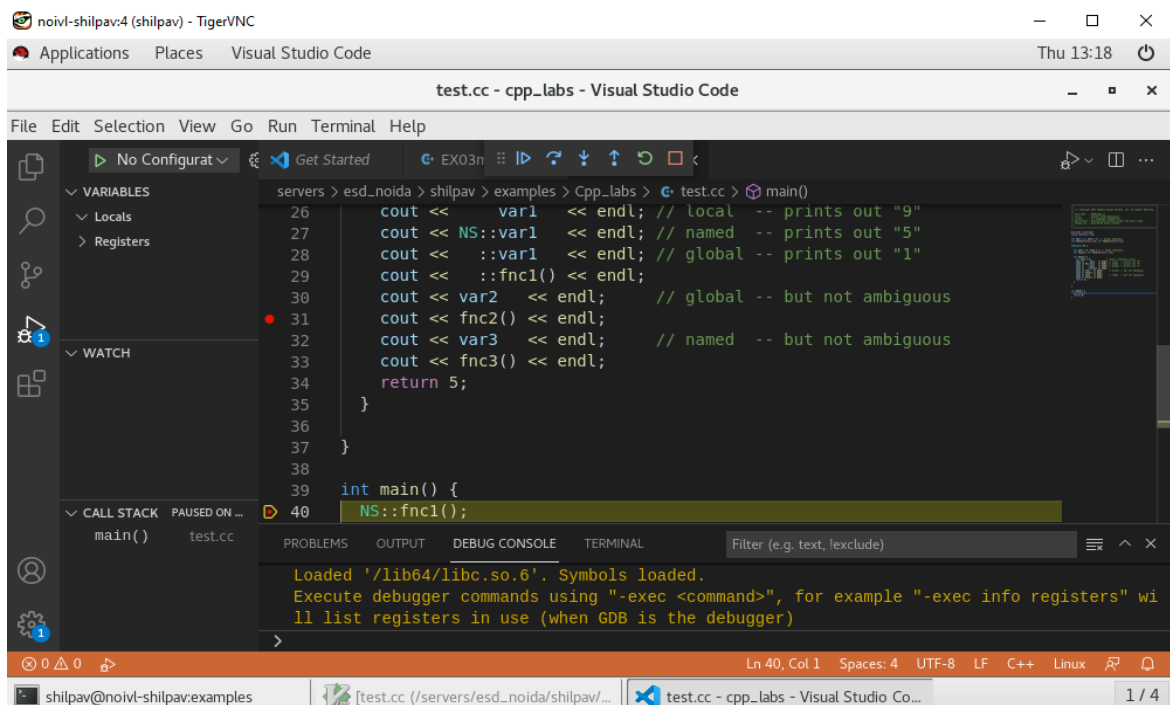


- The Play icon has two options and defaults to the last one chosen. If you see the Bug icon along with the Play icon, then you can click on the Play icon instead of selecting it from the drop-down menu.

- Select the **g++ build** and debug the active file from the list of compiler options.



You can see that the debugger is up and running in the Debug Console tab.



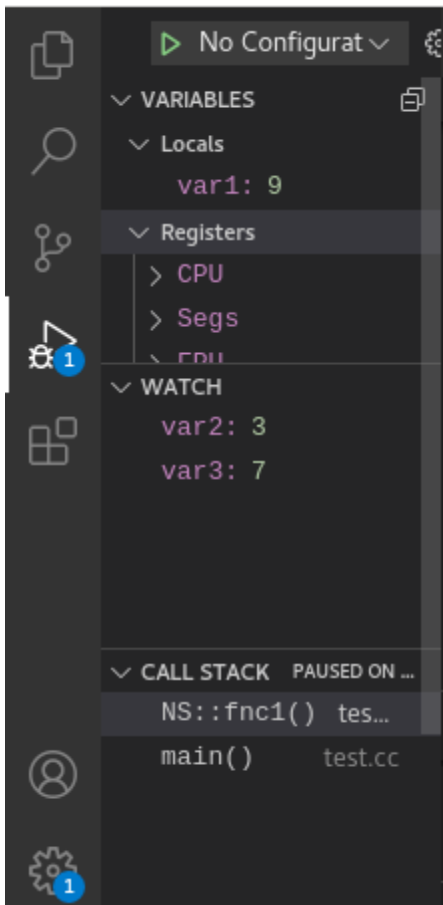
You can also see the breakpoint at line 31 in the source code editor.

- At the top of the code editor, the following icons are from left to right:
 - play icon: To run the code as usual.
 - step-over(F11): Goes to the next statement, skipping over the function calls.
 - step-into(F10): Similar to step-over but steps into the function calls and executes the code within the function line by line.

- step-out: Steps out of the function that we are executing.

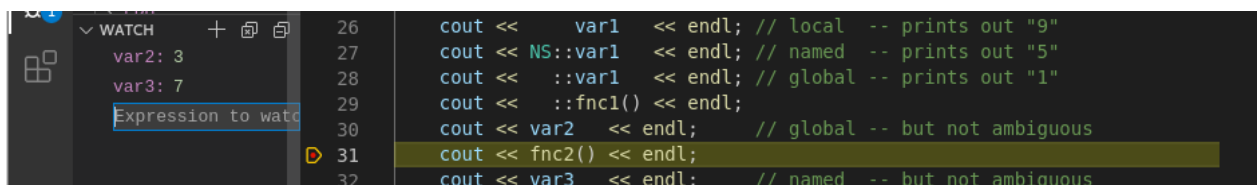


- The run and debug information is shown on the left side.



You can see the local variable values when the execution is paused at the breakpoint. Hover the mouse over the variable in the source code. You can see all the variables in the current scope listed under variable->locals. You can also view the register content if you wish.

6. Add a variable to the watch window by placing the insertion point inside the function. Click on the plus (+) icon in the WATCH window, and in the textbox, type the variable name.



Note: Using all these options, we can debug any given C++ code.

Part 3: Makefile

Makefile is set up for all the following labs. Users just have to run the **make** commands and check the output. If you want to execute any lab step-by-step, you can do so by following the GCC/GDB/Microsoft Visual Studio instructions previously discussed.

1. Execute `'make test'`

This compiles the C++ code using g++ to generate a binary executable file.

2. Execute `'make test.log'`

This executes the binary executable and saves the output onto a log file.

3. Execute `'make diff'`

Compares the actual output saved in the log file in Step 2 with the expected output stored in the saved log file.

4. Execute `'make clean'`

Removes the executable and the actual log file.

Note: For Lab 4-1, there is a different setup, as this lab is interactive and accepts input from the user. Instructions for this lab are discussed separately from the rest of the labs. Module 2 has no labs and Lab 3-1 has nothing to execute.



Module 2: C Language Review

There are no labs for this module

Module 3: Object-Oriented Programming and C++

Lab 3-1 Organizing an OOP Project

Objective: To determine a set of classes to support transaction recording.

For this lab, you generate a document describing your class-based organization of transaction recording components.

Components supporting transaction recording must implement the following roles:

- ◆ Maintain a transaction database
- ◆ Maintain a transaction stream
- ◆ Generate transactions
- ◆ Represent transactions
- ◆ Represent the data item being transacted

For each class, describe the member properties and the public class interface.

The solution for this lab is provided in the lab03_solutions.txt file under the folder *CppLabs/LAB03-OOP-C++/Solutions*.



Module 4: C++ Basics

Lab 4-1 Writing a Simple C++ Program

Objective: To write and execute a simple C++ program with structured input/output.

In the first part of this lab, the program interacts with the user to obtain their first name and birth date and writes that information to a file. In the second part, the program reads the information from the file, calculates the user's age, and displays a message to the user informing them of their age.

Part 1: Write Code

1. Write, debug, and execute a C++ program to interact with the user to obtain their first name and birth date, and write that information to a file named 'file'.

In the file lab04_main_test.cc:

- a. Define and declare a C++ struct to contain the user's first name and birth date.
- b. Interact with the user to obtain their first name and birth date and store them in the structure.

Useful <iostream> methods include:

```
basic_ostream<charT,traits>& operator<<(argument);
```

Inserts a value representation into an output stream.

```
basic_istream<charT,traits>& operator>>(argument);
```

Extracts a value representation from an input stream.

- c. Write data from the structure to a file.

Useful <fstream> methods include:

```
void open(const char* s, ios_base::openmode mode =
        ios_base::in|ios_base::out);
```

```
bool is_open();
```

```
basic_ostream<charT,traits>& write(const char_type* s, streamsize n);
```

Note: Use the sizeof() operator to obtain the structure size.

```
void close();
```

2. Modify, debug, and execute the C++ program to read data from the file to the structure, calculate the user's age, and inform the user.
 - a. Read data from the file to the structure.

Useful <fstream> functions include:

```
basic_istream<charT,traits>& read(const char_type* s, streamsize n);
```

- b. Define a structure method to calculate the user's age.
 - c. Display a message informing the user of their age.
3. [OPTIONAL] Obtain the current date from the operating system and use it to better calculate the user's age.

Useful <ctime> functions include:

```
time_t time ( time_t *timer );
```

Returns the current calendar time (and also places it in the timer if the pointer is not 0).

```
struct tm *localtime ( const time_t *timer );
```

Converts the timer calendar time to broken-down local time.

The **tm** structure contains the following members:

```
int tm_sec; // seconds after the minute - [0, 60]
int tm_min; // minutes after the hour - [0, 59]
int tm_hour; // hours since midnight - [0, 23]
int tm_mday; // day of the month - [1, 31]
int tm_mon; // months since January - [0, 11]
int tm_year; // years since 1900
int tm_wday; // days since Sunday - [0, 6]
int tm_yday; // days since January 1 - [0, 365]
int tm_isdst; // Daylight Saving Time flag
```

Part 2: Execute the Steps

1. Go to the lab directory *CppLabs/LAB04/solutions*. Compile the code using the command:

```
make test option="Dpass1"
```

The program prompts you to enter your name and date of birth in “yyyymmdd” format. It calculates the user's age.

For example, Cadence's birthdate.

19940101

2. Next, run the command:

```
make test option="Dpass2"
```

This will output your name and age.

For example, Cadence, you are 29 years old.

Alternatively, you can directly run the following commands without using Makefile.

3. Execute the following command:

a. `g++ lab03_main_test.cc -Dpass1`

b. `./a.out`

This asks for your name and birth date as input.

Enter your first Name: Cadence

Enter your birth date in the form yyymmdd: 19940101

4. Next, execute the command:

a. `g++ lab03_main_test.cc -Dpass1`

b. `./a.out`

This outputs your name and age.

For example, Cadence, you are 29 years old.



Module 5: Constructors and Destructors

Lab 5-1 Defining Constructors and a Destructor

Objective: To define and test a counter class with constructors and a destructor.

For this lab, you define a counter class with default, copy, and initialization constructors, a destructor, and methods to set, get, and increment the count. You write a test to confirm the constructor and method operation.

Part 1: Review Constructors

Review the training materials where they describe the definition and use of class constructors.

Constructors are in three categories:

- ◆ *A default constructor has no parameters.*

This is the constructor the compiler calls when you construct an object and provide no arguments. If you do not define any constructors for your class, then the compiler defines a default constructor.

- ◆ *A copy constructor has one parameter that is a reference to another object of the class.*

This is the constructor you call to initialize a newly constructed object from another object of the class. If you do not define a copy constructor for your class, then the compiler defines a copy constructor. The compiler cannot know about any objects you allocate from the heap, so it performs only a shallow copy. You define your own copy constructor to perform a deep copy. The compiler, by default, will also use the copy constructor if you do not define an assignment operator to assign one object of the class to another.

- ◆ *An initialization constructor has one or more parameters.*

This is the constructor you call to initialize a newly constructed object using one or more values. If the constructor has exactly one parameter that is not a reference to another object of the class, then it is also known as a conversion constructor. The compiler, by default, will use the conversion constructor if you do not define an assignment operator to assign the type of that parameter to an object of the class. You can prevent this by declaring the conversion constructor explicit.

Part 2: Define a Counter Class

1. In a new header file, lab05_counter.h, declare a counter class.

Note: The copy constructor parameter is a **const** reference to prevent the constructor from modifying the copied object. With this **const** reference, you can access only the **const** methods of the referenced object. For example:

```
Counter ( const Counter& other ) { count = other.get_count(); }  
unsigned get_count ( void ) const { return count; }
```

See where the training materials describe the **const** qualifier.

- a. Declare default, copy, and initialization constructors.
 - b. Declare the destructor.
 - c. Declare methods to set, get, and increment the count.
2. In a new body file, lab05_counter.cc completes the counter class definitions.
 3. In the file lab05_counter_test.cc, write a test to confirm all counter features.
 4. Compile counter and test; execute; debug as needed.
 5. Compare the output with the output stored in the file lab05_counter_test_save.log.
 6. You can do so by executing the make test, make test.log. make diff and make clean commands are discussed in Lab1-1.



Module 6: References

Lab 6-1 Experimenting with Reference Variables

Objective: To declare, initialize, and test a reference variable.

For this lab, you add a reference variable to your counter class, initialize the variable, and use it in at least one class method.

Part 1: Review References

Review the training materials where they describe the use of reference variables. A reference variable is conceptually an additional symbol representing the same storage area. When you initialize a reference variable, you are linking a variable address instead of assigning a variable value. You initialize a reference variable during construction. If it is a member variable, then the initialization must occur in the member initialization list of every constructor. Furthermore, the compiler cannot then use those copy or conversion constructors to implement assignment operations.

Part 2: Add a Reference Variable

1. Copy the header file, design file, and test file from the previous lab into a new directory. Name them `lab06_counter.h`, `lab06_counter.cc` and `lab06_counter_test.cc` respectively.
2. For your counter class, declare a reference variable named “value” of the same type as the count and initialize the reference variable in the member initialization line of every constructor.
3. Modify your test to comment out any assignments to counter objects and execute the test.
4. The solution for this lab is provided in the *CppLabs/LAB06-References/Solutions* folder. You can verify the same.
5. You can run- `make test`, `make test.log`, `make diff` commands to execute, and compare the actual results with expected ones as mentioned in Lab 1-1.



Module 7: Functions

Lab 7-1 Defining Class Member Functions

Objective: To pass arguments by reference, overload functions, and use static data members.

For this lab, you overload class methods to set variable values, and you use a static data member to track the number of class objects and a static member method to get that number.

Part 1: Review Class Member Functions

Review the training materials where they describe pass-by-reference, function overloading, and static data members.

- ◆ Pass-by-reference is equivalent in performance to passing pointers but greatly reduces the probability of error when less-experienced programmers use pointers. You have already seen that the copy constructor has a const reference parameter. You will later see several operators that take or return references.
- ◆ You can overload functions for programming convenience. The functions have the same name and return type and differ only in the names and types of their parameters. Thus, you can have a family of related functions that behave slightly differently depending upon their argument set.
- ◆ Static members are members that exist separately from any class objects. Any object can access a static member, and application code can access a static member even when no objects exist. Static member methods have no knowledge of the object, so they can access only static members.

Part 2: Define Overloaded Methods and a Static Variable

1. Copy the header, design and test file from the previous lab into a new folder. Name them lab07_counter.h, lab07_counter.cc and lab07_counter_test.cc.
2. Modify your counter class to define overloaded methods.
 - a. Define an enumerated type with at least two values to represent the counter mode, such as binary and BCD.
 - b. Declare a variable of the mode type.
 - c. Modify all constructors to initialize the mode value to a default value.
 - d. Declare and define a set of overloaded methods that, depending upon the number and type of their arguments, set only the counter value, only the mode value, or both the counter value and the mode value.

Functions

- e. Add a method to return the mode value.
 - f. Modify your count method so that it counts differently depending on the mode value.
3. Modify your test to test your counter modifications; compile counter and test; and execute the test.
4. Modify your counter class to maintain an object count in a static variable.
 - a. Declare a static class variable to maintain the number of objects.
 - b. Define and initialize the static class variable outside the class declaration.
 - c. Declare and define a static method to return the value of the static class variable.
 - d. Modify all constructors to increment the variable.
 - e. Modify the destructor to decrement the variable.
5. Modify your test to test your counter modifications; compile counter and test; and execute the test.
6. The solution for this lab is in the *CppLabs/LAB07-Functions/Solutions* folder. You can execute and verify the lab using make commands as discussed in Lab 1-1.



Module 8: Type Conversion

Lab 8-1 Exploring Cast Operations

Objective: To explore the C++ cast operators.

For this lab, you experiment with the **reinterpret_cast** operator. You cast a pointer type to integer types that are the same size and smaller than the pointer type and see that casting to the smaller integer type potentially loses the pointer value.

Part 1: Review Cast Operations

Review the training materials where they describe implicit and explicit type conversion. The C++ standard introduces four cast operators with which you can specify your cast more precisely.

- ◆ The **static_cast<T>(v)** expression statically casts the expression ‘v’ to the related type ‘T’. Pointers to classes in a given class hierarchy are related, as are integer and enumerated types, and even integer and floating-point types.
- ◆ The **reinterpret_cast<T>(v)** expression statically casts the expression ‘v’ to the unrelated type ‘T’. Pointers to classes in unrelated class hierarchies are unrelated, as are pointer and integer types. The result of this cast is that the memory bit pattern representing the cast object is reinterpreted as some other type, which, if smaller, results in loss of precision. This cast is defined by the implementation and thus not portable.
- ◆ The **const_cast<T>(v)** expression statically casts the expression ‘v’ to the type ‘T’ if both types are pointer or reference types that differ only in their const or volatile qualifiers. You typically use this operator to “cast away” the “constness” of a pointer or reference so that you can modify the non-const object pointed to or referenced.
- ◆ The **dynamic_cast<T>(v)** expression dynamically casts the expression ‘v’ to the type ‘T’, which must be either a void pointer or a pointer or reference to a complete class type. You typically use this operator during run time to “down-cast” a base class pointer to the type of the object the pointer actually “points to”, with which you can then access the incremental members not declared in the base class. To perform a “down” or “cross” cast requires that the types be polymorphic.

Note: This lab explores only the **reinterpret_cast** because the **static_cast** is a relatively simple subset of what the C cast operator that you are already familiar with does, and later labs more appropriately explore the **dynamic_cast** and **const_cast** operators.

Part 2: Explore the reinterpret_cast Operator

1. Copy the header, design and test files from the previous lab and rename them.
2. Modify the counter test to prepare to explore the **reinterpret_cast** operator.

Type Conversion

- a. Declare and initialize a counter pointer to point to one of your counter objects.
 - b. Using the `sizeof()` operator and `cout` output stream, determine and display the size on your platform of a counter pointer and of various integral built-in types.
 - c. Test your test modifications – compile counter and test and execute the test.
3. Modify your counter test to cast the counter pointer to an integral type at least as big as the pointer and then cast the integral type back to a counter pointer.
- a. Using the **`reinterpret_cast`** operator, convert the counter pointer to an integral type at least as big as the pointer.
 - b. Using the **`reinterpret_cast`** operator, convert the integral type back to a counter pointer.
 - c. Use the resulting counter pointer to access the count method.
 - d. Test your test modifications – compile counter and test and execute the test.

Does the test succeed?

Answer: The test should succeed – the bit pattern representing the pointer value did not change.

4. Modify your counter test to cast the counter pointer to an integral type smaller than the pointer and then cast the integral type back to a counter pointer.
- a. Using the **`reinterpret_cast`** operator, convert the counter pointer to an integral type smaller than the pointer.
 - b. Using the **`reinterpret_cast`** operator, convert the integral type back to a counter pointer.
 - c. Use the resulting counter pointer to access the count method.
 - d. Test your test modifications – compile counter and test and execute the test.

Does the test succeed?

Answer: The test will very likely not succeed – the bit pattern representing the pointer value has lost some precision, which probably changed the value.

5. The solution for this lab is in the *CppLabs/LAB08-TypeConversions/Solutions* folder. You can run the lab by using the make commands discussed in Lab 1-1.



Module 9: Operator Overloading

Lab 9-1 Overloading Member Operators

Objective: To provide an intuitive interface to class operations by defining overloaded operators.

For this lab, you overload assignment operators to accept `const` counter references and count arguments so that counter objects can participate in assignments.

Part 1: Review Operator Overloading

Review the training materials where they describe operator overloads. Some operators you cannot overload. For some operators, the overload must be a member method. For some operators, you should define a non-member method.

- ◆ You cannot create new operators.
- ◆ Make overloads act intuitively, e.g., don't overload '+' to do a multiply.
- ◆ You cannot overload the operators: :: . * ? : sizeof typeid.
- ◆ Some operators must be member methods: = [] () ->.
- ◆ For member functions, the first or only operand must be an object of the class.

Part 2: Define an Assignment Operator

1. Copy the header, design and test files, and rename them as `lab09_counter.h`, `lab09_counter.cc`, and `lab09_counter_test.cc`, respectively.
2. Modify the counter class.
 - a. Declare and define an assignment operator to accept a **`const`** counter reference argument and return a reference to the current object.
 - b. Declare and define an assignment operator to accept a count argument and return a reference to the current object.
3. Modify the counter test.
 - a. If you included assignment tests in Lab 5 and commented them out in Lab 6, you can now remove the comments.

Note: The compiler will not use copy and conversion constructors to implement assignments if it can find appropriate assignment operators.
 - b. If you do not yet have assignment tests, then include them now.
4. Test your modifications – compile counter and test and execute the test.

Operator Overloading

5. The solution for this lab is in the *CppLabs/LAB09-Operator-Overload* folder. You can execute this lab using the make commands as discussed in lab1-1.



Module 10: Inheritance

Lab 10-1 Deriving Subclasses

Objective: To define a down-counter class based upon the counter class.

For this lab, you derive a down-counter class from your counter class and determine which methods to re-implement and which methods to remain inherited “as-is”.

Part 1: Review Class Inheritance

Review the training materials where they describe class inheritance. Which members are inherited? Of these, which are useful as they are, and which must be modified?

- ◆ Constructors are not inherited but are, by default, called in a top-down manner, starting with the most base class of the pointer or reference to the constructed object and working toward the class of the object. The effect is as if each constructor in its member initialization list is immediately called the default constructor of its immediate superclass. For anything other than a default constructor, this is probably not the desired behavior, e.g., a copy constructor should call the superclass copy constructor. A usual and good practice is to, in the member initializer list, explicitly invoke the base class constructor to initialize the object’s base class data members and then continue on to initialize the derived class, incremental data members.
- ◆ Destructors are not inherited but are, by default, called in a bottom-up manner, starting with the class of the pointer or reference to the destroyed object and working toward the most base class. The effect is as if the last action of each destructor is to call the destructor of its immediate superclass.
- ◆ Normal methods that are not private are inherited. This may or may not suffice. For example, an inherited assignment operator may expect a const base class reference and return a base class reference. While the compiler is happy to “up-cast” a derived class reference to a base class reference, it cannot “down-cast” the base class reference result to a derived class reference without your help. You should expect to either define new versions of the methods to return derived class references or modify the calling code to utilize base class references.

Part 2: Define a Derived Class

1. Copy the header, design, and testbench from the previous lab into a new directory. Rename them as `lab10_counter.h`, `lab10_counter.cc`, respectively.
2. Define a derived down-counter class based upon your counter class in the file `lab10_countdown.cc`.
 - a. Copy the header file from the previous lab again and rename it **`lab10_countdown.h`**.

Inheritance

- b. In the down-counter header file, `lab10_countdown.h`:
 - Include the counter header file, `lab10_counter.h`.
 - Base the down-counter class on the counter class.
 - Change all counter references to down-counter references.
 - Modify the down-counter constructors to invoke the counter constructors.
 - Remove methods that can be inherited “as-is”.
 - c. Copy the counter body file from the previous lab and name it **`lab10_countdown.cc`**.
 - d. In the down-counter body file, `lab10_countdown.cc`:
 - Make modifications to match the new header file.
 - Modify the count method to decrement the counter instead of increment.
3. Test your down-counter class.
- a. Copy the test file from the previous lab and rename the test file as **`lab10_countdown_test.cc`**.
 - b. Modify the test to change all references from the counter type to references from the down-counter type.
 - c. Compile counter, down-counter, and test; execute; debug as needed.
4. The solution for this lab is provided in the *CppLabs/Lab10-Inheritance/Solutions* folder. You can execute this lab by using make commands as mentioned in Lab 1-1.



Module 11: Polymorphism

Lab 11-1 Defining and Using Polymorphic Classes

Objective: To make the counter class hierarchy polymorphic.

For this lab, you access a down-counter method whose name is statically bound, access a down-counter method whose name is dynamically bound, and “down-cast” a base class pointer or reference to enable using it to access a derived class, incremental member.

Part 1: Review Polymorphism

Review the training materials where they describe polymorphism.

- ◆ You can use a pointer or reference of a base type to point to or reference an object of a derived type. This capability is very useful; for example, it can be used to utilize a container that contains references to multiple different counter types that are all based upon a basic counter. However, the pointer or reference “knows” about only the members of its own class type and does not know about any incremental members of derived classes.
- ◆ You cannot use a pointer or reference of a derived type to point to or reference an object of a base type. If you could do so, it would “promise” more than it could deliver, as the object of the base type would not have the incremental members of the derived type that the pointer or reference expects.
- ◆ Imagine that you have a pointer or reference of the counter type that is currently pointing to or referencing an object of the down-counter type, and using the pointer or reference, you access the count method. Does the counter increment or decrement?

Part 2: Make the Counter Class Polymorphic

1. Copy the header, design and test files from the previous lab. Rename them as lab11_counter.h, lab11_countdown.h, lab11_counter.cc, lab11_countdown.cc and lab11_countdown_test.cc respectively.
2. Modify the down-counter test to access a down-counter method whose name is statically bound.
 - a. Declare and initialize a counter pointer or reference to a down-counter object.
 - b. Use the pointer or reference to access the count method.
 - c. Compile counter, down-counter, and test; execute and debug as needed.

Does the down-counter increment its count or decrement it?

Answer: If the count method is a normal method, then the method as it is known in the class of the pointer or reference executes, and the counter increments. This is known as early name binding – the compiler binds the method call directly.

3. Modify the counter base class to make the count method dynamically bound.
 - a. Preface the count function declaration with the **virtual** function specifier. You need to do this only in the base class – a virtual method remains virtual in all further derivations.

Does the down-counter increment its count or decrement it?

Answer: If the count method is a virtual method, then the method as it is known in the pointed-to or referenced object executes, and the counter decrements. This is known as late name binding – the compiler generates a table of method addresses that direct the call during run time.

4. Modify the counter base class to make the destructor dynamically bound.
 - a. Preface the destructor declaration with the **virtual** function specifier.

The operand you provide to the delete operator is a pointer. If the destructor is non-virtual, then the destructor call is bound early to the destructor of the pointer class, regardless of what object the pointer points to. If the destructor is virtual, then the destructor call is bound late to the destructor of the object class.

5. Down-cast a base class pointer or reference to allow using it to access a derived class incremental member.
 - a. Modify your down-counter class to add a public void method that does nothing.
 - b. Modify your down-counter test to attempt to use your counter pointer or reference that is pointing to or referencing a down-counter object to access the new down-counter do-nothing method.
 - c. Compile counter, down-counter, and test.

Do they successfully compile?

Answer: You should not be successful – the counter pointer or reference does not know about the members of the down-counter class.

- d. Modify your down-counter test to dynamically cast the counter pointer or reference to a down-counter pointer or reference, then use the “down-cast” pointer or reference to access the new down-counter do-nothing method.
 - e. Compile counter, down-counter, test; execute and debug as needed.

Do they successfully compile?

Answer: You should be successful – the down-counter pointer or reference does know about the members of the down-counter class.

Why not just do a static cast?

Answer: A dynamic cast verifies that your cast is correct because it occurs during run time and so knows what type of object is being pointed to or referenced.

6. The solution for this lab is in the *CppLabs/LAB11-Polymorphism/Solutions* directory. You can execute this lab by using the make commands mentioned in Lab 1-1.



Module 12: Constant Objects and Constant Functions

Lab 12-1 Experimenting with Constant and Mutable Objects

Objective: To declare class variables whose values cannot be changed, enable calling member functions of a **const** class object, and remove the “constness” of pointers and references.

For this lab, you cast “away” the “constness” of a **const** reference and use it to access a non-const member method of a non-const object, cast “away” the “constness” of a const reference and use it to access a non-const member method of a **const** object and modify a **mutable** variable of a const object.

Part 1: Review const Objects and const Functions

Review the training materials where they describe **const** functions and **const** objects.

- ◆ You cannot modify a **const** variable.
- ◆ Changing the “constness” of a pointer or reference to a **const** variable does not change the “constness” of the variable.
- ◆ All member variables of a **const** object are by default **const**.
- ◆ You can call only the **const** or **static** member functions of a **const** object.
- ◆ A **const** member function guarantees it does not modify the object’s variables.
- ◆ Exception – A **const** member function can modify a **mutable** variable even if the object is **const**.

Part 2: Cast “away” the “constness” of a const Reference

1. Cast “away” the “constness” of a **const** reference and use it to access a non-const member method of a non-const object:
 - a. Modify your down-counter test.
 - Declare a **const** down-counter reference initialized to one of your down-counters.
 - Using the **const** reference, access the count method.
 - b. Compile counter, down-counter, and test.

Is the compile successful?

Answer: The compile should not be successful. The reference is **const**, and thus, you cannot use it to modify the referenced object.

c. Modify your down-counter test:

- Using the **const_cast** operator, cast the **const** down-counter reference to a non-const down-counter reference.
- Using the non-const reference, access the count method.

d. Compile counter, down-counter, test and execute.

Is the access successful?

Answer: The access should be successful. The reference is not **const**, and thus you can use it to modify the referenced object.

2. Cast “away” the “constness” of a **const** reference and use it to access a non-const member method of a const object:

a. Modify your down-counter test.

- Declare a **const** down-counter initialized to some integer value.
- Declare a **const** down-counter reference initialized to the **const** down-counter.

b. Using the **const_cast** operator, cast the **const** down-counter reference to a non-const down-counter reference.

c. Using the non-const reference, access the count method.

d. Compile counter, down-counter, test and execute.

Is the access successful?

Answer: The standard does not define the result of an attempt to modify a **const** object through a non-const pointer or reference. It may or may not be detected during compile time or during runtime, and it may or may not work.

3. Modify a **mutable** variable of a const object.

a. Modify your counter class.

Note: The **mutable** qualifier does not apply to pointer or reference variables; thus, you cannot reliably apply it to the count variable, as in a previous lab, you declared a reference to the count variable, to which reference you cannot apply the qualifier.

- Preface the declaration of the mode variable with the **mutable** qualifier.
- Add a **const** method to set the mode variable.

b. Modify your down-counter test.

- Call the new **const** method to set the mode variable of the **const** down-counter object.

- c. Compile counter, down-counter, test and execute.

Is the access successful?

Answer: A **const** method can modify a **mutable** variable, even if the object itself is **const**. You declare a variable **mutable** if you have an overwhelmingly good reason to modify that one variable, even if the object is **const**.



Module 13: Templates

Lab 13-1 Defining a Class Template

Objective: To define generic code to be specialized upon each use by defining a class template.

For this lab, you define a pool class template and instantiate and test a pool.

Part 1: Review Function and Class Templates

Review the training materials where they describe function templates and class templates.

- ◆ A function template defines a family of functions that differ only in the type(s) and/or value(s) of their parameter(s).
- ◆ The compiler can deduce template type parameters from the call argument types.
- ◆ You can explicitly specify template arguments as you make the call.
- ◆ A class template defines a family of classes that differ only in the type(s) and/or value(s) of their parameter(s).
- ◆ You can specify default values for the template parameters.
- ◆ All element types and values must be specified upon reference to the class type.
- ◆ The class definition does not need to use the element types or values.

Part 2: Define and Test a Class Template

For many applications, it is convenient to maintain a pool of objects from which the application can “check-out” an object and, when no longer needing the object, check it back into the pool. Pooling objects facilitates their reuse and may be more efficient than dispersing object allocation among the application methods. As the pool behaves identically regardless of the object type, it is usual and good programming practice to generate a class template for creating pool classes.

1. In a header file, define a pool class template.

Note: Most compilers require that the class template definition be available when referencing the class type. Thus, it is usual and good programming practice to place both the class template declarations and the class template definitions within the header file rather than separate header and body files.

- a. Define a **struct** to represent a list node with the three fields:
 - Variable of type *pointer-to-object* to point to an allocated object.
 - Variable of type **bool** indicating whether the object is checked-out.

- Variable of type *pointer-to-node* to point to the next node in the list.
 - b. In a pool template **protected** region:
 - Declare a variable of type *pointer-to-node* to point to the first node in the list.
 - Define the pool copy constructor – we protect the constructor to prevent inadvertently copying the pool, which would be only a shallow copy anyway.
 - c. In a pool template **public** region:
 - Define the pool default constructor, and in its member initialization list, initialize the “first” pointer to the nullptr value.
 - Define the destructor to traverse the list, and for each node, delete that node’s pointed-to object and then delete that node.
 - Define a method to “check-out” an object: traverse the list to find an available object, and if none exists, add to the list a new node pointing to a new object; mark the node as “checked-out”; and return a pointer to the object.
 - Define a method to “check-in” an object: traverse the list to find the node matching the object pointer argument; if not found, then do nothing; if found, then mark the node as “checked-in”.
2. Write a test of the pool.
- a. Allocate a pool. The pooled object template argument is immaterial to this exercise, so you can use a simple type such as an **int**.
 - b. Check out a small number (recommended 3) of objects.

Note: To simplify a later debugging lab, you should keep the test program short.
 - c. Perform a trite operation using the objects (just verify that they truly exist).
 - d. Check the objects back into the pool.
 - e. Delete the pool (to verify that the destructor can execute).
3. Compile the pool and test; execute; debug as needed.



Module 14: Exceptions

Lab 14-1 Throwing and Catching Exceptions

Objective: To gracefully handle “exceptional” conditions.

For this lab, you define an exception class, throw an object of your exception class type upon an attempt to check an object into the pool that was not checked out of the pool, and catch and display the exception object.

Part 1: Review Exceptions

Review the training materials where they describe exceptions.

- ◆ The exception infrastructure provides a standard way for a function encountering a problem to pass control back to a calling function.
- ◆ The C++ standard defines a base exception class and several standard exceptions for use by the implementation.
- ◆ Your exception can be any type, but usual and good programming practice is to derive it from the standard base exception class.
- ◆ You can throw an exception only from within a try block scope.
- ◆ A catch block that catches your exception type must appear lexically after the try block and not later in the stack than the try block.
- ◆ Upon throwing an exception, the stack unwinds to the frame containing the catch block and executes the catch block and lexically subsequent code.

Part 2: Define and Test an Exception Class

1. In a new header file, define an exception class, an object that you will throw upon an attempt to check an object into the pool that does not belong there.
 - a. You can elect whether to derive it from `std::exception`.
 - b. Define a default constructor and remember to specify that it throws nothing.
 - c. Define the destructor and remember to specify that it throws nothing. If you do not inherit `std::exception` then remember to specify the destructor as **virtual**.
 - d. Define the `what()` method to return a static message and define the message.
2. Modify your pool class template to throw an object of your exception type upon an attempt to check an object into the pool that was not checked out of the pool.

Exceptions

3. Modify your test to catch and display the exception object:
 - a. Within a try block, attempt to check an object into the pool that you did not check out of the pool.
 - b. Include a catch block immediately after the try block, and in it, catch the exception object, and through the exception object's `what()` method, retrieve the message and display it.
4. Compile exception, pool, and test; execute; debug as needed.



Module 15: Input and Output

Lab 15-1 Inputting and Outputting Program Data

Objective: To input and output data to and from programs.

For this lab, you define an ostream insertion operator for objects of type pointer-to-pool, define an ostream insertion operator for objects of type pointer-to-node, and define a new void method to display the pool contents by inserting the pool pointer into the cout output stream.

Part 1: Review Input and Output Operations

Review the training materials where they describe input and output operations.

- ◆ Managing I/O state
- ◆ Opening and closing files
- ◆ Formatting input and output

Part 2: Define an Output Operator

1. Modify your pool class header to define an ostream insertion operator for objects of type *pointer-to-pool*:

- a. After the pool class definition, insert the following overloaded ostream insertion operator:

```
template<class T> ostream& operator<< ( ostream& os, const Pool<T>*
    pool ) {
    const typename Pool<T>::Node* next = pool->first ;
    while ( next ) {
        ::operator<< <T> ( os, next ) ;
        os << "\n" ;
        next = next->next ;
    }
    return os ;
}
```

Change type and member names as needed. If you defined the node structure in the file scope, then qualification is not needed, and you can directly insert the node pointer into the output stream.

When you insert a pointer-to-pool object into an output stream, this function traverses the pool nodes and for each pool node, inserts the node into the output stream, followed by a new-line character. The function then returns the output stream reference to the caller.

- b. If anything in the pool class that the overloaded ostream insertion operator accesses is not public, then in the pool class definition, declare the operator a **friend** function so that it can access all regions of the pool class.

Input and Output

```
template<class TP> friend ostream& operator<< ( ostream&, const
    Pool<TP>* ) ;
```

Note: The identifier “TP” can be any identifier not already in use.

2. Modify your pool class header to define an ostream insertion operator for objects of type pointer-to-node. What you insert into the output stream should be similar to:

```
{obj_ptr = 0x804b068, available = false, next = 0x804b038}
```

After the pool class definition, insert the overloaded ostream insertion operator. Cast all pointer types to pointer-to-void before inserting them to prevent inadvertently calling overloaded ostream insertion operators for those classes. If your checked-out indicator is type **bool**, then display “false” or “true” instead of “0” or “1”.

3. If anything in the node class that the overloaded ostream insertion operator accesses is not public, then in the node class definition, name the operator a friend function so that it can access all regions of the node class.
4. Modify your pool class to define a new void method to display the pool contents by inserting the pool pointer into the cout output stream. The method should be **const**.
5. Modify the pool test to call your new method to display the pool contents. Do so after all objects are checked out and again after all objects are checked back in.
6. Compile exception, pool, and test; execute; debug as needed.



Module 16: Debugging

Lab 16-1 Debugging a Program

Objective: To diagnose incorrect program behavior.

For this lab, you watch nodes become added to the list as your program checks out objects, and then watch nodes become checked back into the pool.

Part 1: Review GDB and DDD

Review the training materials where they describe debugging with GDB and DDD.

Part 2: Watch Nodes Being Checked Out and In

1. Compile with the -g compiler option, your exception class, pool, and test.
2. Invoke the debugger with your executable file as its invocation argument.
3. Select and view the pool class header source file:
DDD: Menu item **File – Open Source...** and in the “Open Source” dialog, filter to and select your header file.
GDB: **list *filename:line_number_start,line_number_end***
4. Place breakpoints on the first line of the destructor, the check-out method, and the check-in method:
DDD: **Double**-click the line in front of the function declaration.
GDB: **break *line_number***
5. Run the program:
DDD: Command tool button **Run**.
GDB: **run**
6. Watch nodes become added to the list as your program checks out objects:
 - a. Upon hitting the first breakpoint (presumably the check-out method), examine the value of whatever variable you use to point to the first list node:
DDD: **Double**-click the variable to send it to the Data Window.
GDB: **print *identifier***
The variable’s value should at this point be null.

- b. Continue the program:

DDD: Command tool button **Cont.**

GDB: **cont**

Your program should again hit the check-out method breakpoint.

- c. Again, examine the value of the pointer variable.
- d. The variable's value at this point should not be null.
- e. Now examine the node object that the pointer variable points to:
DDD: **Double**-click the variable's value where displayed in the Data Window.
GDB: **print {type}address**
Note: Alternatively, call your display method, e.g.: **call display()**
You should see the node's checked-out indicator showing it is checked-out.
You should see the node's next-node pointer value is null.
- f. Continue in this manner until the program hits the check-in breakpoint, indicating that no more nodes will be checked out.
Note: The DDD Data Window may clear when the scope shifts from the check-out method to the check-in method. If it does, then simply re-display the list nodes as you originally did.

- 7. Watch nodes become checked back into the pool:

- a. Continue the program:

Your program should again hit the check-in method breakpoint.

- b. Now examine your node list to confirm that an object is checked in:

DDD: The Data Window displays the updated node values.

GDB: Follow the list node pointers as before...

Note: As the list structure has not at this point changed since you last looked at it, you can '**set history expansion on**' and '**show history**' and then use the C-shell style '!number' command to redo a command on the command history list.

You should see the node's checked-out indicator showing it is checked-in.

- c. Continue in this manner until the program hits the destructor breakpoint, indicating that no more nodes will be checked in.
Note: If using DDD, again re-display the list nodes if necessary.

8. Exit the debugger:

DDD: Menu item **File – Exit**

GDB: **detach** and **quit**



Module 17: Containers and Algorithms

Lab 17-1 Instantiating and Using Standard Containers

Objective: To store and manipulate data by using standard containers and standard algorithms.

In this lab, you determine which container can most appropriately replace the list structure of your object pool, then modify your pool class to utilize that standard container.

Part 1: Review Containers and Algorithms

Review the training materials where they describe standard containers and standard algorithms.

- ◆ Standard containers
- ◆ Sequence containers
 - vector, list, deque
- ◆ Sequence adapters
 - stack, queue, priority_queue
- ◆ Associative containers
 - map, set, multimap, multiset
- ◆ Standard algorithms
- ◆ Non-modifying
 - for_each(), count(), find(), search(), equal()

Part 2: Utilize a Standard Container

1. Determine which container can most appropriately replace the list structure of your object pool.

Consider that in a program that runs for a long time, most of its access to the pool would be searched for objects that already exist. The most obvious choice might be the standard list container, which efficiently inserts and erases elements of an arbitrary type based on an iterator. To search a list requires an algorithm to visit an average of half the elements. However, we do not have an arbitrary element type. We know that each element of our type is unique because each element has a pointer to a unique pooled object. That suggests we store the elements as a sparse array, where the pointer is the index and the value at each index is the checked-out indicator. To search a sparse array requires an algorithm to visit an average of \log_2 the elements. A standard map container is a sparse array.

2. Review the training materials more closely, where they describe the standard map container.

The standard map container is a class template with parameters for the key type and the mapped type, i.e., `map<Key,T>`. A map manages elements of the standard pair type. The pair is a **struct** (described below) having a field first of the key type and a field second of the mapped type. Maps support all standard container methods except `push()` and `pop()`, and support additional methods specific to maps.

```
template <class T1, class T2> struct pair {
    typedef T1 first_type; typedef T2 second_type;
    T1 first;
    T2 second;
    pair(); // default
    pair(const T1& x, const T2& y); // init
    template<class U, class V> pair(const pair<U, V> &p); // copy
};
```

3. Modify your pool class to utilize a standard map container:
 - a. Include the `<algorithm>` and `<map>` libraries.
 - b. Remove the node pointer and instead instantiate a map.
 - c. Modify the pool destructor to use a `const_iterator` to iterate through the map, and for each pair element, delete whatever object the first field of the element points to.
 - d. Modify the check-out method to do a predicate-based `find_if` for the first element whose second field indicates that it is not checked out. You will need to write the static predicate function or functor. It must accept a const element reference and return a bool result. Remember that the element type is `pair<T*,bool>`. If an element is available for check-out, mark it as checked-out and return the object pointer. If no element is available for check-out, `insert()` (use a map-specific `insert`) a new element with its object pointer initialized to a new object, marked as checked out, and return the object pointer.
 - e. Modify the check-in method to do a map-specific `find` for the passed-in pointer. You can alternatively use the `find` standard algorithm but remember that it looks for an element and knows nothing about keys. Having found the element, clear its checked-out indicator.

Why can you NOT use the map subscript operator to access the indicator?

Answer: Answer: Because `map[key]=value;` adds a new element if it does not exist!
 - f. Modify the node ostream insertion operator to accept a `const_iterator` argument and, through it, display the element's object pointer and checked-out indication (very similar to the previous lab).

- g. Modify the pool ostream insertion operator to use a `const_iterator` to iterate through the map and pass the iterator to the modified node ostream insertion operator.
4. Compile exception, pool, test; execute and debug as needed.



Module 18: Introduction to System-C

Lab 18-1 Implementing FIR Filter in C++ and Running Stratus Tool to Generate the RTL

Objective: To perform high-level synthesis of the FIR filter written in C++ using the Stratus tool.

In this lab, you implement FIR filter using C++, put a System-C wrapper on it, and then run the Stratus™ tool on the same to generate the Verilog RTL. We simulate both C++ implementation and the RTL code and check if both behave the same way or not. We can also synthesize the design and view the Synthesis report.

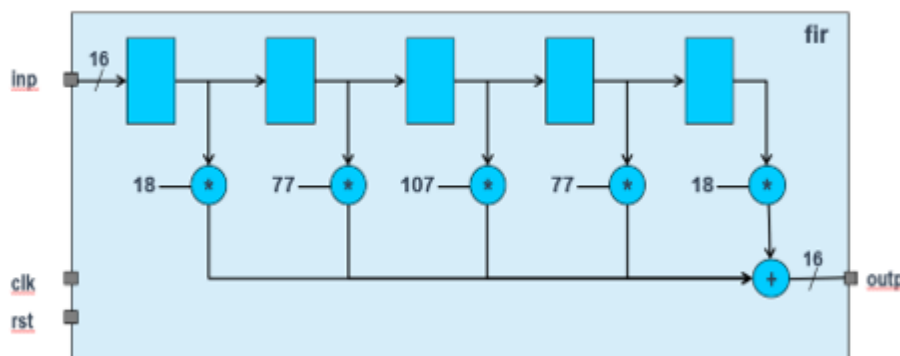
Part 1: Review System-C

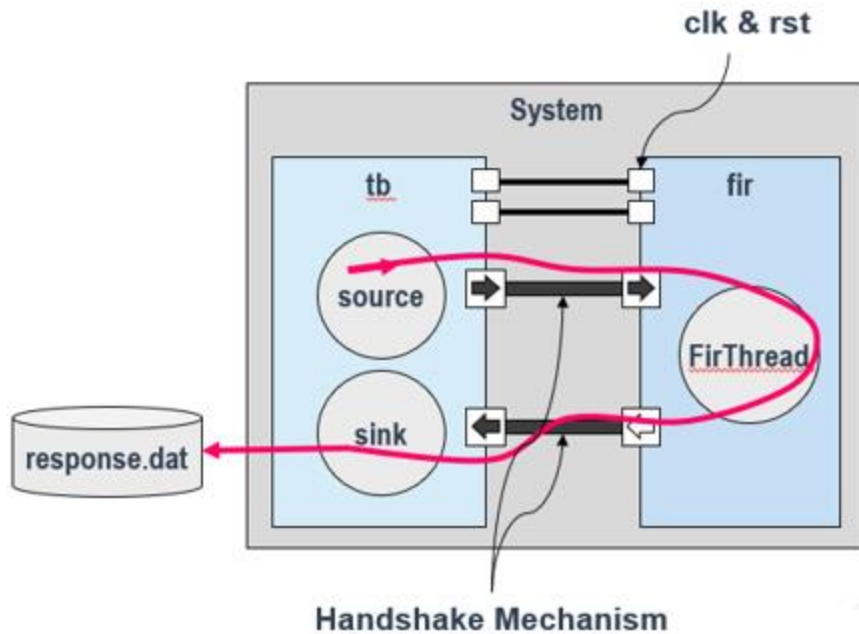
Go through the System-C fundamentals in order to be able to put a wrapper around the C++ code that implements the FIR filter.

Part 2: FIR Filter Design

In this lab, we also perform behavioral simulation on high-level code, synthesize the high-level System-C to generate Verilog RTL, and run the RTL simulation.

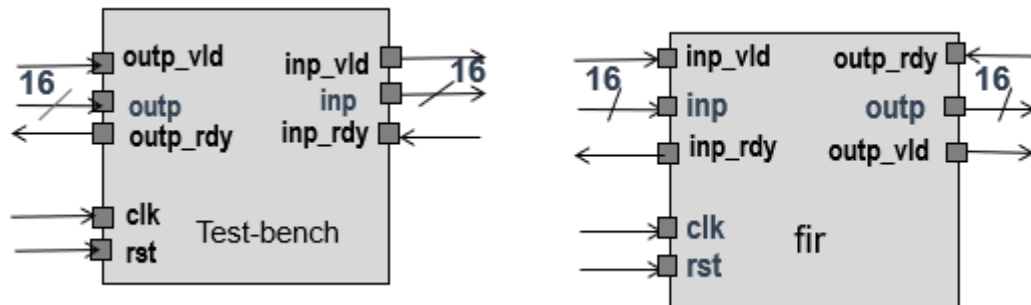
- ◆ This is a simple design. It receives input from the testbench, computes a 5-tap FIR, writes the result to the output file, and compares the actual output with the golden reference.
- ◆ System module instances *tb* and *fir* modules. We will learn this later in the verification section. I/O uses a handshaking mechanism.





Handshake Mechanism

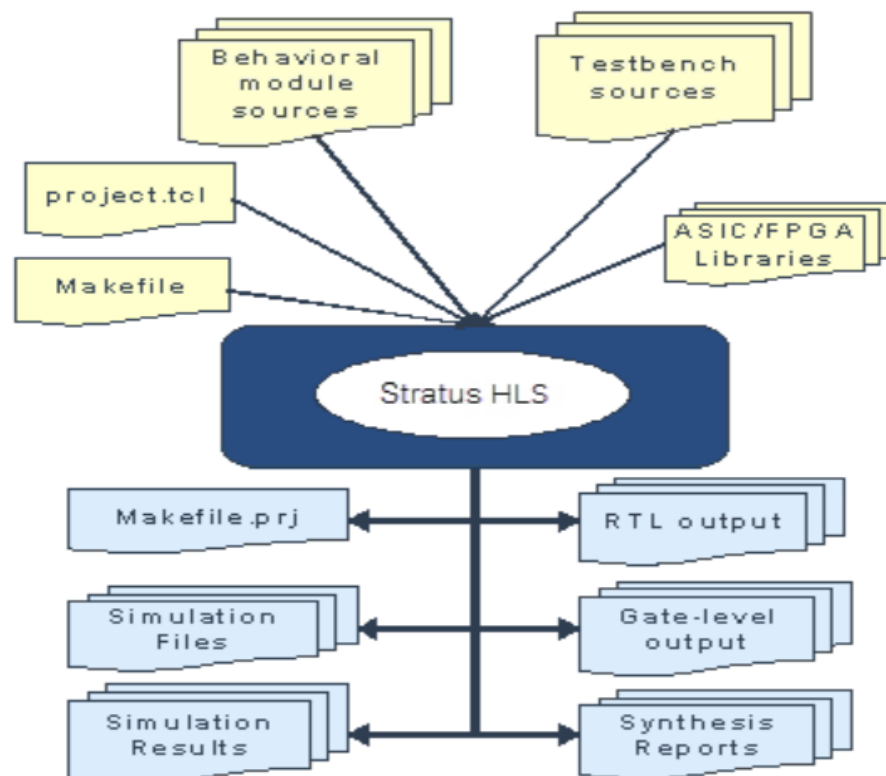
We implement the handshake using valid and ready signals. It ensures no data is lost.



1. In the testbench sink task (output from dut is captured):
 - a. Set `outp_rdy = 1`, denoting tb is ready to receive the output of dut.
 - b. Read the output, `outp` from the dut, when `outp_vld` goes high, indicating that there is data to be read.
 - c. Drive the `outp_rdy` low.
 - d. Repeat the same for every sample.

2. In the testbench source task:
 - a. Drive input sample onto the input of dut, inp, by setting inp_vld high. Wait until inp_rdy goes high, i.e., until dut receives the input.
 - b. Then, drive inp_vld low.
 - c. Repeat the same for every sample.
3. Similarly, in the DUT fir_main thread:
 - a. Set inp_rdy high, indicating dut is ready to receive input from tb; wait until inp_vld goes high, indicating an input from tb.
 - b. Read inp port, inp into inp_val variable, drive low on inp_rdy.
 - c. Process the value read. Shift the input read into the shift register, multiply with the coefficients, and accumulate the result.
 - d. Write the processed output onto the output port, outp by driving outp_vld high. Wait until outp_rdy goes high. Drive outp_vld low.
 - e. Repeat this for every sample.

Part 3: Stratus Project Setup



A project directory has source code, project configuration files, testbench, stimulus data and other related files. For example, this lab project has the following.

Name	Required File Contents
project.tcl	Defines libraries, configurations, simulators, VCD logging, integration settings, etc.
Makefile	Has user-written rules for project setup, result comparison, cleanup, etc.
Makefile.prj	Has Stratus-generated rules for running HLS, simulation, etc., using project.tcl file.
bdw_work	Stratus-created directory that holds all project-specific files generated by Stratus.
cachelib	Stratus-created directory that caches resources created by Stratus.
Name	Project-Specific File Contents
Readme.txt	Brief project description.
defines.h	Project-specific #defines, typedefs, etc.
fir.cc, fir.h	Module to be synthesized: you can have multiple modules with a hierarchy.
tb.cc, tb.h	Testbench module.
golden.dat	Golden results file used by the testbench.
main.cc	sc_main function instantiates the System module and starts the simulation.
system.cc, system.h	The system module that instantiates the design and the testbench.

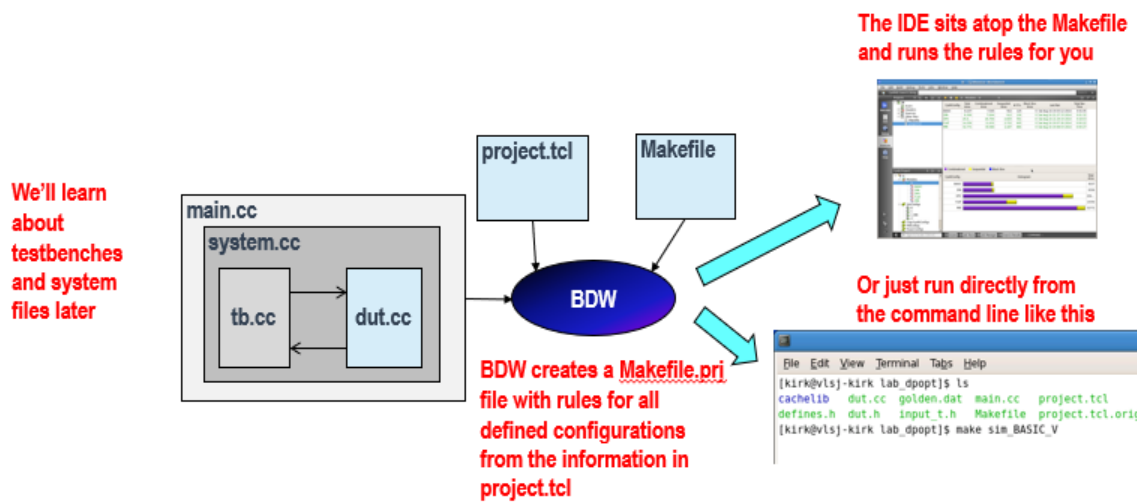
project.tcl Structure

Stratus project files are simple Tcl files.

1. They use Tcl syntax with specialized Stratus Tcl commands.
2. The Stratus Tcl commands allow you to define:
 - a. Technology libraries
 - b. Memory and interface libraries

- c. Global attributes
- d. System-C and Verilog simulators with waveform logging
- e. Synthesis modules and configurations
- f. Simulation configurations
- g. Logic synthesis configurations and more

project.tcl File



1. The project.tcl file defines synthesis, simulation, and other configurations.
2. The BDW script generates make rules in a project "Makefile.prj" to run all of them from the GUI or command line based on the project.tcl file.

project.tcl File

```

# Libraries
set LIB_PATH
"[get_install_path]/share/stratus/techlibs/GPDK045/gsclib045_svt_v4.4/gsclib045/timing"
use_tech_lib "$LIB_PATH/slow_vdd1v2_basicCells.lib"

# C++ compiler options
set CLOCK_PERIOD 6
set_attr cc_options " -DCLOCK_PERIOD=$CLOCK_PERIOD"
set_attr hls_cc_options " -DCLOCK_PERIOD=$CLOCK_PERIOD"

# Global synthesis attributes
set_attr clock_period $CLOCK_PERIOD
set_attr message_detail 2
set_attr default_input_delay 0.1
set_attr path_delay_limit 120
set_attr prints off
set_attr rtl_annotation op,stack
set_attr flatten_arrays all
set_attr unroll_loops off
set_attr default_protocol on

```

Generic 45nm library included with Stratus HLS

Command line parameters for C++ compilation

Project-wide settings

Note- ensures wait statements are not optimized away . Else, RTL and systemC simulations mismatch

```

# Module Configurations
define_system_module main main.cc
define_system_module System system.cc
define_system_module tb tb.cc

define_hls_module fir fir.cc

# Synthesis Module Configurations
define_hls_config fir BASIC
define_hls_config fir DPOPT

```

3 system_modules (testbench) defined

1 hls_module (design) defined

2 hls_configs (micro-architecture) defined

```

# Simulation Configurations
use_verilog_simulator xcelium ;# 'xcelium' or 'vcs'
enable_waveform_logging -vcd ;# to store signal transitions vcd or fsdb
set_attr end_of_sim_command "make cmp_result"

# A behavioral PIN-level configuration
define_sim_config B {fir BEH BASIC}

# The following rules are TCL code to create a simulation configuration
# for RTL_V for each hls_config defined.
foreach config [find -hls_config *] {
    set cname [get_attr name $config]
    define_sim_config ${cname}_V "fir RTL_V $cname"
}

# Genus Logic Synthesis Configurations
define_logic_synthesis_config G {fir -all} \
    -options \
    {BDW_LS_NOGATES 1}

```

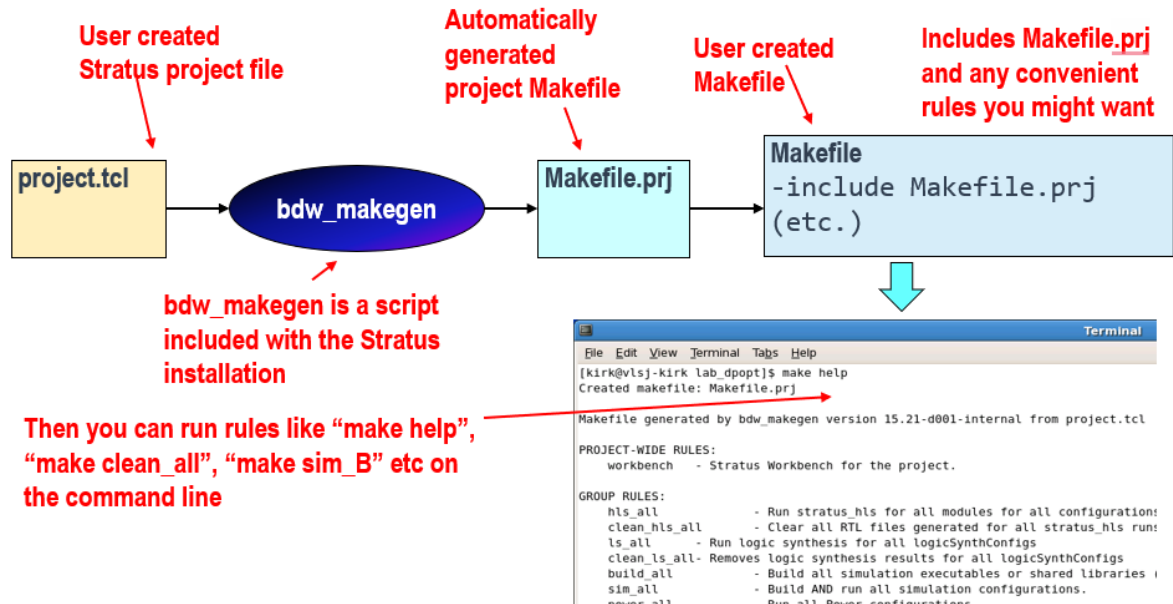
The "cmp_result" Makefile rule will run after each simulation

Create a simulation configuration for each microarchitecture

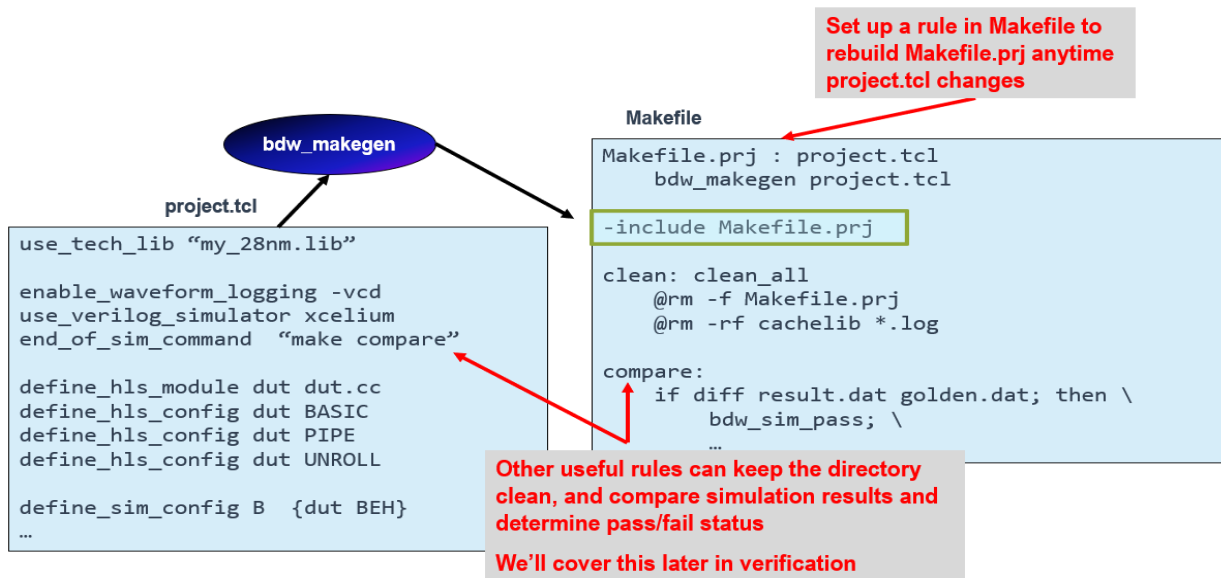
Create a logic synthesis configuration for each microarchitecture

Makefile-project.tcl Relationship

BDW creates a Makefile.prj from project.tcl, which you then include in your top-level Makefile.



Use a make rule to keep Makefile.prj fresh.



Mechanisms for Controlling Synthesis

Stratus HLS combines the use of a number of mechanisms for controlling synthesis and other processes.

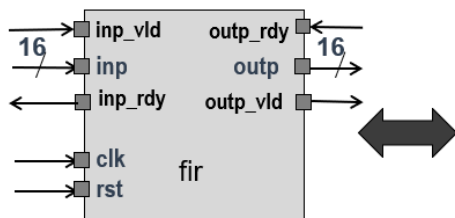
1. Project commands and project attributes:

- a. Set using Tcl in the project.tcl file.
 - b. Used for defining modules, various kinds of configs, etc.
 - c. Used for configuring project-wide settings.
2. Synthesis control attributes:
- a. Set using Tcl in the project.tcl file.
 - b. Used for controlling the HLS process and for controlling microarchitecture decisions.
 - c. It can be used on a project-wide or hls_config-specific basis.
3. Synthesis control directives:
- a. Set using source code directives in the C++ code.
 - b. These are an alternative way to set synthesis control attributes for controlling microarchitecture decisions.

Part 4: Implementation

FIR Module Declaration

The fir module declaration goes into fir.h header file. This header file contains a declaration of various System-C function variables for your design and port definitions.



```
#include <systemc.h>

SC_MODULE( fir ) {
    sc_in< bool >      clk, rst;
    sc_in< sc_int<16> > inp;
    sc_in< sc_uint<1> > inp_vld;
    sc_out< sc_uint<1> > inp_rdy;
    sc_out< sc_int<16> > outp;
    sc_in< sc_uint<1> > outp_rdy;
    sc_out< sc_uint<1> > outp_vld;

    void fir_main();

    SC_CTOR( fir ) {
        SC_CTHREAD( fir_main, clk.pos() );
        reset_signal_is( rst, true );
    }
};
```

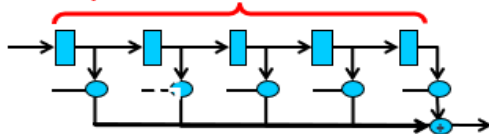
fir.h

FIR Module Definition

fir.cc is the main source file for your module. This file can contain all of the sources for your module or can reference lower-level files.

taps[] is a variable array to represent the shift register of tap values

coefs[] is a constant array with the 5 tap multiplier values



```

#include "fir.h"                                     fir.cc

const sc_uint<7> coefs[5] = {18,77,107,77,18};

void fir::fir_main() {
    sc_int< 16 >      taps[5];

    outp.write( 0 );                                Reset output
    wait();                                           port and wait
                                                    one cycle

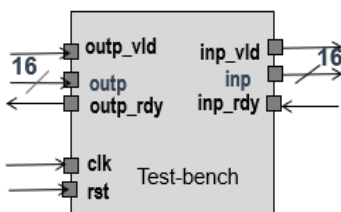
    while( true ) {
        //load the first value with the value read on
        //the input port and shift tap values
        //Multiply each tap by its coefficient and
        //accumulate on variable val
        //Write accumulated value to output
        //wait for one cycle and repeat while loop
        wait();
    }
}

```

Testbench

The testbench header(tb.h) file contains the declaration of various System-C functions for your testbench. tb.cc file contains the testbench module that drives data into the design and reads output data from the design.

It also contains a System-C function that is called at the end of all stimulus and responses that stops the simulation.



```
#include <systemc.h>
SC_MODULE(tb)
{
    sc_in<bool>          clk;
    sc_out<bool>         rst;
    sc_out<sc_int<16>>   inp;
    sc_out<sc_uint<1>>   inp_vld;
    sc_in<sc_uint<1>>    inp_rdy;
    sc_in<sc_uint<1>>    outp;
    sc_in<sc_int<16>>    outp_vld;
    sc_out<sc_uint<1>>   outp_rdy;

    void source();
    void sink();

    SC_CTOR(tb) {
        SC_CTHREAD( source, clk.pos() );
        SC_CTHREAD( sink, clk.pos() );
    };
};
```

tb.h

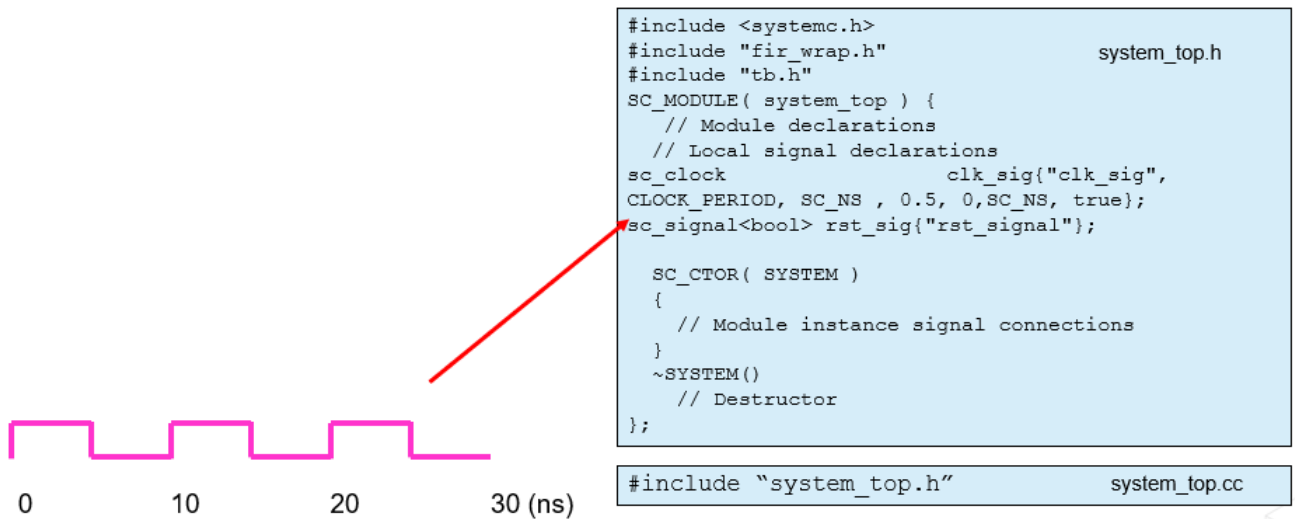
```
#include "tb.h"
void tb::source() {
    //drive reset pulse
    //drive input values to FIR...
}
void tb::sink() {
    //read FIRoutput and write it to the output.dat file
    //stop simulation using sc_stop
}
```

tb.cc

System_top

1. Instantiates the design and tb modules, declare local signals.
2. In the constructor, connect the module instance signals, and delete instances in the destructor once done.
3. Has the clock and reset definition.

Note: Here, we instantiate fir_wrap generated by the Stratus tool instead of the fir module.



Main File

1. Creates a new instance of the system for the System-C simulator.
2. It consists of `esc_elaborate`, `esc_cleanup`, and `esc_initialize` functions.
3. This file holds functionality that starts the simulation and initializes signals and objects in the simulation.
4. Deletes the system from memory when a simulation run is complete.
5. Contains `esc_*` functions necessary for SystemC or mixed SystemC/Verilog co-simulation.

```
#include <systemc.h>           // SystemC definitions
#include "system_top.h"        // Top-level System module header file
#include "esc.h"
static system_top * m_system = NULL; // The pointer that holds the top-
                                     level System module instance.

// This function is where elaboration is done to support SystemC-
// verilog co-simulation.
void esc_elaborate()
{
    m_system = new system_top( "system_top" );
    // top-level module should be created using new
}

// This function is called at the end of the simulation by the Stratus
// co-simulation hub.
// It should delete the top-level module instance.
void esc_cleanup()
{
    delete m_system;
}
```

```
// This function is called by the Accellera SystemC kernel
// It is not executed by the Xcelium SystemC simulation
int sc_main( int argc, char ** argv )
{
    // esc_initialize() passes in the cmd-line args. This initializes the
    // Stratus simulation environment (such as opening report files for later
    // logging and analysis).
    esc_initialize( argc, argv );

    // esc_elaborate() (defined above) creates the top-level module
    // instance. In a SystemC-Verilog co-simulation, this is called during cosim
    // initialization rather than from sc_main.
    esc_elaborate();

    // Starts the simulation. Returns when a module calls esc_stop(),
    // which finishes the simulation.
    // esc_cleanup() (defined above) is automatically called before
    // sc_start() returns.

    sc_start();
    // Returns the status of the simulation. Required by most C
    // compilers.
    return 0;}
```

Part 5: Execution Steps

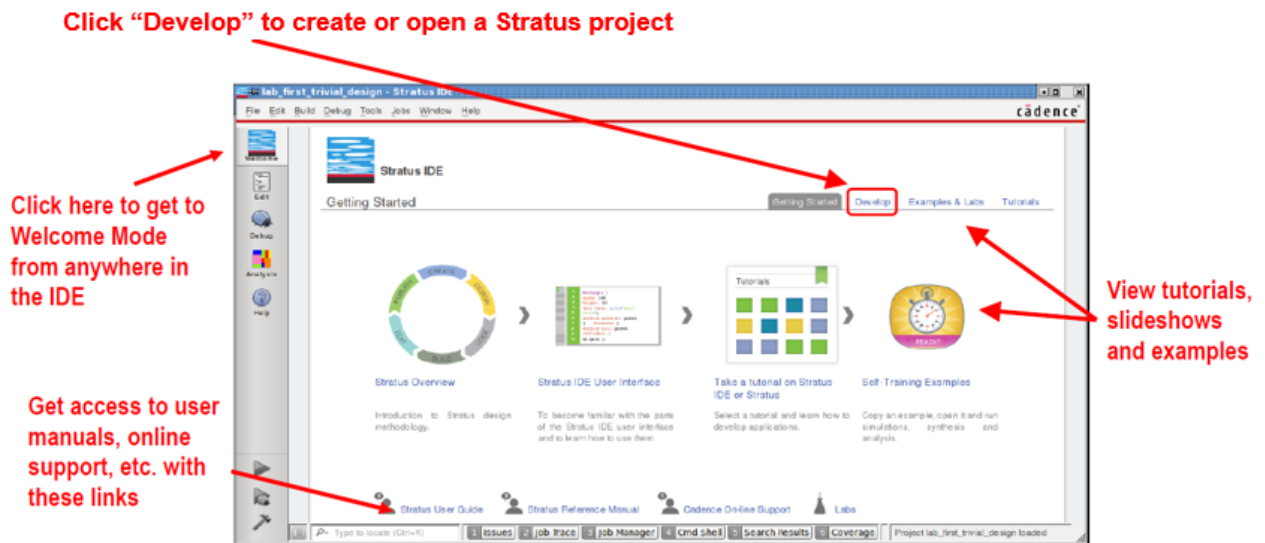
Open the file `tb.h` and go through the port list and the function declaration. Open the file `tb.cc` and add the missing code.

1. In the source thread:
 - a. We drive 64 values, with values from 23 to 29 as 256.
 - b. All other samples are set to 0s.
 - c. We assign values to `temp` and write `temp` value onto the `inp` port.Implement the handshake mechanism for driving input:
 - d. Drive each sample by setting `inp_vld` high and `inp` port with `temp`.
 - e. Wait until `inp_rdy` goes high.
 - f. When `inp_rdy` goes high, drive `inp_vld` to 0.
2. In the sink thread, implement the handshake mechanism for reading output. For each sample:
 - a. Drive `outp_rdy` high until `outp_vld` goes high.
 - b. Read `outp` port into `indata` variable.
 - c. Drive `outp_rdy` low.
 - d. Write the value onto the output file, `output.dat`.
3. Open the file `system_top.h` and add the missing code:
 - a. Create the `fir_wrap` (and not `fir` instance) instance just like `tb` instance.
 - b. Declare all the local signals to connect `dut` and `tb`.
 - c. Bind all the `dut` ports and `tb` ports to the local signals in the constructor.
 - d. In destructor, delete the `fir` instance like the `tb` instance.
4. `System_top.cc` only includes `system_top.h`.
5. The `main.cc` file instantiates:
 - a. `system_top`.

- b. Uses `esc_*` functions to facilitate hardware/ software co-simulation.
6. The `project.tcl`, `makefile`, and the main file are provided in the lab database.
7. Use the following attribute in the `project.tcl` file:
To ensure that the RTL and system-c simulations don't mismatch due to the optimization of wait statements, use:
 - a. `set_attr default_protocol`

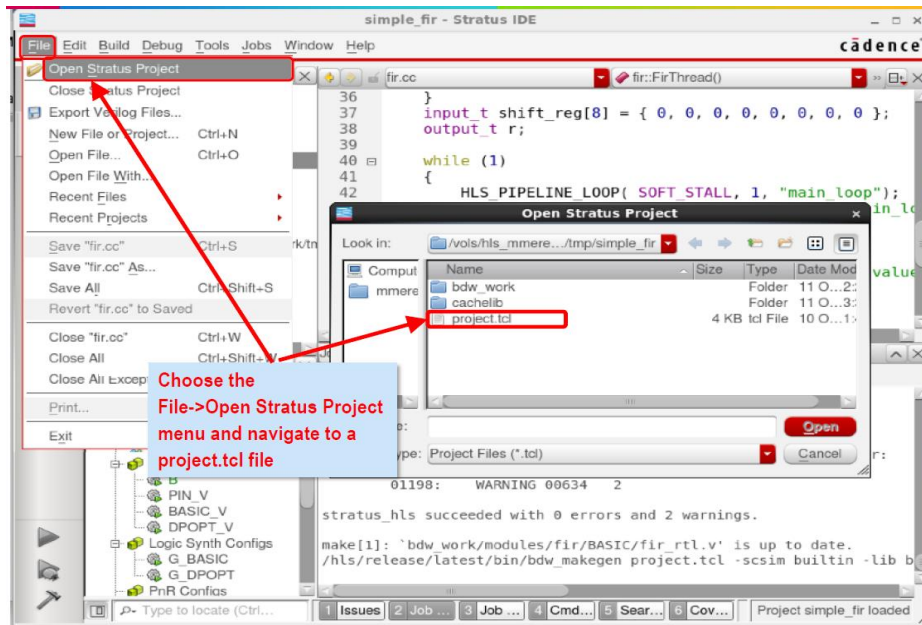
Invoking the Tool

1. Go to the directory where you have saved the project.
2. Start the Stratus IDE GUI by executing the command "`stratus_ide &`".
3. The Stratus IDE comes up in its Welcome mode.



Opening a Project

From the file menu, navigate to any project.tcl file. Or, from the Develop tab in Welcome mode, choose a recently used project.



Stratus IDE Edit Mode

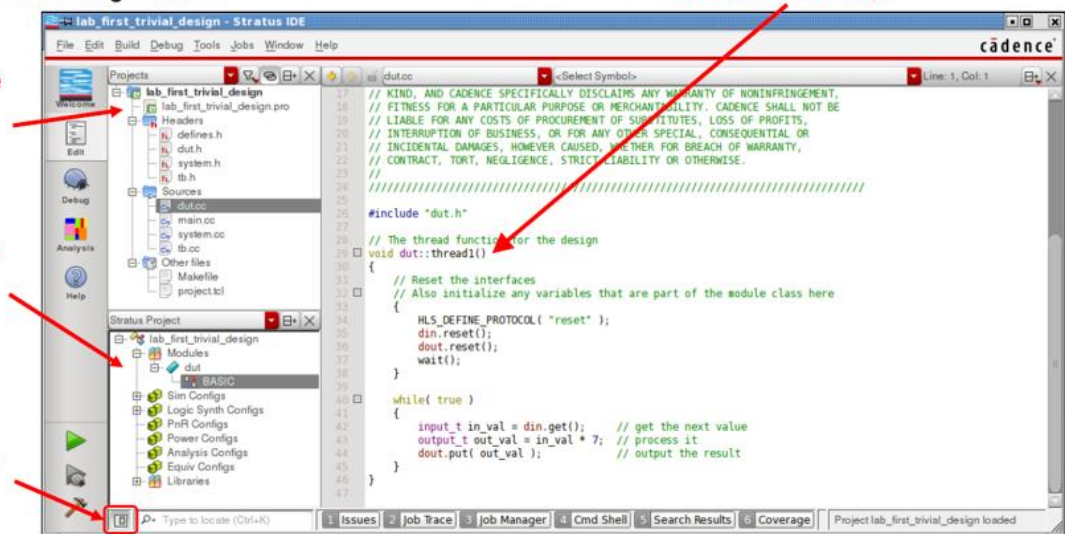
- Edit files and run configurations

Use the Projects pane to edit header, source, project and Makefiles

Use the Stratus Project pane to run all defined synthesis, simulation, logic synthesis, and other configurations

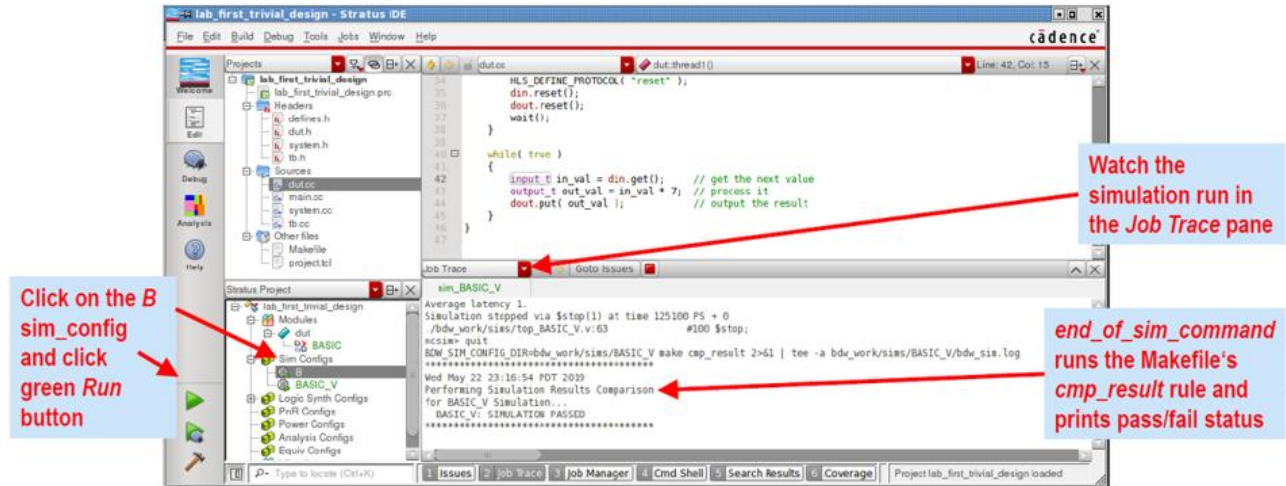
This button can be used to hide the tree panes for extra real estate

Edit text files directly



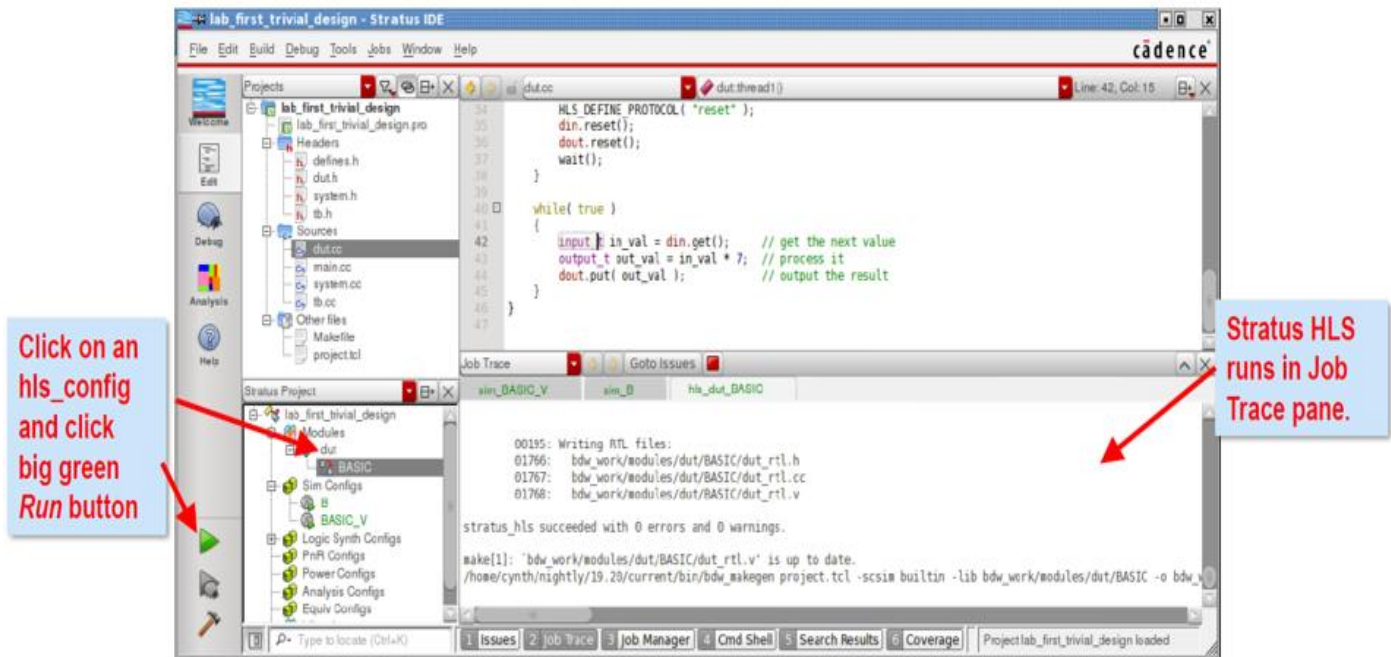
Run a Behavioral Simulation

Run the behavioral simulation configuration, B. Similarly, run the RTL simulation configuration, BASIC_V. In both cases, the simulation should pass.



Run Stratus HLS from the IDE

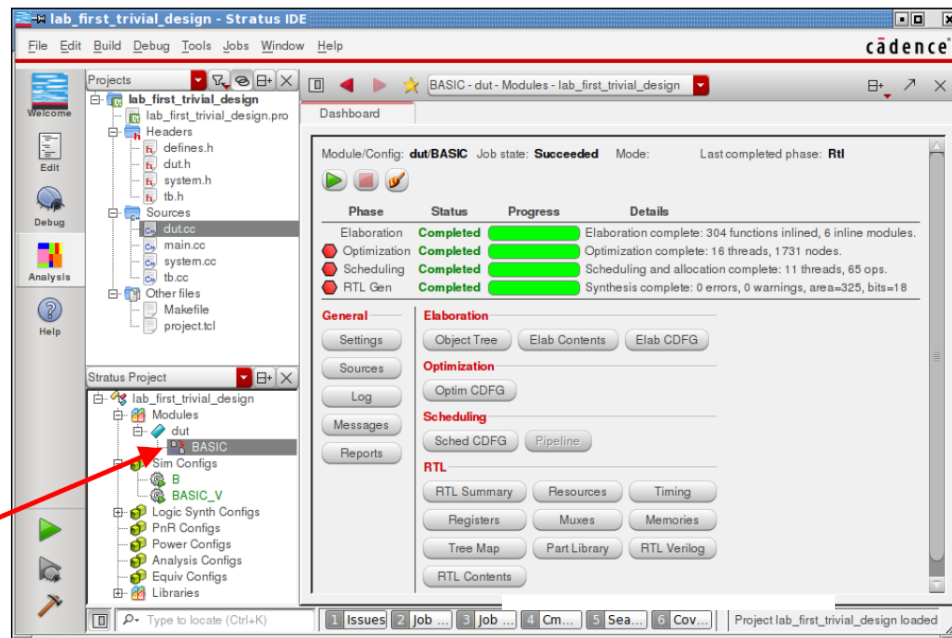
Create an RTL architecture out of basic synthesis configuration.



Stratus IDE Analysis Mode

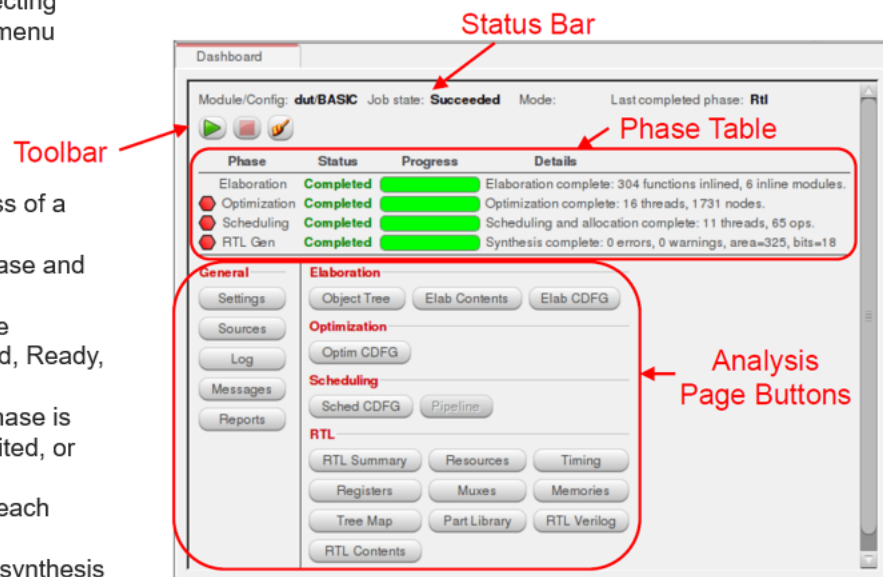
- The Analysis pane provides a central location for viewing synthesis results
- But during a synthesis, the analysis pane disappears
- Analysis results are unavailable until the synthesis is done
- The early access feature addresses this limitation

Double-click on **BASIC hls config** IDE switches to Analysis Mode or select the hls_config and choose the Analysis mode



Analysis Window Dashboard

- The dashboard is raised by selecting Analyze in the config's context menu
 - Components
 - Status Bar
 - Toolbar
 - Phase Table
 - Analysis page buttons
- The phase table tracks the progress of a running Stratus HLS job
- Phase-Indicates a synthesis phase and provides access point controls
 - Status-Current status of a phase (Completed, Error, Exited, Failed, Ready, Running)
 - Progress-Shows whether the phase is running, paused, completed, exited, or failed
 - Details-Describes the status of each phase
 - Updated in real-time during the synthesis



To Run Jobs from a Linux Shell Window

1. Use the “make” command.
2. Run “make help” to see options.
3. To run Stratus HLS:
 - a. Make hls <hls_config name>.
 - b. E.g., “make hls_BASIC”.
4. To run the simulation:
 - a. Make sim_<sim_configname>.
 - b. E.g., “make sim_B”.
5. To run logic synthesis:
 - a. Make ls_<logic_synthesis_configname>.
 - b. E.g., “make ls_G_BASIC”.

Part 6: Instructions Summary

Use the IDE Edit mode to examine the project.tcl file, and other files.

1. Run a behavioral simulation.
 - a. Execute sim_config B.
2. Run high-level synthesis.
 - a. Execute hls_config BASIC on the fir module.
 - b. Use the IDE Analysis mode to investigate the results of HLS.
3. Run an RTL simulation.
 - a. Execute sim_config BASIC_V.

Examining a Generated RTL Verilog

1. **Double**-click the **BASIC** hls config to get the Resources tab back.
2. Click the “+” tab and choose the **RTL Verilog** button.
3. The Contents of the *dut_rtl.v* RTL Verilog file is displayed. This is the RTL generated for the BASIC hls config.
4. Examine the project.tcl file.
 - a. See the commands for library setup, attribute settings, and configurations for synthesis and simulation.
5. Click the “+” next to Sources and **double**-click the file to view its content.
6. Run a behavioral simulation.
 - a. Select sim config **B** in the Stratus Project pane and click the green **Run** button.
Or, **right**-click on sim config **B** and select **Run** in the pop-up menu.
 - b. The Job Trace window appears in the lower right with a “sim_B” tab. Watch the source files be compiled and the B simulation run to completion.
 - c. Note that “*B: Simulation PASSED*” is printed. This is done in the cmp_result target in the Makefile.
7. Run Stratus HLS.
 - a. Select the **BASIC** hls config and click the green **Run** button.
Or, **right**-click on hls config **BASIC** and select *Synthesize* in pop-up menu.
 - b. A new “hls_fir_BASIC” tab appears in the Job Trace window, where you can watch the Stratus HLS schedule the design and print an area estimation.
8. Analyze the Stratus HLS results.
 - a. **Double**-click on the BASIC hls config.
 - b. The dashboard appears. Many analysis tabs are now available.
 - c. Click the **Resources** button.
 - d. Click on **Area** in the table header to sort by area until the largest parts are at the top.

- e. Check the largest part is “fir_Mul_8Ux8U_11U_4_9” with area 315.

This is a multiplier with 8-bit unsigned inputs and an 11-bit unsigned output.

- f. Click the “+” sign in the row next to this part.

Check a row with a “*” operator appears underneath.

Right-click on the “*” operator and select **Show In C++ Source** in the pop-up menu.

- g. A new Sources tab appears containing the fir.cc source. Multiply operator (*) is highlighted.

- 9. Run the RTL_V simulation of the generated Verilog.

- a. Select the **BASIC_V** sim config and click the green **Run** button.

Or, **right-click** on **BASIC_V** and select **Run** in the pop-up menu.

- b. A new “sim_BASIC_V” tab appears in the Job Trace window, and the simulation runs. “BASIC_V: SIMULATION PASSED” is printed.

- 10. Examine generated RTL Verilog.

- a. **Double-click** the BASIC hls config to get the Resources tab back.

- b. Click “+” tab and choose the RTL Verilog button.

- c. Contents of dut_rtl.v RTL Verilog file is displayed. This is the RTL generated for the BASIC hls config.

- 11. Experiment with other analysis tabs from the dashboard.

- a. Line numbers in the “Log” tab are hyperlinks.

- b. Timing tab shows the longest combinational paths.

- c. Pipeline tab shows which operations are scheduled in which pipeline stage, if any.



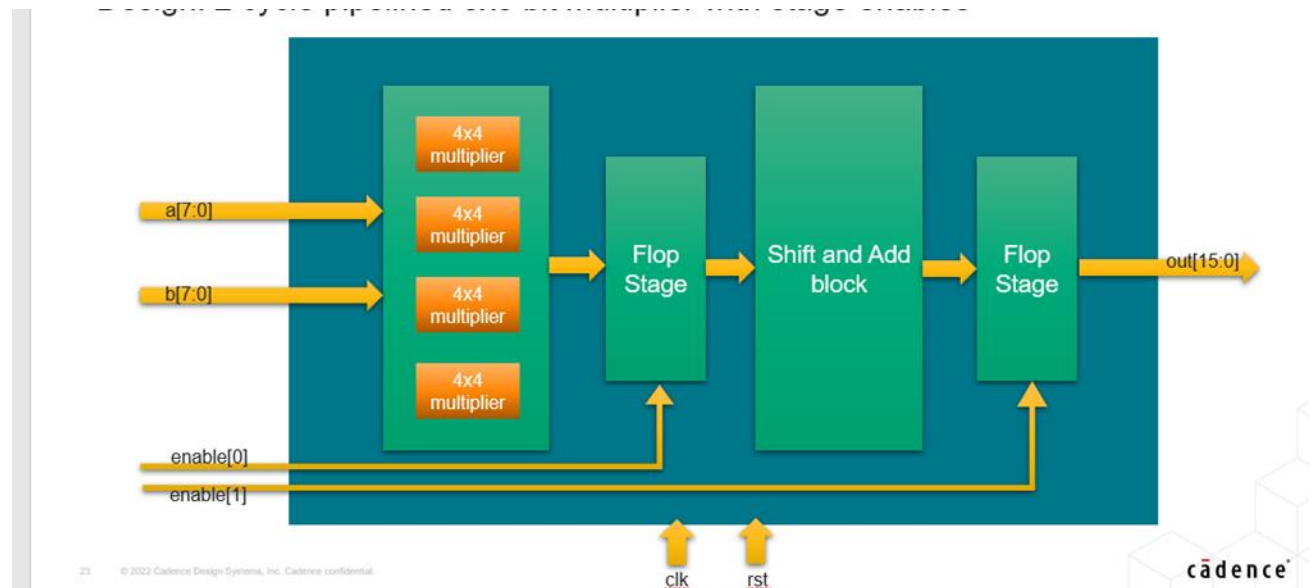
Module 19: Equivalence Checking C++ for Verification

Lab 19-1 Equivalence Checking of a Pipelined Multiplier

Objective: To perform an equivalence check between various implementations of a pipelined multiplier.

In the lab, we will perform an equivalence check between a C++ and RTL implementation of a pipelined multiplier, as shown in the diagram. In addition, equivalence check 2 different C++ models. The scripts to run each configuration are specified in the lab README file in the lab 19 directory.

Detailed in the steps below are the model files to use in each run and the goals of each run.



Part 1: Fixed Delay Standard Miter Model Setup

1. Change the directory to <Labs Directory>/LAB19-C2RTL_equivalence.
2. Read the README file, which details which scripts run which configuration files:

```
Spec : c_model/mult.cpp
Imp  : rtl/mul.sv
```

The goal is to set up a fixed delay equivalence setup (use assumes to tie off stage enables to active values).

Part 2: Arbitrary Slice Miter Model Setup

Files

```
Spec   : c_model/mult.cpp
Imp    : rtl/mul.sv
```

Goals

- To set up an arbitrary slice equivalence setup (use reset –none).
- Prove the setup.

Part 3: Variable Delay Standard Miter Model Setup

Files

```
Spec   : c_model/mult.cpp
Imp    : rtl/mul.sv
```

Goals

- Set up an equivalence check miter model for variable delay.
HINT: Use a constraint model provided in rtl/sva_env.sv.
- Analyze and elaborate this is ‘-repository 1’.
- Refer to help on ‘analyze’ and ‘elaborate’ for working with repositories.
- Use the connect command to instantiate it and connect to the Spec side and Imp side signals.
- Refer to help on ‘connect’.

Part 4: C Versus C Equivalence Checking Setup

Files

```
Spec   : c_model/mult.cpp
Imp    : c_model/mult_imp.cpp
```

Goals

- Set up a C versus C equivalence setup.
- Prove the setup.

Setup 5: RTL Versus RTL Equivalence Checking Setup for Datapath Designs

Files

Spec : rtl/mul_comb.sv
Imp : rtl/mul.sv

Goals

- Set up an RTL versus RTL equivalence checking setup (for variable delay – refer to Setup 3, just replace the C++ model with this new SV model on the spec side).
- Prove the setup.

