UNIVERSITY OF OXFORD

Makefiles and ROOT

Sean Brisbane

12/12/11

Introduction and purpose

- By the end of today you should know:
  - The basics of the g++ compiler;
  - How to write Makefiles for medium-sized projects;
  - How to build a program incorporating external libraries
    - i.e. ROOT libraries
- I assume you have minimal familiarity with the ROOT interpreter and writing ROOT macros.
- I don’t assume any OOP knowledge

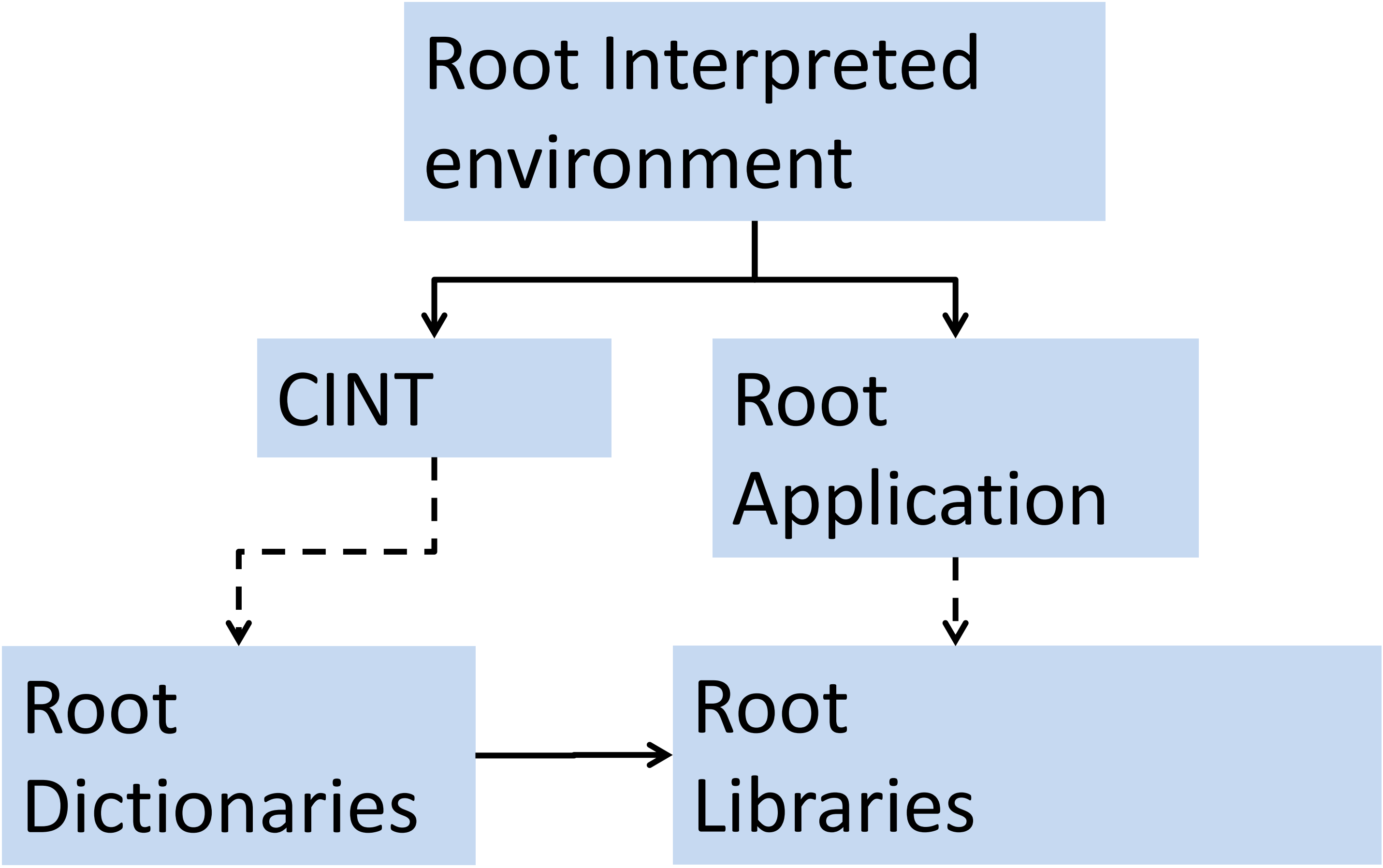
Contents

- ROOT introduction / reminder
- Compiling, linking and dependencies
- Automating the build process with Make
- Your compiled root application
  - TApplication
- Excercises

Section 1

ROOT INTRO/REMINDER

What is‘ROOT’



Root Interpreted environment

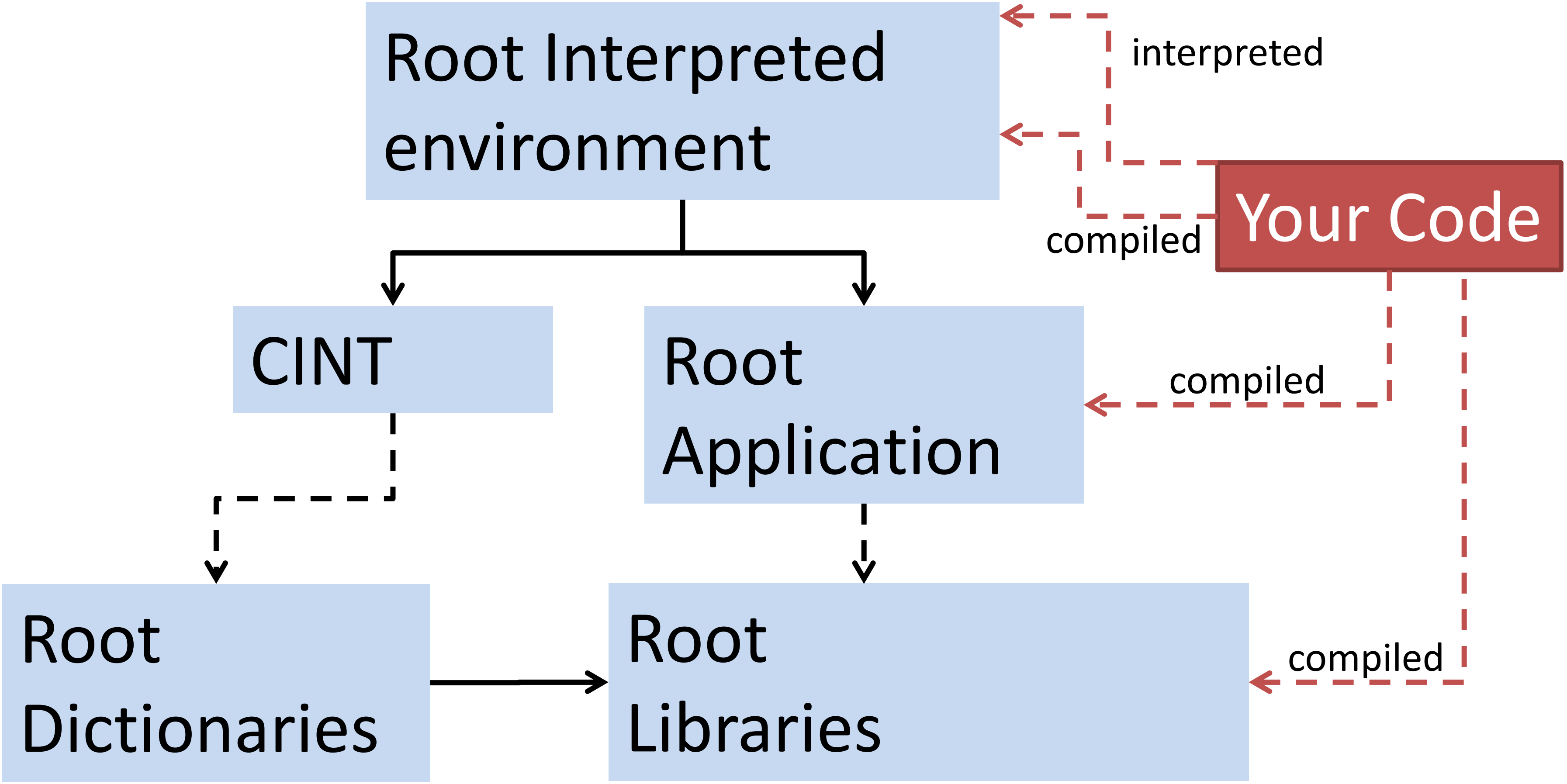
CINT

Root Application

Root Dictionaries

Root Libraries

Ways to use ‘ROOT’



Running code in ROOT

- Load “macro”

```
root [0] .L ${ROOTSYS}/tutorials/hsimple.C
```

- Compile into shared library :

```
root [0] .L ${ROOTSYS}/tutorials/hsimple.C+
```

- Run code:

```
root [1] hsimple()
```

- Compile into shared library and run in one go:

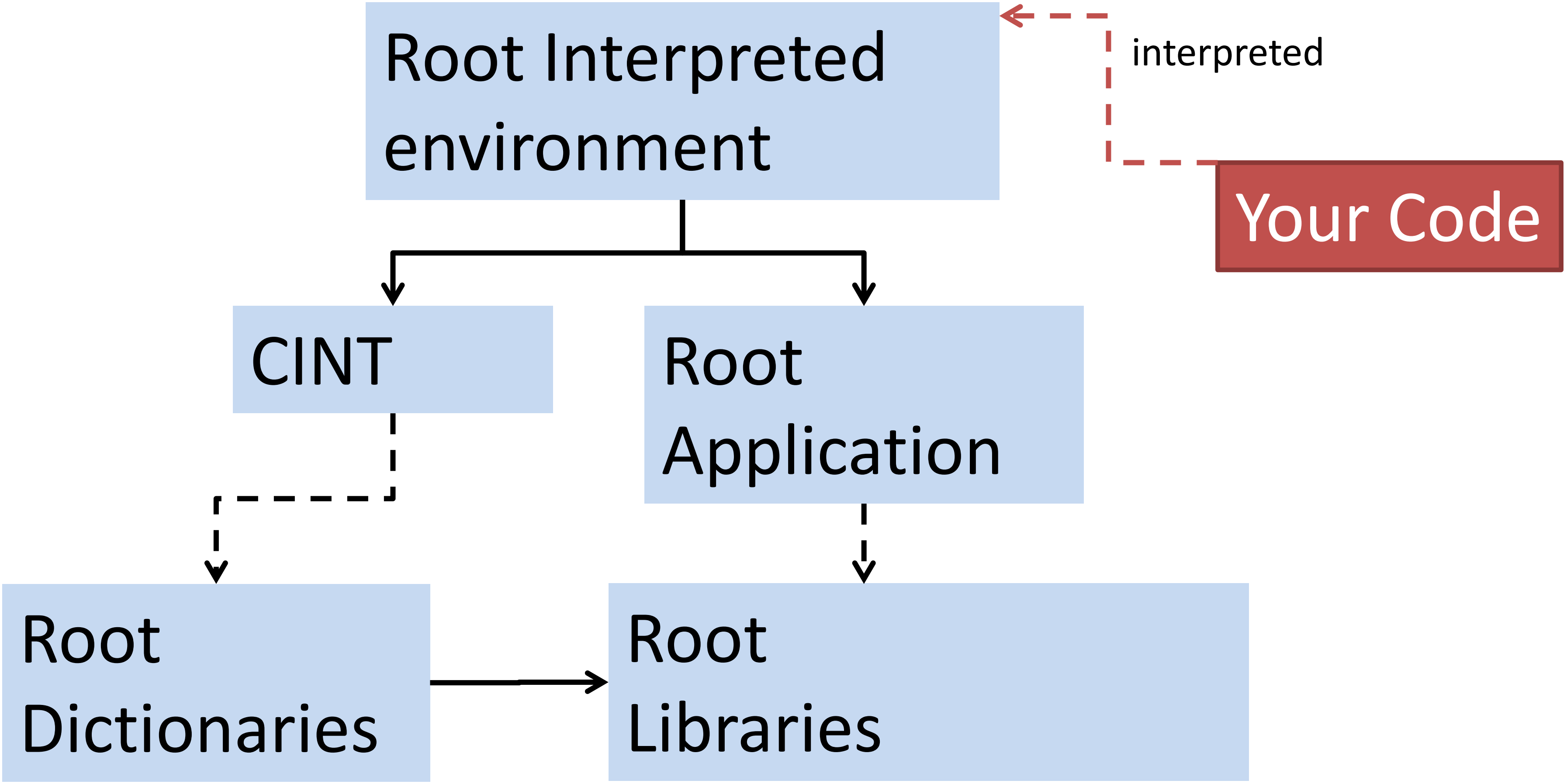
```
root [0] .x ${ROOTSYS}/tutorials/hsimple.C+
```

  Or from command line:

```
> root “${ROOTSYS}/tutorials/hsimple.C+”
```

- Add include path to root (path to additional header files):
  - `root [0] gROOT->ProcessLine(".include ./include")`

Demo (1)
Running an interpreted ROOT macro



Root Interpreted environment

interpreted

Your Code

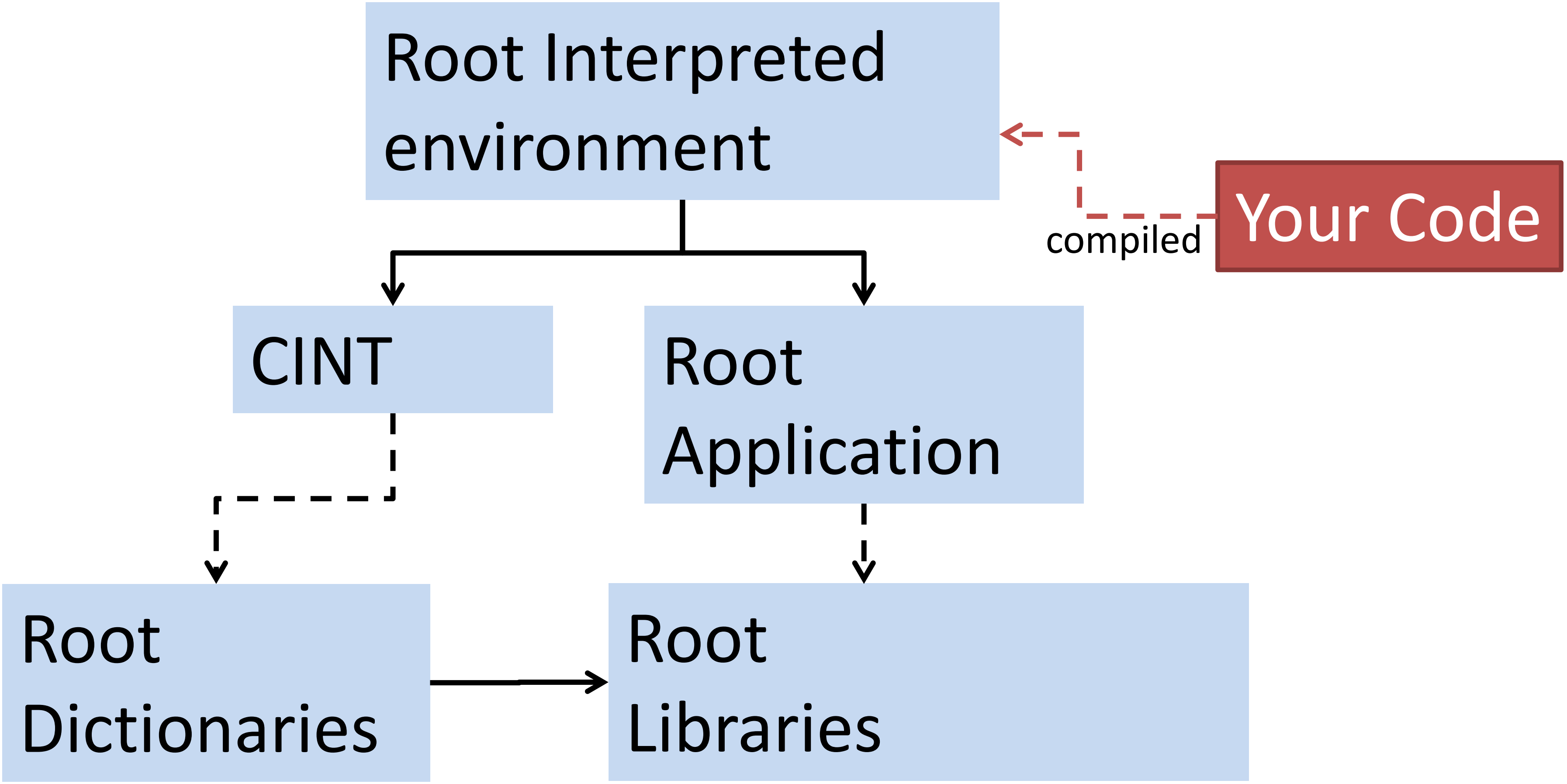
CINT

Root Application

Root Dictionaries

Root Libraries

Demo (2)
Compiling within ROOT



Section 2

COMPILING, LINKING AND DEPENDENCIES

Source code, objects and Libraries

- Header files, .h
  - Forward **declarations** of functions, classes, variables etc. Sould be fairly light, and may be included many times.
  - Is informative, says to the human or the compiler that “Something of this name exists with these properties”
- Source code .C, .cpp, .cxx
  - Usually contains the **definition** of one class or the definition of a few related functions.
  - Implementation of your code.
- Compilation
  - Code is compiled in separate chunks and stitched together at the end;
  - Object files (.o) usually one source file compiled into machine code.
- Libraries and linking
  - A collection of one or more objects
  - Static libraries (libmycode.a) can are compiled directly into your executable
    - Large but portable executable, hard to upgrade.
  - Dynamic libraries (libmycode.so) are picked up at load time (or runtime)
    - ‘Linking‘ is performed to allow your program to know which library contains the implementation for each symbol.
    - Small executable, modularity and reusability. Requires the shared libraries to be installed on the systems.
- A program or executable is basically an object file containing a main function linked to a number of libraries.

Compilation and linking with g++

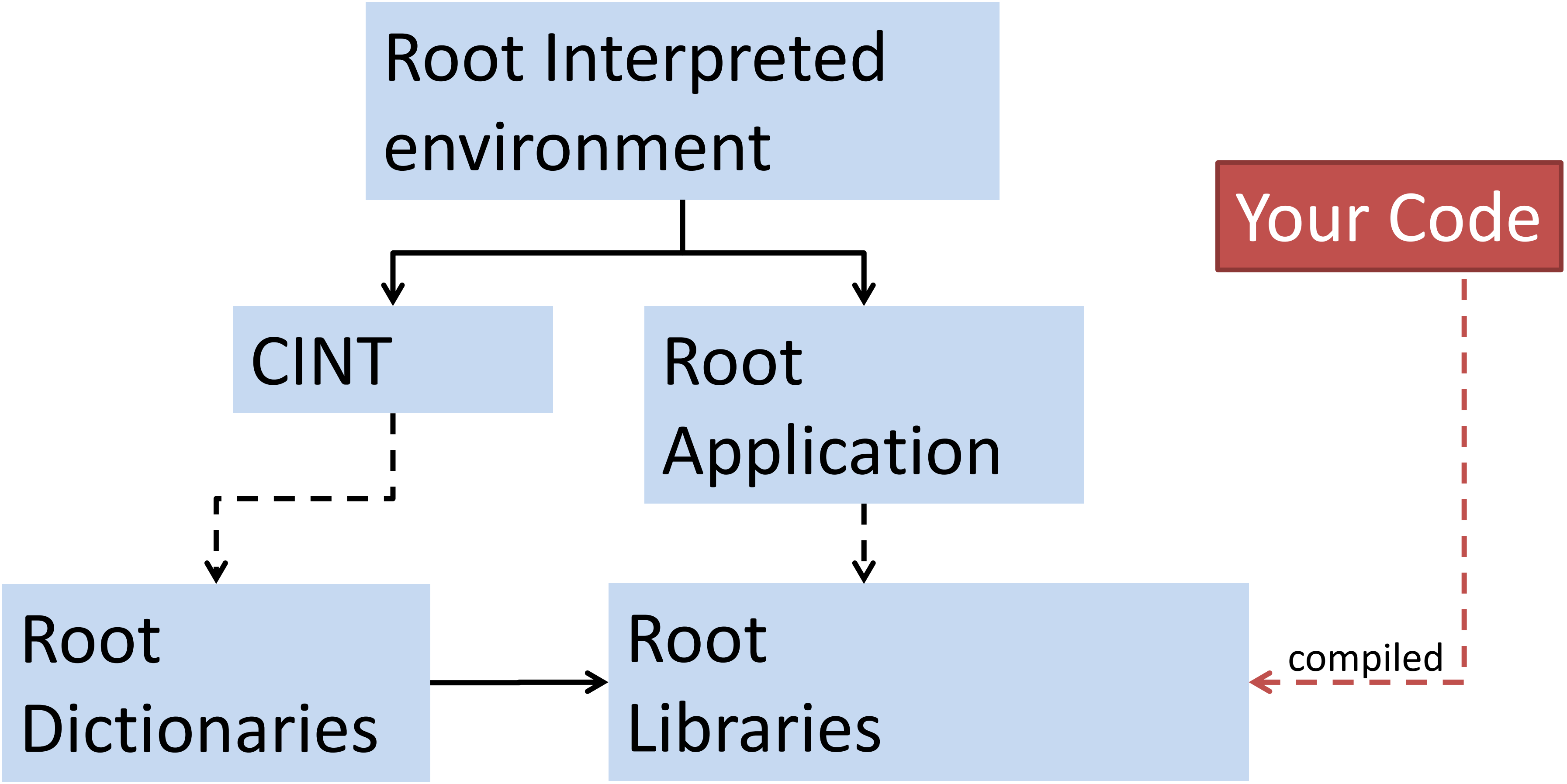
- Object:

  -
    ```
    g++ -I$ROOTSYS/include -fPic -Wall -c hsimple.C -o ./hsimple.o
    ```
  - -c : Do not link to shared libraries
  - -o : specify the output file
  - -Wall: switch on all compiler warnings
  - -fPic : (position independent code) is required for objects destined for shared libraries
  - -Idir : Add directory dir to the list of directories to be searched include files.
- Shared Library:

  -
    ```
    g++ -shared hsimple.C  -o ./libhsimple.so
    ```
- Executable from object:

  -
    ```
    g++ -Wall -L$ROOTSYS/lib mainSimple1.cxx -lCore -lHist -lCint -lRIO -lTree -lGpad  hsimple.o -o main
    ```
  - -Ldir: Add directory dir to the list of directories to be searched for libraries.
  - -l[libname] Link with this library, to be found on the search path(s) specified with -L
- ./main
  - Run Executable

Demo (3c)
Compiling outside of ROOT



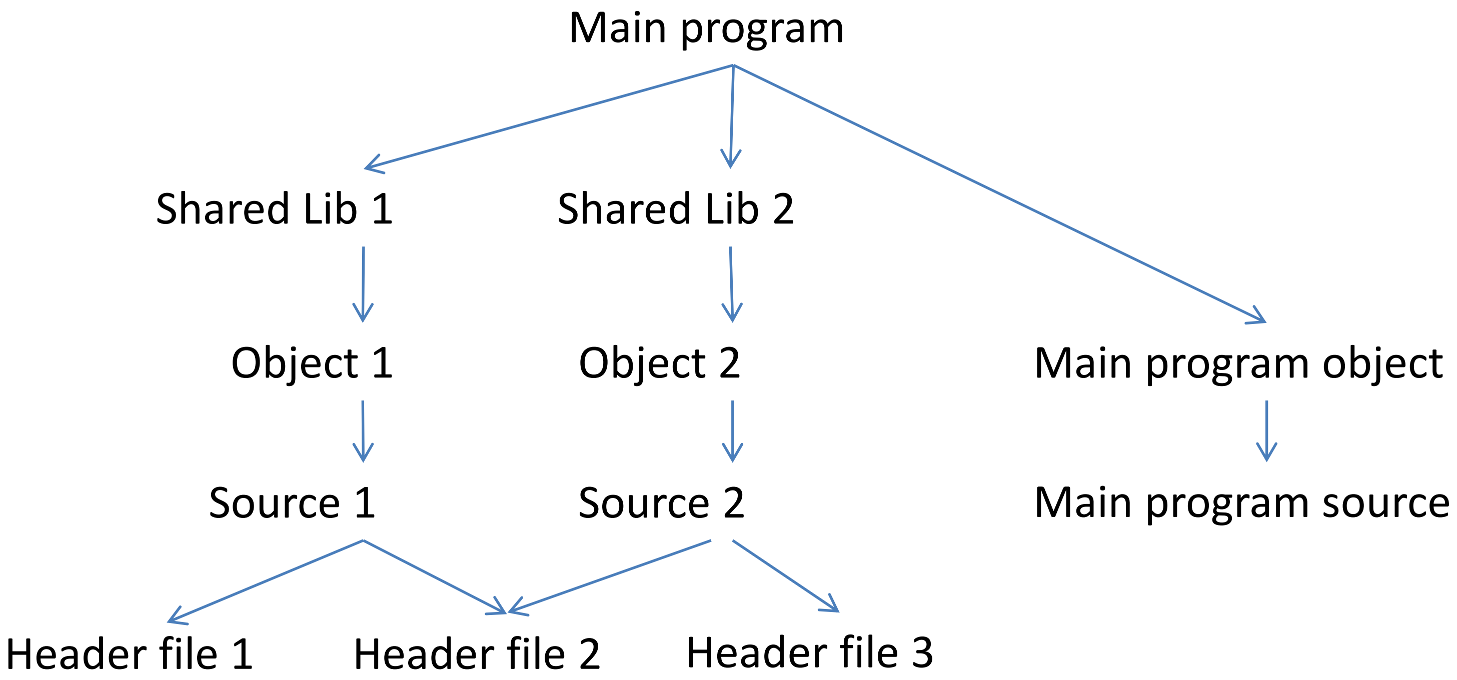
In your own time look at demo 3a and 3b, which introduce the gdb debugger

Dependencies

- There are a lot of interrelated files which go to make up a c++ program.
- Object files rely on a large number source files (.cpp and .h)
  - Re-build when changes are made
- When the .o file changes, re-build any files that depend on this
- Modularity of libraries is important in large programs for build times

Dependency tree (1)

- Main program made up of three objects, which depend on a header file.



Main program

Shared Lib 1

Shared Lib 2

Object 1

Object 2

Main program object

Source 1

Source 2

Main program source

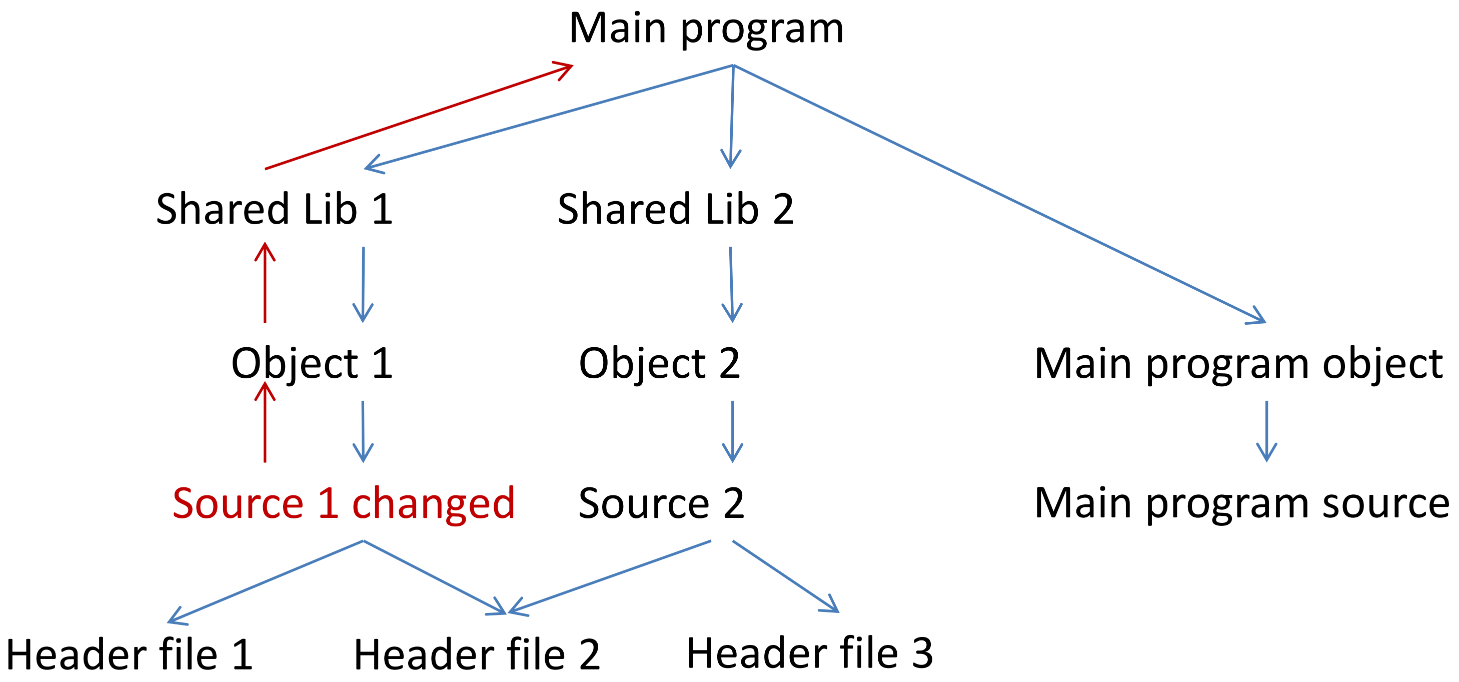
Header file 1

Header file 2

Header file 3

Dependency tree (2)

- One file changes, only re-build those that require it.



Main program

Shared Lib 1

Shared Lib 2

Object 1

Object 2

Main program object

Source 1 changed

Source 2

Main program source

Header file 1

Header file 2

Header file 3

Section 3

MAKE

Make

- Make automates the build process
- Specify how to build a given file type
- Resolve file dependencies
  - Rebuild target is source is more recent
- Not limited to c++ programs
  - Use to automate latex build of thesis
- Make and Makefiles alone are versatile enough for most mid-sized programs
- A target can recursively depend on a source file that is itself a target of another rule

First Makefile (1)

- By default, the ‘make’ tool looks in the local directory for files named Makefile
- The core component of Makefiles is the 'rule', which takes the form:

```
target:  dependancy
 #[TAB]          line to make target
```

- The first target defined in the makefile is the default target
- It is possible to build other targets by typing :

```
> make -f [makefilename] [targetname]
```

- The following rule says that the target main must be rebuilt if depend.o changes. The command below then says how to make it:

```
main : depend.o
        g++ depend.o -o mainmain4
```

First Makefile (2)

- It is possible to use ${} or $() to expand shell environment variables, but in makefiles, they MUST be enclosed in parenthesis of some kind.
- It is also possible to define variables within the Makefile:

```
MYVAR = foo
MYVAR += bar
```

- And write a rule in the Makefile to print these :

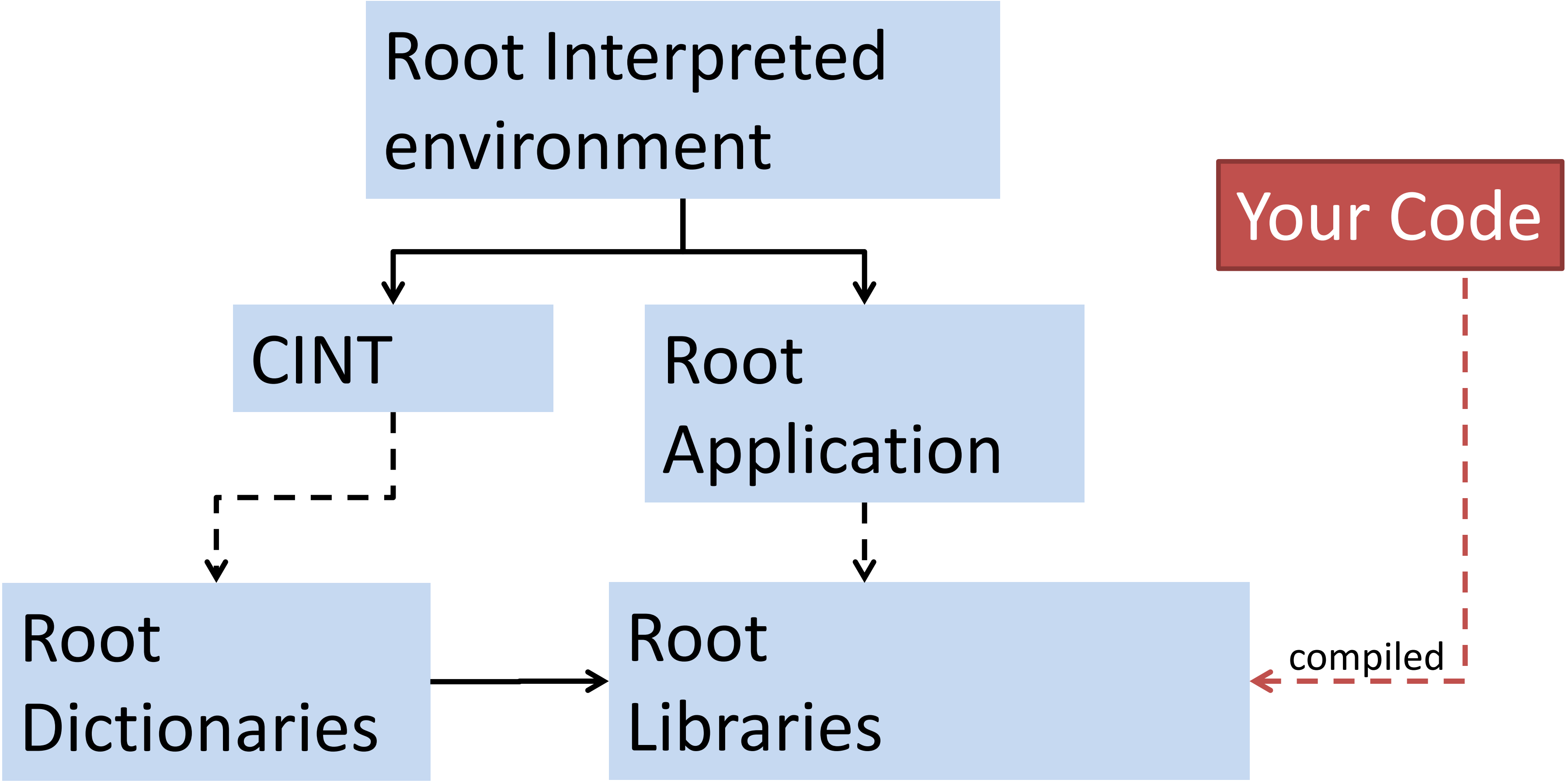
```
foobar:
        echo $(MYVAR) $(MYVAR1)
```

- Now, on the command line type

```
> make foobar
```

- Use ':=' to force make to evaluate the variable immediately, the default is to evaluate it when it is used.
- The convention is to stick to ${} for shell variables and $() for those defined in the Makefile.

Demo (4a)
Automating compilation (Makefiles)



Root Interpreted environment

CINT

Root Application

Your Code

Root Dictionaries

Root Libraries

compiled

Adding local and ROOT shared libraries

Creating Shared Libraries:

- A shared library is created with the ‘shared’ g++ flag from objects compiled with the ‘fPic’ flag:
- libhsimple4.so: hsimple4.o

```
>    g++ -shared hsimple4.o -o libhsimple4.so
```

- Remember to set your LD_LIBRARY_PATH to the current directory

```
> export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./
```

- Later, we use the rpath linker command to write the search path into the executable.

Adding root Libraries:

- Root provides the ‘root-config’ tool, this helps:
  - Setup include paths

```
> root-config --cflags
```

  - Setup library paths and a list of commonly used libraries.

```
> root-config --glibs
```

Adding helper (phony) targets

- The target ‘all’ ensures that the rules for each of the ‘end products’ i.e. the executable and shared libraries are called:

```
    all: $(ALLLIBS) $(ALLEXES)
```

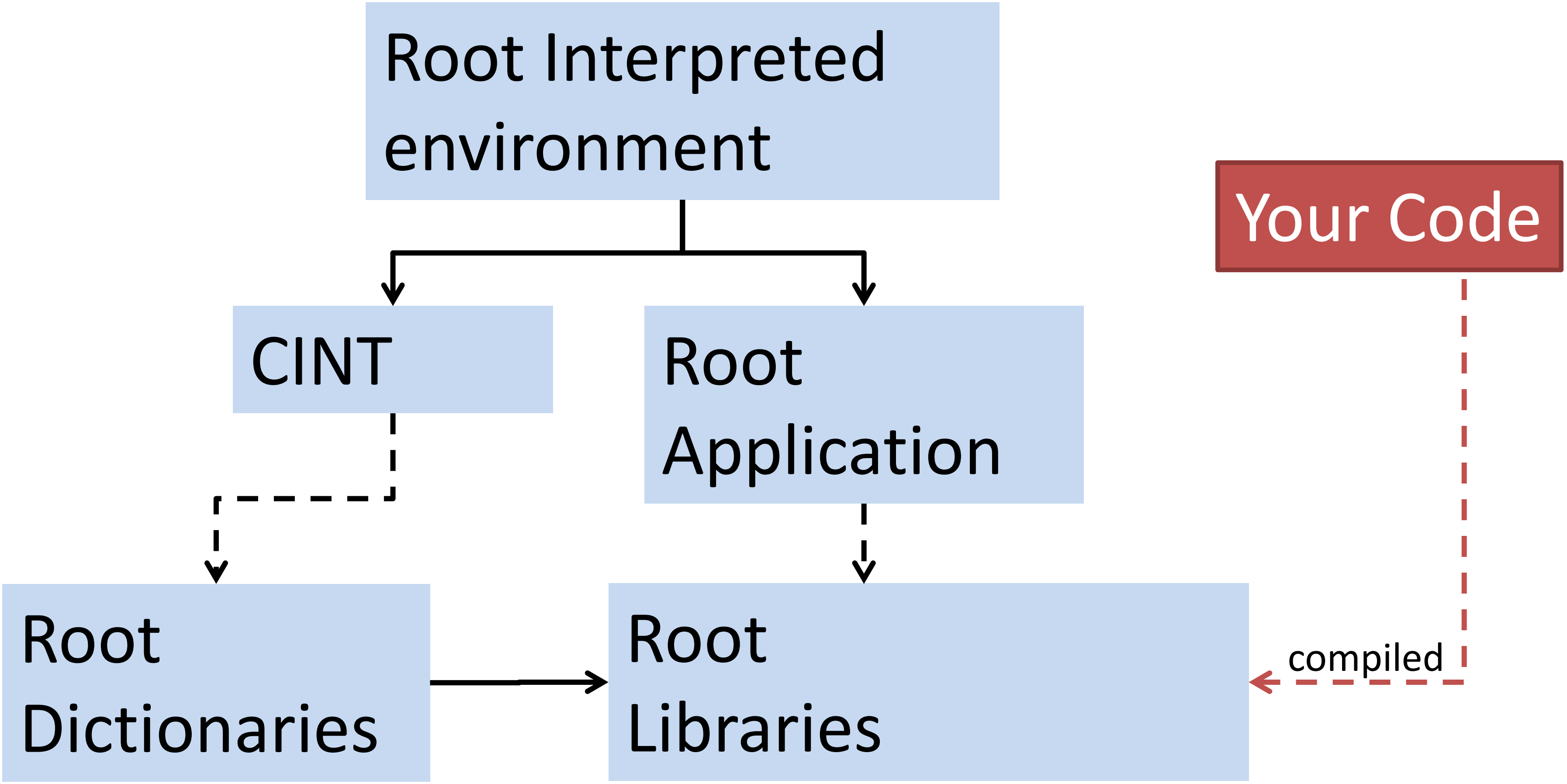
- The target ‘clean’ is set to remove all auto-generated files, useful if a re-compile is needed

```
    clean:
        $(RM) $(ALLLIBS) $(ALLEXES)
$(ALLOBJS) *.d
```

- Add these to a list of .PHONY special targets, since they do not generate files.

```
    .PHONY: all clean
```

Demo (4b)
Shared libraries and phony targets



Root Interpreted environment

CINT

Root Application

Root Dictionaries

Root Libraries

Your Code

compiled

Shortcuts and automatic build rules

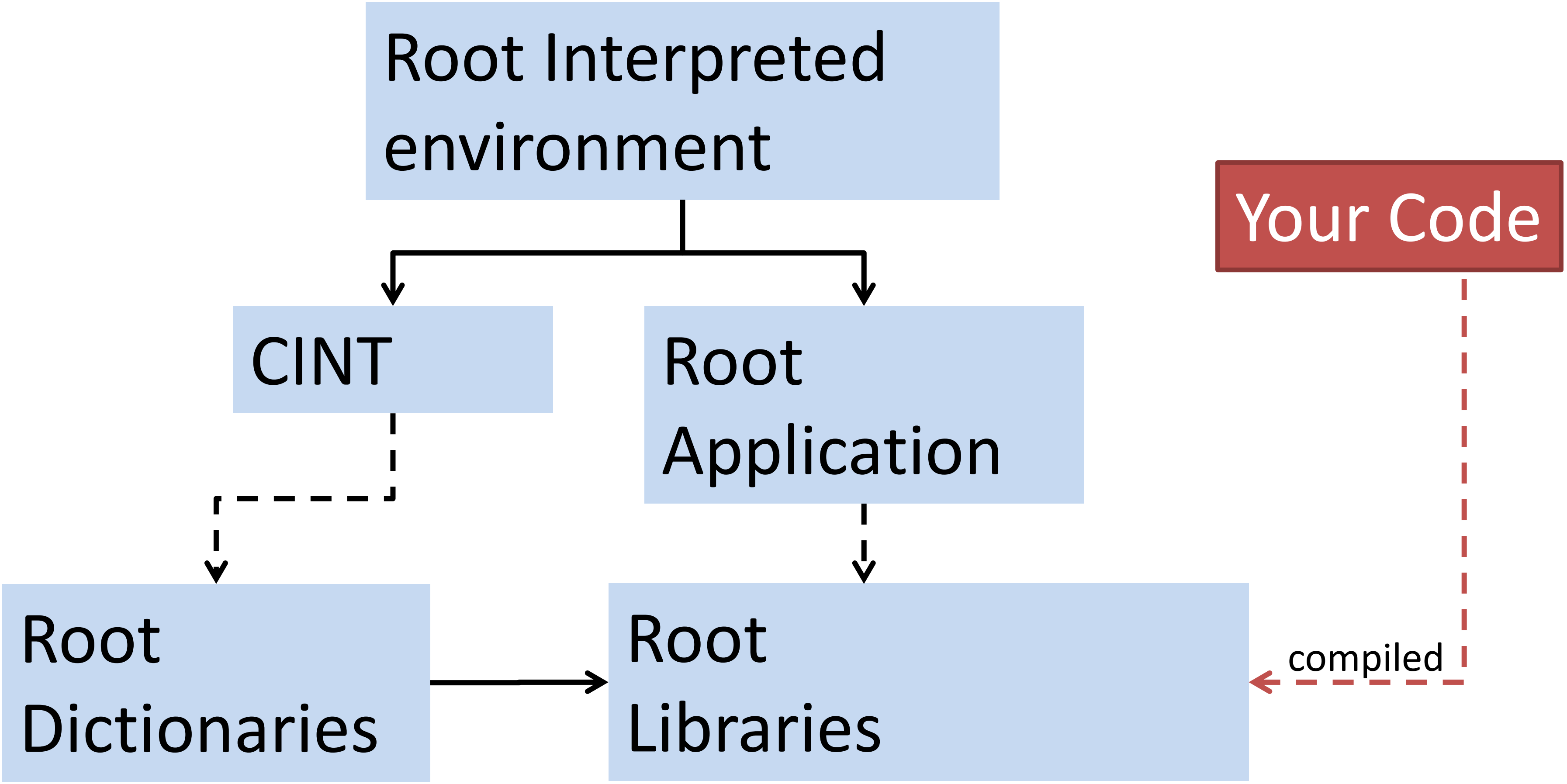
- Make defines a number of helpful shortcuts:
  - $@ : shortcut for the 'target';
  - $< : shortcut for the first dependency;
  - $^ : shortcut for all dependencies;
  - % : signifies string substitution.
- Putting it together into an automatic build rule:

```
%.o: %.cxx
        $(CXX) $(CXXFLAGS) -c $< -o $@
```

- If a file 'foo.o' is required by another rule, make looks for 'foo.cxx' and runs the command:

```
g++ $(CXXFLAGS) -c foo.cxx -o foo.o
```

Demo (4c)
Automatic rules and rpath



Header Dependencies

- When there is a 1:1 mapping between source files and .o files, the automatic build rules rules work well.
- Your object files however in general depend on a number of header files.
- We don’t want to pass our header files directly to the build command.
- Resolution :
  - We specify our header dependencies separately

```
Target   : dep1 dep2
Target   : dep3
    g++ $^ -o target
```

Expands to :

```
    g++ dep3 -o target
```

Advanced topic: Automatic dependency generation

- Specifying header files like this is duplicating work.
  - We have already written this in our source code in `#include “header.h”` statements
- g++ can generate a list of these for us* and place them into a Separate dependency files (with extension ‘.d’) if we pass g++ the `–MD` flag.
- We then include these dependency files in our Makefile with the `“-include”` directive

*You may see other utilities used such as ‘makedepend’

Section 4

MISC

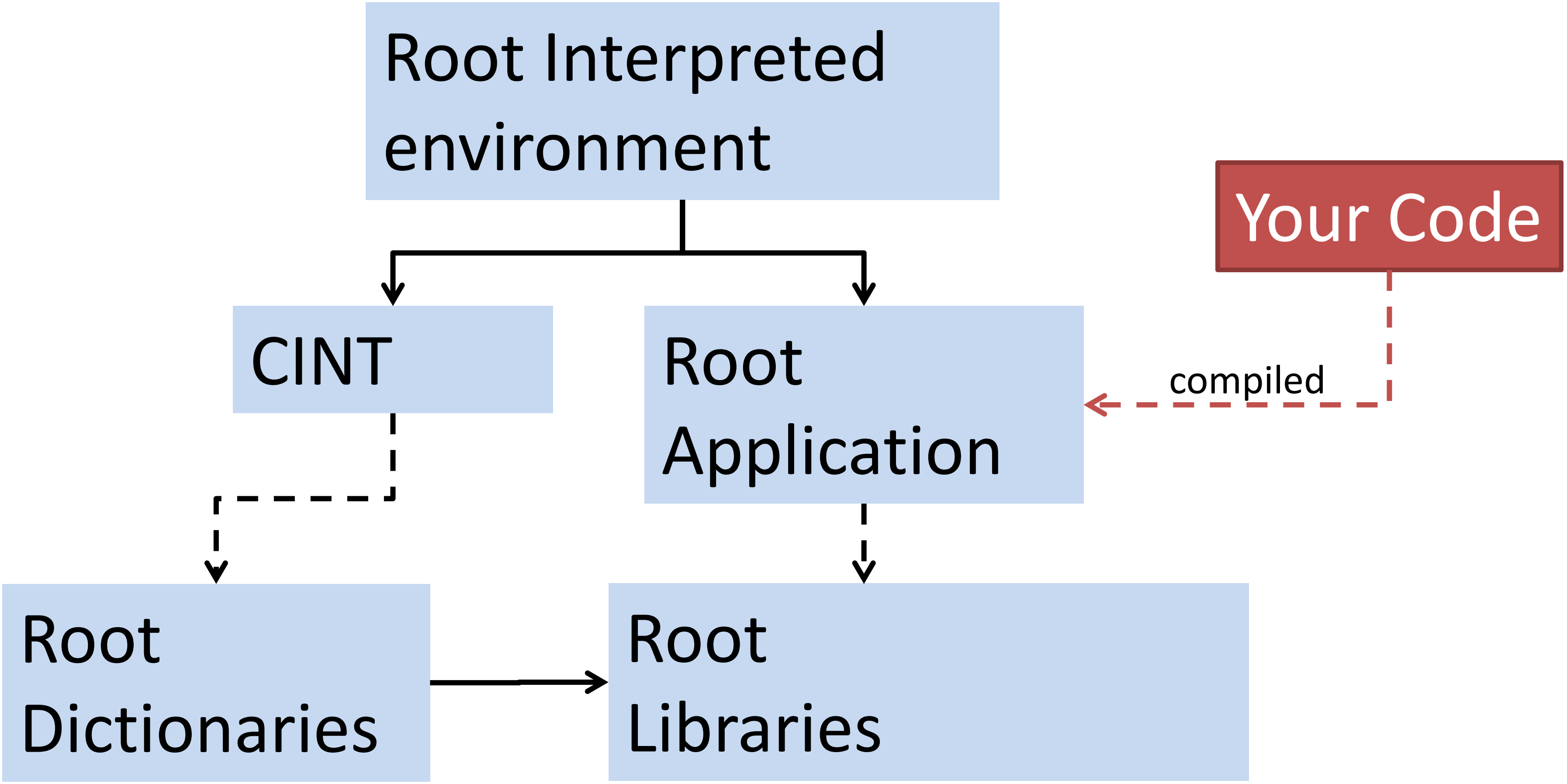
Graphics - TApplication

- So far, our canvases and histograms have not been displayed.
- However, canvases can still be written to file for later viewing:

```
Can->Print(“myHist.eps”,”eps”)
```

- The TApplication ROOT class provides the event loop handling required for graphics.
- If you want visuals, demo5 gives the boiler plate code in rootApp.cxx and extends this in rootAppThreaded.cxx.

Demo (5)
Visual feedback



Root Interpreted environment

CINT

Root Application

Your Code

compiled

Root Dictionaries

Root Libraries

Section 4

EXCERCISES

Your working environment

- Go to the teaching labs on level 2
- Log in to Macintosh
  - Notes beside you. Please fill out the tear off slip. Your Mac login is guest[N]
- Login to pplxint6:
  - ssh pplxint6 –l teaching[N]

  -
    ```
    User : teaching[N]
    ```

  -
    ```
    Password : teach115btU
    ```
- change your password

  -
    ```
    > yppasswd
    ```
- Start the graphical desktop

  -
    ```
    > startkde
    ```
- When loaded, right click and open a konsole
- Setup the root environment and check root loads

  -
    ```
    > source /system/SL5/cern/root/x86_64/OxfordSetup-current-pro.sh
    ```

  -
    ```
    > root -l
    ```
- Quit root

  -
    ```
    root [0] .q
    ```

Getting the exercises and help

- The comments in the source code and Makefiles themselves make up the documentation. This is available at:
  - www-pnp.physics.ox.ac.uk/~brisbane/Teaching/Makefiles/MakefileTutorial.tgz
  - When you are logged in to pplxint6 as a teaching account, open a terminal and:

```
> ./getExcercises.sh
```

- Further info/material can be found at :
  - Internal
    - www-pnp.physics.ox.ac.uk/~west/intro_manual/node105.html
  - External, basic
    - http://mrbook.org/tutorials/make/
  - External, advanced
    - http://www.cs.wfu.edu/~burg/Courses/Fall99/CSC112/course-materials/makefilesHemler.html

Format

- Each exercise is self contained.
- In exercises/ex1a e.t.c. are one or more Makefiles and a README.
- The README is the place to start
  - Contains overall aims for the exercise and instructions.
  - The Makefile also contains useful instructions and comments
- Ex0, Ex1a-d are purely on Makefiles
- Ex 2, 3 &4 include the use of ROOT

ROOT basics

- ROOT is both a useful interpreter and a collection of reusable libraries

- Run a tutorial or script:

  -
    ```
    > root ${ROOTSYS}/tutorials/hsimple.C
    ```
- Open a root file and browse it’s contents

  -
    ```
    > root hsimple.root
    ```

  -
    ```
    root [0] TBrowser cBrowser
    ```

- [Force Re-]Compile a tutorial using roots default compiler (ACLICK):

  -
    ```
    > root ${ROOTSYS}/tutorials/hsimple.C+[+]
    ```

- Documentation:
  - http://root.cern.ch/drupal/content/documentation
- Where to get ideas and examples:

  -
    ```
    > ls ${ROOTSYS}/tutorials
    ```