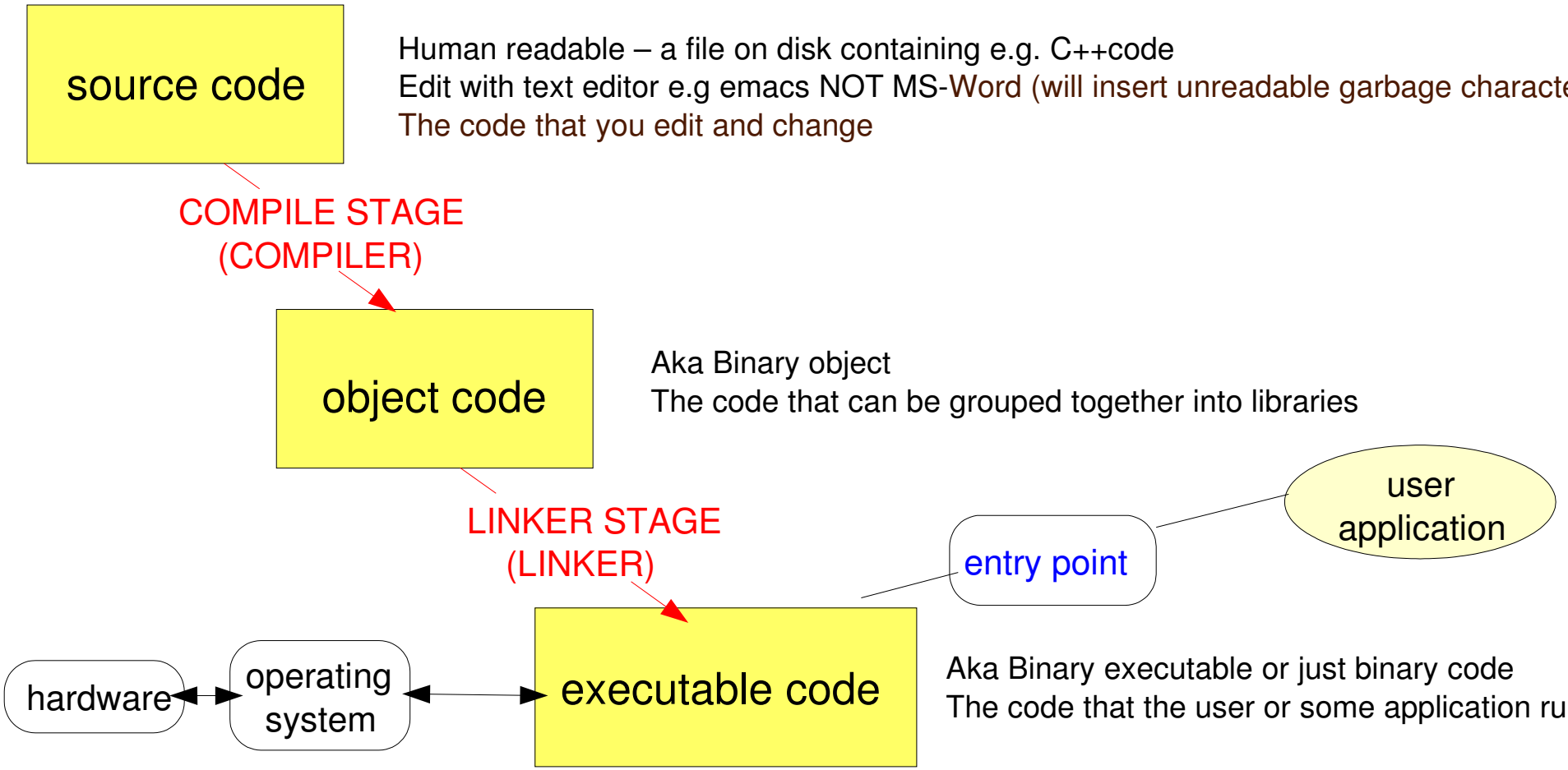
Building Programs : Compiling, Linking, Running

There are two types of code run:

- interpreted code e.g. script files, perl, python
  - run as-is in sequential order (not explicitly compiled by a user)
- compiled (and then linked) code e.g. C++, fortran, java (ish)
  - this generally produces faster code but is more complex



- On Windows – compile, link, run steps are largely hidden by a click
- On Linux – you will explicitly (1000s of times) compile, link and run code

- Object & executable code is operating system and hardware dependent
    - an executable produced on Linux will not run on a Mac or Windows PC

- To run on multiple machines (aka the Grid) requires either :
    - the code is compiled/linked before running on each machine
        - you need to know how to do this for each type of machine and embed this in your job
    - you run only on identical machines
        - then you can simply copy the executable code

- a computing grid of mixed operating system (OS) PCs is extremely difficult
    – at some level Java gets round this by producing identical executable code and making all machines look the same by having a Java Virtual machine(JVM) that runs the executable code such that executable code can be copied between machines; but then each machine has to have the virtual machine software installed (which **IS** different on each machine type).

- The types of files : source, object, executable are generally distinguished by differing file extensions although this is not mandatory – just sane.

- Source file extension depends on language
  - C++ : file.cc or file.cpp or file.cxx (and header files file.hh)
  - C : file.c (and header files file.h)
  - fortran : file.f or file.for or file.f77 or file.F (and header files file.inc)
  - java : file.java

- Object file extension:
  - C++/C/Fortran : file.o
  - java : file.class

- One source file is compiled to one object file
- Many object files can be grouped together into a single library file

- Library file extension:
  - Static Libraries : libName.a
  - Dynamic/Shared Object Libraries : libName.so, libName.dll (M$ Windows)
  - Java library : Name.jar

- At least one object file plus any amount of other object files and libraries are linked to produce one executable file generally with the file extension .exe
  (except java where there is no link step; the jar or class files are already executable and executed by the JVM)

- So a single executable file can come from many (thousands) of source files.

- In the following I will do everything with reference to C++.
  - Java is slightly different and much simpler !

- I will consider four cases:
  1. A single C++ file with no user-defined header files and no classes
  2. A main C++ file linked with a C++ class defined in a separate source & header file
  3. As above but with two C++ classes merged into a single library file.
  4. As above but using some useful 3rd party C++ code

- The syntax / explicit understanding of the C++ is not important – it is the structure and the type of files and the commands we use to create, compile and link them that are important.

Single C++ file with no user defined header files and no classes

- create a file using the emacs editor

```
> emacs main.cc &
```

  - using emacs type this into the file and save it (Ctrl-x s)

- compile the code and create an object file

```
> g++ -c main.cc
```

  `-c` means compile, there are other options as well (`man g++` will list them)

  - we say we are “using the g++ compiler” - there are others....

built in C++ definitions (for writing to screen) i.e. cout and endl

tells it to use the “standard” versions of cout and endl

entry point

```
#include <iostream>
using namespace std;

int main(int nargc,char argc[]) {

    cout << "The program is starting ... " << endl;
   for (int i = 1 ; i < 11; i++) {
       cout << i << endl;
   }

   return 0;
}
```

```
> g++ -c main.cc
```

- the default is to create an object file with the same name but file extension .o; viz `main.o`
- one could change this by specifying the -o option e.g.

```
> g++ -c main.cc -o mymain.obj
> ls -all *main*
```

- will list all files containing the word `main`

- behind the scenes g++ invokes the command `cpp` (c-pre processor)
  - this interprets the `#include` directives (and others) and effectively copies the contents of the file `/usr/include/g++-3/iostream.h` into `main.cc` before it is passed into the compile phase.

- if you have a syntax error in your code e.g written `Cout` instead of `cout` you will get an error message e.g

```
main.cc: In function `int main(int, char*)':
main.cc:6: `Cout' undeclared (first use this function)
main.cc:6: (Each undeclared identifier is reported only once for each function
it appears in.)
```

- and the `main.o` file will not get produced until you fix the problem and re-compile.

- if everything is OK – nothing happens – there are no errors
- if you get lots of errors you could consider putting them into a file for reference/printing e.g.

```
> g++ -c main.cc >& compile_errors.txt
> more compile_errors.txt
```

- Some compilers are more pedantic than others – so for mission critical code it can be useful to use more than one.
- It is also possible to tailor the level of error/warning – some mistakes e.g. defining a variable that you don't actually use do not matter but the compiler can warn you.

- Occasionally things are added into C++ which a compiler doesn't support e.g. version 3 of g++ has many features not in version 2.

The default on the HEP systems is version 2 (which is fine for everything for now)

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

- will print the version you are using

For information to use version 3 you would edit your `.bashrc` file as follows:

```
export GCC_HOME=/usr/local/gcc-alt-3.2.3
export PATH=$GCC_HOME/bin:$PATH
export LD_LIBRARY_PATH=$GCC_HOME/lib:$LD_LIBRARY_PATH
```

But then you have to be careful and ensure that any other 3rd part code you are using has also been compiled with the same version (c.f my comments on the grid)

Simple Linking -

```
> g++ main.o -o main.exe
```
- produces the executable file `main.exe` from the one object file
- behind the scenes this passes `main.o` and some `g++` system object files to the linker "`ld`"
  - generally it is far easier to use "`g++`" as the linker command (aswell as the compiler) rather than running "`ld`"

- since `g++` acts as compiler and linker; then for simple cases e.g. A single c++ file which contains a "main" method/entry point one can actually compile and link in one step:

```
> g++ main.cc -o main.exe
```
behind the scenes this is doing the two commands:
```
> g++ -c main.cc -o main.o
> g++ main.o -o main.exe
```

Entry Points -

- a g++ link step will fail unless one of the object files (compiled from a .cc file) contains a `main` method – you will get this error:

```
/usr/lib/crt1.o: In function `_start':
/usr/lib/crt1.o(.text+0x18): undefined reference to `main'
  collect2: ld returned 1 exit status
```

- when you run the executable the operating system "looks for" the main method – it cannot actually run without one.

- if another application is running your code; then invariably you will not link the code – you will build your code into a library and the application will call a specified method which you must have defined in your code.

- it is not magic – the computer tries to look for a “main” entry point or one as specified in the application (the application itself will have a “main” entry point) since this will be run first and that will invoke your code

- so if your code or the application your are using doesn't have a “main” entry point it will not run. The same is also true of C, Fortran and Java.

Executing the simple example

```
g++ main.o -o main.exe
```

- produces an executable file `main.exe` in the current directory

- You run this simply by issuing the command:

```
> ./main.exe
```

- The ./ tells the operating system to look in the current directory (just typing main.exe will not generally work unless you have your $PATH variable configured to look in the current directory i.e. “`./`” - it is dumb – you have to tell the system **EXACTLY** where main.exe is)

```
> g++ main.o -o someOtherDirectory/test.exe
> someOtherDirectory/test.exe
```

- will also work

```
> ./main.exe
```

- should then simply print to the screen the results of the “cout” commands i.e

```
The program is starting ...
1
2
3
4
5
6
7
8
9
10
```

- you could save the output into a file as follows:

```
> ./main.exe > main.output
```

Defining a simple C++ class and linking it and running it

- don't worry about the C++ details/syntax

1) Define a header file which defines the class

```
> emacs simple1.hh &
```

2) Create the source code for this class:

```
> emacs simple1.cc &
```

```
#ifndef SIMPLE1_HH
#define SIMPLE1_HH

class simple1 {

public:
   simple1();
  ~simple1();

  void printHello();

};


#endif
```

```
#include "simple1.hh"
#include <iostream>

using namespace std;

simple1::simple1() {
  cout << "simple1 object is being created..." << endl;
}

simple1::~simple1() {
  cout << "simple1 object is being destroyed..." << endl;
}

void simple1::printHello() {
  cout << "Hello ..." << endl;
}
```

Include Paths

- we compile this in the same way as before

```
> g++ -c -I ./ simple1.cc
```

- the only difference is the `-I ./` directive; this tells it in which directories to search for files prefixed with #include directives.

- the `-I ./` directive sets the "include path" to look for header files. It can be many directories

- one could explicitly put the full filename in for the simple1.hh file in the simple1.cc source code e.g.

```
#include "/home/markl/test_code/include/simple1.hh"
```

- but this is not very portable (e.g. if you give the code to someone else with a different username).

- far better instead to have:

```
> g++ -c -I /home/markl/test_code/includes simple1.cc
```

- now create a simple main program

```
> emacs simpleMain.cc &
```

```
#include <iostream>
#include "simple1.hh"
using namespace std;

int main(int nargc,char argc[]) {

  cout << "The program is starting ... " << endl;
  simple1 s;
  s.printHello();

  simple1 *sp = new simple1();
  sp->printHello();
  delete sp;

  return 0;
}
```

- compile :

```
> g++ -c -I ./ simpleMain.cc
> ls -all *.o
```

- you should now have `simpleMain.o` and `simple1.o`

- we now want to “link” simpleMain.o and simple1.o together into one executable file:

```
> g++ simpleMain.o simple1.o -o simpleMain.exe
```

- you can link together as many .o files as you want (and they can be in different directories) BUT one (and only one) must have been compiled from a source file that had a “main” entry point.

- and run this executable :

```
> ./simpleMain.exe

  The program is starting ...
  simple1 object is being created...
  Hello ...
  simple1 object is being created...
  Hello ...
  simple1 object is being destroyed...
  simple1 object is being destroyed...
```

You can do this indefinitely and create many classes.

Generally each class you define should be in a separate .cc file and have an associated .hh file but you can have many classes in one .cc file and may definition in one .hh file. But it is not good practise.

(Java for instance has no .hh files just a .java file and it is one class per .java file)

- If you create 3 more classes : simpleN.hh , simpleN.cc (1 < N < 5)

```
> cp simple1.cc simple2.cc ; cp simple1.hh simple2.hh
```

and replace all “1” in simple2 files with “2” using emacs editor.

- Then you could compile and link as follows:

```
> g++ -c -I ./ simple1.cc
> g++ -c -I ./ simple2.cc
> g++ -c -I ./ simple3.cc
> g++ -c -I ./ simple4.cc
```

- or you can do them all at once:

```
> g++ -c -I ./ simple1.cc simple2.cc simple3.cc simple4.cc

> g++ -c -I ./ simpleMain.cc

> g++ simpleMain.o simple1.o simple2.o simple3.o simple4.o -o simpleMain_4.exe
```

If you change any C++ code then you must re-compile and re-link each time in order to get a new executable with the changes. It is dumb – it does not know you have changed code.

Link Errors

- if you do not have an object file containing a main entry point
- if you try and reference a method/function/object that is not included in any of the .o files e.g.

```
#include <iostream>
#include "simple1.hh"
#include "simple2.hh"
using namespace std;

int main(int nargc,char argc[]) {

  cout << "The program is starting ... " << endl;
  simple1 s;
  s.printHello();

  simple2 *sp2 = new simple2();
  delete sp2;

  return 0;
}
```

- compiles OK:

```
> g++ -c -I ./ simple1.cc simple2.cc simpleMain.cc
```

- but in link we forget the simple2.o

```
> g++ simpleMain.o simple1.o -o test.exe
```

```
simpleMain.o: In function `main':
simpleMain.o(.text+0x65): undefined reference to `simple2::simple2[in-charge]()'
simpleMain.o(.text+0xbb): undefined reference to `simple2::~simple2 [in-charge]()'
collect2: ld returned 1 exit status
```

Libraries

- are simply a bundle of .o files conveniently collected into one file in order to simplify the link procedure and to facilitate easier distribution code.
- e.g CDF software ~11,000 .cc/.hh files with 2.5 million lines of code – instead it is organised in ~ 50 libraries.

Two types of library
- **STATIC LIBRARY (.a)**
- **SHARED OBJECT LIBRARY (.so) /DYNAMIC LINK LIBRARY (.dll)**

They are made in a similar way and linked in an identical way but they are used differently at execution time

- When an executable is made from a main .o file and a static library then the resultant executable is entirely defined and it can be copied to an identical machine in isolation and executed.

- When an executable is made from a main .o file and a dynamic library then the resultant executable actually contains pointers to executable code in the shared object library. If code is to be run on another machine then both the executable **AND** the shared object libraries must be copied. The executable file is not self-sufficient/enough and when the executable is run the directories which contain the shared object libraries need to be defined via the environment variable **LD_LIBRARY_PATH**

Often 3rd party software e.g. the ROOT package comes as a bunch of shared object libraries (xxx.so files)

Pros & Cons are:

- static libraries gives a very large executable which can take a long time to copy and the link step can take a long time.
  e.g. CDF software with only static libraries takes several hours to link.
  This is because it has to cross check every piece of code and make sure it is in the library.

- static library gives an executable that is standalone and can be copied to identical machines without the need to copy shared object libraries or define the LD_LIBRARY_PATH variable.

- static library executables start up quicker than shared object library executables since they do not need to check if all code is defined (that is done at the link stage)

- shared object library executables are small and are linked quickly

- shared object library executables need to have the associated shared object library files present and the LD_LIBRARY_PATH defined.

- shared object library executables tend to start slower since the object code is loaded into memory from the shared object libraries to create a “dynamic executable”.

In reality both are used. Many common applications e.g the Linux commands you type at the keyboad and the “root” command rely on shared object libraries being present

e.g. If you deleted the file : `/lib/libc.so.6` then many common Linux command e.g. “`cp`”, “`ls`” would fail to work – they would complain about missing symbols.

Creating Libraries

N.B. The .o file containing the main entry point should be kept as a .o file and not included in a library.

- a static library e.g. from the files simple1.o and simple2.o

```
> ar -r libSimple.a simple1.o simple2.o
```

- list what is in the library

```
> ar -t libSimple.a
```

If you update simpleN.o by recompiling simpleN.cc then you must repeat the “ar -r ...” command before linking the library again.

- a shared object library

```
> g++ -shared simple1.o simple2.o -o libSimpleShared.so
```

- list (sort of) what is in the .so with:

```
> nm linSimpleShared.so
```

The link command including libraries

There are a few variants on this:

1. Explicitly give the fullname of the library including directory.
- static library

```
> g++ simpleMain.o libSimple.a -o simpleMain.exe
```

- shared object library

```
> g++ simpleMain.o libSimpleShared.so -o simpleMain.exe
```

2. Using the -L and -l directives
 - static library

```
> g++ simpleMain.o -L ./ -lSimple -o simpleMain.exe
```

 - shared object library

```
> g++ simpleMain.o -L ./ -lSimpleShared -o simpleMain.exe
```

-L is like -I for the include in the g++ compile step. It tells the linker in which directories to look for libraries either .a or .so files.

The `-lSimple` just says look for a file called `libSimple.a` or `libSimple.so` in the directories specified by the -L directive.
./ : just means the current directory.

You can have many directories in the -L directive and many -l libraries. The -L, -l method is more flexible since it allows a mix of static (.a) and shared-object (.so) by not requiring an explicit filename.

Running the executable

- if it has been linked with only static (.a) libraries then just the usual:

```
> ./simpleMain.exe
```

- if it has been linked with any static (.so) libraries then you need to make sure the environment variable LD_LIBRARY_PATH is set OK.

```
> echo $LD_LIBRARY_PATH
```

– make sure it has the directories containing your .so file. If the .so files are in the current directory it must contain a "./"

- if it hasn't define as follows (you need only do this once per session):

```
> export LD_LIBRARY_PATH=/home/markl/shlibs
or
> export LD_LIBRARY_PATH=./
```

– Then you just run as normal with:

```
> ./simpleMain.exe
```

– If `LD_LIBRARY_PATH` is incorrect or some of the .so files for instance have been deleted inbetween the link step and the execution step then you will get an error e.g.

```
./simpleMain.exe: error while loading shared libraries: libSimpleShared.so:
   cannot open shared object file: No such file or directory
```

Compiling against and linking with someone elses code

- An example here is the ROOT code to generate a random Gaussian number.

- To do this you need:
  - to know where the C++ header files are and what they are called – so you can the appropriate `-I` directive in the compiler
  - the name of the libraries and their location so you can create the appropriate link command and define `LD_LIBRARY_PATH` if any of the 3rd party libraries are shared object libraries.

```
#include <iostream>
#include "TRandom.h"

using namespace std;

int main(int arg,char argc[]) {


  TRandom *ran = new TRandom();
  for (int i = 0; i < 5; i++) {

    double GausNumber = ran->Gaus(0.0,1.0);
    cout << " Gaussian Random # = " << GausNumber << endl;

  }

  delete ran;
}
```

EXTERNAL header file

rootTest.cc

```
# This next line should be in your ~/.bashrc file
> export ROOTSYS=/usr/local/root-pro
> g++ -c -I $ROOTSYS/include rootTest.cc
```

List of root libraries to link to is quite long – so define a simple environment variable
(ideally again this would be in your ~/.bashrc file)

```
> export ROOT_LIBS=-L$ROOTSYS/lib -lCore -lCint -lHist -lGraf -lGraf3d -lGpad
-lTree -lRint -lPostscript -lMatrix -lQuadp -lPhysics -lpthread -lm -ldl -rdynamic
```

Then link:

```
> g++ rootTest.o $ROOT_LIBS -o rootTest.exe
```

Then define the LD_LIBRARY_PATH (this could also be in your ~/.bashrc file)

```
> export LD_LIBRARY_PATH=$ROOTSYS/lib
```

Then run as usual:

```
> ./rootTest.exe

 Gaussian Random # = -1.10228
 Gaussian Random # = 1.20281
 Gaussian Random # = 0.39294
 Gaussian Random # = -0.524181
 Gaussian Random # = -0.594172
```

- a failure to get LD_LIBRARY_PATH right will produce:

```
./rootTest.exe: error while loading shared libraries: libCore.so: cannot open
  shared object file: No such file or directory
```

- typing lots of g++ and knowing when files have changed and export LD_LIBRARY_PATH is a pain

- there is an easier way:

- Makefiles and shell scripts/.bashrc file

– NEXT WEEK

http://www.hep.ucl.ac.uk/homepage.shtml