COMPILING, BUILDING, AND INSTALLING PROGRAMS ON THE CLUSTER
• The process of converting a human-readable file to a machine-readable file.

C program (simple text file written in C programming language)

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Binary executable file (a set of CPU instructions encoded in 0’s and 1’s)

```
0101010100001010101001
0101110101010010100000
0101110101001011000111
0110101010101010101010
1010010101010101011111
0010111101110000111010
1010100111101010111101
0101010001101010101001
```

Sophisticated programs (e.g. a compiler) are used to perform this multi-step conversion.
The Build Process

Not all languages are compiled languages! The process to the left applies to programs written in C, C++, and Fortran.

Human-readable

Source code (e.g. C program)

Preprocessor

Expanded source code

Compiler

Assembly code

Assembler

Object code

Linker

Binary executable

Lower-level language

External libraries

Higher-level language

Machine-readable (i.e. can be executed by CPU)
In C, preprocessor directives begin with the # symbol and are NOT considered C code.

### Preprocessor

- Expands or removes special lines of code prior to compilation.

#### Include statements:

- Copies contents of stdio.h into file.

#### Define statements:

- Replaces all instances of PI within file with 3.1415.

#### Header guards:

- Prevents expanding multiple copies of the same header file by defining a unique “macro” for each header file.
• Converts expanded source code to assembly code.

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

• Assembly-level instructions are specific to a processor’s Instruction Set Architecture (ISA).

• Example ISAs are x86, x86_64, and ARM. Most machines in HPC today support x86_64.

Portability is an issue with compiled languages since assembly language contains instructions that are specific to a CPU’s architecture.
ASSEMBLER AND LINKER

• **Assembler**: converts assembly code to object code.

• Object code is in a binary format but cannot be executed by a computer’s OS.
• External libraries are often distributed as shared object files that are object code.
  • Hides specific implementation since these files are not human readable.
  • No need to be recompiled for each application that uses the library.
  • Stored efficiently in binary format.

• **Linker**: stitches together all object files (including any external libraries) into the final binary executable file.

• Many applications often contain multiple source files, each of which need to be included in the final executable binary.
• The job of the linker is to combine all these object files together into a final executable binary (a.k.a. “executable” or “binary”) that can be run.
IMPORTANT NOTE: In practice, the steps performed by the preprocessor, compiler, assembler, and linker are generally obscured from the user into a single step using (in Linux) a single command. In the next several slides, we will refer to this single command as a compiler, but note that we’re actually talking about a tool that is a preprocessor + compiler + assembler + linker.

• GCC: GNU Compiler Collection
  • Free and open source
  • Most widely used set of compilers in Linux
  • C compiler: gcc
  • C++ compiler: g++
  • Fortran compiler: gfortran

• Intel Compiler Suite
  • Licensed and closed source, but ACCRE purchases a license
  • Often produces faster binaries than GCC
  • Occasionally more difficult to build code due to lack of community testing
  • C compiler: icc
  • C++ compiler: icpc
  • Fortran compiler: ifort
**Using Compilers on the Cluster (2/3)**

```bash
gcc hello.c
```

- Builds C program with the GCC C compiler.
- Produces a binary called `a.out` that can be run by typing `./a.out`

```bash
gcc -o hello hello.c
```

- Produces a binary called `hello` that can be run by typing `./hello`

---

Error messages result when the build process fails. The compiler should provide details about why the build failed.

Warning messages occur when a program’s syntax is not 100% clear to the compiler, but it makes an assumption and continues the build process.

```bash
$ gcc -o hello hello.c
hello.c: In function ‘main’:
hello.c:33:4: error: expected ‘;’ before ‘return’
    return 0;

$ gcc -o hello hello.c
hello.c: In function ‘main’:
hello.c:23:4: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
    printf("Hello world!\n");

hello.c:23:4: warning: incompatible implicit declaration of built-in function 'printf'
hello.c:23:4: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
```
Using Compilers on the Cluster (3/3)

**gcc -o hello -Wall hello.c**
- `-Wall` will show all warning messages

**gcc -E hello.c**
- Show expanded source code

**gcc -o hello -g hello.c**
- `-g` will build the binary with debug symbols

**gcc -S hello.c**
- Create assembly file called `hello.s`

**gcc -o hello -O3 hello.c**
- `-O3` will build the binary with level 3 optimizations
- Levels 0 to 3 (most aggressive) available
- Can lead to faster execution times
- Default is `-O0` in GCC and `-O2` in Intel suite

**gcc -c hello.c**
- Create object file called `hello.o`

**icc -o hello -xHost hello.c**
- Use Intel's C compiler to aggressively optimize for the specific CPU microarchitecture

Vectorized loop execution is enabled with `-O3` for GCC and `-O2` for Intel.

Using the `-xHost` option leads to poor binary portability. Only use this option if you are sure the binary will always be executed on a specific processor type.
• **Statically Linked Library**: naming convention: liblibraryname.a (e.g. libcurl.a is a static curl library)

  • Linker copies all library routines into the final executable.
  • Requires more memory and disk space than dynamic linking.
  • More portable because the library does not need to be available at runtime.

• **Dynamically Linked Library**: naming convention: liblibraryname.so (e.g. libcurl.so is a dynamic curl library)

  • Only the name of the library copied into the final executable, not any actual code.
  • At runtime, the executable searches the LD_LIBRARY_PATH and standard path for the library.
  • Requires less memory and disk space; multiple binaries can share the same dynamically linked library at once.
  • By default, a linker looks for a dynamic library rather than a static one.

• **Do NOT need to specify the location of a library at build time if it’s in a standard location** (/lib64, /usr/lib64, /lib, /usr/lib). For example, libc.so lives in /lib64.
External Libraries (2/2)

- Linking to libraries in non-standard locations requires the following information at build-time:
  - Name of library (specified with `-llibraryname` flag)
  - Location of library (specified with `-L/path/to/non/standard/location/lib`)
  - Location of header files (specified with `-I/path/to/non/standard/location/include`)

```bash
gcc -L/usr/local/gsl/latest/x86_64/gcc46/nonet/lib -I/usr/local/gsl/latest/x86_64/gcc46/nonet/include -l gsl -lgslcblas bessel.c -Wall -O3 -o calc_bessel
```

- In this example, two libraries (gsl and gslcblas) are linked to the final executable.
- Alternatively, use `LIBRARY_PATH` and `C_INCLUDE_PATH` to specify locations of libraries and headers.

- Check the `LD_LIBRARY_PATH` and output of the `ldd` command before running the program:
  - `LD_LIBRARY_PATH` shows list of directories that linker searches for dynamically linked libraries
  - Run `ldd ./my_prog` to see the dynamically linked libraries needed by an executable and the current path to each library
Can I build an executable on computer A and run it on computer B?

*It depends! Are the platforms the same?*

- CPU instruction set architecture (e.g. x86_64)
- Operating system
- External libraries

**Platform**

- This is why you often see different installers for different operating systems – the installer is simply copying a pre-built binary to your machine!
- Different CPU architectures are present on the cluster, so be sure to compile without overly aggressive optimizations or specify the target CPU architecture/family in your SLURM script (e.g. `#SBATCH --constraint=haswell`)

Support for specific vectorization extensions is also required for portability. For example, you cannot build a program with AVX2 on platform A and run it on platform B if AVX2 is not supported by platform B!
Many different compilers exist but not all compilers are created equal!

- GCC, Intel, Absoft, Portland Group (PGI), Microsoft Visual Studio (MSVS), to name a few.
- Some are free, others are not!
- It is not unusual (especially with large projects) for compiler A to build a program while compiler B fails.
- Error messages and levels of verbosity can also vary widely.

Performance of program can be very compiler-dependent!

- This is especially true in scientific and high-performance computing involving a lot of numerical processing.
- Compiler optimizations are especially tricky, sometimes the compiler needs help from the programmer (e.g. re-factoring code so the compiler can make easier/safer decisions about when to optimize code).
- Some compilers (especially Intel’s) tend to outperform their counterparts because they have more intimate/nuanced information about a CPU’s architecture (which are often Intel-based!).
**Automating the Process: Makefiles (1/3)**

- The **Make** tool allows a programmer to define the dependencies between sets of files in programming project, and sets of rules for how to (most often) build the project.

- Default file is called **Makefile** or **makefile**.
- Allows build process to be broken up into discreet steps, if desired. For example, separate rules can be defined for (i) compiling+assembling, (ii) linking, (iii) testing, and (iv) installing code.
- Make analyzes the timestamps of a target and that target’s dependencies to decide whether to execute a

---

**By defining dependencies, you can avoid unnecessarily rebuilding certain files. For example, in the example on the right, project2.c does not need to be re-compiled if changes have been made to project1.c.**
AUTOMATING THE PROCESS: MAKEFILES (2/3)

- Make analyzes the timestamp of a target’s last modification and compares it to that of the target’s dependencies to decide whether to execute the command(s) defined for that target’s rule.

Makefile Template

```
target: dependencies          # rule
  command1                   # shell command
  command2                   # shell command
```

- A “target” is a label/identifier for a rule
- Often the target is either the name of a file or a conventional rule (e.g. “install”)
- Dependencies are files that the target depend on
- Commands must be preceded by a tab

Example Makefile (see previous slide)

```
executable: project1.o project2.o
  gcc -o executable project1.o project2.o

project1.o: project1.c common.h
  gcc -c project1.c      # generates project1.o

project2.o: project2.c common.h
  gcc -c project2.c      # generates project2.o
```

- There are often multiple rules defined per Makefile
- By just typing “make”, the first rule in the file will be executed
AUTOMATING THE PROCESS: MAKEFILES (3/3)

$ ls
common.h Makefile project1.c project2.c
$ make
gcc -c -o project1.o project1.c
gcc -c -o project2.o project2.c
gcc -o executable project1.o project2.o
$ make
make: `executable' is up to date.
$ touch project2.c
$ make
gcc -c -o project2.o project2.c
gcc -o executable project1.o project2.o
$ make clean
rm -f project1.o project2.o executable

- Notice that Make is smart enough to not rebuild the program if no files have been modified since our last build.
- Make is also smart enough to only re-compile project2.c when it has been changed but project1.c has not.

To learn more about Makefiles, check out the following tutorial:
https://swcarpentry.github.io/make-novice/

make
- Generally builds the entire project.

make clean
- Deletes intermediate build files to start the build process from scratch.

make test
- Generally runs unit tests.

make install
- Generally installs the software.

“make install” generally fails with “permission denied” errors if you do not have administrative privileges or have not configured the build to install into a local directory.
AUTOMATING THE PROCESS: CONFIGURE SCRIPTS (1/2)

• A configure script is an executable file responsible for building a Makefile for a project.

• Determining the dependencies on a given system is difficult to predict and subject to constant change – writing a Makefile by hand for each system (or even a subset of representative systems) would be an enormous challenge and an administrative hassle.

• Instead, a configure script can be used to scan a system in search of all the needed dependencies (including versions of software, locations of external libraries), and build a Makefile that is specific to that system.

• Configure scripts are indispensible for large projects especially where the number of dependencies is large and difficult to manage/track.

• Alternatives to the configure script exist (cmake being the most common).

```
./configure
make
make test
make install
```

• Building projects on Linux at times this simple.
• Run only if you have administrative rights on system.

```
./configure --prefix=/my/local/dir
make
make test
make install
```

• --prefix option needed if installing in home directory on the cluster.
Many configure scripts support a number of different options for configuring your build.

-./configure --help

- Show command line options.

```bash
$ ./configure --help
'configure' configures meep 1.2.1 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help          display this help and exit
  --help=short        display options specific to this package
  --help=recursive    display the short help of all the included packages
  -V, --version       display version information and exit
```
**Make and Configure Macros**

- There are a number of “macros” (think of as variables) that have standard meanings in Make and configure scripts. These macros can generally be exported as environment variables to customize your build.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>C compiler command (e.g. gcc)</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>C compiler flags (e.g. –Wall –O3)</td>
</tr>
<tr>
<td>CPP</td>
<td>C preprocessor command (e.g. gcc)</td>
</tr>
<tr>
<td>CXX</td>
<td>C++ compiler command (e.g. g++)</td>
</tr>
<tr>
<td>CXXFLAGS</td>
<td>C++ compiler flags (e.g. –Wall –O3)</td>
</tr>
<tr>
<td>LDFLAGS</td>
<td>Linker flags (e.g. –L/path/to/lib)</td>
</tr>
<tr>
<td>LIBS</td>
<td>Library names (e.g. –lcurl)</td>
</tr>
<tr>
<td>FC</td>
<td>Fortran compiler command (e.g. gfortran)</td>
</tr>
<tr>
<td>FFLAGS</td>
<td>Fortran compiler flags (e.g. –O3)</td>
</tr>
<tr>
<td>MPICC</td>
<td>MPI C compiler wrapper command (e.g. mpicc)</td>
</tr>
</tbody>
</table>
Compiled vs. Interpreted Languages

What about interpreted languages?

Compiled Language

- Faster execution time
- Slower development time
- Less portable
- C, C++, Fortran

Interpreted Language

- Slower execution time
- Faster development time
- More portable
- Python, Matlab, R, Ruby, Julia

The tradeoffs listed to the left are not universally true but in general apply.

Many popular modules/packages (e.g. NumPy, SciPy) loaded from interpreted languages are compiled shared object files and offer comparable performance to pure compiled languages.