CS2141 – Software Development using C/C++

Compiling a C++ Program

g++

- g++ is the GNU C++ compiler.
- A program in a file called hello.cpp:

```
#include <iostream>
using namespace std;
int main( ) {
   cout << “Hello world!” << endl;
}
```

- Typing '**g++ hello.cpp**' in a shell will produce an executable called **a.out**.
- Typing '**./a.out**' will run the program.

g++ Options

- Use the **`-o`** flag to change the executable name:

```
> g++ -o hello hello.cpp
> ./hello
Hello world!
```

- Some other flags:
  - **`-g`** Adds debugging information to the executable
  - **`-Wall`** Turns on all the warnings. Sometimes this complains about things that the programmer did on purpose, which is why it is turned off by default.

```
> g++ -g -Wall -o hello hello.cpp
```

Compiling Multiple Files

- If the source is in multiple files, all the files can be listed on one line and compiled together:

```
> g++ -o quack quack.cpp moo.cpp
```

- Always recompiling every file when just one file has changed is inefficient.
- Use the **`-c`** flag to create an *object file*:

```
> g++ -c quack.cpp
> g++ -c moo.cpp
```

- This produces files called **`quack.o`** and **`moo.o`**.

Compiling Multiple Files cont.

- The object files are then linked to form the executable:

```
> g++ -o quack quack.o moo.o
```

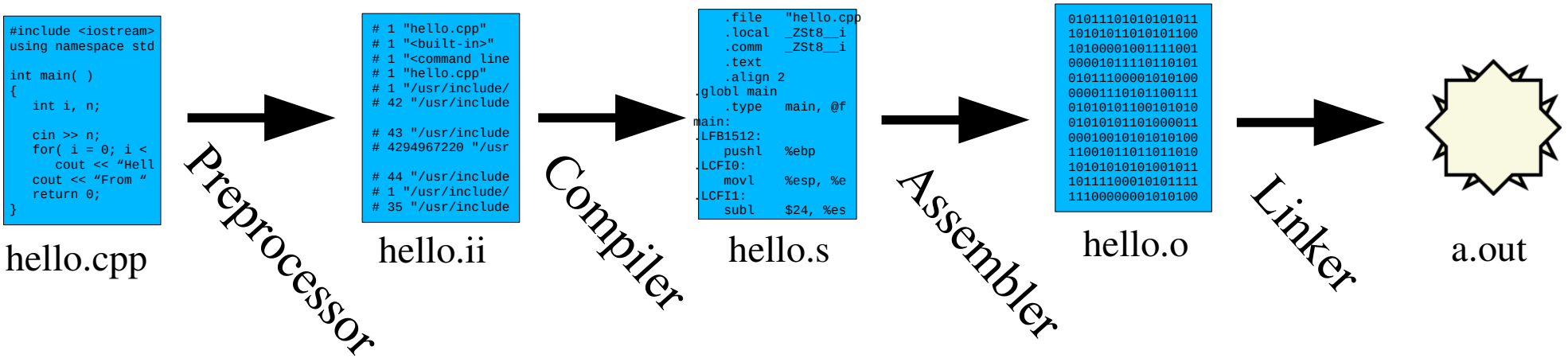
- Now if a source file is changed, only that one file needs to be recompiled:

```
> g++ -c quack.cpp
> g++ -o quack quack.o moo.o
```

- An object file contains machine code and must be linked with other object files to form an executable program.

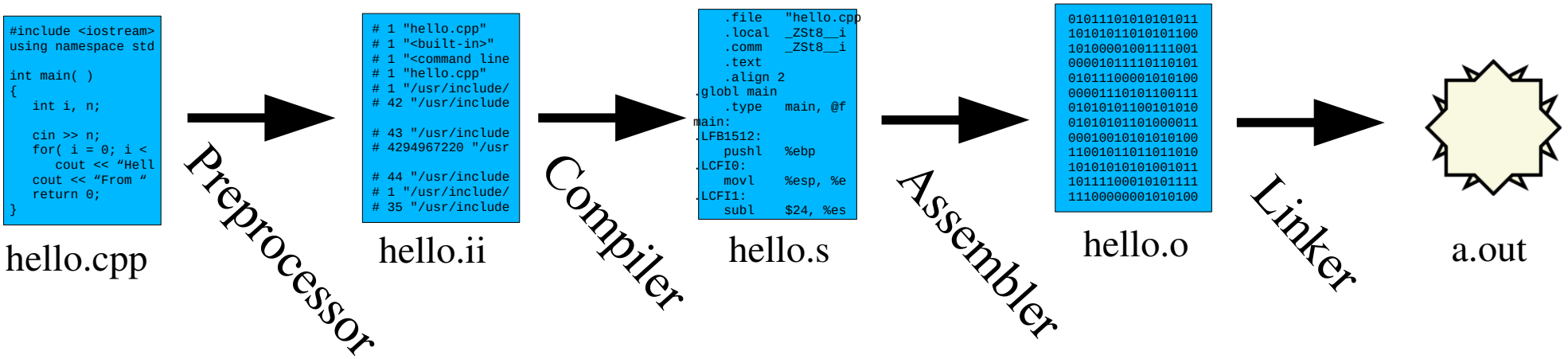
The Compilation Process

- What happens after typing '**`g++ hello.cpp`**':
  - First **`hello.cpp`** gets preprocessed
  - The preprocessed file is compiled to assembly code.
  - The assembly code is assembled to machine code.
  - The machine code is linked to produce the executable.



The Compilation Process cont.

- Compilation can be stopped at any stage:
  - **`cpp hello.cpp > hello.ii`** produces hello.ii
    (cpp is the C and C++ preprocessor)
  - **`g++ -s hello.cpp`** produces hello.s
  - **`g++ -c hello.cpp`** produces hello.o



Header Files and Source Files

- Programs are usually divided into multiple files.
- *Header files* (end in .h) contain things like class definitions and function prototypes.
- *Source files* (end in .cpp) contain implementations of functions and class methods.
- Header files are included into source files and other header files using the `#include` directive:

```
#include <file.h> or
#include “file.h”
```

Header Files and Source Files cont.

- The **`#include`** directive tells the preprocessor to paste the contents of the included file at that spot.
  - Filenames in angle brackets are system headers.
  - Filenames in quotes are local headers.
- Usually header files use *include guards* to prevent a header from being included more than once in the same compilation unit:

```
#ifndef FILENAME_H  // If FILENAME_H is not defined ...
#define FILENAME_H  // ... then define FILENAME_H and ...
...                 // ... whatever else ...
#endif              // ... up until here.
```

Example Program

- Consider a small program with three files: **`hello.h`**, **`hello.cpp`**, and **`main.cpp`**
- **`hello.h`** is a header file. It contains prototypes for a couple of functions:

```
#ifndef HELLO_H
#define HELLO_H

void hello( );
void goodbye( int );

#endif
```

Example Program cont.

- **hello.cpp** implements functions from **hello.h**:

```
#include <iostream>
using namespace std;
#include “hello.h”

void hello( ) {
   cout << “Hello world!” << endl;
}

void goodbye( int n ) {
   for( int i = 0; i < n; ++i )
      cout << “Goodbye!” << endl;
}
```

Example Program cont.

- **main.cpp** contains the main function:

```
#include “hello.h”

int main( )
{
   hello( );
   goodbye( 28 );
   return 0;
}
```

Makefiles

- To compile the example program:

```
> g++ -c hello.cpp
> g++ -c main.cpp
> g++ -o hello hello.o main.o
```

- To avoid repeatedly issuing the commands to compile the program, a *makefile* can be used

- Once a makefile is written and saved in a file called **`Makefile`**, the program can be compiled by simply running **`make`**:

```
> make
```

Creating a Makefile

- A makefile mostly contains rules, with each rule looking something like this:

```
target: dependencies
(tab) command
(tab) another command
...   ...
```

  - The *target* is the name of a file or an action.
  - *Dependencies* are files needed to create the target.
  - Each *command line* **must start with a tab**.

- Comments start with a #

Makefile Example 1

- A very simple makefile for the example program:

```
hello: hello.cpp hello.h main.cpp
   g++ -o hello hello.cpp main.cpp
```

- To use the makefile and run the program:

```
> make
> ./hello
Hello world!
Goodbye!
...
```

- This is not a very good makefile since it will force **`make`** to always recompile everything

Makefile Example 2

- A better makefile:

```
hello: hello.o main.o
   g++ -o hello hello.o main.o

hello.o: hello.cpp hello.h
   g++ -c hello.cpp

main.o: main.cpp hello.h
   g++ -c main.cpp
```

- Now **`make`** will only have to recompile the files that were changed

Cleaning

•Makefiles often include a target called **`clean`** that removes all the object files and executables:

```
# ... other targets ...

clean:
   rm -f hello main.o hello.o
```

•By default, **`make`** builds the first target in the makefile. To use the **`clean`** target, just pass it as an argument to **`make`**:

```
> make clean
```

Makefile Variables

- Makefiles often contain a lot of redundancy Variables can eliminate some of the redundancy
  - To declare a variable:
    ```
    VARIABLENAME = value
    ```
  - To use the variable:
    ```
    ${VARIABLENAME}
    ```
- Some common variables:
  - **`CPP`** The name of the C++ compiler
  - **`EXEC`** The name of the executable
  - **`FLAGS`** Any flags for the C++ compiler

Makefile Example 3

```
CPP = g++
FLAGS = -g
EXEC = hello
OBJS = hello.o main.o  # List of object files needed to
                       # build the executable.

${EXEC}: ${OBJS}
   ${CPP} ${FLAGS} -o ${EXEC} ${OBJS}

hello.o: hello.cpp hello.h
   ${CPP} ${FLAGS} -c hello.cpp

main.o: main.cpp hello.h
   ${CPP} ${FLAGS} -c main.cpp

clean:
   rm -f ${EXEC} ${OBJS}
```

Abstract Rules

- An abstract rule tells how to build a file **`*.s2`** from a file **`*.s1`**, where **`s1`** and **`s2`** are suffixes.
- The following variables can be used in an abstract rule:
  - **`$<`** The dependencies that changed.
  - **`$@`** The target.
  - **`$^`** All the dependencies.
- A line is needed listing the suffixes used:

```
.SUFFIXES: s1 s2 ... sn
```

Makefile Example 4

```
.SUFFIXES: .cpp .o
CPP = g++
FLAGS = -g
EXEC = hello
OBJS = hello.o main.o

${EXEC}: ${OBJS}
   ${CPP} ${FLAGS} -o ${EXEC} ${OBJS}

.cpp.o:                                    # Abstract rule
   ${CPP} ${FLAGS} -c $<

# Still need to list the dependencies for object files
hello.o: hello.cpp hello.h
main.o: main.cpp hello.h

clean:
   rm -f ${EXEC} ${OBJS}
```