Compiling a C++ Program

- g++ is the GNU C++ compiler.
- A program in a file called hello.C:
  ```cpp
  #include <iostream>
  using namespace std;
  int main( ) {
    cout << "Hello world!" << endl;
  }
  ```
- Typing 'g++ hello.C' in a shell will produce an executable called a.out.
- Typing './a.out' will run the program.
g++ Options

• Use the \texttt{-o} flag to change the executable name:
  \begin{verbatim}
  > g++ -o hello hello.C
  > ./hello
  Hello world!
  \end{verbatim}

• Some other flags:
  \begin{itemize}
    \item \texttt{-g} Adds debugging information to the executable
    \item \texttt{-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.
  \end{itemize}
  \begin{verbatim}
  > g++ -g -Wall -o hello hello.C
  \end{verbatim}

Compiling Multiple Files

• If the source is in multiple files, all the files can be listed on one line and compiled together:
  \begin{verbatim}
  > g++ -o quack quack.C moo.C
  \end{verbatim}

• Always recompiling every file when just one file has changed is inefficient.

• Use the \texttt{-c} flag to create an \textit{object file}:
  \begin{verbatim}
  > g++ -c quack.C
  > g++ -c moo.C
  \end{verbatim}

• This produces files called \texttt{quack.o} and \texttt{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.C
  > 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.C':
  - First **hello.C** 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.C > hello.ii` produces hello.ii
  (cpp is the C and C++ preprocessor)
  
  • `g++ -s hello.C` produces hello.s
  
  • `g++ -c hello.C` produces hello.o

![Compiling a C++ Program](image)

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 .C) 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:

```c
#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.C`, and `main.C`
- `hello.h` is a header file. It contains prototypes for a couple of functions:

```c
#ifndef HELLO_H
#define HELLO_H

void hello( );
void goodbye( int );

#endif
```
Example Program cont.

• **hello.C** implements functions from **hello.h**:
  
  ```cpp
  #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.C** contains the main function:

  ```cpp
  #include "hello.h"
  
  int main( )
  {
    hello( );
    goodbye( 28 );
    return 0;
  }
  ```
Makefiles

• To compile the example program:
  > g++ -c hello.C
  > g++ -c main.C
  > 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:

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

• 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:

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

```c
hello.o: hello.C hello.h
g++ -c hello.C
```

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

• 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:

```make
# ... 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`:

```bash
> make clean
```

Makefile Variables

• Makefiles often contain a lot of redundancy. Variables can eliminate some of the redundancy

  • To declare a variable:
    ```make
    VARIABLENAME = value
    ```

  • To use the variable:
    ```make
    ${VARIABLENAME}
    ```

• Some common variables:

  • `CXX` The name of the C++ compiler
  • `EXEC` The name of the executable
  • `CXXFLAGS` Any flags for the C++ compiler
Abstract Rules

• An abstract rule tells how to build a file *.$s1$ from a file *.$s2$, 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:
  $\.SUFXIES: s1 s2 \ldots sn$
.SUFFIXES: .C .o
CXX = g++
CXXFLAGS = -g
EXEC = hello
OBJS = hello.o main.o

${EXEC}: ${OBJs}
    ${CXX} ${CXXFLAGS} -o ${EXEC} ${OBJs}

.C.o: # Abstract rule
    ${CXX} ${CXXFLAGS} -c $<

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

clean:
    rm -f ${EXEC} ${OBJs}