## Compilers, compilation

- Compilers are simply programs that read files of source code written in one HLL language and translate it into another language (e.g. another HLL, assembler, or machine code)
- Interpretters are programs that read instructions in a source language, translate an execute them (i.e. read, translate, and execute one instruction then read, translate, and execute the next, etc)

## Compiler actions

- Read the source code and put into standardized format (e.g. standardize whitespace)
- Apply any preprocessor instructions (#includes etc)
- Turn the resulting source code into a list of symbols or tokens (keywords, identifiers, brackets, etc)
- Analyse the list of symbols for syntactic corrrectness, build symbol table (e.g. with info about defined items), generate intermediate representation (implicit “meaning” of the code)
- Generate and optimize target code from intermediate rep

## Symbol table

- The symbol table(s) contains information about all the programmer-defined classees, functions, data types, constants, variables, etc
- While processing the code, whenever it encounters a symbol it refers back to the table to access relevant information
- Symbol table kept in memory during compilation, not needed once the code has been translated into target language (i.e. not stored with the compiled code)

## Compiling single-file programs

- When source code for a program is entirely contained in one file, everything the compiler needs is in that file
- e.g. when you call a function, the actual implementation of the function is available in the same file
- e.g. if entire program is in prog.cpp, then we can compile "directly" to an executable (say progx) using command like

```
g++ prog.cpp -o progx
```

## Compiling multi-file programs

- Suppose a program is divided into multiple files, with some classes, functions, methods etc implemented in one file and others implemented in other files
- Either we would have to give the compiler all the files to recompile at once (suppose there were hundreds, or thousands...) or the compiler needs some way to compile files separately, then join the separate parts together later to build the full executable
- The latter approach, separate compilation, is far more effective, and far more widely used

## Separate compilation, linking

- Suppose a function, foo, is called from one file, e.g. prog.cpp, but implemented in another, e.g. funcs.cpp
- compile prog.cpp into one file of machine code (with call to foo), compile funcs.cpp into another file of machine code (with body of foo), then link together to form an executable, e.g.

```
g++ -c funcs.cpp -o funcs.o
g++ -c prog.cpp -o prog.o
g++ funcs.o prog.o -o progx
```

(the -c option tells g++ that we’re doing separate compilation)

## Coordination across files

- Problem: when separately compiling prog.cpp, the compiler still needs to know profile of function foo (parameters, types, return type, etc) so it can check if the function call to foo is valid
- Header files (.h files): one solution is to have a file with the definition (though not the implementation) of the shared items – e.g. the prototype for function foo
- E.g. we create funcs.h, which contains definitions for anything funcs.cpp wishes to share, then prog.cpp references this to check prototype of foo

## Header files

- Header files act like a contract between the file supplying a function or method and the file calling the function or method
- One file provides an implementation matching the header file's description, the other file calls the function in a way matching the header file's description
- When the compiler links the two object files it then links up the call from prog.o with the implementation from funcs.o

## Example: the three files

```
// funcs.h

int foo(int x);
```

```
// funcs.cpp

#include “funcs.h”

int foo(int x)
{
    return x + x;
}
```

```
// prog.cpp

#include “funcs.h”

int main()
{
   int y;
   y = foo(3);
}
```

## C/C++ header style

- In C/C++ we use the .h extension for header files, and the .c or .cpp (or .cc or .cxx or .CC etc) extensions for the implementation files
- Each of the .cpp files use a “#include” line to indicate which header files they’re using, e.g. #include “funcs.h”
- The #include is a preprocessor directive, telling the compiler that, before actually compiling, go get all the code from the specified file and (virtually) copy/paste it here

## Example: the compilation process

- Compile the two .cpp files, creating .o (object)

```
g++ -c prog.cpp -o prog.o
g++ -c funcs.cpp -o funcs.o
```

- Link the two .o files, creating an executable

```
g++ prog.o funcs.o -o progx
```

- Run the executable

```
./progx
```

## Includes: our own files vs built in

- when a C++ compiler is installed, it is configured with the location of built-in libraries, .h and pre-compiled .o files
- programs #include these using < > without specifying a path to them, e.g. #include <iostream>
- we've done this many times already, with <iostream>, <cstdio>, <string>, <cstring>, <cmath>, etc
- when linking .o files to form an executable we don't need to specify the location of these library .o files

## include directories, -I and -L

- we can tell the compiler the location of additional directories of .h files use -I and the path/directoryname, e.g. g++ -Isomedir/myincludes myprog.cpp
- the compiler will find the .h files in such directories even if we forget to specify a path in our .cpp file's #includes
- similarly a -L option exists in g++ to specify the location of additional directories of .o files to automatically support

## What if we edit a .cpp or .h file?

- If we edit a .cpp file, we need to recompile that one file (updating its .o) and then relink the .o’s for a new executable (we don’t need to recompile the other .cpp file)
- If we edit the .h file, since it contains definitions used by both other files, we need to recompile both .cpp files and also relink the .o’s to create a new executable
- Ideally, when we edit some .h/.cpp files we want to recompile exactly the right set of files to correctly update what needs updating, but not recompile anything unnecessarily

## Automated recompilation

- Many software development tools automate this process for us: looking at what’s been edited and going through the heirarchy of #includes to figure out exactly what should be recompiled
- We can also build a set of rules that define the conditions under which a file should be recompiled, and the instructions to use to recompile it
- Makefiles are a type of file used to store such information, and the make program is can then be used to recompile, e.g. if we had created a correct makefile we could say “make progx” and the make program would figure out what to do to update progx