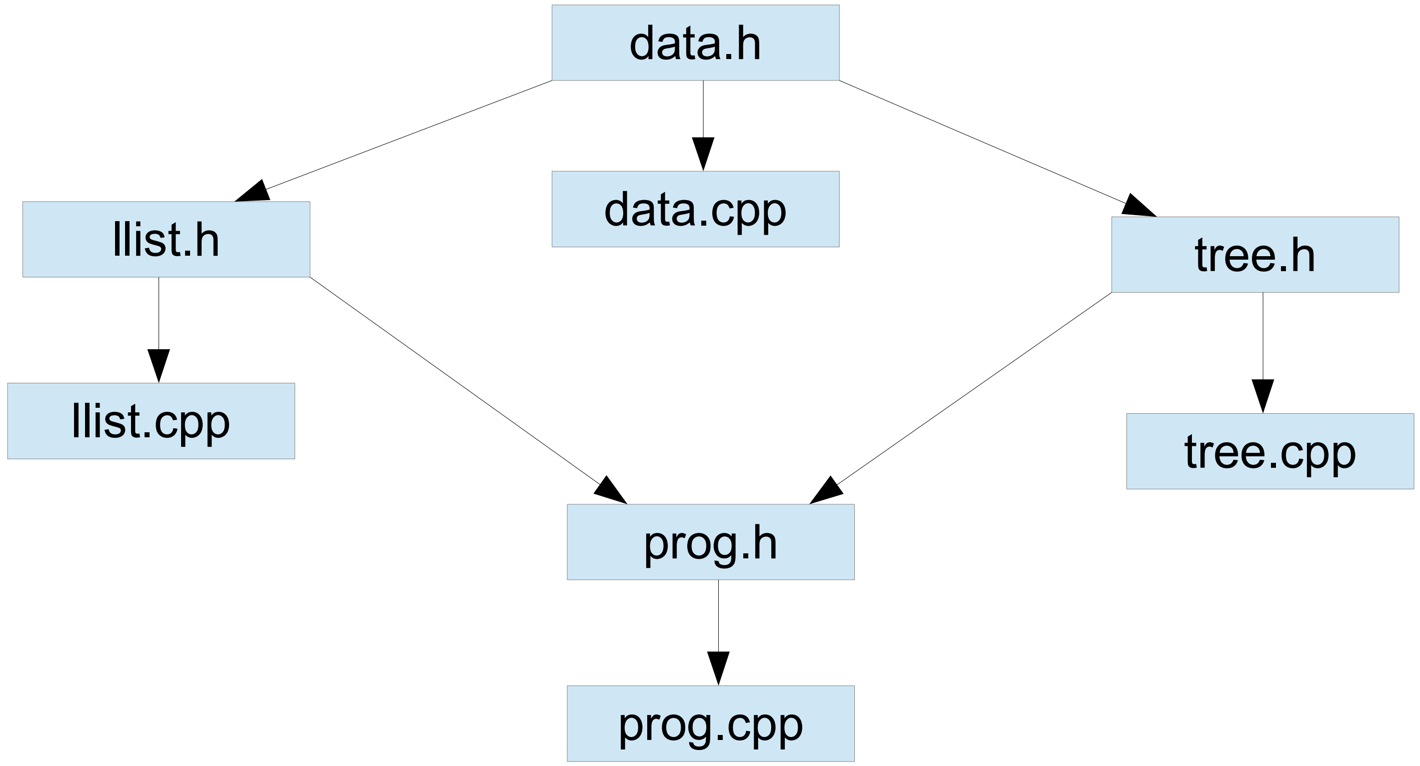
Separate compilation, C++, g++

- Example with more complex set of #includes, suppose:
  - data.h defines a data type and some functions, implemented in data.cpp
  - llist.h defines a linked list, using stuff from data.h, and list implementation is in llist.cpp
  - tree.h defines a binary tree, using stuff from data.h, and tree implementation is in tree.cpp
  - prog.h defines some types used in the main routine, which is implemented in prog.cpp

Include heirarchy

- data.cpp needs to #include “data.h”
- llist.h and tree.h each need to #include “data.h”
- llist.cpp needs to #include “llist.h”
- tree.cpp needs to #include “tree.h”
- prog.h needs to #include “llist.h” and also #include “tree.h”
- prog.cpp needs to #include “prog.h”

What do the includes copy, to where



Headers in different directories

- built in libraries: `<someheader>` or `<someheader.h>`
- headers in same directory `#include “someheader.h”`
- headers in another directory, specifying path

  ```
  #include “../../otherdir/someplace/someheader.h”
  ```
- headers can be placed in directory named `include` then

  ```
  #include “someheader.h”
  ```
- can specify alternate name of include directories with -I flag when compiling, e.g. `g++ -Isomedirname .....`
- can similarly specify libraries with -L flag

#includes are transitive...

- Note that prog.h #include llist.h, which #includes data.h, so the contents of both get copied into prog.h before compilation
- prog.h also #includes tree.h, which #includes data.h, so the contents of both get copied into prog.h before compilation
- This means that, just before compilation, prog.h has all of the definitions from data.h *twice,* which would naturally cause errors when we try to compile

Need to prevent duplicated includes

- Difficult to prevent on the #include’ing side: suppose A #includes B and C, B #includes X and Y, C #includes D and E, etc, it can become very challenging to trace back all the chains
- Solution is a the included end: is there a way to say “if I’ve already been #included, please skip me this time”?
- Yes, using the preprocessor directives #ifndef and #define...

#ifndef guards in header files

- We make up an identifier name, say “if this is the first time you’ve seen this identifier then include me, otherwise skip me” and inside the if statement we define the identifier and include all the “real” code for the header file

```
#ifndef MADEUPIDENTIFIER
#define MADEUPIDENTIFIER
// all the “real” code goes in here
...
// then we end the #ifndef with a #endif
#endif
```

pragma once

- alternative to #ifndef guards in the headers is to begin the header file with a directive specifying only include once:

  ```
  #pragma once
  ```

- only real drawback is that some compilers implement this less effectively, especially if header files are referred to be a variety of different paths
  - e.g. a relative path in one place, an absolute path in another, through a symbolic link in another

Compiling our example

```
 g++ -c data.cpp -o data.cpp
g++ -c llist.cpp -o llist.o
g++ -c tree.cpp -o tree.o
g++ -c prog.cpp -o prog.o
g++ data.o llist.o tree.o prog.o -o progx
```

Changing a .cpp file

- As with earlier examples, if we edit a single .cpp file we then need to update its .o file and relink the executable, e.g. for tree.cpp
- g++ -c tree.cpp -o tree.o
- g++ data.o llist.o tree.o prog.o -o progx

Changing a .h file

- If we edit a header file, we need to recompile all the .cpp files that include it (directly or indirectly) and then relink the executable
- e.g. if we edit tree.h then we need to recompile tree.cpp, prog.cpp, and then relink
- e.g. if we edit data.h then we need to recompile all four .cpp files and then relink

Common compiler options

- As with the -c and -o, we can add a variety of compilation options with g++, here are a few:
- -Wall turns on “all” warnings
- -Wextra turns on a few that “all” doesn’t
- -O turns on basic code optimizations
- -g turns on debugger support (adds symbol table info to the compiled file)
- `-I` specify name of an include directory, e.g. `-Ifoo`
- -L specify name of library to use

Language and compiler versions

- most languages go through regular revisions (fixes, clarifications, feature additions), with changes to the language standards released regularly
  - C++: 98, 11, 14, 17, 20, ...23...
- new versions of each compiler also tend to be released regularly, each supporting a different subset of the features from the recent/upcoming standards
  - sometimes introducing a feature before it makes it into the standard, sometimes after
- when examining any code segment it's very important to be aware of which features it is reliant on, and whether the relevant compiler is compatible