Large Programs and External Libraries

1 Scope - again

- Scope in a single file has two specific uses:
  - Local identifiers visible only in code blocks
  - Global identifiers visible to all functions in one file
- Larger programs need multiple source files
- Therefore scope has to be managed across files in large programs and external libraries like OpenGL
- In C, functions and variables must be *declared* before they are used, but can be *defined* later

2 Multiple source files

- A C program may be divided among any number of source files
- By convention, source files have the extension `.c`
- Each source file contains part of the program
  - primarily definitions of functions and variables
- One source file must still contain a function named `main()`, which is the entry point for the program

3 Header files

- Problems that arise when a program is divided into several source files:
  - How can a function in one file call a function that's defined in another file?
  - How can a function access an external variable in another file?
  - How can two files share the same macro definition or type definition?
- The answer lies with the `#include` directive, which makes it possible to share information among any number of source files

4 Header and multiple source files

- `func.h`
  - declarations
  - The header file contains the declarations needed to use the functions in `func.c`
- `func.c`
  - `#include "func.h"`
  - definitions
  - The source file contains all the global and private functions and variables chosen
- `main.c`
  - `#include "func.h"`
  - This should contain at least the `main()` function

5 Sharing identifier declarations

- When variables and functions need to be shared between files there often needs to be a way to separate declarations & definitions
- We can then declare identifiers so that they can be used in any file, while keeping the definition in a single place in one file
- The solution to this is the `extern` modifier

6 extern use with variables

- Use the header file to contain the declarations of variables that are shared with other files

```
func.h
  extern int cost; // declaration

func.c
  int cost = 1; // definition
```

7 extern use with functions

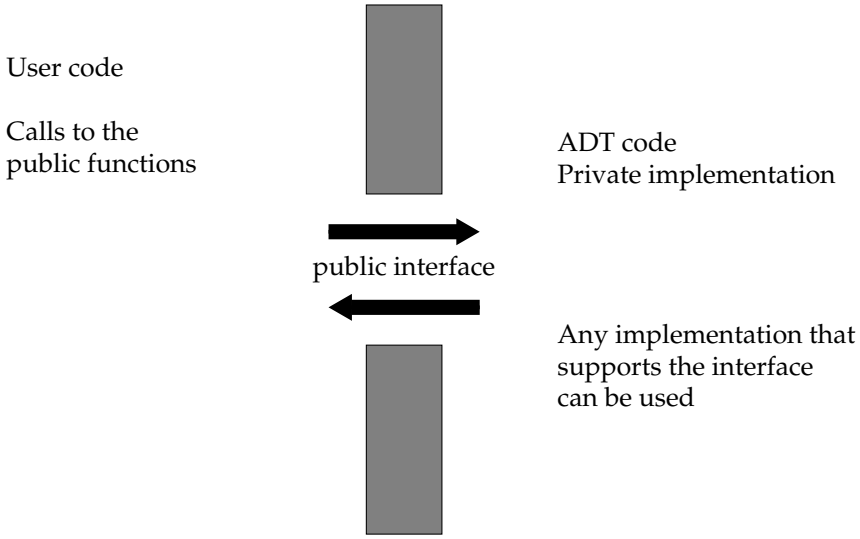
- Use the header file to contain the declarations of just the functions shared with other files

```
func.h
 +graytext[extern] void set_cost( int val ); // declaration
func.c
  void set_cost( int val ) {
    cost = val;
  }                                  // definition
```

8 Abstract data types

- How should we divide functions into files?
- Abstract data types pre-date O-O concepts
- Identify key data types and encapsulate them in separate files
- Access the instances using the public interface, functions and variables
- Hide other implementation details from the users

9 Walls of abstraction



10 ADT Benefits

- Abstraction
  - from the implementation details
- Encapsulation
  - user cannot access internals

- Independence
  - reduces number of interactions
- Flexibility
  - implementation change transparent
- Another protection from our brain's limited powers to manage complexity in systems

11 ADT implementation

- C usually implements complex types with a `struct` definition
- In part to hide the details of the `struct` ADTs are sometimes implemented with only a pointer type visible to the user, the `struct` itself remains private to the ADT source file
- More modern languages than C have clearer ways to handle this through class definitions

12 ADT implementation for `POINT_T`

- Publicly in the header file `point.h` define a new type:

```
typedef struct PointStructType *POINT_T;
```

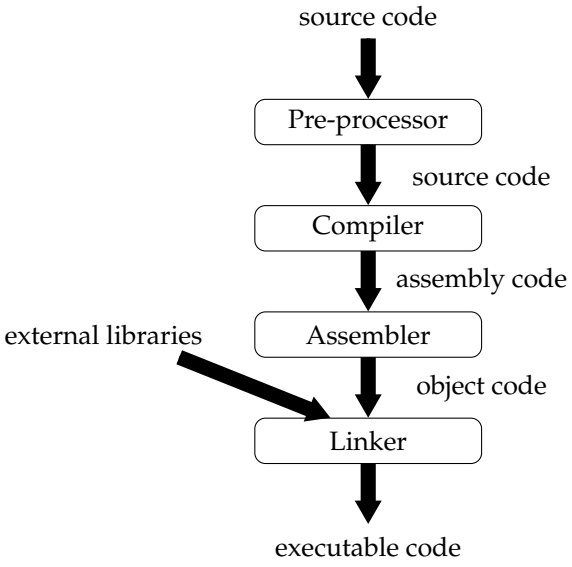
- Privately in the source file point.c declare the underlying structure:

```
struct PointStructType {
  double array[NUM_DIMS];
};
```

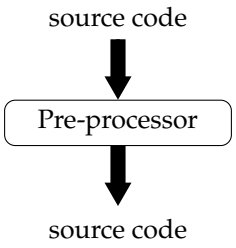
13 Summary: large programs

- Split large software projects into separate files to manage complexity
- `extern` allows variables and functions to be declared and shared in header files
- `#include` allows header (`.h`) files to be included wherever needed
- `typedef` allows the creation of new abstract data types that encapsulate implementation privately

14 Recap - Compilation Model



15 The C Preprocessor



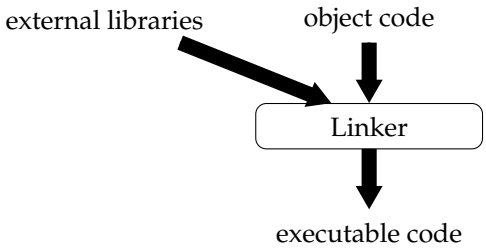
- Directives such as `#define` and `#include` are handled by the preprocessor, a piece of software that edits C programs just prior to compilation
- Its reliance on a preprocessor makes C (and C++) unique among major programming languages

16 Conditional include

```
#ifndef CODE_H
  #define CODE_H // define the identifier
  extern void setCount( int val );
#endif
```

- This allows the header file to be `#included` many times
- If the header file has not been seen before - set definitions
- And set that we've seen it before - `#define CODE_H`
- Otherwise skip

17 The link editor (linker)



- The linker's job is to combine all the files needed to form the executable
- It specifically has to resolve all symbols, functions and variables, it most often fails when it can't find required object code, for example because it is in the wrong folder, or you have forgotten to specify which external library to link with e.g. the maths library with `-lm`

18 Makefiles

- When we have a number of files to compile together we need a rule-set to perform this
- The `make` command provides this
- Requires a rule-file called the `Makefile`
- Declarative programming style set of rules for building the program
- Format of each rule:

```
target [target ...]: [component ...]
        [command 1]
        ...
        [command n]
```

- N.B. Tab character
- `target` - what you want to make
- `component` - something which needs to exist (might need another rule)

19 Makefiles: Example

- Files: `main.c`, `counter.h`, `counter.c`, `sales.h`, `sales.c`

```
all: counter.o sales.o main.c
	gcc -o program main.c counter.o sales.o

counter.o: counter.c counter.h
	gcc -c counter.c

sales.o: sales.c sales.h
	gcc -c sales.c

clean:
	rm -rf program counter.o sales.o
```

20 Makefiles: Macros

- Macros can be used to store definitions
  - `AUTHOR = Konrad Dabrowski`
- They can be generated from commands
  - `DATE = `date``
- And used in the `Makefile`

```
all:
	echo $(AUTHOR) compiled this on $(DATE)
```

- `all:`
- Running this gives:
- `echo Konrad Dabrowski compiled this on `date``
  - `Konrad Dabrowski compiled this on Thu 16 Jan 10:54:36 GMT 2020`

21 Makefiles: Pattern Rules

- We can specify a pattern rule which matches multiple files
- e.g. compile C files into object files:

```
DEPS = counter.h sales.h
%.o: %.c $(DEPS)
	gcc -c $< -o $@
```

- This would change our original Makefile example to:

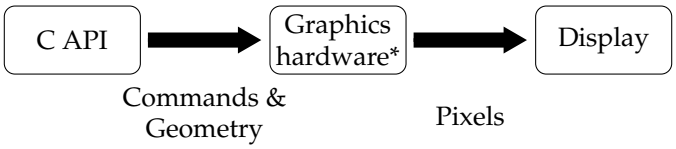
```
all: counter.o sales.o main.c
	gcc -o program main.c counter.o sales.o

%.o: %.c $(DEPS)
	gcc -c $< -o $@

clean:
	rm -rf program counter.o sales.o
```

22 Makefiles: A few comments

- Comments - lines starting with #
- Lazy evaluation
- If a target exists and has a timestamp later than all of its components assume it is up to date and don't bother to re-process
- Nothing to do with C
- Although Makefiles are often used with C programs there is no intrinsic link - can use with any code/work
- You can run any specific rule by invoking its target:
- `make sales.o`

23 External libraries

- One of the reasons why C is so popular is the huge collection of tried and tested libraries available across many different computing platforms. E.g. OpenGL



- Commands from your program are sent by the API to the graphics hardware which generates pixels for display
- *in OpenGL the hardware behaves as a state machine

24 OpenGL programming

- On its own OpenGL is:
  1. Low level
  2. O/S independent
- Hence it is usually used with:
  - GLU a utility library with high level shape support
  - GLUT utility library for window creation and I/O

25 Commonly used C libraries

- general: `libglib` / `libgobject` / `libpthread`
- console: `libncurses`
- 2D graphics: `libX11` / `libSDL`
- 3D graphics: `libGL` / `libGLU` / `libGLUT`
- GUI toolkits: `libgtk` / `libQT`
- Images: `libjpeg` / `libpng` / `libgif`
- text rendering: `libpango` / `libfreetype`
- sound: `libasound` / `libSDL`
- compression: `libz` (`zlib`) / `libgzip` / `libbz2`
- encryption: `libcrypt` / `libssl` / `libgssapi` / `libkrb5`
- XML: `libxml2`
- web: `libcurl`

26 Usage of libraries

- If a library is *statically linked* then a copy of the library is included in the executable
- C/C++/assembly can be combined
- Often bound to other languages e.g. php, XML, curl
- Many of these libraries will be *dynamically linked*
- LGPL (Lesser Gnu Public License) often used
- Try `ldd /usr/bin/php` on Linux to list dynamic dependencies

27 Dynamic vs static linking

- Dynamic linking takes place at run-time not build-time
- Reduces filespace demands (bloat) by keeping only one copy of the library
- Can help with updates e.g. for security
- Dynamic libraries are called differently by OSs
- Linux: shared objects (`.so`)
- Windows: Dynamic Link Libraries (`.dll`)
- OSX: `.dylib`
- Can lead to "DLL Hell": many versions of the same dynamic library
- Best to include version number with library