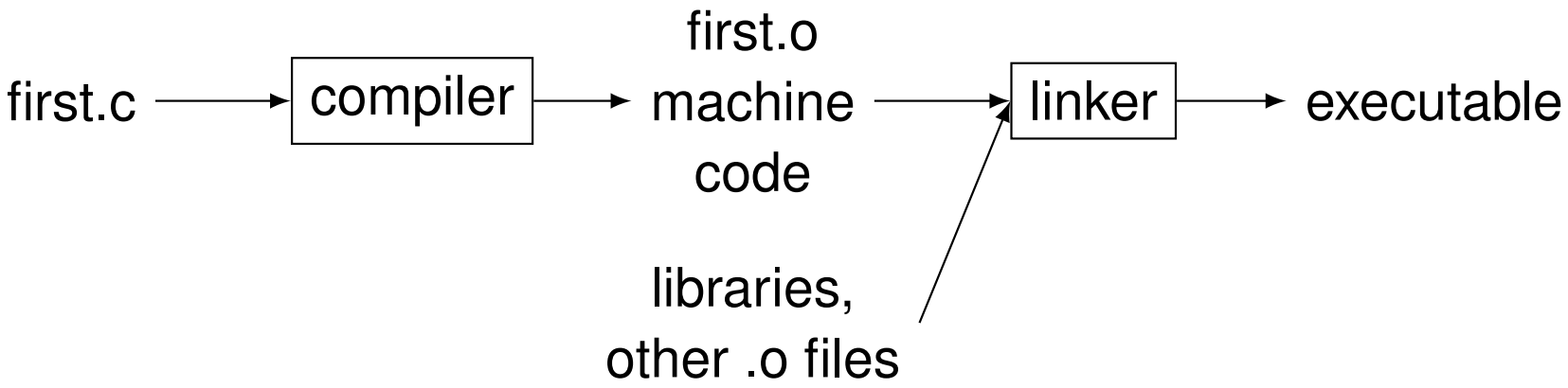
Compiler And Linker Stages



(Incomplete picture. Longer story for “dynamic linking”.)

first.o is called “object [code] file”.

Libraries: where e.g. `printf` really comes from.

Linker: Merges object files and libraries into executable.
Actual linker is ‘ld’, but too technical to use directly.
‘gcc’ serves as convenient linker frontend.

C Compiler Stages

C compiler further divides into:

1. Pre-processor: For `#include` and other # directives.
   (Use '`gcc -E`' to see what it does.)
   This determines the actual C code seen by...
2. Compiler proper: Translates C code to machine code.
   '`gcc -c`' gives object file containing machine code.
   (If curious about assembly version: `gcc -S`)

Pre-processor Directive: Macros

```
#define TABLE_SIZE 20
#define GREETING "Hello"
#define MY_DEBUG_FLAG
```

Define macro. Textual substitution.

Pro-tip: `#define` array sizes to give them good names and only one place to change.

Macros can take parameters (not shown).

`#undef GREETING`
Remove macro.

Pre-processor Directive: `#include`

`#include <foo.h>`
Insert foo.h content here (and pre-process it too). Look for foo.h in system-wide places, e.g., '/usr/include'.

`#include "foo.h"`
Also look in the same directory as current file.

Either way, can tell gcc to search more directories:
`gcc ... -Imydir -Iyourdir ...`
⇒ also search mydir and yourdir

Header files usually contain:

- Macro definitions, type definitions (e.g., `struct`, `typedef`).
- Types of exported functions and global variables, e.g.,
  ```
  double sqrt(double x);
  extern FILE *stdin;
  ```

Pre-processor Directive: Conditional Compilation

```
#ifdef MY_DEBUG_FLAG
    fprintf(stderr, "x = %d\n", x);
#endif
```

That shows a common technique for “debugging code”.
‘`gcc -DMY_DEBUG_FLAG`’ to enable.
Omit ‘`-DMY_DEBUG_FLAG`’ to disable.

Demo code: `conditional.c`

Also useful for e.g. “if Windows”, “if Linux”.

Modularity & Separate Compilation

Would not want one big C file for the whole program:

- Unbrowsable.
- Re-compiling too slow for small changes. (xkcd 303)

Want closely related code in the same file, unrelated code in different files.

E.g., linked list implementation in one file, BST implementation in another file, main program (user of both) in third file.

If you change just one file, want to re-compile just that.

Example to Be Split

```
// Part 1: Rectangle
typedef struct rect { double w, h; } rect;
double rect_area(const rect *r) { ... }

// Part 2: Bounding box
typedef struct point { double x, y; } point;
void bounding_box(rect *r, const point *p, unsigned n)
{ ... }

// Part 3: Main program
int main(void)
{
  rect myrect; point myp[4];
  ...
  bounding_box(&myrect, p, 4);
  ... rect_area(&myrect) ...
}
```

Complete file: monolith.c

Example Splitting Scheme

- `rect.h`:
  `rect` type definition
  `rect_area` name and type
- `rect.c`:
  `rect_area` definition (implementation)
- `bb.h`:
  `point` type definition
  `bounding_box` name and type
- `bb.c`:
  `bounding_box` definition (implementation)
- `mainprog.c`:
  `main program`

Header File Explanation: Overview

Terminology:
“declaration” = just name and type
“function prototype” = function declaration: name and type
“definition” = has implementation

Header files usually contain:

- Macro definitions, type definitions (e.g., `struct`, `typedef`).
- Function prototypes, global variable declarations, e.g.,
  ```
  double sqrt(double x);
  extern FILE *stdin;
  ```

Header File Explanation: Macros, Types, Functions

Why define macros and types in header files:

- When compiling a file that uses a macro or type, compiler wants to see its definition.
- But you don't want to manually copy it to multiple files.

So: Define once in header file, users `#include` it.

Why function prototypes without implementations in header files:

- Compiler wants to check types, but doesn't need actual code for now. (Worry about actual code when linking.)

So: Put type in header file, users `#include` it.

Good habit: Implementer also `#include` it to check consistency.

Header File Explanation: Abstract Type

I said: usually struct definitions in header files.

But struct declaration without definition is also legal:
`struct myrecord;`
Postpones fields to later or another file.

Consequences:

- Fields unknown (obviously).
- `sizeof(struct myrecord)` unknown (consequently).
- Ah but: Pointer type `struct myrecord *` available.

Acts like declaring an abstract type.

Example: stdio.h does this to `FILE`. So you can have `FILE *` but you don't know its fields.

Digression: struct Life Hack

We had this example:

```
typedef struct node {
  int i;
  struct node *next;
  // "nodetype" not available until next line
} nodetype;
```

Now we can solve it:

```
typedef struct node nodetype;
// Now "nodetype *" available.
struct node {
  int i;
  nodetype *next;
};
```

Header File Explanation: Global Variables

Why

`extern int myvar;`

in header file (and can be in multiple .c files), but

`int myvar;`

in exactly one .c file:

`int myvar;`

requests compiler/linker to allocate an address.
It counts as a definition.
We want only one .c file to request it.

`extern int myvar;`

just states existence, name, type.
It counts as a declaration.

Double Include DoubleUnGoodPlusPlus

Illegal to see a type definition the second time.

But mainprog.c risks seeing `rect` definition twice: from rect.h directly, from bb.h which `#includes` rect.h.

Inconvenient to impose “so don’t `#include` rect.h”.

Simpler convention: I use rect.c stuff, I `#include` rect.h. I use bb.c stuff, I `#include` bb.h.

Solution:

- Each header file `#define` a macro to flag “I have been seen”.
- Conditional compilation to skip code if the flag is set.

Conditional Compilation for Unique Include

E.g., ‘foo.h’ can go:

```
#ifndef _FOO_H
#define _FOO_H

typedef struct node {
  int i;
  struct node *next;
} node;

#endif
```

User’s view: First time, `_FOO_H` not yet defined, don’t skip, define `_FOO_H` and `node`. Second time onwards, since `_FOO_H` now defined, skip. Happy.

Namespacing (none)

TL;DR: C has no namespacing mechanism at all. Unlike C++, Java, Python, everyone. . .

People just think up hopefully non-clashing prefixes, e.g., The GTK+ library is full of ‘`gtk_button_new`’, ‘`gtk_window_close`’,. . .

Separate Compilation Example

1. Compile to object files:
   - ‘`gcc -c rect.c`’ (but only if necessary)
   - ‘`gcc -c bb.c`’ (but only if necessary)
   - ‘`gcc -c mainprog.c`’ (but only if necessary)

   Recall: Want to compile changed C files only.
2. Link to executable (but only if any of the above happened): ‘`gcc rect.o bb.o mainprog.o -o mainprog`’

Wouldn’t you like to automate “but only if necessary”? That’s what the ‘make’ program and “Makefile”s are for.

Full doc: GNU Make Manual

Makefile

Most basic Makefile clause (rule) goes like:

```
bb.o : bb.c bb.h rect.h
        gcc -c bb.c
# tab character there, not 8 spaces
```

Meaning: If bb.o absent or older than at least one of bb.c, bb.h, rect.h, then run
`gcc -c bb.c`

Terminology:

- bb.o is a “target”.
- bb.c, bb.h, rect.h are “prerequisites” of bb.o.
  (Exercise: Why is rect.h involved?)
- ‘`gcc -c bb.c`‘ is a “recipe”.

make

If multiple rules in a Makefile:

- ‘`make`’ runs the first rule.
- ‘`make` *target*‘ runs a rule matching that target.

Either way, may recursively trigger running other rules.

Order of rules does not matter otherwise.

Customary to write first rule like

```
all : myexe1 myexe2 myexe3
.PHONY: all
```

which triggers using other rules to build the 3 exes.

‘`.PHONY: all`’ means: ‘`all`’ is just a label, not a file to be produced.

Using Makefile to Reset (“Clean”)

Also customary to add

```
clean :
        rm -f *.o myexe1 myexe2 myexe3
.PHONY: clean
```

Use ‘`make clean`’ to invoke this rule (instead of 1st rule in file).

Variables

Setting a variable (from within):
`CFLAGS = -g`

Using a variable:

```
    gcc $(CFLAGS) -c bb.c
```

Setting a variable from outside:
`make CFLAGS='-g -DMY_DEBUG_FLAG'`
This overrides settings from within.

Environment variables also become make variables.

Conversely, make variables become environment variables when running recipes.

Complete Rules (But Repetitive)

```
mainprog : mainprog.o bb.o rect.o
        gcc -g mainprog.o bb.o rect.o \
            -o mainprog

mainprog.o : mainprog.c bb.h rect.h
        gcc -g -c mainprog.c

bb.o : bb.c bb.h rect.h
        gcc -g -c bb.c

rect.o : rect.c rect.h
        gcc -g -c rect.c
```

(File: Makefile-1)

This gets a little repetitive. . . Also annoying to update as your program grows or gets re-organized.

Automatic Variables And Pattern Rules

```
mainprog : mainprog.o bb.o rect.o
        gcc -g $^ -o $@
```

$^ = all prerequisites
$@ = target

```
%.o : %.c
        gcc -g -c $<
```

Pattern rule, any ‘foo.o’ can be built from ‘foo.c’ by the provided recipe.
$< = first prerequisite

```
mainprog.o : bb.h rect.h
bb.o : bb.h rect.h
rect.o : rect.h
```

Additional prerequisites. Accumulative, not overriding.

(File: Makefile-2)

Automatic Prerequisite Listing

Prerequisites are annoying to manually list too. What if you re-organize your files and forget to update them?

Solution: Add this to Makefile:

```
.depend: mainprog.c bb.c rect.c
        gcc -MM $^ > .depend
include .depend
```

(You can choose another filename for .depend)

Exercise: Examine the content of .depend and why it helps.

(File: `Makefile-3`)