CSCI 2132: Software Development

Writing Large Programs & Make

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Types of Variables in C

a.c

```
int ext = 7;
static int x = 0;

int f() {
     ...
    int y = 2;
     ...
    if x < y {
         ...
        int z = 3;
         ...
    }
     ...
}
```

Static:

- Global
- Global (file scope)
- Static local

On stack:

- Local (function)
- Local (block)

b.c

```


extern int ext;

static int h() {
     ...
}

int g() {
    static w = 5;
     ...
}
```

a.c

```
int ext = 7;
static int x = 0;

int f() {
    int y = 2;
    if x < y {
        int z = 3;
    }
}
```

b.c

```
extern int ext;

static int h() {
}

int g() {
    static w = ext;
}
```

a.o

```
TEXT (code)
  T   f   (public)

DATA
  D   ext (public)
  d   x   (private)
```

Linker

b.o

```
TEXT (code)
  T   g   (public)
  t   h   (private)

DATA
  U   ext (public)
  d   w   (private)
```

Sharing via Header Files

a.c

```
int ext = 7;
static int x = 0;

int f() {
     …
     int y = 2;
     …
     if x < y {
          …
          int z = 3;
          …
     }
     …
}
```

a.h

```
#ifndef A_H
#define A_H

extern int ext;

int f();

#endif // A_H
```

b.c

```
#include “a.h”

static int h() {
     …
     return f();
     …
}

int g() {
     static w = ext;
     …
}
```

Avoid Using Global Variables Like the Plague

Any piece of code that has access to them can modify them.

Tracking the flow of data through the code becomes hard to track.

Better:

- Store “global” data in a `struct`.
- Initialize the `struct` in main (for example).
- Pass the “global” `struct` to every function that needs it.

Command Line Arguments

```
int main(int argc, char *argv[]) {
    ...
}
```

or

```
int main(int argc, char **argv) {
    ...
}
```

- `argc` = number of command line arguments
- `argv` = array of "strings" (`char *`) storing command line arguments
- `argv[0]` = program name

Command Line Arguments

```
$ myprog These are some arguments
```

- argc = 5
  - argv[0] = “myprog”
  - argv[1] = “These"
  - argv[2] = “are”
  - argv[3] = “some”
  - argv[4] = “arguments”
  - argv[5] = NULL

Writing Large Programs

Divide into multiple files:

- Each C file `XXX.c` should be a “module” (one component of your program)
- External interface of each module defined in corresponding header file `XXX.h`

`stack.c`

```
#include “stack.h”

struct _stack_t {
    …
};

stack_t make_stack() {
    …
}

…
```

`stack.h`

```
#ifndef STACK_H
#define STACK_H

typedef struct _stack_t * stack_t;

stack_t make_stack();
void destroy_stack(stack_t);

void push(stack_t, void *);
void *pop(stack_t);

#endif
```

Include Statements

`#include <filename.h>`

- Includes file `filename.h` into the source code (textually)
- Search "system" header files:
  `/usr/include`, `/usr/local/include`, …
  - Search path can be augmented with `gcc` option `-Idir`

`#include "filename.h"`

- Includes file `filename.h` into the source code (textually)
- Search current directory first
- If not found, behave like `#include <filename.h>`

Compiling Large Programs

a.c

```
#include “b.h”
#include “c.h”

void a() {
    …
}
```

b.c

```
#include “c.h”

void b() {
    …
}
```

c.c

```
#include “c.h”

void c() {
    …
}
```

b.h

```
#ifndef B_H
#define B_H

void b();

#endif
```

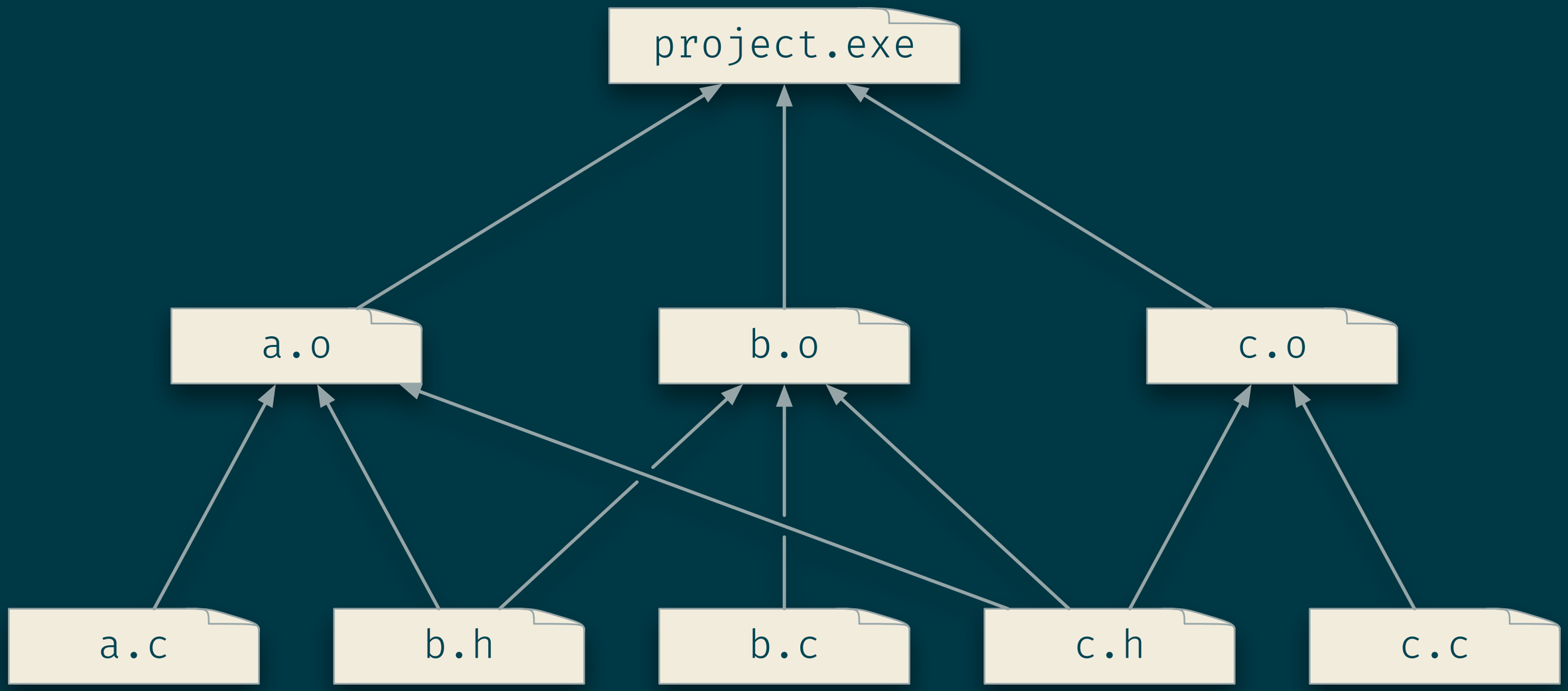
c.h

```
#ifndef C_H
#define C_H

void c();

#endif
```

Compiling Large Programs



- Can we avoid manually compiling each file and calling the linker?
- When we change only some files, can we avoid recompiling all files?

Make

Manages compilation and linking of multi-file projects.

Needs a `Makefile` that specifies:

- Dependencies (which files are needed to build which output file?)
- Rules how to produce output files from input files

Make Rules

```
# Link a.o, b.o, c.o to produce project.exe
project: a.o b.o c.o
	gcc -o $@ $^

# Recompile a.c every time a.c, b.h or c.h changes
a.o: a.c b.h c.h
	gcc -o $@ -c $<

# Recompile b.c every time b.c, b.h or c.h changes
b.o: b.c b.h c.h
	gcc -c $<

# Recompile c.c every time c.c or c.h changes
c.o: c.c c.h
	gcc -c $<
```

Make Special Rules

```
# Phony targets are built no matter whether a file
# with this name exists
.PHONY: all clean

# “make all” builds project.exe and project2.exe
all: project.exe project2.exe
# No recipe because this is only about forcing
# the dependencies to be built

# Delete all generated files
clean:
        rm -f project.exe project2.exe a.o b.o c.o
```

Make Variables

```
PROGRAMS: project.exe project2.exe

PROJFILES=a.o b.o \
          c.o

PROJ2FILES=a2.o b2.o

all: $(PROGRAMS)

project.exe: $(PROJFILES)
        gcc -o $@ $^

project2.exe: $(PROJ2FILES)
        gcc -o $@ $^
```

A Complete Makefile

```
CFLAGS=-Wall -std=c99 -O2
LDFLAGS=-O2

PROGRAMS: project.exe project2.exe

PROJFILES=a.o b.o c.o
PROJ2FILES=a2.o b2.o
OBJFILES=$(PROJFILES) $(PROJ2FILES)

.PHONY: all clean

all: $(PROGRAMS)

clean: $(PROGRAMS) $(OBJFILES)
```

A Complete Makefile

```
project.exe: $(PROJFILES)
	gcc -o $@ $(LDFLAGS) $^

project2.exe: $(PROJ2FILES)
	gcc -o $@ $(LDFLAGS) $^

a.o: a.c b.h c.h
	gcc -c $(CFLAGS) $<

b.o: b.c b.h c.h
	gcc -c $(CFLAGS) $<

c.o: c.c c.h
	gcc -c $(CFLAGS) $<

# ... and more rules to build a2.o and b2.o
```

Invoking Make

Build all programs

```
$ make all
```

Clean all output files and rebuild from scratch

```
$ make clean && make all
```

Suppress make's output

```
$ make -s
```

Suppress specific output in the Makefile

```
project.exe: $(PROJFILES)
        @echo "Linking project.exe"
        gcc -o $@ $(PROJFILES)
```