CSCI 331:
Introduction to Computer Security

Lecture 4: C wrap-up

Instructor: Dan Barowy

Williams

Announcements

- Congratulate your classmates Jihong Lee and Atlas Yilmaz, your new CoSSAC representatives!

Topics

Pointers

Makefiles

Static vs shared libraries

Your to-dos

1. Lab 1 **out**.
   i. Note that it includes some reading.
   ii. Lab 1 **due Sunday 9/26** by **11:59pm**.
   iii. If your RPi is not set up, what are you waiting for?
2. Reading response 2 (Schneier) **due Wed, 9/22**.
3. Keep on reading *The Cuckoo's Egg*.

Quiz

Quiz solution: caveat

The C specification says **nothing** about the **location** of a variable.

The words **stack** and **heap** literally **do not appear** in the document.

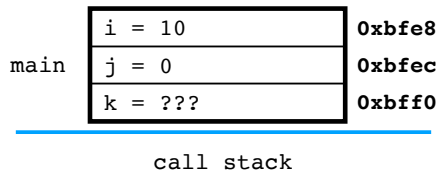
It only says how short-lived (**automatic**) and long-lived (**allocated**) storage should **behave**.

Virtually every compiler uses the **stack** for **automatic** variables, and the **heap** for **allocated** variables.

Practically, it **does not matter where** you put your variables as long as you put them in **stack** and **heap** locations as appropriate.

Quiz solution: after step 1

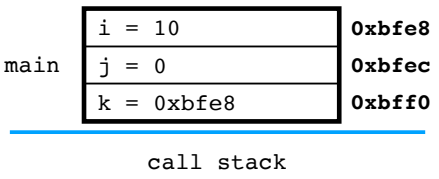
```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: after step 2

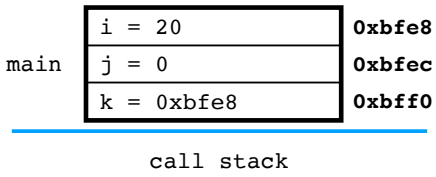
```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: after step 3

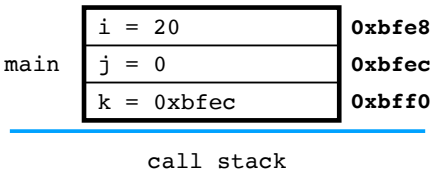
```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: after step 4

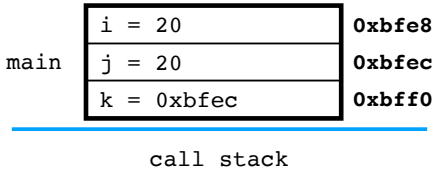
```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```



(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: after step 5

```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```

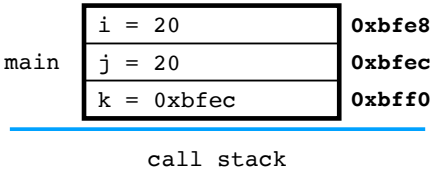


(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: print output

printf prints “i = 20, j = 20, *k = 20”

```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```

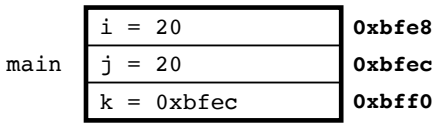


(state **just before** the line indicated by the **arrow** is executed)

Quiz solution: static data?

Yes. "i = %d,\nj = %d,\n*k = %d\n"

```
#include <stdio.h>

int main() {
  int i = 10, j = 0, *k;
  k = &i;
  *k = 20;
  k = &j;
  *k = i;
  printf("i = %d,
          j = %d,
         *k = %d\n",
         i, j, *k);
  return 0;
}
```



call stack

(state **just before** the line indicated by the **arrow** is executed)

How might you verify my solution?

gdb

Makefiles

Makefiles

A `Makefile` is a **specification** used by the `make` tool to **automate** the compilation of programs.

Rationale

Programmers build software **frequently**.

Lazy
(don't want to retype)

Impatient
(don't want to wait for gcc)

Insight

An entire project does not need to rebuilt every time.

a.c

```
int foo (int a) {
  return a + 1;
}
```

b.c

```
int bar (int b) {
  return b - 1;
}
```

depends on

depends on

c.c

```
#include <stdio.h>
#include "a.h"
#include "b.h"

int main() {
  int c = bar(foo(1));
  printf("c = %d\n", c);
  return 0;
}
```

Insight

An entire project does not need to rebuilt every time.

a.c

```
int foo (int a) {
  return a + 1;
}
```

b.c

```
int bar (int b) {
  return b - 1;
}
```

depends on

depends on

c.c

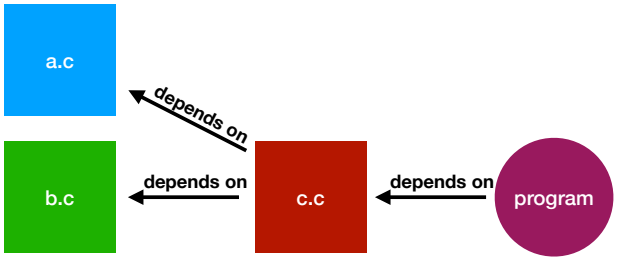
```
#include <stdio.h>
#include "a.h"
#include "b.h"

int main() {
  int c = bar(foo(2));
  printf("c = %d\n", c);
  return 0;
}
```

make a change

`a.c` and `b.c` do not change.

Do we really need to rebuild them?

A `Makefile` encodes dependencies



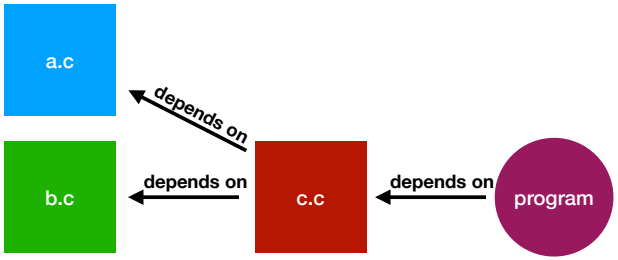

```
$ gcc a.c b.c c.c -o program
```

Small catch: `make` can only avoid rebuilding if there is a **produced thing** that it can avoid rebuilding.

There is only one **produced thing** here: `program`

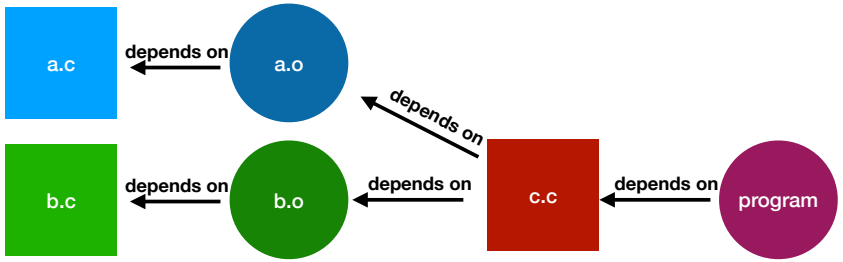
(produced things are circles; source files are squares)

A `Makefile` encodes dependencies



Fix: make more **produced things**.

A `Makefile` encodes dependencies

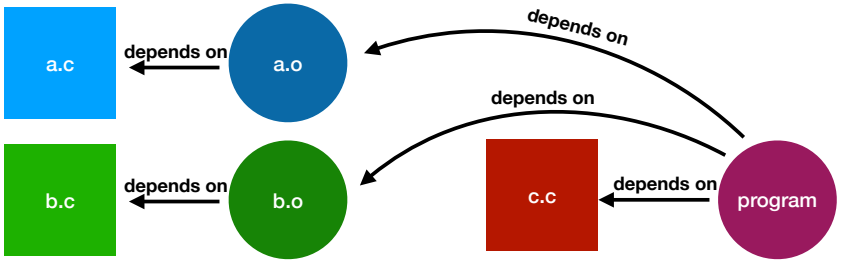


Fix: make more **produced things**.

This still has a problem.

`c.c` is not a **produced thing**.
Only **produced things** can depend on other things.

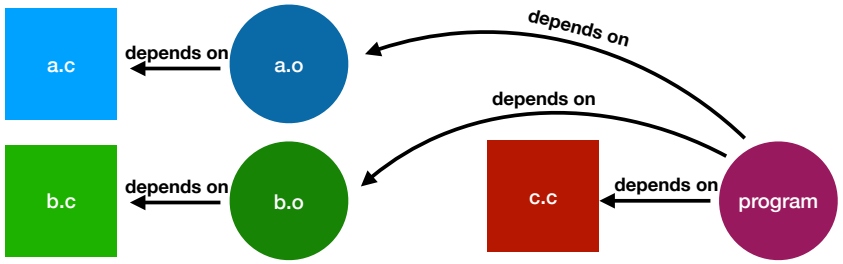
A `Makefile` encodes dependencies



Fix: make program depend on `a.o` and `b.o`.

Observe: The same amount of work is being done. But the **things** are **smaller**.

A `Makefile` encodes dependencies



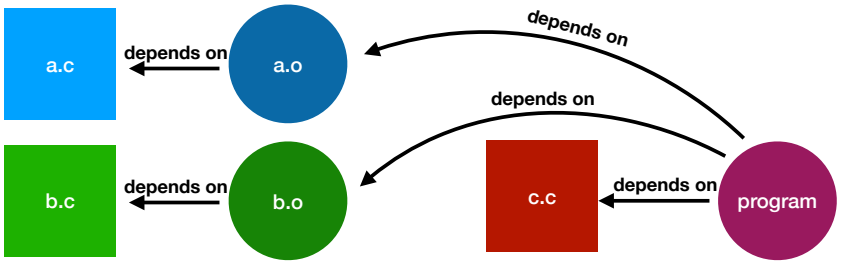
Suppose we update `c.c`.

What needs to be rebuilt?

Just `program`.

We don't need to rebuild `a.o` or `b.o` at all.

A `Makefile` encodes dependencies

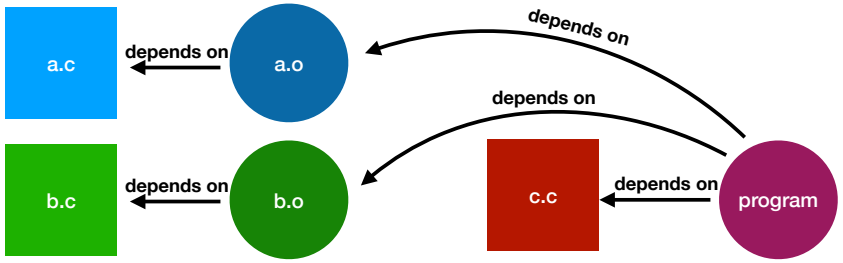


Let's write a `Makefile` for this, starting with `program`.

```
program: c.c b.o a.o
tab→ gcc -o program c.c b.o a.o
```

3 **things**, 3 **rules**.

A `Makefile` encodes dependencies



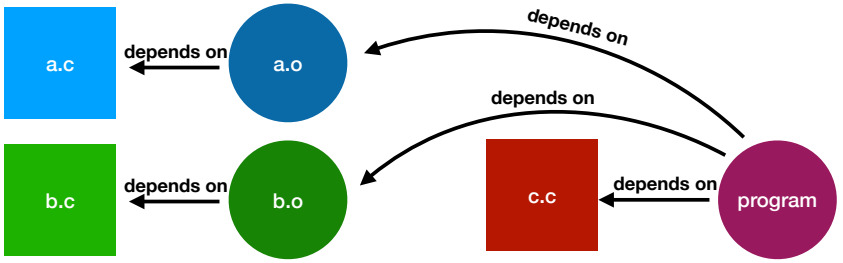
Let's write a `Makefile` for this, starting with `program`.

```
program: c.c b.o a.o
tab→ gcc -o program c.c b.o a.o
```

```
b.o: b.c
tab→ gcc -c b.c
```

3 **things**, 3 **rules**.

A `Makefile` encodes dependencies



Let's write a `Makefile` for this, starting with `program`.

```
program: c.c b.o a.o
tab→ gcc -o program c.c b.o a.o
```

```
b.o: b.c
tab→ gcc -c b.c
```

```
a.o: a.c
tab→ gcc -c a.c
```

3 **things**, 3 **rules**.

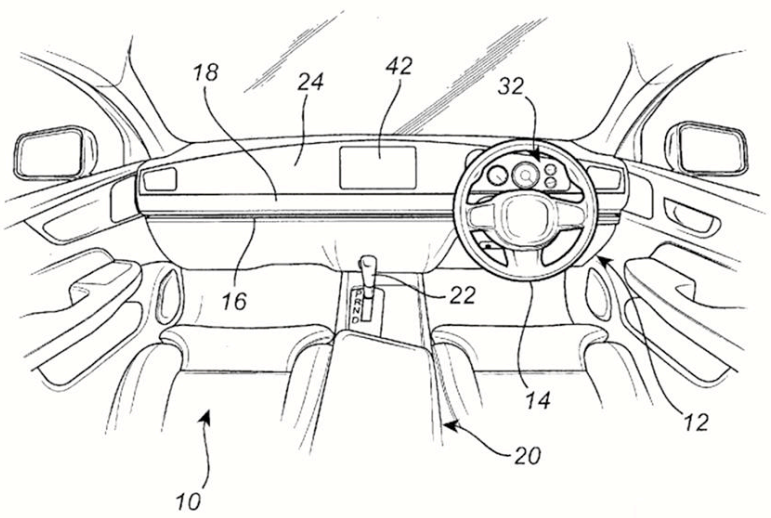
`Makefile` syntax

```
program: c.c b.o a.o
tab→ gcc -o program c.c b.o a.o
```

```
target: dep1 … depn
tab→ command
```

`command` should produce `target`.

What are `.h` files?



What are `.h` files?

A `.h` file provides **interface** information so that a compiler can **separately compile** sources.

a.c

```
int foo (int a) {
  return a + 1;
}
```

a.h

```
int foo (int a);
```

Should we put `.h` files in our `Makefile`?

Ask yourself: “if a file changes, should I rebuild?”

Answer: yes! If an interface changes, we should recompile.

Activity

```
login0: console.o database.o login.c
  gcc -o login0 console.o database.o login.c

console.o: console.c console.h
  gcc -c console.c

database.o: database.c database.h
  gcc -c database.c
```

1. Draw the dependence graph for this `Makefile`.
2. Assume that the project is built with `make login0`, `database.h` is then updated, and then the user types `make login0` again. What commands are run?

Libraries: static vs shared

Libraries: static vs shared

Static libraries are copied into program.

Shared libraries leave a “forwarding address”.

Static library: `library.o`

Shared library: `library.so`

Shared libraries must be linked with the `-l<libraryname>` linker flag for `gcc`.

Recap & Next Class

Today we learned:

- Stack layouts
- Makefiles
- Static vs. shared libraries

Next class:

- Pseudoterminals
- Password security