PURDUE UNIVERSITY®

CS 240: Programming in C

Lecture 25: Core Files and goto
Makefiles
Networking

Prof. Jeff Turkstra

Announcements

- Course Evaluations Available
- Homework 13 Extra Credit
- Hardware lecture next Monday will be mostly video
  - Includes interview with campus squirrel!
  - I'll be here for questions, though

Final Exam

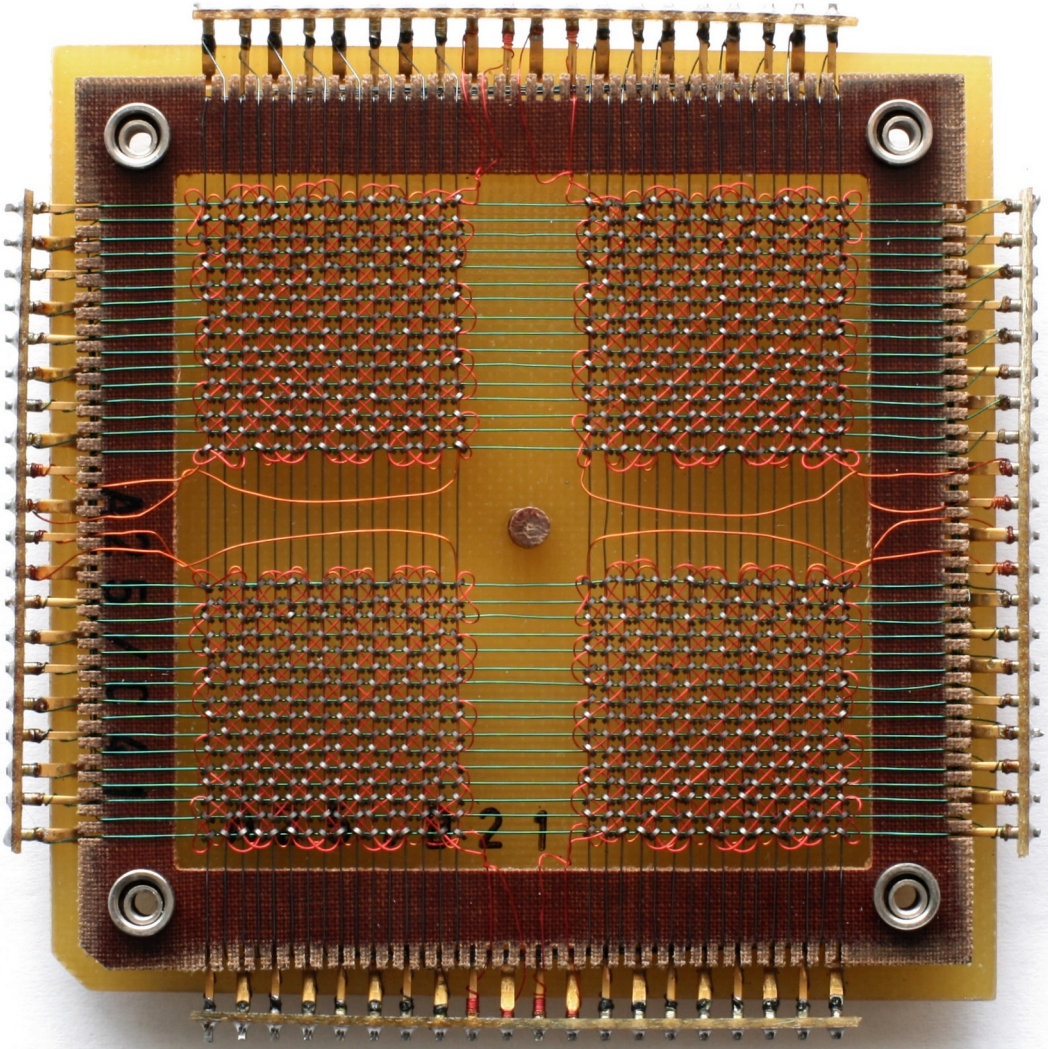
- Thursday, May 8
- 10:30am – 12:30pm
- Check the seating chart
  - Available sometime before the exam
- Coding, short answer, multiple choice, true/false

Security

- You have a very, very small taste of what kind of problems can arise in terms of program security
- We've only touched the "tip of the iceberg" in terms of buffer overflows
- If you want to know more, check out:
  - https://turkeyland.net/projects/overflow/
  - ...or find "Smashing the Stack for Fun and Profit" using a search engine
- There are many, many other types of vulnerabilities
- If you enjoy this stuff, take a security course!

Core Files

- Does anyone know what “core” memory was?
- When your program has an unrecoverable error, the operating system saves the heap/stack memory at the exact time of the failure into a file named “core”.
- You can use the core file with the debugger



Core dump file

- $ man 5 core
- May have to enable it (e.g., on data.cs.purdue.edu)
  - $ ulimit -c unlimited

The Official Disclaimer with respect to “goto”

- "For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so."
  --Edsger W. Dijkstra, March 1968, Comm. of ACM,
  "Go To Statement Considered Harmful"

Why is goto bad?

- Dijkstra made the case that goto was harmful for the following reasons:
  - It prevents the compiler from being able to make “nice” computer-sciency reductions of the program
  - It makes your code unreadable
  - It is really not necessary
    - You can always rewrite to have the same functionality without goto

Why does C have a goto?

- Because…
  - The compiler doesn't have any more difficulty analyzing a program with gotos in it
  - It often makes the program clearer to read
  - It is very useful – at a certain level, at least
- Contradictions?
- More enlightened languages have even more dangerous control flow operations

What does goto look like?

- You can define labels and goto those labels…

```
int func(int x) {
  int sum = 0;

 again:
  sum = sum + x;
  x = x – 1;
  if (x <= 0)
    goto get_out;
  else
    goto again;
 get_out:
  return sum;
}
```

How can goto make a program clearer to read?

- When you really need to ditch the control flow of your program and take drastic measures:

```
start_over:
 for (int x = 0; x < 5000; x++) {
   ptr = array[x];
   while (ptr->val < level) {
     while (ptr->next != 0 && ptr->val < level) {
       if (ptr->total == 0) {
         level++;
         goto start_over;
       }
     }
     sum += ptr->total;
   }
 }
}
```

When is goto useful?

- When it is necessary to break out of deeply nested loops (previous example)
- When you're building a state machine in software

- In general, you should still avoid using gotos unless there is a really good reason

Makefile

- Simple way to help organize code compilation
- Composed of rules
  - Target – usually a file to generate
    - Can be an action (“make clean”)
  - Prerequisites – used to create the target
  - Recipe – action to carry out
    - Must start with a tab!

```
gcc -o hello hello.c hellofunc.c -I.
```

Simple, hard coded

```
hello: hello.c hellofunc.c
    gcc -o hello hello.c hellofunc.c -I.
```

Or...

```
CC=gcc
CFLAGS=-I.

hello: hello.o hellofunc.o
    $(CC) -o hello hello.o hellofunc.o $(CFLAGS)
```

More generic

```
CC=gcc
CFLAGS=-I.
DEPS = hello.h

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

hello: hello.o hellofunc.o
    gcc -o hello hello.o hellofunc.o -I.
```

More Variables

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ=hello.o hellofunc.o


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


hello: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)
```

```
.PHONY: clean

clean:
    rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Lot's More

- https://www.gnu.org/software/make/manual/html_node/index.html

Bubble Sort

- Sorting is a big part of computer science
- Lot's of different ways with different performance/complexity
  - More in CS 251
- Bubble sort is one approach to sorting

Bubble Sort

```
void bubble_simple(int *arr, int size) {
  for (int k = 0; k < size; k++) {
    for (int i = 1; i < size; i++) {
      if (arr[i-1] > arr[i]) { // swap
        arr[i-1] ^= arr[i];
        arr[i] ^= arr[i-1];
        arr[i-1] ^= arr[i];
        print_array(arr, ARRAY_SIZE);
      }
    }
  }
}
```

Better

```
void bubble(int *arr, int size) {
  int swapped = 1;

  while (swapped) {
    swapped = 0;
    for (int i = 1; i < size; i++) {
      if (arr[i-1] > arr[i]) { // swap
        int tmp = arr[i-1];
        arr[i-1] = arr[i];
        arr[i] = tmp;
        swapped = 1;
        print_array(arr, ARRAY_SIZE);
      }
    }
  }
}
```

Networking Basics

- Given a network and a set of systems, how do we actually send data between a subset of systems?
  - First have to find the system(s)
    - Establish a route
  - Decide how we're going to communicate
    - Protocol
  - Establish a connection
    - Socket

Internet protocol

- IP is an addressing and fragmentation protocol
- Breaks communication into chunks (packets)
- Routes packets from a source to a destination
- Inherently unreliable
  - No guarantee anything will make it
  - No acknowledgments
- Different versions (IPv4 vs. IPv6)

IP address

- Each host or system has at least one IP address
  - nnn.nnn.nnn.nnn – dotted decimal notation, 4 bytes, 32 bits
  - 128.10.116.31
- Packets are sent from/to IP addresses
  - May traverse multiple routers
- Public/private
- NATs

Domain Name System

- But what about google.com?
- DNS is a distributed system that resolves host names to IP addresses
  - Hierarchical
    - Root servers
  - Many authoritative servers
  - Name resolution can involve multiple queries
  - Caching servers
- Reverse lookups

Transmission Control Protocol

- Remember, IP is unreliable
- TCP builds on IP to create a reliable network connection (often referred to as TCP/IP)
- Supports acknowledgments and retransmission
- Hosts identified by IP address and a port number

Clients and servers

- A server is a process that waits for a connection
- A client is a process that connects to a server
- Processes can be both!
- Machines can have multiple servers and clients running at once
- Once connected, clients and servers read and write data from/to each other much like a file

Networking Basics

- Given a network and a set of systems, how do we actually send data between a subset of systems?
  - First have to find the system(s)
    - Establish a route
  - Decide how we're going to communicate
    - Protocol
  - Establish a connection
    - Socket

Internet protocol

- IP is an addressing and fragmentation protocol
- Breaks communication into chunks (packets)
- Routes packets from a source to a destination
- Inherently unreliable
  - No guarantee anything will make it
  - No acknowledgments
- Different versions (IPv4 vs. IPv6)

IP address

- Each host or system has at least one IP address
  - nnn.nnn.nnn.nnn – dotted decimal notation, 4 bytes, 32 bits
  - 128.10.116.31
- Packets are sent from/to IP addresses
  - May traverse multiple routers
- Public/private
- NATs

Domain Name System

- But what about google.com?
- DNS is a distributed system that resolves host names to IP addresses
  - Hierarchical
    - Root servers
  - Many authoritative servers
  - Name resolution can involve multiple queries
  - Caching servers
- Reverse lookups

Transmission Control Protocol

- Remember, IP is unreliable
- TCP builds on IP to create a reliable network connection (often referred to as TCP/IP)
- Supports acknowledgments and retransmission
- Hosts identified by IP address and a port number

Clients and servers

- A server is a process that waits for a connection
- A client is a process that connects to a server
- Processes can be both!
- Machines can have multiple servers and clients running at once
- Once connected, clients and servers read and write data from/to each other much like a file

Network sockets

- Clients and servers communicate via sockets
- A socket is an endpoint for sending and receiving data
- A socket address consists of the IP address and port number
  - At least for TCP
- Port number is 16-bits (0-65535)
  - Ports < 1024 are privileged
- 128.10.116.31:80

Server

- man 7 ip
- Steps for listening...
  - Create a socket()
  - bind() that socket to an address and port
  - listen() for a connection
  - accept() the connection
  - [communicate – read, write, recv, send]
  - close() the connection

socket()

- Create an endpoint for communication

  int socket(int domain, int type, int protocol);

bind()

- “assigning a name to a socket”

  bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- socket() gives us a fd
- bind() assigns an “address” to the socket

```
struct sockaddr_in {
  sa_family_t   sin_family; /* address family: AF_INET */
  in_port_t     sin_port;   /* port in network byte order */
  struct in_addr sin_addr;  /* internet address */
};
```

listen()

- Marks the socket as a passive socket
  - Accepts incoming connections

```
int listen(int sockfd, int backlog);
```

  - backlog is the maximum length for the pending connections queue
    - Max number of waiting connections

accept()

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Removes first connection request from pending connections queue
  - Must be called on a listen()ing socket
  - addr is filled with peer information
- Creates a new connected socket
  - This socket does not listen
  - Original socket is left alone

recv() and send()

- recv() is read() when flags = 0
  - Can behave differently depending on the flag(s)
- send() is write() when flags = 0
  - Can behave differently depending on the flag(s)

close()

- You should always check the return value of close :-)

Client

- Steps for connecting…
  - Create a socket()
  - Optionally bind()
    - For a specific source port
  - connect() to an address:port
  - [communicate – read, write, recv, send]
  - close() the connection

connect()

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- Connects sockfd to address addr

server.c

- Simple echo server
- Can connect using telnet
  - Consider writing your own client – it's not that hard!

Boiler Up!