Discussion Week 1

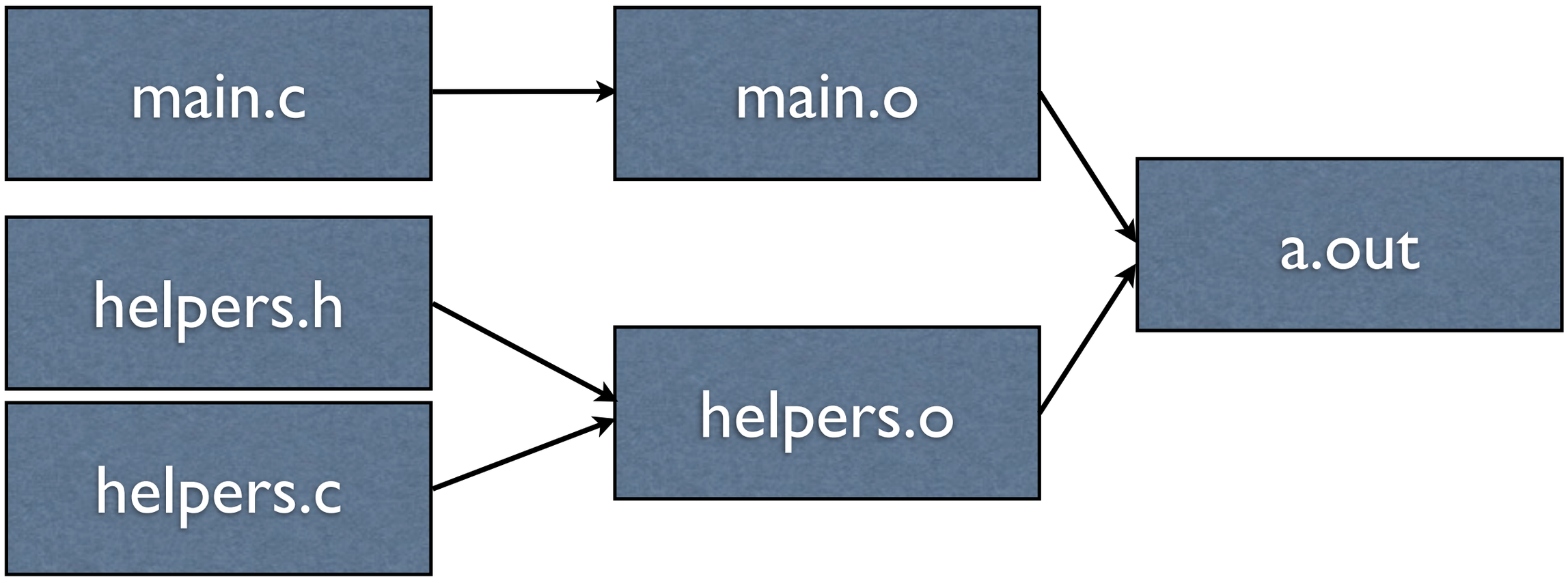
TA: Kyle Dewey

Project 0 Walkthrough

Makefiles

What?

- A programmable command that can generate new files based on existing ones
- Only that which is needed is made



Why?

- The standard "`gcc *.c`" or "`javac *.java`" scales poorly
  - Everything recompiled
  - Cannot handle directory hierarchy
- Arbitrary builds may need an arbitrary sequence of commands

Basics

- Makefiles consist of a series of rules
- Each rule has optional dependencies
- The first rule is the default

```
rule_name: target1 target2
  how to build output
```

Basics

- Dependencies can be either rule names or file names
- The process recursively follows dependencies

Macros

- Macros can be used to define common strings and utilities

```
MACRO_NAME = definition
```

Example

Including

- Makefiles can reference other makefiles
  - Common rules
  - Common macros

```
include ../Makefile
```

NACHOS Makefiles

C++ as it applies to NACHOS

Recommendation

- `c++.pdf` **in the** `c++example` **directory** is an excellent tutorial
- A bit dated, but applicable to NACHOS

What is Not Seen

- Templates
- Polymorphism
- Inheritance
- References

Header Files

```
#ifndef FILE_H
#define FILE_H
// code
/* more code
 * some more code */
#endif
```

Class Definition

```
class MyClass {
  public:
    MyClass();
    int doSomething( int x );
  private:
    int privateFunction();
    int privateVariable;
};
```

Class Definition

- Generally, class definition should be done in header file
- The header file defines the interface to the class

Class Implementation

- Classes should generally be implemented in C++ code files (`.cpp`, `.c++`, `.cc`...)
- NACHOS uses the "`.cc`" extension

Class Implementation Example

```
#include “MyClass.cc”

MyClass::MyClass() :
 privateVariable( 5 ) {}
int MyClass::doSomething( int x ) {
 return x + 1; }
int MyClass::privateFunction() {
 return privateVariable * 2; }
```

Memory Management

- C++ lacks a garbage collector
- Classes have user-defined destructors that specify how to perform such cleanup
- Destructor for "`MyClass`" has the method signature"`~MyClass()`"

Instantiating a Class

- On the stack:
  - `MyClass foo( 5 );`
- On the stack (no-arg constructor):
  - `MyClass foo;`
- On the heap:
  - `MyClass* foo = new MyClass( 5 );`
  - `MyClass* foo = new MyClass();`

Destructing an Instance

- On the stack, once a class goes out of scope, the destructor is automatically called
- On the heap:
  - `delete foo;`
  - "`foo`" is a pointer to the class instance

Instantiating an Array

- On the stack:
  - `int foo[ 5 ];`
  - `int foo[] = { 0, 1, 2, 3, 4 };`
- On the heap:
  - `int* foo = new int[ 5 ];`

Destructing an Array

- Performed automatically for once out of scope for arrays on the stack
- On the heap:
  - `delete[] myArray;`
  - "`myArray`" is a pointer to the array

Destructing an Array

- There is only a single dimensional "`delete[]`" operator
- For a two dimensional array "`myArray`" of size "`size`":

```
for( int x = 0; x < size; x++ ) {
  delete[] myArray[ x ];
}
delete[] myArray;
```

`code/threads/list`

Example from NACHOS

Assembly (Time Permitting)

Registers

- Programs need to use processor registers in order to execute

Registers

Process #100	
Register	Value
A	1
B	2
C	3

Process #101	
Register	Value
A	30
B	40
C	50

Swapping In

- State of registers is copied from memory to the registers
- Process resumes execution with the restored register values

Swapping Out

- The process' execution is paused
- The values of the registers is saved to memory

Unportable

- The need to deal directly with registers prevents the usage of portable, high-level language code
- Assembly must be used

switch.s