Why choose C?

• Compiler emits machine instructions
• More “realistic” machine model
• Explicit control
• Yields better performance...
• ...if you know what you’re doing
The plan for today

• Syntax and so forth
• Enough to get you started
• Help you avoid common mistakes
• But: only real way to learn is to do
• You will be doing plenty of Googling
A simple program

/* This program doesn’t actually do anything. */
int
main( int argc, char** argv )
{
    return 0;
}

Compile with a command such as
$ cc –std=c99 –o test test.c

And then run it with
$ ./test
$
A simple program

```c
#include <stdio.h>

int main( int argc, char** argv )
{
    printf( "Hello, world!\n" );
    return 0;
}

$ ./test
Hello, world!
$
A simple program

#include <stdio.h>

int main( int argc, char** argv )
{
    // Report favorite number
    int fav_num = 42;
    printf( "My favorite number is %d.\n", fav_num );
    return 0;
}

$ ./test
My favorite number is 42.
$
Printing to standard output

• Great, easy way to figure out what’s actually happening (debugging)
• Usually bugs result in divergence between actual and expected program state
• Strings: %s, floats: %f, unsigned ints: %u

printf( “Beginning some_func(): foo = %d, bar = %d\n”,
foo, bar );
Using the preprocessor

- `printf()` gets messy!
- Manually adding and removing debugging statements sucks

```
#define DEBUG

#ifdef DEBUG
printf( "Beginning some_func(): foo = %d, bar = %d\n", 
        foo, bar );
#endif
```

- Compile-time, unlike if()
- Also --DSOME_SYMBOL compiler option
Control flow

• Most of this will carry over from Java

• for(int i = 0; i < n; ++i) stmt;
• while(condition) stmt;
• if(condition) stmt; else stmt;

• Anything nonzero is “true”
Primitive data types

- No boolean (why?)
- Generic int types, char
- If you need an int of certain size
  - #include <stdint.h>
  - Use intN_t or uintN_t for N in \{8,16,32,64\}
- Same floating-point types (float, double)
Composite data types

- NOT objects
- How do you think this is implemented?

```c
struct user_record
{
    int user_id;
    char* name;
};

struct user_record u;
u.user_id = 503;
struct user_record* uptr = &u;
uptr->user_id += 1;
```
What is memory?

• Linear array of memory locations
• 32-bit or 64-bit address space
• Sort of like a huge array
• What does memory mean?
Pointers and memory addresses

• A pointer is just a memory address
  – 32-bit or 64-bit number (depending on arch.)
  – just an index into the array of memory
• Syntax is type* (means “pointer to type”)
• Operator to get pointer to something

```c
int* a = &b; /* b is an int */
```

• Operator to get value at memory location

```c
b = *a;
```
Memory management

• No garbage collection
• Everything is explicit!
• Stack space is allocated when fn is invoked
• Ask for heap memory via malloc()
• Release it via free()
• One allocation requires exactly one deallocation
Stack vs. heap memory

• Stack: small, local-scope
  – Automatically deleted when it goes out of scope
  – Much more efficient
  – Must be careful not keep stale pointers

• Heap: big, or you want it to stick around
Arrays

- Syntax is similar to Java
- Data type is really a pointer to first element
- No length, bounds checking, etc.
- Elements accessed as a[i]
- Syntactic sugar for memory access

```c
int a[n]; // type of a is just int*

j = a[i];
j = *( a + i );
```
What is a string?

- In Java: an object
- In C: a pointer to a char
- String continues until hitting a null character
  - Has integer value zero
- Consequences
  - Finding length is $O(n)$
  - Strings can’t store arbitrary binary data
A second look at main

• Function prototype is

```c
int main( int argc, char** argv )
```

• This passes in a list of strings (the command-line arguments)

• Now you tell me: why is this the appropriate prototype?
Types and casting

• Remember: memory is just an array of values

• Syntax is

```c
char* foo;
...
int* bar = (int*) foo;
```

• You’ll have to do this frequently
  – `malloc()` always returns a `void*`
Passing by value or pointer

• Pass-by-value copies
• Pass-by-pointer copies... the pointer
• Actual data is NOT copied!
Gotchas

- Initialize all variables and memory!
- Don’t dereference null pointers
- Don’t keep stale pointers
Linkage

• How headers work
• Why headers & source are separate
• External keyword