Smallest program with output

```c
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

Better:
- Outputs to stdout using printf, defined in stdio.h
- Returns EXIT_SUCCESS, defined in stdlib.h

Why C?

- Powerful, efficient access to hardware
  - Pointers!
- Crazy fast, no interpreter nonsense
- Compiles on almost every piece of hardware ever
  - Core "ANSI C" is portable
  - If you use OS calls, you probably aren't portable, but that's justified
- C is close to the hardware and OS, so code must differ between architectures — no abstraction to help/hinder you

Syntax (1)

- Whitespace is irrelevant
- Comments:
  ```c
  /* anything, including newlines */ */
  // anything to end of line
  ```
  - Really a C++ comment, but ok in modern C
- Literals:
  - Numbers: 5, 25, 0xFF, 0777
  - Characters: 'c', 'b', '
', '\0'
  - Strings: "This is an example", "So is this"
  - More on this later...
- Include directives:
  ```c
  #include <sys/file.h>  // system file
  #include "myfile.h" // our file
  ```

Smallest program

```c
int main(){}
```

Not very good:
- Doesn't do anything
- Doesn't specify what is returned

Syntax (2)

- Data types:
  - char, int, long, float, double
  - Size of int? long? It depends. Use sizeof(<TYPE>) to get byte size of a data type. Both int and long are 32-bit=4-byte for us.
  - Can put "unsigned" in front of an integer type:
    ```c
    int x=0xFFFFFFFF;  // This is -1
    unsigned int y=0xFFFFFFFF;  // 2^32-1 = 4294967295
    ```
### Syntax (3)
- Functions:
  ```c
  float magnitude(float x, float y) {
    return sqrt(x*x + y*y);
  }
  void printWlouO() { printf("Wlou\n") }
  ```
- Global variables, outside of functions:
  ```c
  int w=2;
  const float pt = 3.14159;
  int main() { w=5; }
  ```
- Local variables, inside of functions:
  ```c
  int main() { int w; }
  ```
- Data type conversions (casts):
  ```c
  int a; float f = (float)a; // f=0.000
  float g=5.9; int b = (int)g; // b=5
  ```

### Pointers (1)
- Pointers are simply **memory addresses**!
- Declaring a pointer:
  ```c
  int* ptr; // ptr refers to an int (4 bytes for us)
  ptr = NULL; // Set to NULL (null)
  ```
- Setting new pointers to NULL makes uninitialized pointers more obvious in the debugger; you can test them for likelihood in conditions.
- Using a pointer:
  ```c
  int* ptr = &x; // Now points to address x+4
  *ptr = 5; // Same as x+=5! WOW!!
  ```
- It is also the “Dereferencing operator” when prepended to pointers
  - Don’t bother trying to guess order of operations with this, too dangerous.
  - Just use parentheses on them all the time.
- Can have multiple levels of indirection:
  ```c
  int x;
  int* p = &x;
  int** pp = &p;
  (*pp)=5;
  ```

### Syntax (4)
- Operators:
  - Arithmetic: + * / % (mod)
  - Binary: & (xor) & (and) | (or) ~ (compliment)
  - Bit shifting: << >>
  - Logical: && (and) || (or) ~ (not)
  - Assignment: =
  - Comparisons: != == < <= >= >
  - Choice: ? : y
    - If b is true, then evaluate to x, else evaluate to y
  - Modifying:
    ```c
    *++ (increment after eval), ++* (increment before eval)
    ```
  - Slap a = after any operation to modify the LHS:
    ```c
    x = y; // SAME AS x += y;
    ```

### Pointers (2)
- Accessing a pointer’s reference:
  ```c
  int x; int* ptr = &x;
  (*ptr) = 5; // Same as x+=5! WOW!!
  ```
- * is also the “Dereferencing operator” when prepended to pointers
  - Don’t bother trying to guess order of operations with this, too dangerous.
  - Just use parentheses on them all the time.
- Can have multiple levels of indirection:
  ```c
  int x;
  int* p = &x;
  int** pp = &p;
  (*pp)=5;
  ```

### Syntax (5)
- Control (assume int i):
  - Conditional:
    ```c
    if (i>0) { printf("I>0d in range\n",i); }
    ```
  - Loops (each outputs “01234”):
    ```c
    for (i=0; i<5; i++) { printf("%d",i); }
    ```
  - Multicase conditional:
    ```c
    switch (i) {
    case 1:
      break;
    case 2:
      printf("i is 1 or 2\n");
      break;
    default:
      printf("i is neither 1 nor 2\n");
    }
    ```

### Arrays
- An array is just a pointer to a block of memory! Wow!!! Two kinds of arrays:
  - On the stack, either as a global or in a function as a local variable
    ```c
    float coords[3] = {1.0,2.0,3.0}; // Initialize
    ```
    - You must know the size at compile time
    - Memory is reclaimed automatically by the operation of the stack
  - On the heap...

### Arrays
Heap Memory Allocation

- Create a pointer:
  ```c
  int* values;
  ```
- Use a memory allocation call to reserve a block of heap memory of any size (even a runtime variable n):
  ```c
  values = (int*)malloc(n*sizeof(int));
  values = (int*)calloc(n,sizeof(int));
  ```
- If you want all the elements to be initialized, you have to do it yourself
  ```c
  for (i=0; i<n; i++) values[i]=0;
  ```
- (calloc might init block with zeros, but don't count on it)
- You can resize dynamically:
  ```c
  num *= 2; values = (int*)realloc(values,num);
  ```
- **YOU must free memory yourself!**
  ```c
  free(values);
  ```

What does main() accept?

- We've shown the simple main:
  ```c
  int main() {};
  ```
- To accept command line arguments, you'll need:
  ```c
  int main(int argc, char** argv) {
    You are given the number of arguments argc, and an
    array of strings argv
    argv[0] is always the name of your binary
    argv[1] is the first argument, argv[n] is the nth
    argument
    If argc==2, then argv has indices 0,1.
  ```

Strings

- What is a string?
  - Just an array of type char! WOW!
    ```c
    char strMax[64] = "This can't be more than 64
    chars ever!"
    char* strMax = malloc(64);
    strcpy(strMax,"Neither can this!");
    ```
  - Why not strMax = "Some stuff"?
    ```c
    strMax = malloc(strlen("Some stuff") + 1);
    ```
  - But the array is one size (64) and the string is
    another...how do we tell when the string ends?
    ```c
    - Null terminator: add a zero character to the end (written as '\0' usually)
      - Done automatically by any string functions, but if you roll
        your own string, don't forget it...!
    ```

Using Pointers as Arguments

- Normal function:
  ```c
  void f(int x) {} // Normal function
  ```
  - the value of x is given ("pass by value")
- How about:
  ```c
  void f(int x) { (*x)=5; }
  ```
- Now the function gets a pointer instead of
  the value ("pass by reference")
  ```c
  - The function can CHANGE the variable referred to:
    ```c
    int a=2; f(&a); printf("x=\%d\n",a);
    ```
  - Outputs "5"
- Parameters can now be thought of as input,
  output, or both

Structures (1)

- Can rename data types with typedef:
  ```c
  typedef char bool; // Make a bool type
  ```
- Can glob together data types to form a larger
  type:
  ```c
  struct Thing {
    int value; char buffer[64];
  };
  ```
- Access (given Thing t, Thing* tp=&t):
  ```c
  t.value = 5;
  (*tp).value = 5;
  tp->value = 5;
  ```

Structures (2)

- Declaring variables based on this is a bit ugly:
  ```c
  int struct thing t;
  ```
  - NOT just: thing t;
- Solution: combine struct with typedef:
  ```c
  typedef struct {
    int value;
    char buffer[64];
  } Thing;
  ```
- Can have pointers to these: Thing* obj;
  ```c
  - This is what objects are in OO languages!
- Custom data types often written as
  something_t
IO (1)

- Terminal output:
  - print(char* format, [varlist]);
  - Example:
    ```c
    int i=32; float x=2.5; char c='x';
    char buf[50] = "banana";
    printf("%x,dx%3.1f x%c buf="%s"\n", i, x, c, buf);
    ```
  - Output:
    ```
    $ cat ./io1.out
    32.0 x 2.5x banana
    $
    ```
- Terminal input:
  - scanf(char* format, [varlist]);
  - You pass POINTERS in varlist to things you want overwritten: int x; scanf("%d", &x);
  - Check the man page for details!

IO Remarks

- **MIND YOUR BUFFER SIZE!!!**
  - Why?
    - File in.dat has "abcdefg", opened as FILE* fp
    - Read word with:
      ```c
      char buf[5];
      fscanf(fp,"%s",buf);
      ```
    - OH MISERABLE DAY! You have just written data into memory you don't own. Prepare to crash.
    - Solution:
      ```c
      fscanf(fp,"%4s",buf);
      ```
    - Why 4 instead of 5? "0"
    - Same goes for fread, fwrite, etc. Read the docs!

IO (2)

- File IO
  - FILE* fp;
  - fopen(char* file, char* mode);
    ```c
    FILE* fp = fopen("filename.dat","r"); // Open for read
    ```
  - fscanf(FILE* fp, char* fmt, [varlist]);
  - feof(FILE* fp); // True if EOF
  - fwrite/ fread(void* buf, size_t size, size_t count, FILE* fp); // Binary read and write are symmetric!
  - fclose(FILE* fp);
  - Check a stdio.h reference for a full list!

The Preprocessor

- The preprocessor does text-level actions to your code based on directives starting with `'#'`:
  ```c
  #include <stdio.h>
  ```
  ```c
  #define PI 3.14159
  ```
  ```c
  #if DEBUG
  ```
  ```c
  #endif
  ```
  ```c
  ```
  - Take out <SOMECODE> unless DEBUG has been defined as a nonzero value
  - Put all of stdio.h right here
  - Replaced the token PI with 3.14159 in code

IO (3)

- Example: Sum all ASCII ints from in.dat and save as binary int in sum.dat
  ```c
  int x, sum;
  FILE* fp; // File handle, internals irrelevant
  if (fp = fopen("filename.dat","r")) { // Open for read
    while (fscanf(fp,"%d", &x) == 1) { // ASCII expressed ints
      sum += x;
    }
    fclose(fp); // Close file
  }
  ```
  ```c
  fp = fopen("sum.dat","wb"); // Open binary write
  fwrite(&sum, sizeof(int), 1, fp); // Write binary int
  fclose(fp); // Close file
  ```

Multi-file Projects

- Two modifiers to apply to globals and functions:
  - extern: Don't define it here, just declare it and note that it is in an *external* file.
  - static: Force it to stay in this file, nobody else can get at it (even with extern)
Multi-file Example

- File myCode1.c:
  ```c
  int a;
  const double pi = 3.1415;
  static int sum(int n, int m) {
    return (n+m);
  }
  ```
- File myCode2.c:
  ```c
  int b;
  extern const double pi;
  static double sum(double n, double m) {
    return(n+m);
  }
  ```

Headers Have the 3 D's

- **Declarations** of externally linked functions
- **Documentation** for those functions!
- **#Defines** of various flags and constants
  - Ex: Put a `#define DEBUG` in a common header, then surround all debug output code with `#if DEBUG` and `#endif`. You can choose to include or discard this code during compilation by changing one flag!
- **Nothing else! No code!**

Multi-file Example Comments

- The integers a and b have external linkage and may be seen in both files (if they are both part of the same program).
- The value of pi is defined to be external in myCode2. It will be the same memory location as pi is in myCode1.
- Both declarations of sum are valid, but the are limited in scope to their respective file.

Avoiding Header Collisions

- Two source files include the same custom header file that declares int x, and you compile both at the same time
  - You've just declared int x twice! Oh no!
- Use the preprocessor to make headers behave well. Example myfile.h:
  ```c
  #include <stdio.h>
  ifndef MYFILE_H
  #define MYFILE_H
  </HEADER STUFF>
  #endif
  ```

Header files (*.h)

- Need a better way than manual declarations
  - You'd have to declare every function you wanted to use out of stdio, for example.
- Solution: Header files!
  - `#include <stdio.h>` means "put all of stdio.h here"
  - stdio.h contains declarations of functions defined elsewhere and compiled into libraries

Compiling

- A single source file S1.c to a binary:
  ```bash
gcc -o MyBinary S1.c
  ```
- Add symbolic debug support to binary:
  ```bash
gcc -g -o MyBinary S1.c
  ```
- Compile two source files:
  ```bash
gcc -g -o MyBinary S1.c S2.c
  ```
- Include some library called potato:
  ```bash
gcc -g -l potato -o MyBinary S1.c S2.c
  ```
Compiling (2)

- But why recompile *every* file *every* time, even if you just change one?
- Compile to *object files*:
  
  ```
  gcc -g -c S1.c
  ```
  - Produces S1.o
- Combine all these .o files into a binary
  
  ```
  gcc -o MyBinary S1.o S2.o ...
  ```

Debugging: the problem

- A bad program called lousy.c:
  ```
  #include <stdio.h>
  int main() {
      int & ptr;
      (*ptr) = 5; // where does this go?
  }
  ```
- Compiles ok (C will never judge you!)
  ```
  $ gcc -g -o lousy lousy.c
  ```
- Run:
  ```
  $ ./lousy
  Segmentation fault
  ```

Makefiles

- Guess what has been changed and manually type all those commands?
  - I'm too lazy for that nonsense.
- Makefiles!
  - Create a graph of dependencies
    - "If X is built on X and X is new, then turn the new X into a new Y"
  - Format:
    - `<VARNAME> = <VALUE>`
    - `<THING> : {[DEPENDENCY] [DEPENDENCY] ...}
    - `<TAB>` = `<COMMANDS TO MAKE DEPENDENCIES INTO THING>`
  - Syntax:
    - make [thing to make, defaults to first in Makefile]

Debugging: gdb to the rescue

```
$ gdb lousy
$<WELCOME MESSAGE>
(gdb) run
<<SOME CODE>>
Program received signal SIGSEGV, Segmentation fault.
0xe048367 in main () at x.c:3
3 (*ptr)=5;
(gdb) list
1    int main() {
2      int & ptr;
3      (*ptr)=5;
4    }
(gdb) br
#0 0xe048367 in main () at x.c:5
(gdb) print ptr
$1 = (int *) 0xe048370
```

Prototype Makefile

```
# Comment out on Linux
#LIBS = -lsocket -lssl
LIBS = $1.o $2.o

all : MyBinary
  .c.o:
    gcc -g -c $<

MyBinary : $(LIBS)
  gcc $(LIBS) -o $0 $(LIBS)

clean:
  rm $(LIBS) MyBinary
```

```
• Type "make" to compile
• Type "make clean" to kill binaries and objects
```

Standard libraries (1)

- `stdio.h`
  - You've seen most of this: printf, fopen, etc.
- `stdlib.h`
  - A lot of very fundamental things, like memory allocation, string/number conversions, environment manipulation, and process execution
Standard libraries (2)

- string.h
  - String copying, concatenating, length, etc.
- time.h
  - Get current time, manipulate time values
- math.h
  - abs, cos, exp, log, pow, sin, tan, etc.

Any questions?

What about C++?

- All C programs are C++ programs, as C++ is an extension to C
- C++ loosens some restrictions, adds some syntactic niceness (such as declaring variables in the initialization of a for loop), and most importantly formalizes OO as part of the language.
- We aren’t doing C++, but some of the niceness has been carried back to C (my favorite being the // single line comments)

Warning:
You now know just enough C to be dangerous!

- I haven’t covered all of the language, only what you need to get going right now
- To learn more, you can:
  - Check out the CSC253 notes at http://courses.ncsu.edu/csc253/lec/001/, which I shamelessly stole from, since that’s where I learned most of this
  - Read about standard library calls at http://www.cplusplus.com/ref/
  - Get any good introductory C book
    - I recommend books published by O’Reilly
  - Read any number of C tutorials on the web