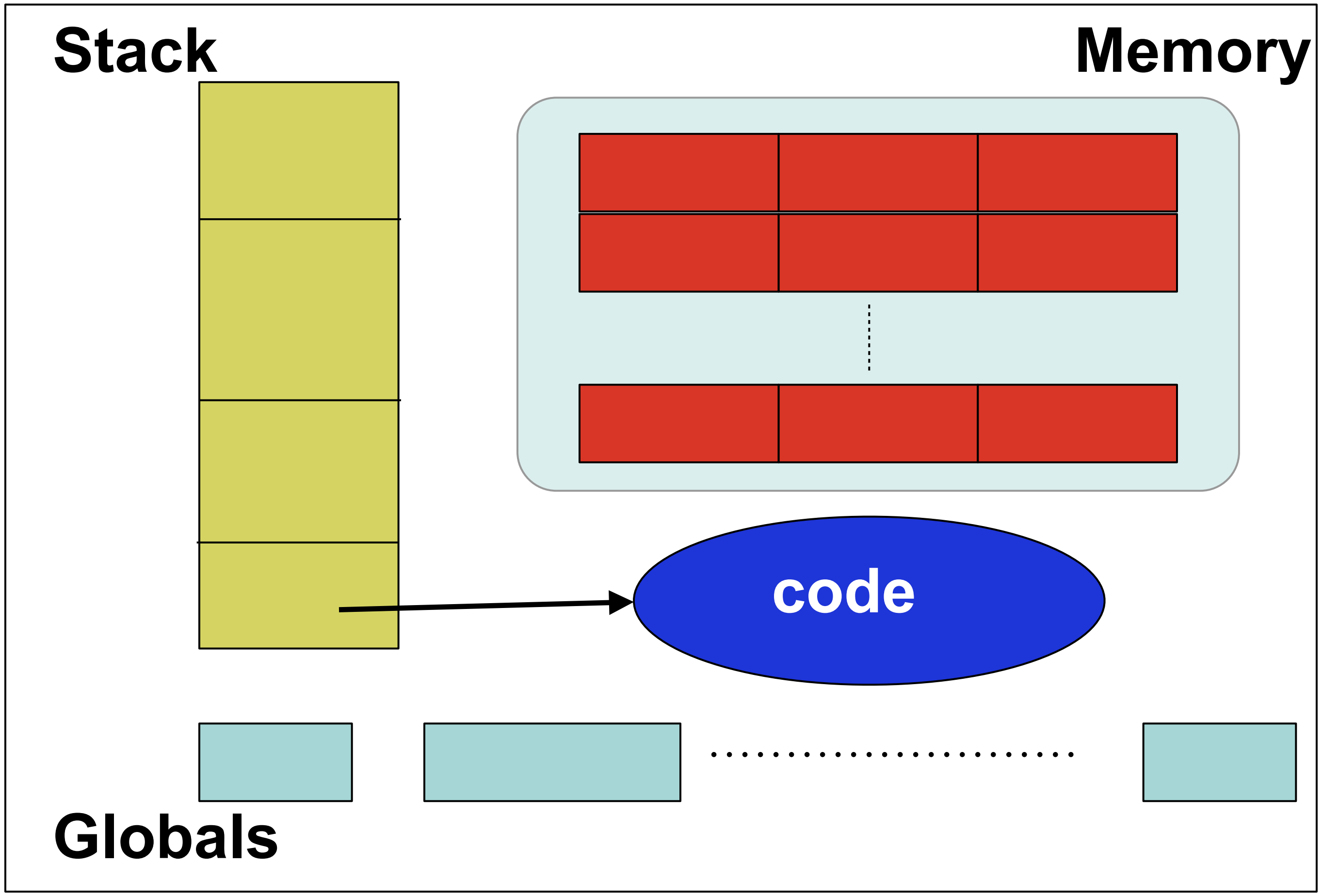
CS240: Programming in C

Lecture 2: Overview

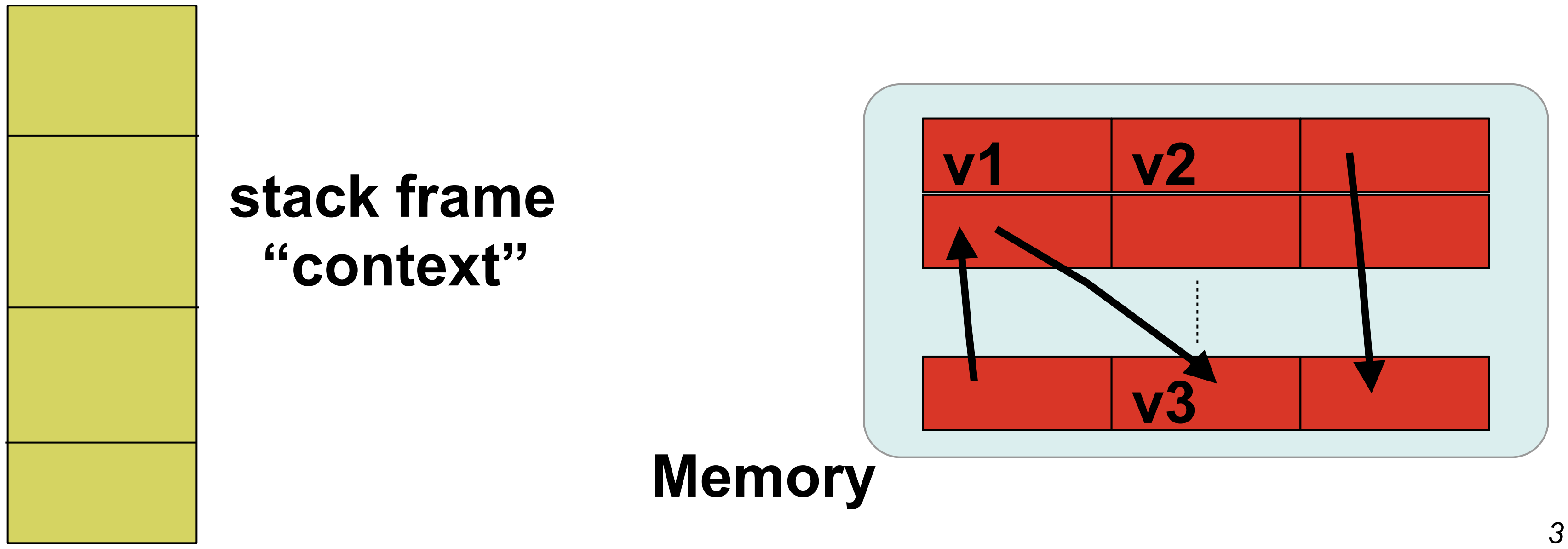
Programming Model

- How does C view the world?



Programming Model

- Execution mediated via a stack
  - function calls and returns
  - local variables
- Variables name memory and globals
  - contents of memory contain values or references (pointers) to other pieces of memory



Programming Model

- Imperative
  - Program execution defined by effects on memory and globals
  - Control-flow expressed primarily through:
    - loops and iteration
    - conditionals
    - “heavyweight” function calls
  - Variables are *containers* for values
    - x = 3 -- *x is a memory cell that holds 3; it’s not the same as 3*
    - *referential transparency (or lack thereof)*
  - Aggregate structures (records, arrays) define contiguous (potentially variable-sized) regions of memory

Types

- Abstractly, we can think of a type as a set of values and operations on those values
    - in Java, a type is a class that represents a set comprising the fields of the object and its methods
- In C, types have a much less rigid definition
    - a convenient way of reasoning about memory layout
    - all values (regardless of their type) have a common representation as a sequence of bytes in memory

Types

- Reflects underlying machine model
  - There are base types (int, char, ...)
  - There are *pointers* to base types (*int, *char,...)
  - There are pointers to pointers to base types (**int, **char, ...) and so on
- Arrays and structs provide a way of aggregating and naming regions of memory
- Functions have types that reflect their input/output behavior:

int f (char c[])

return type *argument type*

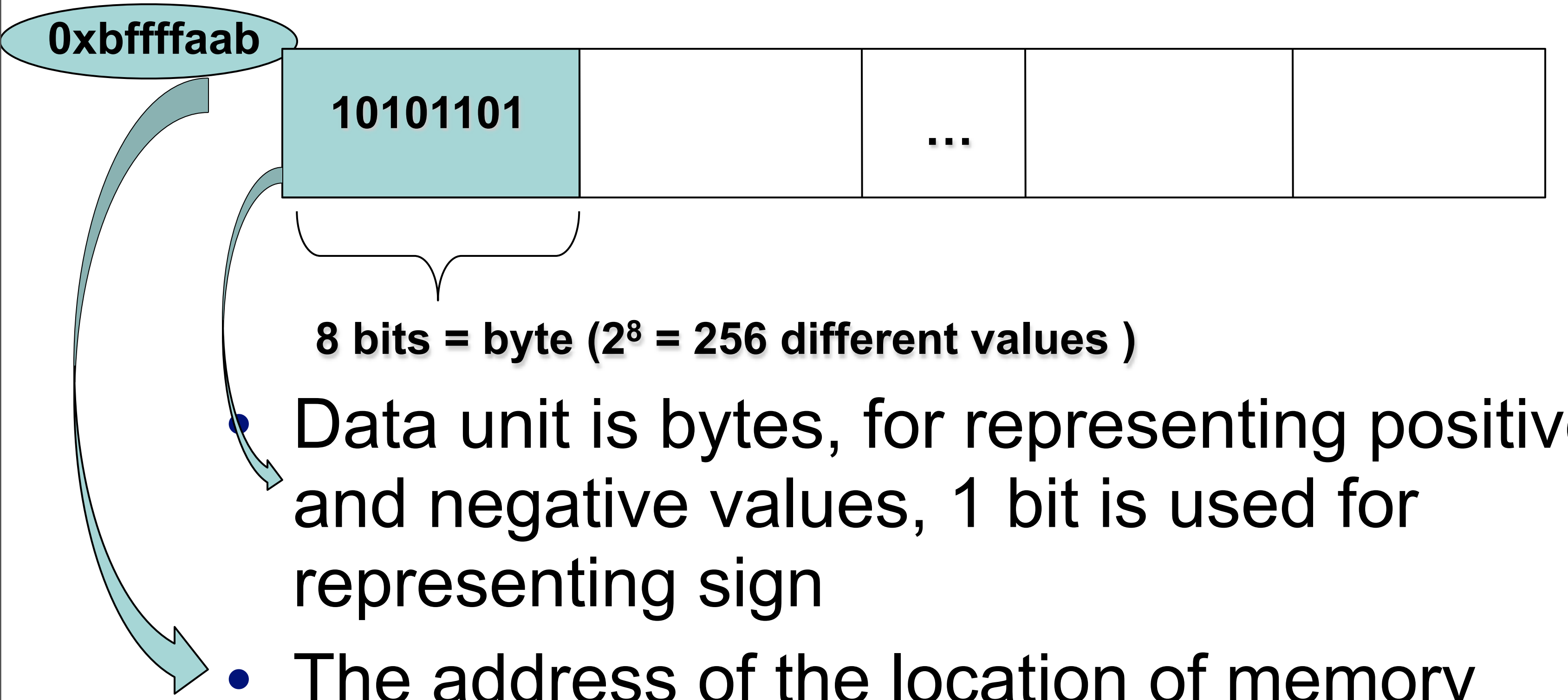
We can have pointers to functions just as we have pointers to base types. But, functions are not values....

Why understanding types matter …

- Types define an abstraction or approximation of a computation....
  - what is a specification?
- More practically, there are implicit conversions that take place and they may result in truncation, and ...
- Some data types are not interpreted the same on different platforms, they are machine-dependent

Type representation

- How is data represented in memory?



- Data unit is bytes, for representing positive and negative values, 1 bit is used for representing sign
- The address of the location of memory where the data is stored is also a number and a type (it is a pointer)

Type representation

- char: 1 byte
- int: an integer, the size of integers on that machine
  - typically 4 bytes
- float: single-precision floating point
  - typically 4 bytes
  - 1 sign bit, 8 bit exponent, 24 bit significand
- double: double-precision floating point
  - typically 8 bytes
  - 1 sign bit, 11 bit exponent, 52 bit significand

Basic types

Type	Size (byes)	Value
char	1	-128 to 127
short(int)	2	-32768 to 32767
long(int)	4 or 8	You compute ☺
int	Machine dependent	
unsigned char	1	0 to 255
unsigned short	2	0 to 65535
unsigned long	4	You compute ☺
float	4	
double	8	
long double	12	

sizeof(x) returns the size in bytes.

Integers

- Its size depends on the machine architecture (often 4 bytes).
- For portability purposes, many programs define int32 as a 4 bytes int and work with this type instead of int.
- Additional types:
  - short int or short (2 bytes)
  - long int or long   (4 bytes)

Signed vs. unsigned

- Other types:
  - unsigned char
  - unsigned int
- The range of int values that can be represented if size of int is 4 bytes ?
  - signed: -2^31 to 2^31 - 1 (two's complement)
  - unsigned: 0 to 2^32

Characters representation

- ASCII code (American Standard Code for Information Interchange): defines 128 character codes (from 0 to 127),
- In addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, and that are platform-dependent.
- Examples:

  The code for ‘A’ is 65

  The code for ‘a’ is 97

  The code for ‘0’ is 48

Static libraries

- Library:
  - file containing several object files used as a single entity in the linking phase of a program.
  - the library is indexed, so it is easy to find symbols (functions, variables) in them.
- Static libraries: collections of object files that are linked into the program during the linking phase of compilation.

Examples:

Unix: XXX.a

Windows: XXX.lib

Static vs. shared libraries

- Shared libraries:
  - Only one copy of the library is stored in memory at any given time (use less memory to run our programs, the executable files are much smaller).
  - Slightly slower start of the program.
- Static libraries:
  - Each process has its own copy of the static libraries is using, loaded in memory.
  - Executable files linked with static libraries are bigger.

Shared libraries

- Shared libraries (dynamic libraries) are linked into the program in two stages.
  - During compilation time, the linker verifies that all the symbols (functions, variables) required by the program, are either linked into the program, or in one of its shared libraries. The object files from the dynamic library <u>are not inserted into</u> the executable file.
  - When the program is started, a program in the system (called a dynamic loader or linker) checks out which shared libraries were linked with the program, loads them to memory, and attaches them to the copy of the program in memory.

libc

- C Standard library – is an interface standard which describes a set of functions and their prototype used to implement common operations
- Libc – is the implementation of the C Standard library on UNIX systems

libc is linked in by default as a shared library

Let’s speak C – Hello World

```
#include<stdio.h>
int main() {
    /* every program must have a main */
    printf("Hello world!\n");

    return 0;
}

gcc –c hello.c          /* compile*/
gcc –o hello hello.o  /* link */
OR
gcc hello.c                /* compile and link */
```

Main function

- Every C program has to have a main
- It has to be declared *int main* for portability
- Returning 0 means the program exited OK
- The return value is interpreted by the operating system
- Main takes arguments:
  - We will see later how to pass parameters to a program using main

More C…

```
#include <stdio.h>

int main() {
   int c;

   c = getchar();
   while(c != EOF) {
       putchar(c);
       c = getchar();
   }
   return 0;
}
```

‘Making’ our coding life easier…

- Source and header files compiled in object files and then linked in an executable. Requires linking with other external libraries. For complex projects, need for an organized and efficient way to do this.
- The *make* utility
  - automatically determines which pieces of a large program need to be recompiled and issues commands to recompile them.
  - can be used with any programming language (not only C) whose compiler can be run with a shell command.
  - not limited to programs: documentation, distribution.

Running *make*

- Write a make file, the default name is “Makefile`` that describes the relationships among files in the program and provides commands for updating each file.
- Then run from the shell the command:

```
make
make –f Makefile_name
```

- The `make' program uses the Makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

An example of a Makefile

- **C files: main.c, command.c, display.c, utils.c**
- **H files: defs.h, command.h**

This is a tab

```
edit : main.o command.o display.o utils.o
	gcc –o edit main.o command.o display.o \
	utils.o
main.o : main.c defs.h
	gcc –c main.c
command.o : command.c defs.h command.h
	gcc –c command.c
display.o : display.c defs.h
	gcc –c display.c
utils.o : utils.c defs.h
	gcc –c utils.c
clean :
```

Variables and implicit rules

- It is not necessary to spell out the commands for compiling the individual C source files, `make' can figure them out: it has an "implicit rule" for updating a `.o' file from a correspondingly named `.c' file using a `gcc -c' command.

- To simplify writing make files, one can define variables:

```
OBJS = main.o command.o display.o utils.o
...
edit : $(OBJS)
        cc -o edit $(OBJS)
```

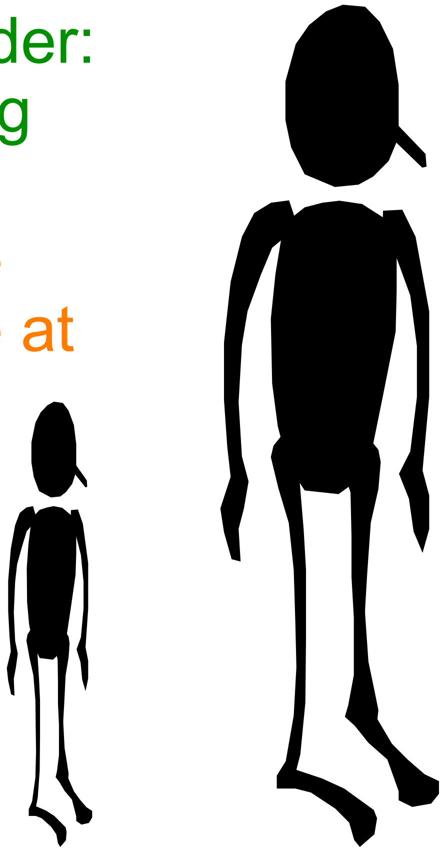
```
all: hello

hello : helloworld.o
        gcc -o hello helloworld.o

helloworld.o : helloworld.c
        gcc -c helloworld.c

clean:
     rm hello helloworld.o
```

Some portability issues

- Representation issues
  - Endianess
  - Integer representation
  - Size
  - Alignment
- Standard libraries
  - Sometimes the functions that have the same functionality have different names/parameters on different platforms; Use the right one
  - Include the right header files
  - Link with the right libraries

Byte order

- Different systems store multibyte values (for example int) in different ways.
  - HP, Motorola 68000, and SUN systems store multibyte values in Big Endian order: stores the high-order byte at the starting address
  - Intel 80x86 systems store them in Little Endian order: stores the low-order byte at the starting address.
- Data is interpreted differently on different hosts.



Printf format

c	Character
d or i	Signed decimal integer
f	Decimal floating point
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
p	Pointer address

NOTE: read printf man pages for additional formats

What will this program output?

```
#include <stdio.h>
int main() {
      char c = ‘a’;

      printf(”%c  %d  %x \n", c, c, c);

      return 0;
}
```

What will this program output?

```
#include <stdio.h>
int main() {
    char c = ‘a’;

    printf(”%c  %d  %x \n", c, c, c);

    return 0;
}
```

a 97 61

What can go wrong?

```
include <stdio.h>
int main () {
   short s = 9;
   long  l = 32768;
   printf("%d\n", s);
    s = l;
    printf("%d\n", s);

    return 0;
}
```

What can go wrong?

```
include <stdio.h>
int main () {
   short s = 9;
   long  l = 32768;
   printf("%d\n", s);
    s = l;
    printf("%d\n", s);

    return 0;
}
```

```
9
-32768
```