C Overview

CSE 333 Winter 2021

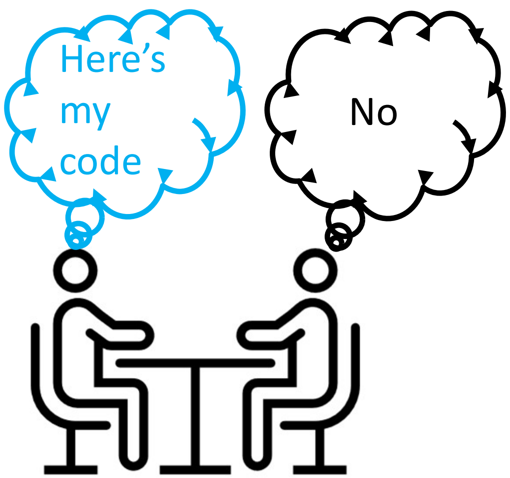
Instructor: John Zahorjan

Teaching Assistants:

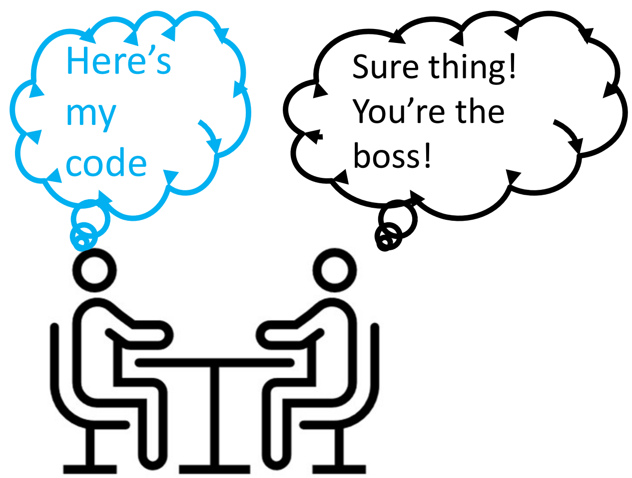
Matthew Arnold	Non Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

Goal

- C isn't Java
- What is it?



You Java



You C

Understanding C

- With modern languages, we understand the source to be the specification of a computation
  - We don't think much about the execution on hardware
    - The code may not even execute directly on hardware

- I recommend that you start by thinking of C as an efficient way to write assembler code
  1. Produce boilerplate code, like procedure entry/return, like for loop control statements, like array index computations, …
  2. Variable names rather than offset(base) addresses

- Types in C are (a) about expressing decisions about memory allocation, and (b) possibly of some use to programmers to detect statically something that may or may not be an error, e.g., adding a `char` to a `float`

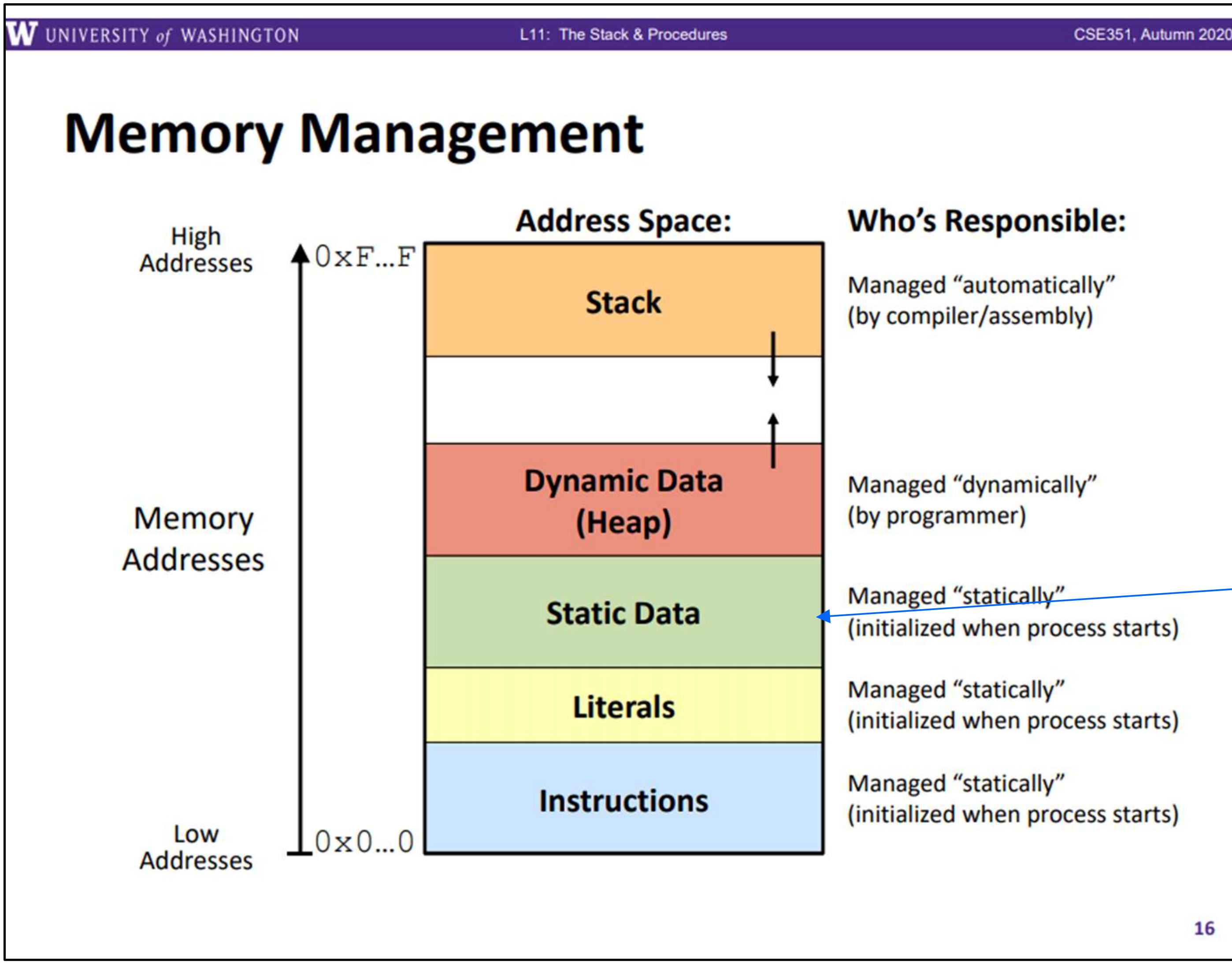
Understanding C

- When you declare a variable, where you place it in the source code expresses a decision about where to allocate memory
  - Local variables → stack
  - Global variables → static data area
- A side effect: initialization
  - Local variables → no default initialization; explicit only
  - Global variables → 0 (which is operationally equivalent to no initialization)

Understanding C

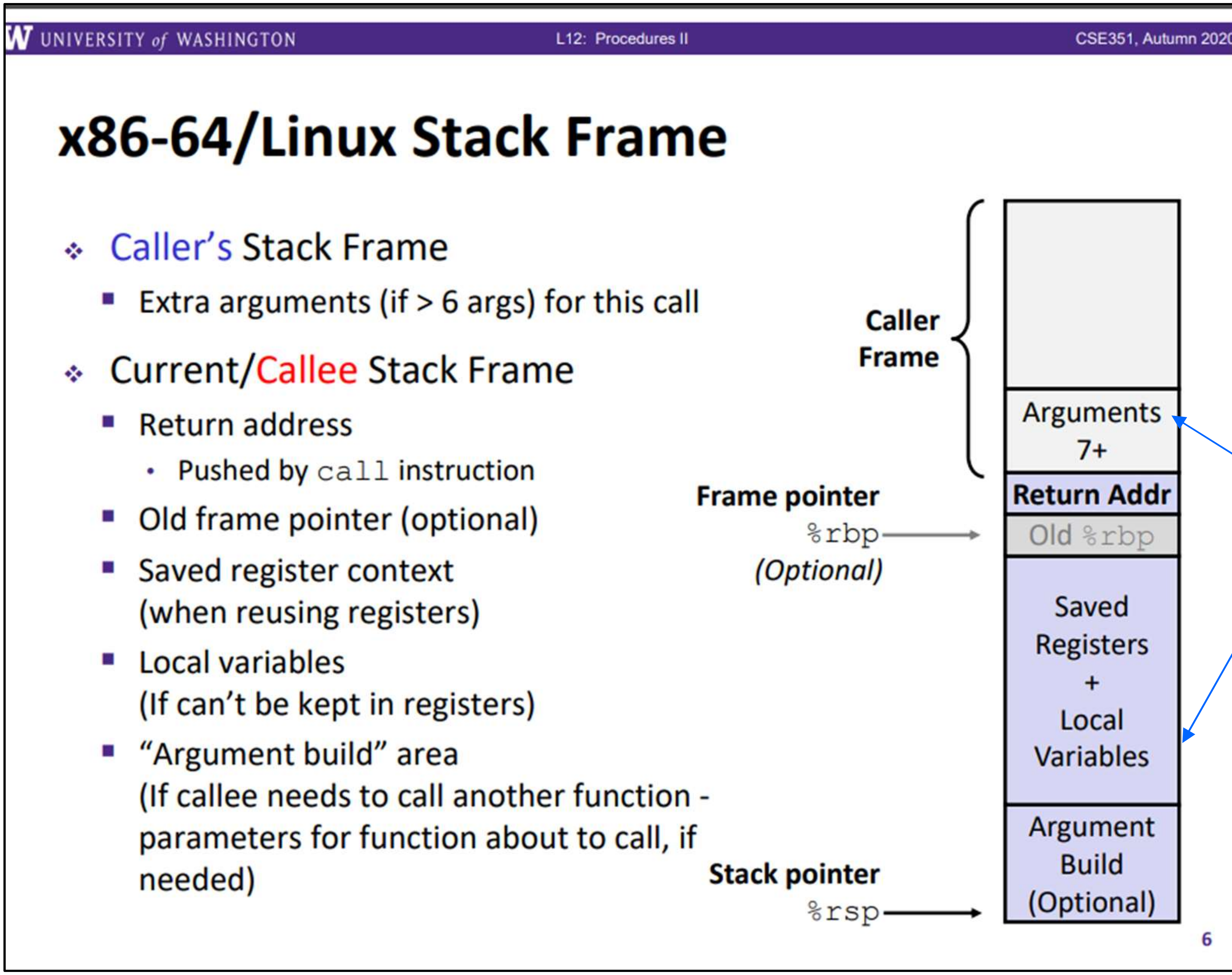
- When you declare a variable's type, you are:
  - expressing a decision about how much memory to allocate
    - How big?
      - char – 1B
      - int – probably 4
      - long - probably 8
      - xxxx* - 4B or 8B; depends on underlying hardware
      - int32_t – 4B
      - intptr_t – whatever the length of int* is
  - Indicating a preference for how to understand operations
      - x + y
        - adds
        - addq (int integer or float reqisters)
        - addb
        - Etc.

Review: Virtual Address Space Layout



C global variables

Review: Stack Frame



C local variables

Example C Program

```
// This code compiles (gcc) without errors or warnings

int globalInt = 16;

int main(int argc, char *argv[]) {
    int localInt = 32;
    char localChar = '1';
    double localDouble;

    localDouble = globalInt + (long)(argc + argv + localInt + localChar) + localDouble;

    return 0;
}
```

There are five locals.
main() is just a normal procedure.

This cast eliminates a compiler warning, but the code compiles without the cast

Questions:

1. Is it a good thing that this code compiles?
2. What does the code mean?

What the C Compiler Thinks It Means: V1

$ gcc –O3 –S example.c

Produce assembler file

Enable aggressive optimization

```
    .file   "00-example.c"
            .text
            .section        .text.startup,"ax",@progbits
            .p2align 4
            .globl  main
            .type   main, @function
main:
.LFB0:
            .cfi_startproc
            xorl    %eax, %eax
            ret
            .cfi_endproc
.LFE0:
            .size   main, .-main
            .globl  globalInt
            .data
            .align 4
            .type   globalInt, @object
            .size   globalInt, 4
globalInt:
            .long   16
            .ident  "GCC: (GNU) 9.2.1 20191120 (Red Hat 9.2.1-2)"
            .section        .note.GNU-stack,"",@progbits
```

```c
int globalInt = 16;

int main() {
  return 0;
}
```

The compiler thinks the original source is equivalent to this

What the C Compiler Thinks It Means: V2

$ gcc –O0 –S example.c

```
	.file	"00-example.c"
	.text
	.globl	globalInt
	.data
	.align 4
	.type	globalInt, @object
	.size	globalInt, 4
globalInt:
	.long	16
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	%edi, -20(%rbp)
	movq	%rsi, -32(%rbp)
	movl	$32, -4(%rbp)
	movb	$49, -5(%rbp)
	movl	globalInt(%rip), %eax
	cltq
	movl	-20(%rbp), %edx
	movslq	%edx, %rcx
	movl	-4(%rbp), %edx
	movslq	%edx, %rdx
	addq	%rdx, %rcx
	movsbq	-5(%rbp), %rdx
	addq	%rcx, %rdx
	leaq	0(,%rdx,8), %rcx
	movq	-32(%rbp), %rdx
	addq	%rcx, %rdx
	addq	%rdx, %rax
	cvtsi2sdq	%rax, %xmm0
	movsd	-16(%rbp), %xmm1
	addsd	%xmm1, %xmm0
	movsd	%xmm0, -16(%rbp)
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
```

Procedure entry

Local var initialization

Expression evaluation

Procedure exit

Local Variable Initialization

- movl %edi, -20(%rbp) // argc
- movq %rsi, -32(%rbp) // argv
- movl $32, -4(%rbp) // localInt
- movb $49, -5(%rbp) // localchar
- movl globalInt(%rip), %eax
- cltq
- movl -20(%rbp), %edx
- movslq %edx, %rcx
- movl -4(%rbp), %edx
- movslq %edx, %rdx
- addq %rdx, %rcx
- movsbq -5(%rbp), %rdx
- addq %rcx, %rdx
- leaq 0(,%rdx,8), %rcx
- movq -32(%rbp), %rdx
- addq %rcx, %rdx
- addq %rdx, %rax
- cvtsi2sdq %rax, %xmm0
- movsd -16(%rbp), %xmm1
- addsd %xmm1, %xmm0
- movsd %xmm0, -16(%rbp)

```
int main(int argc, char *argv[]) {
    int localInt = 32;
    char localChar = '1';
    double localDouble;
```

Arguments are "passed by value".

The variables in the function's parameter lists are local variables, just like those declared in the body of the function. They're initialized to the values of the arguments from the call.

C Compiler: Variable Names to Addresses

```
movl   %edi, -20(%rbp)     // argc
movq   %rsi, -32(%rbp)     // argv
movl   $32, -4(%rbp)       // localInt
movb   $49, -5(%rbp)       // localchar
movl   globalInt(%rip), %eax     // globalInt
cltq
movl   -20(%rbp), %edx
movslq  %edx, %rcx
movl   -4(%rbp), %edx
movslq  %edx, %rdx
addq   %rdx, %rcx          // argc + localInt
movsbq  -5(%rbp), %rdx
addq   %rcx, %rdx          // argc + localInt + localchar
leaq   0(,%rdx,8), %rcx
movq   -32(%rbp), %rdx
addq   %rcx, %rdx          // argc + localInt + localchar + argv
addq   %rdx, %rax          // argc + localInt + localchar + argv + globalInt
cvtsi2sdq     %rax, %xmm0
movsd  -16(%rbp), %xmm1  // fetch localDouble (uninitialized)
addsd  %xmm1, %xmm0      // argc + localInt + localchar + argv + globalInt + localDouble
movsd  %xmm0, -16(%rbp)
```

```
localDouble = globalInt +
              (long)(argc + argv + localInt + localChar)
              + localDouble;
```

C Compiler: (Implicit) Type Conversion

- movl %edi, -20(%rbp) // argc
- movq %rsi, -32(%rbp) // argv
- movl $32, -4(%rbp) // localInt
- movb $49, -5(%rbp) // localchar
- movl globalInt(%rip), %eax // globalInt
- cltq
- movl -20(%rbp), %edx
- movslq %edx, %rcx
- movl -4(%rbp), %edx
- movslq %edx, %rdx
- addq %rdx, %rcx // argc + localInt
- movsbq -5(%rbp), %rdx
- addq %rcx, %rdx // argc + localInt + localchar
- leaq 0(,%rdx,8), %rcx
- movq -32(%rbp), %rdx
- addq %rcx, %rdx // argc + localInt + localchar + argv
- addq %rdx, %rax // argc + localInt + localchar + argv + globalInt
- cvtsi2sdq %rax, %xmm0
- movsd -16(%rbp), %xmm1 // fetch localDouble (uninitialized)
- addsd %xmm1, %xmm0 // argc + localInt + localchar + argv + globalInt + localDouble
- movsd %xmm0, -16(%rbp)

```
localDouble = globalInt +
              (long)(argc + argv + localInt + localChar)
              + localDouble;
```

Types

- Note that there was no assembler code that checked the type of anything
  - E.g., main doesn't check that the bits in the 8 bytes named argv are a pointer to an array of characters

- Type checking happens entirely at compile time
  - Static type checking

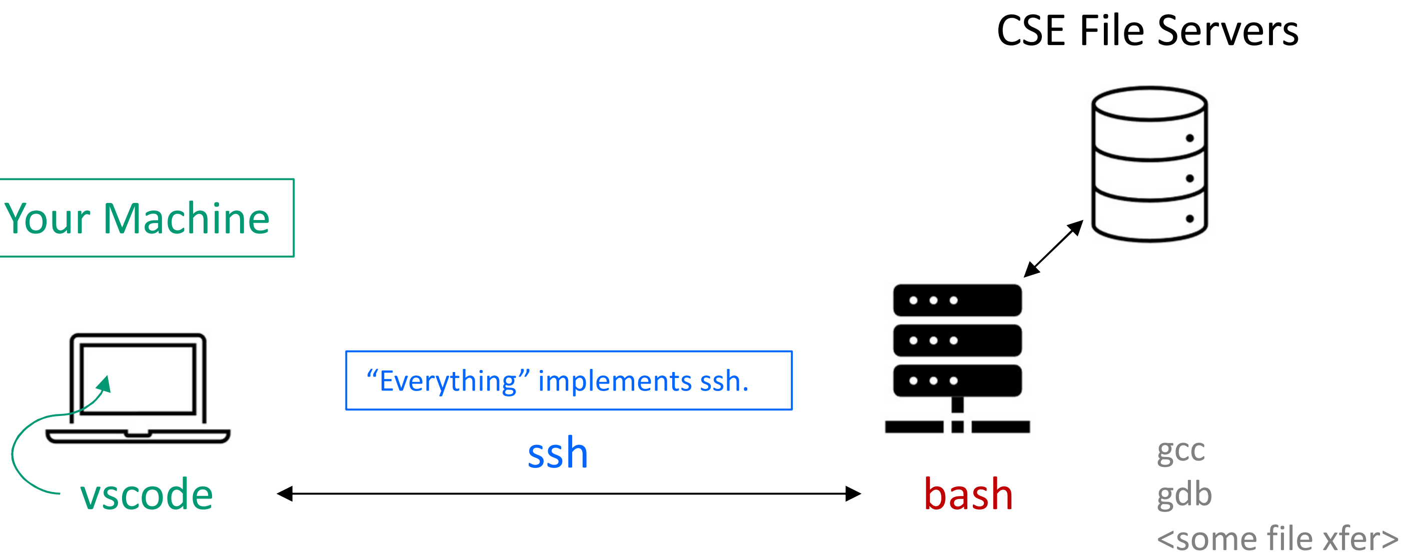
- The compiler sometimes needs to know the type of a variable to figure out what code to generate
  - E.g., x+y // which assembler instruction is required? float regs vs int regs

- Sometimes the compiler wants to know types (mainly) to help the programmer
  - ```
    int x,y;
    x = sub(x,y);  // I can generate the code for this call
                   // but if sub takes 3 args it's not my fault…
    ```
    ```

Types

- I think it will help to keep in mind that the point of types in C is mainly/somewhat to say how much space the compiler should allocate
  - It's not types in the Java sense
  - The rules of C aren't trying to guarantee anything about the correctness of programs written in it

- Example:
  - All pointer types are the same size (8 bytes)
    - int* / char* / double*/ char *[] / void* / int**
  - "void*" is sometimes used to mean "any value up to 8 bytes"
  - Important to some implementations of generics in C

Declaration vs. Definition

- x = sub(x,y); // sample statement
- If I know the type of sub I can
  - Complain if the call doesn't conform
    - int sub(int x, int y, int z);
  - Perform implicit type conversion if necessary
    - int sub(int x, float y);
- To do those things, all I need to know is the type of sub, not the code of sub
- Declaration: gives the type signature for something
- Definition: creates something
- C **wants** to generate code

Example Declarations

- Example Declarations
  - int sub(int x, int y); // says there will be an int sub(int,int)
  - extern int x; // says there will be a global x, but doesn't create it
  - int x; // equivalent to extern int x; in some cases
  - struct student; // declares but doesn't define a type

Example Definitions

- Example Definitions
  - ```
    int sub(int x, int y) {
        return 0;
    }
    ```
  - `int x;          // may or may not be a definition`
  - `int x = 0;      // definitely a definition`
  - ```
    struct animal* pAnimal;  // defines the pointer pAnimal even if
                             // struct animal isn't defined
    ```
  - ```
    typedef struct {
        int id;
        char name[20];   // yikes!
    } student_t;      // defines type student_t but no variable of that type
    ```
  - `student_t TA;   // defines variable TA of  type student_t`
    ```

Aside: using vscode

attu.cs.washington.edu

Your Machine



vscode runs on your machine and so
can do window operations on your screen.

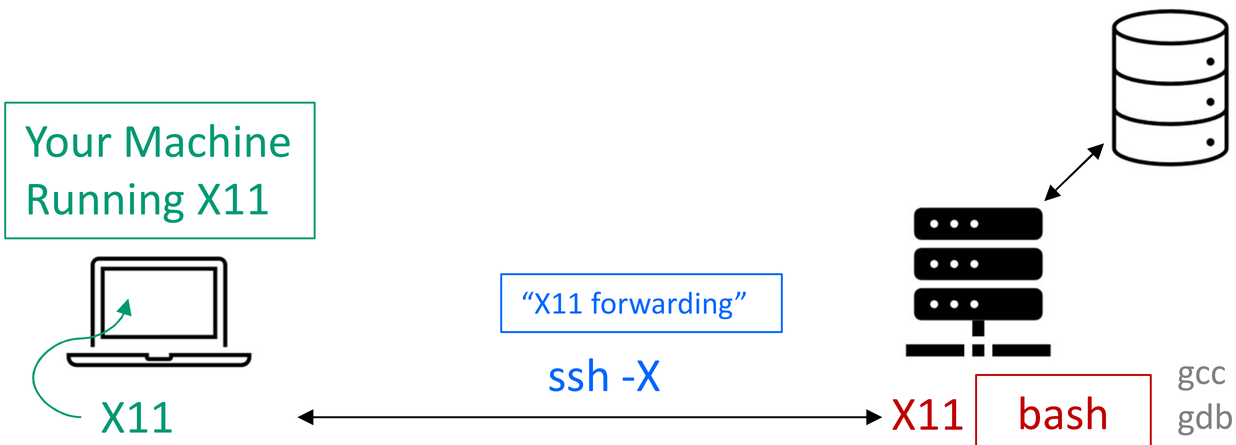
Your files are here.
You are building / running / debugging here.

Aside: alternative configuration 2

attu.cs.washington.edu

CSE File Servers

Your Machine
Running X11



bash

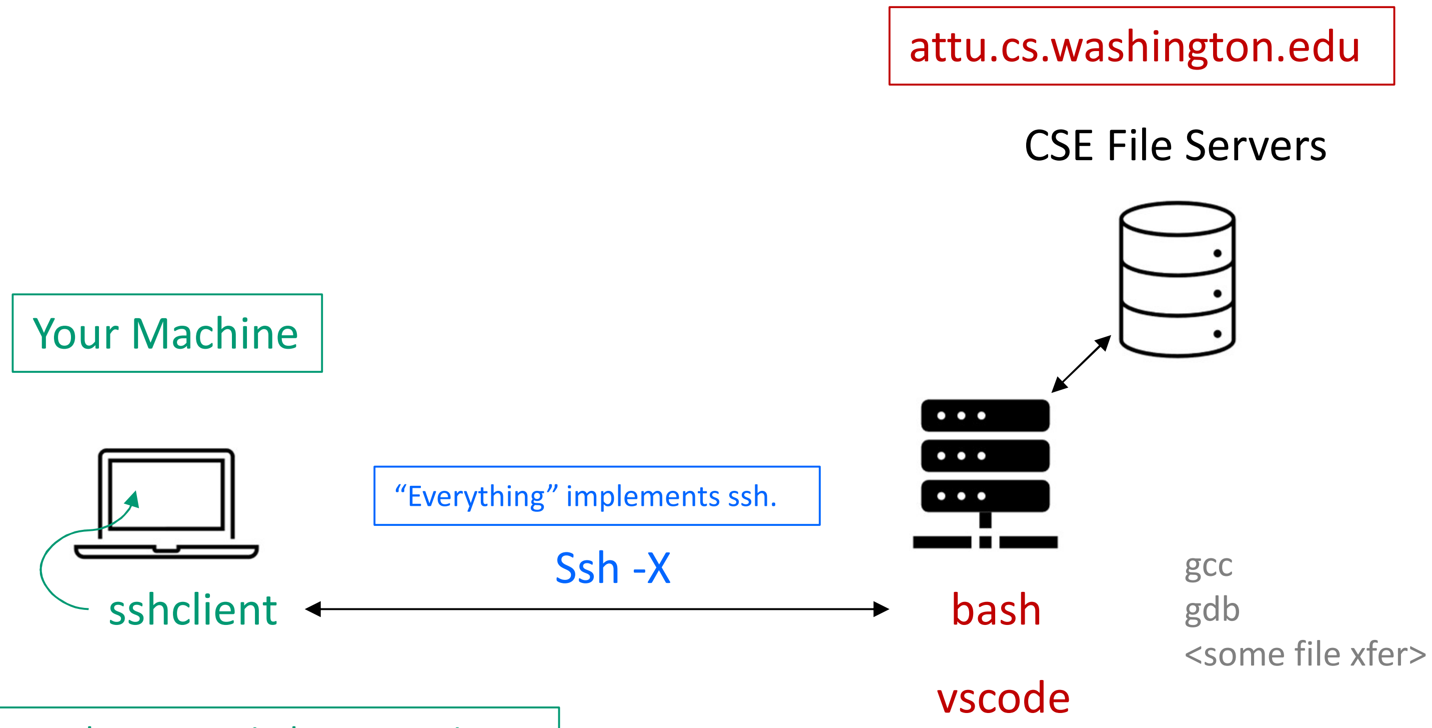
vscode

gcc
gdb
<some file xfer>

X11 on Linux: easy.
X11 on Windows: possible (Xming)
X11 on macOS: apparently possible

Your files are here.
You are building / running / debugging here.
You are also running a visual editor here.

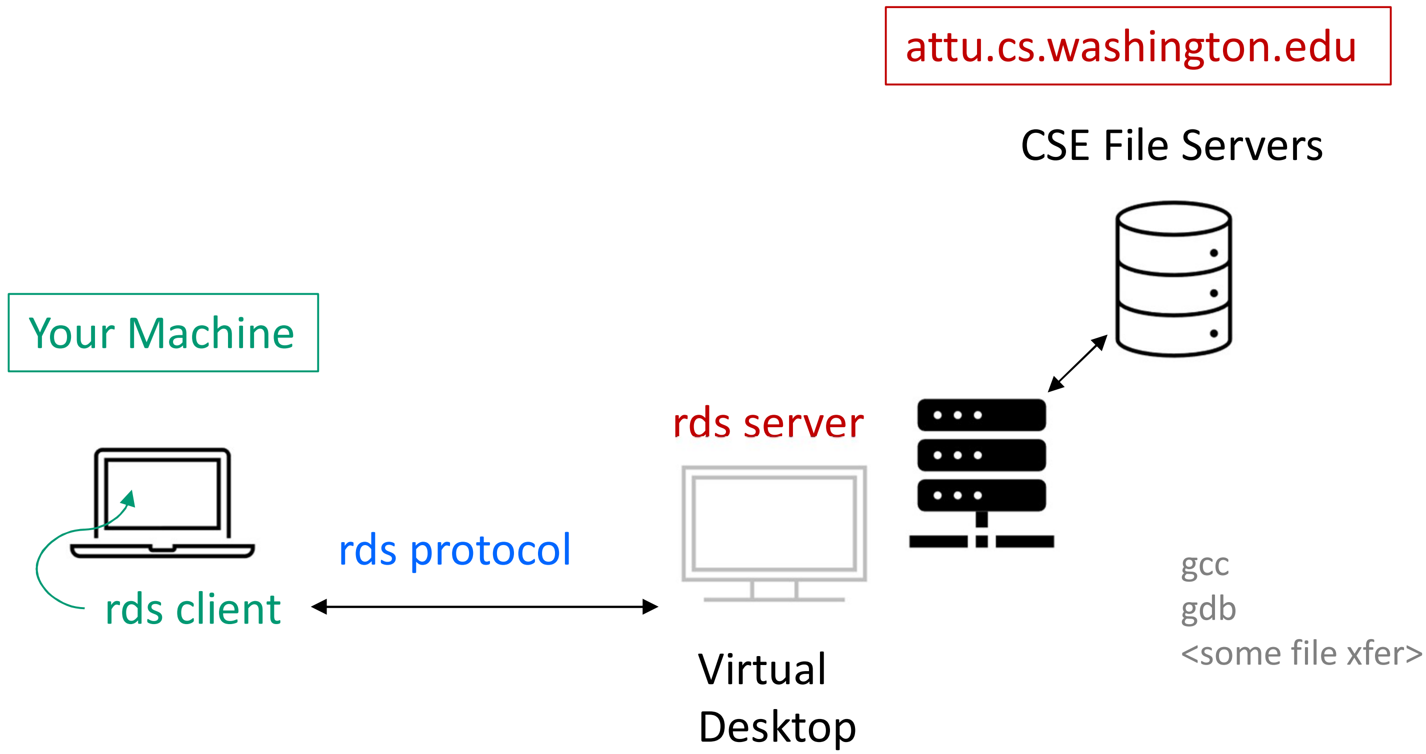
Aside: alternative configuration 2



Forget vscode or any window operations, all you have is a text window.
ssh redirects client input to remote shell
remote shell output back to client.

Your files are here.
You are building / running / debugging here.

Aside: alternative configuration 3



An Opportunity

- It's obviously useful to allow code to be divided into many files
- Imagine the following situation where the caller in file A.c calls a function from file B.c

File A.c:

```
int sub(int x);
int Acode() {
    …
    x = sub(x);
    …
}
```

File B.c:

```
int sub(int x) {
    …
}
```

- When compiling A.c, the compiler will generate code that correctly puts arguments in registers, saves caller saved registers, and transfers control to sub
- When compiling B.c, the compiler will generate code that correctly saves t he return address, locates the arguments, processes the body, and returns a result
- The compiler doesn't need B.c to compile A.c, nor vice versa. (You can name the files whatever you want…)
- Yeah!

A Disaster

- But suppose some human edits B.c but forgets to modify A.c

File A.c:

```
int sub(int x);
int Acode() {
    …
    x = sub(x);
    …
}
```

File B.c:

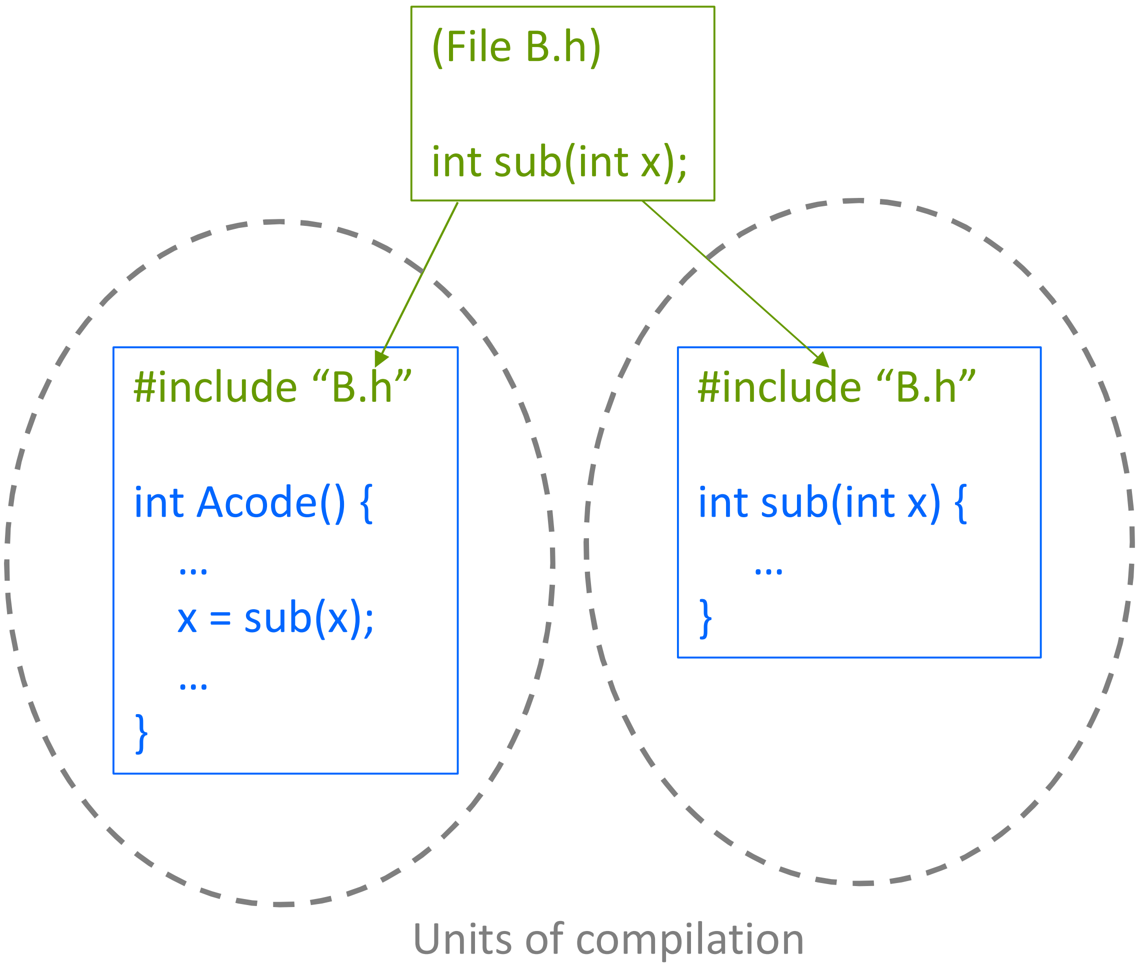
```
int sub(int x, int y) {
    …
}
```

- Looking at the code in A.c, the compiler thinks everything is fine and generates code corresponding to the promised type of sub()
- Looking at the code in B.c, the compiler thinks everything is fine and generates code expecting the caller to provide two arguments
- When the program is run, strange things happen at run time
  - What happens cannot be explained based on the language semantics

The Fix (and a programming/life lesson)

- The more copies there are of some piece of information (or things derived from that piece of information) the larger the number of opportunities to forget to update one when the information changes
- Rule: There should be only one copy!
  - If you update, you update completely

.h files (and #include)



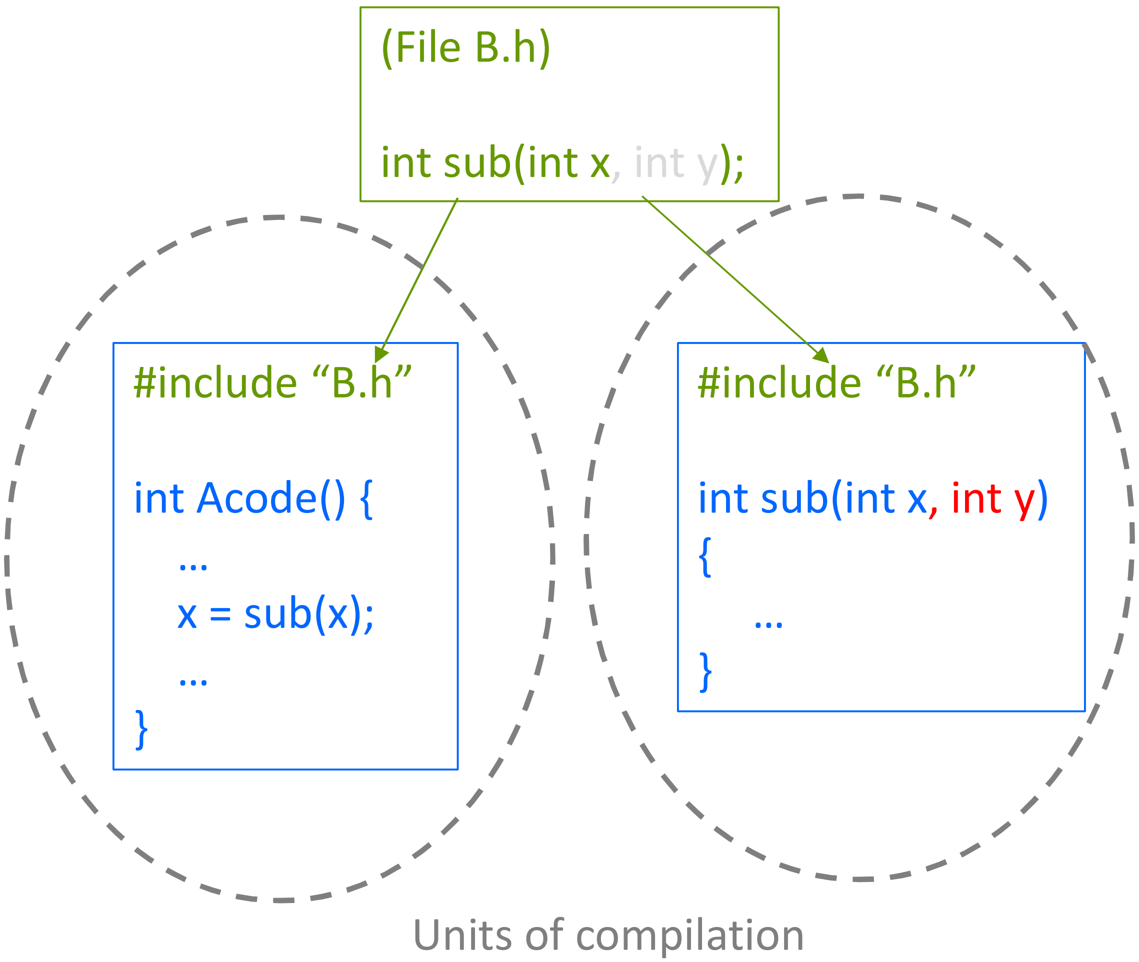
Each file is compiled independently even if you compile using a single command, like

```
$ gcc A.c B.c
```

That's the same as

```
$gcc -c A.c
$gcc -c B.c
$gcc A.o B.o
```

.h files (and #include)



If you update B.c but not B.h, then you get an error when compiling B.c

If you update B.c and B.h, you get an error when compiling A.c

If you update B.c and B.h and A.c, no compiler errors!

#include

- The "#include" directive is a preprocessor directive
  - What it does happens pre-compile
- Example:
  - The preprocessor reads A.c and outputs it line-by-line as it goes into a new, temporary file
  - When it sees the line '#include "B.h"' it switches to reading file B.h line-by-line, outputting each to the temporary file
  - When it reaches the end of B.h, it resumes reading A.c line-by-line from where it left off
  - (This happens recursively if B.h has #include directives in it)
- The compiler compiles the temporary file, which contains all the text from A.c and each #included file

Quick Question

- What do you think happens when you compile this file?

```
#include "myCV.pdf"
int Acode() {
   return 0;
}
```

A. The preprocessor rejects myCV.pdf and terminates with an error
B. The preprocessor rejects myCV.pdf and doesn’t include anything but keeps on going
C. The compiler complains that myCV.pdf is a pdf file and terminates
D. The compiler complains about something that makes no sense
E. Nothing, everything is fine

Quick Answer

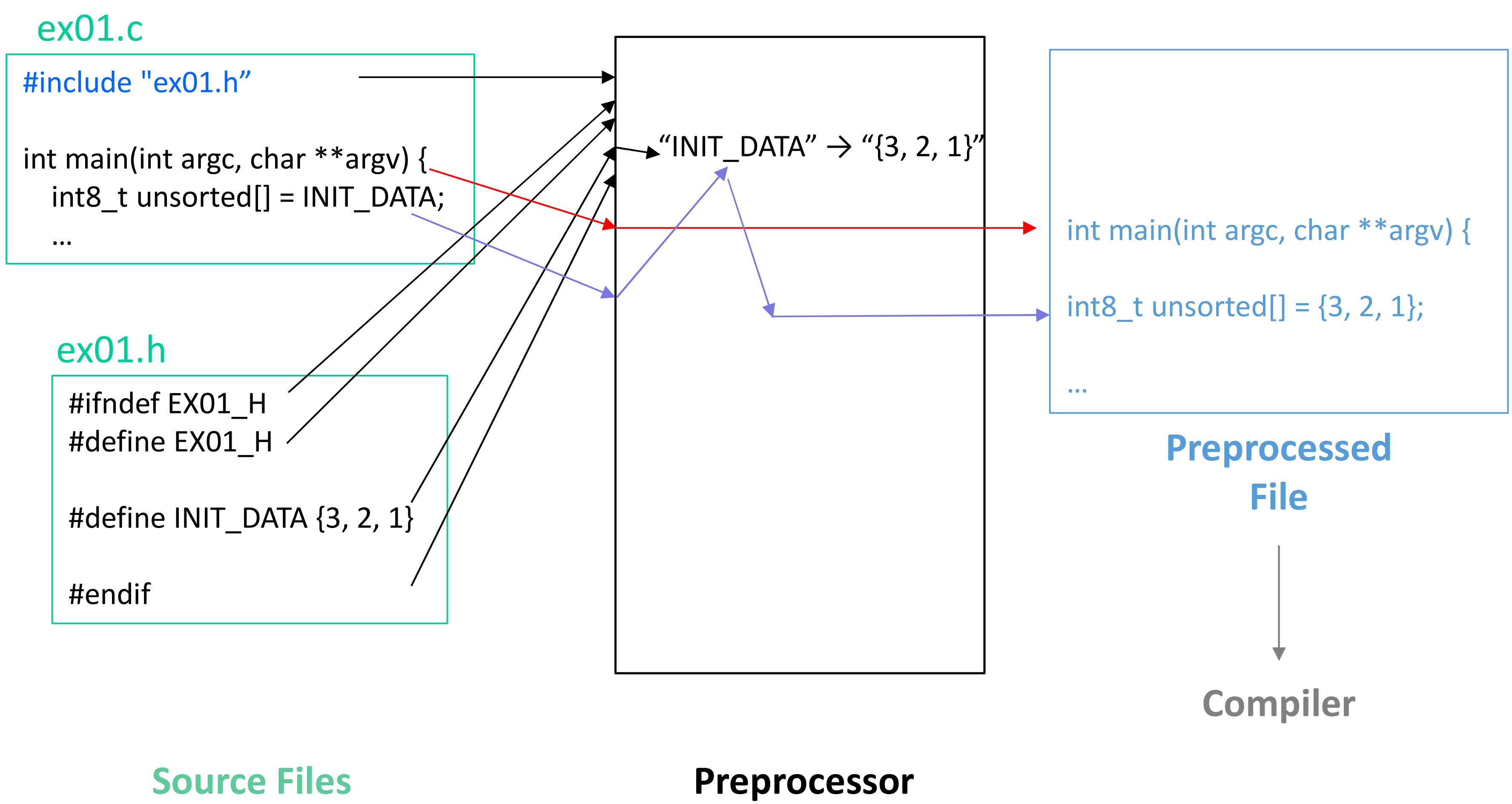
```
In file included from include.c:1:
myCV.pdf:1:1: error: expected identifier or ‘(’ before ‘%’ token
    1 | %PDF-1.3
      | ^
myCV.pdf:2:2: error: stray ‘\304’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |  ^
myCV.pdf:2:3: error: stray ‘\345’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |   ^
myCV.pdf:2:4: error: stray ‘\362’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |    ^
myCV.pdf:2:5: error: stray ‘\345’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |     ^
myCV.pdf:2:6: error: stray ‘\353’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |      ^
myCV.pdf:2:7: error: stray ‘\247’ in program
    2 | %<C4><E5><F2><E5><EB><A7><F3><A0><D0><C4><C6>
      |       ^
…
```

Aside: #define and ex01

- #define is another string manipulation operation
  - It's a preprocessor command, and takes place before compilation
  - The programmer is using the preprocessor to compose the actual C program to be compiled

- #define SIZE 16 vs. int SIZE 16
  - Former takes no memory
  - total = SIZE + 2;
    - For #define, the compiler sees total = 16 + 2; and can recognize that it can compute 16+2 right now and doesn't need to generate code to do an add at run time
    - For int, the compiler may or may not be able to figure out the value is 16. It's complicated… (Can the variable's value have changed since initialization?)

Aside: #define and ex01



Back to C Declarations: Compile-Time Symbol Table

- The compiler needs to keep track of "the meaning" of each symbol that has been declared/defined
- As it reads lines of the (preprocessed) file, it adds to its symbol table
- The symbol table keeps track of, at least, the symbol (name), its type, it's scope, and how to reference it
- When a symbol is referenced (used), the compiler looks at the symbol table and decides if the use is okay

Compile-Time Symbol Table Maintenance

```
#include <stdio.h>

int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}
```

Essential Idea of the Symbol Table

Symbol	**Type**	**Value**
printf	int printf(const char *, ...)	
sum	int	<static data>
main	int main(int, char *[])	

Compile-Time Symbol Table Maintenance

```
#include <stdio.h>

int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}
```

Symbol	Type	Value
printf	int printf(const char *, ...)	
sum	int	<static data>
main	int main(int, char *[])	
argc	int	-x(%rbp)
argv	char *[]	-y(%rbp)
i	int	-z(%rbp)

Compile-Time Symbol Table Maintenance

```
#include <stdio.h>

int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}
```

⇦

Symbol	Type	Value
printf	int printf(const char *, ...)	
sum	int	<static data>
main	int main(int, char *[])	
argc	int	-x(%rbp)
argv	char *[]	-y(%rbp)
i	int	-z(%rbp)
i	int	-w(%rbp)

Compile-Time Symbol Table Maintenance

```
#include <stdio.h>

int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}
```

Symbol	Type	Value
printf	int printf(const char *, ...)	
sum	int	<static data>
main	int main(int, char *[])	
argc	int	-x(%rbp)
argv	char *[]	-y(%rbp)
i	int	-z(%rbp)

Use Before Declaration

```
#include <stdio.h>

int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  sum = abs(argc);
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}

int abs(int x) {
   return  x > 0 ? x : -x;
}
```

Symbol	Type	Value
printf	int printf(const char *, …)	
sum	int	<static data>
main	Int main(int, char *[])	
argc	int	-x(%rbp)
argv	char *[]	-y(%rbp)
i	int	-z(%rbp)

Use After Declaration, Before Definition

```
#include <stdio.h>

int abs(int);
int sum = 0;  // global just so we have a global

int main(int argc, char *argv[]) {
  int i = -1;
  sum = abs(argc);
  for (int i=0; i<100; i++) {
    sum += i;
  }
  printf("i = %d\n", i);
  printf("sum = %d\n", sum);
}

int abs(int x) {
  return  x > 0 ? x : -x;
}
```

⇦

Symbol	Type	Value
printf	int printf(const char *, …)	
abs	int abs(int)	
sum	int	<static data>
main	Int main(int, char *[])	
x	Int	-4(%rbp)

Final Observations

- Notice that we couldn’t put anything in the value column for procedures
  - We (typically) don’t know where they’ll be in memory, so don’t know exactly how to address them
  - We need the linker to resolve this
- Notice also that how we address globals is a little squirrely
  - It’s basically the same issue, and the same solution
- Finally, notice that if the compiler is convinced to generate code, that there can’t be any runtime type errors
  - The machine instruction defines how the data bits are interpreted
  - Those bits don’t carry any type information…