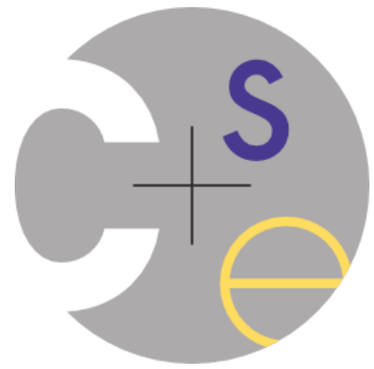
CSE 333

Lecture 2 – Memory

John Zahorjan

Department of Computer Science & Engineering

University of Washington



Today's goals

- *some terminology*
- *review of memory resources*
- *reserving memory*
- *type checking*

Some Terminology

- virtual address space
  A set of addresses naming values, usually related to physical memory addresses through *address translation*
- stack, heap
  Software managed regions of main memory distinguished by how memory is allocated and free'd
- identifier
  A programmer assigned name in a program, e.g., `main` or `total`

Some Terminology

- declaration

  Creating an identifier (and often giving it a type), e.g.,

```
int x;
void foo(int y);
```

- definition

  Declaring a name and "giving it substance" – for functions, giving code; for variables, causing a decision about allocating space to be made

```
int x;
void foo(int y) {}
```

- use

  The use of an identifier (not a declaration or definition)

```
x = 2;
foo(6);
```

Some Terminology

- binding
  Associating an identifier with a memory address
- lifetime
  The time between the allocation of space for the thing named by an identifier and when that space may be reallocated for other purposes
  - in C, locals live for the duration of the enclosing scope
  - in C, globals live from program load time to program termination
  - in C, code lives from program load time to program termination
- scope
  The region of code over which an identifier has the same binding

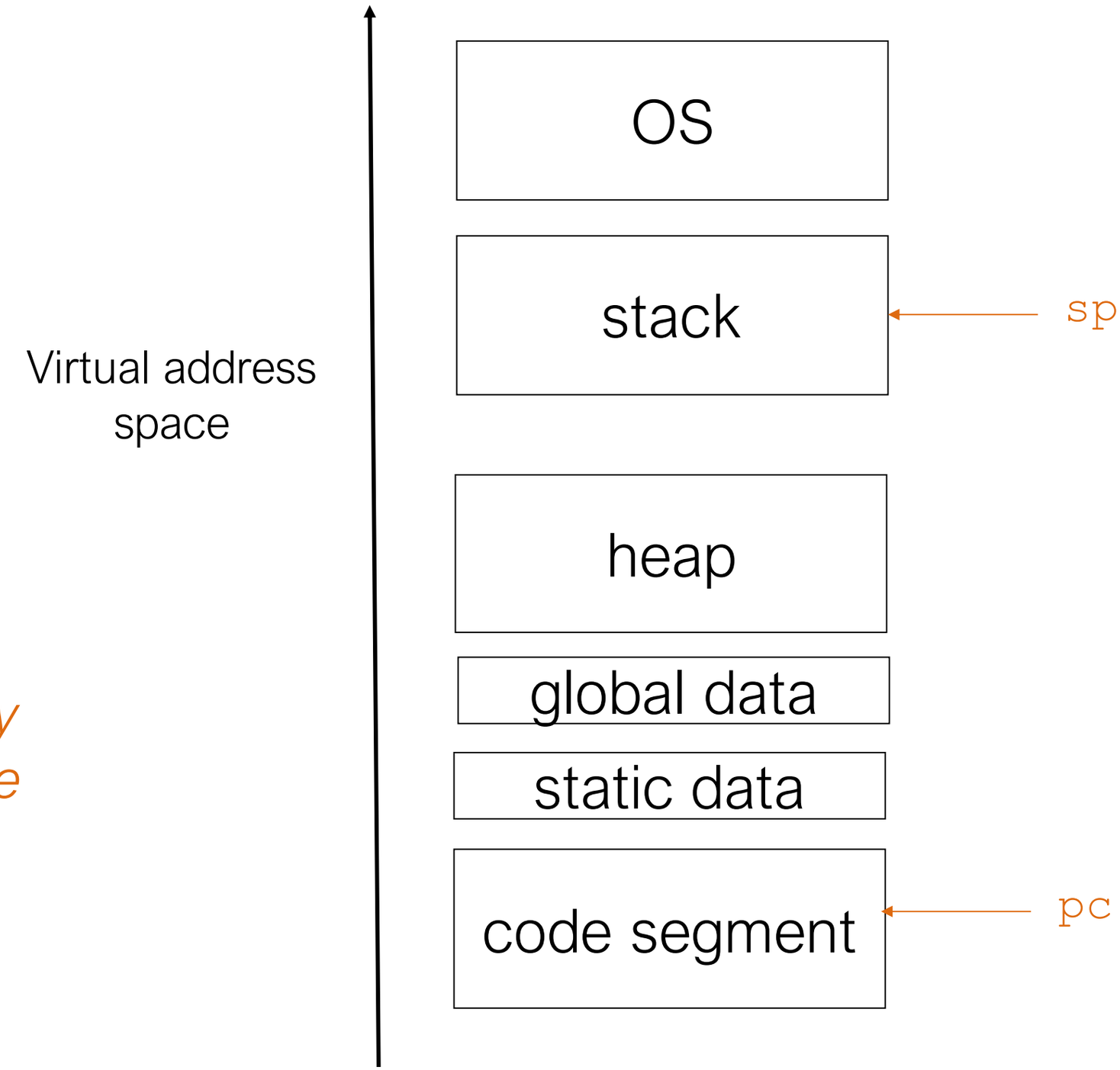
Example

```
int check(char *candidatePassword) {
   return 0;
}

int main(int argc, char *argv[]) {
   char password[7];
   for (password[0]='a'; password[0]<='z'; password[0]++) {
      for (password[1]='a'; password[1]<='z'; password[1]++) {
        check(password)
      }
   }
   return 0;
}
```

check and main are global
password is local

Review of memory resources

This model is supported by the compiler, the linker, the loader, and the OS



Reserving memory: basic rules

Identifier definitions cause memory to be reserved

- A function body reserves the required amount of code (text) segment memory
- A local variable definition reserves space in the stack frame
- A global variable definition reserves space in the global data area

To generate code for uses of identifiers, the compiler must have some idea how to access the memory they name

- Where, exactly, is it?

At compile time, the compiler knows exactly how to reference locals (a known offset relative to the sp).
It doesn't know exactly where globals will be (because it can't make a "map" of global memory since it doesn't know what all the globals will be)

- That isn't known until link time

When are exact addresses determined?

Code

- link time: the **linker** collects all code and lays it out in the text segment

Local variables

- compile time: the **compiler** knows (a) the stack frame layout, and (b) that the stack pointer register points at the stack

Global data area

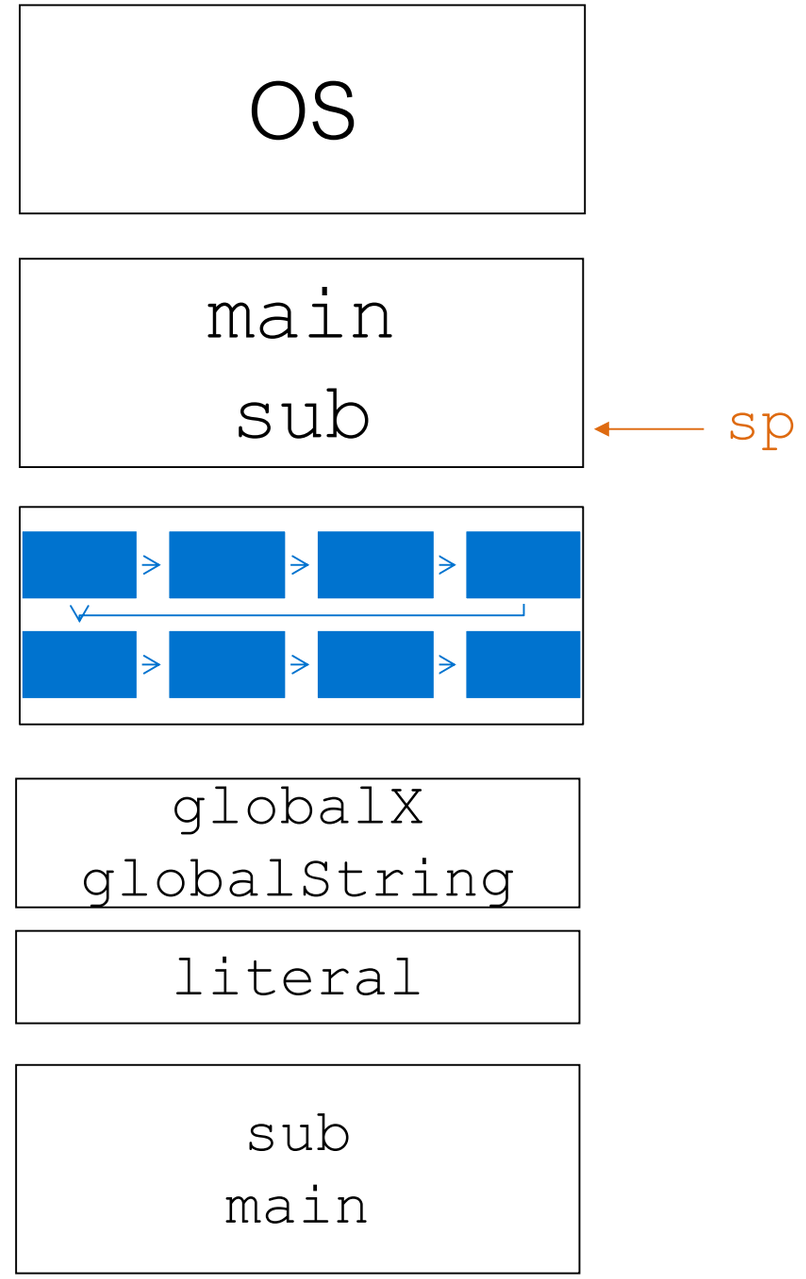
- link time: the **linker** collects all globals and puts them in some order

Static data area

- link time: the **linker** collects all literals that require storage and puts them in some order

Example

```
int  globalX;
char *globalString;

void sub(int x) {
   return x;
}

int main(int argc, char *argv[]) {
    char *prompt = “literal”;
    int y = sub(2);
    return y;
}
```



Compiler generates instructions that move sp to allocate space for x in sub (and assigns it value 2)

compiler generates code moving sp to allocate space for prompt and y on entry to main

Example uses

```
int  globalX;
char *globalString;

int sub(int x) {
  return x;
}

int main(int argc, char *argv[]) {
   char *prompt = "str";
   int y = sub(2);
   return y;
}
```

Compiler understands stack frame layout, so knows how to reference local variables (as offset from sp)

Compiler can't (in general) know where `sub` will be located, so leaves an annotation for linker to "patch" instructions

Symbols in object files main.c separate

```
$ gcc -std=c11 -c linkerMain.c
$ objdump -t linkerMain.o

linkerMain.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 linkerMain.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack        0000000000000000
.note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000004       O *COM*  0000000000000004 globalX
0000000000000008       O *COM*  0000000000000008 globalString
0000000000000000 g     F .text  0000000000000029 main
0000000000000000         *UND*  0000000000000000 sub
```

Symbols in object files

sub.c separate

```
$ gcc -std=c11 -c linkerSub.c
$ objdump -t linkerSub.o

linkerSub.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 linkerSub.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000
.note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000004       O *COM*  0000000000000004 globalX
0000000000000008       O *COM*  0000000000000008 globalString
0000000000000000 g     F .text  000000000000000c sub
```

Symbols in object files main/sub one file

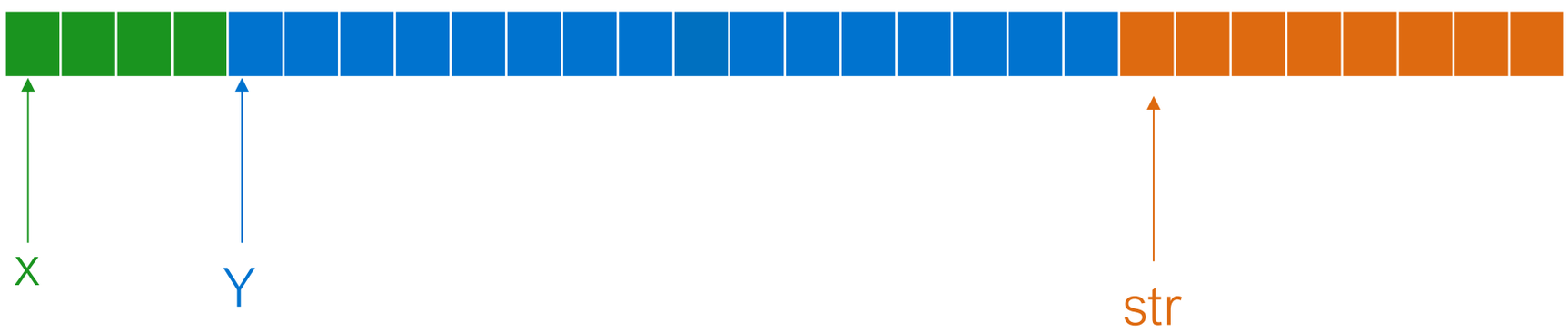
```
$ gcc -std=c11 -c linkerMainSub.c
$ objdump -t linkerMainSub.o

linkerSub.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 linkerSub.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000
.note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000004       O *COM*  0000000000000004 globalX
0000000000000008       O *COM*  0000000000000008 globalString
0000000000000000 g     F .text  000000000000000c sub
000000000000000c g     F .text  0000000000000029 main
```

Types: Why?

```
int    x;
int    Y[10];
char   *str;
```

What is the point of the type specification in each declaration?

- It indicates how much space to reserve
- It allows some type checking of operations



Types: Sizes `sizeof`

```
#include <stdio.h>

void sub(char s[]) {
  printf("sizeof(s) = %lu\n", sizeof(s));                  // 8
}

int main(int argc, char *argv[]) {
  char *pString = "literal string";
  char string[] = "literal string";
  char string20[20] = "literal string";

  printf("sizeof(int) = %lu\n", sizeof(int));              // 4
  printf("sizeof(argc) = %lu\n", sizeof(argc));            // 4

  printf("sizeof(pString) = %lu\n", sizeof(pString));      // 8
  printf("sizeof(string) = %lu\n", sizeof(string));        // 15
  printf("sizeof(string20) = %lu\n", sizeof(string20));    // 20

  sub(string);

  return 0;
}
```

Types: Sizes pointers

What does this print?

```
char  stringArray[] = "example string";
char *pString = stringArray;
printf("%lu\n", sizeof(pString));
```

Primitive types in C

integer types

- char, int

floating point

- float, double

modifiers

- short [int]
- long [int, double]
- signed [char, int]
- unsigned [char, int]

typical sizes

type	bytes (32 bit)	bytes (64 bit)	32 bit range	printf
char	1	1	[0, 255]	%c
short int	2	2	[-32768,32767]	%hd
unsigned short int	2	2	[0, 65535]	%hu
int	4	4	[-214748648, 2147483647]	%d
unsigned int	4	4	[0, 4294967295]	%u
long int	4	8	[-2147483648, 2147483647]	%ld
long long int	8	8	[-9223372036854775808, 9223372036854775807]	%lld
float	4	4	approx [10^{-38}, 10^{38}]	%f
double	8	8	approx [10^{-308}, 10^{308}]	%lf
long double	12	16	approx [10^{-4932}, 10^{4932}]	%Lf
pointer	4	8	[0, 4294967295]	%p

C99 extended integer types

Portable, known size integers

```
#include <stdint.h>

void foo(void) {
  int8_t  w;     // exactly 8 bits, signed
  int16_t x;     // exactly 16 bits, signed
  int32_t y;     // exactly 32 bits, signed
  int64_t z;     // exactly 64 bits, signed

  uint8_t a;     // exactly 8 bits, unsigned
  ...etc.
}
```

Types: Type Checking

Type information is used to determine which version of an operator to apply:

conversion.c

```
long int x;
float y;
struct { int a; int b; } z;
x = y;
y = x + y;
x = y + x;
x = z;     // error
```

Types: Explicit conversion `casting`

Type information is used to determine which version of an operator to apply:

conversion.c

```
long int x;
float y;
struct { int a; int b; } z;
x = y;
y = x + y;
x = y + x;
x = (long int)z;     // error
x = *(long int*)&z; // okay!
```

Types: Implicit Conversions

- While it's best to explicitly cast, C will implicitly convert operands so that it can successfully apply operators

- Hierarchy of types

```
char < short < int < long < long long <
    float < double < long double
```

- *Integer promotion*
  - If operand is of rank lower than int, promote to int

Types: Implicit Conversions

- Usual arithmetic conversions
  - If either operand is a floating point type, convert operand of lower rank to same rank as floating type
  - If both are integer types, use *integer promotion* (promote to integer, and if that doesn't work try unsigned int, and…)
- Note: conversion from int to float can lose precision!

Arithmetic Conversions

intFloat.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int x = 123456789;
  float y;
  y = x;
  x = y;
  printf("%d\n", x);

  return EXIT_SUCCESS;
}
```

This compiles with no warnings

It prints 123456792

Non-arithmetic conversions

- Arrays are converted to pointer to the base type, *except when*
  - the array is the operand of `sizeof()`
  - the array is the operand of the address operator &
    - when a string literal is used to initialize an array of `char`
- Function names are implicitly converted to a pointer to a function
  - *except* when used as operand of the address operator &

Implicit Pointer Conversion

- Any pointer can be implicitly converted to `void*`
  - And *vice versa*
- A pointer of a given type can be converted to a pointer of a more qualified type
  - int* to const int*
- A *null pointer constant* (NULL) can be converted to any pointer type

Implicit Pointer Conversion

pointerConversion.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  void *p = NULL;
  int i;
  int *pI;
  const int *cpI;

  // none of the statements generate errors or warnings
  pI = &i;
  cpI = pI;
  pI = p;

  pI = malloc(sizeof(int));

  if ( p == cpI ) printf("equal\n");
  else printf("not equal\n");

  return EXIT_SUCCESS;
}
```

Explicit Pointer Conversion

- You can cast any pointer type to any other type
  - It's guaranteed that if you eventually cast it back to the original pointer type, it has the same value
    - p == (int*)(float*)(char*)p is always true

Conversions: When?

conversionOpportunities.c

```
#include <stdio.h>
#include <stdlib.h>

double sub(int x) {
  return x;          // conversion to return type
}

int main(int argc, char *argv[]) {
  int i = 2.0;       // conversion on initialization
  double d = 2.0;
  d = i + d;         // conversion on operator (+): i -> double
  i = sub(d);        // conversion of argument on call
                     // conversion of result on assignment
  printf("%d\n", i);// prints 4

  return EXIT_SUCCESS;
}
```

Conversions: Performance

Sit in a loop calling passing a double argument to either

```
double sub(double x){return x;}
```

or

```
int sub(double x){return x;}
```

The version that doesn't have to convert is about 25% faster than the version that does.

Revisiting Declarations `scope segment`

- An identifier declared outside of any code block has global scope
- An identifier declared within a block has scope that is the rest of that block

```
int globalX;
int main(int argc, char *argv[]) {
    for (int i=0; i<10; i++) {
      int y = i;
    }
    return 0;
}
```

Scope example

scope.c

```
int globalX;

int main(int argc, char *argv[]) {
  for (int i=0; i<10; i++) {
    int y = i;
    printf("y = %d\n", y);
  }

  for (i=0; i<4; i++) {   // error
    y += i;               // error
  }

  return EXIT_SUCCESS;
}

int globalX;            // okay
float globalX;          // error

int globalX() {         // error
  return 2;
}
```

The Keyword static `globals`

- Applied to a global, it means the identifier has file scope
  - it has meaning only within the rest of that file
  - that identifier will not be exported to the linker, so there's no way to use it in some other file
- ```
  static int globalX;
  static void privateFunc(char *str) { …}
  ```
  ```

The Keyword static `locals`

- Applied to a local, it means reserve memory in the global data segment, not on the stack
  - the scope is still local
  - but a new copy of the variable is **not** created each time the block is entered
- ```
  void sub() {
      static int counter = 4;
      printf("%d ", counter);
      if ( counter-- > 0 ) sub();
  }
  ```
  - When invoked a first time print 4 3 2 1 0
  - When invoked a second time prints -1
  ```

For next time

- The first project assignment is out
- Next time:
  - the preprocessor
  - pointers: arrays, strings, functions