CS/SE 3377

C - Review

Sridhar Alagar

C

- Created in 1972 by Dennis Ritchie
  - Designed for creating system software
  - Portable across machine architectures
  - Most recently updated in 1999 (C99) and 2011 (C11)

- Characteristics
  - "Low-level" language that allows us to exploit underlying features of the architecture – but easy to fail spectacularly (!)
  - Procedural (not object-oriented)
  - "Weakly-typed" or "type-unsafe"
  - Small, basic library compared to Java, C++

Generic C Program Layout

```
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
  /* the innards */
}

/* define other functions */
```

C: main

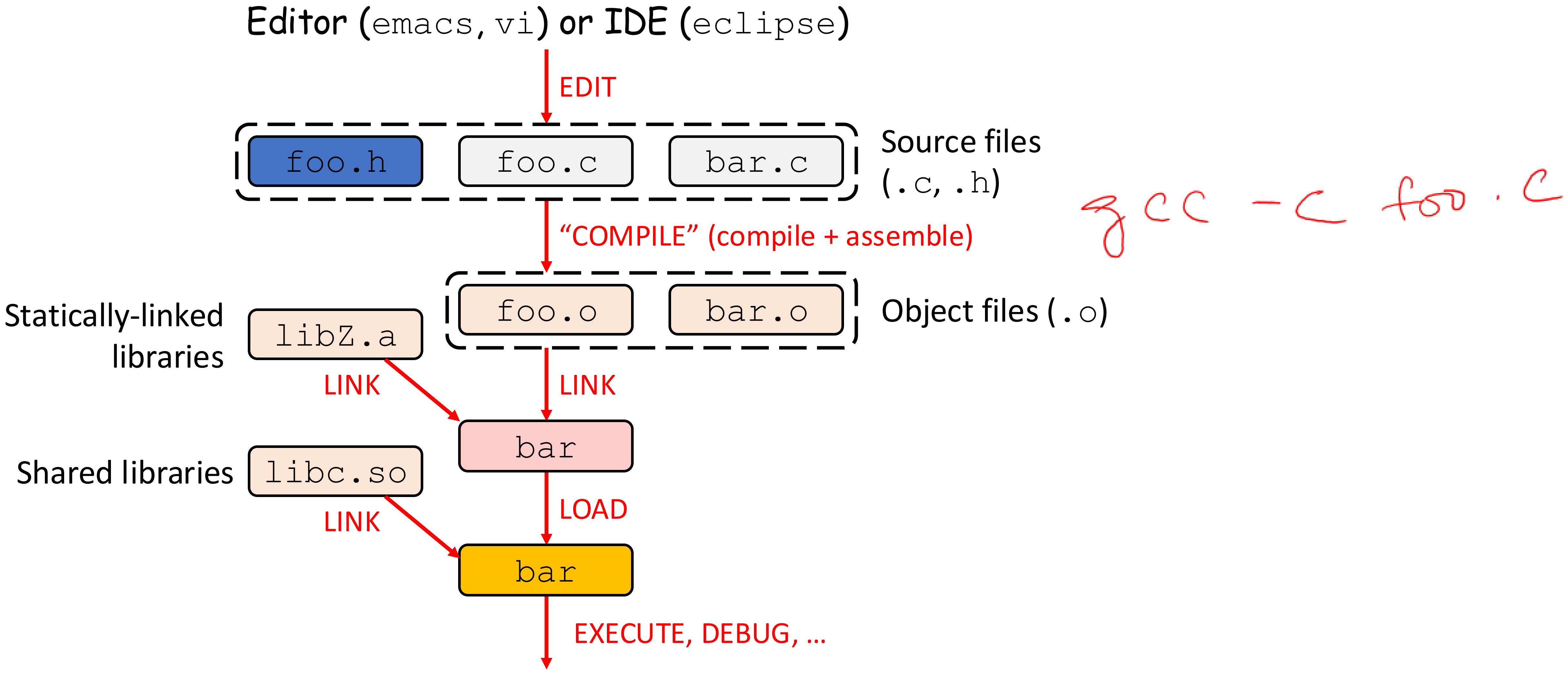
- To get command-line arguments in `main`, use:

```
int main(int argc, char* argv[])
```

- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

- Example: `$ foo hello 877`
  - `argc = 3`
  - `argv[0]="foo", argv[1]="hello", argv[2]="877"`

C programming to execution



[Check this link for how to create static and shared libraries]

C to machine code

```
void sumstore(int x, int y,
              int* dest) {
  *dest = x + y;
}
```

C source file
(sumstore.c)

C compiler (gcc -S)

C compiler
(gcc -c)

```
sumstore:
        addl    %edi, %esi
        movl    %esi, (%rdx)
        ret
```

Assembly file
(sumstore.s)

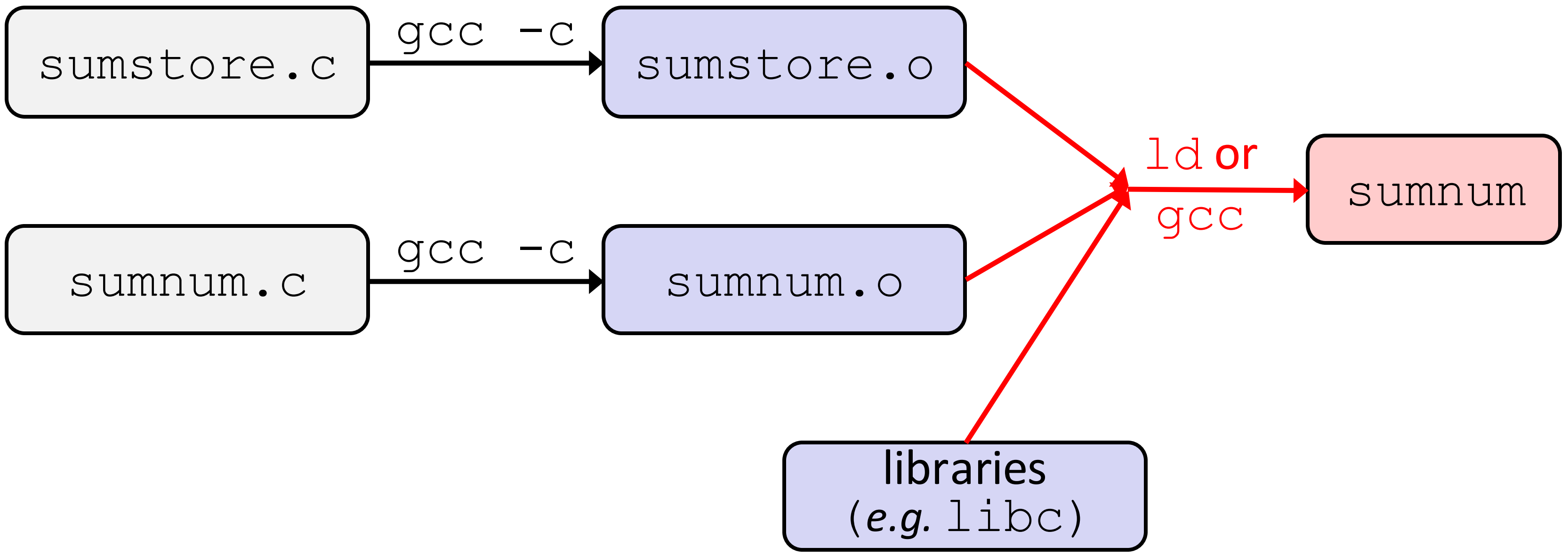
Assembler (gcc -c or as) → gas

```
400575: 01 fe
        89 32
        c3
```

Machine code
(sumstore.o)

Compiling Multi-file Programs

- The **linker** combines multiple object files plus statically-linked libraries to produce an executable
  - Includes many standard libraries (*e.g.* `libc`, `crt1`)
    - A *library* is just a pre-assembled collection of `.o` files



When things go wrong…

- Errors and Exceptions
  - C does not have exception handling (no `try`/`catch`)
  - Errors are returned as integer error codes from functions
    - Standard codes found in `stdlib.h`:
      `EXIT_SUCCESS` (usually 0) and `EXIT_FAILURE` (non-zero)
    - Return value from **`main`** is a status code
  - Because of this, error handling is ugly and inelegant

- Crashes
  - If you do something bad, you hope to get a "segmentation fault" (believe it or not, this is the "good" option)

Java vs C

- Are Java and C mostly similar (S) or significantly different (D) in the following categories?

Language Feature	S/D	Differences in C
Control structures	S	
Primitive datatypes	S/D	char – 1 byte ASCII
Operators	S	>>> not in C
Casting	D	not strongly type
Arrays	D	not an object. don't know length
Memory management	|	no garbage collector. free()

Primitive types in C

- Integer types
  - `char`, `int`
- Floating point
  - `float`, `double`
- Modifiers
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p

C99 extended integer types

- Solves the conundrum of "how big is a `long int`?"

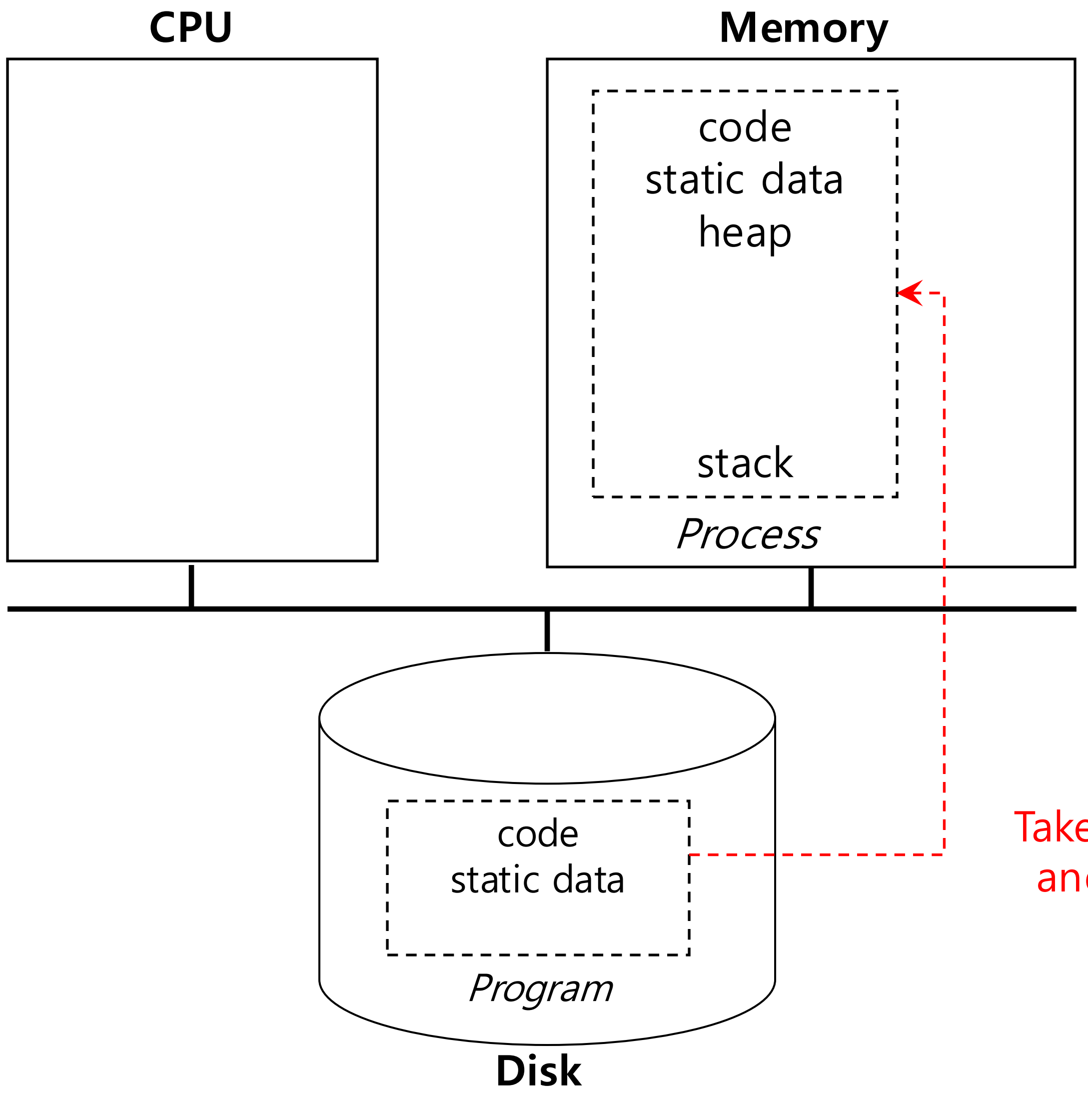
```
#include <stdint.h>

void foo(void) {
  int8_t  a;  // exactly 8 bits, signed
  int16_t b;  // exactly 16 bits, signed
  int32_t c;  // exactly 32 bits, signed
  int64_t d;  // exactly 64 bits, signed
  uint8_t w;  // exactly 8 bits, unsigned
  ...
}
```

```
void sumstore(int x, int y, int* dest) {
```

```
void sumstore(int32_t x, int32_t y, int32_t* dest) {
```

Running Program's Layout

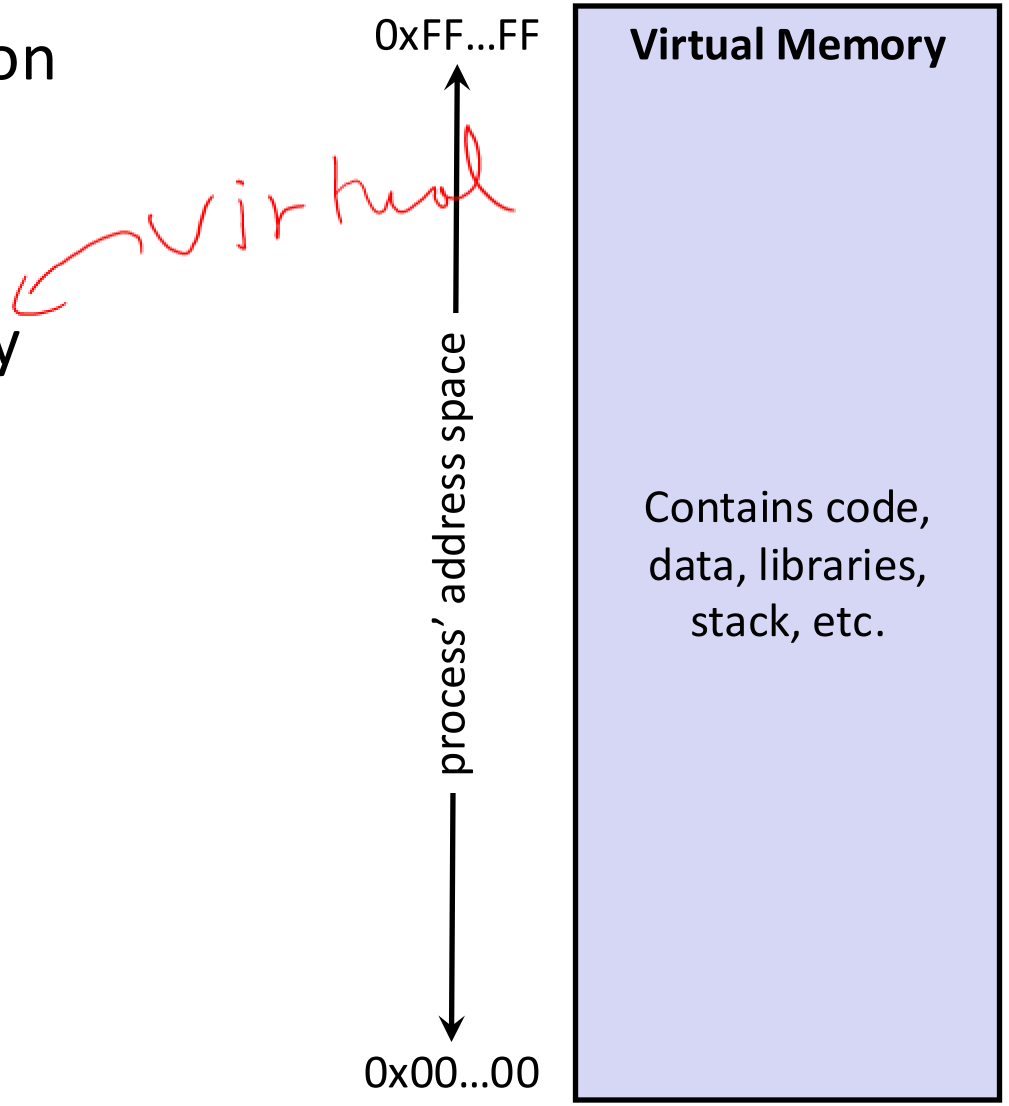
Executing a Program



Loading:
Takes on-disk program and loads it into the memory

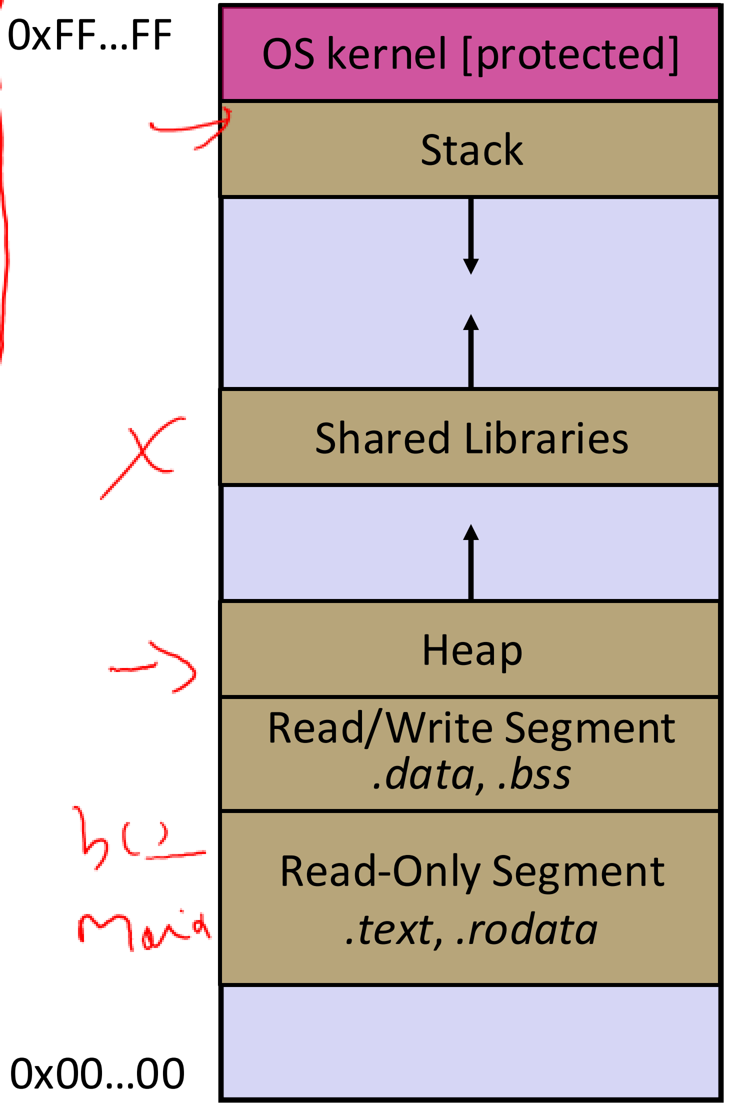
Process and its memory

- A process is a program in execution
- The OS gives each process the illusion of its own private memory
  - Called the process' address space
  - 2^{64} bytes on a 64-bit machine



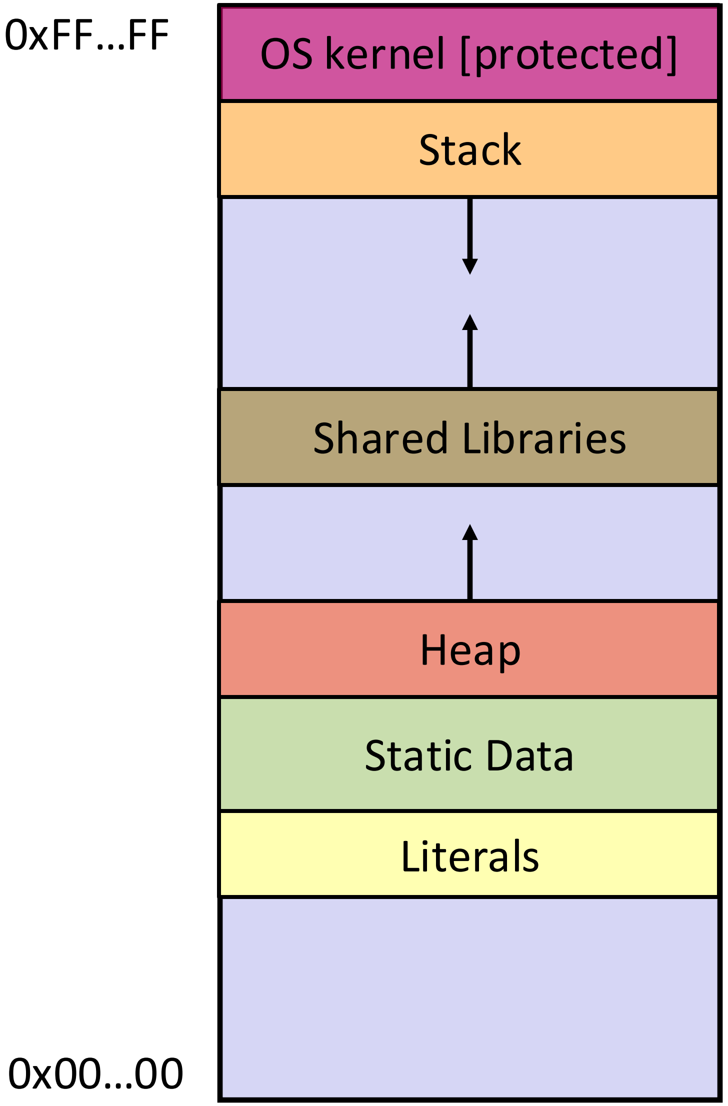
Loading

- When the OS loads a program it:
  1) Creates an address space
  2) Inspects the executable file to see what's in it
  3) Copies regions of the file into the right place in the address space
  4) Does any final linking, relocation, or other needed preparation



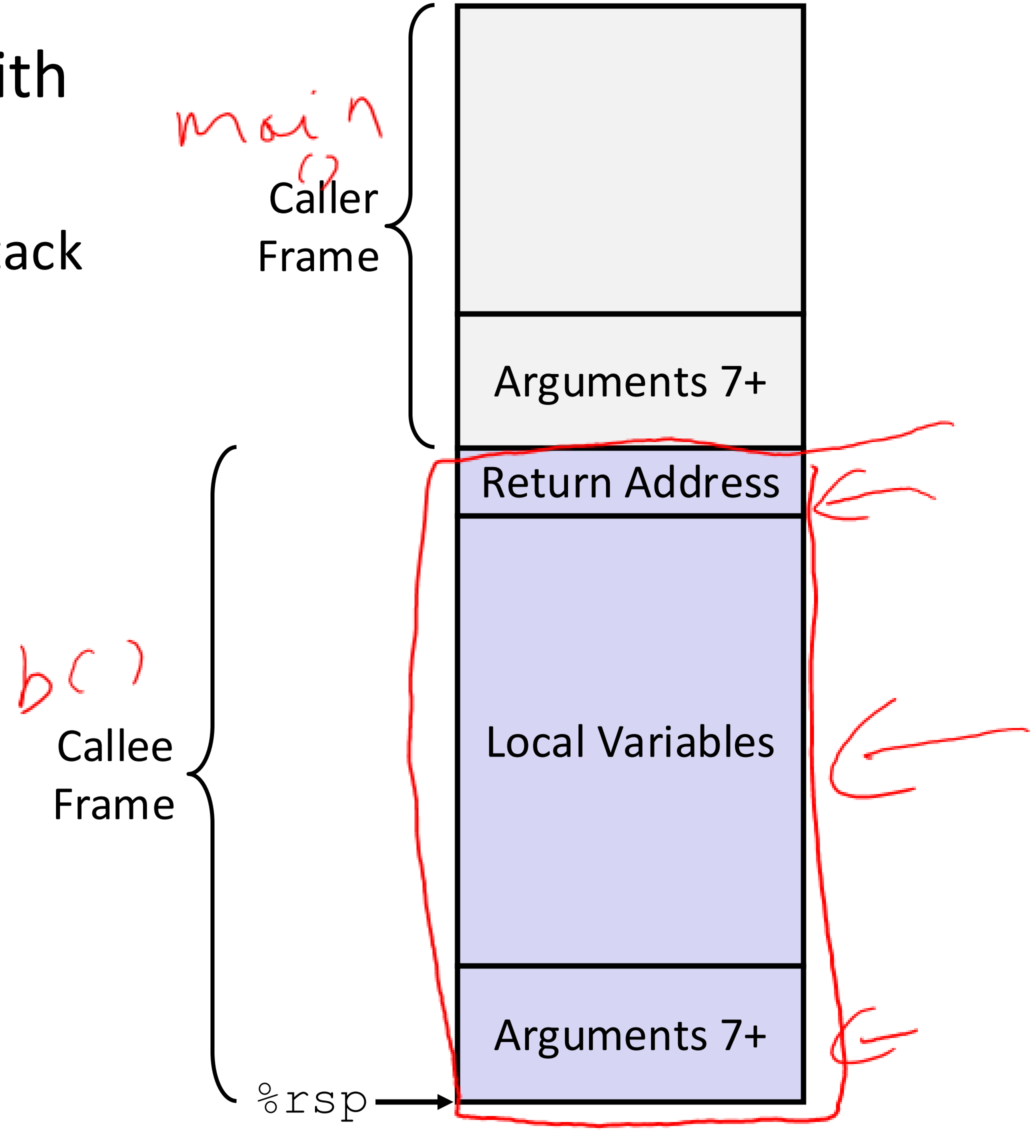
Memory management

- *Local* variables on the Stack
  - Allocated/freed during functions call/ret (`push`, `pop`, `mov`)
- *Global* and *static* variables in Data
  - Allocated/freed when the process starts/exits
- *Dynamically-allocated* data on the Heap
  - `malloc()` to request; `free()` to free, otherwise memory leak



The Stack

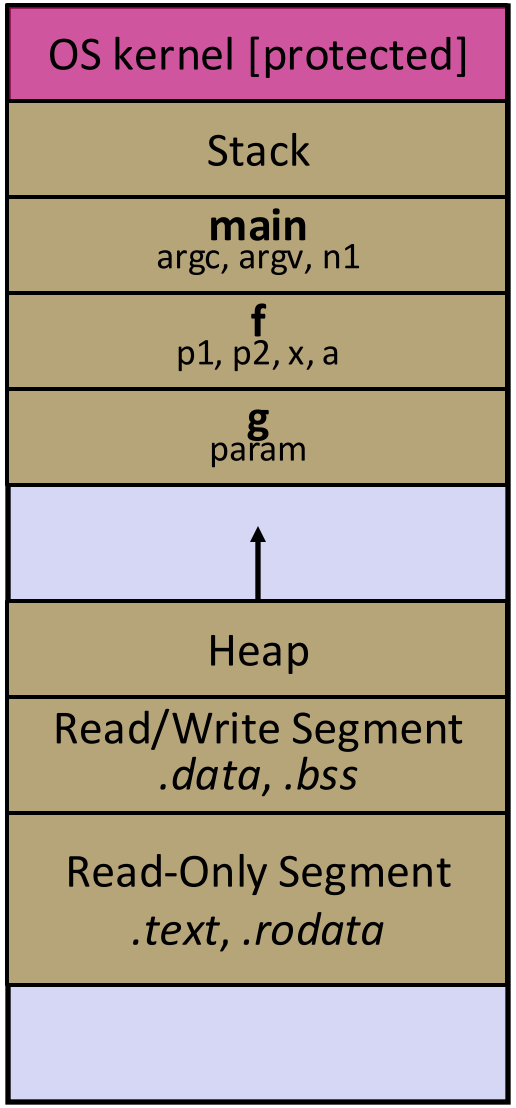
- Used to store data associated with function calls
  - Compiler-inserted code manages stack frames for you
- Stack frame typically includes:
  - Address to return to
  - Local variables
  - Argument build
    - Only if > 6 used



Stack in action

stack.c

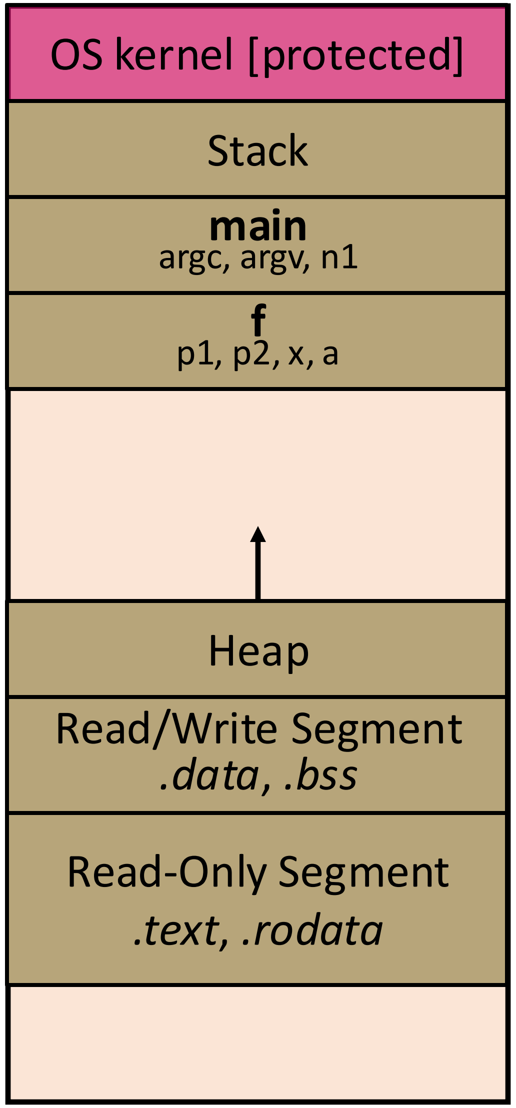
```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
  int32_t n1 = f(3,-5);
  n1 = g(n1);
  return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
  int32_t x;
  int32_t a[3];
  ...
  x = g(a[2]);
  return x;
}

int32_t g(int32_t param) {
  return param * 2;
}
```



Stack in action

stack.c

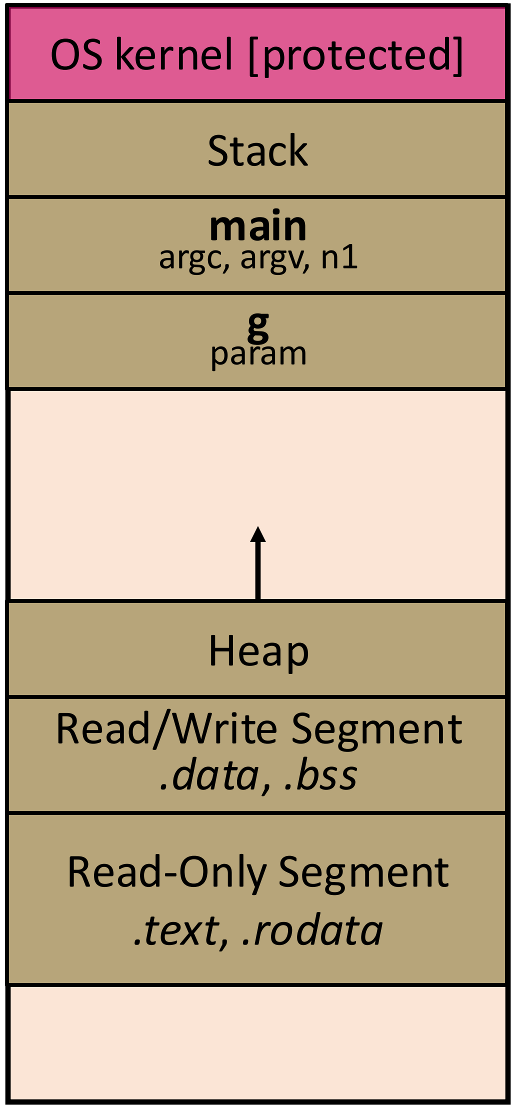
```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
  int32_t n1 = f(3,-5);
  n1 = g(n1);
  return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
  int32_t x;
  int32_t a[3];
  ...
  x = g(a[2]);
  return x;
}

int32_t g(int32_t param) {
  return param * 2;
}
```



Stack in action

stack.c

```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
  int32_t n1 = f(3,-5);
  n1 = g(n1);
  return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
  int32_t x;
  int32_t a[3];
  ...
  x = g(a[2]);
  return x;
}

int32_t g(int32_t param) {
  return param * 2;
}
```



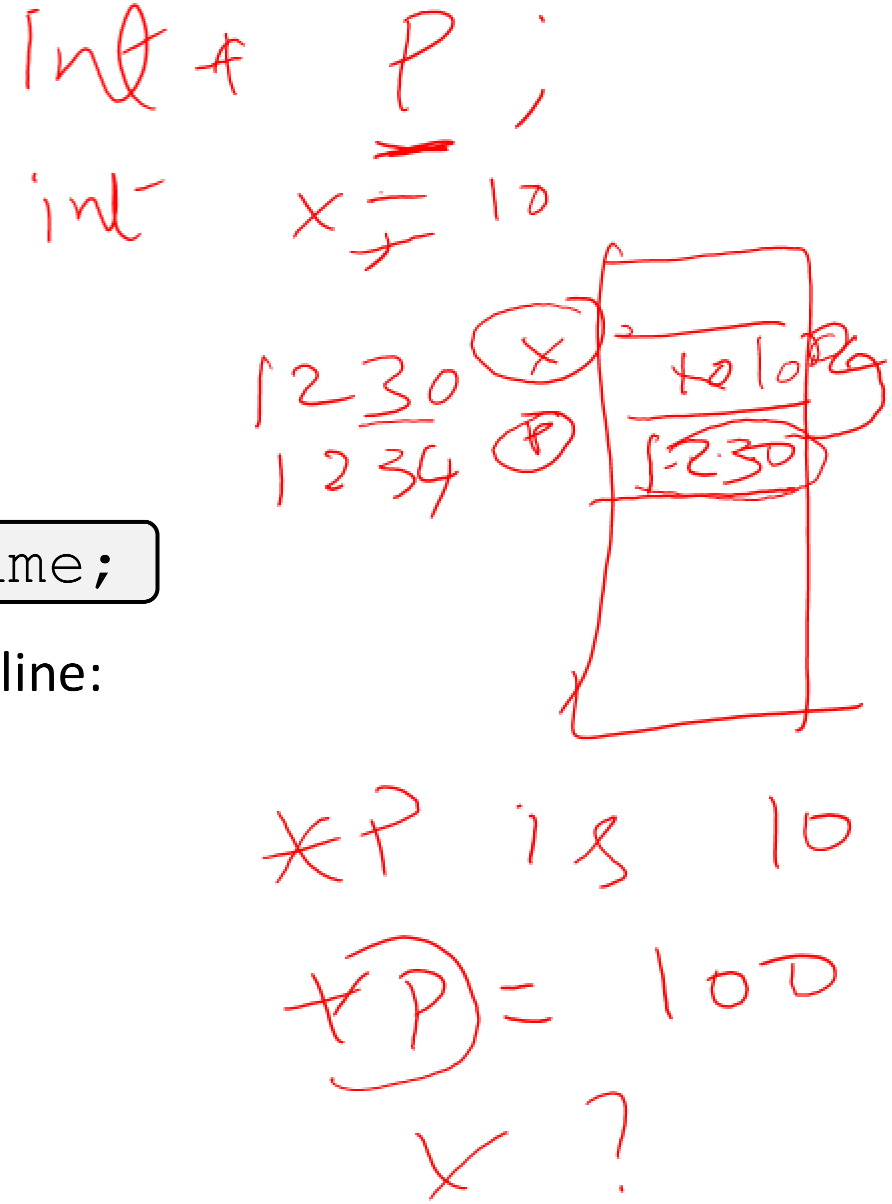
Pointers

Pointers - basics

- **Variables** that store addresses
  - It points to somewhere in the process' virtual memory
  - `&foo` produces the virtual address of `foo`

- Generic definition: `type* name;` or `type *name;`
  - Recommended: do not define multiple pointers on same line: `int* p1, p2;` not the same as `int *p1, *p2;`
  - Instead, use:

```
int *p1;
int *p2;
```



- *Dereference* a pointer using the unary `*` operator
  - Access the memory referred to by a pointer

Pointer example

pointer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

int main(int argc, char** argv) {
  int32_t x = 351;
  int32_t* p;      // p is a pointer to a int

  p = &x;      // p now contains the addr of x
  printf("&x is %p\n", &x);
  printf(" p is %p\n",  p);
  printf(" x is %d\n",  x);

  *p = 333;   // change the value of x
  printf(" x is %d \n",  x);

  return EXIT_SUCCESS;
}
```

Pointers illustrated

boxarrow.c

```
int main(int argc, char** argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int* p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

  return EXIT_SUCCESS;
}
```

address	name	value

Pointers illustrated

boxarrow.c

```c
int main(int argc, char** argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int* p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

  return EXIT_SUCCESS;
}
```

address	**name**	value

&x	**x**	value
&arr[2]	**arr[2]**	value
&arr[1]	**arr[1]**	value
&arr[0]	**arr[0]**	value
&p	**p**	value

Pointers illustrated

boxarrow.c

```
int main(int argc, char** argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int* p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

  return EXIT_SUCCESS;
}
```

address	name	value

address	name	value
&x	**x**	1
&arr[2]	**arr[2]**	4
&arr[1]	**arr[1]**	3
&arr[0]	**arr[0]**	2
&p	**p**	&arr[1]

Pointers illustrated

boxarrow.c

```
int main(int argc, char** argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int* p = &arr[1];

  printf("&x: %p;  x: %d\n", &x, x);
  printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
  printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
  printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

  return EXIT_SUCCESS;
}
```

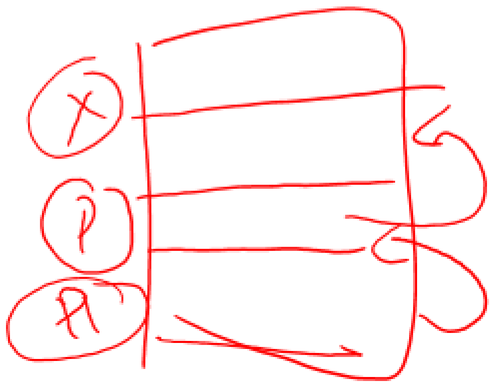
address	name	value
0x7fff…4c	**x**	1
0x7fff…48	**arr[2]**	4
0x7fff…44	**arr[1]**	3
0x7fff…40	**arr[0]**	2
0x7fff…38	**p**	0x7fff…44

Pointers key takeaway

int x
int* p = &x
int** p1 = &p

"Pointers are just variables that contain memory addresses"

"Since pointers are variables, we can do all these things recursively!"



Pointer arithmetic

int * p = 1234

p = p + 1

1235

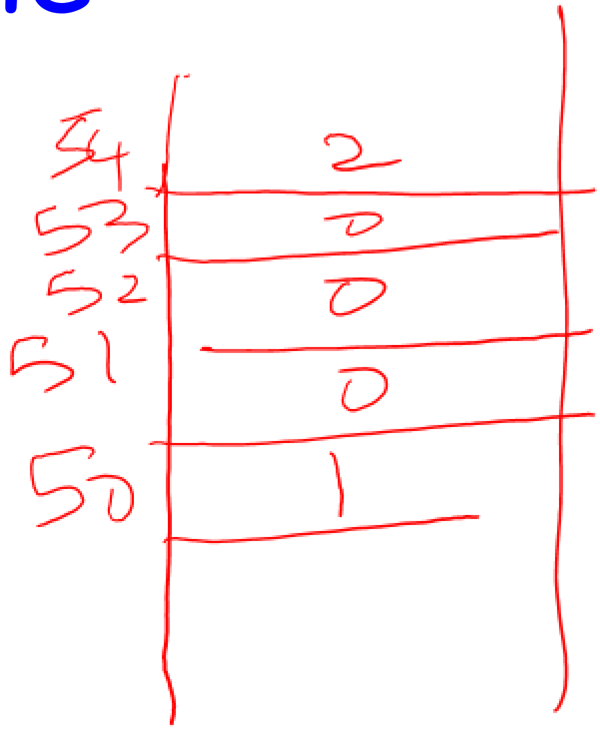
p = p + sizeof(int)

- Pointers are typed
  - Tells the compiler the size of the data you are pointing to
- Pointer arithmetic is scaled by sizeof(*p)
  - Works nicely for arrays
- Valid pointer arithmetic:
  - Add/subtract an integer to/from a pointer
  - Subtract two pointers (within stack frame or malloc block)
  - Compare pointers (<, <=, ==, !=, >, >=), including NULL
  - … but plenty of valid-but-inadvisable operations, too

little endian

Pointer Arithmetic - Example

```c
int main(int argc, char** argv) {
  int arr[3] = {2, 3, 4};
  int* p = &arr[0];
  int* p1 = &arr[1];
  int* p2 = p1 + 1;

}
```



Function declaration

- Informs the compiler arguments and return types; function definitions can then be in a logical order
  - Function comment usually by the *prototype*

```
// sum of integers from 1 to max
int32_t sumTo(int32_t);   // func prototype

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return EXIT_SUCCESS;
}

int32_t sumTo(int32_t max) {
  int32_t i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

Function Declaration vs. Definition

- C/C++ make a careful distinction between these two

- Definition: the *thing* itself
  - *e.g.* code for function, variable definition that creates storage
  - Must be **exactly one** definition of each thing (no duplicates)

- Declaration: description of a thing
  - *e.g.* function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in **all files** that use that thing
    - Should appear before first use

C is Call-By-Value

- C (and Java) pass arguments by *value*
  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller's copy *isn't* modified

```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(a, b);
  ...
```

Faking Call-By-Reference in C

- Can use pointers to *approximate* call-by-reference
  - Callee still receives a **copy** of the pointer (*i.e.* call-by-value), but it can modify something in the caller's scope by dereferencing the pointer parameter

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char** argv) {
  int a = 42, b = -7;
  swap(&a, &b);
  ...
```

Basic data structure

- C does not support objects!!!
- **Arrays** are contiguous chunks of memory
  - Arrays have no methods and do not know their own length
  - Can easily run off ends of arrays in C – security bugs!!!
- **Strings** are null-terminated char arrays
  - Strings have no methods, but `string.h` has helpful utilities

```c
char* x = "hello\n";
```

x →

h	e	l	l	o	\n	\0

- **Structs** are the most object-like feature, but are just collections of fields – no "methods" or functions

Arrays

- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (*e.g.* `int32_t scores[175];`)

- Size of an array
  - Not stored anywhere – array does not know its own size!
    - `sizeof(array)` only works in variable scope of array definition

Array as parameters

- It's tricky to use arrays as parameters
  - What happens when you use an array name as an argument?
  - Recall: arrays do not know their own size

```
// prototype
int32_t sumAll(int32_t a[]);

int main(int argc, char** argv) {
  int32_t numbers[] = {9, 8, 1, 9, 5};
  int32_t sum = sumAll(numbers);
  return EXIT_SUCCESS;
}

int32_t sumAll(int32_t a[]) {
  int32_t i, sum = 0;
  for (i = 0; i < ...???
}
```

Solution: Pass Size as Parameter

```c
// prototype
int32_t sumAll(int32_t a[], int size);

int main(int argc, char** argv) {
  int32_t numbers[] = {9, 8, 1, 9, 5};
  int32_t sum = sumAll(numbers, 5);
  printf("sum is: %d\n", sum);
  return EXIT_SUCCESS;
}

int32_t sumAll(int32_t a[], int size) {
  int32_t i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}
```

arraysum.c

- This is the standard idiom in C programs

sizeof(numbers) / sizeof(numbers[0])

Returning an Array

- Local variables, including arrays, are allocated on the Stack
  - They “disappear” when a function returns!
  - Can’t safely return local arrays from functions

```
int32_t* copyArray(int32_t src[], int32_t size) {
  int32_t i, dst[size];    // OK in C99

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }

  return dst;   // no compiler error, but wrong!
}
```

Solution: Output Parameter

- Create the "returned" array in the caller
  - Pass it as an output parameter to `copyarray()`
    - A pointer parameter that allows the called function to store values that the caller can use
  - Works because arrays are "passed" as pointers
    - "Feels" like call-by-reference, *but technically it's not*

main( ) {
int a[5] = {1,2,3,4}
int b[5]
copyArray(a, b, 5)

```
void copyArray(int32_t src[], int32_t dst[], int32_t size) {
  int32_t i;

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }
}
```

Arrays: Call-By-Value or Call-By-Reference?

- **Technical answer:** a `T[]` array parameter is "decayed" to a pointer of type `T*`, and the *pointer* is passed by value
  - So it acts like a call-by-reference array (if callee changes the array parameter *elements* it changes the caller's array)
  - But it's really a call-by-value pointer (the callee can change the pointer *parameter* to point to something else(!))

```
void copyArray(int32_t src[], int32_t dst[], int32_t size) {
  int32_t i;
  int32_t copy[size];   // OK in C99, still stylistically bad
  for (i = 0; i < size; i++) {
    copy[i] = src[i];
  }
  dst = copy;              // doesn't change caller's array
}
```

Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches

malloc()

- General usage: `var = (type*) malloc(size in bytes)`

- `malloc` allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns `NULL` if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Behave similarly to primitive variables
- Generic declaration:

```
struct tagname {
  type1 name1;
  ...
  typeN nameN;
};
```

```
// the following defines a new
// structured datatype called
// a "struct Point"
struct Point {
  float x, y;
};

// declare and initialize a
// struct Point variable
struct Point origin = {0.0,0.0};
```

free()

- Usage: `free(pointer);`

- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **`malloc`**)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to free
    - Defensive programming: can set pointer to `NULL` after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...              // do stuff with arr
free(arr);
arr = NULL;      // OPTIONAL
```

Using struct

- Use "." to refer to a field in a struct
- Use "->" to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
struct Point {
  float x, y;
};

int main(int argc, char** argv) {
  struct Point p1 = {0.0, 0.0};  // p1 is stack allocated
  struct Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return EXIT_SUCCESS;
}
```

simplestruct.c

Dynamically allocated Structs

- You can **`malloc`** and **`free`** structs, just like other data type
  - `sizeof` is particularly helpful here

```
// a complex number is a + bi
typedef struct complex_st {
  double real;    // real component
  double imag;    // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
  Complex* retval = (Complex*) malloc(sizeof(Complex));
  if (retval != NULL) {
    retval->real = real;
    retval->imag = imag;
  }
  return retval;
}
```

complexstruct.c

Strings

- **Strings** are not explicitly defined
- **Strings** are null-terminated char arrays
  - Strings have no methods, but `string.h` has helpful utilities

```c
char* x = "hello\n";
```

x →

h	e	l	l	o	\n	\0

gdb – Gnu debugger

- Must learn to use it. Otherwise, debugging can be miserable
- Very useful to understand code also
- Source code should be compiled with '-g' option to use gdb
- Check https://sourceware.org/gdb

Disclaimer

Some of the materials in this lecture slides are from the lecture slides of CS333 Univ of Washington

Multiple C programs

C source file 1 (sumstore.c)

```
void sumstore(int x, int y, int* dest) {
  *dest = x + y;
}
```

C source file 2 (sumnum.c)

```
#include <stdio.h>

void sumstore(int x, int y, int* dest);

int main(int argc, char** argv) {
  int z, x = 351, y = 333;
  sumstore(x, y, &z);
  printf("%d + %d = %d\n", x, y, z);
  return 0;
}
```

Compile together:

```
$ gcc -o sumnum sumnum.c sumstore.c
```