计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (百人计划研究员)

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800

Levels of Representation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

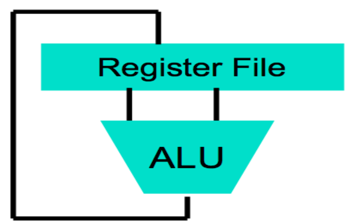
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

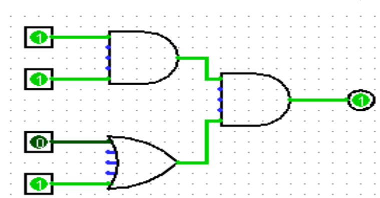
Current Focus

```
lw   t0, 0(s2)
lw   t1, 4(s2)
sw   t1, 0(s2)
sw   t0, 4(s2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
               0101 1000 0000 1001 1100 0110 1010 1111
```





Hello World

C

```c
#include <stdio.h>

int main(void) {
	printf("Hello World!\n");
	return 0;
}
```

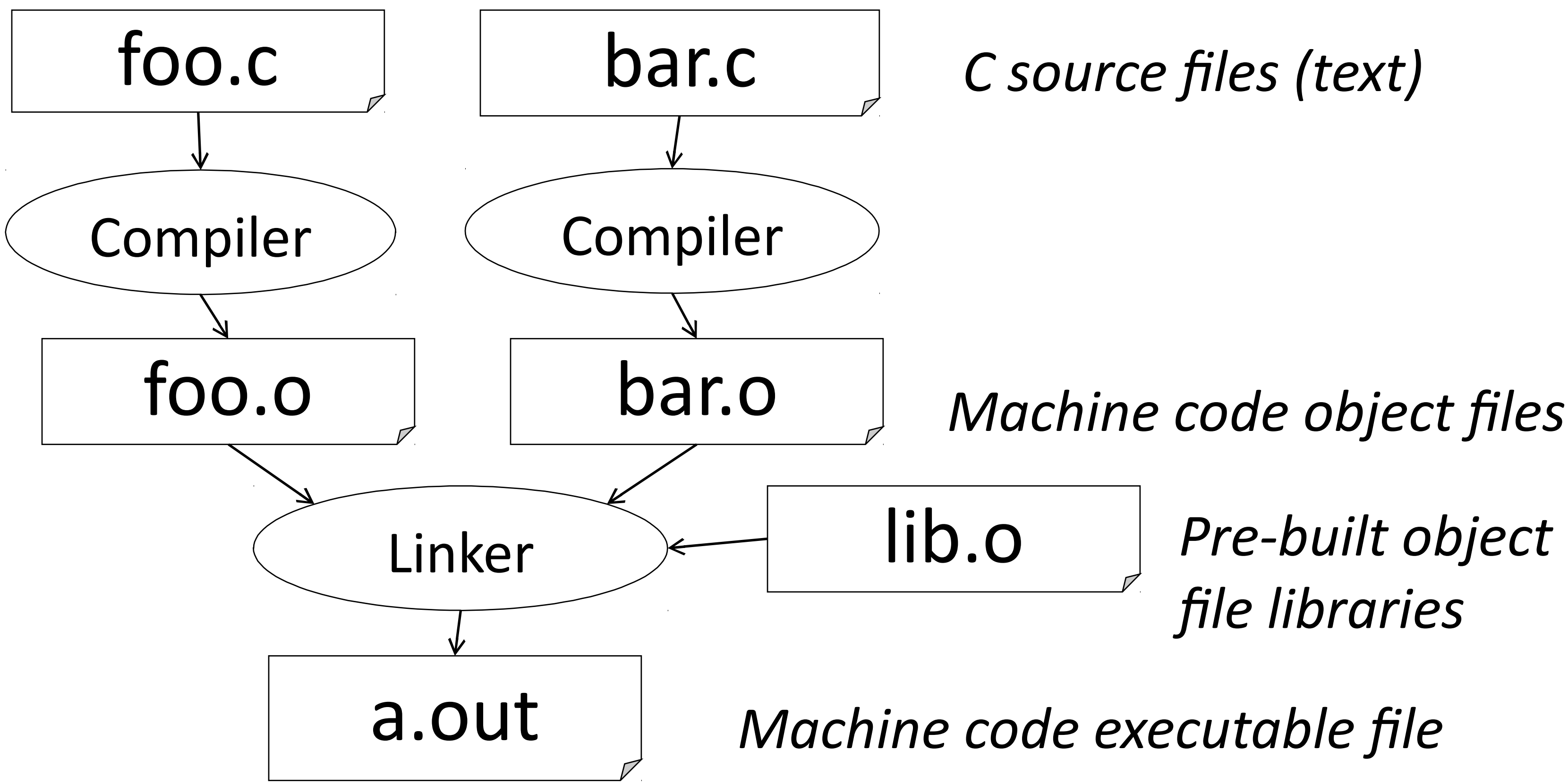
Compilation & Running

$ gcc HelloWorld.c
$ ./a.out
Hello World!
$

Java

```java
public class L02_HelloWorld {
	public static void main(String args[]) {
		System.out.println("Hello World!");
	}
}
```

C Compilation Simplified Overview



Compilation versus Interpretation

- C (compiled)
  - Compiler (& linker) translates source into machine language
  - Machine language program is loaded by OS and directly executed by the hardware

- Python (interpreted)
  - Interpreter is written in some high-level language (e.g. C) and translated into machine language
  - Loaded by OS and directly executed by processor
  - Interpreter reads source code (e.g. Python) and “interprets” it

Java “Byte Code”

- Java compiler (javac) translates source to “byte code”

- “Byte code” is a particular assembly language
  - Just like i86, RISC-V, ARM, …
  - Can be directly executed by appropriate machine
    - implementations exist(ed), not commercially successful
  - More typically, “byte code” is
    - interpreted on target machine (e.g. i86) by java program
    - compiled to target machine code (e.g. by JIT)
  - Program runs on any computer with a “byte code” interpreter (more or less)

Compilation

- Excellent run-time performance:
  - Much faster than interpreted code (e.g. Python)
  - Usually faster than Java (even with JIT)

- Note: Computers only run machine code
  - Compiled application program, or
  - Interpreter (that interprets source code)

Compilation: Disadvantages

- Compiled files, including the executable, are
  - architecture-specific, depending on processor type
    - e.g., RISC-V vs. ARM
  - and the operating system
    - e.g., Windows vs. Linux

- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture

- “Change $\rightarrow$ Compile $\rightarrow$ Run [repeat]” iteration cycle can be slow during development
  - Recompile only parts of program that have changed
  - Tools (e.g. make) automate this

C Dialects

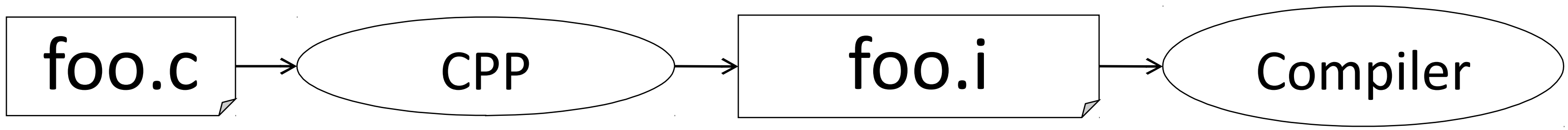
```
$ gcc x.c
```

```c
int main(void) {
    const int SZ = 5;
    int a[SZ];    // declare array
    for (int i=0; i<SZ; i++) a[i] = 0;
    return 0;
}
```

```
$ gcc -ansi -Wpedantic x.c
```

```c
#define SZ 5
int main(void) {
    int a[SZ];  /* declare array */
    int i;
    for (i=0; i<SZ; i++) a[i] = 0;
    return 0;
}
```

C Pre-Processor (CPP)



- C source files pass through macro processor, CPP, before compilation
- CPP replaces comments with a single space
- CPP commands begin with “#”

```
#include "file.h" /* Inserts file.h */
#include <stdio.h> /* Loads from standard loc */
#define M_PI (3.14159) /* Define constant */
#if/#endif /* Conditional inclusion of text */
```

- Use –save-temps option to gcc to see result of preprocessing
- Full documentation at: **`http://gcc.gnu.org/onlinedocs/cpp/`**

Typed Variables in C

```
int a = 4;
float f = 1.38e7;
char c = 'x';
```

- Declare before use
- Type cannot change
- Like Java

Type	Description	Examples
int	integers, positive or negative	0, 82, -77, 0xAB87
unsigned int	ditto, no negatives	0, 8, 37
float	(single precision) floating point	3.2, -7.9e-10
char	text character or symbol	'x', 'F', '?'
double	high precision/range float	1.3e100
long	integer with more bits	427943

Constants and Enumerations in C

- Constants
  - Assigned in typed declaration, cannot change
  - E.g.

```c
const float pi = 3.1415;
const unsigned long addr = 0xaf460;
```

- Enumerations

```c
#include <stdio.h>

int main() {
    typedef enum {red, green, blue} Color;
    Color pants = green;
    switch (pants) {
        case red:
            printf("red pants are hip\n");  break;
        case green:
            printf("green pants are weird\n");  break;
        default:
            printf("yet another color\n");
    }
    printf("pants = %d\n", pants);
}
```

Integers: Python vs. Java vs. C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

- C: int
  - integer type that target processor works with most efficiently
- Only guarantee:
  - sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - Also, short >= 16 bits, long >= 32 bits
  - All could be 64 bits
- Impacts portability between architectures

Variable Sizes: Machine Dependent!

```c
#include <stdio.h>
int main(void) {
    printf("sizeof ... (bytes)\n");
    printf("char:          %lu\n",
           sizeof(char));
    printf("short:         %lu\n",
           sizeof(short));
    printf("int:           %lu\n",
           sizeof(int));
    printf("unsigned int:  %lu\n",
           sizeof(unsigned int));
    printf("long:          %lu\n",
           sizeof(long));
    printf("long long:     %lu\n",
           sizeof(long long));
    printf("float:         %lu\n",
           sizeof(float));
    printf("double:        %lu\n",
           sizeof(double));
}
```

```
sizeof ... (bytes)
char:          1
short:         2
int:           4
unsigned int:  4
long:          8
long long:     8
float:         4
double:        8
```

Boolean

- No boolean datatype in C
  - Declare if you wish:

```
typedef int boolean;
const boolean false = 0;
const boolean true = 1;
```

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of pointer: more on this later)

- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Similar to Python:

    only 0's or empty sequences are false, everything else is true!

Functions in C

```
int number_of_people() {
  return 3;
}

void news() {
  printf("no news");
}

int sum(int x, int y) {
  return x + y;
}
```

- Like Java
- Declare return & argument types
- **`void`** for no value returned
- Functions MUST be declared before they are used

Uninitialized Variables

- Code

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

void undefined_local() {
    int x;   /* undefined */
    printf("x = %d\n", x);
}

void some_calc(int a) {
    a = a%2 ? rand() : -a;
}

int main(void) {
    for (int i=0;  i<5;  i++) {
        some_calc(i*i);
        undefined_local();
    }
}
```

- Output

```
$ gcc test.c

$ ./a.out
x = 0
x = 16807
x = -4
x = 282475249
x = -16
```

Struct's in C

- Struct's are structured groups of variables
- A bit like Java classes, but no methods
- E.g.

```
#include <stdio.h>

int main(void) {
    typedef struct { int x, y; } Point;
    Point p1;
    p1.x = 0;   p1.y = 123;

    Point p2 = { 77, -8 };
    printf("p2 at (%d,%d)\n", p2.x, p2.y);
}
```

More C …

- Lecture does not cover C completely
  - You’ll still need your C reference for this course
  - K&R, The C Programming Language
  - & other references on the course website

- Next few lectures’ focus:
  - Pointers & Arrays
  - Memory management

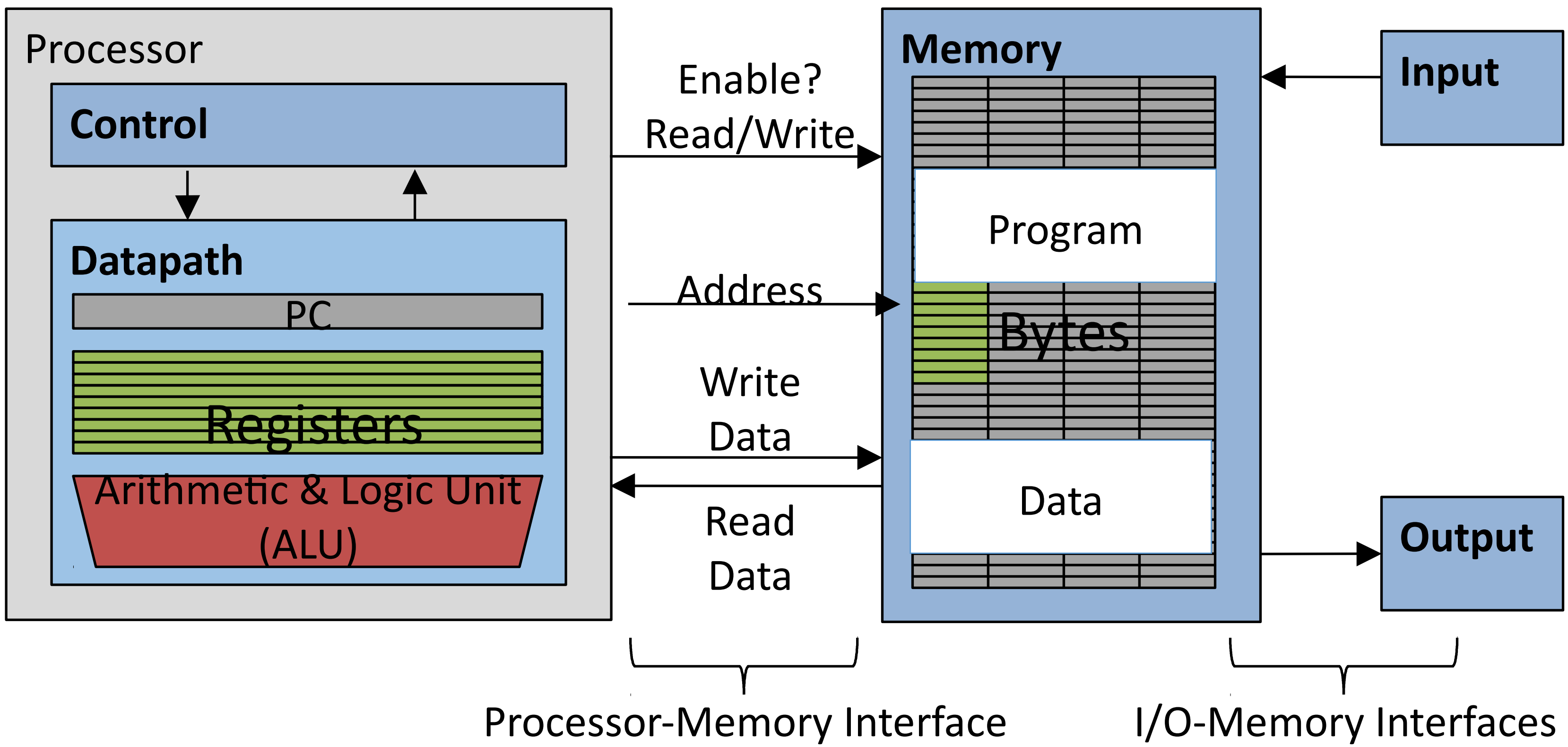
And In Conclusion,…

- Signed integers represented in 2’s complement

- C Programming Language
  - Popular (still!)
  - Similar to Java, but
    - no classes
    - explicit pointers (next lecture)
  - Beware
    - variables not initialized
    - variable size (# of bits) is machine & compiler dependent

- C is compiled to machine code
  - Unlike Python or Java, which are interpreted
  - Compilation is faster than interpretation

Components of a Computer



Computer Memory

```
int a;

a = -85;

printf("%d", a);
```

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Do not confuse memory address and value.
Nor a street address with the person living there.

Pointers

- C speak for “memory addresses”

- Notation

```
int *x;    // variable x is an
       // address to an int
int y = 9;    // y is an int
x = &y;    // assign address of
   // y to x
       // “address operator”
int z = *x;   // assign what x is
          // pointing to to z
       // “dereference        //
operator”
*x = -7;  // assign -7 to what
   // x is pointing to
```

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

What are the values of x, y, z?

Pointer Type

- Pointers have types, like other variables
  - “type of object” the pointer is “pointing to”

- Examples:

```
int *pi;   // pointer to int
double *pd;  // pointer to double
char *pc; // pointer to char
```

Generic Pointer (void *)

- Generic pointer
  - Points to any object (int, double, …)
  - Does not “know” type of object it references
    (e.g. compiler does not know)

- Example:

```
void *vp;        // vp holds an address to
                 // object of ”arbitrary” type
```

- Applications
  - Generic functions e.g. to allocate memory
  - malloc, free
    - accept and return pointers of any type
    - see next lecture

Pointer to struct

```
// type declaration
typedef struct { int x, y; } Point;

// declare (and initialize) Point "object"
Point pt = { 0, 5 };

// declare (and initialize) pointer to Point
Point *pt_ptr = &pt;

// access elements
(*pt_ptr).x = (*pt_ptr).y;

// alternative syntax
pp->x = pp->y;
```

Your Turn!

```
#include <stdio.h>

int main(void) {
      int a = 3, b = -7;
      int *pa = &a, *pb = &b;
      *pb = 5;
      if (*pb > *pa) a = *pa - b;
      printf("a=%d b=%d\n", a, b);
}
```

Answer	a	b
RED	3	-7
GREEN	4	5
ORANGE	-4	5
YELLOW	-2	5

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

What's wrong with this Code?

```c
#include <stdio.h>

int main(void) {
    int a;
    int *p;
    printf("a = %d,  p = %p,  *p = %d\n",
           a, p, *p);
    return 0;
}
```

Output:

a = 1853161526,

p = 0x7fff5be57c08,

*p = 0

Pointers as Function Arguments

```
#include <stdio.h>

void f(int x, int *p) {
    x = 5;  *p = -9;
}

int main(void) {
    int a = 1, b = -3;
    f(a, &b);
    printf("a=%d b=%d\n", a, b);
}
```

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

- C passes arguments by value
  - i.e. it passes a copy
  - value does not change outside function

- To pass by reference use a pointer

Parameter Passing in Java

- “primitive types” (int, char, double)
  - <u>by value</u> (i.e. passes a copy)

- Objects
  - <u>by reference</u> (i.e. passes a pointer)
  - Java uses pointers internally
    - But hides them from the programmer
  - Mapping of variables to addresses is not defined in Java language
    - No address operator (&)
    - Gives JVM flexibility to move stuff around

Your Turn!

```
#include <stdio.h>

void foo(int *x, int *y) {
    if ( *x < *y ) {
        int t = *x;
        *x = *y;
        *y = t;
    }
}

int main(void) {
    int a=3, b=1, c=5;
    foo(&a, &b);
    foo(&b, &c);
    printf("a=%d b=%d\n", a, b);
}
```

Type	Name	Addr	Value
		…	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Answer	a	b	c
RED	5	3	1
GREEN	1	5	3
ORANGE	3	3	1
YELLOW	3	5	1

C Arrays

- Declaration:

```
// allocate space
// unknown content
int a[5];

// allocate & initialize
int b = { 3, 2, 1 };
```

- Element access:

```
b[1];
a[2] = 7;
```

- Index of first element: 0

Type	Name	Addr	Value
		…	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Beware: no array bound checking!

```
#include <stdio.h>

int main(void) {
    int a[] = { 1, 2, 3 };

    for (int i=0;  i<4;  i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

Output:
```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = -1870523725
```

Often the result is much worse:

- erratic behavior
- segmentation fault, etc.
- C does not know array length!
- Pass as argument into functions

Use Constants, Not Literals

- Assign size to constant
  - Bad pattern

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```

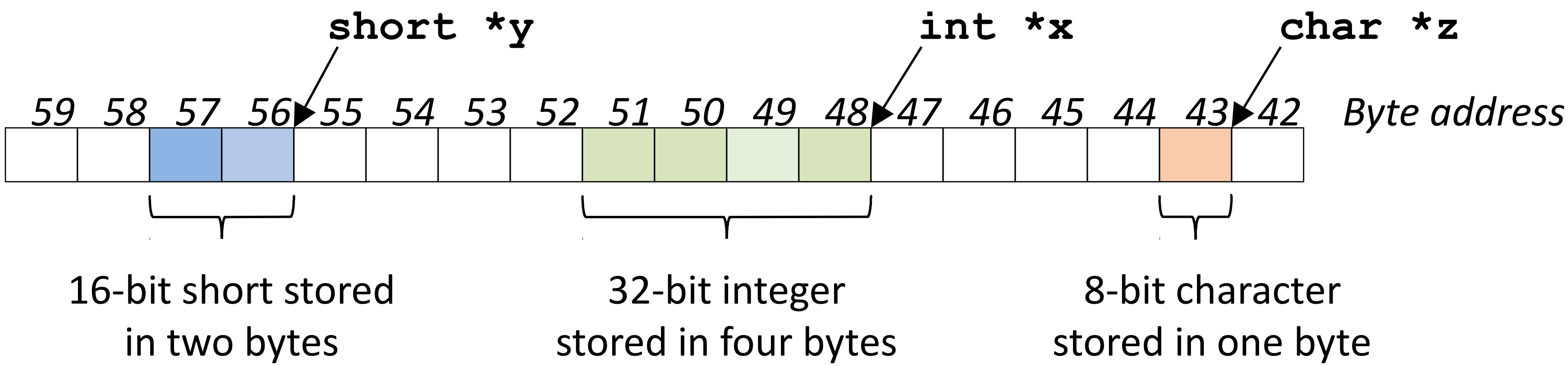
  - Better pattern

```
const int ARRAY_SIZE = 10;
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- “Single source of truth”
  - Avoiding maintaining two copies of the number 10
  - And the chance of changing only one
  - DRY: “Don’t Repeat Yourself”

Pointing to Different Size Objects

- Modern machines are “byte-addressable”
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes



sizeof() operator

```c
#include <stdio.h>

int main(void) {
    double d;
    int array[5];
    struct { short a; char c; } s;

    printf("double: %2lu\n", sizeof(d));
    printf("array:  %2lu\n", sizeof(array));
    printf("s:      %2lu\n", sizeof(s));
}
```

Output:

```
double:   8
array:   20
s:        4
```

- sizeof(type)
  - Returns number of bytes in object
  - Number of bits in a byte is not standardized
    - All modern computers: 8 bits per byte
    - Some “old” computers use other values, e.g. 6 bits per ”byte”
- By definition, in C

  `sizeof(char)==1`
- For all other types result is hardware and compiler dependent
  - Do not assume - Use sizeof!

Pointer Arithmetic - char

```c
#include <stdio.h>

int main(void) {
    char c[] = { 'a', 'b' };
    char *pc = c;
    pc++;
    printf("*pc=%c\n c=%p\npc=%p\npc-c=%ld\n",
            *pc, c, pc, pc-c);

    int i[] = { 10, 20 };
    int *pi = i;
    pi++;
    printf("*pi=%d\n i=%p\npi=%p\npi-i=%ld\n",
            *pi, i, pi, pi-i);
}
```

```
*pc  = b
c    = 0x7fff50f54b3e
pc   = 0x7fff50f54b3f
pc-c = 1
```

Type	Name	Byte Addr*	Value
		...	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		...	

*Computer only uses byte addresses. Tables with blue headers are simplifications.

Pointer Arithmetic - int

```c
#include <stdio.h>

int main(void) {
    char c[] = { 'a', 'b' };
    char *pc = c;
    pc++;
    printf("*pc=%c\n c=%p\npc=%p\npc-c=%ld\n",
            *pc, c, pc, pc-c);

    int i[] = { 10, 20 };
    int *pi = i;
    pi++;
    printf("*pi=%d\n i=%p\npi=%p\npi-i=%ld\n",
            *pi, i, pi, pi-i);
}
```

Type	Name	Byte Addr	Value
		...	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		...	

```
*pi  = 20
i    = 0x7fff50f54b40
pi   = 0x7fff50f54b44
pi-i = 1
```

*Computer only uses byte addresses. Tables with blue headers are simplifications.

Array Name / Pointer Duality

- Array variable is a "pointer" to the first (0^{th}) element
- Can use pointers to access array elements
  - `char *pstr` and `char astr[]` are nearly identical declarations
  - Differ in subtle ways: `astr++` is illegal
- Consequences:
  - `astr` is an array variable, but works like a pointer
  - `astr[0]` is the same as `*astr`
  - `astr[2]` is the same as `*(astr+2)`
  - Can use pointer arithmetic to access array elements

Arrays versus Pointer Example

```c
#include <stdio.h>

int main(void) {
    // array indexing
    int a[] = { 10, 20, 30 };
    printf("a[1]=%d, *(p+1)=%d, p[2]=%d\n",
            a[1], *(a+1), *(&a[2]));
    // pointer arithmetic
    int *p = a;
    p++;
    *p = 22;
    p[1] = 33;
    p[-1] = 11;
    for (int i=0;  i<3;  i++)
        printf("a[%d] = %d, ", i, a[i]);
}
```

Type	Name	Addr	Value
		...	
		104	
		103	
		102	
		101	
		100	
		...	

Output:

```
a[1]=20, *(p+1)=20, p[2]=30
a[0]=11, a[1]=22, a[2]=33
```

Mixing pointer and array notation can be confusing → avoid.

Pointer Arithmetic

- Example:

```
int n = 3;
int *p;
p += n;     // adds n*sizeof(int) to p
p -= n;     // subtracts n*sizeof(int) from p
```

- Use only for arrays. Never:

```
char *p;
char a, b;
p = &a;
p += 1;     // may point to b, or not
```

Arrays and Pointers

- Array ≈ pointer to the initial (0th) array element

  `a[i]  ≡  *(a+i)`

- An array is passed to a function as a pointer
  - The array size (# of bytes) is lost!

- Usually bad style to interchange arrays and pointers

Passing arrays:

Really `int *array`

explicitly pass size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

… because **`array`** is really a pointer (and a pointer is architecture-dependent, but likely to be 8 on modern 64-bit machines!)

What does this print? **40**
(provided `sizeof(int)==4`)

Arrays and Pointers

These code sequences have the same effect:

```
int  i;
int  array[5];

for (i = 0; i < 5; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[5];

for (p = array; p < &array[5]; p++)
{
  *p = …;
}
```

Name	Type	Addr	Value
		…	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Point past end of array?

- Array size n; want to access from 0 to n-1, but test for exit by comparing to address one element past the array

```
const int SZ = 10;
int ar[SZ], *p, *q, sum = 0;
p = &ar[0]; q = &ar[SZ];
while (p != q){
     // sum = sum + *p; p = p + 1;
     sum += *p++;
}
```

- Is this legal?
- C defines that one element past end of array must be a valid address, i.e., not cause an error

Valid Pointer Arithmetic

- Add/subtract an integer to/from a pointer
- Difference of 2 pointers (must both point to elements in same array)
- Compare pointers (<, <=, ==, !=, >, >=)
- Compare pointer to NULL
(indicates that the pointer points to nothing)

Everything makes no sense & is illegal:

- adding two pointers
- multiplying pointers
- subtract pointer from integer

Pointers to Pointers

```c
#include <stdio.h>

// changes value of pointer
void next_el(int **h) {
    *h = *h + 1;
}

int main(void) {
    int A[] = { 10, 20, 30 };
    // p points to first element of A
    int *p = A;
    next_el(&p);
    // now p points to 2nd element of A
    printf("*p = %d\n", *p);
}
```

Your Turn …

```
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);
```

Name	Type	Addr	Value
		…	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		…	

Answer	
RED	2
GREEN	3
ORANGE	4
YELLOW	5

C Strings

- C strings are null-terminated character arrays

```
char s[] = "abc";
```

Type	Name	Byte Addr	Value
		...	
		108	
		107	
		106	
		105	
		104	
		103	
		102	
		101	
		100	
		...	

String Example

```c
#include <stdio.h>

int slen(char s[]) {
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}

int main(void) {
    char str[] = "abc";
    printf("str = %s,  length = %d\n", str, slen(str));
}
```

Output: str = abc, length = 3

Concise strlen()

```
int strlen(char *s) {
    char *p = s;
    while (*p++)
   ; /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?

Arguments in `main()`

- To get arguments to the main function, use:

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

  – **`argc`** is the *number* of strings on the command line
  – **`argv`** is a *pointer* to an array containing the arguments as strings

```
#include <stdio.h>

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

Example

```
#include <stdio.h>

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

UNIX:

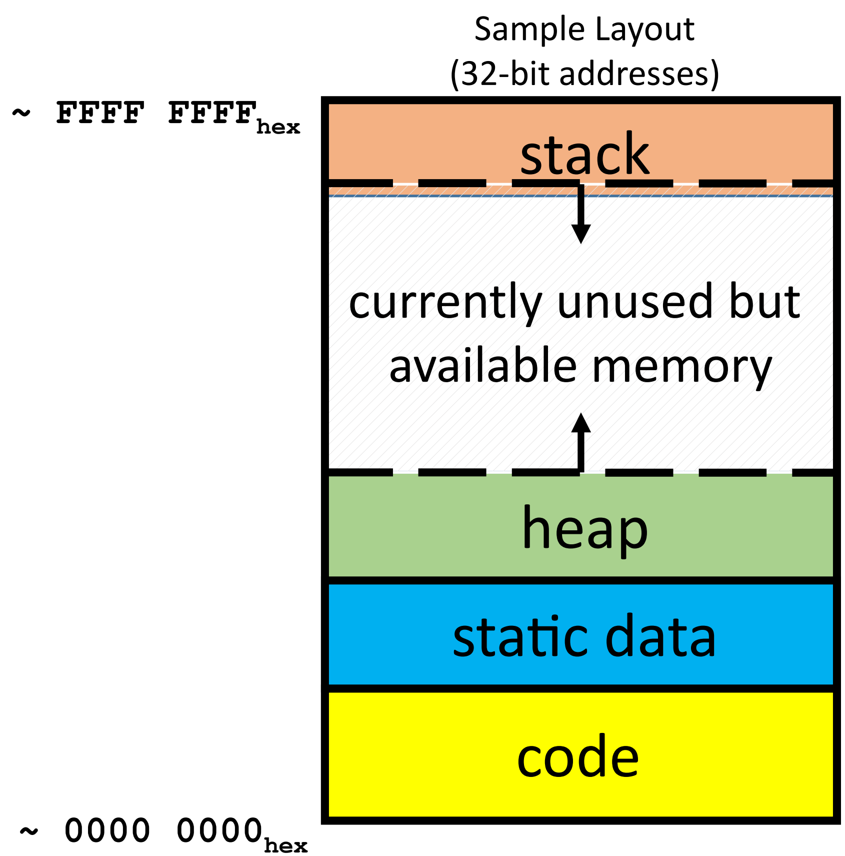
```
$ gcc -o ex Argc.c
$ ./ex -g a "d e f"
arg[0] = ./ex
arg[1] = -g
arg[2] = a
arg[3] = d e f
```

And in Conclusion, …

- Pointers are "C speak" for machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type & size of the object they point to (except void *)
- Like most things, pointers can be used for
  - Pointers are powerful
  - But, without good planning, a major source of errors
  - Plenty of examples in the next lecture!

C Memory Management

- How does the C compiler determine where to put code and data in the machine's memory?
- How can we create dynamically sized objects?
- E.g. array of variable size depending on requirements



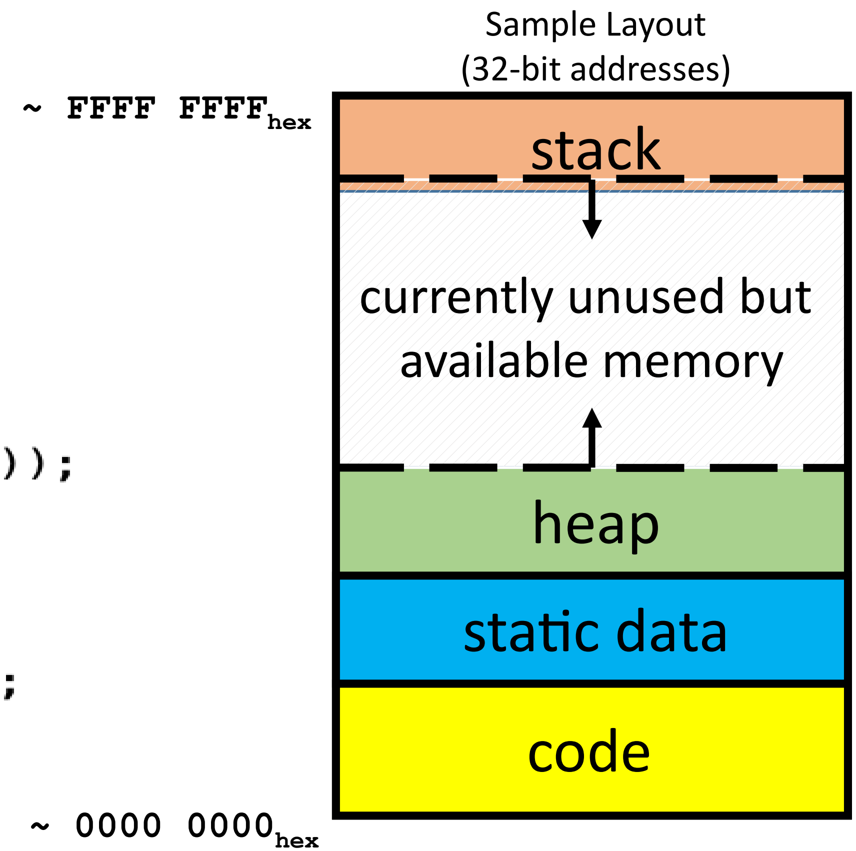
C Memory Management: Code

- Code
  - Loaded when program starts
  - Does not change

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



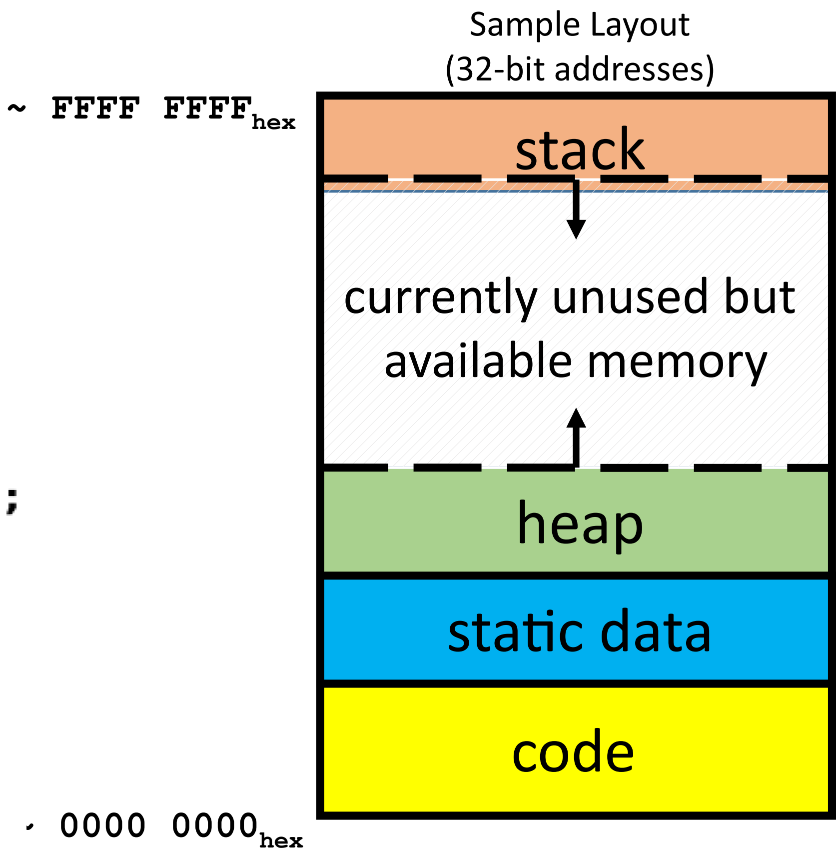
C Memory Management: Static Data

- Static Data
  - Loaded when program starts
  - Can be modified
  - Size is fixed

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```



C Memory Management: Stack

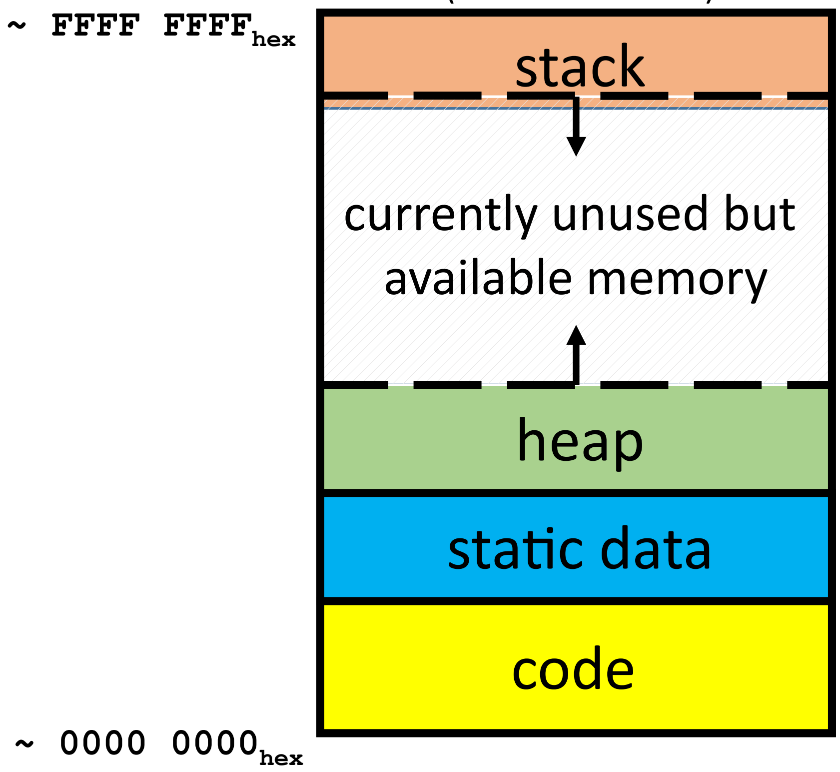
- Stack
  - Local variables & arguments inside functions
  - Allocated when function is called
  - Stack usually grows downward

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```

Sample Layout
(32-bit addresses)



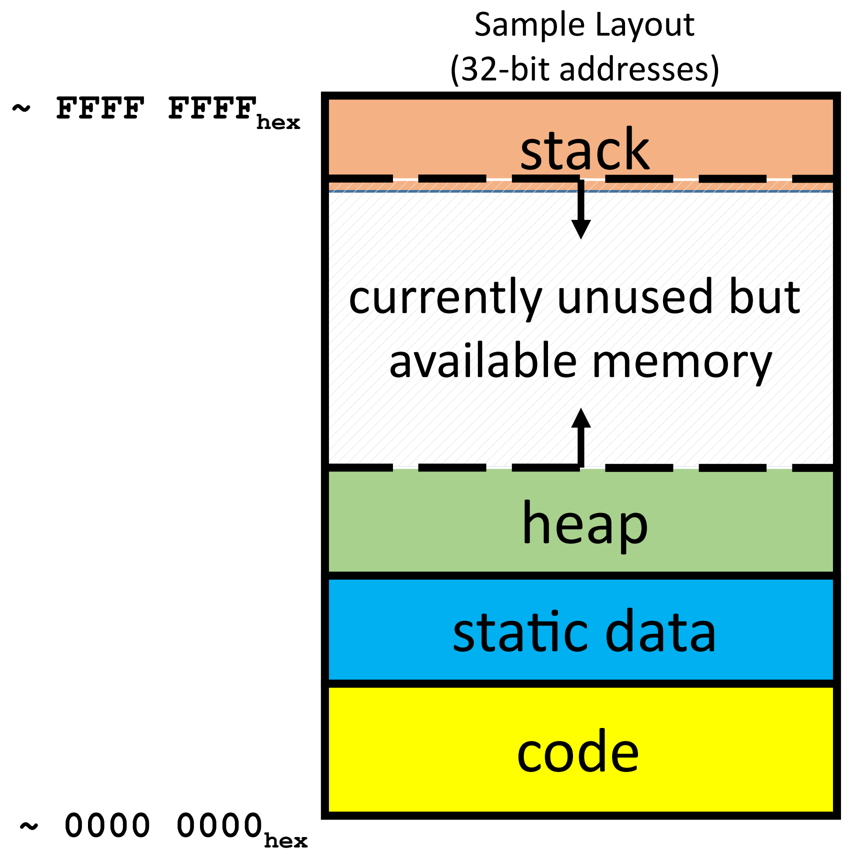
C Memory Management: Stack

- Heap
  - Space for dynamic data
  - Allocated and freed by program as needed

```
int static_data = 55;

void f(char c) {
    int local_data = 4;
    int *heap_data
        = malloc(50*sizeof(int));
}

int main(void) {
    int local_data;
    char more_local_data = 'X';
    f(more_local_data);
}
```

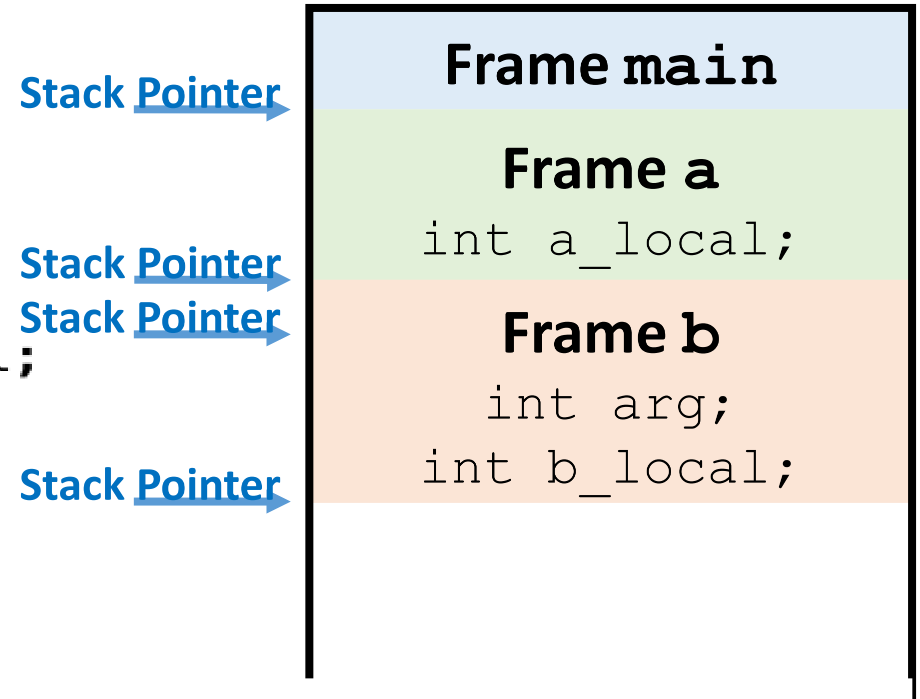


Stack

```c
void a() {
     int a_local = 0;
     b(a_local);
}

int b(int arg) {
     int b_local = 5;
     return 2*arg + b_local;
}

int main(void) {
     a();
     b(7);
}
```



Stack

- Every time a function is called, a new frame is allocated
- When the function returns, the frame is deallocated
- Stack frame contains
  - Function arguments
  - Local variables
  - Return address (who called me?)
- Stack uses continuous blocks of memory
  - Stack pointer indicates current level of stack
- Stack management is transparent to C programmer
  - We'll see details when we program assembly language

Your Turn …

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

int x = 2;

int foo(int n) {
    int y;
    if (n <= 0) {
        printf("End case!\n");
        return 0;
    } else {
        y = n + foo(n - x);
        return y;
    }
}

int main(void) {
    foo(10);
}
```

Right after the **`printf`** executes but before the **`return 0`**, how many copies of **`x`** and **`y`** are allocated in memory?

Answer	#x	#y
RED	1	1
GREEN	1	6
ORANGE	1	5
YELLOW	6	6

What's wrong with this Code?

```c
#include <stdio.h>
#include <math.h>

int *f() {
    int x = 5;
    return &x;
}

int main(void) {
    int *a = f();
    // ... some calculations ...
    double d = cos(1.57);
    // ... now use *a ...
    printf("a = %d\n", *a);
}
```

Output:

```
a = -1085663214
```

- `*a` is a pointer to a local variable
  - allocated on the stack
  - "deallocated" when `f()` returns
  - stack reused by other functions
    - e.g. `cos`
    - which overwrite whatever was there before
  - `*a` points to "garbage"
- Obscure errors
  - depend on what other functions are called after `f()` returns
  - assignments to `*a` corrupt the stack and can result in even more bizarre behavior than this example
  - errors can be difficult to reproduce

Managing the Heap

C functions for heap management:

- **`malloc()`** allocate a block of uninitialized memory
- **`calloc()`** allocate a block of zeroed memory
- **`free()`** free previously allocated block of memory
- **`realloc()`** change size of previously allocated block

Beware: *previously allocated contents might move!*

Malloc()

- **`void *malloc(size_t n):`**
  - Allocate a block of uninitialized memory
  - **`n`** is an integer, indicating size of requested memory block in bytes
  - **`size_t`** is an unsigned integer type big enough to "count" memory bytes
  - Returns **`void*`** pointer to block
  - **`NULL`** return indicates no more memory

- Example:

```
#include <stdlib.h>

int main(void) {
    // array of 50 ints ...
    int *ip = (int*)malloc(50*sizeof(int));

    // typecast is optional
    double *dp = malloc(1000*sizeof(double));
}
```

What’s wrong with this Code?

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

int main(void) {
    const int SZ = 10;
    int *p = malloc(SZ*sizeof(int));
    int *end = &p[SZ];
    while (p < end) *p++ = 0;
    free(p);
}
```

```
$ gcc FreeBug.c; ./a.out
a.out(23562,0x7fff78748000) malloc:
*** error for object 0x7fdcb3403168:
    pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

free()

- **void free(void *p):**
  - Release memory allocated by malloc()
  - **p** must contain address originally returned by **malloc()**

- Example:

```
#include <stdlib.h>

int main(void) {
    // array of 50 ints ...
    int *ip = (int*)malloc(50*sizeof(int));

    // typecast is optional
    double *dp = malloc(1000*sizeof(double));
}
```

Fix

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

int main(void) {
    const int SZ = 10;
    int *array = malloc(SZ*sizeof(int));
    for (int *p = array;  p<&array[SZ]; )
        *p++ = 0;
    free(array);
}
```

- Do not modify return value from **`malloc`**
  - You'll need it to call **`free`**!

Why Call `free()`?

- Recycle no-longer-used memory to avoid running out

- Two common approaches:
  - malloc/free (Explicit memory management)
    - C, C++, …
    - "manually" take care of releasing memory
    - Requires some planning: how do I know that memory is no longer used? What if I forget to release?
    - Drawback: potential bugs
      - memory leaks (forgot to call free)
      - corrupted memory (accessing memory that is now longer "owned")
  - garbage collector (Automatic memory management)
    - Java, Python, …
    - No-longer-used memory is free'd automatically
    - Drawbacks:
      - performance hit
      - unpredictable:
        - what if garbage collector starts when a self-driving car enters a turn at 100mph?

Out of Memory

- Insufficient free memory: **`malloc()`** returns **`NULL`**

```
int main(void) {
    const int G = 1024*1024*1024;
    for (int n=0; ;n++) {
        char *p = malloc(G*sizeof(char));
        if (p == NULL) {
            fprintf(stderr,
                    "failed to allocate > %g TiBytes\n", n/1000.0);
            return 1;  // abort program
        }
        // no free, keep allocating until out of memory
    }
}
```

```
$ gcc OutOfMemory.c; ./a.out
```

```
failed to allocate > 131.064 TiBytes
```

Example: Dynamically Allocated Tree

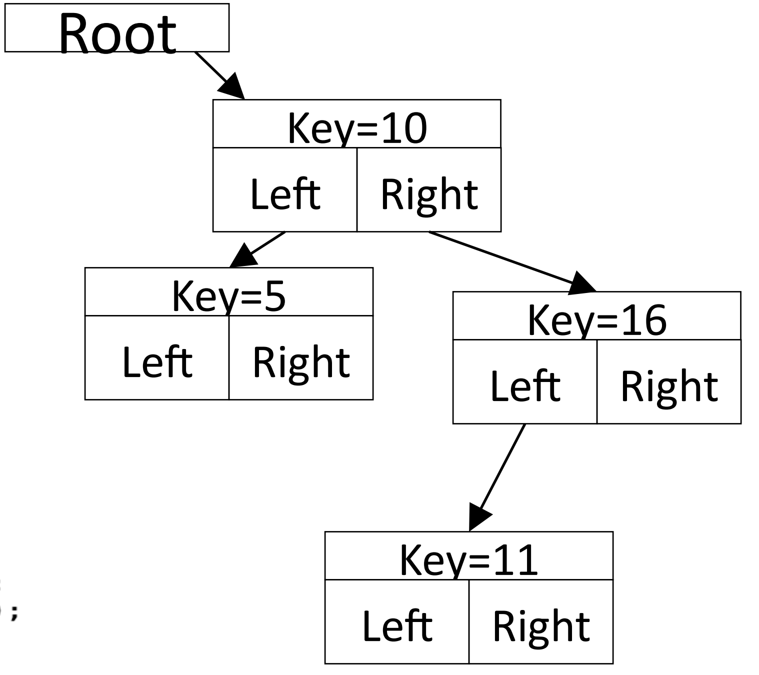
```
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int key;
    struct node *left;
    struct node *right;
} Node;

Node *create_node(int key, Node *left, Node *right) {
    Node *np;
    if ( (np = malloc(sizeof(Node))) == NULL) {
        printf("Out of Memory!\n"); exit(1);
    } else {
        np->key = key;
        np->left = left;   np->right = right;
        return np;
    }
}

void insert(int key, Node **tree) {
   if ( (*tree) == NULL) {
       (*tree) = create_node(key, NULL, NULL);  return; }
   if (key <= (*tree)->key) insert(key, &((*tree)->left));
                       else insert(key, &((*tree)->right));
}

int main(void) {
    Node *root = NULL;
    insert(10, &root);
    insert(16, &root);
    insert(5, &root);
    insert(11 , &root);
}
```



`malloc` and `free` are buddies!

- **`malloc`** and **`free`**
- If you call **`malloc`** somewhere, you'll need to call **`free`** on the result (or accept having memory leak)
- E.g.

```
int *p = malloc(…);
// potentially very
// complicated code
// when done, call:
free(p);
```

- no **`malloc`**, no **`free`**
- If you have not called **`malloc`**, do not call **`free`**!
- E.g.

```
int x;
int *p = &x;
// do whatever with p,
// but do not call free!
int a[5];
// do whatever with a, but
// do not call free(a)!
```

Observations

- Code, Static storage are easy:
  - they never grow or shrink
  - taken care of by OS

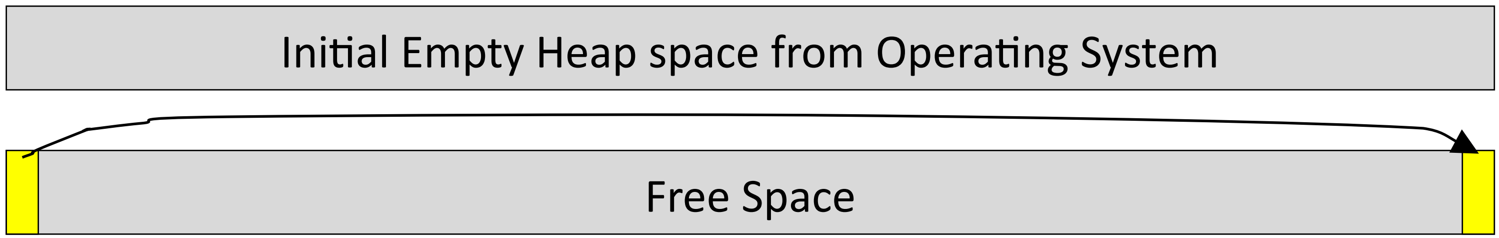
- Stack space is relatively easy:
  - stack frames are created and destroyed in last-in, first-out (LIFO) order
  - transparent to programmer
  - but don't hold onto it (with pointer) after function returns!

- *Managing the heap is the programmer's task:*
  - memory can be allocated / deallocated at any time
  - how tell / ensure that a block of memory is no longer used *anywhere* in the program?
  - requires planning before coding …

How are **`Malloc/Free`** implemented?

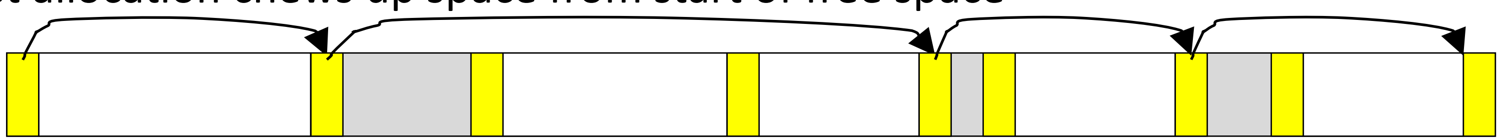
- Underlying operating system allows **`malloc`** library to ask for large blocks of memory to use in heap
  - E.g., using Unix **`sbrk()`** call
- C standard **`malloc`** library creates data structure inside unused portions of the heap to track free space
  - Writing to unallocated (or free'd) memory can corrupt this data structure.
  - Whose fault is it? The library's?

Simple `malloc()` Implementation



Malloc library creates linked list of empty blocks (one block initially)

First allocation chews up space from start of free space



Problems after many **`malloc`**'s and **`free`**'s:

- Memory fragmentation (many small and no big free blocks, merge?)
- Long chain of blocks (slow to traverse)

Better `malloc` Implementations

- Keep separate pools of blocks for different sized objects

- E.g. “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks
  - https://en.wikipedia.org/wiki/Buddy_memory_allocation

Power-of-2 “Buddy Allocator”

Program A requests memory 34 K, order 0
Program B requests memory 66 K, order 1
Program C requests memory 35 K, order 0
Program D requests memory 67 K, order 1

Time

<table>
<tr><th>Step</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th><th>64K</th></tr>
<tr><td>1</td><td colspan="16">2^4</td></tr>
<tr><td>2.1</td><td colspan="8">2^3</td><td colspan="8">2^3</td></tr>
<tr><td>2.2</td><td colspan="4">2^2</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>2.3</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>2.4</td><td>2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>2.5</td><td>A: 2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>3</td><td>A: 2^0</td><td>2^0</td><td colspan="2">B: 2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>4</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>5.1</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="8">2^3</td></tr>
<tr><td>5.2</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="2">D: 2^1</td><td colspan="2">2^1</td><td colspan="8">2^3</td></tr>
<tr><td>6</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="2">D: 2^1</td><td colspan="2">2^1</td><td colspan="8">2^3</td></tr>
<tr><td>7.1</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="8">2^3</td></tr>
<tr><td>7.2</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>8</td><td>2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>9.1</td><td>2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>9.2</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>9.3</td><td colspan="4">2^2</td><td colspan="4">2^2</td><td colspan="8">2^3</td></tr>
<tr><td>9.4</td><td colspan="8">2^3</td><td colspan="8">2^3</td></tr>
<tr><td>9.5</td><td colspan="16">2^4</td></tr>
</table>

https://en.wikipedia.org/wiki/Buddy_memory_allocation

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - De-allocated stack or heap variable
  - Out-of-bounds reference to array
  - Using **`NULL`** or garbage data as a pointer
- Improper use of **`free/realloc`** by messing with the pointer returned by **`malloc/calloc`**
- Memory leaks
  - you allocated something but forgot to free it later

Common Memory Problems

Code

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

int main(void) {
    int* a = malloc(sizeof(int));
    // ...
    free(a);  // a no longer exists!
    int* b = malloc(sizeof(int));
    *b = 128;
    *a = 55;  // ERROR!
    printf("a=%d,  b=%d (==128!)\n",
                *a, *b);
}
```

Output

```
$ gcc test.c
$ ./a.out
a=55, b=55 (==128!)
```

- Assignment to a corrupts b
  - or something else happens that is even more undesirable
- Error may go undetected!

“Defensive” Programming

Code

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

int main(void) {
    int* a = malloc(sizeof(int));
    // ...
    free(a);
    a = NULL; // <-- point to illegal loc
    int* b = malloc(sizeof(int));
    *b = 128;
    *a = 55;
}
```

Output

```
$ ./a.out
Segmentation fault: 11
```

- Problem is evident
- But where is the error?
  - May not be obvious in a BIG program

Debugger (gdb)

```
$ gcc -g DefensiveB.c
$ gdb a.out
GNU gdb (GDB) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
Reading symbols from a.out...Reading symbols from
/a.out.dSYM/Contents/Resources/DWARF/a.out...done.

(gdb) run
Starting program: /defensiveB/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f76 in main () at DefensiveB.c:11
11*a = 55;
(gdb)
```

What’s wrong with this code?

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

int main(void) {
    int *a = malloc(10*sizeof(int));
    int *b = a;  // b is an "alias" for a
    b[0] = 1;
    // work with a (or b) ... then increase size
    a = realloc(a, 1000*sizeof(int));
    // yet another array
    int *c = malloc(10*sizeof(int));
    c[0] = 3;
    b[0] = 2;
    printf("a[0]=%d, b[0]=%d, c[0]=%d\n", a[0], b[0], c[0]);

    printf("a = %p\n", a);
    printf("b = %p\n", b);
    printf("c = %p\n", c);
}
```

Warning:

- This particular result (with **`realloc`**) is a coincidence. If run again, or on a different computer, the result may differ.
- After realloc, **`b`** is no longer valid and points to memory it does not own.
- Using b after calling **`realloc`** is a BUG. Do not program like this!

Output:

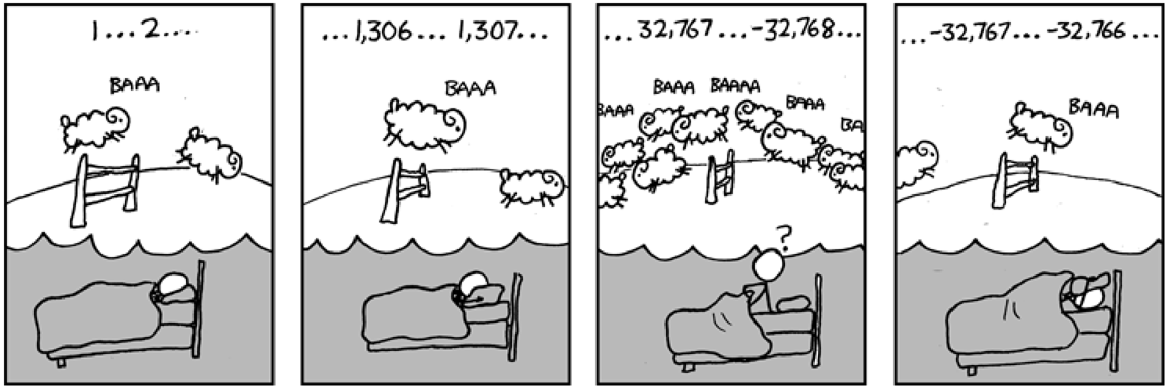
```
a[0]=1, b[0]=2, c[0]=2
a = 0x7fc53b802600
b = 0x7fc53b403140
c = 0x7fc53b403140
```

Output: realloc commented out

```
a[0]=2, b[0]=2, c[0]=3
a = 0x7f9bdbc04be0
b = 0x7f9bdbc04be0
c = 0x7f9bdbc04c10
```

Great Reality #1:
Ints are not Integers, Floats are not Reals

- **Example 1: Is $x^2 \geq 0$?**
  - Float's: Yes!
  - Int's:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??



- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 → 3.14
    - 1e20 + (-1e20 + 3.14) → ??

Computer Arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Great Reality #2: You've Got to Know Assembly

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you are

- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Great Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

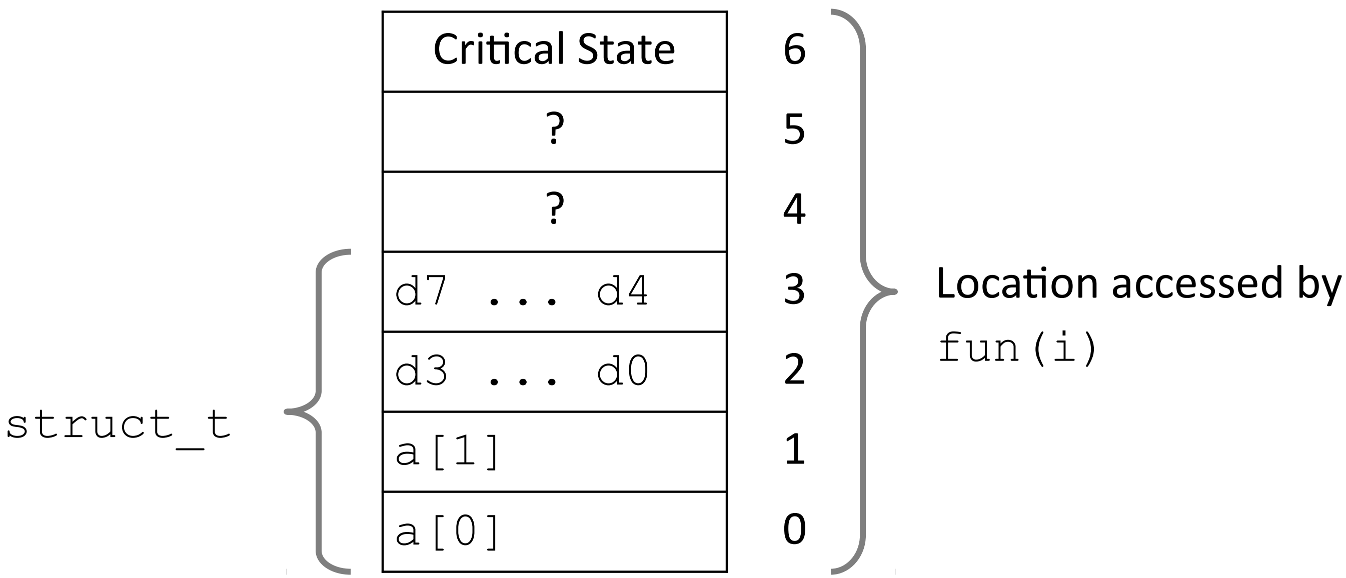
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

Explanation:



Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free

- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated

- How can I deal with this?
  - Program in Java, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Great Reality #4: There's more to performance than asymptotic complexity

- Constant factors matter too!

- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops

- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Memory System Performance Example

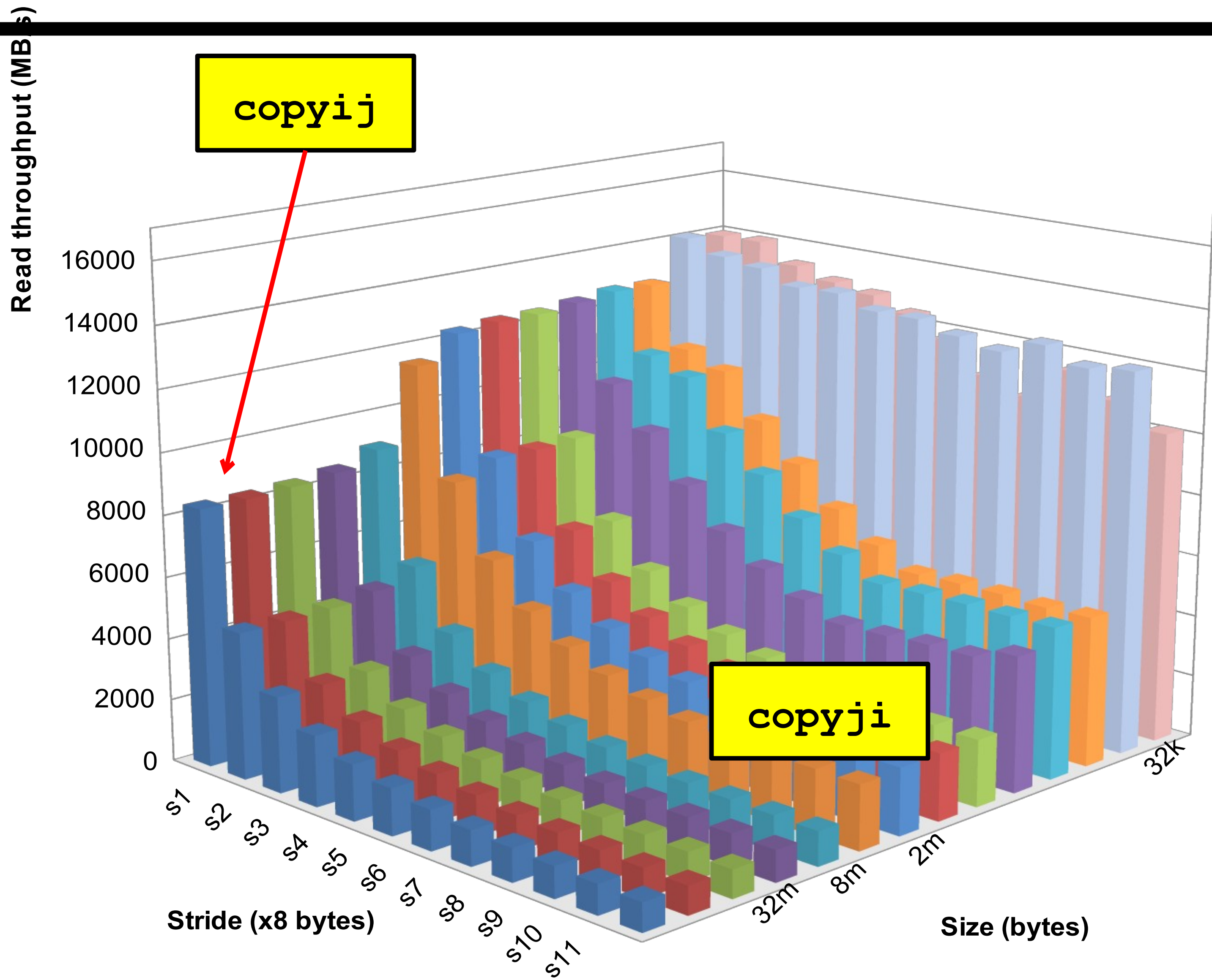
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms 2.0 GHz Intel Core i7 Haswell 81.8ms

- Hierarchical memory organization
  - Performance depends on access patterns
  - Including how step through multi-dimensional array

Why The Performance Differs



Great Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Information

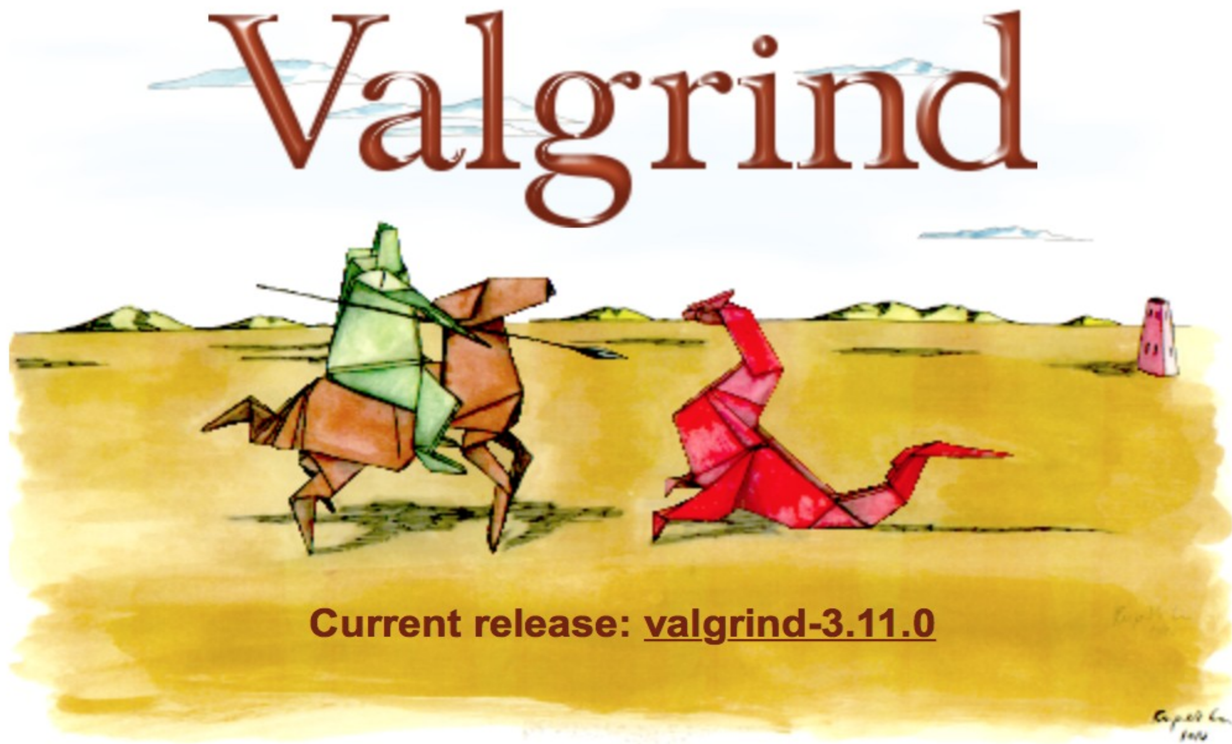
About
News
Tool Suite
Supported Platforms
The Developers

Source Code

Current Releases
Release Archive
Variants / Patches
Code Repository
Valkyrie / GUIs

Documentation

Table of Contents
Quick Start
FAQ
User Manual
Download Manual
Research Papers
Books



Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can

http://valgrind.org

Valgrind Example

```c
#include <stdlib.h>

void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;      // problem 1: heap block overrun
}                   // problem 2: memory leak -- x not freed

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

Valgrind Output (abbreviated)

```
$ gcc -o test -g -O0 test.c
$ valgrind --leak-check=yes ./test
==8724== Memcheck, a memory error detector
==8724==
==8724== Invalid write of size 4
==8724==    at 0x100000F5C: f (test.c:5)
==8724==    by 0x100000F83: main (test.c:9)
==8724==
==8724== HEAP SUMMARY:
==8724==
==8724== 40 bytes in 1 blocks are definitely lost …
==8724==    at 0x100008EBB: malloc …
==8724==    by 0x100000F53: f (test.c:4)
==8724==    by 0x100000F83: main (test.c:9)
==8724==
==8724== LEAK SUMMARY:
==8724==    definitely lost: 40 bytes in 1 blocks
==8724==    indirectly lost: 0 bytes in 0 blocks
```

Classification of Bugs

Bohrbugs

- Executing faulty code produces error
  - syntax errors
  - algorithmic errors (e.g. sort)
  - dereferencing NULL
- “Easily” reproducible
- Diagnose with standard debugging tools, e.g. gdb

Heisenbugs

- Executing faulty code may not result in error
  - uninitialized variable
  - writing past array boundary
- Difficult to reproduce
- Hard to diagnose with standard tools
- Defensive programming & Valgrind attempt to convert Heisenbugs to Bohrbugs
  - crash occurs during testing, not $&^#!

<u>Disclaimer</u>: classification is controversial. Just do not write buggy programs …

<u>Ref</u>: J. Gray, Why do computers stop and what can be done about them?, Tandem TR 85.7

And in Conclusion …

- C has three main memory segments to allocate data:
  - Static Data: Variables outside functions (globals)
  - Stack: Variables local to function
  - Heap: Memory allocated explicitly with malloc/free

- Heap data is an exceptionally fertile ground for bugs
  - memory leaks & corruption
  - send me your best examples (EPA credit? - you paid for them!)

- Strategies:
  - Planning:
    - Who “owns” malloc’d data?
    - Often more than one ”owner” (pointer) to same data
    - Who can safely call free?
  - Defensive programming, e.g.
    - Assign NULL to free’d pointer
    - Use const’s for array size
  - Tools, e.g.
    - gdb, Valgrind

Additional Examples of

C Memory Errors

(to peruse at your leisure)

Using Memory You Don't Own

What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      ipr = (int *) malloc(4 * sizeof(int));
          i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }

   void WriteMem() {
      ipw = (int *) malloc(5 * sizeof(int));
      *(ipw - 1000) = 0; *(ipw + 1000) = 0;
      free(ipw);
   }
```

Using Memory You Don’t Own

Using pointers beyond the range that had been malloc’d

- May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      ipr = (int *) malloc(4 * sizeof(int));
        i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }

   void WriteMem() {
      ipw = (int *) malloc(5 * sizeof(int));
      *(ipw - 1000) = 0; *(ipw + 1000) = 0;
      free(ipw);
   }
```

Faulty Heap Management

What is wrong with this code?

```
int *pi;
void foo() {
    pi = malloc(8*sizeof(int));
    …
    free(pi);
}

void main() {
    pi = malloc(4*sizeof(int));
    foo();
    …
}
```

Faulty Heap Management

Memory leak: more mallocs than frees

```
int *pi;
void foo() {
    pi = malloc(8*sizeof(int));
    /* Allocate memory for pi */
    /* Oops, leaked the old memory pointed to by pi */
    …
    free(pi);
}

void main() {
    pi = malloc(4*sizeof(int));
    foo(); /* Memory leak: foo leaks it */
  …
}
```

Faulty Heap Management

What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
   … … …
    plk++;
}
```

Faulty Heap Management

Potential memory leak – handle (block pointer) has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
   … … …
   plk++; /* Potential leak: pointer variable incremented
past beginning of block! */
}
```

Faulty Heap Management

What is wrong with this code?

```
void FreeMemX() {
    int fnh = 0;
    free(&fnh);
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    free(fum);
    free(fum);
}
```

Faulty Heap Management

Can’t free non-heap memory; Can’t free memory that hasn’t been allocated

```
void FreeMemX() {
    int fnh = 0;
    free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
/* fum+1 is not a proper handle; points to middle
of a block */
    free(fum);
    free(fum);
/* Oops! Attempt to free already freed memory */
}
```

Using Memory You Haven’t Allocated

What is wrong with this code?

```
void StringManipulate() {
    const char *name = “Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
    printf("%s\n", str);
}
```

Using Memory You Haven’t Allocated

Reference beyond array bounds

```
void StringManipulate() {
    const char *name = “Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
   /* Write Beyond Array Bounds */
    printf("%s\n", str);
   /* Read Beyond Array Bounds */
}
```

Using Memory You Don't Own

What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[128];
    int i=0, j=0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
    }
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
    }
    result[++i] = '\0';
    return result;
}
```

Using Memory You Don't Own

Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[128];
    int i=0, j=0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
    }
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
    }
    result[++i] = '\0';
    return result;
}
```

result is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Managing the Heap

- **`realloc(p,size)`:**
  - Resize a previously allocated block at p to a new size
  - If p is NULL, then realloc behaves like malloc
  - If size is 0, then realloc behaves like free, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

- E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don't Own

What's wrong with this code?

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Using Memory You Don't Own

Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}
```

Remember: **`realloc`** may move entire block

```
int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

What if array is moved to new location?

Where is my stuff?

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

int global = 1;

int twice(int i) { return 2*i; }

int main(void) {
    int stack = 2;
    int *heap = malloc(sizeof(int));
    *heap = 3;
    int (*code)(int);  // pointer to "int func(int) {}"
    code = twice;
    printf("global  = %d, &global= %p\n", global, &global);
    printf("stack   = %d, &stack = %p\n", stack, &stack);
    printf("*heap   = %d, heap   = %p\n", *heap, heap);
    printf("code(4) = %d, code   = %p\n", twice(4), twice);
}
```

```
global  = 1, &global= 0x10923f020
stack   = 2, &stack = 0x7fff569c1bec
*heap   = 3, heap   = 0x7fa7b8400020
code(4) = 8, code   = 0x10923ee40
```

Aside: Memory “Leaks” in Java

- Accidentally keeping a reference to an unused object prevents the garbage collector to reclaim it

- May eventually lead to “Out of Memory” error

- But many errors are eliminated:
  - Calling **`free()`** with invalid argument
  - Accessing free’d memory
  - Accessing outside array bounds
  - Accessing unallocated memory (forgot calling new)
    (get null-pointer exception – error, but at least no “silent” data corruption)
  - All this can happen in a C program!

Using the Heap ... Example

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

typedef struct {
    int year_planted;
    // ... kind, cost, ...
} Tree;

Tree* plant_tree(int year) {
    Tree* tn = malloc(sizeof(Tree));
    tn->year_planted = year;
    return tn;
}

int main(void) {
    const int ORCHARD = 100;
    // lets grow some apple trees ...
    Tree* apples[ORCHARD];
    for (int i=0;  i<ORCHARD;  i++)
        apples[i]= plant_tree(2014);

    // apples don't sell ... let's try pears
    Tree* pears[ORCHARD];
    for (int i=0;  i<ORCHARD;  i++)
        pears[i]= plant_tree(2016);
}
```

- New problem:
  - how do we get rid of the apple trees?
- Need a way to free no longer used memory
  - or may eventually run out