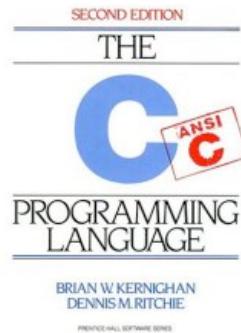


{C'}

CS240



The Craft of C Programming

C

2

```
int a[]={0,1,2,3,4};
```

```
r((int)a, 0, 5);
```

```
void r(int a, int i, int E) {
    printf("%d\n",*((int*)a+i));
    if (++i < E) r(a, i, E);
}
```

Learning objectives

3

CS240 is your introduction to the craft of programming

You will learn ...

- ▶ to solve problems computationally
- ▶ to design, implement, test, debug and evaluate algorithms
- ▶ to use state of the art technologies (C, Unix, Emacs, gdb, make, shell...)

Our programming language of choice is C, because

- ▶ it is widely used in the industry
 - complex systems (from web browsers to operating systems) written in C/C++/Objective-C*
- ▶ it gives you fine-grain control over resources
 - whereas Java hides everything from you*
- ▶ it allows to explore the interaction between software and hardware
 - C exposes architectural features*

Workload

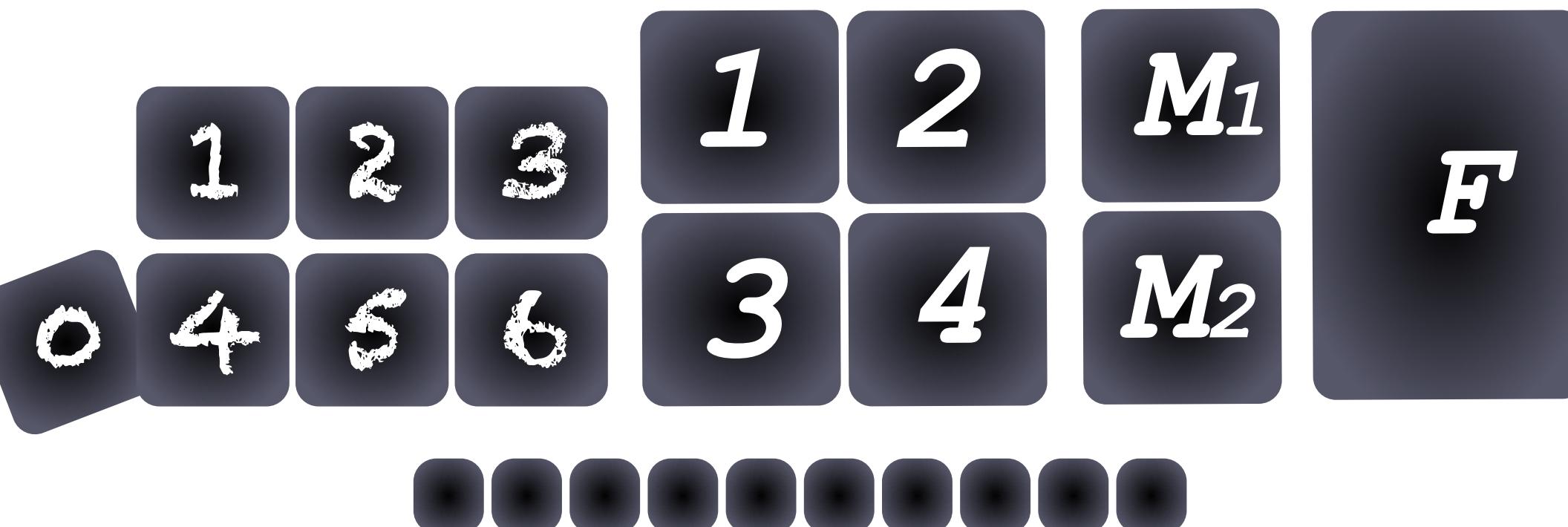
4

6 labs, 4 projects, 2 midterms, 1 final, 10 reading quizzes

<http://www.cs.purdue.edu/homes/jv/courses/240f11>

Grade mix:

- ▶ labs 20%, projs 30%, midterms 20%, final 20%, quizzes 10%
- ▶ pick 5 best labs and 8 best quizzes



Late policy

5

None.

Rationale: In a large class it is not possible to accommodate requests for extensions

All assignments are handed on time.

- ▶ You are allowed to miss one lab for any reason
- ▶ You are allowed to miss two quizzes for any reason

Rationale: you can discard your worse grade or miss a class without need for justification

Academic integrity

6

Any case of cheating will handled by the Dean of students

You are encouraged to discuss problems and approaches but:

Sharing solution is not allowed.

Buying solutions is not allowed.

Copying code from the internet is not allowed.

Copying code from other students is not allowed.

Copying partial code from other students is not allowed.

<http://homes.cerias.purdue.edu/~spaf/cpolicy.html>

**Each year we catch students... they end up with an
F... and a record... is it worth it?**

Attendance

7

Class attendance and Lab attendance is mandatory

Rationale:

- ▶ *Slides are not complete nor always accurate
At the exam, trust your notes and the book*
- ▶ *Announcements are made in class.
The website often incorrect/out of date*
- ▶ *It's a good way to start your morning*

Labs

8

Lab sessions will involve short problems that are mostly done in the lab session

20% of grade

► *Rationale: ensures you can get help from TAs while solving the problem*

Projects

9

Projects are 2 week long programming exercises that will culminate in a small group assignment

Projects are done outside of the lab sessions and should take around 12 hours each

30% of grade

Midterms

10

Midterms are in class, closed-book, exams that are representative of the difficulty of the final

20% of grade

Final

11

A closed-book exam

20% of grade

Reading quizzes

12

Short in-class tests of your understanding of the reading assignment for the day using clickers

Quizzes cover material that has **not** been presented in class

It is your responsibility to read the book and ask questions ahead of class in case something is unclear

Quizzes will take place in the first five minutes of class and are usually unannounced

Don't come late

10% of grade

Questions

13

Use the following algorithm

- ▶ Ask on Piazza
- ▶ Ask TA at lab
- ▶ Ask Prof during class

Rationale: This focuses the interaction and ensure best use of your time

Regrading questions

- ▶ Regrading will only be done **the week following** release of the grade
- ▶ Send mail to the course staff mail alias (`admin240@cs`) with a precise description of your grading request, and get a ticket
- ▶ Midterm/Final issues are dealt by the Instructor, project/labs are dealt by your TA and can be escalated to the Faculty member

It is your responsibility to check your grades!

Questions

14

How to ask a question on Piazza:

- Read the book, slides, notes
- Describe the problem clearly, using the right terms
- Add code in attached files
- Add output from compiler
- Add any other relevant information

Try to be polite and we will do the same!

Avoid #IDW questions...

Environment

15

Linux

gcc

gdb

emacs/vi/jedit

Firefox

VMware image

Warning

Warning

17

**reading the book is no
substitute for attending
lectures**

Warning

18

attending lectures is no substitute for reading the book

Warning

19

slides may be partial

slides may have mistakes

think critically

Warning

20

information given at the
lectures may not be in
slides or in piazza...

Programming

21

Is programming a craft? an art? a science?

There are many ways to express some task

How do we know which is best?

We need to understand the tradeoffs...

- ▶ e.g. iteration vs. recursion

```
int a[]={0,1,2,3,4};  
for (int i=0; i<5; i++)  
    printf("%d\n",a[i]);  
  
void r(int* a, int i, int E) {  
    printf("%d\n",*(a+i));  
    if (++i < E) r(a, i, E);  
}
```

Programming

22

```
int a[]={0,1,2,3,4};    create a 5 elem array
for
(int i=0;                loop from
i<5;                  zero to
i++)                  five
printf(                step by one
    "%d\n",            print
    a[i];              i-th elem followed by
                      newline
```

... and in Java

23

```
int[ ] a = new int[ ]{0,1,2,3,4};  
for (int i=0; i<5; i++)  
    System.out.println(a[i]);
```

```
int a[ ]={0,1,2,3,4};  
for (int i=0; i<5; i++)  
    printf("%d\n",a[i]);
```

Getting started

24

```
#include <stdio.h>
int main() {
    printf("Hello World! \n");
}
```

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

Compilation

25

```
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
    printf(HELLO);
}
```



gcc -std=c99 -c hello.c

gcc -o a.out hello.o

./a.out

Start with a human readable file containing your source program and possibly some references to libraries

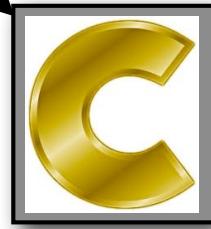
Call the C compiler to obtain an executable file, which is a sequence of numbers that can run on the target hardware

```
33 % hexdump a.out
00000000 cf fa ed fe 07 00 00 01 03 00 00 80 02 00 00 00
00000010 0d 00 00 00 20 06 00 00 85 00 20 00 00 00 00 00
00000020 19 00 00 00 48 00 00 00 5f 5f 50 41 47 45 5a 45
00000030 52 4f 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00 00 00 00 00 19 00 00 00 28 02 00 00
00000070 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00
00000080 00 00 00 00 01 00 00 00 10 00 00 00 00 00 00 00
```

Compilation

26

```
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
    printf(HELLO);
}
```



```
gcc -std=c99 -E hello.c
```

One of the first step is to expand macro definitions and include external declarations.

The compiler works in phases that transform the program into the executable one step at a time.

```
# 1 "h.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "h.c"
# 37 "/usr/include/i386/_types.h" 3 4
typedef signed char __int8_t;
typedef unsigned char __uint8_t;

...
# 238 "/usr/include/stdio.h" 3 4
int fprintf(FILE * , const char * , ...) __attribute__;
void perror(const char *);
int printf(const char * , ...) __attribute__((__format__))
int puts(const char *);

...
# 500 "/usr/include/stdio.h" 2 3 4
# 2 "h.c" 2

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

Compilation

27

```
#include <stdio.h>
#define HELLO "Hello World!\n"
int main() {
    printf(HELLO);
}

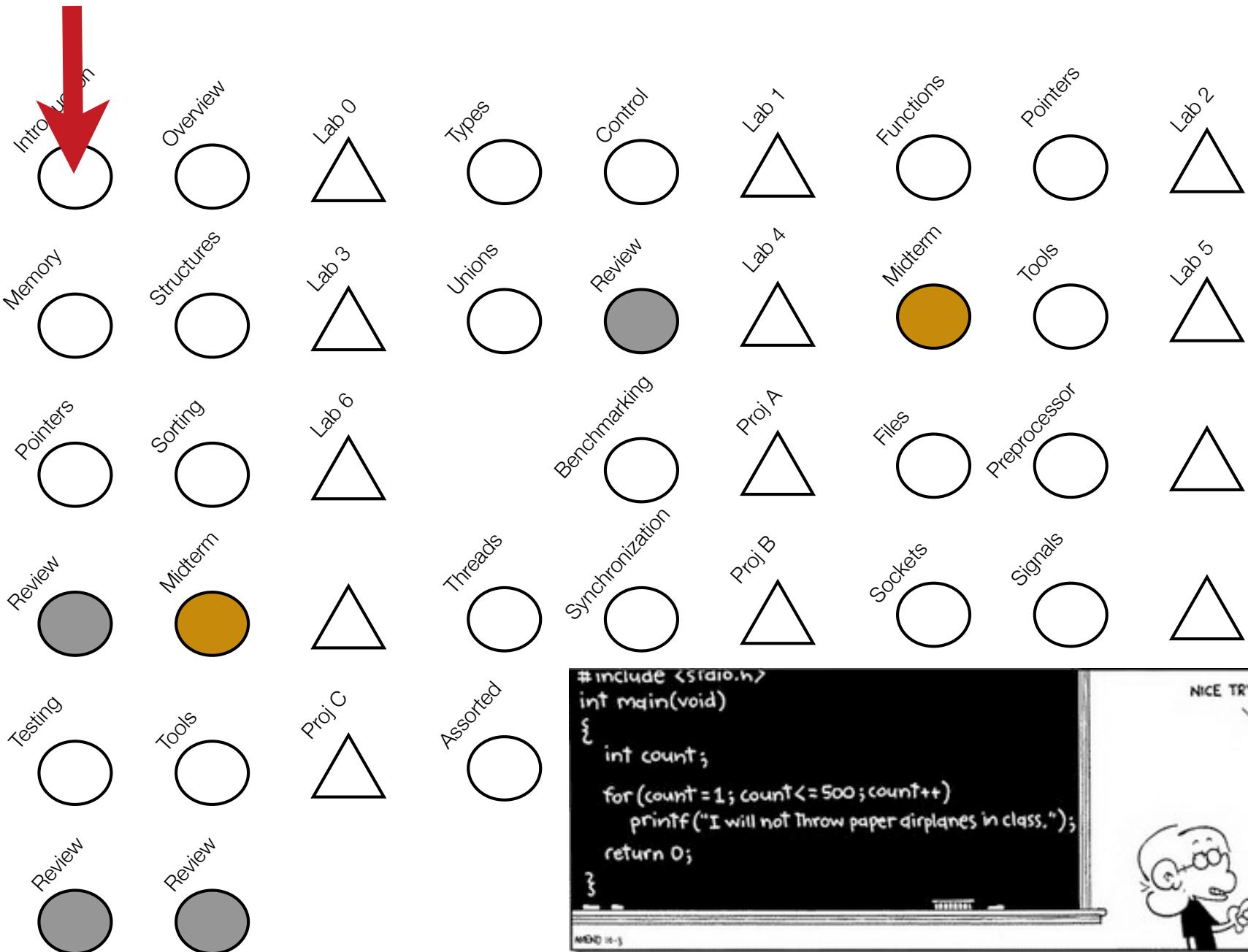
gcc -c hello.c
gcc -o a.out hello.o
./a.out
```



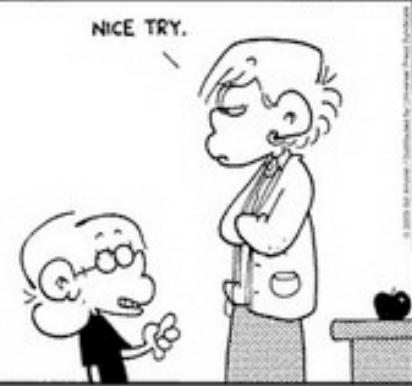
The compiler works in phases that transform the program into the executable one step at a time.

The last step links libraries, e.g. I/O, to create a stand alone executable binary

```
42 % nm a.out
0000000100001048 S _NXArgc
0000000100001050 S _NXArgv
0000000100001060 S __progname
0000000100000000 A __mh_execute_header
0000000100001058 S _environ
U _exit
0000000100000f00 T __main
U __puts
U dyld_stub_binder
0000000100000ec0 T start
```



```
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```



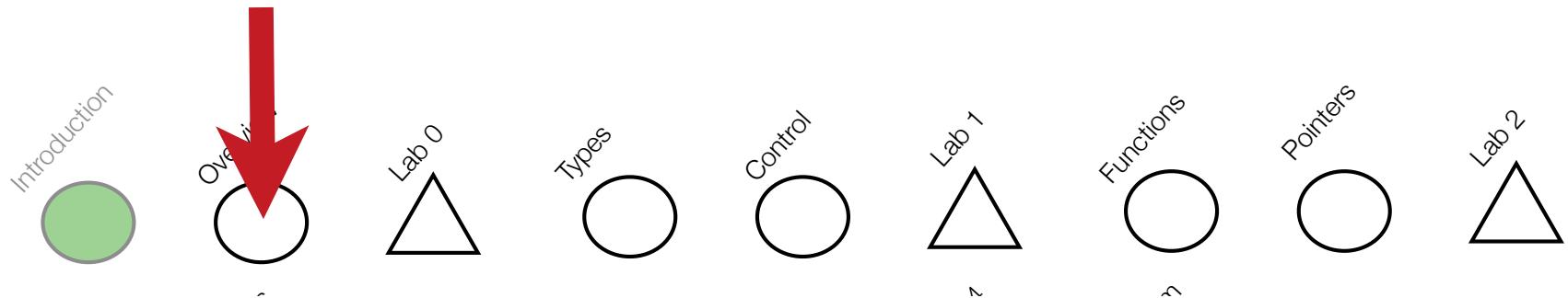
<http://tinyurl.com/4yvv79a>

{C}

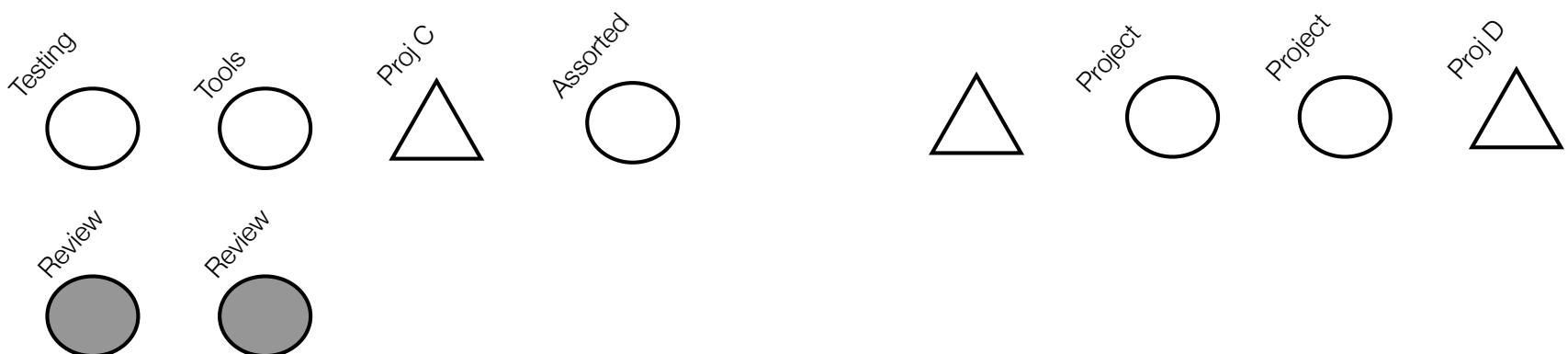
Lecture 2

A gentle introduction to C programming





New students should
contact their TA to get
added to Piazza



<http://tinyurl.com/4yvv79a>

strip.c

31

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

int main() {
    int c = 0, in = 0;
    char buf[2048]; char *p = buf;

    while((c = getchar()) != EOF) {
        if(c=='<' || c=='&') in=1;
        if(in) *p++=c;
        if(c==',' || c==';') {
            in = 0;
            *p++ = '\0';
            if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))
                printf(" ");
            p = buf;
        } else if(!in) printf("%c", c);
    }
}
```

Includes

32

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

- ▶ Tell the compiler about external functions that may be used by the program
- ▶ Pre-processor directives, expended early in the compilation
- ▶ stdio defines functions getchar/printf
- ▶ string defines strstr

Main

33

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

- ▶ C programs must have a `main()` function
- ▶ `main()` called first when the program is started by the OS
- ▶ `main()` returns an integer
- ▶ without a return statement, undefined value is returned
- ▶ The correct signature for `main()` is:

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

Getchar/printf

34

```
int c = 0;
```

```
while((c = getchar()) != EOF)  
    printf("%c", c);
```

- ▶ `getchar()` returns 1 character from “standard input” converted to an int
- ▶ If the stream is at end-of-file or a read error occurs, EOF is returned
- ▶ `printf()` outputs a string to the standard output
- ▶ `printf()` takes a format string and a variable numbers of arguments that are converted to characters according to the requested format

Looping

35

```
int c = getchar();
while(c != EOF) {
    printf("%c", c);
    c = getchar();
}
```

- ▶ another way to express the same behavior
- ▶ in C assignments are expressions, the same program without nesting

... in Java

36

```
public static void main(String[] args) throws IOException {  
    InputStreamReader ins = new InputStreamReader(System.in);  
    BufferedReader in = new BufferedReader(ins);  
    int c;  
    while ((c = in.read()) != -1)  
        System.out.print(c);  
}
```

- ▶ Similar but more wordy...

Arrays & pointers

37

```
int c = 0, in = 0;
char buf[2048]; char *p = buf;

while((c = getchar()) != EOF) {
    if(c=='<' || c=='&') in = 1;
    if(in) *p++=c;
    if(c==',' || c==';') {
        in = 0; *p++ = '\0';
    }
}
```

- ▶ `buf` is an array of 2048 characters;
- ▶ `p` is pointer in the buffer
- ▶ boolean value false is 0, any non-0 is true

Arrays

38

```
char buf[2048]; int pos=0;  
while((c = getchar()) != EOF) {  
    ...  
    if(in) buf[pos++] = c;  
    if(c=='>' || c==';') {  
        buf[pos++] = '\0';  
        pos=0;  
    }  
}
```

- ▶ the same program without pointers
- ▶ an alternative to pointers is to use an index in the array of chars
- ▶ strings must be \0 terminated (or risk a buffer overflow...)

Strstr

39

```
char buf[2048]; char *p = buf;  
...  
if(state) *p++=c;  
...  
*p++ = '\0';  
if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))  
    printf(" ");  
p = buf;
```

- ▶ `strstr(s1,s2)` locates the first occurrence of the null-terminated string `s2` in the null-terminated string `s1`.
- ▶ if `s2` occurs nowhere in `s1`, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of `s2` is returned
- ▶ `NULL` is false, `||` is logical or

strip.c

40

```
int main( ) {
    int c = 0, in = 0;
    char buf[2048]; char *p = buf;
    while((c = getchar()) != EOF) {
        if(c=='<' || c=='&') in=1;
        if(in) *p++=c;
        if(c==',' || c==';') {
            in = 0;
            *p++ = '\0';
            if(strstr(buf,"nbsp")||strstr(buf,"NBSP"))
                printf(" ");
            p = buf;
        } else if(!in) printf("%c", c);
    }
}
```

Symbolic constants

41

```
#define NUM 42
```

- ▶ The `#define` directive is followed by the name of the macro and the token sequence it abbreviates
- ▶ By convention, macro names are written in uppercase.
- ▶ There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens.
- ▶ If the expansion of a macro contains its name, it is not expanded again

x = 1;

y = x++;

z = ++x;

a[i++]

a[+i]

- ▶ The prefix operators ++ and -- increment/decrement their operand
- ▶ The operand of prefix ++ is incremented. The expression value is the new value of the operand
- ▶ When postfix ++ is applied to a modifiable lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented by 1

(n++) is (tmp=n, n=n+1, tmp)

(++n) is (n=n+1, n)

//one fewer register...faster...

Strings

43

"Hello"



- ▶ A string literal is a sequence of characters delimited by double quotes
- ▶ It has type **array of char** and is initialized with the given characters
- ▶ The compiler places a null byte (\0) at the end of each string literal
- ▶ A double-quote ("") in a string literal must be preceded by a backslash (\)
- ▶ Creating an array of character:

```
char c[6] = "Hello";
```

Basic Data Types

44

char int float

- ▶ Characters (**char**) are large enough to store any element of the character set. If a genuine character from the character set is stored in a **char** variable, its value is equivalent to the integer code for that character
- ▶ Up to five sizes of integral types (signed and unsigned) are available: **char, short, int, long, and long long**. Up to three sizes of floating point types are available

Arrays

45

```
char buf[2048];  
buff[0] = 'a'; buff[1] = buff[0];
```

- ▶ Array variables are declared with the T[] syntax
- ▶ Items that are not explicitly initialized will have an indeterminate value unless the array is of static storage duration
- ▶ Initialize x as a one-dimensional array with 3 members, because no size was specified and there are 3 initializers:

```
int x[] ={1,3,5};
```

- ▶ Bracketed initialization: 1, 3, and 5 initialize the first row of the array y[0], namely y[0][0], ... The initializer ends early:

```
float y[3][3] = {  
    { 1, 3, 5 },  
    { 2, 4, 6 },  
    { 3, 5, 7 } };
```

An array of strings

46

```
char my[ 2 ][ 3 ] = { "me", "my" } ;  
my[ 1 ][ 0 ] == 'm'
```

- ▶ For fixed sized strings, arrays of arrays of characters work well
- ▶ For variable sized strings, char pointers must be used

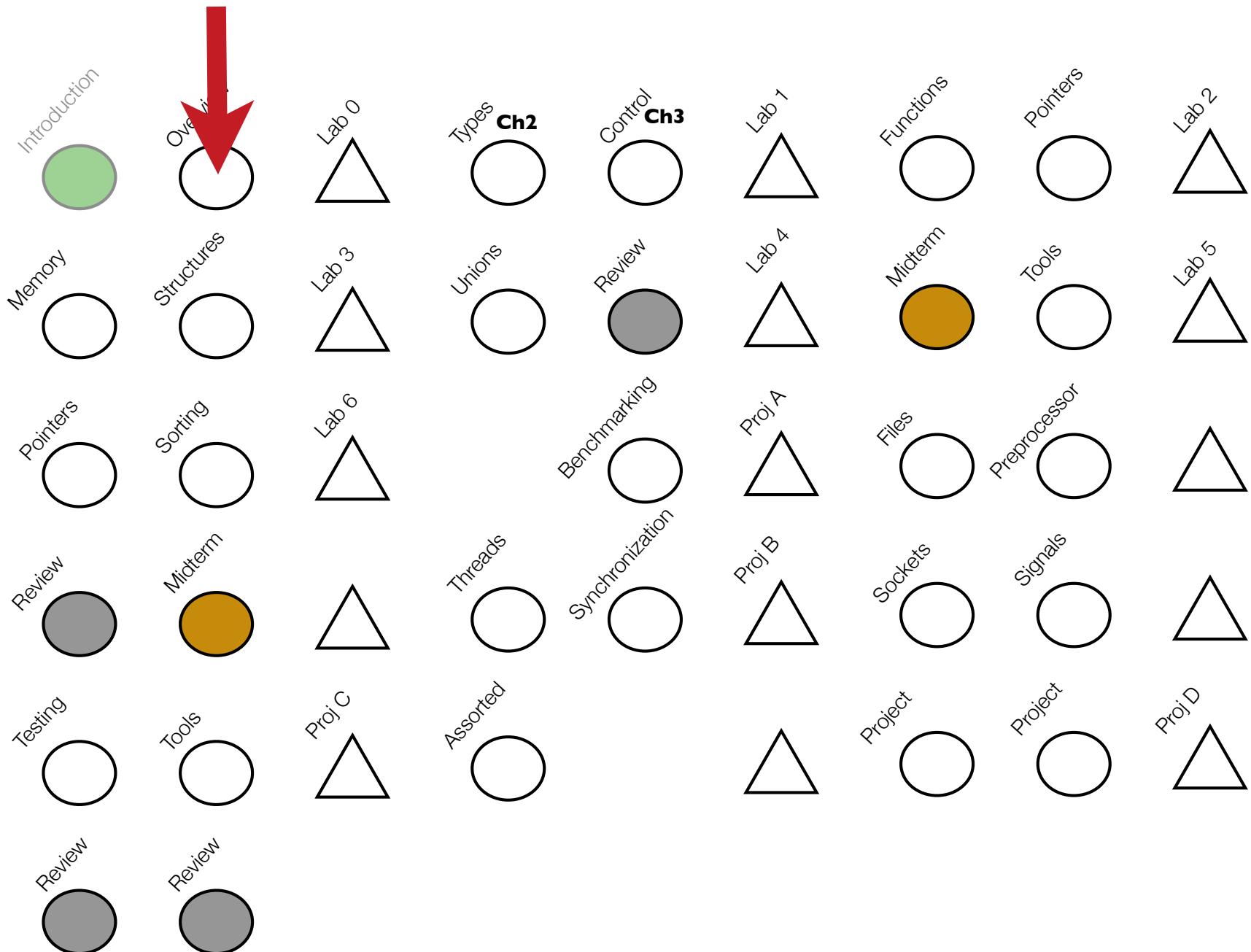
```
char* you[ 2 ] = { "you", "yours" } ;
```

Of chars and ints & conversions

47

```
int c;  
char buf[1];  
c = getchar();  
buf[0] = c;
```

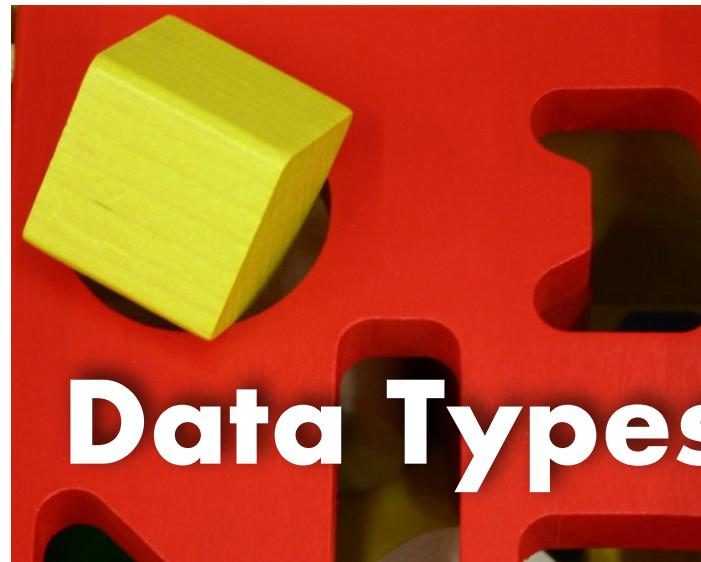
- ▶ Conversions from an integer value to a character do not lose information if the integer is in the valid range for characters
- ▶ The value EOF is not a valid character value



<http://tinyurl.com/4yvv79a>

{LoC}

Lecture 3



Quiz 1

50

Warm up your clickers...

Quiz #1 - 1

51

Type **long int** and **double** hold larger numbers than **int** and **float** respectively. Which one can hold the largest value?

- (a) **long int**
- (b) **double**
- (c) **same**

Answer

52

long int is typically 32-bits signed ==> +2147483647

double is typically 64-bits floating point signed ==> 10^{308}

Quiz #1 - 2

53

Is the following true?

$$145 == 0145$$

(a) true

(b) false

Answer

54

Octal 145 is decimal 101

Quiz #1 - 3

55

What is printed?

```
if (42) printf("thanks"); else printf("fish");
```

(a) thanks

(b) fish

Answer

56

In C all non-zero values are treated as "true"

Quiz #1 - 4

57

Precedence is used by the compiler to parse complex expressions, consider

c=getchar() != '\n'

Is it parsed as:

(a) **(c=getchar()) != '\n'**

(b) **c=(getchar()) != '\n'**

Answer

58

```
c=getchar() != '\n'
```

Is it parsed as: `c=(getchar() != '\n')`

Which means that the variable has the value zero most of the time and one when an end of line is encountered

A correct version is

```
(c=getchar()) != '\n'
```

Quiz #1 - 5

59

In C operator + has higher precedence than operator == and both are left to right associative. Should:

$$a + b + c == d$$

be read as:

- (a) $((a + b) + c) == d$
- (b) or as $(a + (b + (c == d)))$

Answer

60

$$(a) ((a + b) + c) == d$$

Higher precedence means $a+b==c \Rightarrow (a+b)==c$

Left to right associativity means $a+b+c \Rightarrow ((a+b)+c)$

Right to left associativity means $a=b=c \Rightarrow (a=(b=c))$

Quiz #1 - 6

61

Is the variable referring to a zero-terminated character string?

```
char c[2] = "hi";
```

- (a) yes
- (b) no
- (c) depends on the architecture

Answer

62

The compiler will not add a zero at the end of the string as it would require a three character array to do so.

```
char c[3] = "hi"
```

Lab recap

63

```
int main() {  
    char b[140]; int c,i=0;  
    while( (c = getchar()) != EOF ) b[i++] = c;  
    tweetIt(b,i);  
}
```

Data Types

64

Data types are sets of values along with operations that manipulate them

Integers in C are made up of the set of values ..., -1, 0, 1, 2, ... along with operations such as addition, subtraction, multiplication, division...

These values must be mapped to the data types provided by the hardware and operations compiled to sequences of hardware instructions

bits – semantics

65

Imagine a data type called **pit** (for pair o' bits) with values:
0, 1, 2, 3

pits have one operation the postfix increment: **++**

The meaning of **++** is the function:

0 ++ = 1, 1 ++ = 2, 2 ++ = 3, 3 ++ = 0

Observations:

- ▶ The data type is finite
- ▶ The increment operation wraps around

bits - implementation

66

A variable of type **pit** is represented by two bits in memory

00 = 0

01 = 1

10 = 2

11 = 3

Pseudo code implementation of **++**

```
pit function<++>(pit p) {  
    int x = p;  
    x += 1;  
    if (x==4) x = 0;  
    return (pit) x;  
}
```

signed pits

67

To represent **signed pits** one bit is need for the sign:

- ▶ decrease the range of values
- ▶ increase the number of bits used to represent the data type

Assume the size of the data type must not change.

signed pits can take the following values: **-1, 0, 1**

The implementation is as follows

$$00 = 0 \quad 01 = 1 \quad 10 = 0 \quad 11 = -1$$

signed pits

68

The implementation is as follows

00 = 0 01 = 1 10 = 0 11 = -1

The increment function is somewhat more complex

```
signed pit function<++>(signed pit p) {
    int s = (p>>1) ? -1 : 0;
    int x = p & 1;
    x += 1;
    if (x==1) x = -1;
    return (x<0) ?(2|(unsigned int)x) :
        x;
```

Data Types

69

Designing a computer language requires choosing which data types to build in, and which ones must be defined by users

The tradeoff is one of expressiveness vs. efficiency

Expressiveness refers to the ability to clearly express solutions to computational problem

Efficiency refers to the ability to map the data type's operation to machine instructions

The design of C typically favors efficiency

Types: Java v. C

70

In Java

- ▶ primitive types (int, float, char,...)
- ▶ object types (each object has a set of fields and methods)
- ▶ type conversion are checked at runtime to forbid nonsensical conversions, e.g., int to Object

In C, types have a less rigid definition

- ▶ are a convenient way of reasoning about memory layout
- ▶ all values (regardless of their type) have a common representation as a sequence of bytes in memory
- ▶ primitive type conversions are always legal

Byte

71

A byte = 8 bits

- ▶ Decimal 0 to 255
- ▶ Hexadecimal 00 to FF
- ▶ Binary 00000000 to 11111111

In C:

- ▶ Decimal constant: 12
- ▶ Octal constant: 014
- ▶ Hexadecimal constant: 0xC

0 _{hex}	=	0 _{dec}	=	0 _{oct}	0	0	0	0
1 _{hex}	=	1 _{dec}	=	1 _{oct}	0	0	0	1
2 _{hex}	=	2 _{dec}	=	2 _{oct}	0	0	1	0
3 _{hex}	=	3 _{dec}	=	3 _{oct}	0	0	1	1
4 _{hex}	=	4 _{dec}	=	4 _{oct}	0	1	0	0
5 _{hex}	=	5 _{dec}	=	5 _{oct}	0	1	0	1
6 _{hex}	=	6 _{dec}	=	6 _{oct}	0	1	1	0
7 _{hex}	=	7 _{dec}	=	7 _{oct}	0	1	1	1
8 _{hex}	=	8 _{dec}	=	10 _{oct}	1	0	0	0
9 _{hex}	=	9 _{dec}	=	11 _{oct}	1	0	0	1
A _{hex}	=	10 _{dec}	=	12 _{oct}	1	0	1	0
B _{hex}	=	11 _{dec}	=	13 _{oct}	1	0	1	1
C _{hex}	=	12 _{dec}	=	14 _{oct}	1	1	0	0
D _{hex}	=	13 _{dec}	=	15 _{oct}	1	1	0	1
E _{hex}	=	14 _{dec}	=	16 _{oct}	1	1	1	0
F _{hex}	=	15 _{dec}	=	17 _{oct}	1	1	1	1

Words

72

Hardware has a `Word size` used to hold integers and addresses

The size of address words defines the maximum amount of memory that can be manipulated by a program

Two common options:

- ▶ 32-bit words => can address 4GB of data
- ▶ 64-bit words => could address up to 1.8×10^{19}

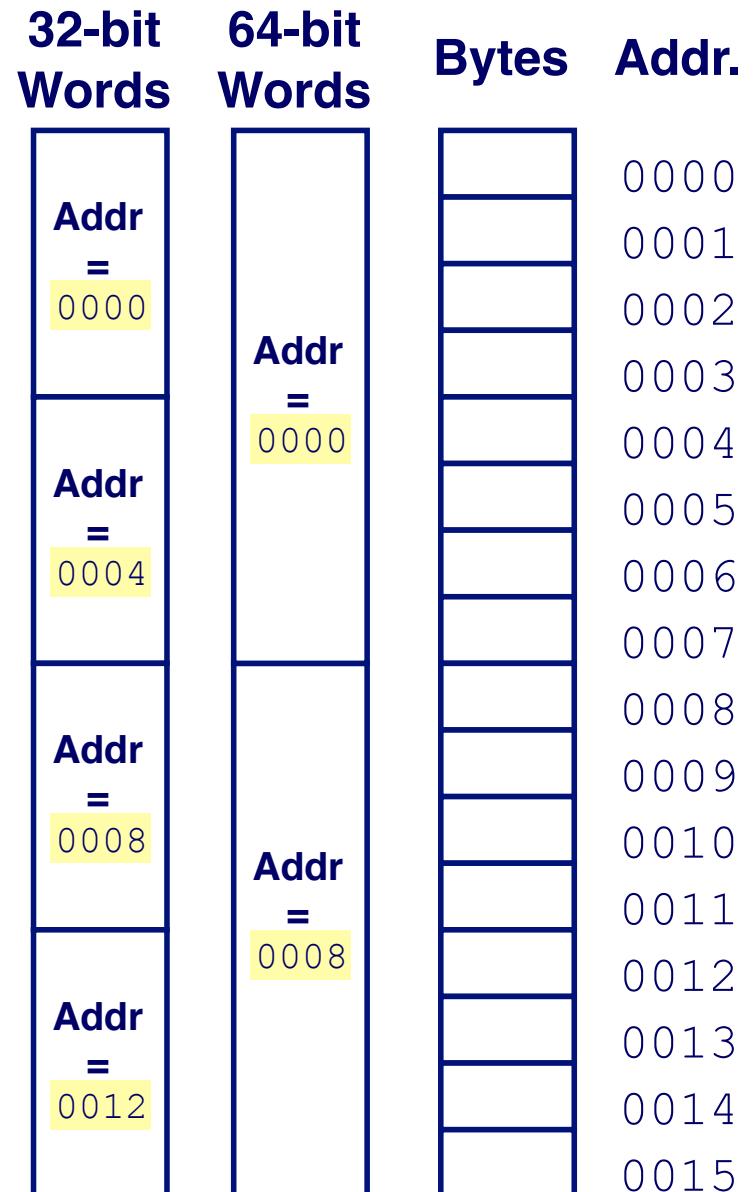
Different words sizes (integral number of bytes, multiples and fractions) are supported

Addresses

73

Addresses specify byte location in computer memory

- ▶ address of first byte in word
- ▶ address of following words differ by 4 (32-bit) and 8 (64-bit)



Data Types

74

The base data type in C

- ▶ **int** - used for integer numbers
- ▶ **float** - used for floating point numbers
- ▶ **double** - used for large floating point numbers
- ▶ **char** - used for characters
- ▶ **void** - used for functions without parameters or return value
- ▶ **enum** - used for enumerations

The composite types are

- ▶ pointers to other types
- ▶ functions with arguments types and a return type
- ▶ arrays of other types
- ▶ **structs** with fields of other types
- ▶ **unions** of several types

Qualifiers, Modifiers & Storage

75

Type qualifiers

- ▶ **short** - decrease storage size
- ▶ **long** - increase storage size
- ▶ **signed** - request signed representation
- ▶ **unsigned** - request unsigned representation

Type modifiers

- ▶ **volatile** - value may change without being written to by the program
- ▶ **const** - value not expected to change

Storage class

- ▶ **static** - variable that are global to the program
- ▶ **extern** - variables that are declared in another file

Sizes

76

Type	Range (32-bits)	Size in bytes
signed char	-128 to +127	1
unsigned char	0 to +255	1
signed short int	-32768 to +32767	2
unsigned short int	0 to +65535	2
signed int	-2147483648 to +2147483647	4
unsigned int	0 to +4294967295	4
signed long int	-2147483648 to +2147483647	4 or 8
unsigned long int	0 to +4294967295	4 or 8
signed long long int	-9223372036854775808 to +9223372036854775807	8
unsigned long long int	0 to +18446744073709551615	8
float	1×10^{-37} to 1×10^{37}	4
double	1×10^{-308} to 1×10^{308}	8
long double	1×10^{-308} to 1×10^{308}	8, 12, or 16

Character representation

77

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

Examples:

- ▶ The code for 'A' is 65
- ▶ The code for 'a' is 97
- ▶ The code for 'b' is 98
- ▶ The code for '0' is 48
- ▶ The code for '1' is 49

Understanding types matter...

78

Types define an abstraction or approximation of a computation....

More practically, there are implicit conversions that take place and they may result in truncation, and ...

Some data types are not interpreted the same on different platforms, they are machine-dependent

- ▶ `sizeof(x)` returns the size in bytes of the object `x` (either a variable or a type) on the current architecture

Declarations

79

The declaration of a variable allocates storage for that variable and can initialize it

```
int lower = 3, upper = 5;
char c = '\\', line[10], he[3] = "he";
float eps = 1.0e-5;
char arrdarr[10][10];
unsigned int x = 42U;
char* ardarr[10];
char* a;
void* v;
void foo(const char[]);
```

Without an explicit initializer local variables may contain random values (static & extern are zero initialized)

Conversions

80

What is the meaning of an operation with operands of different types?

```
char c; int i; ... i + c ...
```

The compiler will attempt to convert data types without losing information; if not possible emit a warning and convert anyway

Conversions happen for operands, function arguments, return values and right-hand side of assignments.

Conversions

81

T op T': //symmetrically for T'

if T=**long double** then convert **long double**
elseif T=**double** then convert **double**
elseif T=**float** then convert **float**
elseif T=**unsigned long int** then convert **unsigned long int**
elseif T=**long int** then convert **long int**
elseif T=**unsigned int** then convert **unsigned int**

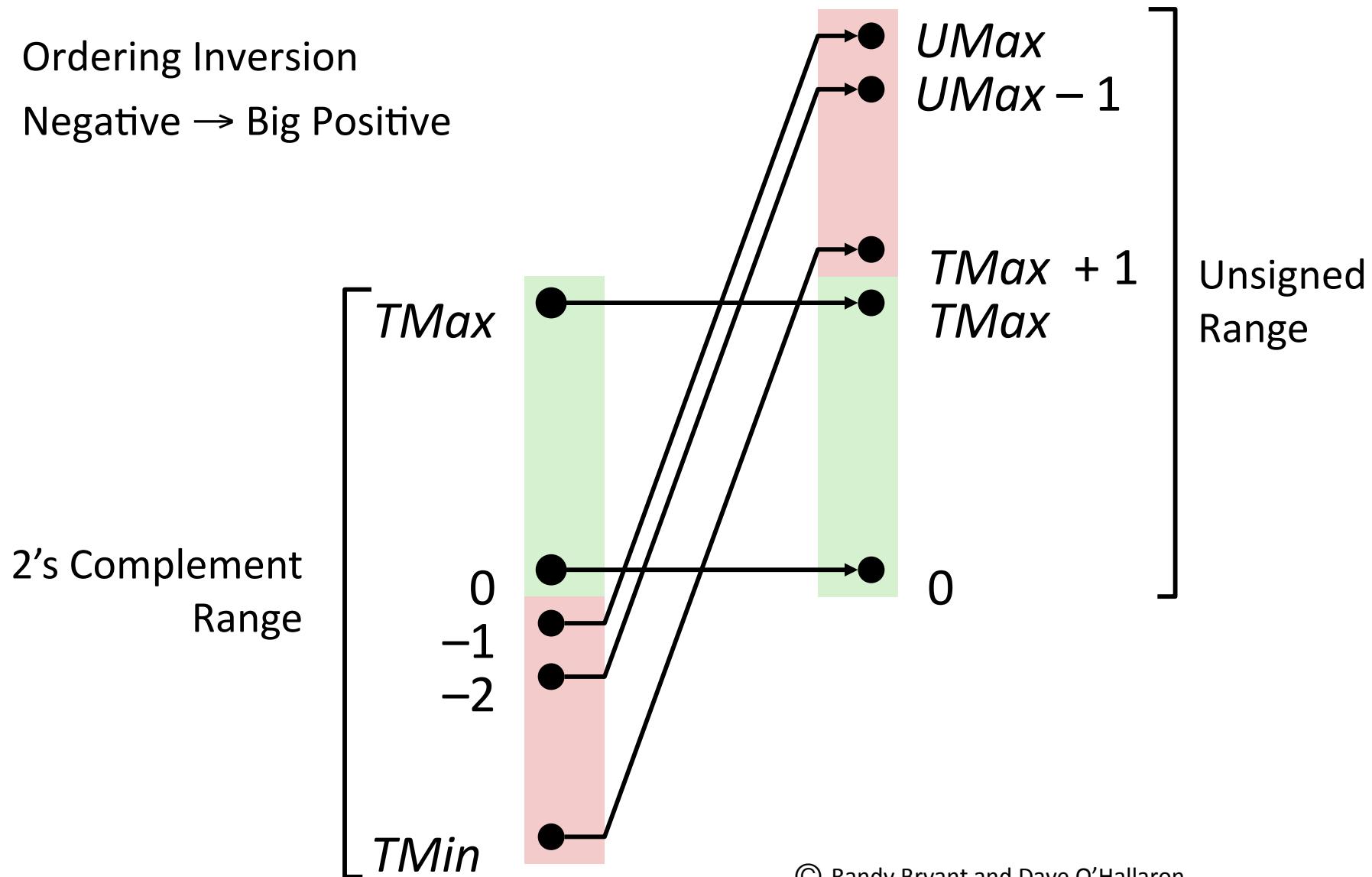
- ▶ Conversions between signed and unsigned integers slightly surprising due to two's complement representation (look it up)
- ▶ character can be converted to integral types

Conversion

82

signed to unsigned conversion

- Ordering Inversion
- Negative \rightarrow Big Positive



Casts

83

(T) x

A cast converts the value held in variable x to type T

With pointers, casts do not affect the content of the variable pointed (merely an indication to the compiler):

```
char* c; int* i;  
i = (int *) c;
```

Relational and Logic operations

84

`<` `<=` `>` `>=` `==` `!=`

- ▶ Compare any two types and return 1 for true and 0 for false

& &, | | , !

- ▶ 0 is false, all other values true
 - ▶ return 0 or 1
 - ▶ terminates as early as possible (first 0 for &&, and first non-0 for ||)

Examples

- ▶ `1 && 0` \Rightarrow `0`
 - ▶ `1 || 0` \Rightarrow `1`
 - ▶ `! 42` \Rightarrow `0`
 - ▶ `p && *p` \Rightarrow `(never follows a null pointer)`

Bitwise operations

85

Boolean algebra has basic operations for manipulating sets

& set intersection

| set union

^ set symmetric difference

~ set complement

A set of n element can be represented as vector of n bits

► {0, 1, 4} can be represented 00001011

Any integral type can be treated as bit vector

► preferably use unsigned values

01101001

01101001

01101001

& 01010101

| 01010101

^ 01010101

~ 01010101

01000001

01111101

00111100

10101010

Aparté: conversion pitfalls

86

A security example (from Bryant & O'Hallaron)

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

Aparté: conversion pitfalls

87

Typical usage

```
#define SIZE 256
...
char buf[SIZE];
copy_from_kernel(buf, SIZE);
```

```
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

Aparté: conversion pitfalls

88

Malicious usage

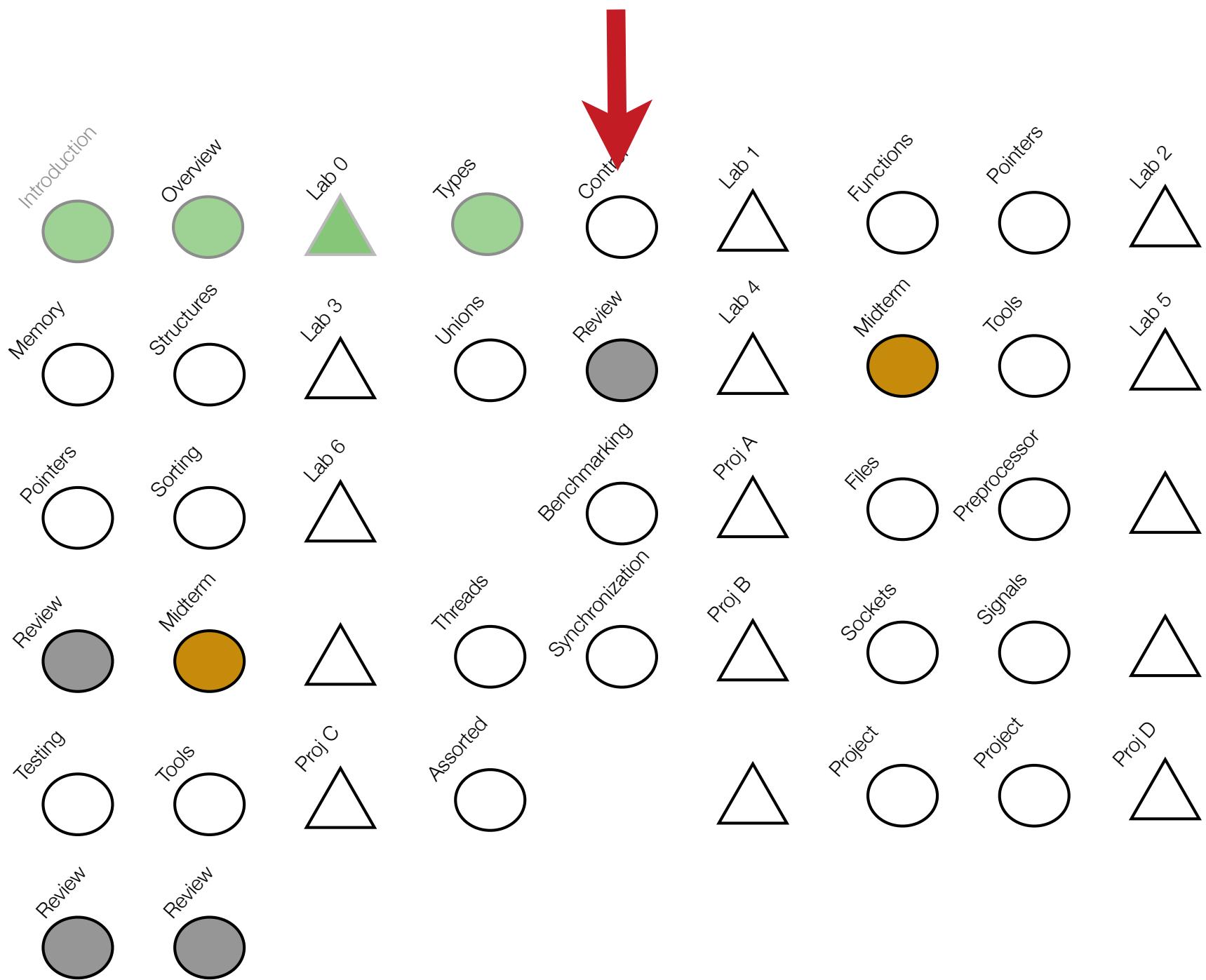
```
copy_from_kernel(buf, -SIZE);
=> int len = KSIZE < maxlen ? KSIZE : maxlen;
=> int len = 0 ? KSIZE : maxlen;
=> int len = maxlen = -256
=> mem_copy(user_dest, kbuf, -256) //3rd arg is
                                         //unsigned int
=> Buffer overflow!
```

```
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

{Loc}

Lecture 4





<http://tinyurl.com/4yvv79a>

Quiz #1 (take two)

Quiz #1 - 1



92

Is the following true?

$$145 == 0145$$

(a) true

(b) false

Quiz #1 - 2



93

What is printed?

```
if (42) printf("thanks"); else printf("fish");
```

(a) thanks

(b) fish

Quiz #1 - 3



94

Precedence is used by the compiler to parse complex expressions, consider

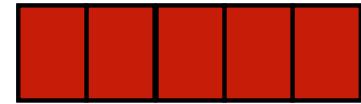
c=getchar() != '\n'

Is it parsed as:

(a) (c=getchar()) != '\n'

(b) c=(getchar()) != '\n'

Quiz #1 - 4



95

In C operator + has higher precedence than operator == and both are left to right associative. Should:

$$a + b + c == d$$

be read as:

- (a) $((a + b) + c) == d$
- (b) or as $(a + (b + (c == d)))$

Quiz #1 - 5



96

Is the variable referring to a zero-terminated character string?

```
char c[2] = "hi";
```

- (a) yes
- (b) no
- (c) depends on the architecture

Quiz #1 - 6



97

What will this program print?

```
for (int i=0; i<3; i++) {  
    if(i==0) continue;  
    printf("1");  
    if(i>0) break;  
}
```

- (a) nothing
- (b) 1
- (c) 11
- (d) 111

Statements and Blocks

98

The semi-colon is a terminator in C which turns an expression into a statement and allow to sequence the execution of multiple statements

```
x =0; i++; f(bar);
```

Another way to sequence expressions is with the comma operator which returns the value of the last expression

```
x = (i++,f(bar),0);
```

A group of declarations and statements can composed into a compound statement using braces { and }.

- ▶ Variable declared within the block are only visible in that block

The conditional statement **if-else** requires an expression and two statements (or one if the **else** is omitted)

Examples:

```
if ( x != 0 ) { x++; } else { x = x - 1; }
if ( x ) { x++; } else { x = x - 1; }
if ( x ) x++; else x = x - 1;
```

The conditional expression **e?e:e** can be used in certain cases for compactness

```
x = ( x ) ? x+1 : x-1;
```

or

```
x += x? 1:-1;
```

Switch

100

The switch statement allows multi-way branching, it takes an integer valued expression and a number of constant-labeled branches

```
switch ( b[i++]) {  
    case '1': case '3': case '5': case '7': case '9':  
        odd++;  
        break;  
    default:  
        even++;  
}
```

Break and continue

101

break leaves the current loop or switch, **continue** goes to the top of the loop

```
while (1) {  
    for(int i=0;i<10;i++) {  
        if (i&1) continue;  
        if (i==8) break;  
    }  
}
```

Goto and labels

102

goto can jump to any label in the current function

```
while (1) {  
    for(int i=0;i<10;i++) {  
        if (i&1) continue;  
        if (i==8) goto error;  
    }  
}  
  
error:  
    printf("oops");
```

External and static variables

103

C differentiate variable **definitions** from **declarations**. The latter allocates storage while the former does not.

A **declaration** is like a *customs declaration*

it is not the thing itself, merely
a description of some baggage that
you say you have around somewhere

a **definition** is the special kind of declaration
that fixes the storage for that thing

External and static variables

104

```
file1.c: int mytime = 0; // definition
```

```
file2.c: extern int mytime; // declaration
```

```
extern char msg[]; //declaration doesn't need dimension
int i; //both declaration and definition
int i = 10; //var definition can be initialized
            // but extern declarations should not
char msg[100]; //array definition must have dimension
```

Static variable retain their value across invocations

```
int count() {
    static int j;
    return j++;
}
```

Extern modifier (functions)

105

On function declaration ensures that params are interpreted correctly

```
extern void solve(int [], int, int);
```

The function definition may be in another file, or in the same file

Appears in header files, included by callee and all the callers

Static modifier (functions)

106

Placed before a function declaration or definition

```
static void solve(int [], int, int);
```

Limits visibility of function to the local file

Functions without static are visible to all other source files

Aparté: What's function call?

107

fact(1)

C code

call fact

...

ret

Assembly

Start of fact

8048394 :

...

call 8048394

...

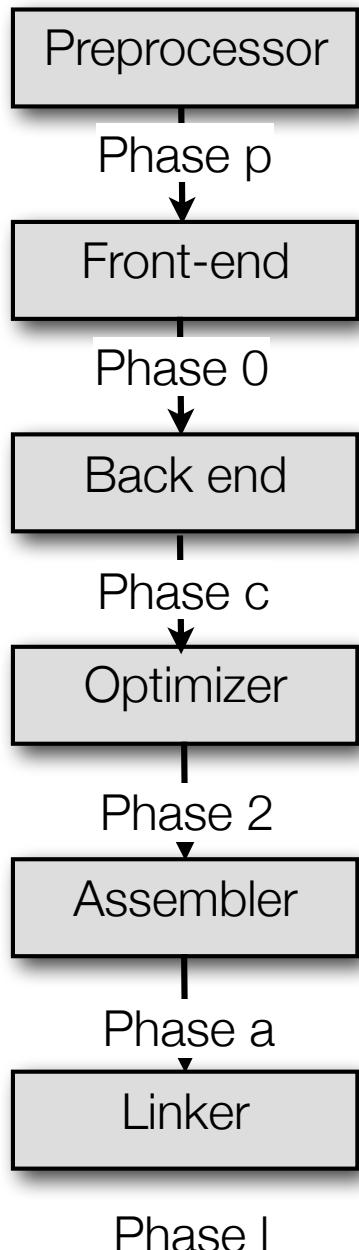
ret

Disassembled code

How does the compiler deal with separate compilation?

The 6 Stages of Compilation

108



Debugging Options

-dletters -dumpsspecs -dumpmachine -dumpversion ...

Optimization Options

-falign-functions=n -finline-functions -fno-inline
-O -O0 -O1 -O2 ...

Preprocessor Options

-Dmacro[=defn] -E -H ...

Assembler Option

-Wa,option -Xassembler option ...

Linker Options

object-file-name -llibrary -nostartfiles -nodefaultlibs
-nostdlib -pie -rdynamic -s -static -shared ..

Code Generation Options

-fcall-saved-reg -fcall-used-reg -ffixed-reg -fexceptions
-fnon-call-exceptions -funwind-tables...

Trivia: Where does the name of for executables, **a.out**, comes from? A: “assembler output” ... even though it is an object file

Libraries

109

A library in C ...

- ▶ contains object files used together in the linking phase of a program
- ▶ is indexed, so it is easy to find function and variable symbols

Static libraries:

- ▶ collections of object files that are linked into the program

Examples:

- ▶ Unix: libXXX.a
- ▶ Windows: XXX.lib

Command line:

- ▶ `gcc -ltweetit.a main.c`

Libraries

110

Shared libraries:

- ▶ Only one copy of the library is stored in memory at any given time
 - use less memory to run programs, executable files are smaller
- ▶ Slightly slower start of the program

Static libraries:

- ▶ Each process has its own copy of the libraries it is using
- ▶ Executable files linked with static libraries are bigger

Shared Libraries

111

... are linked into the program in two stages

- ▶ At compile-time, the linker verifies that the symbols required by the program, are either linked into the program, or in one of its shared libraries.

The object files from the dynamic library are not inserted into the executable file

- ▶ When the program is started, a program in the system (called a dynamic loader or linker) checks out which shared libraries were linked with the program, loads them to memory, and attaches them to the copy of the program in memory

Libraries

112

Creating a static library is a two step process

First compile the code to an object file

```
gcc -c tweet.c -o tweet.o
```

Then, the archiver (ar) is invoked to produce a static library (named tweetit.a) out of the object file tweet.o.

```
ar rcs libtweetit.a tweet.o
```

Aparté: Writing Unmaintainable Code

113

© Roedy Green

Principles

- ▶ To foil the maintenance programmer, you have to understand how he thinks. He has your giant program. He has no time to read it all, much less understand it. He wants to rapidly find the place to make his change, make it and get out with no unexpected side effects.
- ▶ He views your code through a tube taken from the center of a roll of toilet paper. He can only see a tiny piece of your program at a time. You want to make sure he can never get the big picture from doing that. You want to make it as hard as possible for him to find the code he is looking for. But even more important, you want to make it as awkward as possible for him to safely ignore anything.

Aparté: Writing Unmaintainable Code

114

Techniques

- ▶ 1. Lie in the comments. You don't have to actively lie, just fail to keep comments up to date with the code.
- ▶ 2. Pepper code with comments like `/*add 1 to i*/` however, never document wooly stuff like the overall purpose of a function.
- ▶ 3. Make sure that every function does a little bit more (or less) than its name suggests. As a simple example, a `isValid(x)` should as a side effect convert `x` to binary and store the result in a database.

Aparté: Writing Unmaintainable Code

115

- ▶ 20. **Never use local variables.** Whenever you feel the temptation to use one, make it a static variable instead to unselfishly share it with all other functions. This will save you work later when other functions need similar declarations.

- ▶ 23. **In naming functions, make heavy use of abstract words** like *it*, *everything*, *data*, *handle*, *stuff*, *do*, *routine*, *perform* and digits e.g. `routineX48`,
`PerformDataFunction`, `DoIt`, `HandleStuff` and
`do_args_method`

Aparté: Writing Unmaintainable Code

116

- ▶ 72. If you cannot find the right English word to convey the meaning of a variable, use a foreign word. For example, instead of using `p` for a point, you may use `punkt`, the German word for it. Maintenance coders without your firm grasp of German will enjoy the multicultural experience of deciphering the meaning. *It breaks the tedium of an otherwise tiring and thankless job.*

Aparté: goto considered harmful

117

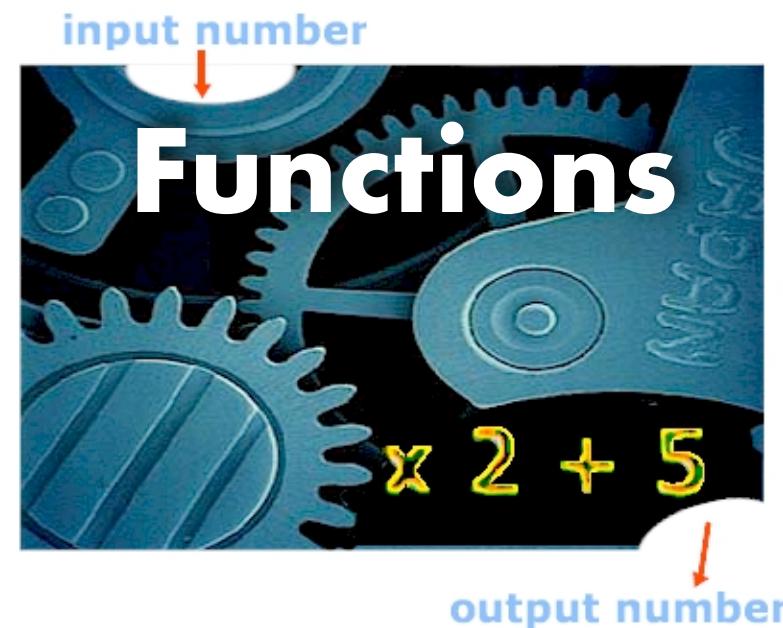
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce.

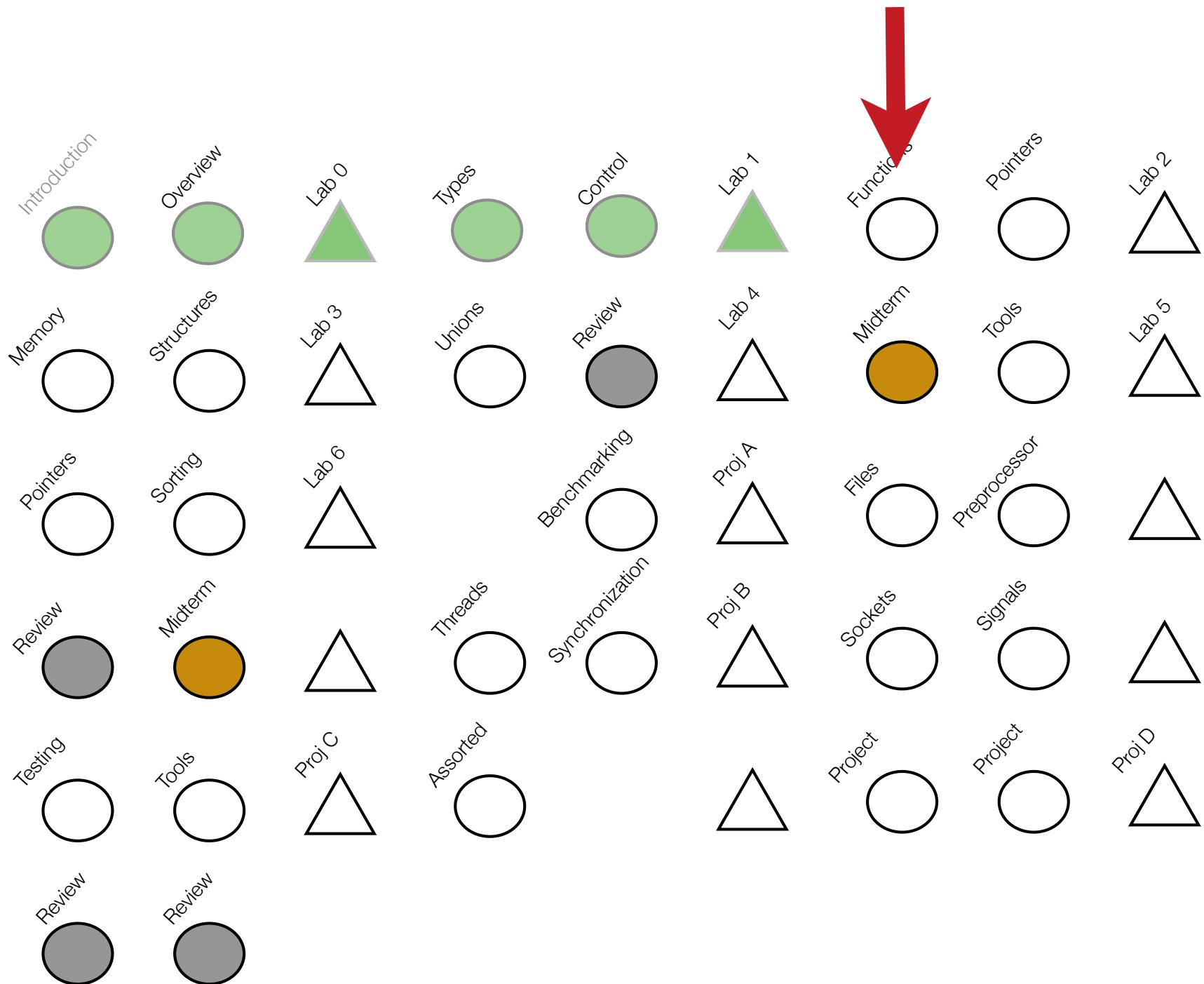
... our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

E.W. Dijkstra, 1968

{LC}

Lecture 5





<http://tinyurl.com/4yvv79a>

Q 2

Quiz #2 - 1



121

A recursive function is

- (a) a function that contains a tight loop
- (b) a function that contains a call to itself
- (c) a function that calls itself directly or indirectly

Quiz #2 - 2



122

What are the values of the external variable x, the static variable y and the automatic variables z and w?

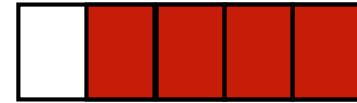
```
int x;  
static int y;  
void f() {  
    int z = 0, w;  
}
```

(a) x=42 y=42 z=42 w=42

(b) x=0 y=0 z=0 w=42

(c) x=42 y=42 z=0 w=42

Quiz #2 - 3



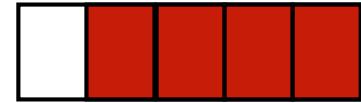
123

Consider the scope of variable i. What will this program print?

```
int main(){ int i = 42;
    if (1) { int i = 0; }
    printf("%d",i);
}
```

- (a) 42
- (b) 0
- (c) architecture dependent

Quiz #2 - 4



124

What is the meaning of the static qualifier in front of a variable declaration?

- (a) the automatic variable retains its value between calls
- (b) the scope of the extern variable is limited to the current file
- (c) both

Functions

125

Declaration defines the interface of a function; i.e., number and types of parameters, type of return value

- ▶ A C prototype is an explicit declaration

```
int strl(char []);
```

First **use** of a function without a preceding prototype declaration implicitly declares the function

- ▶ If prototype follows the first use => error

Definition gives the implementation of a function

```
int strl(char s[]) {  
    int i = 0;  
    while(s[i] != '\0')  
        ++i;  
    return i;  
}
```

Aparté: one liner

126

```
int strl(char* s) {  
    for(int i=0;1;++i) if(*(s+i)) return i;  
}
```

Aparté: one liner

127

```
int strl(char* s) {  
    for(int i = 0; 1; ++i)  
        if(!*(s+i)) return i;  
}
```

Aparté: shorter

128

```
int strl(char* s) {  
    for(char* p=s; 1; ++p)  
        if(!*p) return p-s;  
}
```

Return statement

129

```
return [ () [ expression] ()];
```

- ▶ Terminates the execution of a function and returns control to caller
- ▶ Parenthesis are optional
- ▶ Converted to declared return type
- ▶ Return value can be ignored by caller

Parameter passing

130

Historically programming languages have offered:

- ▶ passed by value
- ▶ passed by reference
- ▶ passed by value-result

By-value semantics:

- ▶ Copy of param on function entry, initialized to value passed by caller
- ▶ Updates of param inside callee made only to copy
- ▶ Caller's value is not changed (updates to param not visible after return)

To swap or not to swap?

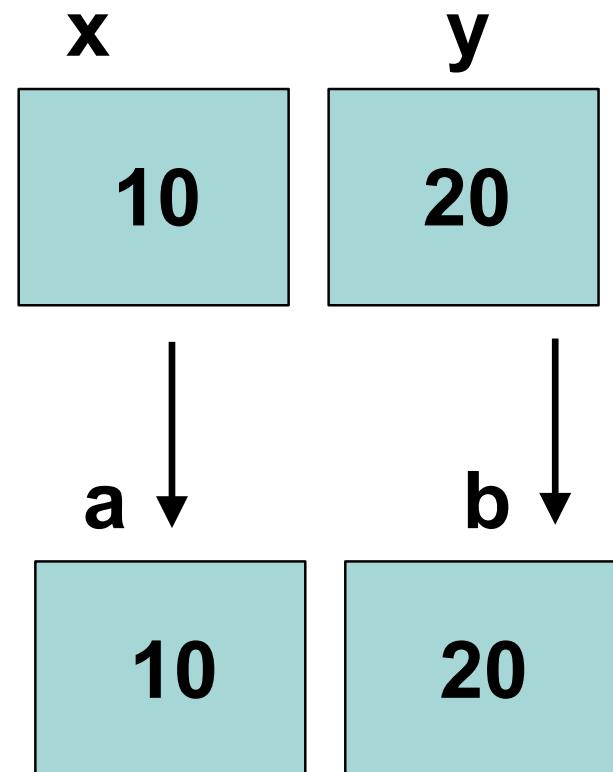
131

```
int y = 20, x = 10;
```

```
swap(x, y);
```

```
void swap(int a, int b) {
```

```
    int t = a;  
    a = b;  
    b = t;  
}
```



To swap!

132

```
int y = 20, x = 10;
```

```
swap(&x, &y);
```

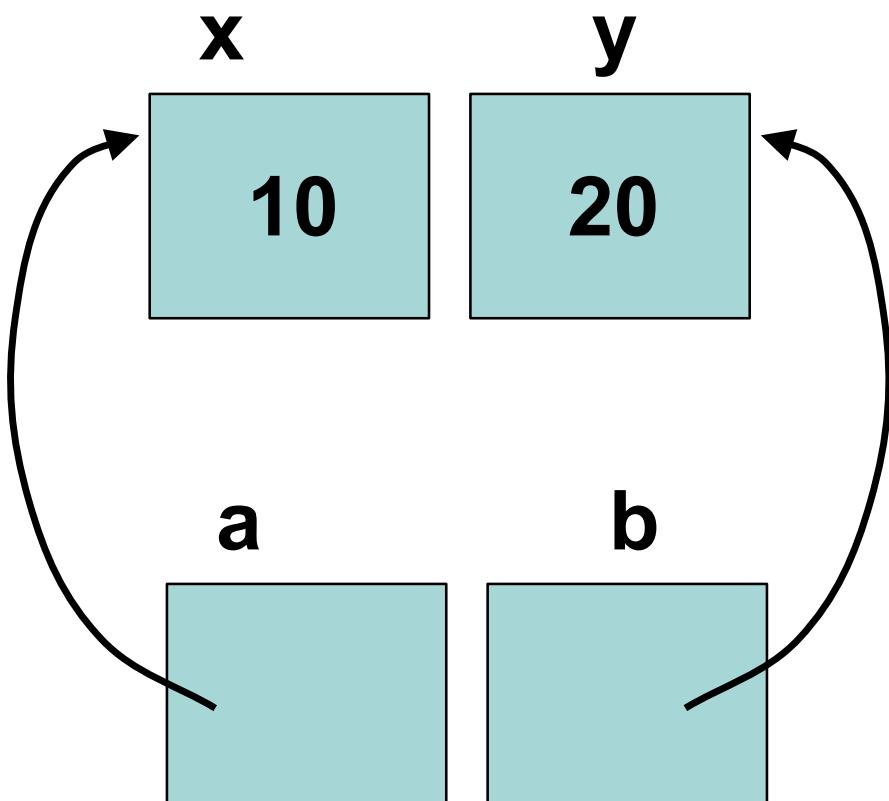
```
void swap(int* a, int* b) {
```

```
    int t = *a;
```

```
    *a = *b;
```

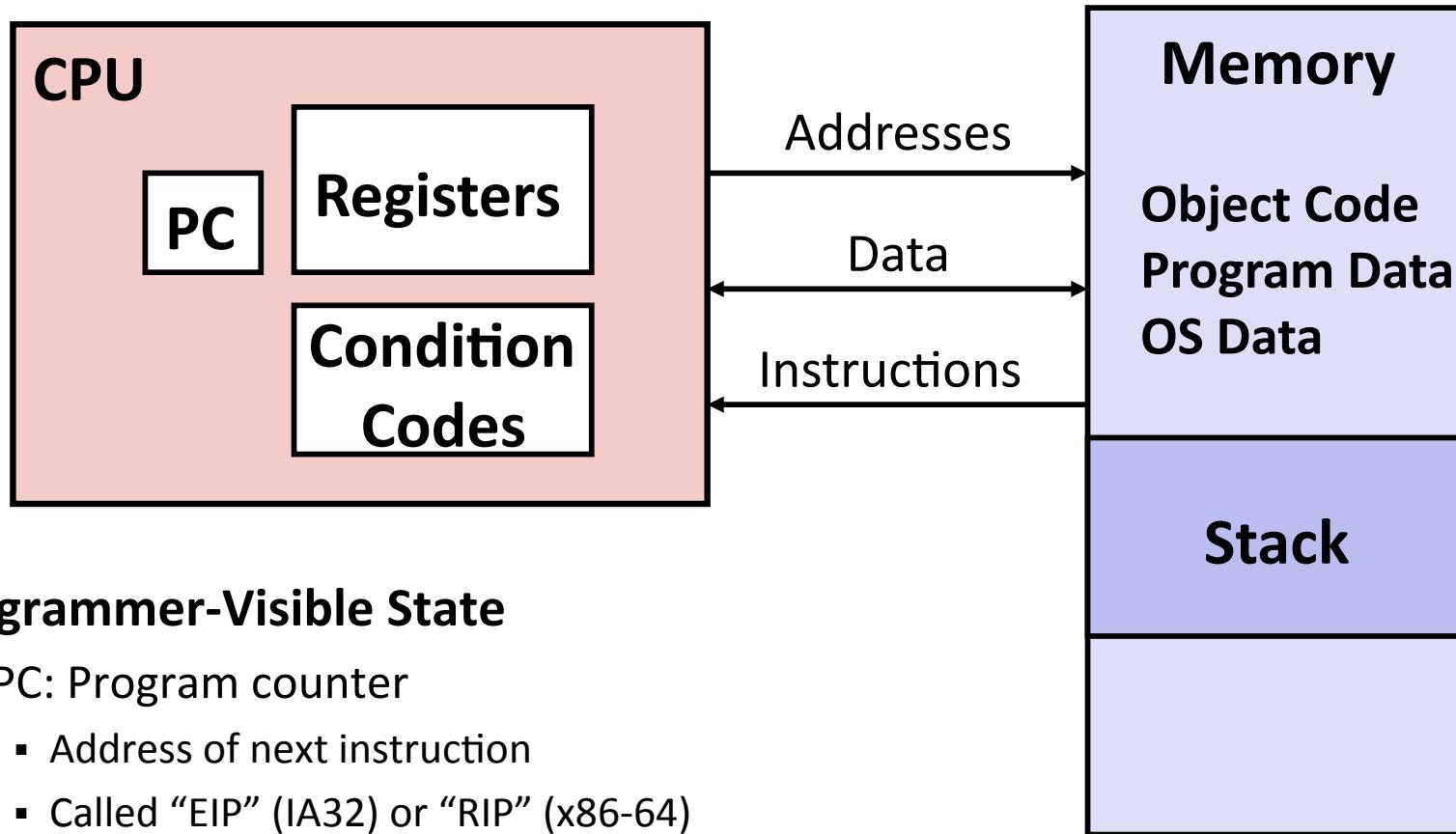
```
    *b = t;
```

```
}
```



Low level programmer's view

133



■ Programmer-Visible State

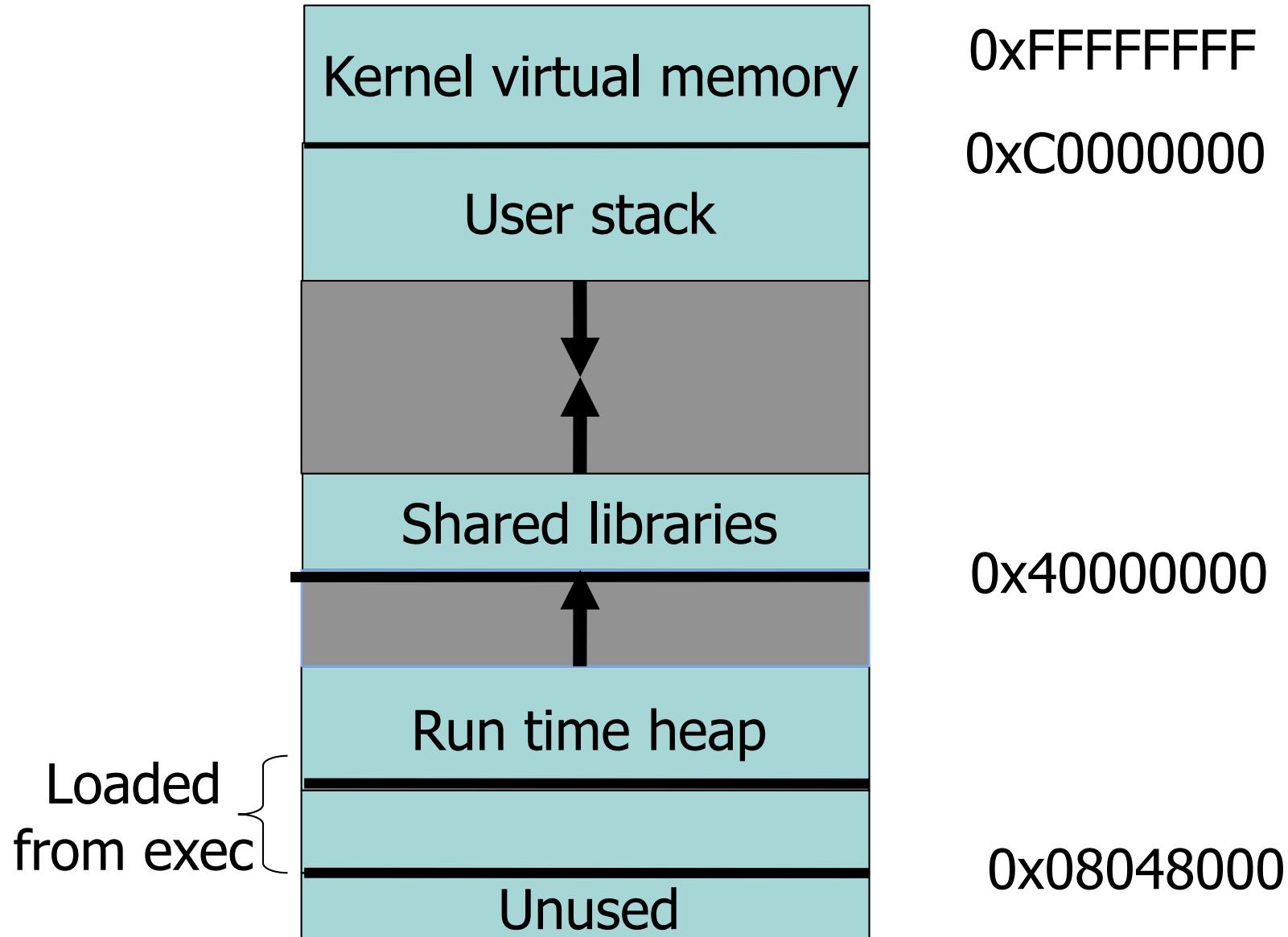
- PC: Program counter
 - Address of next instruction
 - Called “EIP” (IA32) or “RIP” (x86-64)
- Register file
 - Heavily used program data
- Condition codes
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

■ Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Linux process memory layout

134



Assembler

135

Low-level language that maps directly to hardware

- ▶ Perform arithmetic on registers or memory
- ▶ Transfer data between memory and registers
- ▶ Transfer control to procedures/returns or within a procedure

Machine Instruction Example

136

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

x += y

More precisely:

```
int eax;  
int *ebp;  
eax += ebp[2]
```

```
0x80483ca: 03 45 08
```

■ C Code

- Add two signed integers

■ Assembly

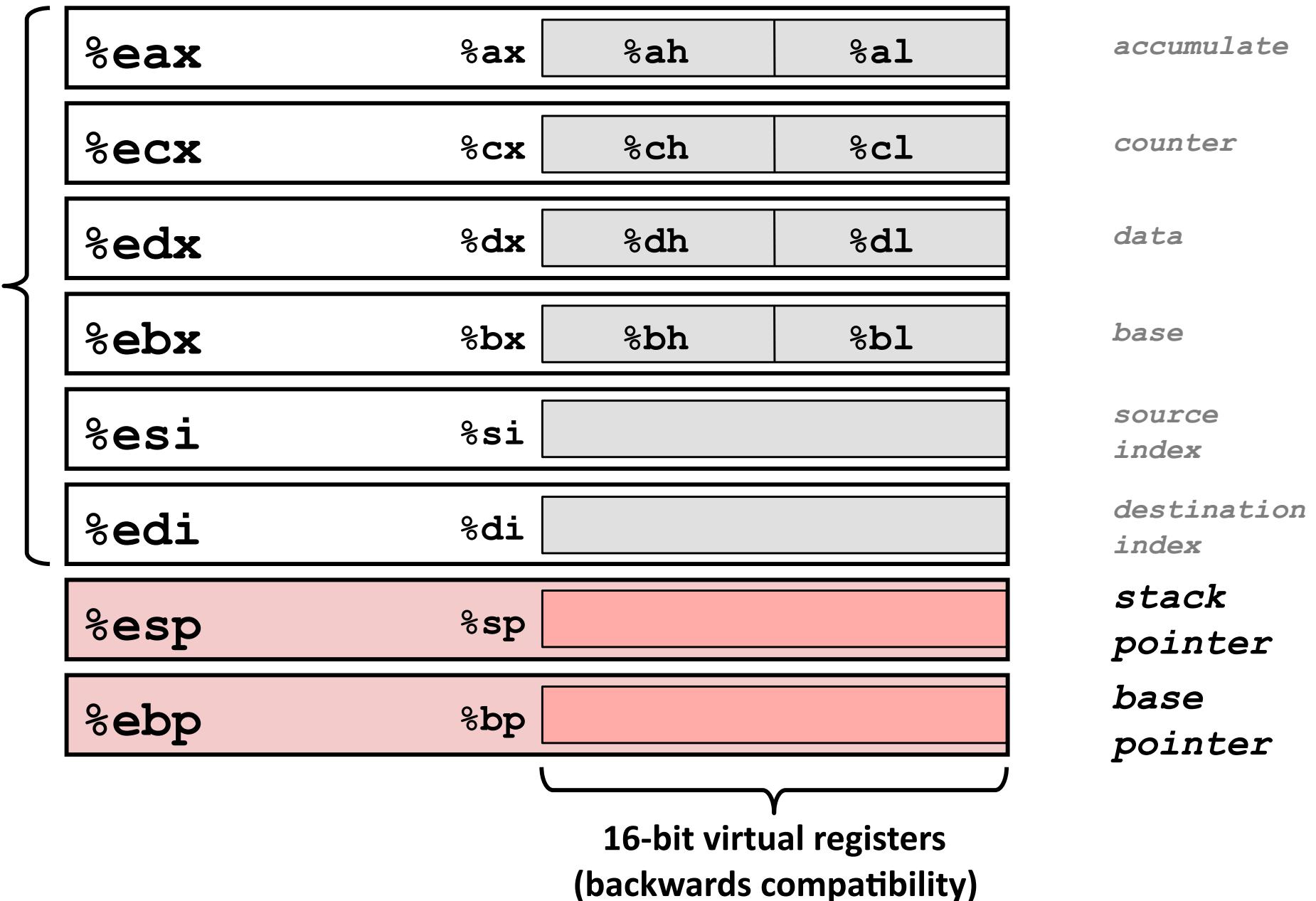
- Add 2 4-byte integers
 - “Long” words in GCC parlance
 - Same instruction whether signed or unsigned
- Operands:
 - x: Register %eax
 - y: Memory M[%ebp+8]
 - t: Register %eax
 - Return function value in %eax

■ Object Code

- 3-byte instruction
- Stored at address 0x80483ca

Intel registers

137



Example: move

138

movl Operand Combinations

	Source	Dest	Src,Dest	C Analog
movl	<i>Imm</i>	<i>Reg</i>	movl \$0x4,%eax	temp = 0x4;
		<i>Mem</i>	movl \$-147,(%eax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movl %eax,%edx	temp2 = temp1;
	<i>Reg</i>	<i>Mem</i>	movl %eax,(%edx)	*p = temp;
	<i>Mem</i>	<i>Reg</i>	movl (%eax),%edx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Swap

139

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

} Set Up

```
movl 8(%ebp), %edx  
movl 12(%ebp), %ecx  
movl (%edx), %ebx  
movl (%ecx), %eax  
movl %eax, (%edx)  
movl %ebx, (%ecx)
```

} Body

```
popl %ebx  
popl %ebp  
ret
```

} Finish

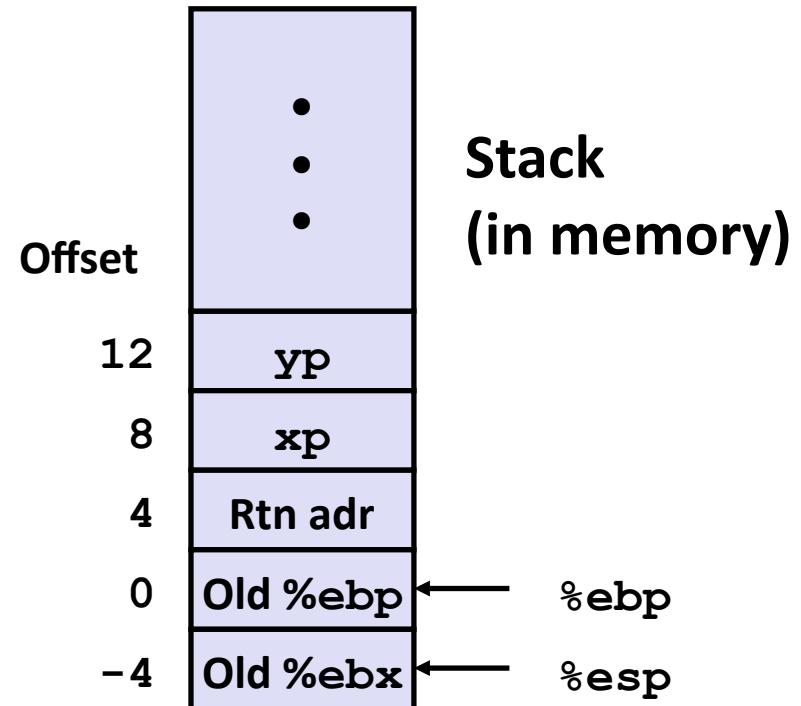
Example

140

Understanding Swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1



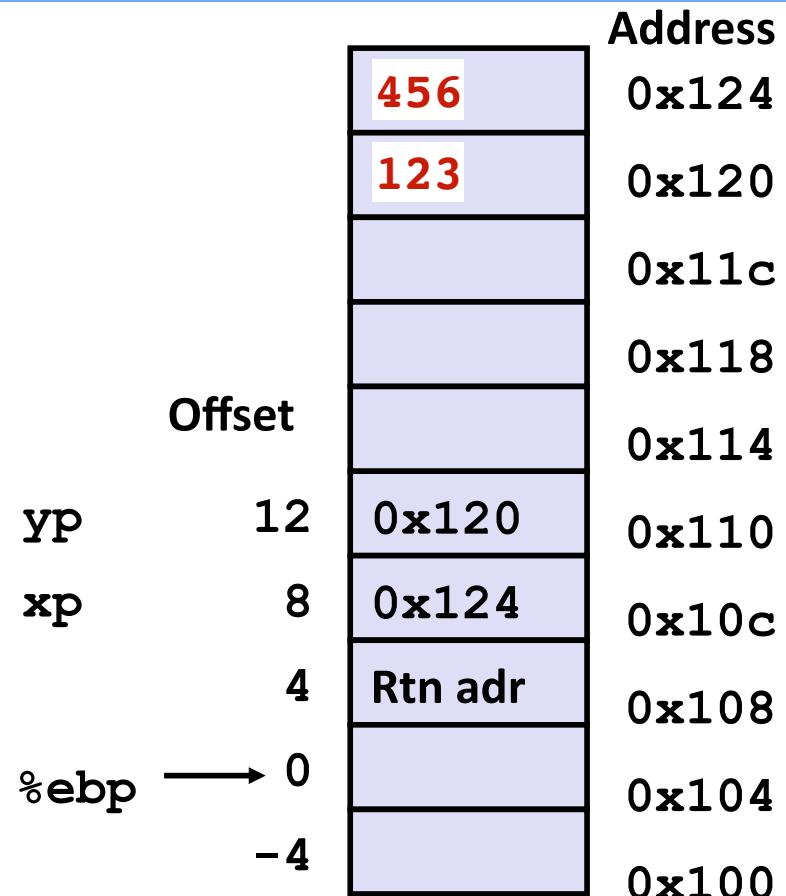
movl 8(%ebp), %edx	# edx = xp
movl 12(%ebp), %ecx	# ecx = yp
movl (%edx), %ebx	# ebx = *xp (t0)
movl (%ecx), %eax	# eax = *yp (t1)
movl %eax, (%edx)	# *xp = t1
movl %ebx, (%ecx)	# *yp = t0

Example

141

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104



```
movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx    # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)    # *yp = t0
```

Recursive functions

142

A function can call itself

- ▶ recursive expression of the function
- ▶ requires a stopping condition

Example: n!

```
int fact(int n) {  
    return (n<=1) ? 1 : n*fact(n-1);  
}
```

Why does it work?

- ▶ typically used to compute an inductively defined property

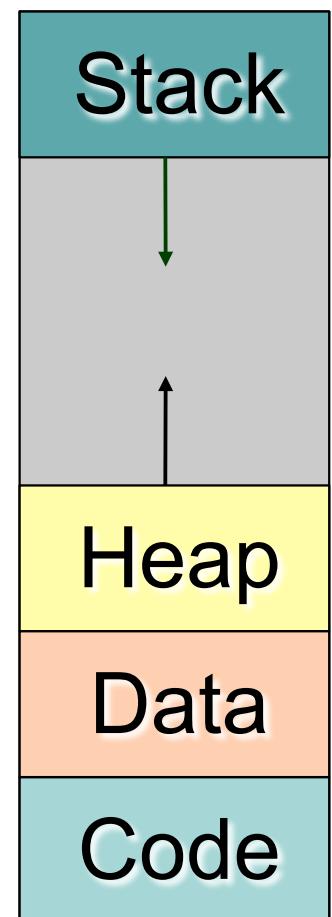
...it's all about the stack

143

The operating system creates a process by assigning memory and other resources

- ▶ Stack: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- ▶ Heap: dynamic memory for variables that are created with malloc, calloc, realloc and disposed of with free
- ▶ Data: initialized variables including global and static variables, un-initialized variables
- ▶ Code: the program instructions to be executed

Virtual Memory



The Stack

144

Logically it's a last-in-first-out (LIFO) structure

Two operations: push and pop

Grows 'down' from high-addresses to low

Operations always happen at the top

Holds "activation frames", i.e. state of functions as they execute

- ▶ top-most (lowest) frame corresponds to the currently executing function

Stores local variables and address of the function that needs to be executed next

C program execution

145

PC (program counter) points to next machine instruction to execute

Procedure call:

- ▶ Prepare parameters
- ▶ Save state (SP (stack pointer) and PC) and allocate on stack local variables
- ▶ Jumps to the beginning of procedure being called

Procedure return:

- ▶ Recover state (SP and PC (this is return address)) from stack and adjust stack
- ▶ Execution continues from return address

Stack frame

146

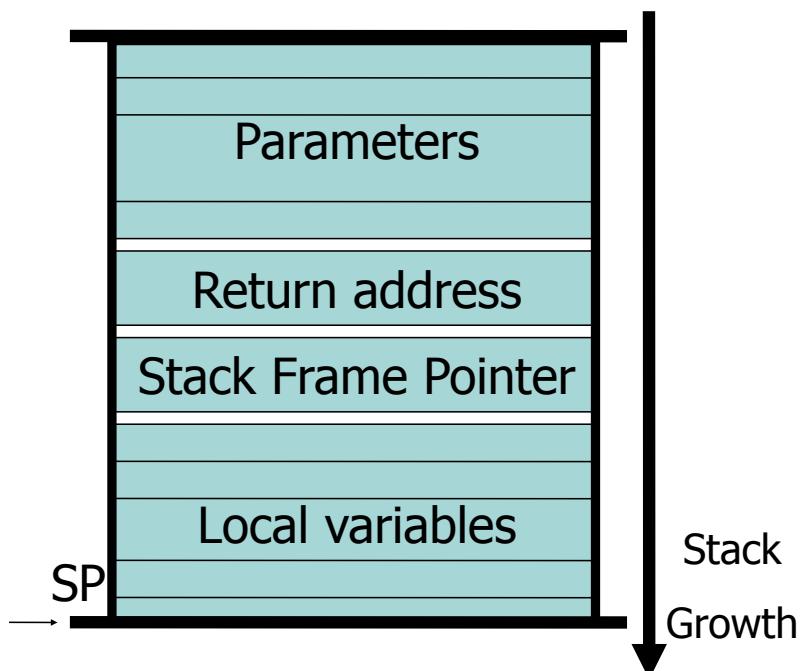
Parameters for the procedure

Save current PC onto stack (return address)

Save current SP value onto stack

Allocates stack space for local variables by decrementing SP by appropriate amount

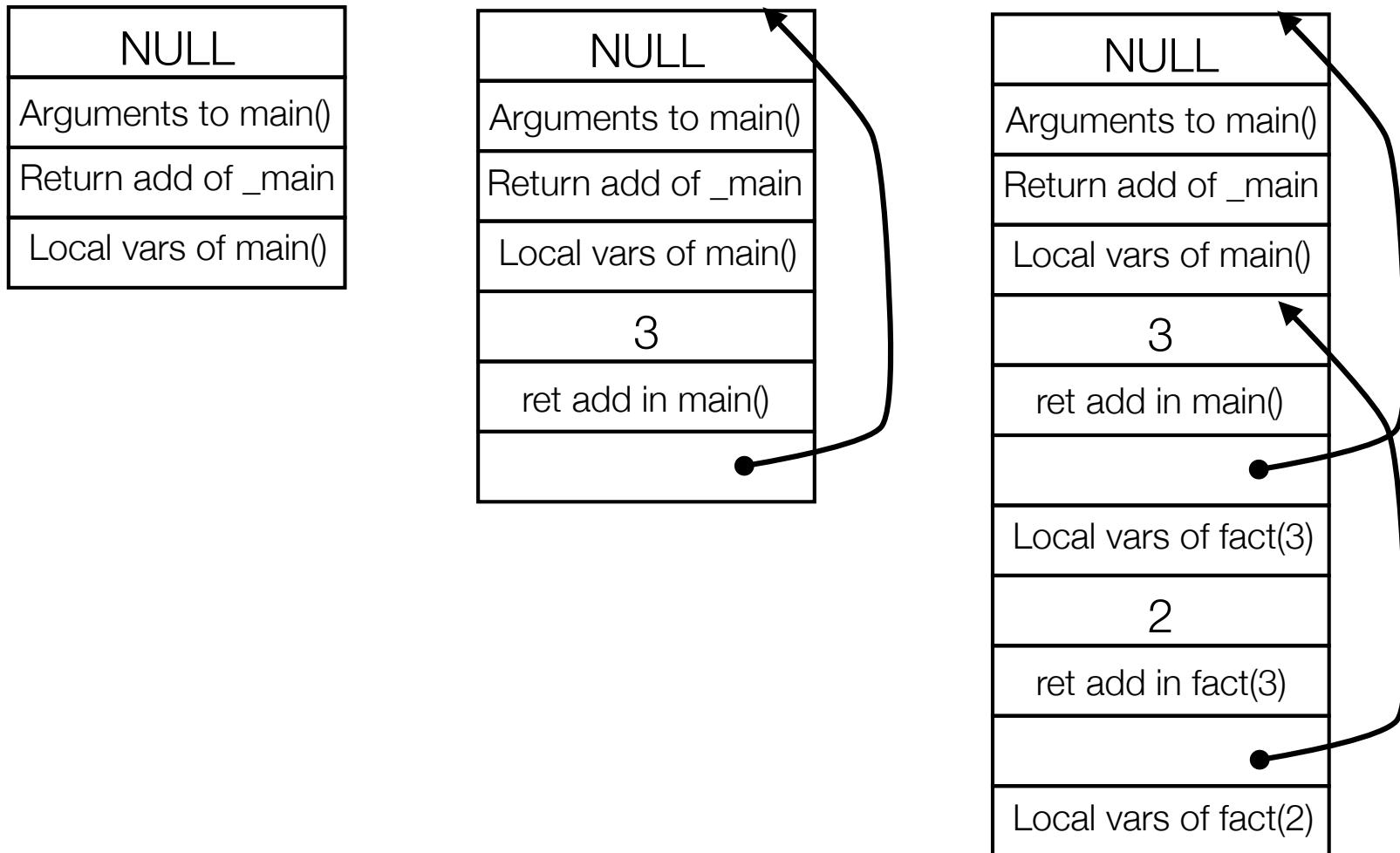
Return value passed by register



factorial(3) = ?

147

```
int fact(int n) {  
    return (n<=1) ?1: n*fact(n-1);  
}
```



Apparté: Tail recursion

148

Is a new stack frame needed for each recursive call?

```
int fact(int i, int acc) {  
    return (i<=1)? acc : fact(i-1,i*acc);  
}
```

Notice: nothing happens after the recursive call returns

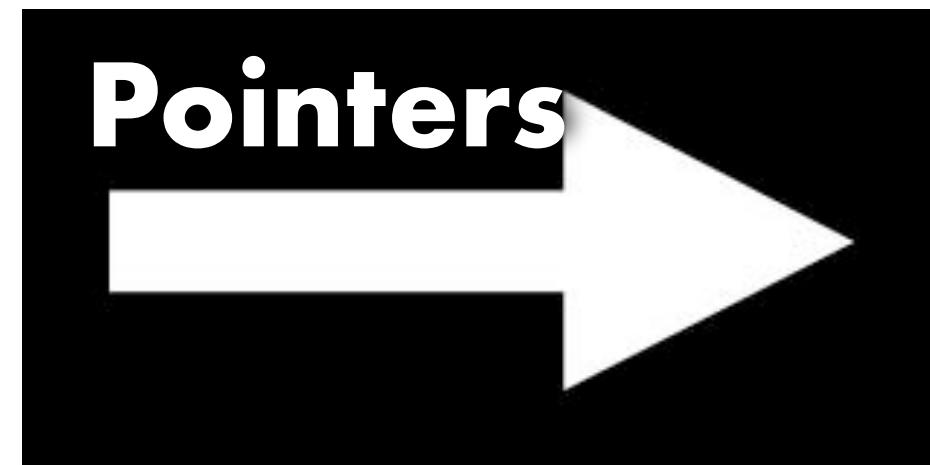
- ▶ Control is immediately transferred to the caller's caller
- ▶ The sequence of recursive calls behaves just like a loop
- ▶ No need to build up stack since no “context” is preserved across calls

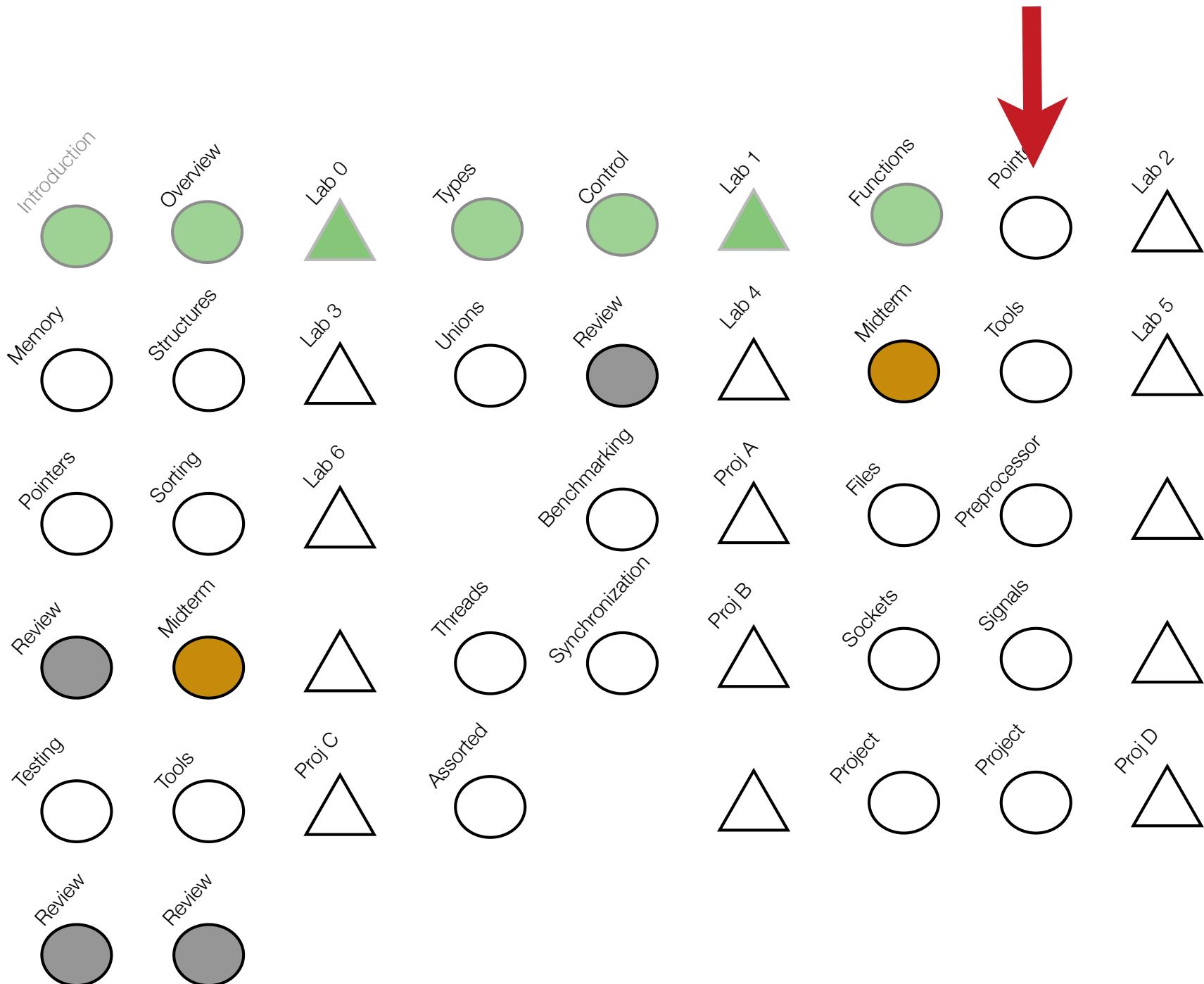
C does not provide support for tail recursion

- ▶ Recursion is not the same as looping

{C'}

Lecture 6





<http://tinyurl.com/4yvv79a>

Q 3

Quiz #3 - 0



152

Did you read K&R Chapter 5?

- (a) yes
- (b) no
- (c) architecture independent

Quiz #3 - 1



153

The unary operator `&x` gives?

- (a) the negation of x (e.g. 0 if x is 1)
- (b) the address of object x
- (c) the address of the location pointed to by x

Quiz #3 - 2



154

What does expression `(*ip)++` do?

- (a) increment the pointer ip
- (b) increment the value held in the location pointed to by ip

Quiz #3 - 3



155

Assume the following declarations

```
int x;  
void swap(int*a,int*b);
```

Is the call `swap(&x, &x)` legal and correct?

- (a) Yes, it will compile and run fine (but do nothing).
- (b) No, the compiler will report an error.
- (c) It will compile but there will be an error during execution

Quiz #3 - 4



156

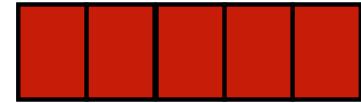
Assume the following declarations

```
char array[100];  
int strlen(char*s);
```

Is the call `strlen(array)` legal ?

- (a) Yes, it will compile
- (b) No, the compiler will report an error

Quiz #3 - 5



157

Does the following function print all command line arguments of a Unix program?

```
int main(int argc, char* argv) {  
    for(int i=1; i<argc; i++)  
        printf("%s ", argv[i]);  
}
```

- (a) Yes
- (b) No, it does not print the first argument
- (c) It depends what version of operating system we are on

Basics

158

char c; declares a variable of type character

char* pc; declares a variable of type pointer to character

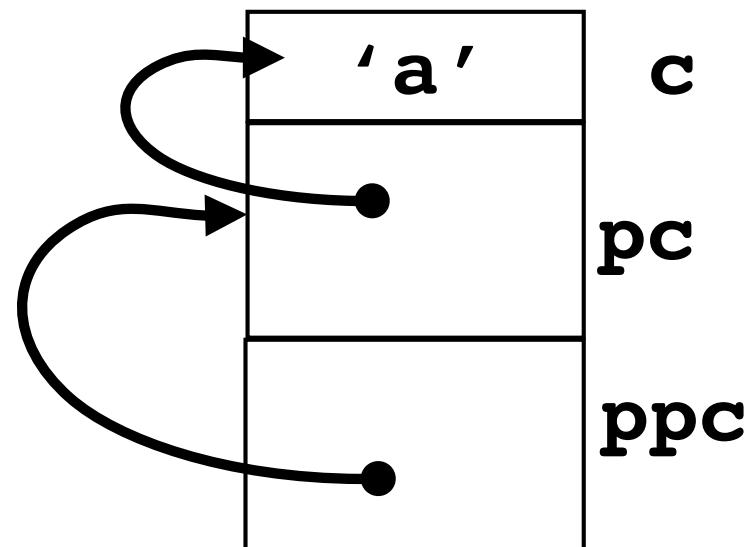
char ppc;** declares a variable of type pointer to pointer to character

c = 'a'; initialize a character variable

pc = &c; get the address of a variable

ppc = &pc; get the address of a variable

c == *pc == **ppc



Experimenting...

159

```
#include <stdio.h>

int main() {

    char c='a';
    char* pc=&c;
    char** ppc=&pc;

    printf("%p\n", pc);
    printf("%p\n", ++pc);
    printf("%p\n", ppc);
    printf("%p\n", ++ppc);
}
```

```
0x7fff6540097b
0x7fff6540097c
0x7fff65400970
0x7fff65400978
```

Basics

160

A variable declared as a pointer has the size of a memory address on the current architecture (e.g. 4 bytes or 8 bytes)

Incrementing a pointer adds a multiple of the pointer target size (e.g. 1 for characters, 2 for short, ...)

Pointers are initialized with addresses obtained by the & operator or the value **NULL**

A pointer can be dereferenced by prefix a pointer value with the * operator

Attempting to dereference a **NULL** pointer will result in an error caught by the hardware (bus error or segmentation fault)

Examples

161

char c = 'a'; value of c = 97, address of c=0xc00f4a20

char* pc = &c; value of pc=0xc00f4a20, address of pc=0xc00ea1c

pc value 0xc00f4a20

***pc** value 97

****pc** compile warning, runtime error

c value 97

&c value 0xc00f4a20

&&c compile error

...into the void

162

The type `void` has no values and `sizeof(void)` is undefined
A pointer of type `void*` is a generic pointer, that can not be used without first casting it to a more precise type

This type is used by functions that return a pointer to a memory area that can be used for any purpose

```
void* malloc(size_t size);
```

A typical use of `malloc` is

```
int* p = (int*) malloc(sizeof(int));
```

Trying to dereference or increment a `void` pointer are errors that will be caught by the compiler

Arrays

163

```
char a[2][3];
```

Creates a two dimensional array of characters

What is the value of a?

What is the address of a?

How is the data stored?

What is the relationship between arrays and pointers?

Can they be converted?

Experimenting...

164

```
char a[2][3];
```

```
printf("%p\n", a); 0x7fff682ba976
printf("%p\n", &a); 0x7fff682ba976
printf("%p\n", &a[0]); 0x7fff682ba976
printf("%p\n", &a[0][0]); 0x7fff682ba976
printf("%p\n", &a[0][1]); 0x7fff682ba977
printf("%p\n", &a[0][2]); 0x7fff682ba978
printf("%p\n", &a[1][0]); 0x7fff682ba979
printf("%p\n", &a[1][1]); 0x7fff682ba97a
```

Arrays

165

```
char a[2][3];
```

An array variable's value is the address of the array's first element

A multi-dimensional array is stored in memory as a single array of the base type with all rows occurring consecutively

There is no padding or delimiters between rows

All rows are of the same size

Pointers and arrays

166

There is a strong relationship between pointers and arrays

```
int a[10];  
int* p;
```

A pointer (e.g. p) holds an address while the name of an array (e.g. a) denotes an address

Thus it is possible to convert arrays to pointers

```
p = a;
```

Array operations have equivalent pointer operations

```
a[5] == *(p + 5)
```

Note that a=p or a++ are compile-time errors.

Pointers to arrays

167

```
char a[2][3];
```

Multi-dimensional array that stores two strings of 3 characters.
(Not necessarily zero-terminated)

```
char a[2][3]={"ah","oh"};
```

Array initialized with 2 zero-terminated strings.

```
char *p = &a[1];
```

```
while( *p != '\0' ) p++;
```

Iterate over the second string

Pointer to pointer

168

```
int i = 5;  
int *p = &i;  
int **pp = &p;
```

Think about it as `*pp` is an `int*`, that is, `p` is a pointer to pointer to int

```
char *s[3] = {"John", "Dan", "Christopher"};
```

`s` is a `char **`

```
char **p = s;
```

Apparté: Bit stealing

169

Many programming language make a heavy use of pointers
(from Lisp to Java most variables hold pointers)

Sometimes the compiler needs to mark objects without increasing their size

Nifty observation: many pointers have to be word-aligned

ptr	bits
... 0	... 0000
... 4	... 0100
... 8	... 1000
.. 12	... 1100

32-bit Words	64-bit Words	Bytes	Addr.
Addr = 0000			0000
Addr = 0004			0001
Addr = 0008			0002
	Addr = 0000		0003
			0004
			0005
			0006
			0007
			0008
			0009
			0010
			0011
			0012

Apparté: Bit stealing

170

Bit stealing refers to the use of the last two bits of a pointer to hold unrelated information

Assume p is a pointer with extra information.

Reading the information can be done by:

```
unsigned twobits = ((unsigned)p) & 3
```

Setting, e.g., the second bit be done by:

```
((unsigned)p) | 2
```

Following the pointer requires masking off the last two bits

```
* (int*) (((unsigned)p)&0xFFFFFFFFC)
```

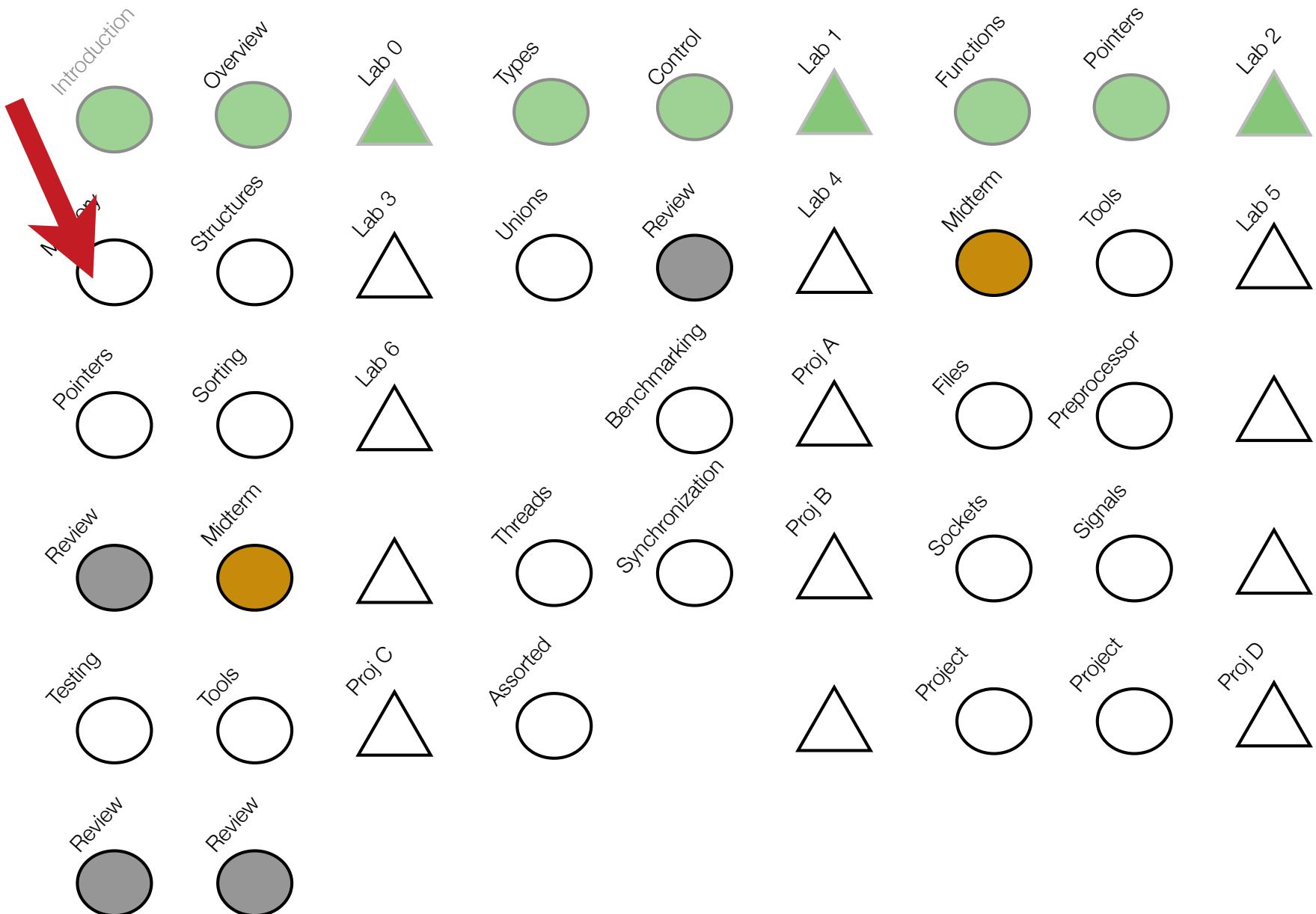
(on 32 bits arch)

{C'}

Lecture 7

Dynamic Memory Allocation





<http://tinyurl.com/4yvv79a>

Q 4

Quiz #4 - 0

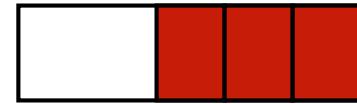


174

Did you read K&R Ch 5.4, Ch 8.7?

- (a) yes
- (b) no
- (c) architecture independent

Quiz #4 - 1



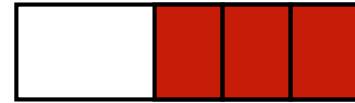
175

The sample allocator described in the text, called `alloc()`, is declared to return a `char*`.

What value does it return when it is out of memory?

- (a) 0
- (b) NULL
- (c) EOF

Quiz #4 - 2



176

The book claims that `size_t`, the type used for size by library routines, is

- (a) char
- (b) unsigned int
- (c) signed int
- (d) long

Quiz #4 - 3



177

The malloc implementation of Ch 8.7 uses a “first fit” strategy, what does that mean?

- (a) a request to malloc returns the first block of the right type (i.e. int, float, char)
- (b) a request to malloc returns the first free block that has the exact requested size
- (c) a request to malloc returns the first free block that is larger or equal to the requested size

Quiz #4 - 4



178

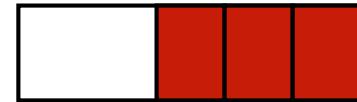
Free blocks have the following definition

```
union header {  
    struct {  
        union header *p; unsigned s;  
    }d;  
    long align;  
}
```

When is the align field used?

- (a) never
- (b) always
- (c) on certain architectures

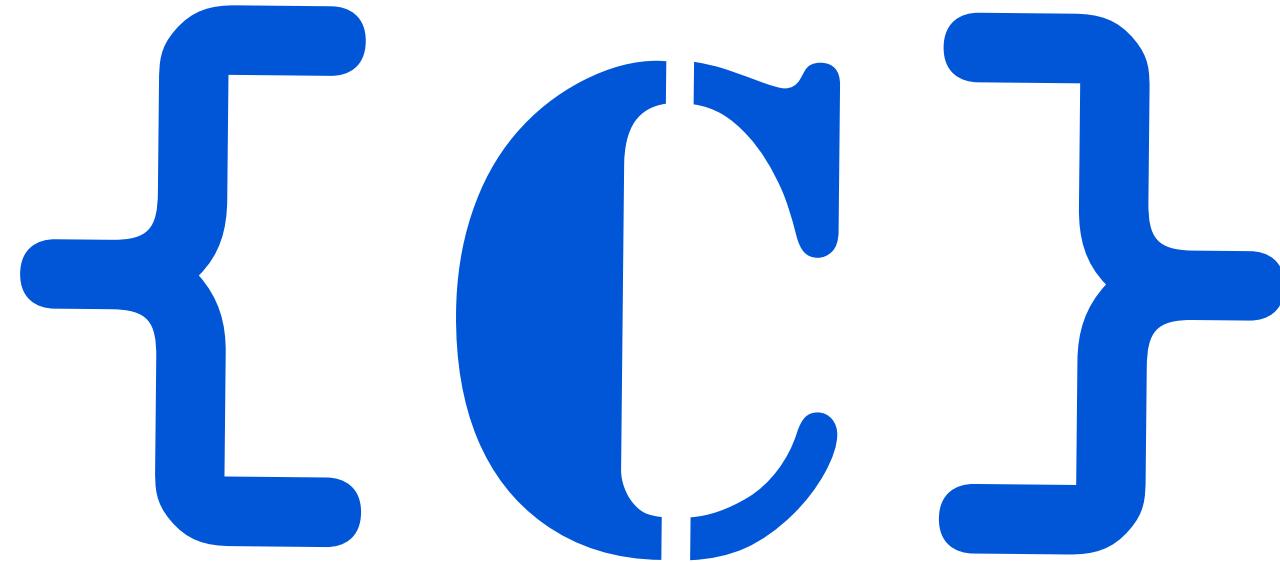
Quiz #4 - 5



179

sbrk() is used in the implementation in order to:

- (a) request memory from the operating system
- (b) set a memory break point for debugging
- (c) signal an out of memory condition

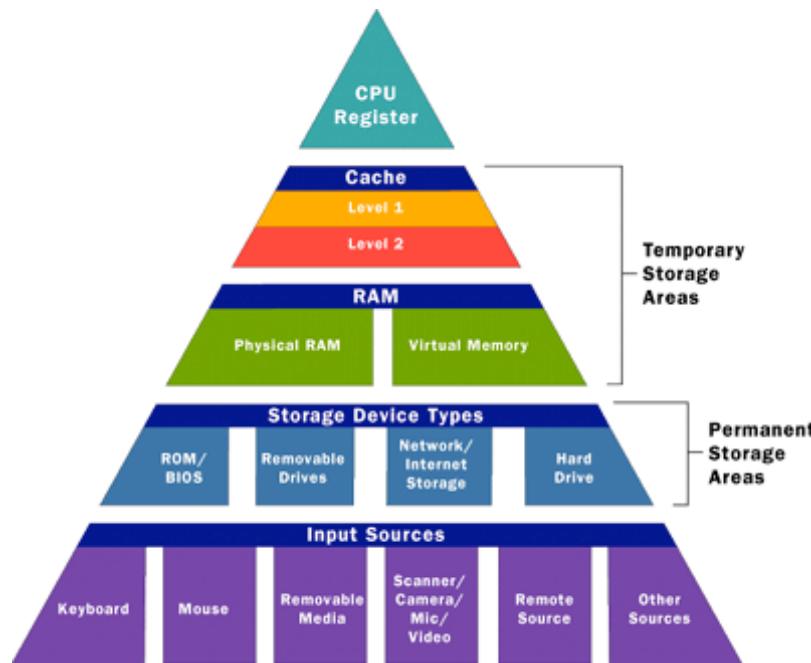
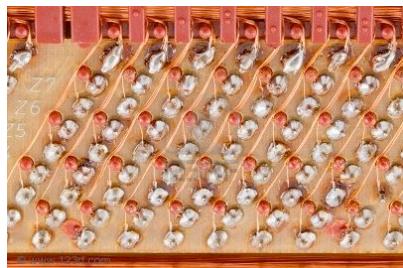


Memory

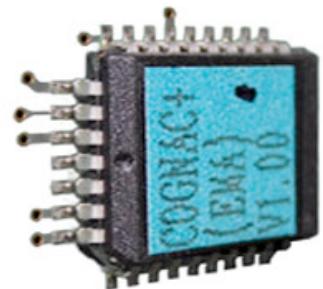
Computer programs manipulate an abstraction of the computer's memory subsystem

Memory: on the hardware side

181



© 2005 HowStuffWorks



Memory: on the software side

182

Each computer programming languages offers a *different abstraction*

The goal is to make programming easier and improve portability of the source code by hiding irrelevant hardware oddities

Each language offers a memory API – a set of operations for manipulating memory

► Sample exam question:

How does the abstraction of memory exposed by the Java programming language differ from that of the C programming language?

Memory: the Java Story

183

Memory is a set of objects with fields, methods and a class + local variables of a method

Memory is read by accessing a field or local variable

Memory is modified by writing to a field or local variable

Location and size of data are not exposed

Memory allocation is done by call in new

► Question:

Does main() terminate?

```
public class Main {  
    static public  
    void main(String[] a){  
        Cell c1, c2 = null;  
        while (true) {  
            c1 = new Cell();  
  
            c2 = c1;  
        }  
    }  
  
    class Cell {Cell next; }  
}
```

Memory: the Java Story

184

Memory is a set of objects with fields, methods and a class + local variables of a method

Memory is read by accessing a field or local variable

Memory is modified by writing to a field or local variable

Location and size of data are not exposed

Memory allocation is done by call in **new**

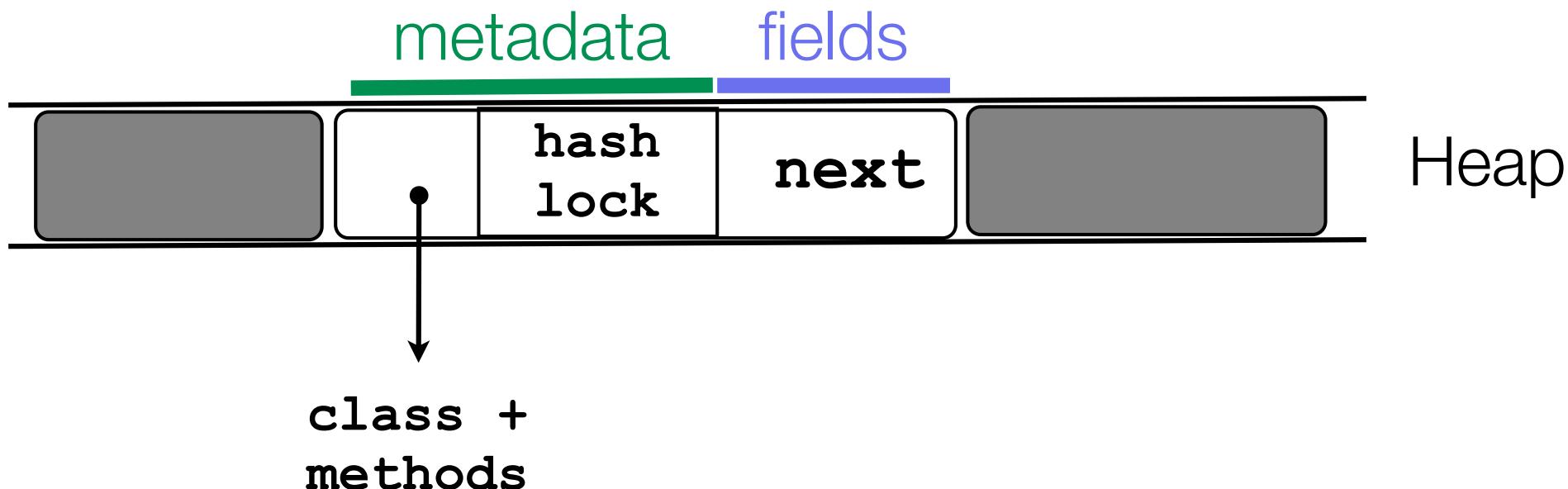
```
public class Main {  
    static public  
    void main(String[] a){  
        Cell c1, c2 = null;  
        while (true) {  
            c1 = new Cell();  
            c1.next = c2;  
            c2 = c1;  
        }  
    }  
}  
  
class Cell {Cell next; }
```

Memory: the Java Story

185

The semantics of `new` is as follows:

- ▶ Allocate space for the object's fields and metadata fields
- ▶ Initialize the metadata fields
- ▶ Set all fields to null/zero/false



Garbage collection is the technology that gives the illusion of infinite resources

Garbage collection or GC is implemented by the programming language with the help of the compiler

- ▶ Though for some well-behaved C programs it is possible to link a special library that provides most of the benefits of GC

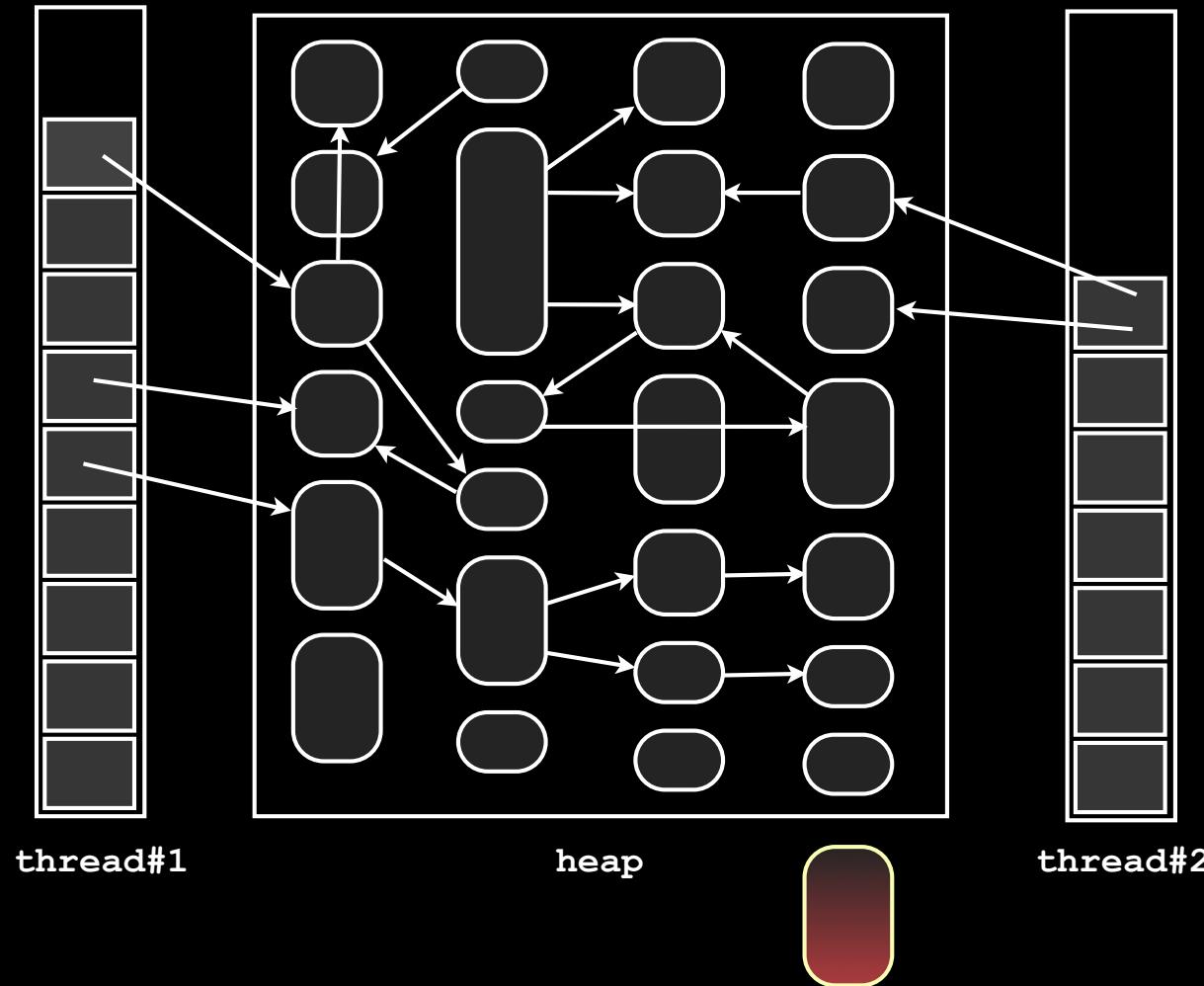
▶ **Question:**

How does GC work?

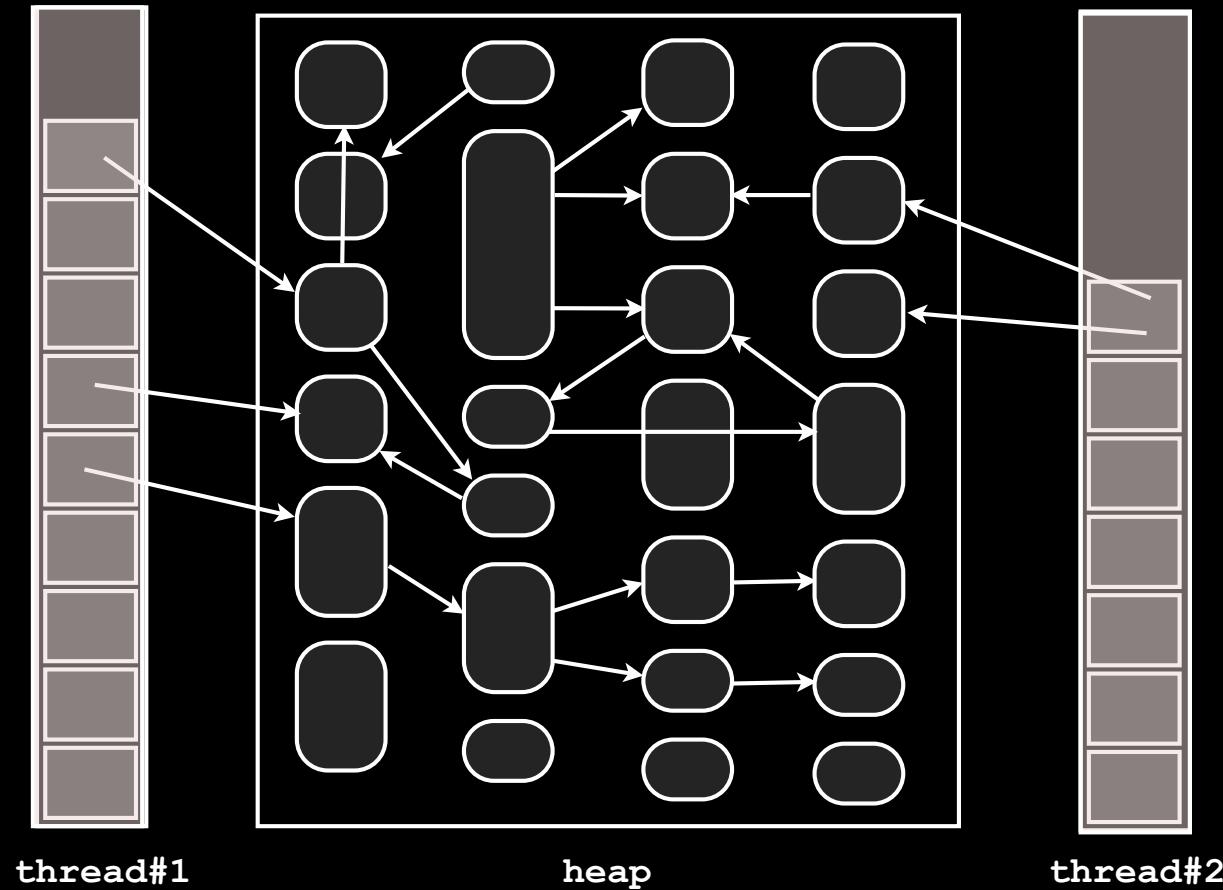
Garbage Collection

Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction



Garbage Collection



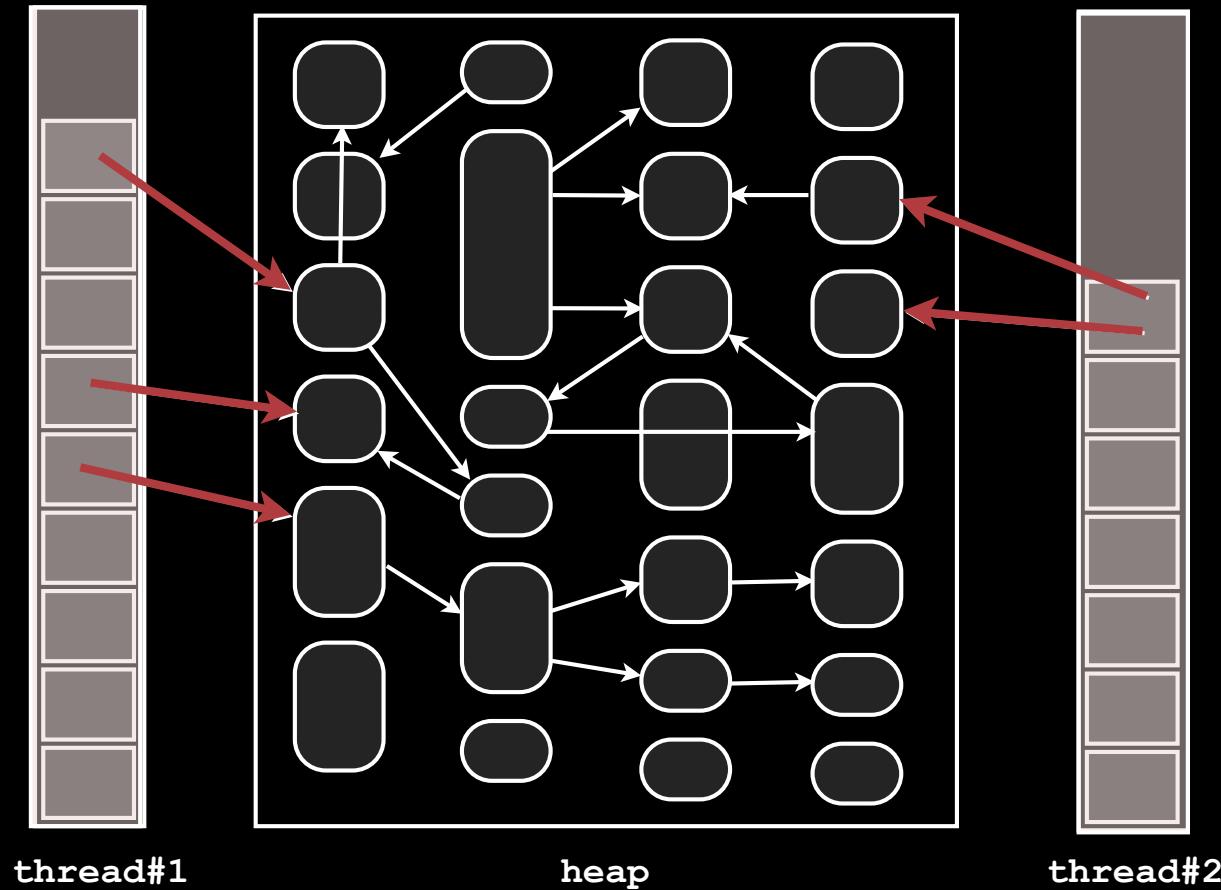
Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

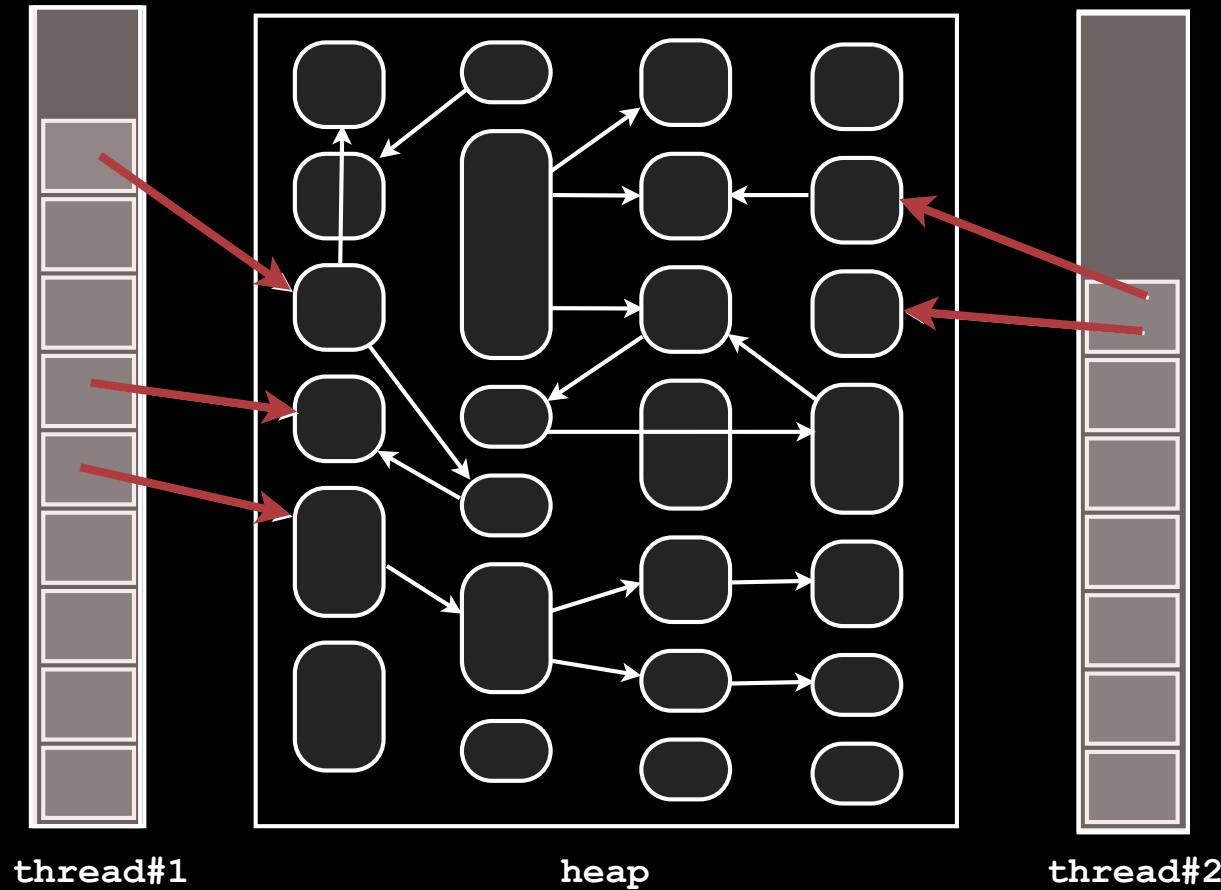
Garbage Collection

Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction



Garbage Collection



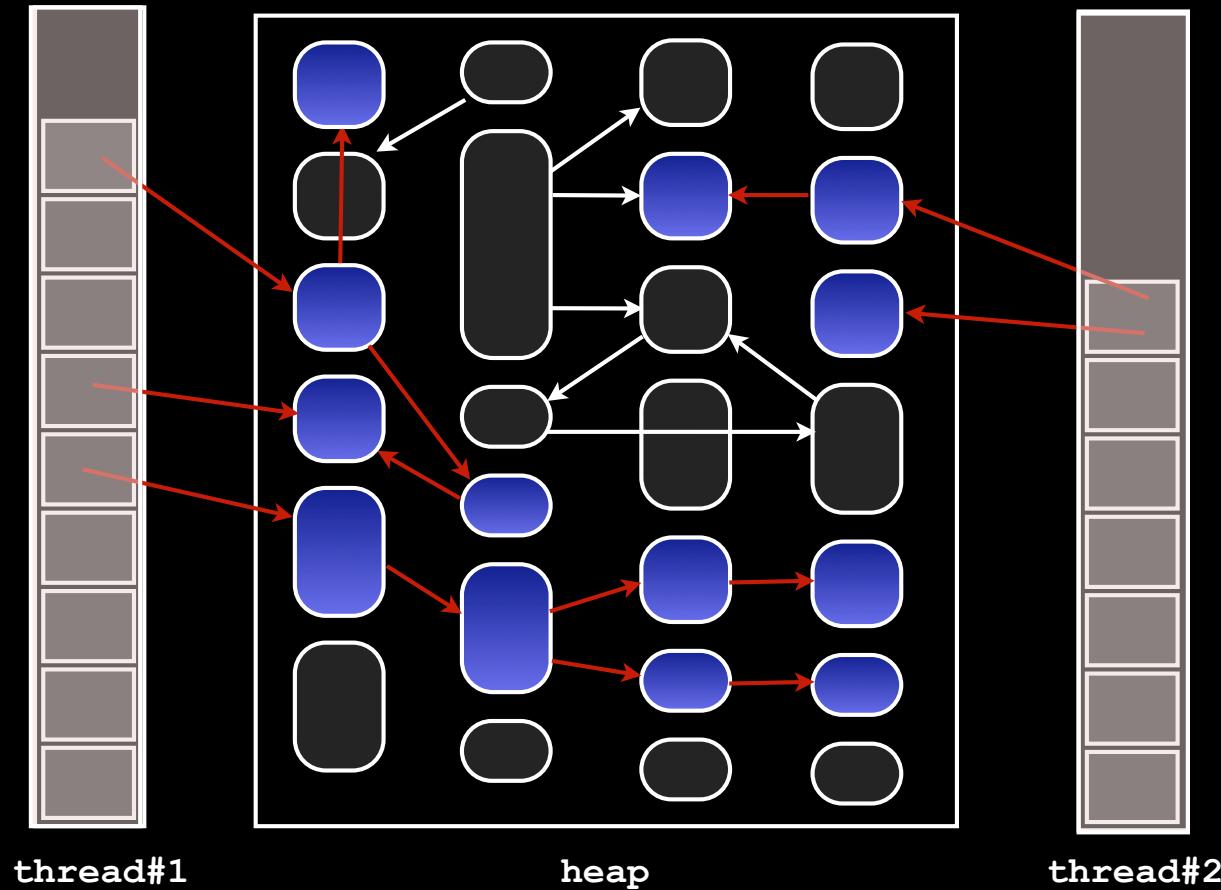
Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

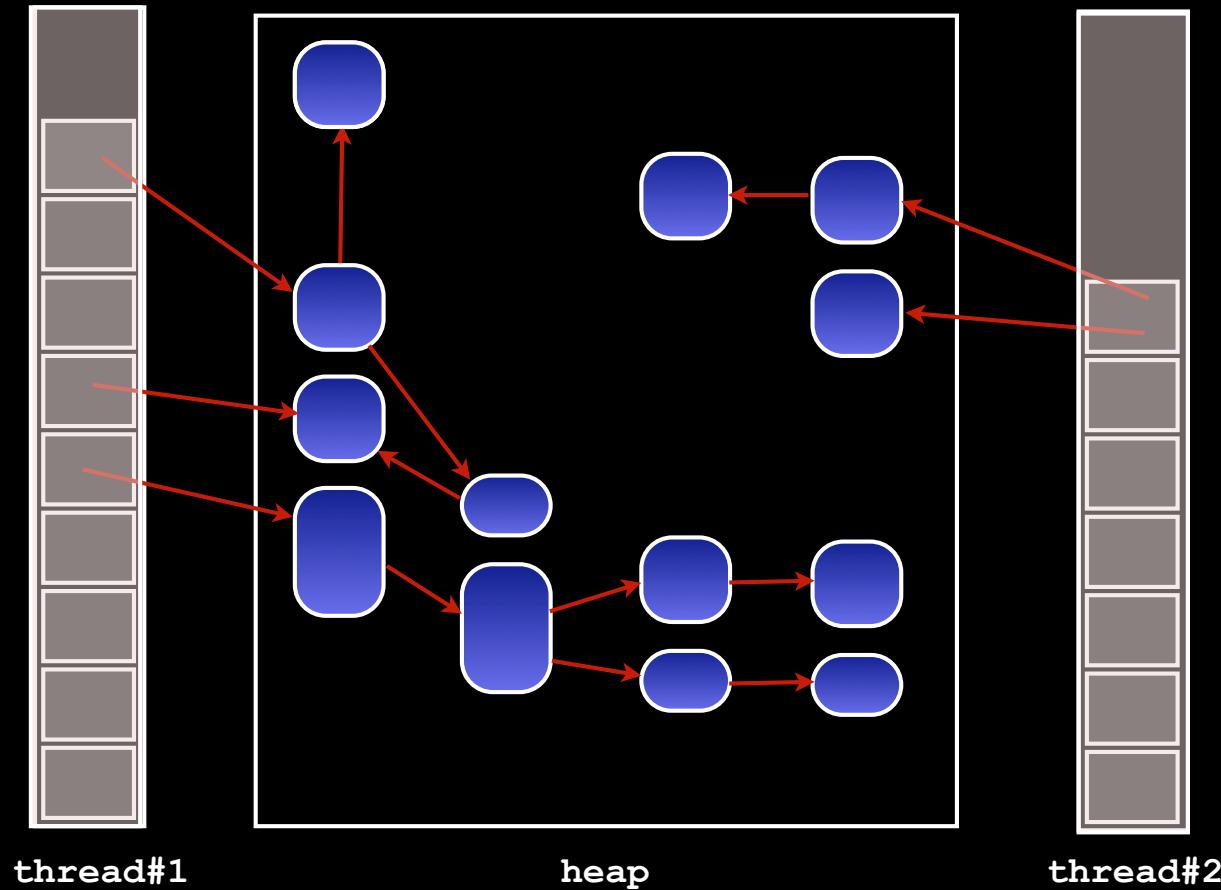
Garbage Collection

Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction



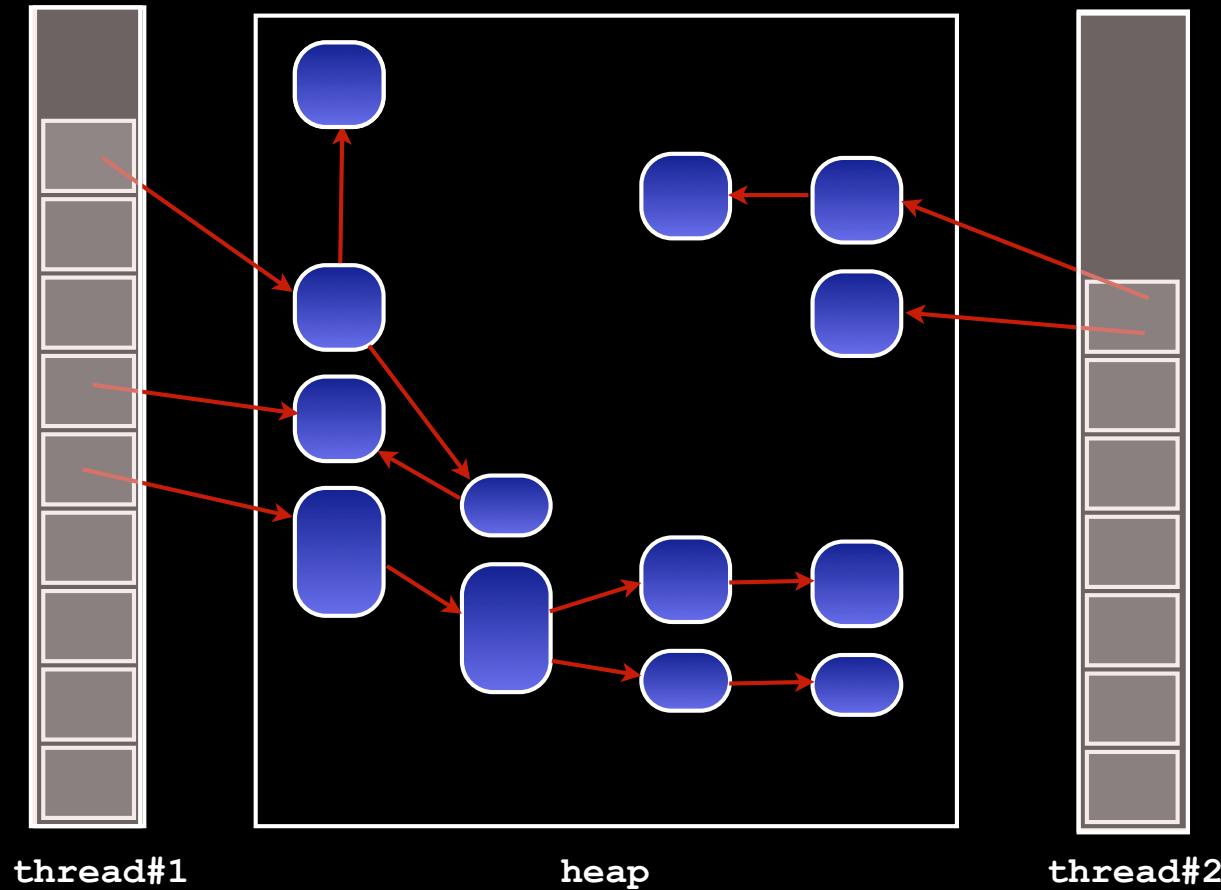
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection



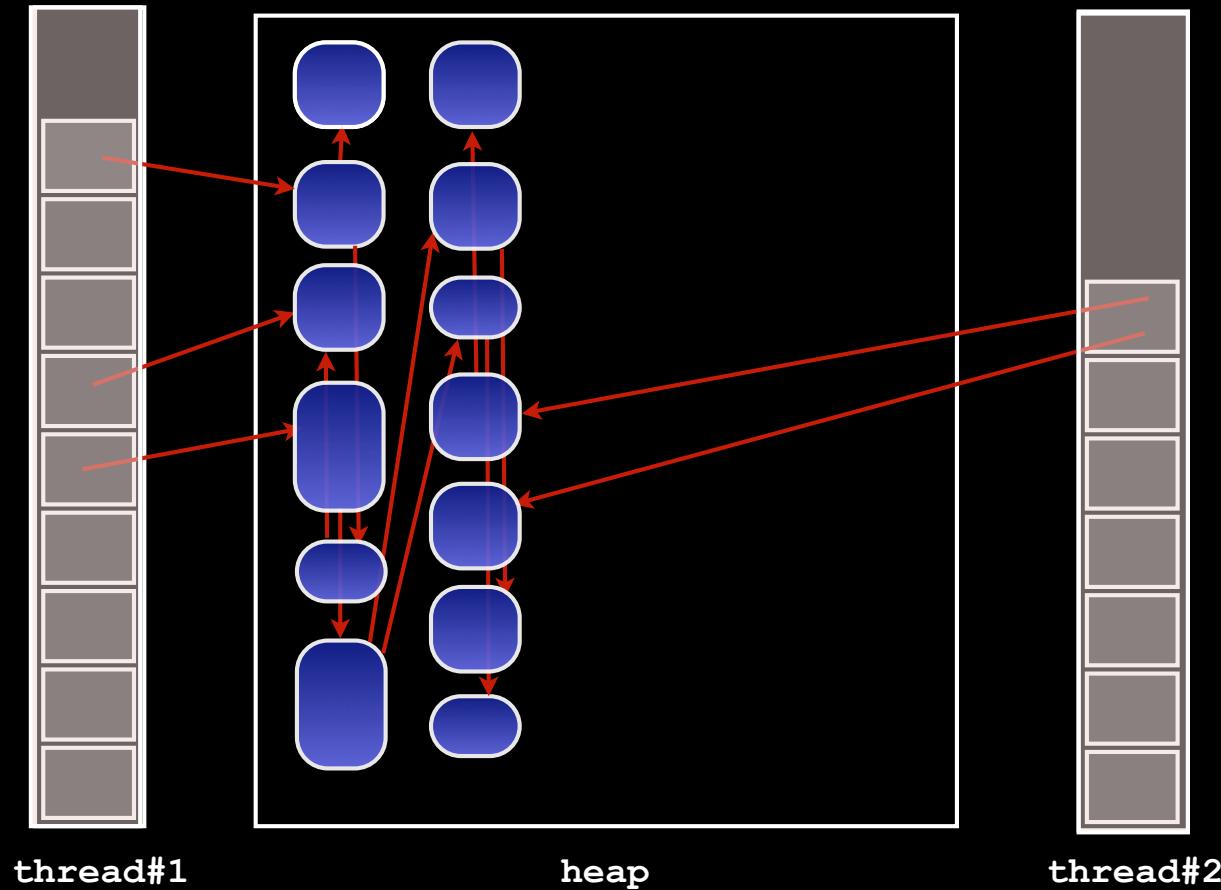
Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection

Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction



{C'}

Isn't this a course about C?

Yes, Virginia

Memory: the C Story

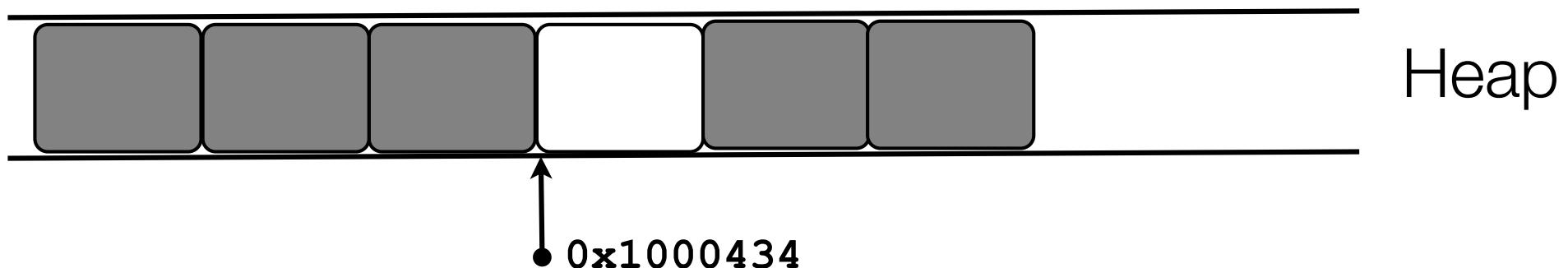
196

C offers a story both simpler and more complex than Java
Memory is a sequence of bytes, read/written by providing an address

Addresses are values manipulated using arithmetic & logic operations

Memory can be allocated:

- ▶ Statically
- ▶ Dynamically on the stack



Static and Stack allocation

197

Static allocation
with the
keyword `static`

Stack allocation
automatic by the
compiler for
local variables

`printf` can
display the
address of any
identifier

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

static int sx;
static int sa[100];
static int sy;

int main() {
    int lx;
    static int sz;

    printf("%p\n", &sx);      0x100001084
    printf("%p\n", &sa);      0x1000010a0
    printf("%p\n", &sy);      0x100001230
    printf("%p\n", &lx);      0x7fff5fbff58c
    printf("%p\n", &sz);      0x100001080
    printf("%p\n", &main);    0x100000dfc
```

Static and Stack allocation

198

Any value can
be turned into
a pointer

Arithmetics on
pointers
allowed

Nothing
prevents a
program from
writing all
over memory

```
static int sx;
static int sa[100];
static int sy;

int main() {
    for(p= (int*)0x100001084;
        p <= (int*)0x100001230;
        p++)
    {
        *p = 42;
    }
    printf("%i\n",sx);          42
    printf("%i\n",sa[0]);       42
    printf("%i\n",sa[1]);       42
```

Memory layout

199

The OS creates a process by assigning memory and other resources

C exposes the layout as the programmer can take the address of any element (with &)

Stack:

- ▶ keeps track of where each active subroutine should return control when it finishes executing; stores local variables

Heap:

- ▶ dynamic memory for variables that are created with `malloc`, `calloc`, `realloc` and disposed of with `free`

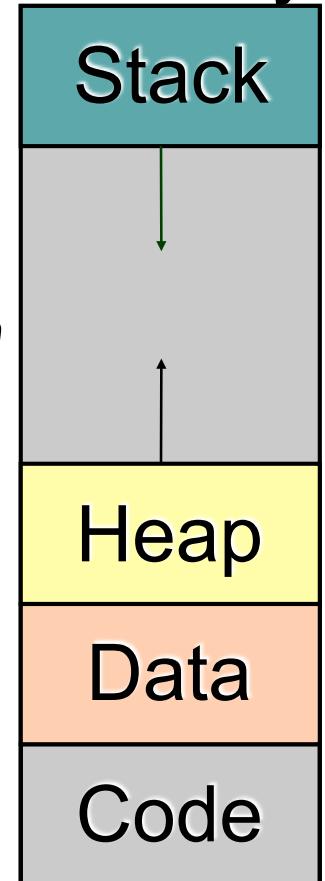
Data:

- ▶ global and static variables

Code:

- ▶ instructions to be executed

Virtual
Memory



Dynamic Memory: The DIY Way

200

A simple dynamic allocation pattern is to ask the OS for a chunk of memory large enough to store all data needed

sbrk(size) returns a chunk of memory of **size** bytes

The downside is that the programmer must keep track of how memory is used

```
int main() {
    int* x; int* start;
    double* y;
    start = (int*) sbrk(5*sizeof(int));
    x = start;
    *x = -42; x++; y=(double*) x;
    *y = 2.1; y++; x=(int*) y;
    *x = 42;
    printf("%i\n", *start); -42
    printf("%i\n", start[0]); -42
    printf("%i\n", start[1]); -858993459
    printf("%i\n", start[2]); 1073794252
    printf("%i\n", start[3]); 42
    printf("%i\n", start[4]); 0
    printf("%i\n", start[5]); 0
    printf("%f\n",
           *(double*)(start+1)); 2.100000
```

Dynamic memory management

201

```
#include <stdlib.h>

void* calloc(size_t n, size_t s)
void* malloc(size_t s)
void free(void* p)
void* realloc(void* p, size_t s)
```

Allocate and free dynamic memory

Operations with memory

202

```
#include <string.h>
```

```
void* memset(void *s, int c, size_t n)
void* memcpy(void *s, const void *s2, size_t n)
```

Initializing and copying blocks of memory

malloc(size_t s)

203

Allocates **s** bytes and returns a pointer to the allocated memory.

Memory is not cleared

Returned value is a pointer to alloc'd memory or **NULL** if the request fails

You must cast the pointer

```
p = (char*) malloc(10); /* allocated 10 bytes */  
if(p == NULL) { /*panic*/ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

calloc(size_t n, size_t s)

204

Allocates memory for an array of **n** elements of **s** bytes each and returns a pointer to the allocated memory.

The memory is set to zero

The value returned is a pointer to the allocated memory or **NULL**

```
p = (char*) calloc(10,1); /*alloc 10 bytes */  
if(p == NULL) { /* panic */ }
```

CAN FAIL, CHECK THE RETURNED POINTER NOT NULL

What's the difference between **int array[10]** and **calloc(10, 4)**

free (void* p)

205

Frees the memory space pointed to by **p**, which *must* have been allocated with a previous call to **malloc**, **calloc** or **realloc**.

If memory was not allocated before, or if **free(p)** has already been called before, undefined behavior occurs.

If **p** is **NULL**, no operation is performed.

free() returns nothing

```
char *mess = NULL;  
mess = (char*) malloc(100);  
...  
free(mess);    *mess = 43;
```

FREE DOES NOT SET THE POINTER TO NULL

free (void* p)

206

Frees the memory space pointed to by **p**, which *must* have been allocated with a previous call to **malloc**, **calloc** or **realloc**

If memory was not allocated before, or if **free(p)** has already been called before, undefined behavior occurs.

```
char *mess = NULL;  
mess = (char*) malloc(100);  
...  
free(&mess);  
mint= (int*) malloc(100);  
*mint = 42;  
*mess = 'a';
```

`realloc(void* p, size_t s)`

207

Changes the size of the memory block pointed to by `p` to `s` bytes

Contents unchanged to the minimum of old and new sizes

Newly alloc'd memory is uninitialized.

Unless `p==NULL`, it must come from `malloc`, `calloc` or `realloc`.

If `p==NULL`, equivalent to `malloc(size)`

If `s==0`, equivalent to `free(ptr)`

Returns pointer to alloc'd memory, may be different from `p`, or
NULL if the request fails or if `s==0`

If fails, original block left untouched, i.e. it is not freed or moved

`memcpy (void*dest,const void*src,size_t n)`

208

Copies **n** bytes from **src** to **dest**

Returns **dest**

Does not check for overflow on copy

```
char buf[100];
char src[20] = "Hi there!";
int type = 9;
memcpy(buf, &type, sizeof(int)); /* copy an int */
memcpy(buf+sizeof(int), src, 10); /*copy 10 chars */
```

memset(void *s, int c, size_t n)

209

Sets the first **n** bytes in **s** to the value of **c**

► (**c** is converted to an **unsigned char**)

Returns **s**

Does not check for overflow

```
memset(mess, 0, 100);
```

Sizeof matters

210

In C, programmers must know the size of data structures

The compiler provides a way to determine the size of data

```
struct {  
    int i; char c; float cv;  
} C;
```

int x[10];	
printf("%i\n", (int) sizeof(char));	1
printf("%i\n", (int) sizeof(int));	4
printf("%i\n", (int) sizeof(int*));	8
printf("%i\n", (int) sizeof(double));	8
printf("%i\n", (int) sizeof(double*));	8
printf("%i\n", (int) sizeof(x));	40
printf("%i\n", (int) sizeof(C));	12

Sizeof matters

211

In C, programmers must know the size of data structures

The compiler provides a way to determine the size of data

Do this:

```
int *p = (int*) malloc(10 * sizeof(*p));
```

Memory Allocation Problems

212

Memory leaks

- ▶ Alloc'd memory not freed appropriately
- ▶ If your program runs a long time, it will run out of memory or slow down the system
- ▶ Always add the free on all control flow paths after a malloc

```
void *ptr = malloc(size);
/*the buffer needs to double*/
size *= 2;
ptr = realloc(ptr, size);
if (ptr == NULL)
    /*realloc failed, original address in ptr
     lost; a leak has occurred*/
return 1;
```



Memory Allocation Problems

213

Use after free

- ▶ Using dealloc'd data
- ▶ Deallocating something twice
- ▶ Deallocating something that was not allocated

Can cause unexpected behavior. For example, malloc can fail if "dead" memory is not freed.

More insidiously, freeing a region that wasn't malloc'ed or freeing a region that is still being referenced



```
int *ptr = malloc(sizeof (int));  
free(ptr);  
*ptr = 7; /* Undefined behavior */
```

Memory Allocation Problems

214

Memory overrun

- ▶ Write in memory that was not allocated
- ▶ The program will exit with segmentation fault
- ▶ Overwrite memory: unexpected behavior



```
int*y= ...
int*x= y+10
for(p= x; p >= y;p++)
{
    *p = 42;
}
```

Memory Allocation Problems

215

Fragmentation

- ▶ The system may have enough memory but not in contiguous region

```
int* vals[10000];
```

```
int i;
for (i = 0; i < 10000; i++)
    vals[i] = (int*) malloc(sizeof(int*));
```

```
for (i = 0; i < 10000; i = i + 2)
    free(vals[i]);
```

A (simple) malloc

216

```
#define SIZE 10000
#define UNUSED -1

struct Cell { int sz; void* value; struct Cell* next; };

static struct Cell*free, *used;
static struct Cell cells[SIZE / 10];

void init() {
    void* heap = sbrk(SIZE);
    for(int i=0;i<SIZE/10;i++){cells[i].sz=UNUSED; cells[i].next=NULL;}
    cells[0].sz = 0;
    free = &cells[0];
    free->next = &cells[1];
    free->next->sz = SIZE;
    free->next->value = heap;
    free->next->next = NULL;
    used = &cells[1];
    used->sz = 0;
    used->value = (void*) UNUSED;
}
```

A (simple) malloc

217

```
void* mymalloc(int size) {
    struct Cell* tmp = free, *prev = NULL;
    if (size == 0) return NULL;
    while (tmp != NULL) {
        if (tmp->sz == size) {
            prev->next = tmp->next; tmp->next = used; used = tmp;
            return used->value;
        } else if (tmp->sz > size) {
            struct Cell* use = NULL;
            for (int i = 0; i < SIZE / 10; i++)
                if (cells[i].sz==UNUSED) { use=&cells[i]; use->sz=size; }
            if (use == NULL) return NULL;
            use->next = used; use->value = tmp->value;
            tmp->sz -= size; tmp->value += size;
            return used->value;
        }
        prev = tmp; tmp = tmp->next;
    }
    return NULL;
}
```

A (simple) malloc

218

```
void myfree(void* p) {  
    struct Cell *tmp = used, *prev = NULL;  
    while (tmp != NULL) {  
        if (tmp->value == p) {  
            prev->next = tmp-> next;  
            tmp->next = free;  
            free = tmp;  
            free->sz = UNUSED;  
            return;  
        }  
        prev = tmp;  
        tmp = tmp->next;  
    }  
}
```

Checklist

219

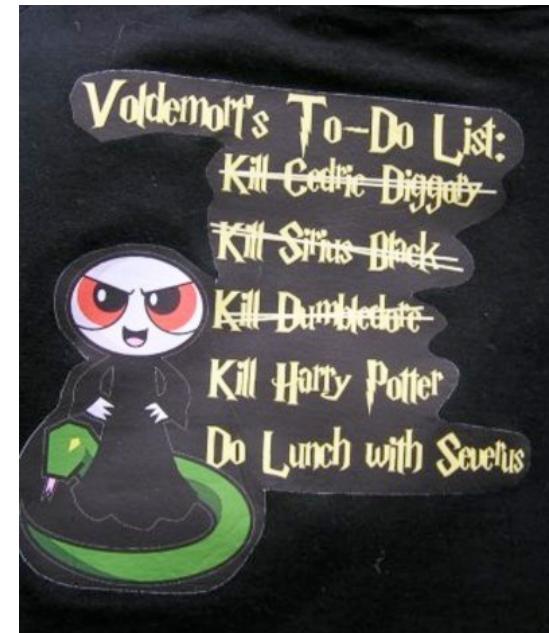
NULL pointer at declaration

Verify `malloc` succeeded

Initialize alloc'd memory

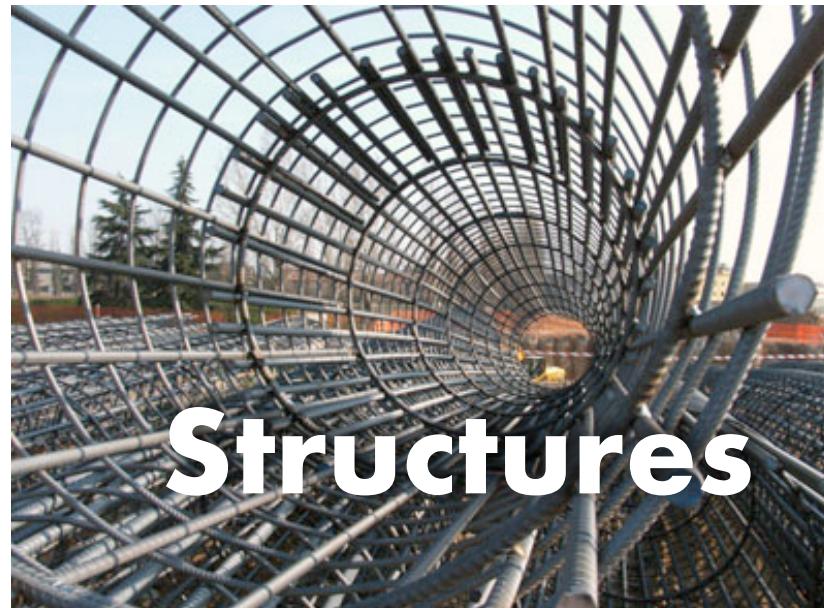
free when you `malloc`

NULL pointer after free

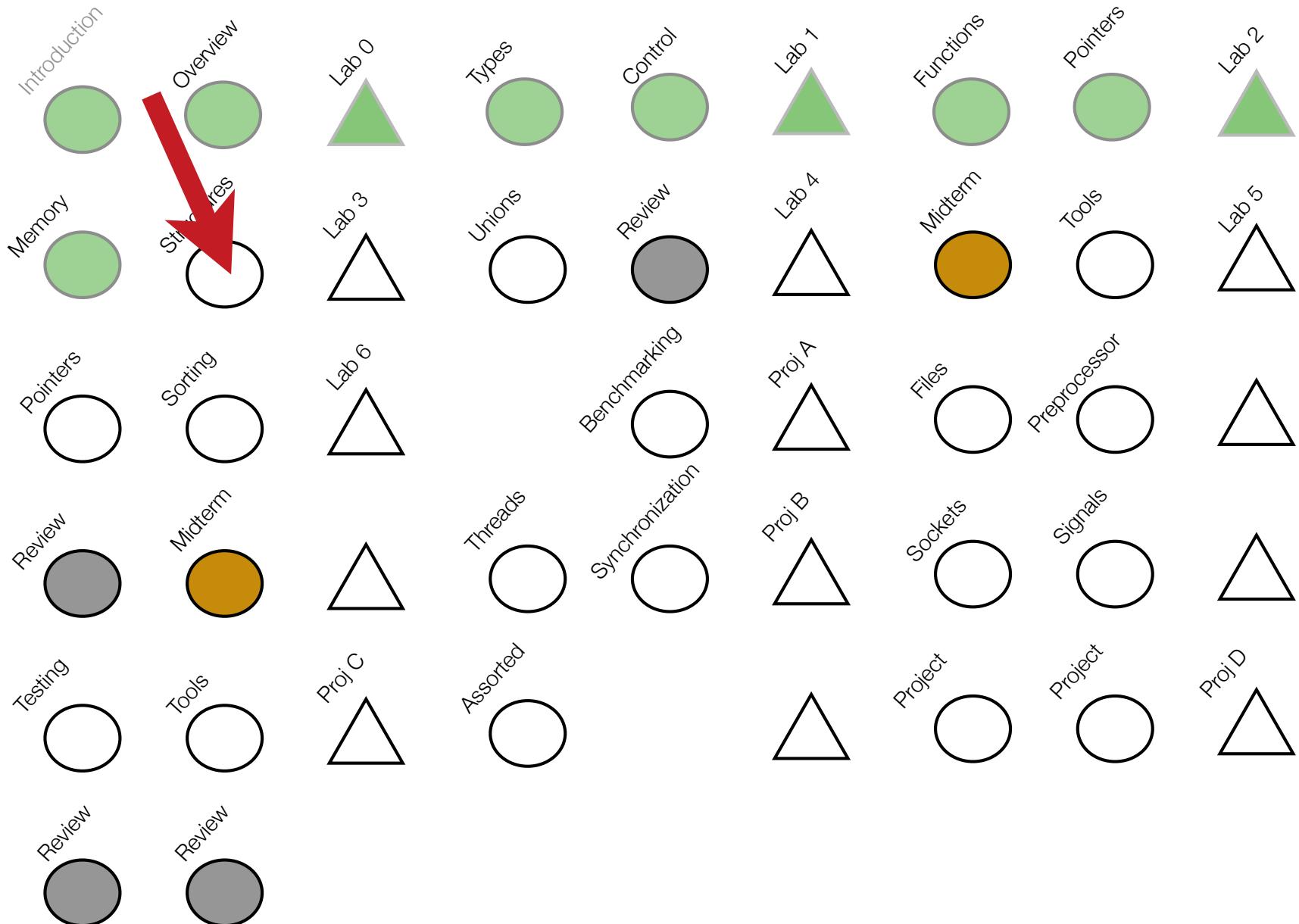


Lecture 8

Lecture 8



Structures



<http://tinyurl.com/4yvv79a>

Q 5

Quiz #4 - 0

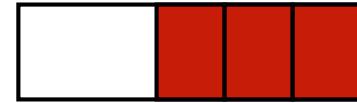


223

Did you read K&R Ch 6?

- (a) yes
- (b) no
- (c) architecture independent

Quiz #4 - 1



224

What is a “structure tag” in the following definition?

```
struct c { int a; char b; } d;
```

- (a) a
- (b) b
- (c) c
- (d) d
- (e) all

Quiz #4 - 2



225

How much space would you expect the following declarations would require (an int being 4 bytes)

struct point { int x; };

struct point x,y,z;

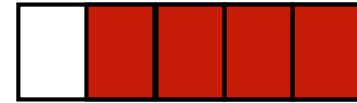
(a) 12 bytes

(b) 16 bytes

(c) 24 bytes

(d) more

Quiz #4 - 3



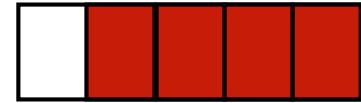
226

Is the following code correct or should it return a pointer to tmp?

```
struct point makepoint(int x,int y) {  
    struct point temp;  
    temp.x = x; temp.y = y; return temp;  
}
```

- (a) yes, it is correct
- (b) no, it should return a pointer
- (c) architecturally dependent

Quiz #4 - 4

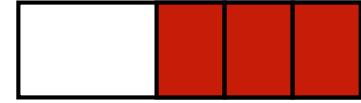


227

Is there a limit to the size of a structure which can be passed or returned to a function

- (a) yes, it can't have more fields than there are registers
- (b) no, none whatsoever
- (c) only if you pass the structure by pointer

Quiz #4 - 5



228

Is the following definition of a tree correct?

```
struct tnode {  
    char* word;  
    struct tnode left, right;  
}
```

- (a) yes, obviously
- (b) no, are you kidding
- (c) unclear at this hour

Structures

229

In C

- ▶ functions organize sequence of instructions into logical units
- ▶ structures groups variables in logical units

A C struct is a named collection of one or more variables, possibly of different types

```
struct slot {  
    int x;  
    char c;  
}
```

slot is the name (tag) of the structure; x and c are members

Comparison with Java

230

```
class Slot {  
    int x;  
    char c;  
}
```

Java

```
struct slot {  
    int x;  
    char c;  
}
```

C

Difference between the two:

- ▶ No inheritance
- ▶ No methods
- ▶ A Java variable of type Slot is a pointer
- ▶ A C variable of type slot denotes the structure with no indirection

Structures

231

```
struct slot { int x; char c; }
```

Tag names can be used after a struct has been declared

```
struct slot s1, s2;
```

The size of a struct is obtained by calling `sizeof`

```
sizeof(s2)
```

Accessing a member is done with the dot operator

```
s1.x
```

Pointers to structures can be defined

```
struct slot* p = &s1;
```

Two equivalent syntactic ways to access members by reference

```
p->x
```

```
(*p).x
```

Size

232

If a structure contains dynamically allocated members, the size of whole struct may not equal sum of its (referenced) parts

```
struct word { char* w; int l; }
```

- ▶ `sizeof(struct word)` is 8 bytes. But this does not account for the space of string pointed to by `w`
- ▶ Internal padding means that `sizeof` may be larger than expected

```
struct ex { int a; char b; int c; };
```

- ▶ Is `sizeof(struct ex) == 2*sizeof(int)+sizeof(char) ?`

Structs in structs...

233

A structure can contain a member of another structure

```
struct pos { int x; int y; }

struct slot {
    struct pos p;
    char c;
}
```

Access x via: s.p.x

The size of slot is exactly the same as if the fields of pos were written inline in slot

In terms of performance there is no cost to nested structures

Recursive structures

234

What is the meaning of

```
struct rec { int i; struct rec r; }
```

A structure can not refer itself directly.

The only way to create a recursive structure is to use pointers

```
struct node {  
    char * word;  
    int count;  
    struct node *left, *right;  
}
```

Structures and functions

235

Structures can be initialized, copied as any other value

They can not be compared directly

- ▶ instead one must write code to compare members one by one
- ▶ Or compare the addresses of the structures (*usually not the right answer*)

Functions can return structure instances

- ▶ What is the cost in terms of memory allocation, copy, and performance?
- ▶ What's the difference between arrays and structures in this sense?

```
struct pt { int x, y; };

struct pt mkpt(int x, int y) {
    struct pt t; t.x = x; t.y = y; return t;
}

struct pt p1 = mkpt(0, 0);
```

Typedef

236

Allows us to create new data name types;

```
typedef int len;
```

```
len l1, l2;
```

```
typedef struct { len x, y; } pos;
```

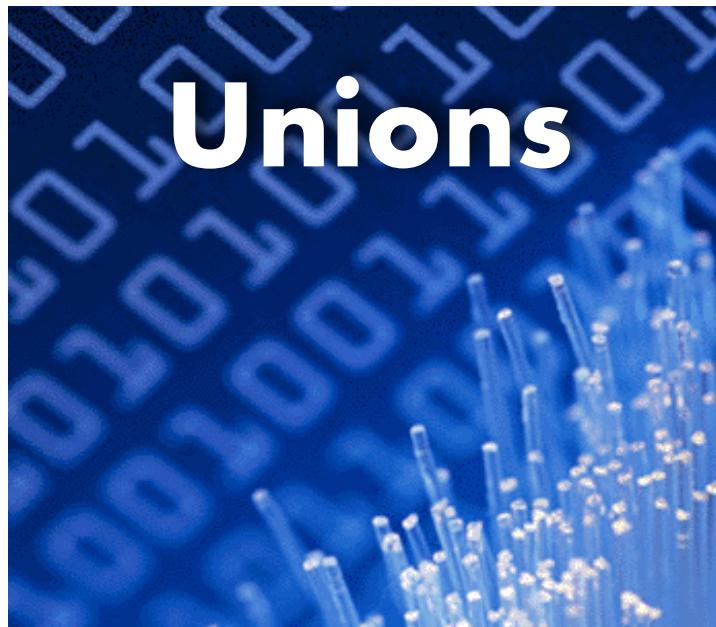
```
pos p1, p2;
```

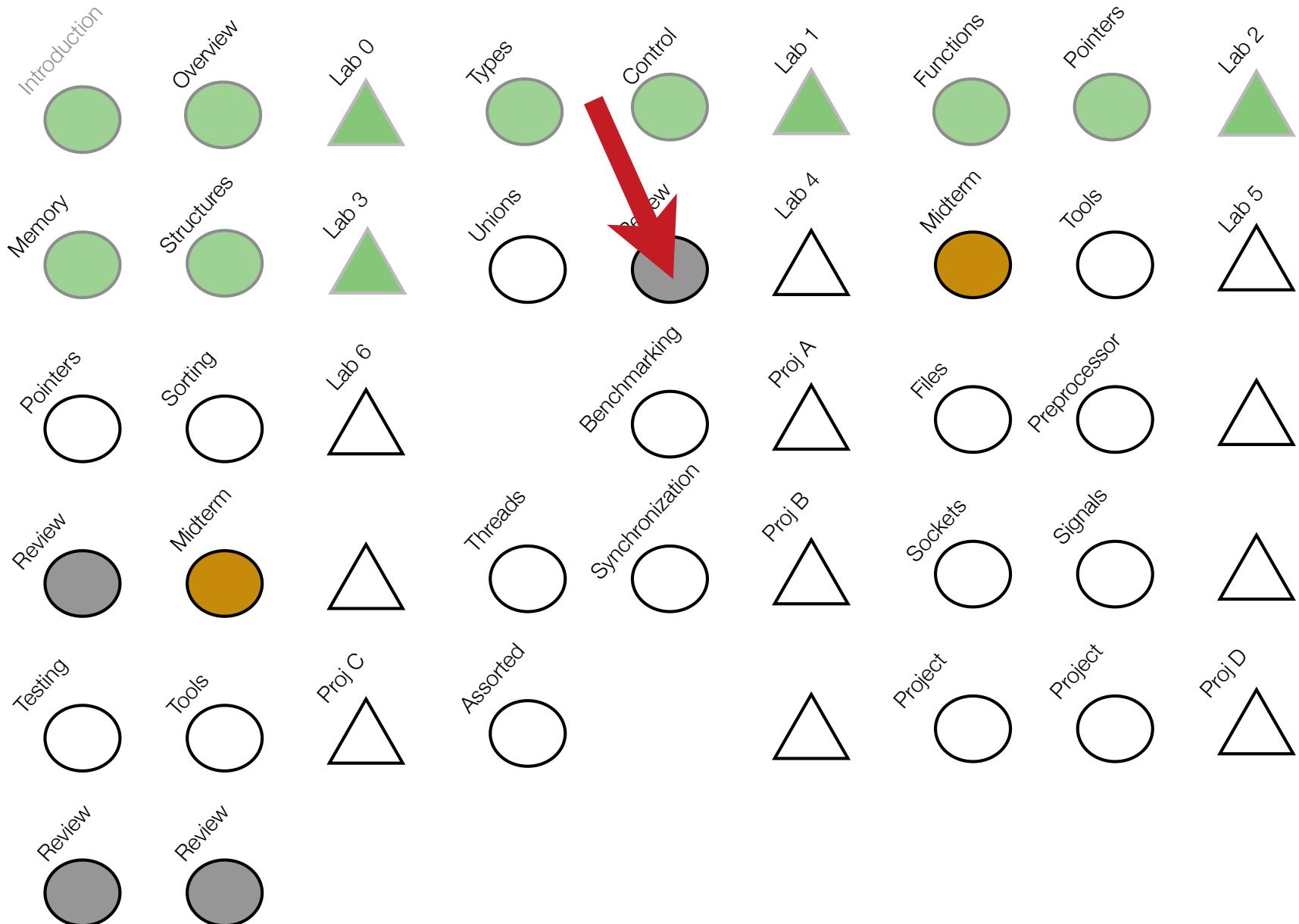
Notice the difference. No struct needed when using the type.

Use **typedef** to define pointer types and function types

{Loc}

Lecture 9





<http://tinyurl.com/4yvv79a>

Q 6

Quiz #4 - 0

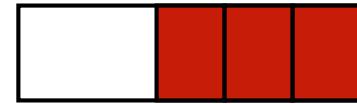


240

Did you read K&R Ch 6.8 and 6.9?

- (a) yes
- (b) no
- (c) architecture independent

Quiz #4 - 1



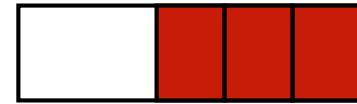
241

What is a union variable in the following definition?

```
union u { int ival; float fval; } ut;
```

- (a) u
- (b) ival
- (c) fval
- (d) ut
- (e) all

Quiz #4 - 2



242

How much space would you expect the following declarations would require (an int being 4 bytes)

```
union point { int x; char c };
```

```
union point u,v;
```

(a) 2 bytes

(b) 8 bytes

(c) 10 bytes

(d) more

Quiz #4 - 3



243

What does the following code do?

```
union u { int i; float f; char c; } v;  
v.c = 'a';  
printf("%d\n", v.i);
```

- (a) the compiler will correctly note the type error in the printf
- (b) the program will compile, but will throw an exception trying print a char as a number
- (c) the program will compile and print something

Quiz #4 - 4



244

What is the size of the variable (32 bit ints)

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

- (a) exactly 12 bytes
- (b) >12 bytes
- (c) <=12 bytes

Structures

245

Recap:

- ▶ Holds multiple items as a unit
- ▶ Treated as scalar in C: can be returned from functions, passed to functions
- ▶ Can not be compared
- ▶ A structure can include:
 - a pointer to itself, but not a member of the same structure;
 - a member of another structure, the latter has to have the prototype declared before
- ▶ Member access
 - (direct) `s.member` and (indirect) `s_ptr->member`
 - Dot operator `.` has precedence over indirect access operator `->`
- What do the following mean?
 - `s.t->u`
 - `(s.t)->u`
 - `s.(t->u)`
 - `(*s).t`
 - `s->t`

Memory layout

246

Data alignment: when the processor accesses the memory reads more than one byte, usually 4 bytes on a 32-bit platform

What if the data structure is not a multiple of 4?

- ▶ Padding: some unused bytes are inserted in the structure by the compiler
- ▶ Or more complex access patterns (must read multiple words)

Structures and functions

247

What happens when a structures is passed as an argument

```
struct Fs { int a; };

typedef struct Fs F;

F doIt(F b) {
    b.a = 13; return b;
}

int main () {
    F f = { 100 };    f = doIt(f);
    printf("%d\n", f.a);
}
```

Structures and functions

248

Alternatively

```
struct Fs { int a; };

typedef struct Fs F

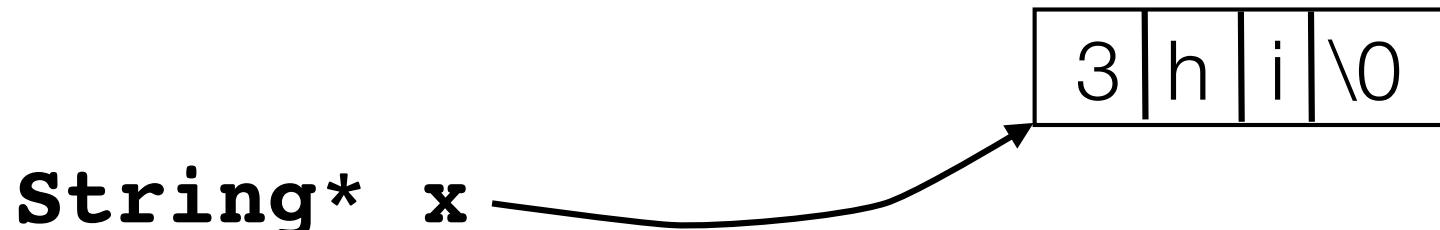
void doIt(F* b) { b->a = 13; }

int main () {
    F f = { 100 };    doIt(&f);
    printf("%d\n", f.a);
}
```

Advanced: Flexible structures

249

A safer string data type for C would include a size and thus avoid buffer overflows



Would this definition work?

```
struct str { unsigned int len; char* val }  
typedef struct str String;
```

Advanced: Flexible structures

250

C99 allows to write the following

```
struct str { unsigned int len; char val[]; }  
typedef struct str String;
```

this means create a structure with a single field integer field, but support access array indexed access outside of the struct

```
String* s =  
(String*)malloc(sizeof(String)+  
               sizeof(char)*3);  
  
s->len = 3;  
  
s->val[0]='h'; s->val[1]='i'; s->val[2]='\0';
```

Unions

251

```
typedef union {int units; float kgs;} amount;
```

Unions can hold different type of values at different times

Definition similar to a structure but

- ▶ storage is shared between members
- ▶ only one field present at a time
- ▶ programmers must keep track of what it is stored

Useful for defining values that range over different types

- ▶ Critically, the memory allocated for these types is shared

Memory layout

- ▶ All members have offset zero from the base
- ▶ Size is big enough to hold the widest member
- ▶ The alignment is appropriate for all the types in the union

Union operations

252

The same operations as the ones on structures

- ▶ Assignment,
- ▶ Copying as a unit
- ▶ Taking the address
- ▶ Accessing a member

Can be initialized with a value of the type of its first member

Example

```
typedef union { int units; float kgs; } amount;

typedef struct {
    char item[15];    float price;
    int type;          amount qty;
} product;
```

Safety

253

C provides no guarantees that unions are correctly accessed

```
void print(amount x){printf("%d\n", x.units);}

int main () {
    product p[10];
    p[0].item = "toys";
    p[0].price = 2.0;
    p[0].type = 2;
    p[0].qty.kgs = 3.0;
    print(p[0].qty)
}
```

Example

254

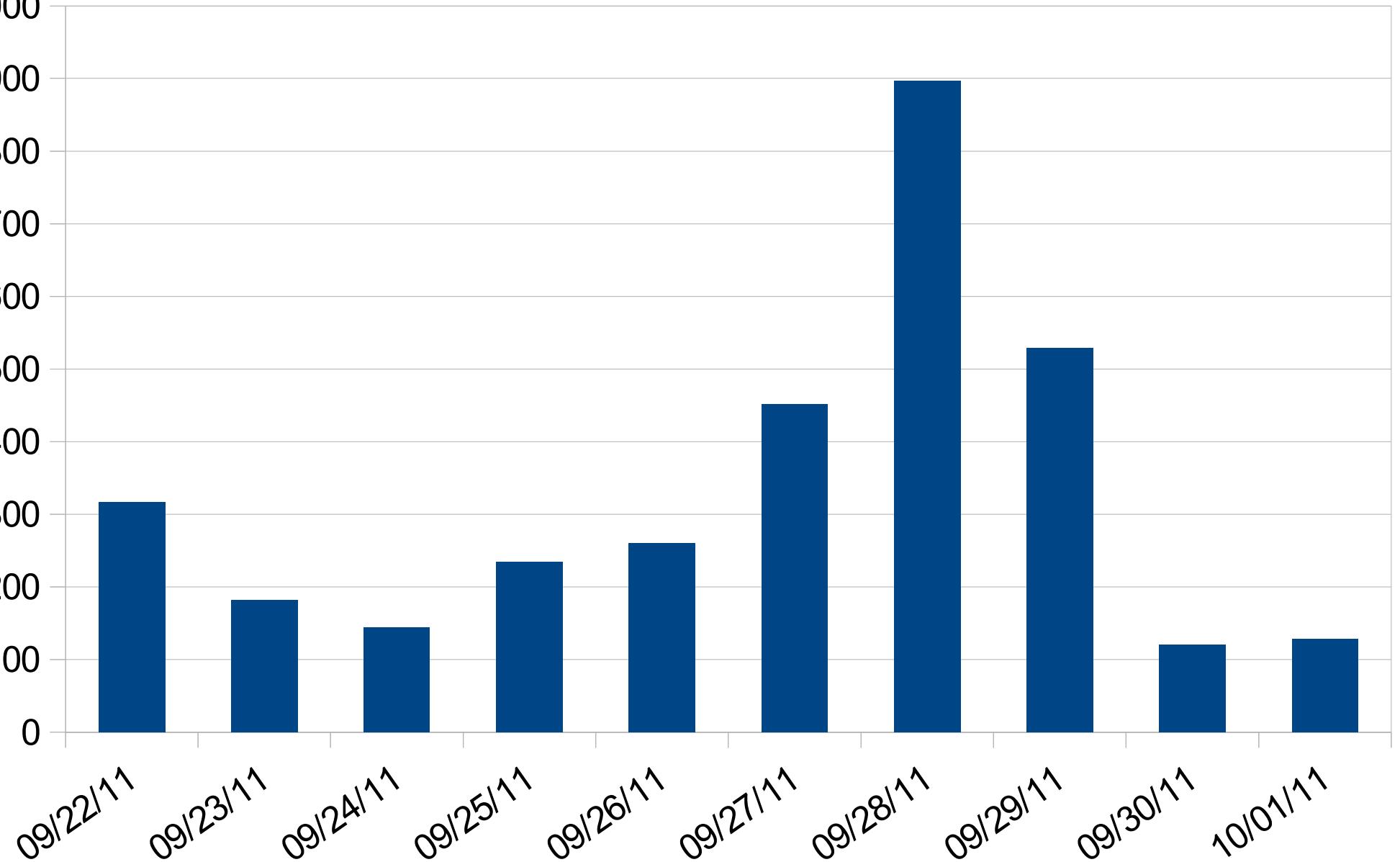
```
void check(int len, product* store[ ]) {
    for (int i=0; i<len; i++) {
        printf("%s\n", store[i]->item);
        switch (store[i]->type) {
            case 1:
                printf("%d units\n", store[i]->qty.units);
                break;
            case 2:
                printf("%f kgs\n", store[i]->qty.kgs);
                break;
        }
    }
}
```

{C}

Lecture 10



Function Pointers



~3000

Testing Lists

257

- * Count every malloc() and free(), the count has to be equal
- * Create the list ["first", "second", "third"]
- * Check that newNode(..., "first") is the head, and newNode(..., "third") is the tail
- * Test getNode with "first", "second" and "third".
- * Test getNode with "dummy", must return NULL.
- * Loop with nodeGetNext, must return each node in the order we expect.
- * Each node that nodeGetWord returns a word in newly-allocated space, which has been copied properly.
- * We verify that nodeIncCount and nodeGetCount are affecting one another.
- * We verify that nodeGetLine gives the same line we put in for each node.
- * We use deleteNode on each node independently
- * We call deleteList on the whole list (it mustn't seqfault and must count properly)

Testing lists

258

```
/* what if the node is not one on the list?? */
void test_deleteNode() {
    deleteNode(&head, &tail, secondN);
    if (nodeGetNext(firstN) != thirdN)
        fprintf(sout, "\nneither deleteNode or nodeGetNext:
                      list not in right order after node deletion\n");
    else putc('.', sout);

    deleteNode(&head, &tail, firstN);
    if (head != thirdN || tail != thirdN)
        fprintf(sout, "\ndeleteNode:
                      wrong result after deleting the first node\n");
    else putc('.', sout);

    deleteNode(&head, &tail, thirdN);
    if (head != NULL || tail != NULL)
        fprintf(sout, "\ndeleteNode:
                      wrong result after deleting the only node in list\n");
    else putc('.', sout);
}
```

Testing lists

259

```
int test_deleteList() {
    deleteList(&head, &tail);
    if (head != NULL || tail != NULL)
        fprintf(sout, "\ndeleteList:
                        list not empty after deletion\n");
    else putc('.', sout);
}
```

Testing Trends

260

- * simple1, simple2: simple test cases, no tricks
- * jj1-6: Various tests to make sure words are copied correctly and in full, the "java" vs "javascript" problem
- * evill100, evill101, evil200, evil201: Test cases with a word repeated on the 100th line, 101st line, 200th line, 201st line. In the latter two cases, the first instance is on the 100th line (so evil200 should return the same as evill100, same with 201/101)
- * msg: a Twitter stream generated with ./get_timeline | ./parser [list of words from

List code

261

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include "list.h"
```

```
struct lnode {
    struct lnode* prev;
    struct lnode* next;
    char* word;
    int count;
    int line;
};
```

List code

262

```
struct lnode *newNode(struct lnode **head, struct lnode **tail,
                      char *word, int line) {
    struct lnode * node =(struct lnode *) malloc(sizeof(struct lnode));
    if (node == NULL) {perror("Failed to allocate memory"); exit(1); }
    node->next = NULL;
    node->word = malloc(strlen(word) + 1);
    strcpy(node->word, word);
    node->line = line;
    node->count = 1;
    if (*head == NULL) {
        *head = node;
        node->prev = NULL;
    } else {
        node->prev = *tail;
        (*tail)->next = node;
    }
    *tail = node;
    return node;
}
```

List code

263

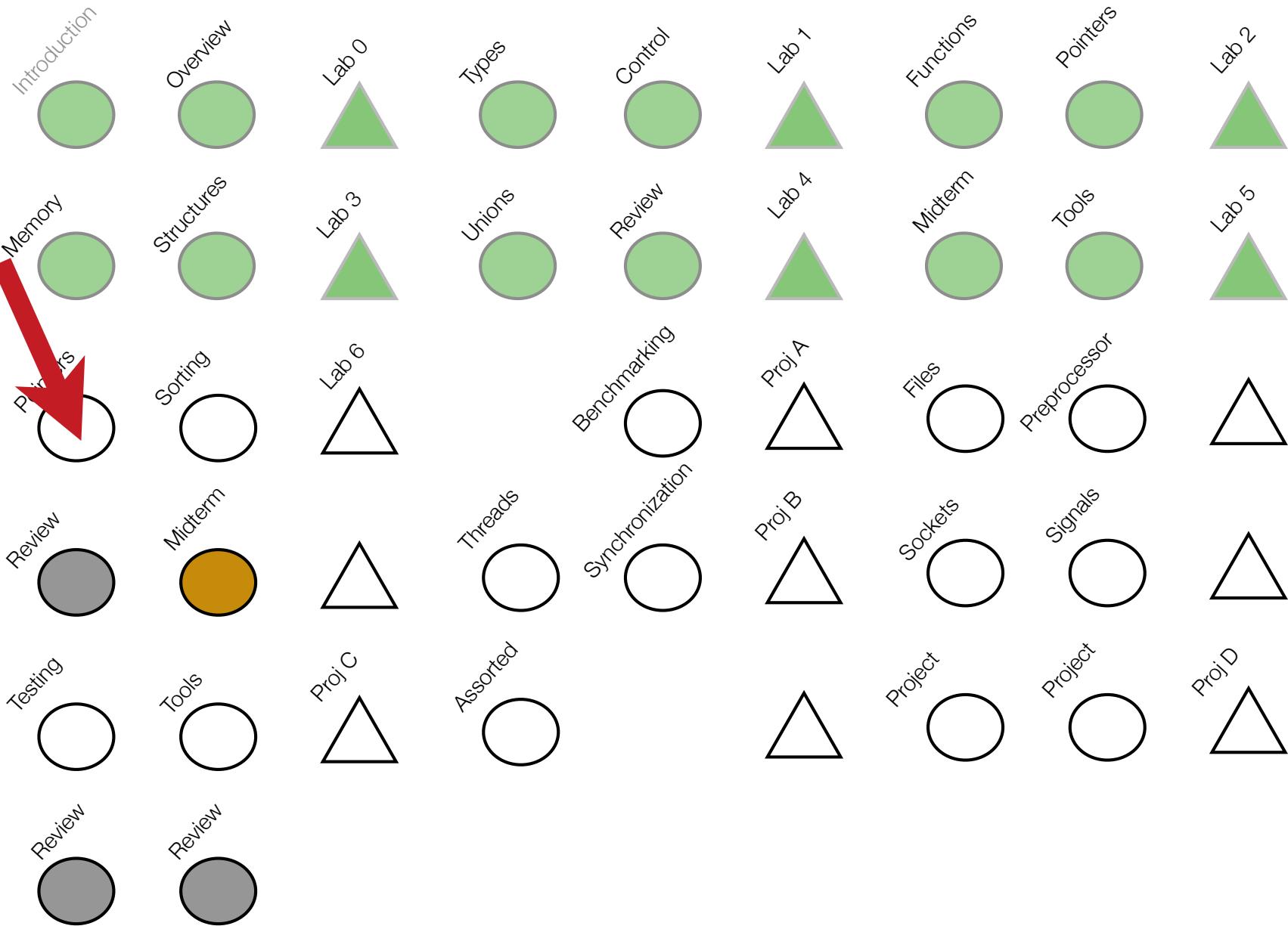
```
void deleteNode(struct lnode **head, struct lnode **tail,
                struct lnode *node) {
    if (node == *head && node == *tail) {
        *head = *tail = NULL;
    } else if (node == *head) {
        *head = node->next;
        (*head)->prev = NULL;
    } else if (node == *tail) {
        *tail = node->prev;
        (*tail)->next = NULL;
    } else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    free(node->word);
    free(node);
}
```

List code

264

```
struct lnode *getNode(struct lnode **head, struct lnode **tail,
                      char *word, int line) {
    for (struct lnode *cur = *head; cur != NULL; cur = cur->next)
        if (strcmp(cur->word, word) == 0) {
            cur->line = line; return cur;
        }
    return NULL;
}
```

```
void deleteList(struct lnode **head, struct lnode **tail) {
    struct lnode *cur = *head, *next;
    while (cur != NULL) {
        next = cur->next;
        free(cur->word);
        free(cur);
        cur = next;
    }
    *head = *tail = NULL;
}
```



Q 7

Quiz #7 - 0



267

Did you read K&R Ch 5.6 and 5.11 and 5.12?

- (a) yes
- (b) no

Quiz #7 - 1



268

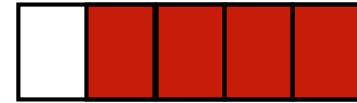
When are the following two functions equivalent (i.e. they print the same thing):

```
void writelines(char* l[], int n) {  
    for(int i=0;i<n;i++) printf("%s\n",l[i]); }
```

```
void writelines(char* l[], int n) {  
    while(n-- > 0) printf("%s\n",*l++); }
```

- (a) always
- (b) only if the argument l passed by the caller was a pointer
- (c) never

Quiz #7 - 2



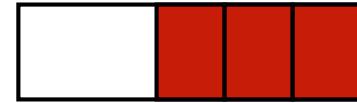
269

What is comp in the book's qsort function

```
void qsort(void *v, int l, int r,  
          int (*comp)(void*,void*));
```

- (a) an integer pointer variable
- (b) a function pointer variable that expects two functions ptr
- (c) a function pointer variable that expects two pointers

Quiz #7 - 3



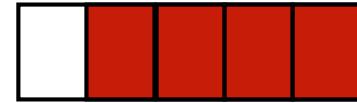
270

What is

```
int *comp(void*,void*);
```

- (a) an integer pointer variable
- (b) a function pointer variable that returns an int.
- (c) the declaration of function that returns an int pointer.

Quiz #7 - 4



271

What is

char (*(*x())[])();

- (a) syntax error
- (b) a function returning a pointer to an array of pointers to function returning a char
- (c) an array of functions returning a char

Abstraction

272

How are abstractions manifested in languages?

- ▶ As structures that encapsulate code and data providing information hiding
E.g. Classes in Java
- ▶ As program structures that refactor common usage patterns
E.g., a sorting routine that can sort lists of different types

C doesn't provide data abstractions like Java classes

- ▶ No obvious way to package related data & code within a single structure

But, it does provide a useful refactoring mechanism

- ▶ Functions are the most obvious example
- ▶ They abstract a computation over input arguments
- ▶ What kinds of arguments can these be?

Types and Computation

273

Functions can be abstracted over

- ▶ basic types (e.g., int, float, double,...)
- ▶ structured types (e.g., structs, unions, ...)

These types are primitive data abstractions

- ▶ They represent a set of values along with operations on them

What about functions themselves?

- ▶ They're obviously a form of abstraction
- ▶ Rather than representing a set of values, they represent a set of computations abstracted over arguments of a fixed type
- ▶ There is exactly one operation allowed on function types: application

Types

274

Following this line of thought:

- ▶ A type is a set of values equipped with a set of operations on those values
- ▶ A function is a computation abstracted over the types defined by its inputs
- ▶ Hence, a function is an abstraction: it represents the set of values produced by its computation when instantiated with specific arguments.

Hence, functions should be allowed to be abstracted over functions, just as they are abstracted over primitives & structures

Concretely...

275

C permits functions to be treated like any other data object

- ▶ A function pointer can be supplied as an argument
- ▶ Returned as a result
- ▶ Stored in any array
- ▶ Compared

Main caveat:

- ▶ Cannot deference the object pointed to by a function pointer on the left-hand side of an assignment

Example

276

Operating on a list of integers...

```
struct list {  
    int val;  
    struct list * next;  
};  
  
typedef struct list List;
```

Creating lists...

277

```
#include <stdio.h>
List *makeList(int n) {
    List *l, *l1 = NULL;
    for (int i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->val = n-i;
        l->next = l1;
        l1 = l;
    }
    return l;
}
```

Given a number n,
build a list of length n where
the ith element of the list
contains i

Creating lists... (2)

278

```
#include <stdio.h>
List *makeList(int n) {
    List *l, *l1 = NULL;
    for (int i = 0; i < n; i++) {
        l = malloc(sizeof(List));
        l->val = i+1;
        l->next = l1;
        l1 = l;
    }
    return l;
}
```

Given a number n,
build a list of length n where
the ith element of the list
contains $n-i+1$

Creating lists... (3)

279

We can imagine many different ways of populating a list

- ▶ The overall control structure remains the same
- ▶ Only the computation responsible for producing the next element changes

How can we abstract the definition to reuse the same control structure for the different kinds of lists we might want?

Function pointers

280

Supply a pointer to the function that computes values

```
int add (int m) {  
    static int n = 0;  
    n++;  
    return m-n+1;  
}  
  
int min(int m) {  
    static int n = 0;  
    n++;  
    return n;  
}
```

The expression **add** returns a pointer to the code of **add**

Abstraction revisited

281

```
List *makeGenList (int n, int (*f)(int)) {  
    List * l, *l1 = NULL;  
    for (int i = 0; i < n; i++) {  
        l = (List*) malloc(sizeof(List));  
        l->val = (*f)(n);  
        l->next = l1;  
        l1 = l;  
    };  
    return l;  
}
```

Applies (invokes) the function pointed to by f with argument n

Expects a function pointer that points to a function which yields an int, and which expects an int argument

```
makeGenList(10,min);  
makeGenList(10,add);
```

Can create lists with different elements (but same structure) without changing

Next step...

282

Now, lets define abstractions that compute over lists

```
int fold(int (*f)(int,int), List *l, int acc) {  
    if (l == NULL)  
        return acc;                                a list of integers      an accumulator  
    else {  
        int x = l->val;  
        fold (f, l->next, (*f)(x,acc));  
    }  
}
```

A function pointer that operates over pairs of integers and returns an int

Each recursive call to *fold* operates on the current value and the current accumulator; the result becomes the new value of the accumulator in the next call

Using fold

283

Each computation expressed using the same definition

```
int sum(int x,int y){ return x + y; }
int mul(int x,int y){ return x * y; }
int max(int x,int y){ return (x>y)?x:y; }
int main () {
    int s,m,x; List *l;
    l = makeGenList(10, min);
    s = fold(sum,l,0);
    m = fold(mul,l,1);
    x = fold(max,l,0);
}
```

Map

284

Fold allows a function to operate over the elements of a list

C's type system conspires against richer kinds of operations

- ▶ the accumulator must be an int
- ▶ one can circumvent the type system with casts, but this is risky

Instead of accumulating a result, suppose we want to apply a function to each element in the list?

- ▶ Such operations are called maps

Map

285

```
List* map(int(*f)(int), List *l) {  
    if (l == NULL) return l;  
    List * ll = malloc(sizeof(List));  
    ll->val = (*f)(l->val);  
    ll->next = map( f, l->next);  
}
```

A function pointer that points to a function which takes an integer argument and produces an integer result

Apply the function pointed to by f to the current list element

Recursively apply map to the rest of the list

Map

286

```
int add(int m) { return m+1; }
int min(int m) { return m-1; }
int eve(int x) { return (x%2 == 0) ? 1 : 0; }

int main () {
    int a,m,e;
    List *l, *eveL, *addL, *minL;
    l = makeGenList(10,...);
    eveL = map( eve,l);
    addL = map (add,l);
    minL = map (min,l);
    ...
}
```

Objects...

287

Typical code that could use an object-oriented solution

```
enum TYPE{SQUARE,RECT,CIRCLE,POLYGON};  
struct shape {  
    float params[MAX];  
    enum TYPE type;  
};  
typedef struct shape Shape;  
  
void draw(Shape* s) {  
    switch(s->type) {  
        case SQUARE: draw_square(s); break;  
        case RECT:   draw_rect(s); break;  
        ...  
    }  
}
```

Objects...are arrays of fun *s

288

Typical code that could use an object-oriented solution

```
void (*fp [4])( Shape* s) = {  
    drawSquare, drawRec,  
    drawCircle, drawPoly};
```

```
void draw (Shape* s ) {  
(*fp[s->type])(s); // call right fun  
}
```

Managing state...

289

Objects are more than array of fun pointers, they have state
Similarly, some functions need to retain private state

Example: Counter

- ▶ implement a function that counts the number of times it was invoked
- ▶ the count should be hidden
- ▶ there can be multiple counters in the program

Counter v1

290

```
int count1 = 0;  
...  
int countn = 0;  
  
int count (int *x) { return ++(*x); }
```

State in a global variable, caller passes a pointer to the variable. Drawbacks:

- ▶ variables public, can be modified from anywhere
- ▶ fixed number of counter variables, to change it requires recompilation
- ▶ client code may make a mistake when invoking the counter

Counter v2

291

```
int count () {  
    static int count;  
    return ++count;  
}
```

Here the state is held in a private static variable. Drawbacks:

- ▶ there can be only one counter per function
- ▶ to add counters, the programmer must create multiple functions

Closures

292

```
typedef void *(*generic)(void *);  
  
typedef struct {  
    generic function;  
    void *environment;  
} closure;
```

To a first approximation, a counter has two parts

- ▶ the code that implements the counter
- ▶ the “environment” that holds the counter value

void type

293

A pointer to a void type points to a value that has no type.

- ▶ This means there are no allowable operations on them.
- ▶ Must cast void pointers to concrete type in order to access its target value
- ▶ One use of void pointers is to pass “generic” parameters to a function

```
void f (void* data, int sz) {  
    if (sz == sizeof(char)) {  
        char* p = (char*)data; ++(*p);  
    } else if (sz == sizeof(int)) {  
        int* p = (int*)data; ++(*p);  
    }  
}
```

void type

294

```
void f (void* data, int sz) {  
    if (sz == sizeof(char)) {  
        char* p=(char*)data; ++(*p);  
    } else if (sz == sizeof(int)) {  
        int* p=(int*)data; ++(*p);  
    }  
}
```

...

```
char a = 'x';  
f(&a,sizeof(a));
```

```
int b = 1602;  
f(&b,sizeof(b));
```

*What is the value of *a and
b after the two calls to f?

void types

295

void pointers can be used to point to any data type

```
int x; void*p=&x; // points to int  
float f;void*p=&f; // points to float
```

void pointers cannot be dereferenced

```
void*p; printf("%d",*p); // invalid  
void*p; int *px=(int*)p;  
printf ("%d",*px); // valid
```

Counter revisited

296

```
int nextval(void *env);

closure make(int start) {
    closure c;
    int *val = malloc(sizeof(int));
    *val = start;
    c.function = (generic)nextval;
    c.environment = val;
    return c;
}
```

Counter generator

297

```
int nextval(void *env) {  
    int *value = env;  
    (*value)++;  
    return (*value);  
}
```

Use

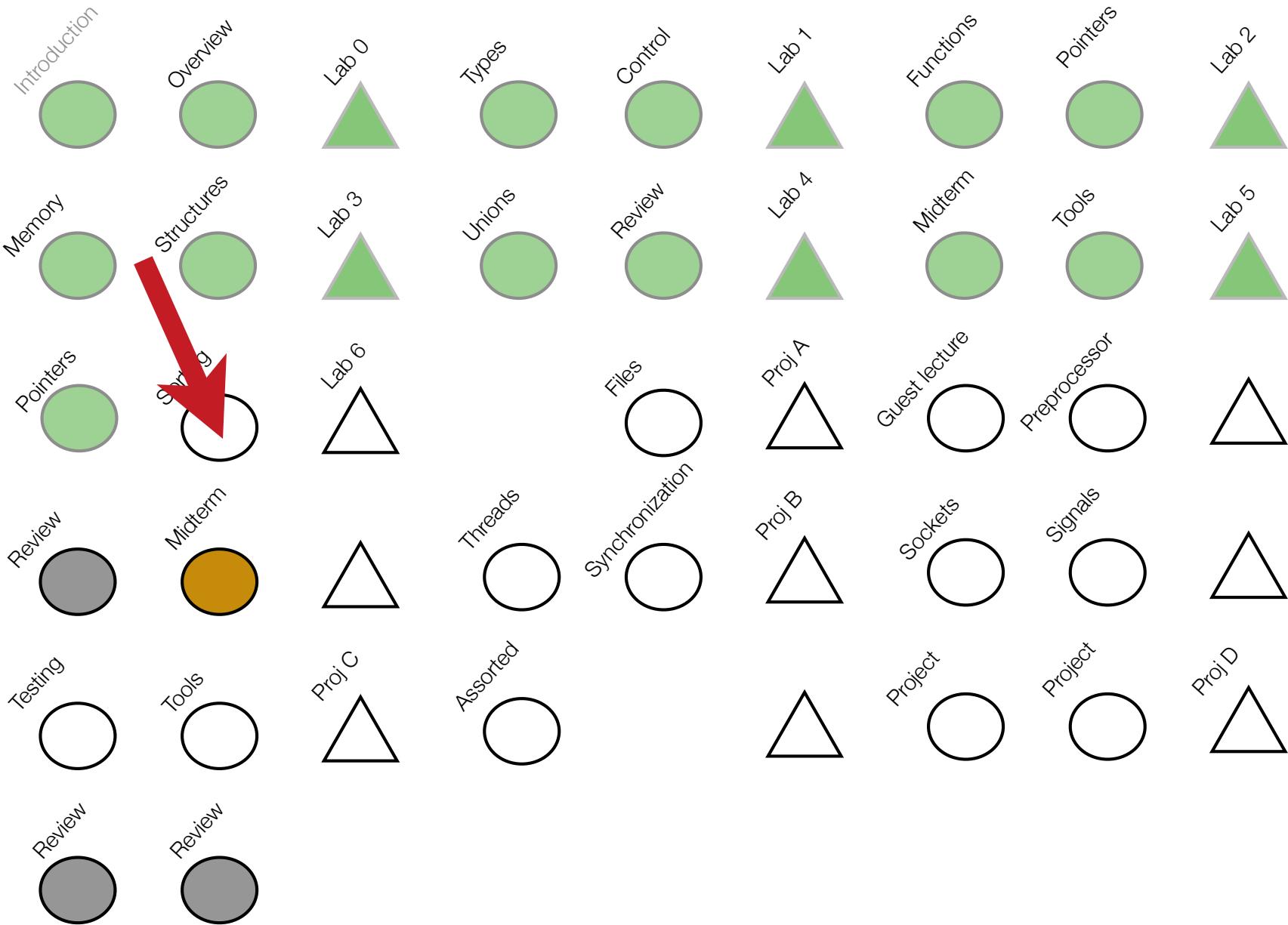
298

```
int main() {
    /* Create the two closures */
closure count1 = make(2);
closure count2 = make(3);
printf("%d\n",
    ((generic)count1.function)(count1.environment));
printf("%d\n",
    ((generic)count2.function)(count2.environment));
printf("%d\n",
    ((generic)count1.function)(count1.environment));
return 0;
}
```

{LC}

Lecture 11





Sorting

301

Given a collection of items and a total order over them, sort the collection under this order.

- ▶ Total order: every item is ordered with respect to every other item

Given an array of numbers, sort them in ascending order

- ▶ sort in descending order
- ▶ sort odd before even, with each sub-collection respecting an ordering

Given an array of strings, sort in lexicographic order

Given an array of structures, sort on the value of a field

Observation

302

The code necessary to perform the sort

▶ swapping elements, rearranging the array as necessary
works independently of the type of the elements being sorted.

If the elements have a comparator that understands ordering,
a sorting function can sort different kinds of elements

Polymorphism:

▶ poly : many
▶ morphism : form

```
void swap(void *v[], int i, int j) {  
    void *temp=v[i]; v[i]=v[j]; v[j]=temp;  
}
```

void is C's mechanism to support polymorphic types

qsort

304

```
void qsort(void* v[], int left, int right,
           int(*comp)(void*,void*)) {
    if (left >= right) return;
    swap(v, left, (left + right)/2);
    int last = left;
    for (int i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left])) < 0)
            swap(v, ++last,i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp)
}
```

Quicksort

305

Start with: 06 34 69 33 75 64 04 74 25 95 15 58 78 36 51 73 13 27

Basic idea: 06 34 69 33 75 64 04 74 95 15 58 78 36 51 73 13 27

pivot = 25 Pick an element and sort around it

 34 69 33 75 64 04 74 95 15 58 78 36 51 73 13 27 pivot = 25

Start checking elements to the right of the hole until we find an element smaller than the pivot

04 34 69 33 75 64 74 06 95 15 58 78 36 51 73 13 27

04 69 33 75 64 34 74 06 95 15 58 78 36 51 73 13 27

After processing 06

04 06 33 75 64 34 74 69 95 15 58 78 36 51 73 13 27 pivot = 25

At the end, we have: 04 06 15 13 64 34 74 69 95 33 58 78 36 51 73 75 27 pivot=25

Repeat the process for each of the two subarrays

Recursion ends with a partition has only two elements

qsort

306

```
char *lineptr[MAXLINES] /* pointers to text lines */

int readlines(char *lineptr[], int nlines);

void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void*,void*));

int numcmp(char*,char*);

int numeric;

qsort((void**)lineptr, 0, nlines-1,
      (int (*)(void*,void*))numeric ? numcmp: strcmp))
```

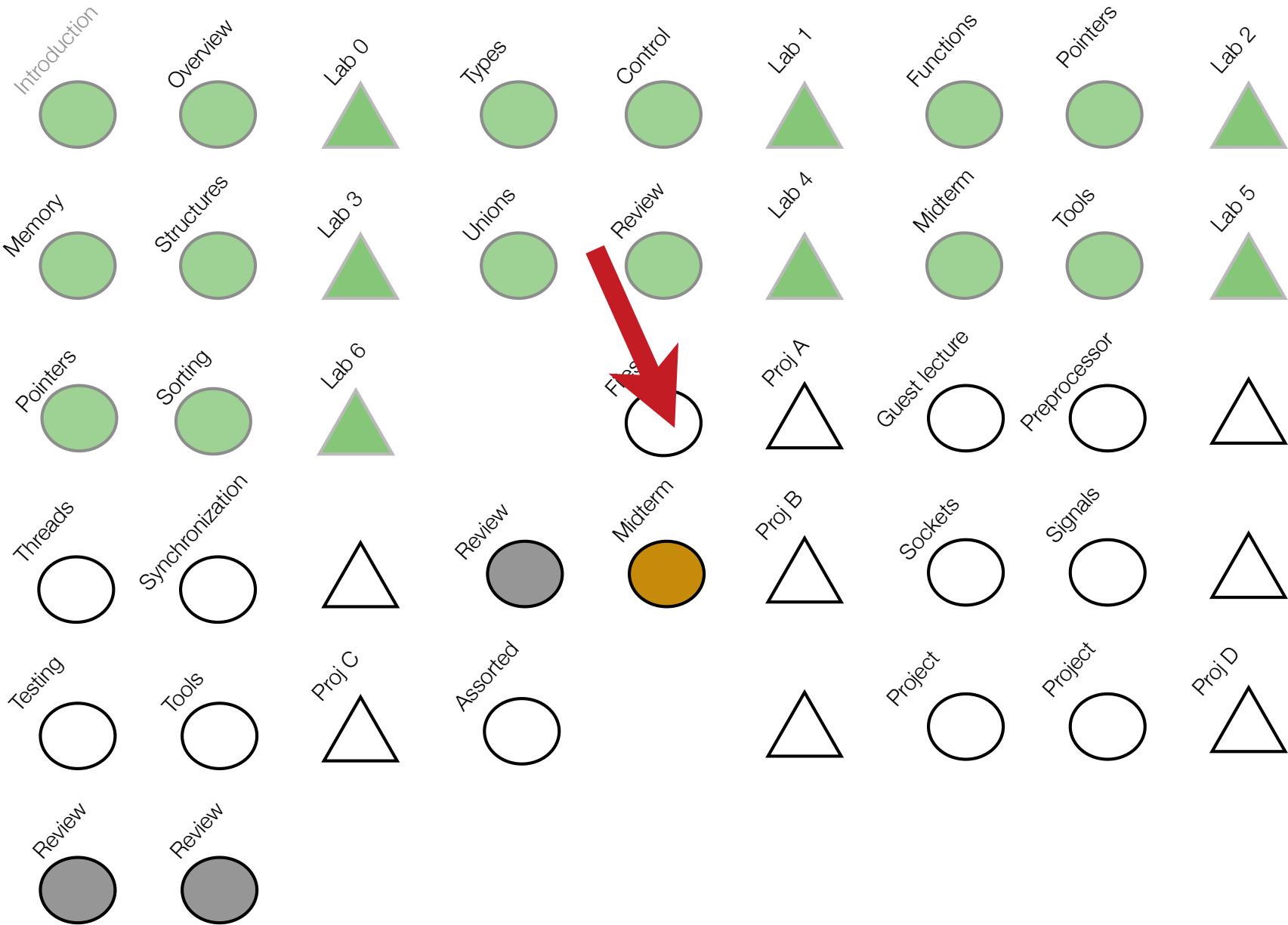
{C}

Lecture 12



Files





Q 8

Quiz #8 - 0



311

Did you read K&R Ch 7?

(a) yes

(b) no

Quiz #8 - 1



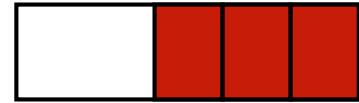
312

What does the following do:

```
printf("%.*s", x, y);
```

- (a) print the contents of string x and string y
- (b) print the string x to the end
- (c) print the first x characters of string y

Quiz #8 - 2

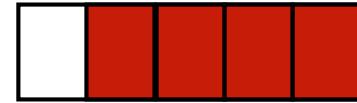


313

How many number of arguments does printf expect

- (a) 2
- (b) either 2 or 3
- (c) at least one
- (d) any number

Quiz #8 - 3



314

Why is the following code incorrect?

```
scanf ("%o %d", &x, 3);
```

- (a) you should never take the address of an unknown variable
- (b) all arguments of scanf must be valid addresses
- (c) there are too few arguments
- (d) scanf does not have a %o argument type

Quiz #8 - 4



315

If `fopen` fails to open the requested file, what is returned?

- (a) EOF
- (b) an empty file
- (c) NULL
- (d) -1
- (e) undefined

How do programs interact ?

316

Communicate via “standard” input and “standard” output

- ▶ Typically bound to the display

Use redirection to read or write to a file

`a.out < inputfile`

`a.out > outputfile`

`printf()` writes to `outputfile` rather than to the screen

`a.out < inputfile > outputfile`

More general approach ...

317

Redirection is really in the operating system, not C

But, C provides a set of library functions for performing I/O

We've used one such library extensively:

- ▶ **stdio.h**

provides operations to read (`getchar`) and write (`putchar`) characters, print formatted strings (`printf`), read formatted strings (`scanf`), etc.

Stdio.h

318

Also provides more general operations on files

A file is an abstraction of a non-volatile memory region:

- ▶ its contents remain even after the program exits
- ▶ C exposes the file abstraction using the **FILE** type:

`FILE *fp` // *fp is a pointer to a file

- ▶ Can only access the file using the interfaces provided by the language

File Systems

319

A file system specifies how information is organized on disk and accessed

- ▶ directories
- ▶ files

In UNIX the following are files

- ▶ Peripheral devices (keyboard, screen, etc)
- ▶ Pipes (inter process communication)
- ▶ Sockets (communication via computer networks)

Files representation

- ▶ Text files (human readable format)
- ▶ Binaries (for example executables files)

System Calls

320

System calls are services provided by the operating system

C Libraries let users invoke system calls through C functions

Example:

- ▶ I/O operations (I/O access is slower than memory access)
- ▶ Memory allocation

File manipulation

321

Three basic actions:

- ▶ open the file: make the file available for manipulation

- ▶ read and write its contents

No guarantee that these operations actually propagate effects to the underlying file system

- ▶ close the file: enforce that all the effects to the file are “committed”

File Descriptors

322

To operate on a file, the file must be opened

An open file has a non-negative integer called file descriptor

For each program the operating system opens implicitly three files: standard input, standard output and standard error, that have associated the file descriptors 0, 1, 2 respectively

- ▶ Primitive, low-level interface to input and output operations
- ▶ Must be used for control operations that are specific to a particular kind of device.

Streams

323

Higher-level interface, layered on top of file descriptor facilities

More powerful set of functions

Implemented in terms of file descriptors

- ▶ the file descriptor can be extracted from a stream and used for low-level operations
- ▶ a file can be open as a file descriptor and then make a stream associated with that file descriptor.

Opening files

324

`FILE* fopen(const char* filename, const char* mode)`

► mode can be "r" (read), "w" (write), "a" (append)

returns NULL on error (e.g., improper permissions)

filename is a string that holds the name of the file on disk

`int fileno(FILE *stream)`

► returns the file descriptor associated with stream

```
char *mode = "r";
```

```
FILE* ifp = fopen("in.list", mode);
```

```
if(ifp==NULL){fprintf(stderr,"Failed");exit(1);}
```

```
FILE* ofp = fopen("out.list", "w");
```

```
if (ofp==NULL) {...}
```

Reading files

325

fscanf requires a **FILE*** for the file to be read

```
fscanf(ifp, "<format string>", inputs)
```

Returns the number of values read or EOF on an end of file

Example: Suppose in.list contains

```
foo 70  
bar 50
```

To read elements from this file, we might write

```
fscanf(ifp, "%s %d", name, count)
```

Can check against EOF:

```
while(fscanf(ifp,"%s %d",name,count)!=EOF);
```

Testing EOF

326

Ill-formed input may confuse comparison with EOF

`fscanf` returns the number of successful matched items

```
while(fscanf(ifp,"%s %d",name,count)==2)
```

Can also use `feof`:

```
while (!feof(ifp)) {
    if (fscanf(ifp,"%s %d",name,count)!=2) break;
    fprintf(ofp, format, control)
}
```

Closing files

327

```
fclose(ifp);
```

Why do we need to close a file?

File systems typically buffer output

```
fprintf(ofp, "Some text")
```

There is no guarantee that the string has been written to disk

Could be stored in a file buffer maintained in memory

The buffer is flushed when the file is closed, or when full

File pointers

328

Three special file pointers:

- ▶ **stdin** (standard input)
- ▶ **stdout** (standard output)
- ▶ **stderr** (standard error)

Typically **stdin** is associated with the keyboard device, **stdout** and **stderr** are associated with the display

redirecting **stdout** doesn't redirect **stderr**

a.out > outfile

Can be used wherever a regular **FILE *** is expected

Other file operations

329

Remove file from the file system:

```
int remove (const char * filename)
```

Rename file

```
int rename (const char * oldname,  
           const char * newname)
```

Create temporary file (removed when program terminates)

```
FILE * tmpfile (void)
```

Raw I/O

330

Read at most **nobj** items of size **sz** from **stream** into **p**

feof and **ferror** used to test end of file

```
size_t fread(void* p, size_t sz, size_t nobj, FILE* stream)
```

Write at most **nobj** items of size **sz** from **p** onto **stream**

```
size_t fwrite(void*p, size_t sz, size_t nobj, FILE* stream)
```

File position

331

`int fseek(FILE* stream, long offset, int origin)`

Set file position in the stream. Subsequent reads and writes begin at this location

Origin can be `SEEK_SET`, `SEEK_CUR`, `SEEK_END` for binary files

For text streams, offset must be zero (or value returned by `f.tell()`)

Return the current position within the stream

`long ftell(FILE * stream)`

Sets the file to the beginning of the file

`void rewind(FILE * stream)`

see page 247-248 in the text

Example

332

```
#include <stdio.h>
int main() {
    long fsize;
    FILE *f;

    f = fopen("log", "r");

    fseek(f, 0, SEEK_END) ;
    fsize = ftell(f) ;
    printf("file size is: %d\n", fsize);

    fclose(f);
}
```

Text Stream I/O Read

333

Read next char from stream and return it as an unsigned char cast to an int, or EOF

```
int fgetc(FILE * stream)
```

Reads in at most size-1 chars from the stream and stores them into null-terminated buffer pointed s. Stop on EOF or error

```
char* fgets(char *s, int size, FILE *stream)
```

Writes c as an unsigned char to stream and returns the char

```
int fputc (int c, FILE * stream)
```

Writes string s without null termination; returns a non-negative number on success, or EOF on error

```
int fputs(const char *s, FILE *stream)
```

File descriptors

334

A handle to access a file, like the file pointer in streams

Small non-negative integer used in same open/read-write/close ops

Returned by `open`; all opens have distinct file descriptors

Once a file is closes, fd can be reused

Same file can be opened several times, with different fd's

Management functions

335

```
#include <fcntl.h>

int open(const char *path, int flags);
int open(char *path, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);

Flags: O_RDONLY, O_WRONLY or O_RDWR bitwise OR
with O_CREAT, O_EXCL, O_TRUNC, O_APPEND,
O_NONBLOCK, O_NDELAY

int close(int fd);
```

```
#include <unistd.h>  
  
ssize_t read(int fd, void *buf, size_t cnt);  
ssize_t write(int fd, void *buf, size_t cnt);
```

fd is a descriptor, not FILE pointer

Returns number of bytes transferred, or -1 on error

Normally waits until operation is enabled (e.g., there are bytes to read), except under O_NONBLOCK and O_NDELAY (in which case, returns immediately with “try again” error condition)

Example

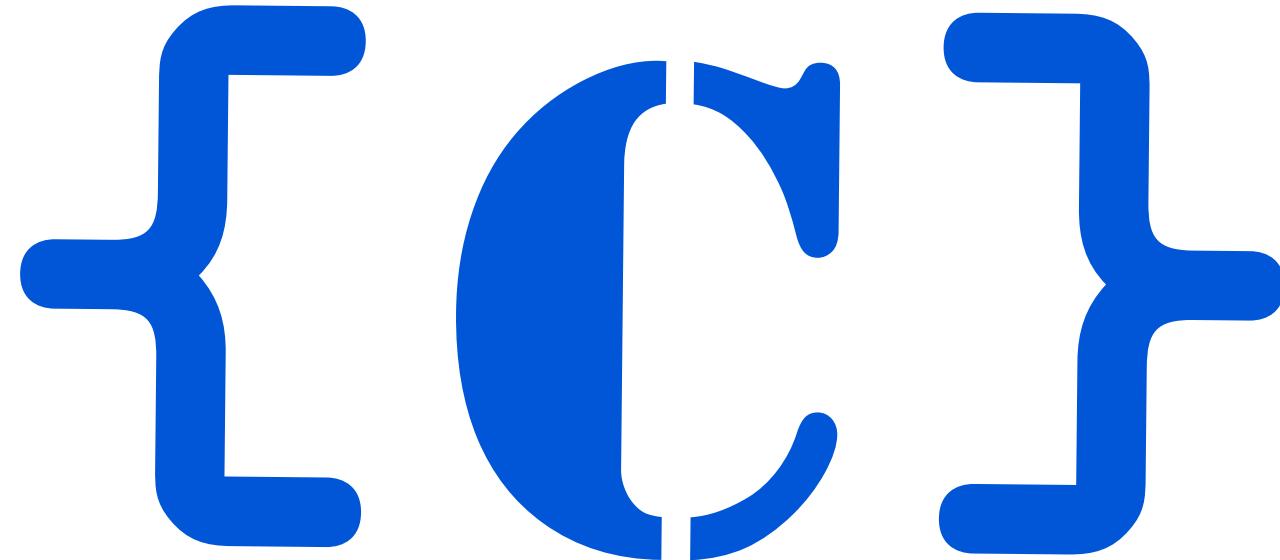
337

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

int main() {
    char buf[100];

    int f1 = open("log1", O_RDONLY);
    int f2 = open("log2", O_RDONLY);
    fprintf(stderr, "Log1 file descriptor is: %d\n", f1);
    fprintf(stderr, "Log2 file descriptor is: %d\n", f2);
    close(f1); close(f2);

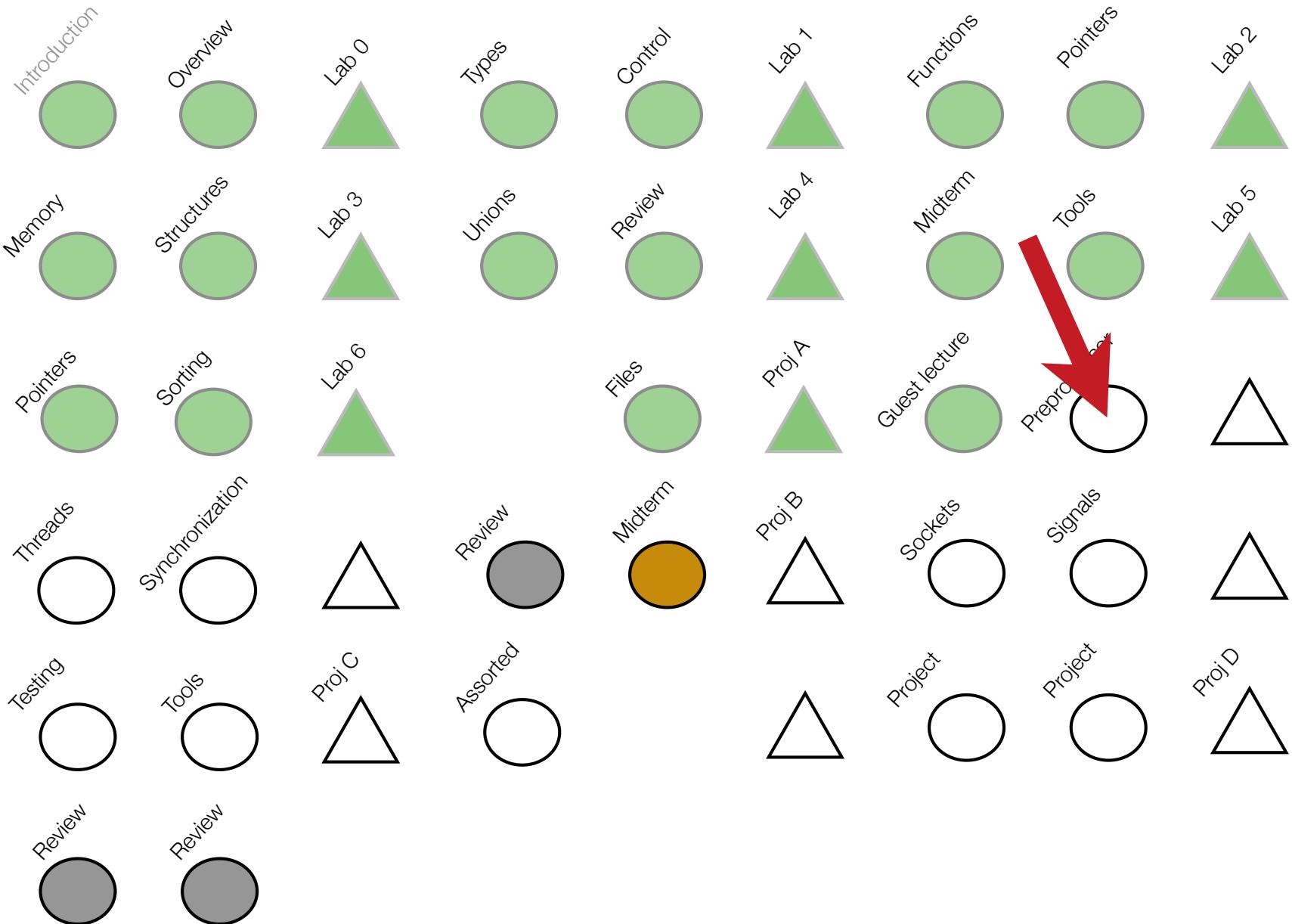
    f2 = open("log2", O_RDONLY);
    fprintf(stderr, "Notice the new file descriptor: %d\n", f2);
    close(f2);
}
```



Lecture 13

Preprocessor

Assertions and Error Handling



Aparté: The C preprocessor

The C compiler performs *Macro expansion and directive handling*

- ▶ Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros

Example:

- ▶ Specify how to expand macros with the DEBUG variable
- ▶ gcc -D option

```
#ifdef DEBUG  
  
#define DPRINT(s) fprintf(stderr,"%s\n",s)  
  
#else  
  
#define DPRINT(s)  
  
#endif
```

Aparté: The C preprocessor

The C compiler performs *macro expansion* and *directive handling*

- ▶ Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros

```
DPRINT( makeString("error", CAUSE) );
```

Question:

- ▶ Is function `makeString()` called?

How can this code fail?

```
#include <stdio.h>

int main() {
    int a, b, c;

    a = 10;
    b = getchar() - 48;
    c = a/b;

    return 0;
}
```

Common Software Vulnerabilities

343

Buffer overflows

Input validation

Format string problems

Integer overflows

Failing to handle errors

Other exploitable logic errors

Weak Types and Errors

344

Would strong typing prevent these kinds of vulnerabilities?

- ▶ What kind of errors do type systems typically catch?

Structural violations: think of types as sets

Not all elements in a set are sensible in all contexts

- ▶ Think of buffers as arrays

Buffer overflow arises when arrays of different sizes than expected are constructed.

- ▶ Similar reasoning for overflow and underflow

Is C more vulnerable than...

345

Weak typing means data can be interpreted in multiple ways

- ▶ This can lead to errors
- ▶ A single datum associated with multiple types

Memory can be indexed arbitrarily, beyond the range(s) of declared arrays and structures

- ▶ Overwrite stack contents

Dangling pointers

- ▶ Pass a pointer to an object allocated locally within a function to the function's caller

What is a Buffer Overflow?

346

Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold

Very dangerous because the extra information may:

- ▶ Affect user's data
- ▶ Affect user's code
- ▶ Affect system's data
- ▶ Affect system's code

Why Buffer Overflows?

347

No check on boundaries

- ▶ Programming languages give user too much control
- ▶ Programming languages have unsafe functions
- ▶ Users do not write safe code

C and C++, are more vulnerable because they provide no built-in protection against accessing or overwriting data in any part of memory

- ▶ Can't know the lengths of buffers from a pointer
- ▶ No guarantees strings are null terminated

Why Buffer Overflow Matter

348

Overwrites:

- ▶ other buffers
- ▶ variables
- ▶ program stack

Results in:

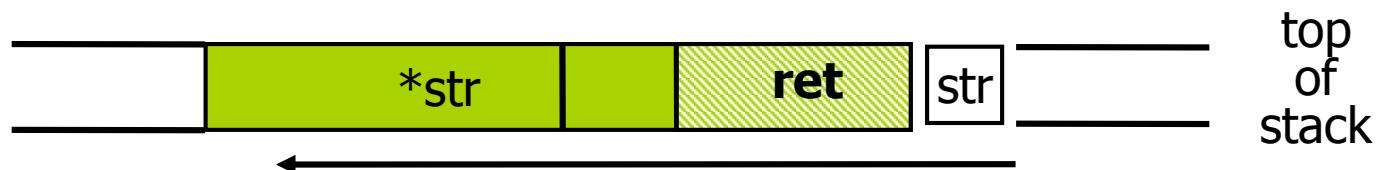
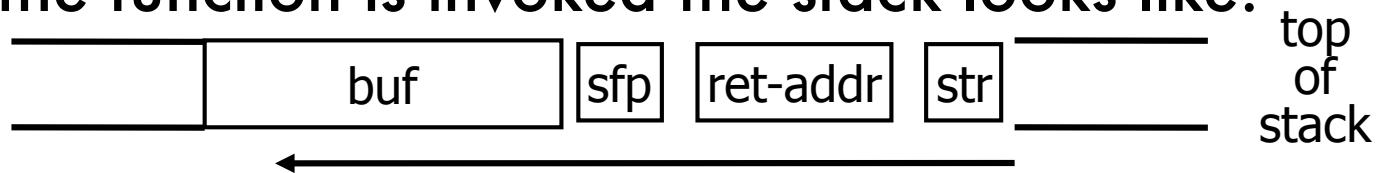
- ▶ erratic program behavior
- ▶ a memory access exception
- ▶ program termination
- ▶ incorrect results
- ▶ breach of system security

Example

- Suppose a web server contains a function:

```
void f(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    do(buf);  
}
```

- When the function is invoked the stack looks like:



Some Unsafe C lib Functions

```
strcpy(char *dest, const char *src)
strcat(char *dest, const char *src)
gets(char *s)
scanf(const char *format, ...)
printf(const char *format, ...)
```



Assertions

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = some_function_computes_something();  
  
    c = a/b;  
    return 0;  
}
```

Assertions

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = some_function_computes_something();  
    assert(b!=0);  
    c = a/b;  
    return 0;  
}
```

Assertions

Used to help specify programs and to reason about correctness
precondition

- ▶ an assertion placed at the beginning of a section of code
- ▶ determines the set of states under which the code is expected to be executed

postcondition

- ▶ placed at the end
- ▶ describes the expected state at the end of execution.

```
#include <assert.h>
```

```
assert (predicate);
```

How can this code fail?

```
#include <stdio.h>
#define MAX 10
char* f(char s[ ]);

int main() {
    char str[MAX];
    char *ptr=f(str);
    printf("%c\n", *ptr);
    return 0;
}
```

How can this code fail?

```
#include <stdio.h>
#include <assert.h>
#define MAX 10
char* f(char s[]);

int main() {
    char str[MAX];
    char *ptr=f(str);
    assert(ptr!=NULL);
    printf("%c\n", *ptr);
}

char *f(char s[]) {
    return NULL;
}
```

What to fear/check for...

Null pointer dereference

 Use after free

 Double free

 Array indexing errors

Mismatched array new/delete

Potential stack/heap overrun

Return pointers to local variables

 Logically inconsistent code

 Uninitialized variables

 Invalid use of negative values

Under allocations of dynamic data

 Memory leaks

 File handle leaks

 Unhandled return codes

Detecting and reporting errors

357

```
FILE * f;
f = fopen("myfile", "r");
if(f == NULL) {
    exit(fprintf(stderr, "Open file failed\n"));
}
```

The C library detects errors and provides support for reporting error conditions and messages.

Errno

358

errno.h defines the error conditions detected by the C Library
Some functions report informative error messages:

char * strerror(int errno);

void perror(const char *message);

errno contains the system error number

Initial value of errno at program startup is 0

Any library function may modify errno

Functions do not change errno when they succeed; the value of errno after a successful call is not necessarily 0

- ▶ Do not use errno to see if a function failed. But if the function failed, examine errno

Error codes

359

All the error codes have symbolic names. Examples:

- ▶ **EPERM** : Operation not permitted; only the owner of the file or processes with special privileges can perform the operation
- ▶ **EINVAL**: Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function

```
#define EPERM          1 /* Operation not permitted */
#define ENOENT          2 /* No such file or directory */
#define ESRCH           3 /* No such process */
#define EINTR            4 /* Interrupted system call */
#define EIO              5 /* I/O error */
#define ENXIO            6 /* No such device or address */
#define E2BIG            7 /* Argument list too long */
#define ENOEXEC          8 /* Exec format error */
#define EBADF            9 /* Bad file number */
#define ECHILD           10 /* No child processes */
#define EAGAIN           11 /* Try again */
#define ENOMEM           12 /* Out of memory */
#define EACCES           13 /* Permission denied */
```

Error messages

360

```
#include <stdio.h>
void perror(const char *message);
```

Prints the string message and the error message corresponding to the value of errno to the stream stderr.

If message is a null pointer or an empty string, perror prints the error message corresponding to errno

Otherwise perror prefixes its output with the string message.

```
int fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) {
    perror("open");
    exit(1);
}
```

open: Permission denied

Error messages

361

```
include <string.h>
char * strerror (int errnum);
```

Maps error code **errnum** to a descriptive error message string and returns a pointer to this string

The value **errnum** normally comes from the variable **errno**

- ▶ You should not modify the string returned by **strerror**
- ▶ Subsequent calls to **strerror**, might overwrite the error string

Error messages

362

Many programs are designed to exit if any system call fails. By convention, the error message from the program should start with the program's name

How do we obtain the program's name?

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
    char name[]="myfile";
    FILE * stream = fopen(name, "r");
    if (!stream) {
        fprintf (stderr, "%s: Couldn't open file %s;" "%s\n",
                argv[0], name, strerror (errno));
        exit(1);
    }
    fclose(stream);
    return 0;
}
```

Varargs

363

```
#include <stdarg.h>
void va_start(va_list ap, arg_name);
```

initializes processing of a varying-length argument list.

`arg_name`, is the name of the parameter to the calling function after which the varying part of the parameter list begins (the parameter immediately before the `,...`). The results of the `va_start` macro are unpredictable if the argument values are not appropriate.

```
void va_end(va_list ap);
```

ends varying-length argument list processing

Varying-length argument lists

364

```
#include <stdarg.h>
(arg_type) va_arg(va_list ap, arg_type);
```

returns the value of the next argument in a varying-length argument list.

`va_list` must be initialized by a previous use of the `va_start` macro, and a corresponding `va_end` should be called after finishing processing the arguments.

`arg_type` is the type of the argument that is expected.

- ▶ the results are unpredictable if the argument values are not appropriate
- ▶ no way to test whether a particular argument is the last one in the list.
Attempting to access arguments after the last one in the list produces unpredictable results.

Example..

365

```
#include <stdarg.h>
void myprintf(const char *fmt, ...) {
    va_list argp;
    int i; char *s; char fmtbuf[256];

    va_start(argp, fmt);
    for(char *p = fmt; *p != '\0'; p++) {
        if(*p != '%') { putchar(*p); continue; }

        switch(*++p) {
            case 'c': i = va_arg(argp, int); putchar(i); break;
            case 'd':
                i = va_arg(argp, int); s = itoa(i, fmtbuf, 10);
                fputs(s, stdout); break;
            case 's':
                s = va_arg(argp, char *); fputs(s, stdout); break;
        }
    }
    va_end(argp);
}
```

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

#define PERMS 0666 /* RW for owner, group and others */

void error(char *, ...);
int main(int argc, char *argv[]) {
    int f1, f2, n;
    char buf[BUFSIZ];
    if(argc!=3) error("Usage: cp from to");
    if((f1 = open(argv[1],O_RDONLY,0))== -1)
        error("cp: can't open %s", argv[1]);
    if((f2 = creat(argv[2],PERMS))== -1)
        error("cp: can't create %s, mode%3o", argv[2], PERMS);
    while((n = read(f1, buf, BUFSIZ)) > 0)
        if(write(f2, buf, n) != n)
            error("cp: write error on file %s", argv[2]);
    return 0;
}
```

{Loc}

Lecture 14

Threads



Quiz #9 - 0

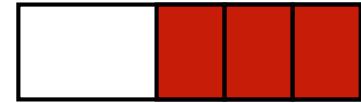


368

Did you read about POSIX threads?

- (a) yes
- (b) no

Quiz #9 - 1

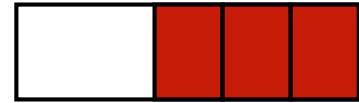


369

In a multi-threaded program, what would happen if a thread's stack did overflow?

- (a) only data stored in the program heap would be corrupted
- (b) the program's code could be modified
- (c) other thread stacks and/or the heap could be changed

Quiz #9 - 2

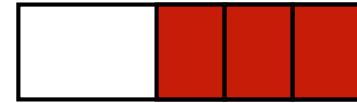


370

The main reason for using Pthreads on a multi-core is:

- (a) to obtain optimum performance
- (b) to improve memory usage
- (c) to decrease the load on the operating system

Quiz #9 - 3



371

Which of the following characteristics of a program is not well suited for pthreads:

- (a) often waits on I/O
- (b) task can be broken up in chunks
- (c) use many CPU cycles in some places but not others
- (d) work must be done without interruption

Quiz #9 - 4

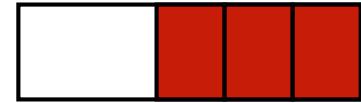


372

Which one of these isn't an argument of `pthread_create`?

- (a) `thread`: opaque, unique identifier for the new thread
- (b) `attr`: opaque obj. that may be used to set thread attributes
- (c) `start`: the C routine the thread will execute once created
- (d) `mode`: indicates if the thread can read or write memory

Quiz #9 - 5



373

After a thread has been created, how do you know when it will be scheduled to run by the operating system?

- (a) you don't
- (b) on most architectures this happens every 20 milliseconds
- (c) you have to periodically ask the operating system

Concurrency and parallelism

374

Concurrency is concerned with the management of logically simultaneous activities

- ▶ Best-fit job scheduling
- ▶ event handling (GUI)
- ▶ web server

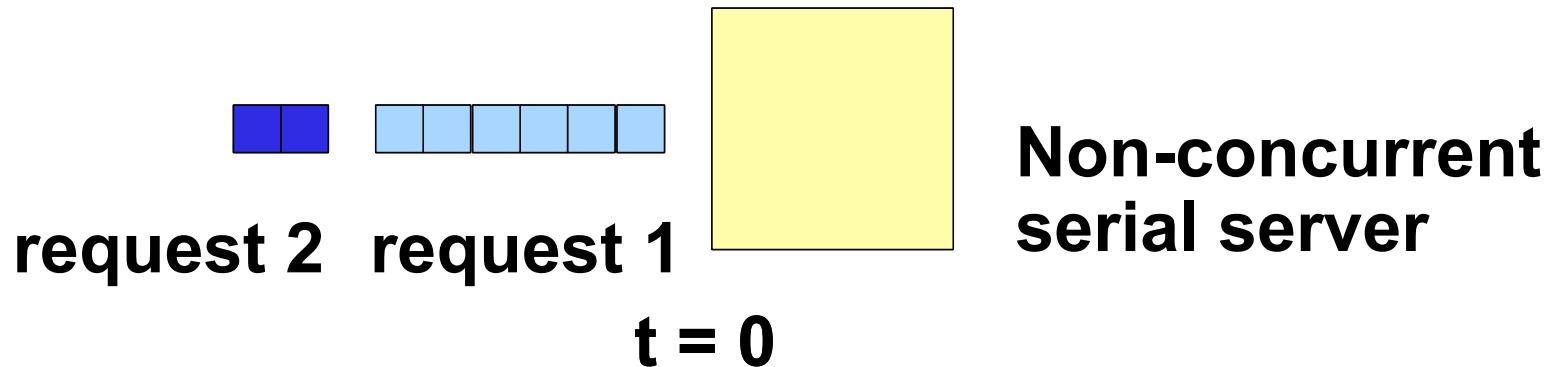
Parallelism is concerned with performance of concurrent activities

- ▶ weather forecasting
- ▶ simulations

Why Concurrency?

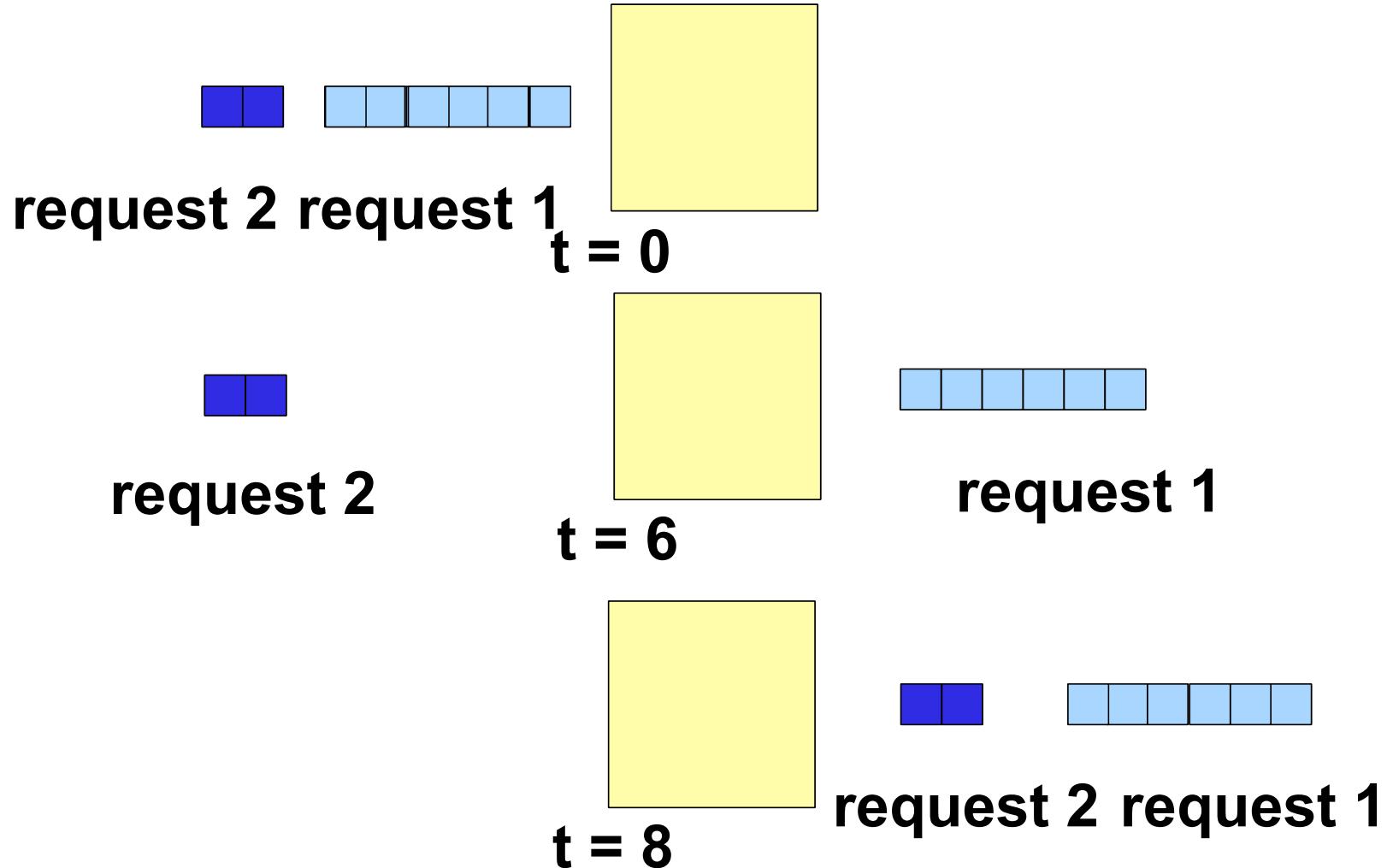
375

In a serial environment, consider the following simple example of a server, serving requests from clients (e.g., web server and web clients)



Seriously...

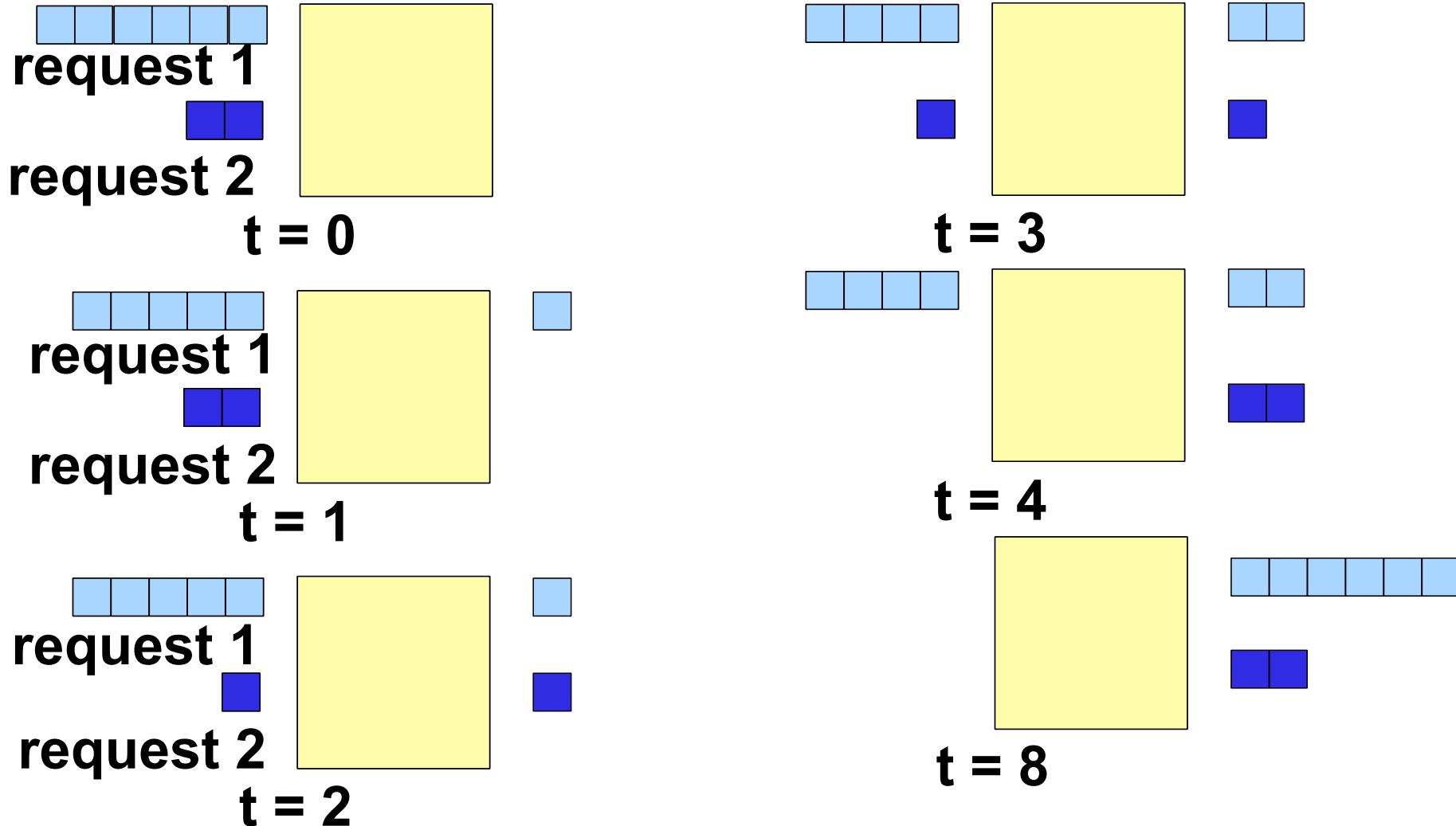
376



Total completion time = 8 units,
Average service time = $(6 + 8)/2 = 7$ units

Concurrently...

377



Total completion time = 8, Average service time = $(4+8)/2 = 6$

Why Concurrency?

378

The lesson from the example is quite simple:

- ▶ Not knowing anything about execution times, we reduced average service time by processing requests concurrently

But what if I knew the service time for each request?

- ▶ Would “shortest job first” not minimize average service time anyway?
- ▶ But what about the poor request at the back never getting any service (starvation/ fairness)?

Why Concurrency?

379

Notions of service time, starvation, and fairness motivate the use of concurrency in virtually all aspects of computing:

- ▶ Operating systems are multitasking
- ▶ Web/database services handle multiple concurrent requests
- ▶ Browsers are concurrent
- ▶ Virtually all user interfaces are concurrent

Why Concurrency?

380

In a parallel context, the motivations are more obvious:

- ▶ Concurrency + parallel execution = performance

Threads and processes

381

Thread: an independent unit of execution that shares resources with other threads

Process: an independent unit of execution isolated from all other processes and shares no resources

Resources:

- ▶ Registers
- ▶ Stack
- ▶ Heap
- ▶ File descriptors
- ▶ Shared libraries
- ▶ Program instructions

Parallelism?

382

The execution of concurrent tasks on platforms capable of executing multiple tasks at a time is referred to as “parallelism”

- ▶ Parallelism integrates elements of execution – and associated overheads
- ▶ For this reason, we typically examine the correctness of concurrent programs and performance of parallel programs

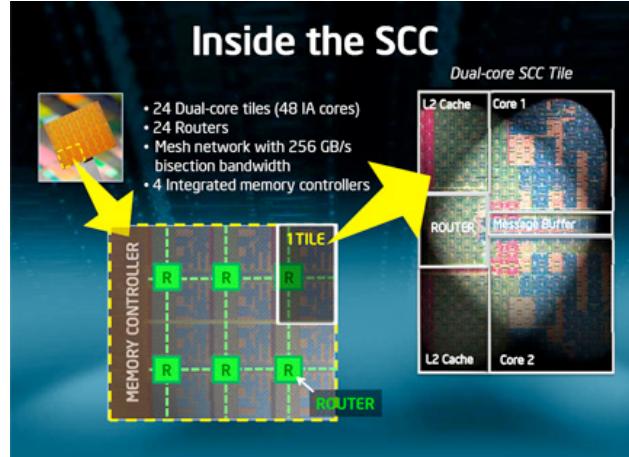
We can broadly view the resources of a computer to include the processor, data-path, memory subsystem, disk, and network

- ▶ Contrary to popular belief, each of these resources is a bottleneck
- ▶ Parallelism alleviates bottlenecks

Modern Architectures



AMD
32 dual cores



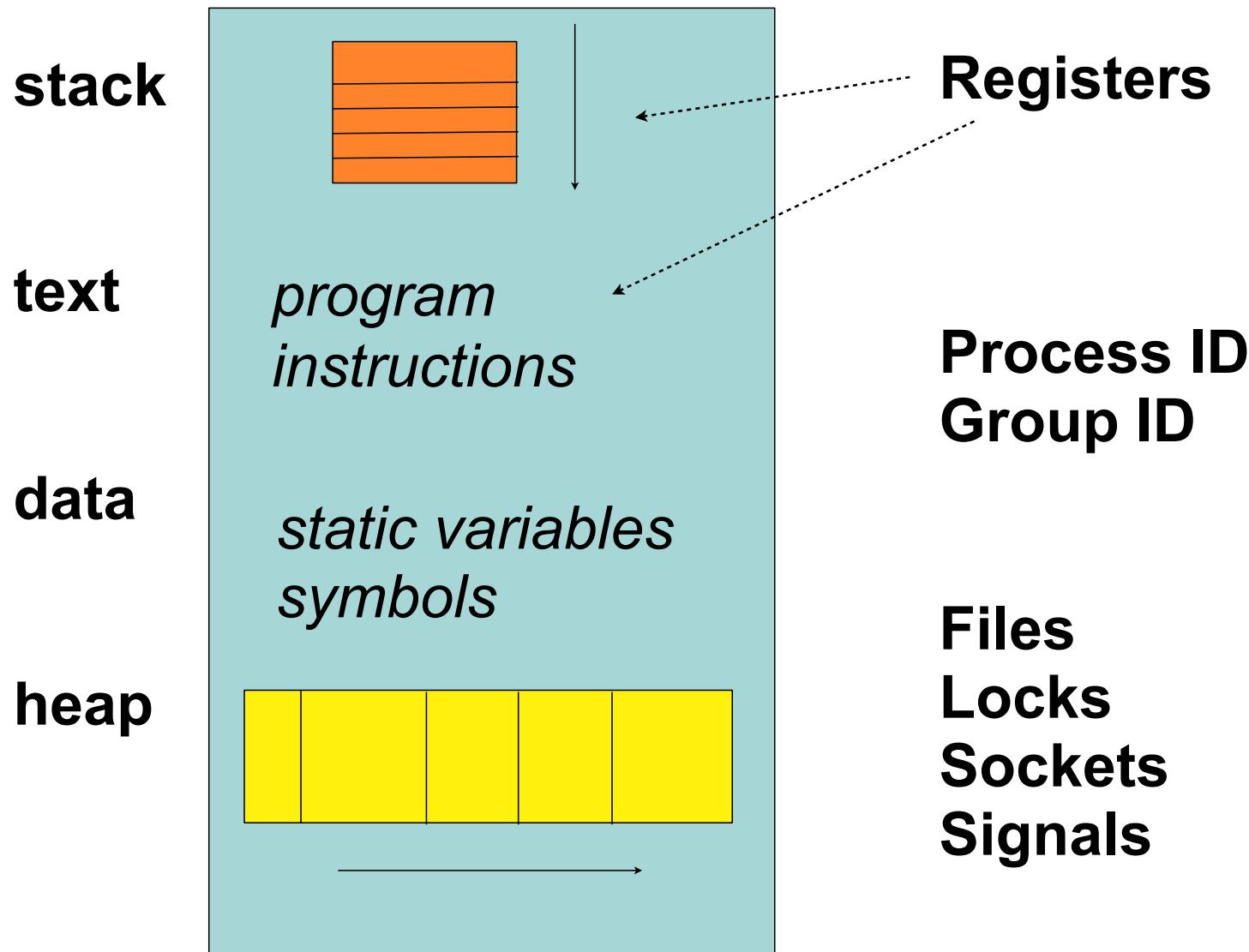
SCC
24 dual cores



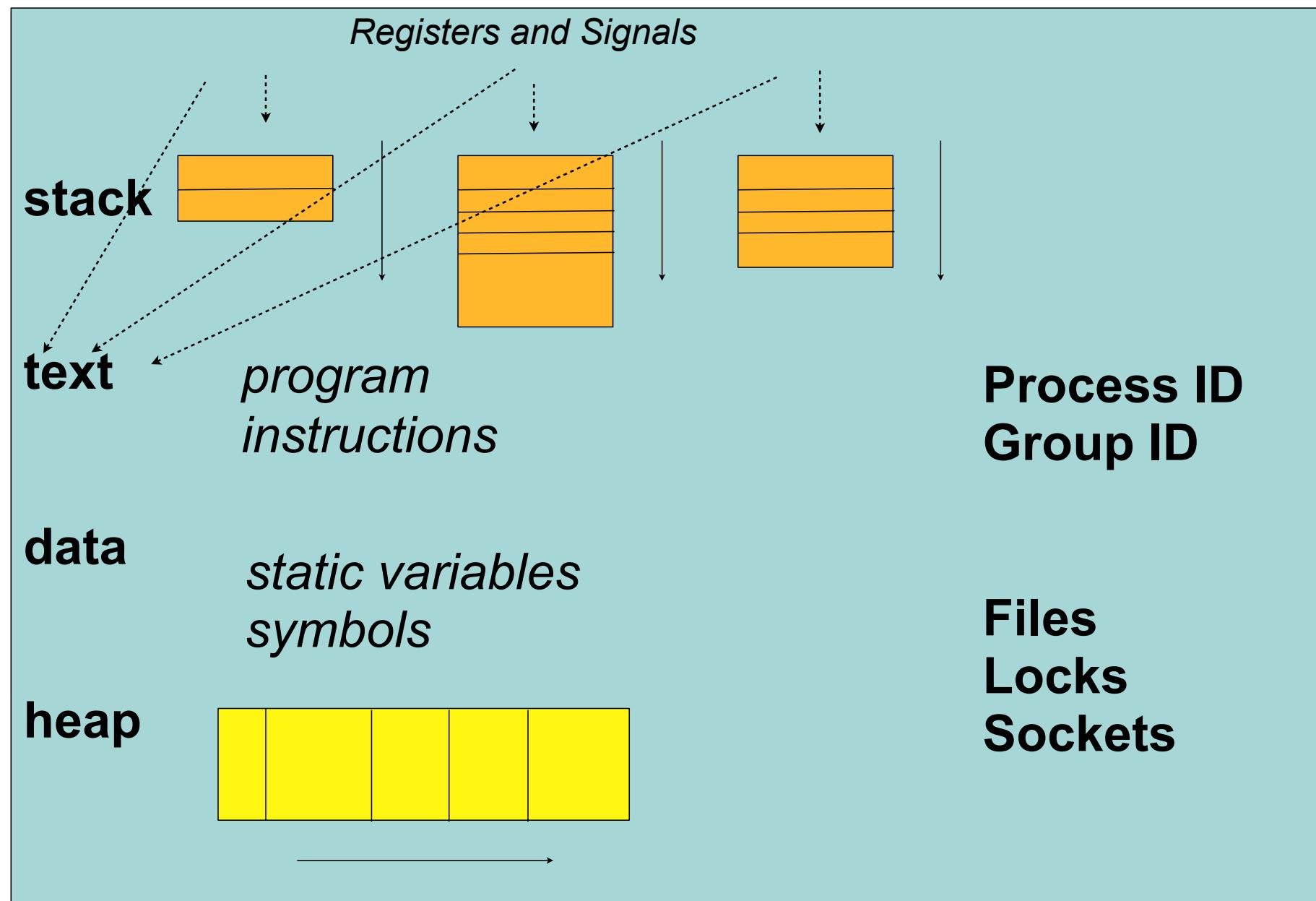
Azul
864 cores
16 x 54 cores

How should we program these kinds of machines?

A Process



Threads within a Process



Threads

386

Exists within a process

- ▶ But, independent control flow
- ▶ share common process resources (like heap and file descriptors)
 - changes made by one thread visible to others
 - pointers have meaning across threads
 - two threads can concurrently read and write to the same memory location

Maintain their own stack pointer

Registers

Pending and blocked signals

Scheduled by the operating system

Desired Structure

387

Programs can be decomposed into discrete independent tasks

The points where they overlap should be easily discerned and amenable for protection

Three basic structures

- ▶ master-worker
- ▶ result-oriented
- ▶ pipeline-oriented

Architectural abstraction

388

Shared memory

- ▶ Every thread can observe actions of other threads on non-thread-local data (e.g., heap)
- ▶ Data visible to multiple threads must be protected (synchronized) to ensure the absence of data races

A data race consists of two concurrent accesses to the same shared data by two separate threads, at least one of which is a write

Thread safety

- ▶ Suppose a program creates n threads, each of which calls the same procedure found in some library
- ▶ Suppose the library modifies some global (shared) data structure
- ▶ Concurrent modifications to this structure may lead to data corruption

Example

389

THREAD 1

```
a = data;  
a++;  
data += a;
```

THREAD 2

```
b = data;  
b++;  
data += b;
```

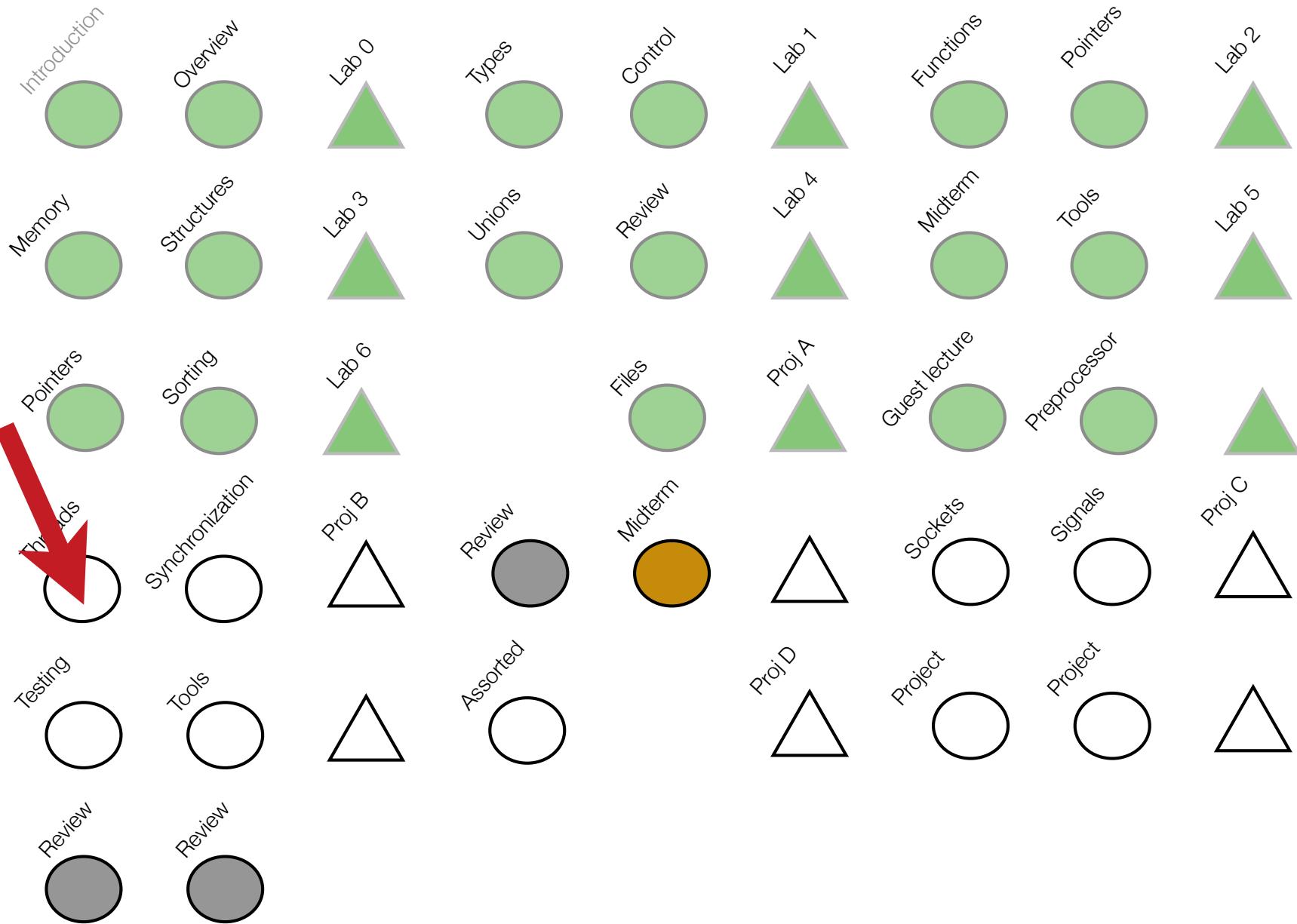
Assuming **data==0** initially, can **data** be 1 at the end?

{Loc}

Lecture 15

Synchronization





Q 10

Quiz #10 - 1



393

What happens when main() returns?

- (a) all threads are destroyed
- (b) existing threads receive a signal to clean up
- (c) other threads are unaffected

Quiz #10 - 2

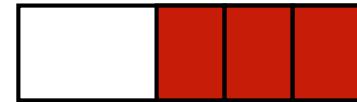


394

To make sure that a thread can be joined

- (a) call `pthread_attr_setdetachstate`
- (b) pass an attribute at creation
- (c) check that the operating system support joins

Quiz #10 - 3

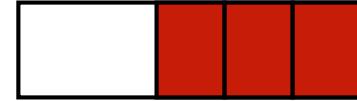


395

A mutex variable can be used to

- (a) prevent data races by limiting concurrency
- (b) multiplex threads on a SMP
- (c) protect the stack of a thread from concurrent access

Quiz #10 - 4



396

When more than one thread is waiting for a locked mutex,
which thread will be granted the lock first after it is released?

- (a) the first thread that tried to acquire the lock
- (b) the last thread that tried to acquire the lock
- (c) one of them

Thread creation

397

Initially, your program is a single, default thread

pthread_create creates a new thread and makes it executable. Arguments:

- ▶ **thread**: An opaque identifier for the new thread returned by the call
- ▶ **attr**: An opaque attribute that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values
- ▶ **start_routine**: the routine that the thread will run once it is created
- ▶ **arg**: A single argument that may be passed to start_routine. It must be passed as a void pointer. NULL is allowed.

The maximum number of threads that may be created by a process is implementation dependent.

Creating threads

398

```
#include <pthread.h>
#include <stdio.h>
#define N 5

void* hello(void *id) {
    printf("Hello %d\n", (long)id);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[N];
    for(long t=0; t<N; t++){
        int rc = pthread_create(&threads[t], NULL, hello, (void *)t);
        if (rc) exit(-1);
    }
    pthread_exit(NULL);
}
```

Passing arguments

399

```
long* args[N];  
  
for(t=0; t<N; t++) {  
    args[t] = (long *) malloc(sizeof(long));  
    *args[t] = t;  
    printf("Creating thread %ld\n", t);  
    rc = pthread_create(&threads[t], NULL, hello, (void*) args[t]);  
    ...  
}
```

Arguments

400

```
struct data{ int id; int sum; char *message; }

struct data tda[N];

void *print(void *arg) {
    struct data *my;
    my = (struct data*) arg;
    tid = my->id;
    sum = my->sum;
    hello_msg = my->message;
    ...
}

int main (int argc, char *argv[]) {
    tda[t].id = t;
    tda[t].sum = sum;
    tda[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, print, (void*)&tda[t]);
    ...
}
```

Wrong?

401

```
int rc;
long t;

for(t=0; t<N; t++) {
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, print, (void *) &t);
    ...
}
```

Mutual exclusion

402

At most one thread can “acquire” a mutex at any given time.

- ▶ Once the acquiring thread “releases” the mutex, another thread waiting for it can acquire it
- ▶ Threads waiting for a mutex are blocked from performing any other work

What are some of the logical errors that can occur when mutexes are used incorrectly

- ▶ Not used when they should be
- ▶ Used when they shouldn't be

Parallel dot product

403

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

typedef struct {
    double      *a;
    double      *b;
    double      sum;
    int         veclen;
} DOTDATA;

#define N 4
#define V 100
DOTDATA dotstr;
pthread_t callThd[N];
pthread_mutex_t mx;
```

Parallel dot product

404

```
void *dotprod(void *arg) {
    /* Define and use local variables for convenience */
    long offset = (long)arg;
    double sum = 0, *x, *y;
    int len = dotstr.veclen;
    int start = offset*len;
    int end   = start + len;
    x = dotstr.a;
    y = dotstr.b;
    /* Perform the dot product and assign result to the appropriate variable in the structure. */

    for (int i=start; i<end ; i++)
        sum += (x[i] * y[i]);
    /* Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating. */
    pthread_mutex_lock (&mx);
    dotstr.sum += sum;
    pthread_mutex_unlock (&mx);
    pthread_exit((void*) 0);
}
```

Parallel dot product

405

```
int main (int argc, char *argv[]){
    void *status; pthread_attr_t attr;
    double *a = (double*) malloc (N*V*sizeof(double));
    double *b = (double*) malloc (N*N*sizeof(double));
    for (i=0; i<N*V; i++){ a[i]=1.0; b[i]=a[i]; }
    dotstr.veclen = V; dotstr.a = a; dotstr.b = b; dotstr.sum=0;
    pthread_mutex_init(&mx, NULL);
    /* Create threads to perform the dotproduct */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for(long i=0; i<N; i++)
        /* Each thread works on a different set of data. The offset is specified by 'i'.
           The size of the data for each thread is indicated by VECLEN. */
        pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    pthread_attr_destroy(&attr);
    /* Wait on the other threads */
    for(i=0; i<N; i++) pthread_join(callThd[i], &status);
    /* After joining, print out the results and cleanup */
    printf ("Sum = %f \n", dotstr.sum);
    free (a); free (b);
    pthread_mutex_destroy(&mx); pthread_exit(NULL);
}
```

Let's hack...

406

```
#include <math.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define SLOW for(int i=0;i<1000;i++)
#define MAX 1000000
#define STEP    1000

int H[MAX]; int pos;

int int_cmp(void *a,void *b){ SLOW; return *(int*)a-*(int*)b; }

int main () {
    for(int i=0;i<MAX;i++) H[i] = MAX-i;

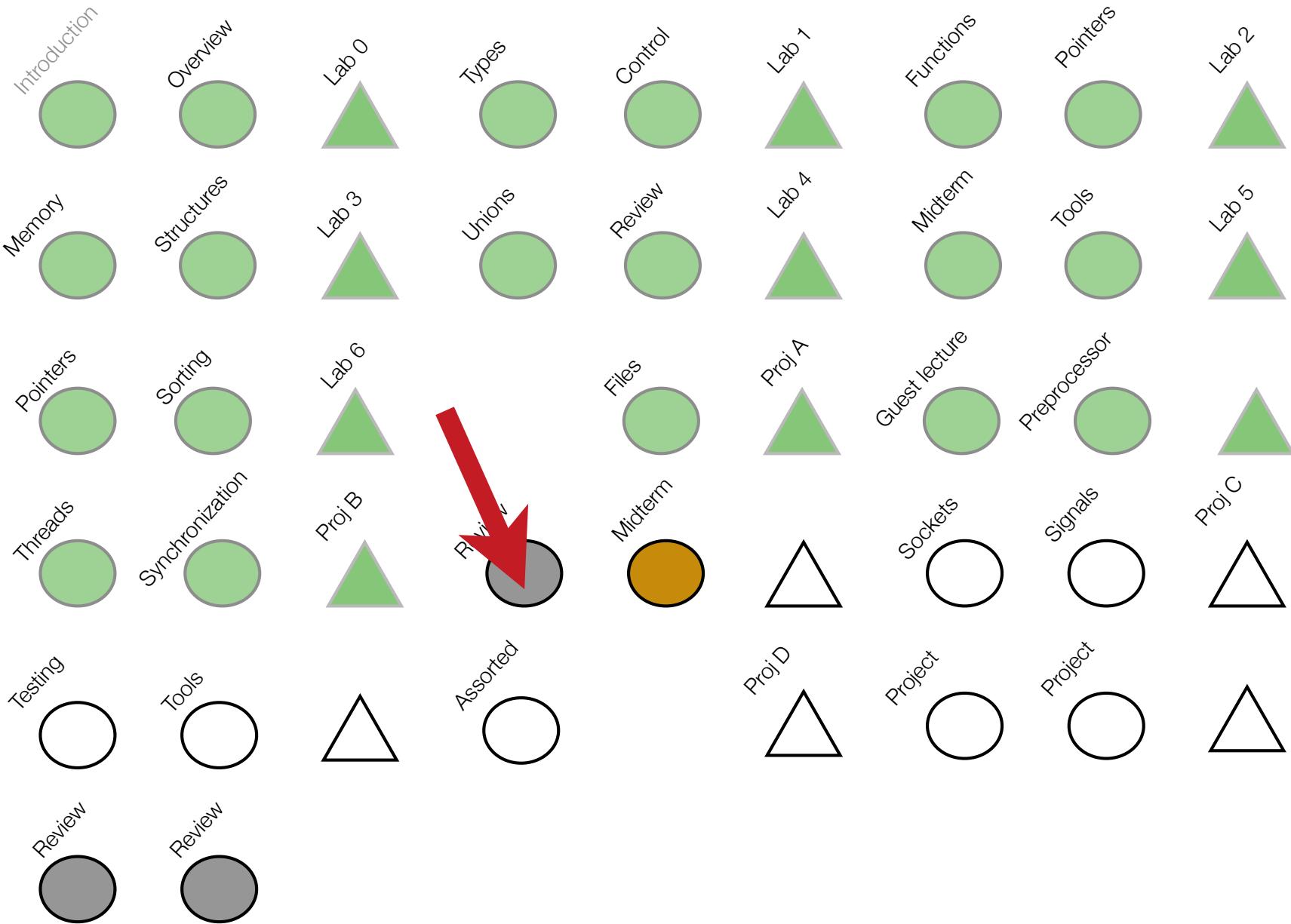
    int* h = NULL;
    while (1) {
        h=H+pos;
        if (h>H+MAX-STEP) break;
        qsort(h, STEP, sizeof(int), int_cmp);
        pos+=STEP;
    }

    int sum = 0; for(int i=0;i<MAX;i++) sum += H[i];
    printf("%d\n",sum-1784293664);
}
```

Let's hack...

407

```
#define NUM 4
#define MAX 1000000
#define STEP 1000
pthread_mutex_t mx;
int H[MAX];
int pos;
void* work(void* v){
    int* h = NULL;
    while (1) {
        pthread_mutex_lock (&mx);
        h=H+pos;
        pos+=STEP;
        pthread_mutex_unlock (&mx);
        if (h>H+MAX-STEP) break;
        qsort(h, STEP, sizeof(int), int_cmp);
    }
    pthread_exit(NULL);
}
int main () {
    pthread_t threads[NUM];
    pthread_mutex_init(&mx, NULL);
    for(long t=0;t<NUM;t++) pthread_create(&threads[t],NULL,work,NULL);
    for(long t=0;t<NUM;t++) pthread_join(threads[t], NULL);
    pthread_exit(NULL);
}
```



Compilers do the darnest things

409

```
char c[10000];
int a=2,b=4; //volatile

int main() {
    for(int i=0;i<1000000000;i++){
        int x=a;
        int y=b;
        c[i%10000]=a*b;
        if(a-(2*b)) { a=b; b=a; } //??
    }
}
```

Algorithms rely memory models

410

Thread1:

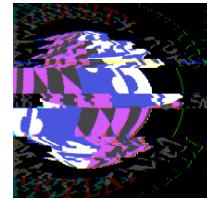
```
done = false;  
accountA -= amount;  
accountB += amount;  
done = true;
```

Thread2:

```
while (!done);  
print( accountA + accountB );
```



the
POWER
of
JAVA™



PURDUE
UNIVERSITY™

JavaOne
Sanjay Kulkarni and Pradeep Verma

The Java™ Memory Model: the building block of concurrency

Jeremy Manson, Purdue University

William Pugh, Univ. of Maryland

<http://www.cs.umd.edu/~pugh/java/memoryModel/>



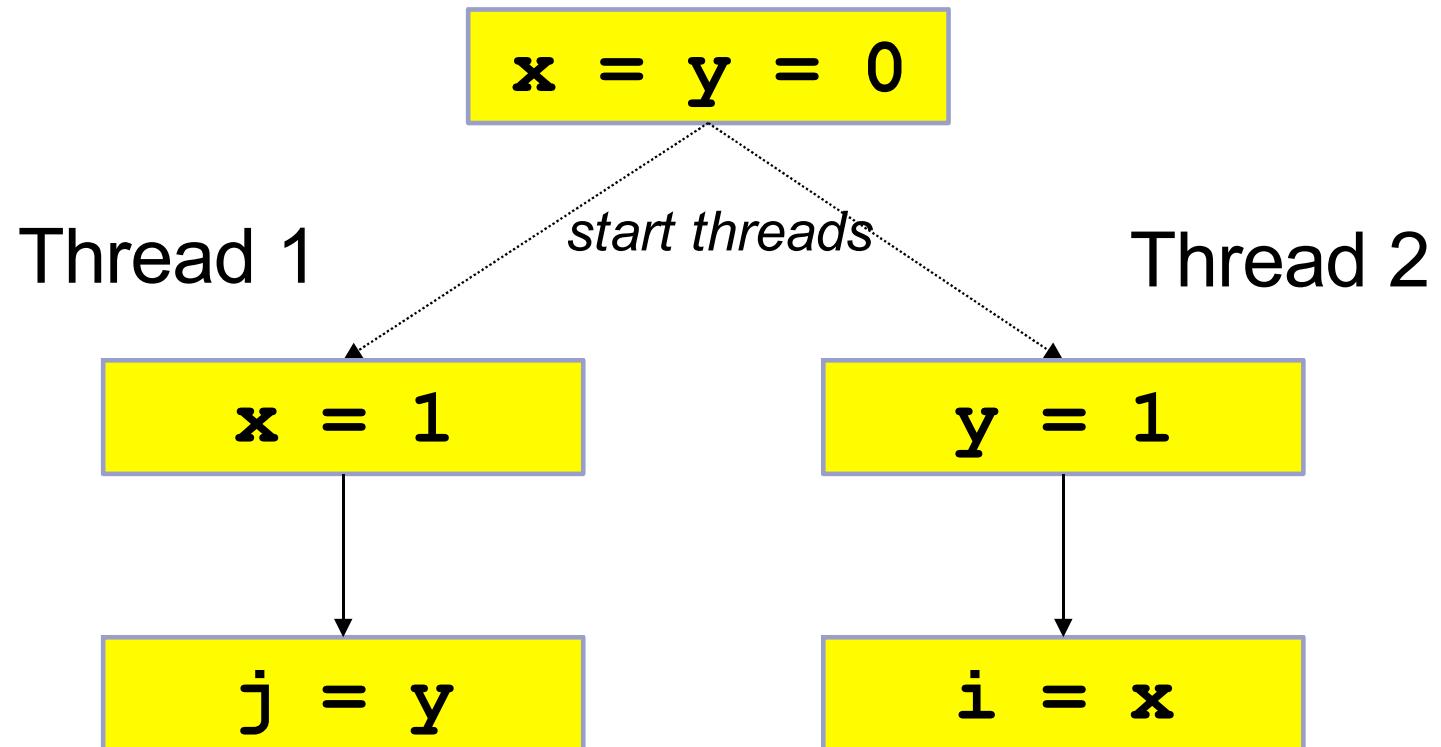
Synchronization is needed for Blocking and Visibility

- Synchronization isn't just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
 - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
 - And break your code

Don't Try To Be Too Clever

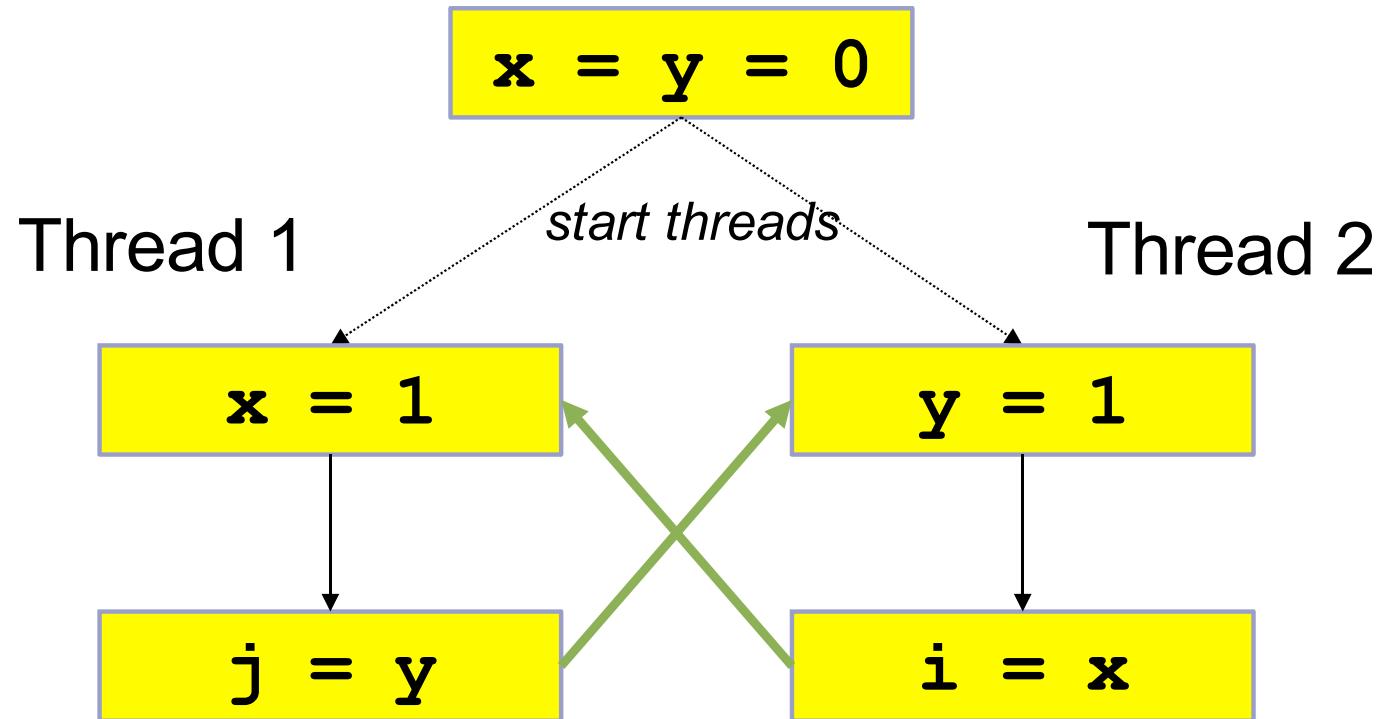
- People worry about the cost of synchronization
 - Try to devise schemes to communicate between threads without using synchronization
 - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
 - Inter-thread communication without synchronization is not intuitive

Quiz Time



Can this result in $i = 0$ and $j = 0$?

Answer: Yes!

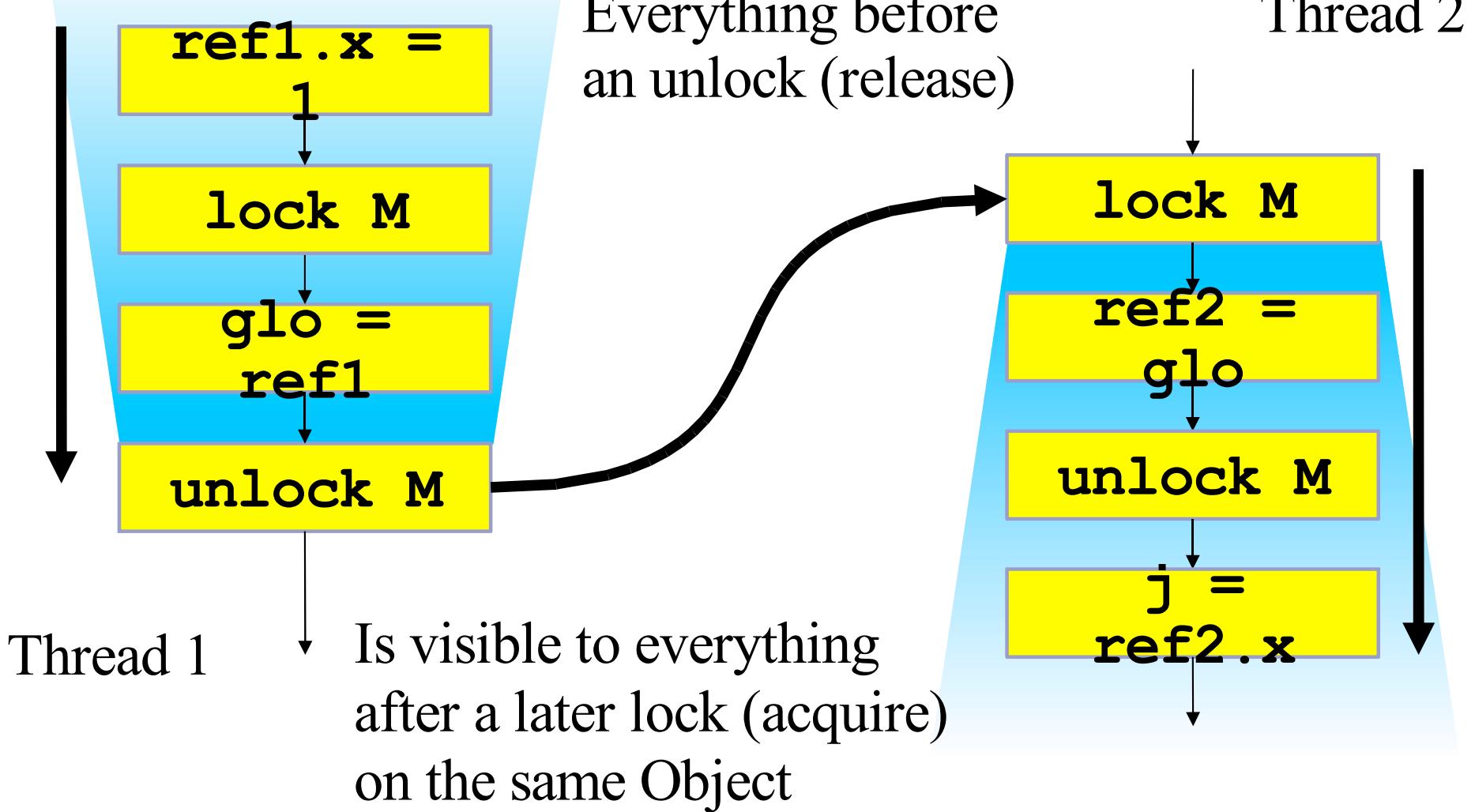


How can $i = 0$ and $j = 0$?

How Can This Happen?

- Compiler can reorder statements
 - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
 - including optimizations no one has implemented yet
- Good for performance
 - bad for your intuition about insufficiently synchronized code

When Are Actions Visible to Other Threads?



Release and Acquire

- All memory accesses before a release
 - are ordered before and visible to
 - any memory accesses after a matching acquire
- Unlocking a monitor/lock is a release
 - that is acquired by any following lock of *that* monitor/lock

Happens-before ordering

- A release and a matching later acquire establish a *happens-before* ordering
- execution order within a thread also establishes a happens-before order
- happens-before order is transitive

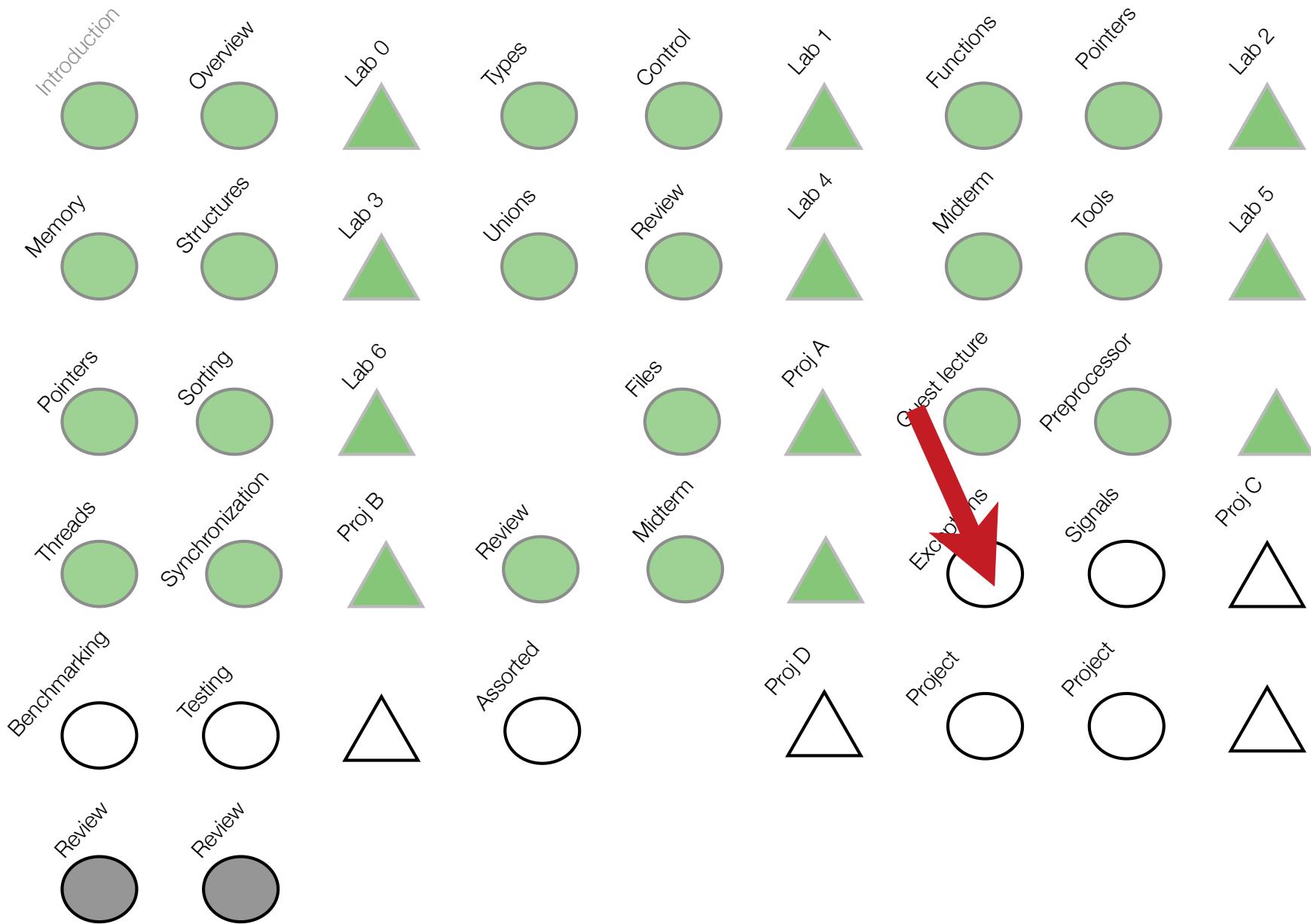
Data race

- If there are two accesses to a memory location,
 - at least one of those accesses is a write, and
 - the memory location isn't volatile, then
- the accesses *must* be ordered by happens-before
- Violate this, and you may need a PhD to figure out what your program can do
 - not as bad/unspecified as a buffer overflow in C

{Loc}

Lecture 16





Exceptions

423

An exception is a control-flow mechanism that allows unexpected conditions to be caught and handled

Key property: the handler for an exception need not be lexically apparent at the point where the exception is raised

Concept: *lexical* vs. *dynamic* scope

The Call Stack

424

Suppose function *f* calls *g* which then calls *h*

provide a handler for exception E



raise exception E:

control directly reverts to handler in f

In Java...

425

```
double divide(int to, int by) throws Bad {  
    if(by == 0) throw new Bad("Cannot / 0");  
    return to / by;  
}  
  
void f(){  
    try {  
        g(2,1);  
        ...  
    } catch (Bad e){ print(e.getMessage()); }  
    print("done");  
}  
  
void g(int x, y){  
    ...  
    r = divide(x,y);  
    ...  
}
```

In C?

426

No primitive exception support in C

Instead C provides two operations for non-local control-flow

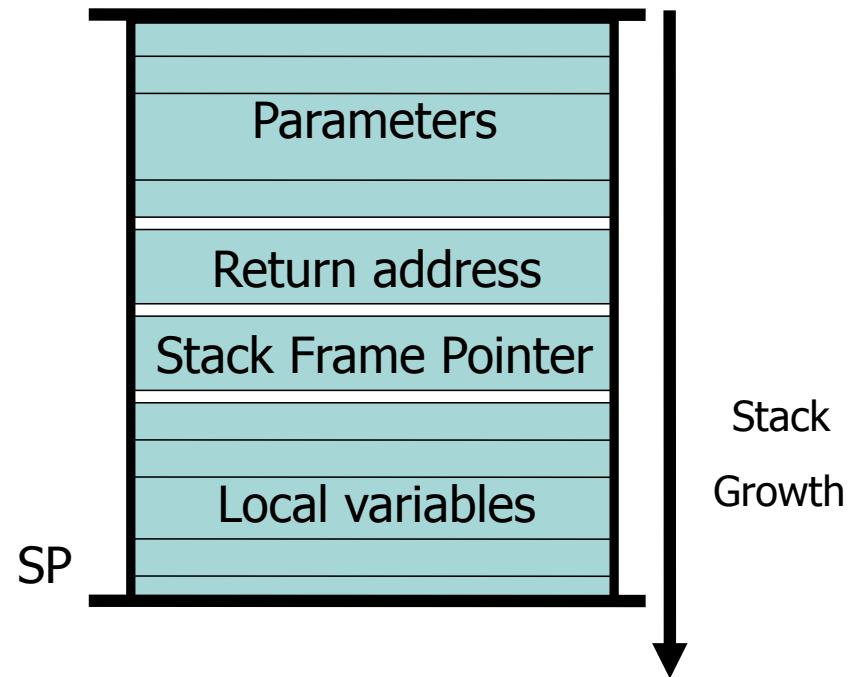
- ▶ save the set of all registers (`setjmp`)
- ▶ jump to the saved set

Since the set of registers includes the stack pointer, frame pointer, and program counter, we have a way of recovering back to any point in the execution.

Stack frame

427

- Parameters for the procedure
- Save current PC onto stack (return address)
- Save current SP value onto stack
- Allocates stack space for local variables by decrementing SP by appropriate amount



setjmp

428

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

setjmp expects an argument of type jmp_buf

```
#define JBLEN 9
typedef struct { int jb[JBLEN + 1]; } jmp_buf[1];
```

Objects of type jmp_buf are defined to be arrays of length JBLEN+1. Each element in this array will hold the contents of a specific register stored by setjmp

longjmp

429

```
longjmp(jmp_buf env, int val);
```

Restores the set of registers stored in env. But, this includes the program counter (pc). This means `longjmp` does not return!

The PC saved by `setjmp` points to the instruction that expects `setjmp`'s return value. The second argument supplied to `longjmp` provides this value

Example

430

```
#include < setjmp.h >
main() {
    jmp_buf env;
    int i;
    i = setjmp(env);
    printf("i = %d\n", i);
    if (i != 0) exit(0);
    longjmp(env, 2);
    printf("This line does not get printed\n");
}
```

Records the state at this point.
Specifically, the PC points here
By default, setjmp returns 0

When we call longjmp, control jumps back to the point where
the last setjmp was called,
supplying 2 as the value
assigned to i.

When run this prints:

i = 0
i = 2

Example

431

```
jump_buf jumper;

int fun(int a, int b) {
    if (b == 0) // can't divide by 0
        longjmp(jumper, -3);
    return a / b;
}

void main(void) {
    if (setjmp(jumper) == 0) {
        int r = fun(7, 0);
        // continue working with result
    } else printf("error\n");
}
```

Example

432

```
#include <setjmp.h>
#include <stdio.h>
int a(char *s, jmp_buf env) {
    int i = setjmp(env);
    printf("Setjmp returned %d\n", i); printf("%s\n", s);
    return i;
}

int b(int i, jmp_buf env) {
    printf("B:%d\n", i);
    longjmp(env, i);
}

int main() {
    jmp_buf env;
    if (a("Me", env) != 0) exit(0);
    b(3, env);
}
```

Example

from wikipedia

433

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>
void first(void); void second(void);
/* This program's output is: calling first; calling second; entering second; second
failed with type 3 exception; remapping to type 1.; first failed, exception type 1 */

jmp_buf env; int type;

int main() {
    void *volatile buf = NULL;
    if (setjmp(env)) {
        printf("first failed %d\n", type);
    } else {
        printf("calling first\n");
        first();
        buf = malloc(300); /* allocate a resource */
        printf(strcpy((char*) buf, "first succeeded!"));
    }
    if (buf) free(buf);
}
```

Example

434

```
void first(void) {
    jmp_buf my;
    memcpy(my, env, sizeof(jmp_buf));
    switch (setjmp(env)) {
        case 3:
            printf("failed with type 3 exception; remapping to type 1.\n");
            type = 1;
        default:
            memcpy(env, my, sizeof(jmp_buf)); /*restore exception stack*/
            longjmp(env, type); /*continue handling exception*/
        case 0: /* normal, desired operation */
            second();
    }
    memcpy(env, my, sizeof(jmp_buf)); /*restore exception stack*/
}

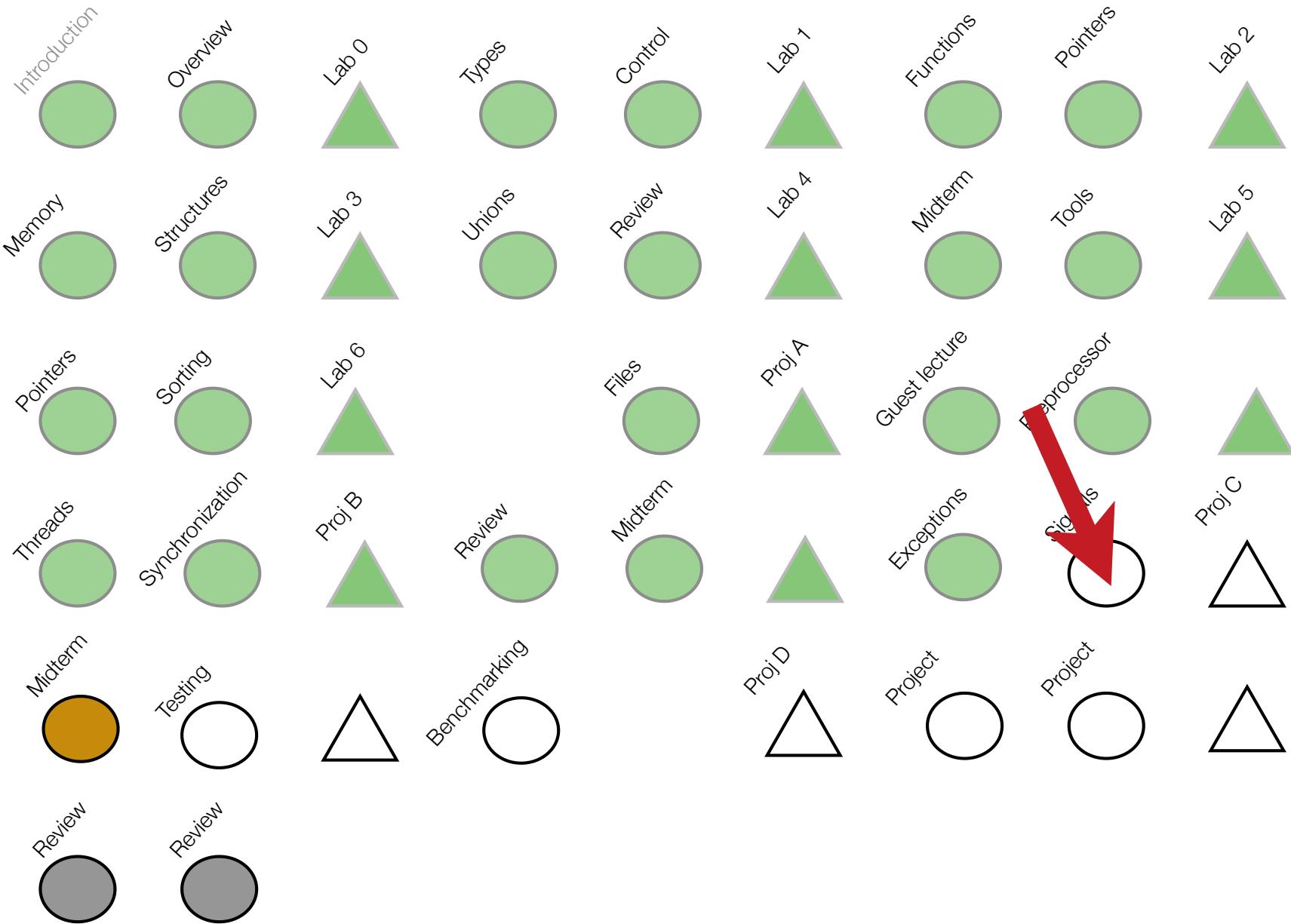
void second(void) {
    printf("entering second\n");
    longjmp(env, type = 3); /*declare that program has failed*/
    printf("leaving second\n"); /* not reached */
}
```

Lecture 17

Lecture 17

Signals





Synchronous v Asynchronous

437

Synchronous action:

- ▶ A procedure call: caller calls callee and waits until callee returns a result
A phone call: both parties must be available for the call to go through.

An asynchronous action:

- ▶ A response from the environment: user types Ctrl-C while a program runs
Email or text messages can be sent without the other party ready to receive

Dealing with Asynchronous Actions

438

Poll:

- ▶ ask the kernel or the OS: did event E take place since the last time I checked?

Handle:

- ▶ inform the kernel that when event E takes place, do the following ...

Kinds of Signals

439

Interrupts

- ▶ Environment-triggered (SIGINT, Ctrl-C)

Hardware

- ▶ (SIGSEGV); divide by 0, invalid memory reference

Software

- ▶ (SIGPIPE, SIGALRM). Timeout on network connection, a broken pipe, ...

Handling a Signal

440

Ignore it

- ▶ Doesn't always work though (e.g., don't ignore a hardware exception...)

Catch it

- ▶ setup a signal handler that gets invoked whenever the signal occurs

All signals have a default action

- ▶ Most of the time, the default is to kill the process

Generating a Signal

441

```
#include <signal.h>
int kill(pid_t pid, int sig);
/* send signal 'sig' to process 'pid' */
```

`kill` is the most common case

`raise(sig)` generates a signal handled by the program that contains the call to `raise`

signal.h

442

SIGABRT

Abnormal termination, such as instigated by the abort function (Abort)

SIGFPE

Erroneous arithmetic operation, such as divide by 0 or overflow (Floating point exception)

SIGILL

An 'invalid object program' has been detected. This usually means that there is an illegal instruction in the program (Illegal instruction)

SIGINT

Interactive attention signal; on interactive systems this is usually generated by typing some 'break-in' key at the terminal (Interrupt)

SIGSEGV

Invalid storage access; most frequently caused by attempting to store some value in an object pointed to by a bad pointer (Segment violation)

SIGTERM

Termination request made to the program (Terminate)

Example

443

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

long prev, i;
void SIGhandler(int);

void SIGhandler(int sig) {
    printf("\nGot SIGUSR1. %ld!=%ld\n", i-1, prev);
    exit(0);
}

void main(void) {
    long fact;
    signal(SIGUSR1, SIGhandler);
    for (prev = i = 1; ; i++, prev = fact) {
        fact = prev*i;
        if (fact < 0) raise(SIGUSR1);
        else if (i % 3 == 0)
            printf("      %ld! = %ld\n", i, fact);
    }
}
```

Defining a signal handler

444

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

signal is a function pointer to a function that takes as arguments:

- ▶ a signal (represented as an **int**)
- ▶ a handler

and returns a function that takes an **int** and returns **void**

The handler is a function pointer to a function that takes an **int** and returns **void**.

Signal handler

445

signal installs a new handler for the supplied signal

It returns the previous value of the handler as its result

- ▶ If no such value exists, it returns **SIG_ERR** and sets **errno** appropriately

Example

446

```
static void sig_usr(int); /* one handler for two signals */
int main (void) {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR2");
    for(;;) pause();
}

static void sig_usr(int signo) {
/*argument is signal number*/
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else printf("received signal %d\n", signo);
    return;
}
```

Example

447

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

FILE *temp_file;
void leave(int sig);

int main() {
    signal(SIGINT, leave);
    temp_file = fopen("tmp", "w");
    for(;;) { printf("Ready...\n"); getchar(); }
    /* can't get here ... */
    exit(EXIT_SUCCESS);
}

/* on receipt of SIGINT, close tmp file */
void leave(int sig) {
    fprintf (temp_file, "\nInterrupted.");
    fclose(temp_file);
    exit(sig);
}
```

Example using setjmp/longjmp

448

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
int i, j; long T0; jmp_buf env;

void alarm_hdlr(int x) {
    long t1 = time(0) - T0;
    printf("%ld sec%s passed: j=%d.i=%d\n", t1, (t1==1)? "" : "s", j, i);
    if (t1 >= 8) { printf("Giving up\n"); longjmp(env, 1); }
    alarm(1);
}

int main() {
    signal(SIGALRM, alarm_hdlr); alarm(1);
    if (setjmp(env) != 0){
        printf("Gave up: j=%d,i=%d\n", j, i); exit(1);
    }
    T0 = time(0);
    for (j = 0; j < 10000; j++) for (i = 0; i < 1000000; i++);
}
```

{Loc}

Lecture 18

Testing

Terminology

450

Program

- ▶ Sort

Specification

- ▶ **Input:** p - array of n integers, $n > 0$
- ▶ **Output:** q - array of n integers such that
 $q[0] \leq q[1] \leq \dots \leq q[n]$
Elements in q are a permutation of elements in p, which are unchanged
- ▶ **Description of requirements of a program**

Tests

451

Test case

- ▶ A set of values given as input to a program
- ▶ Includes environment variables
- ▶ {2, 3, 6, 5, 4}

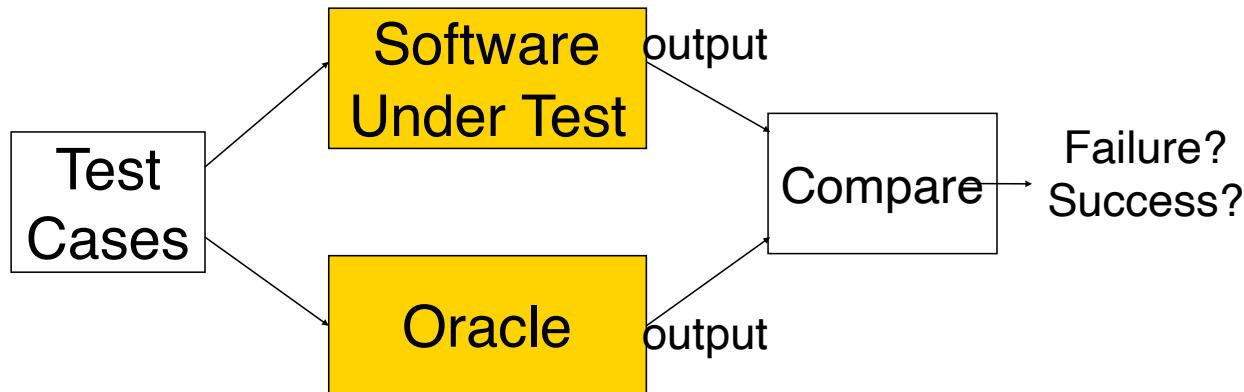
Test set

- ▶ A set of test cases
- ▶ { {0}, {9, 8, 7, 6, 5}, {1, 3, 4, 5}, {2, 1, 2, 3} }

Oracle

452

Function that determines whether or not the results of executing a program under test is as per the program's specifications



Problems

- Correctness of Oracle
- Correctness of specs
- Generation of Oracle
- Need more formal specs

Correctness

453

Program Correctness

- A program P is considered with respect to a specification S , if and only if:
For each valid input, the output of P is in accordance with the specification S

What if the specifications are themselves incorrect?

Errors, faults, failure

454

Error:

- ▶ Mistake made by programmer.
- ▶ Human action that results in the software containing a fault/defect.

Fault:

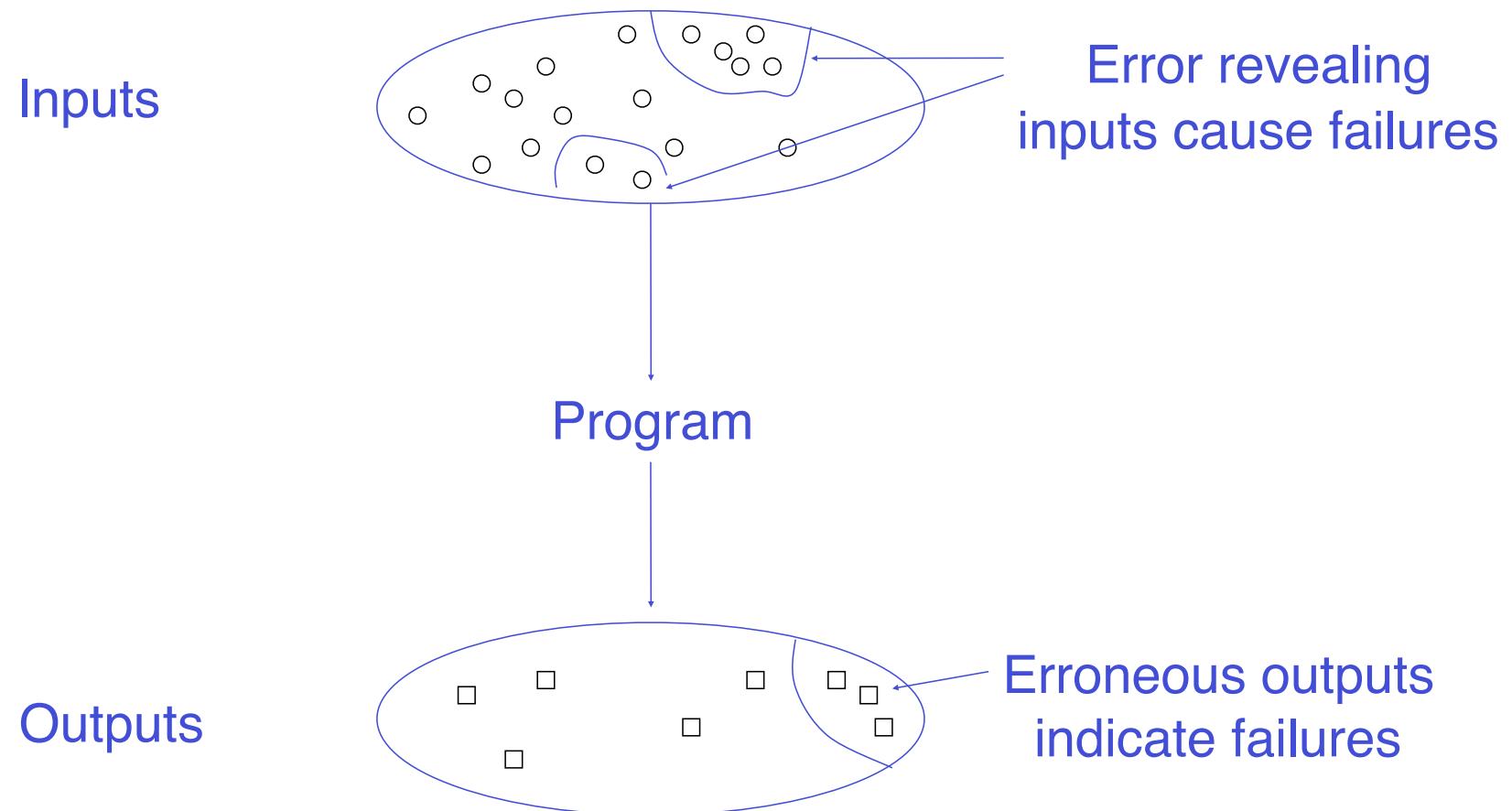
- ▶ Manifestation of the error in a program.
- ▶ Condition that causes the system to fail.
- ▶ Synonymous with **bug**

Failure

- ▶ Incorrect program behavior due to a fault in the program.
- ▶ Failure can be determined only with respect to a set of requirement specs.
- ▶ For failure to occur, the testing of the program should force the erroneous portion of the program to be executed.

Errors and failure

455



Nonexecution-Based Testing

456

Person creating a product should not be the only one responsible for reviewing it.

A document is checked by a team of software professionals with a range of skills.

Increases chances of finding a fault.

Types of reviews

- ▶ Walkthroughs
- ▶ Inspections

Execution-based testing

457

Execution-based testing is a process of **inferring** certain behavioral properties of a program, based, in part, on the results of executing the program in a **known environment with selected inputs**.

Depends on environment and inputs

- ▶ How well do we know the environment?
- ▶ How much control do we have over test inputs?

Levels of testing

458

Unit testing

Integration testing

System testing

Acceptance testing

The methodology

459

The derivation of test inputs is based on *program specifications*.

Clues are obtained from the specifications.

Clues lead to *test requirements*.

Test requirements lead to *test specifications*.

Test specifications are then used to actually execute the program under test.

Specifications-continued

460

Two types of pre-conditions are considered:

- ▶ **Validated:** those that are required to be validated by the program under test and an error action is required to be performed if the condition is not true.
- ▶ **Assumed:** those that are assumed to be true and not checked by the program under test.

Preconditions for sort

461

Validated:

$N > 0$

On failure return -1; sorting considered unsuccessful.

Assumed:

The input sequence contains N integers.

The output area has space for at least N integers.

Post-conditions

462

A post-condition specifies a property of the output of a program.

Example:

For the sort program a post-condition is:

if N>0 then {the output sequence has the same elements as in the input sequence and in ascending order.}

Incompleteness of specifications

463

Specifications may be incomplete or ambiguous.

Example post-condition:

if user places cursor on the name field then read a string

This post-condition does not specify any limit on the length of the input string hence is incomplete.

- It also does not make it clear as to
 - whether a string should be input only after the user has placed the cursor on the name field and clicked the mouse or simply placed the cursor on the name field.
- and hence is ambiguous.

Clues: summary

464

Clues are:

Pre-conditions

Post-conditions

Variables,

e.g. A is a length implying thereby that its value cannot be negative.

Operations,

e.g. "search a list of names" or "find the average of total scores"

Definitions,

e.g. "filename(name) is a name with no spaces."

- ▶ Ideally variables, operations and definitions should be a part of at least one pre- or post-condition.
- ▶ However, this may not be the case as specifications are not always written formally.

Equivalence partitioning

465

Why?

- ▶ Input domain is usually too large (e.g. infinite) for exhaustive testing.

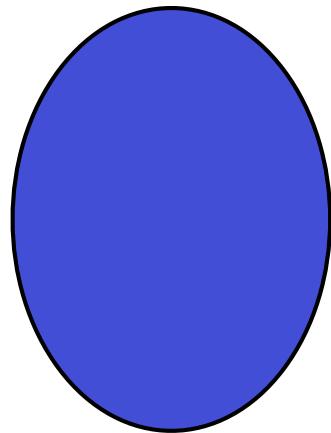
How?

- ▶ Partition into a finite number of sub-domains and select test inputs.
- ▶ Each sub-domain is an equivalence class and serves as a source of at least one test input.

Equivalence partitioning

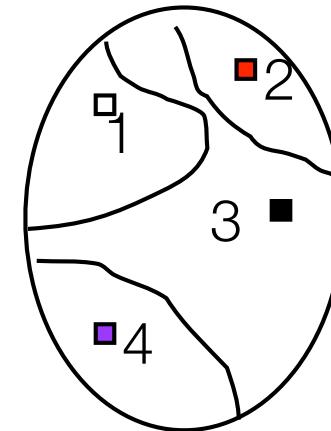
466

Input domain



Too many
test inputs.

Input domain
partitioned into four
sub-domains.



Four test inputs, one
selected from each
sub-domain.

How to partition?

467

Inputs to a program provide clues to partitioning.

Example:

- ▶ Suppose that program P takes an integer input X
- ▶ For $X < 0$ perform task T1 and
- ▶ for $X \geq 0$ perform task T2.

How to partition?

468

The input domain is prohibitively large as X can assume the whole range of integer values.

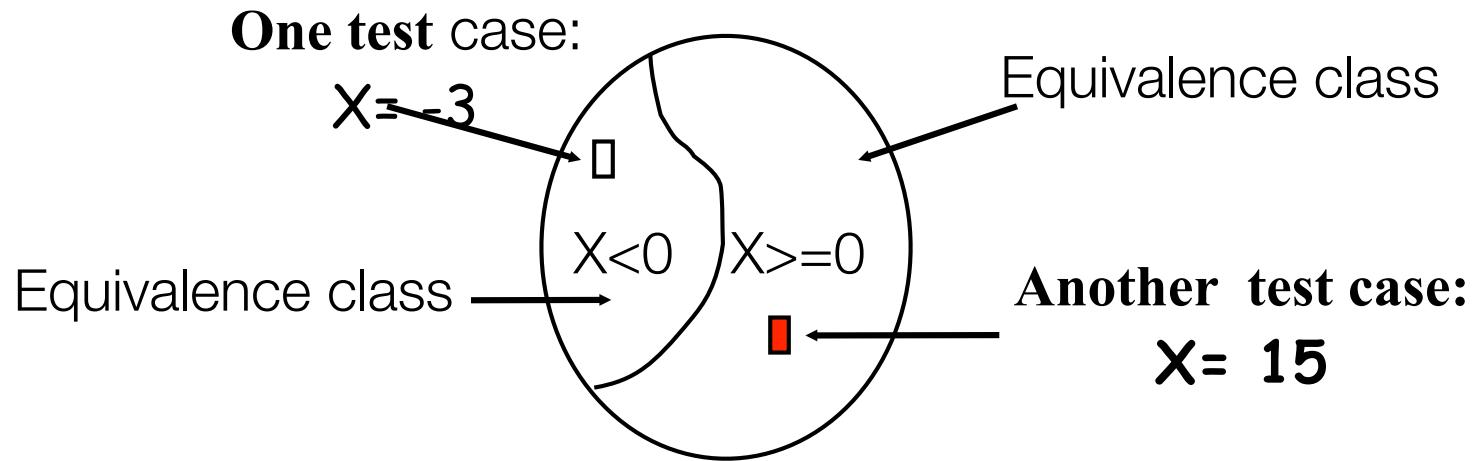
However, we expect P to behave the same way for all $X < 0$.

Similarly, we expect P to perform the same way for all values of $X \geq 0$.

We therefore partition the input domain of P into two sub-domains.

Two sub-domains

469



All test inputs in the $X < 0$ sub-domain are considered equivalent.
The assumption is that if one test input in this sub-domain reveals
an error in the program, so will the others.

This is true of the test inputs in the $X \geq 0$ sub-domain also.

Non-overlapping partitions

470

In the previous example, the two equivalence classes are non-overlapping. In other words the two sub-domains are disjoint.

When the sub-domains are disjoint, it is sufficient to pick one test input from each equivalence class to test the program.

An equivalence class is considered **covered** when at least one test has been selected from it.

In partition testing our goal is to cover all equivalence classes.

Overlapping partitions

471

Example:

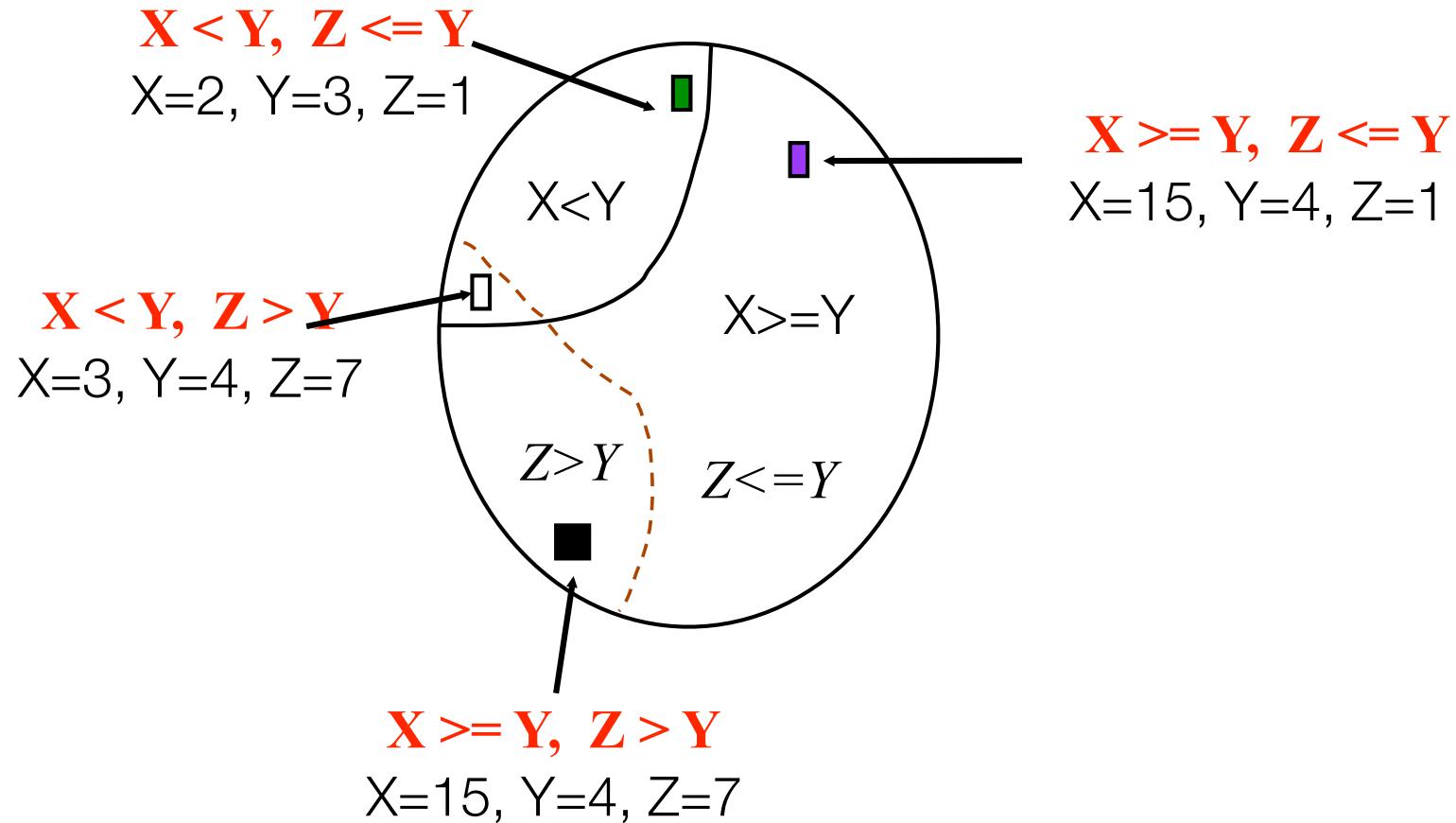
- ▶ Suppose that program P takes three integers X, Y and Z.
- ▶ It is known that:

$$X < Y$$

$$Z > Y$$

Overlapping partitions

472



Overlapping partition-test selection

473

In this example, we could select 4 test cases as:

- ▶ $X=4, Y=7, Z=1$ satisfies $X < Y$
- ▶ $X=4, Y=2, Z=1$ satisfies $X \geq Y$
- ▶ $X=1, Y=7, Z=9$ satisfies $Z > Y$
- ▶ $X=1, Y=7, Z=2$ satisfies $Z \leq Y$

Thus, we have one test case from each equivalence class.

Overlapping partition-test selection

474

However, we may also select only 2 test inputs and satisfy all four equivalence classes:

- ▶ $X=4, Y=7, Z=1$ satisfies $X < Y$ and $Z \leq Y$
- ▶ $X=4, Y=2, Z=3$ satisfies $X \geq Y$ and $Z > Y$

Thus, we have reduced the number of test cases from 4 to 2 while covering each equivalence class.

Partitioning using non-numeric data

475

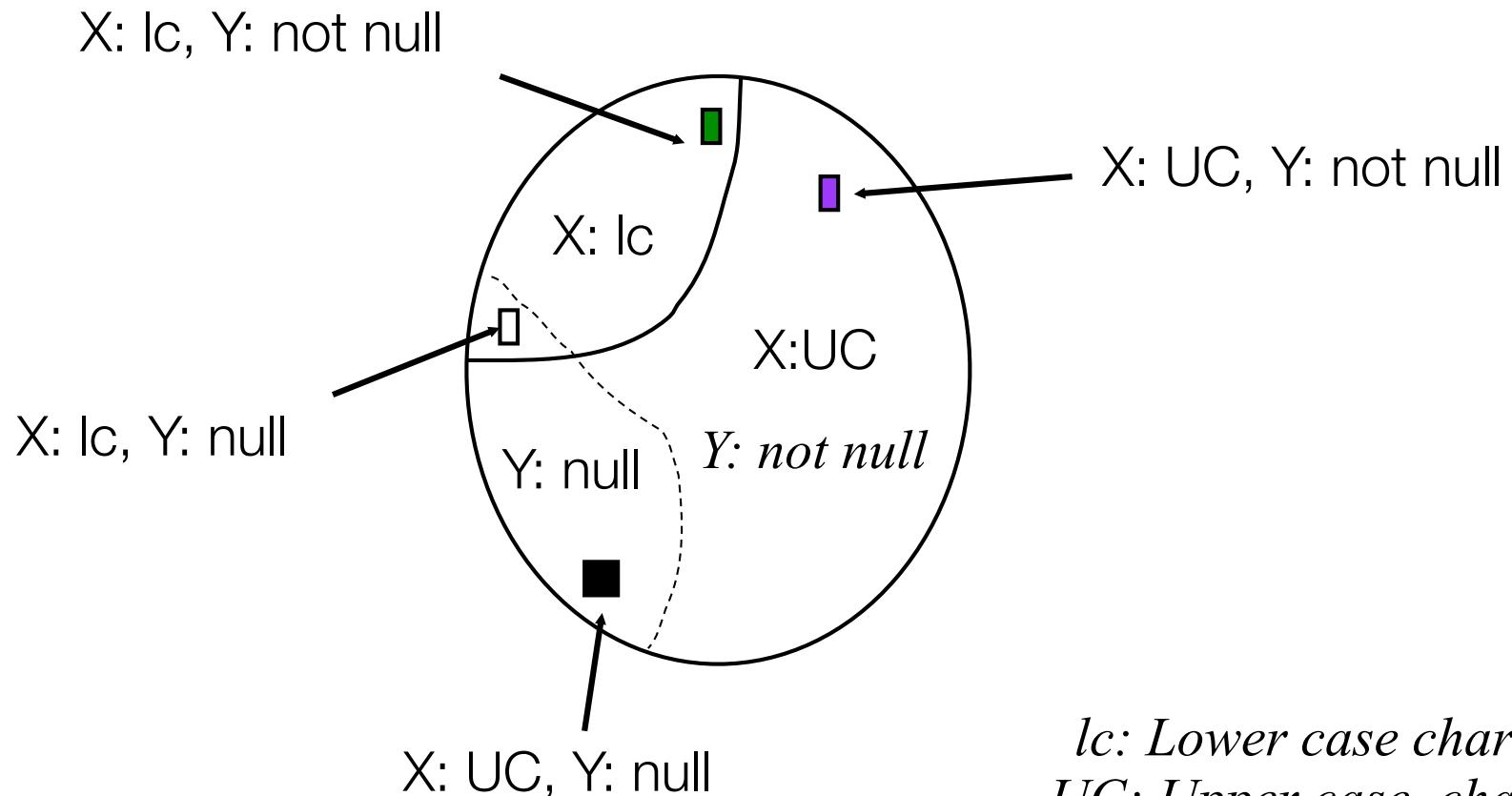
In the previous two examples the inputs were integers. One can derive equivalence classes for other types of data also.

Example 3:

- ▶ Suppose that program P takes one character X and one string Y as inputs.
- ▶ P performs task T1 for all lower case characters and T2 for upper case characters.
- ▶ Also, it performs task T3 for the null string and T4 for all other strings.

Partitioning using non-numeric data

476



*lc: Lower case character
UC: Upper case character
null: null string.*

Non-numeric data

477

Once again we have overlapping partitions.

We can select only 2 test inputs to cover all four equivalence classes. These are:

- ▶ X: lower case, Y: null string
- ▶ X: upper case, Y: not a null string

Guidelines for equivalence partitioning

478

Input condition specifies a range: create one for the valid case and two for the invalid cases.

- ▶ e.g. for $a \leq X \leq b$ the classes are

$A \leq X \leq b$ (valid case)

$X < a$ and $X > b$ (the invalid cases)

Input condition specifies a value: create one for the valid value and two for incorrect values (below and above the valid value). This may not be possible for certain data types, e.g. for boolean.

Input condition specifies a member of a set: create one for the valid value and one for the invalid (not in the set) value.

Sufficiency of partitions

479

In the previous examples we derived equivalence classes based on the conditions satisfied by the input data.

Then we selected just enough tests to cover each partition.

Think of the advantages and disadvantages of this approach!

Boundary value analysis (BVA)

480

Another way to generate test cases is to look for boundary values.

Suppose a program takes an integer X as input.

In the absence of any information, we assume that $X = 0$ is a boundary. Inputs to the program might lie on the boundary or on either side of the boundary.

BVA: continued

481

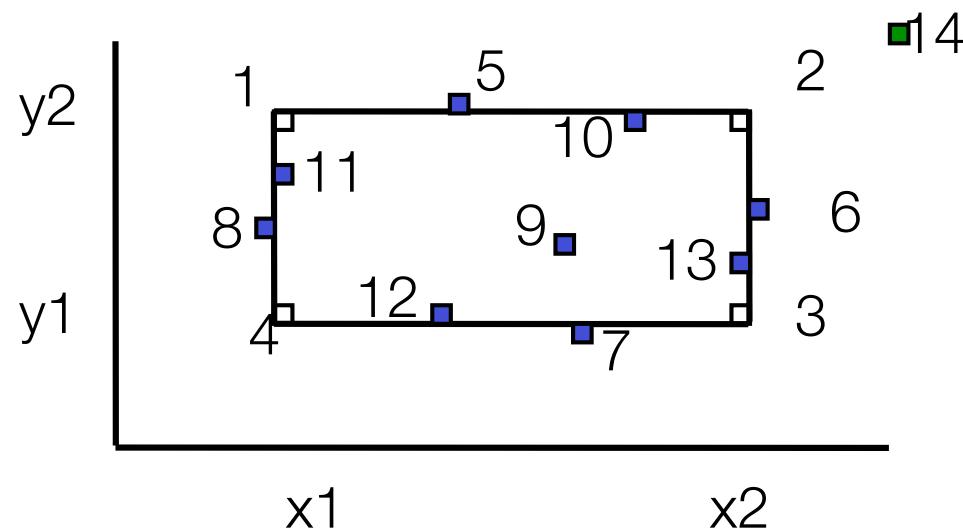
This leads to 3 test inputs:

$X = 0$, $X = -20$, and $X = 14$.

Note that the values -20 and 14 are on either side of the boundary and are chosen arbitrarily.

Notice that using BVA we get 3 equivalence classes. One of these three classes contains only one value ($X=0$), the other two are large!

Now suppose a program takes two integers X and Y and
▶ $x_1 \leq X \leq x_2$ and $y_1 \leq Y \leq y_2$.



BVA-continued

483

In this case the four sides of the rectangle represent the boundary.

The heuristic for test selection in this case is:

Select one test at each corner (1, 2, 3, 4).

Select one test just outside of each of the four sides of the boundary (5, 6, 7, 8)

BVA-continued

484

Select one test just inside of each of the four sides of the boundary (10, 11, 12, 13).

Select one test case inside of the bounded region (9).

Select one test case outside of the bounded region (14).

How many equivalence classes do we get?

BVA -continued

485

In the previous examples we considered only numeric data.
BVA can be done on any type of data.

For example, suppose that a program takes a string S and an integer X as inputs. The constraints on inputs are:

$$\text{length}(S) \leq 100 \text{ and } a \leq X \leq b$$

Can you derive the test cases using BVA?

BVA applied to output variables

486

Just as we applied BVA to input data, we can apply it to output data.

Doing so gives us equivalence classes for the output domain.

We then try to find test inputs that will cover each output equivalence class.

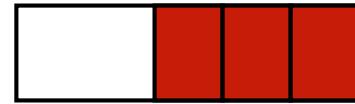
{Loc}

Lecture 19

Benchmarking and Performance

Q 11

Quiz #11 - 1



489

Do you want to have the final on Saturday as scheduled?

(a) yes

(b) no

Quiz 11 - 2

490

Is Thursday, 7:30 - 9:20 an option

- ▶ (a) yes
- ▶ (b) no

Quiz 11 - 3

491

Is Tuesday of finals week, 7:30 - 9:20, an option

- ▶ (a) yes
- ▶ (b) no

Monday

Optimization Rationales

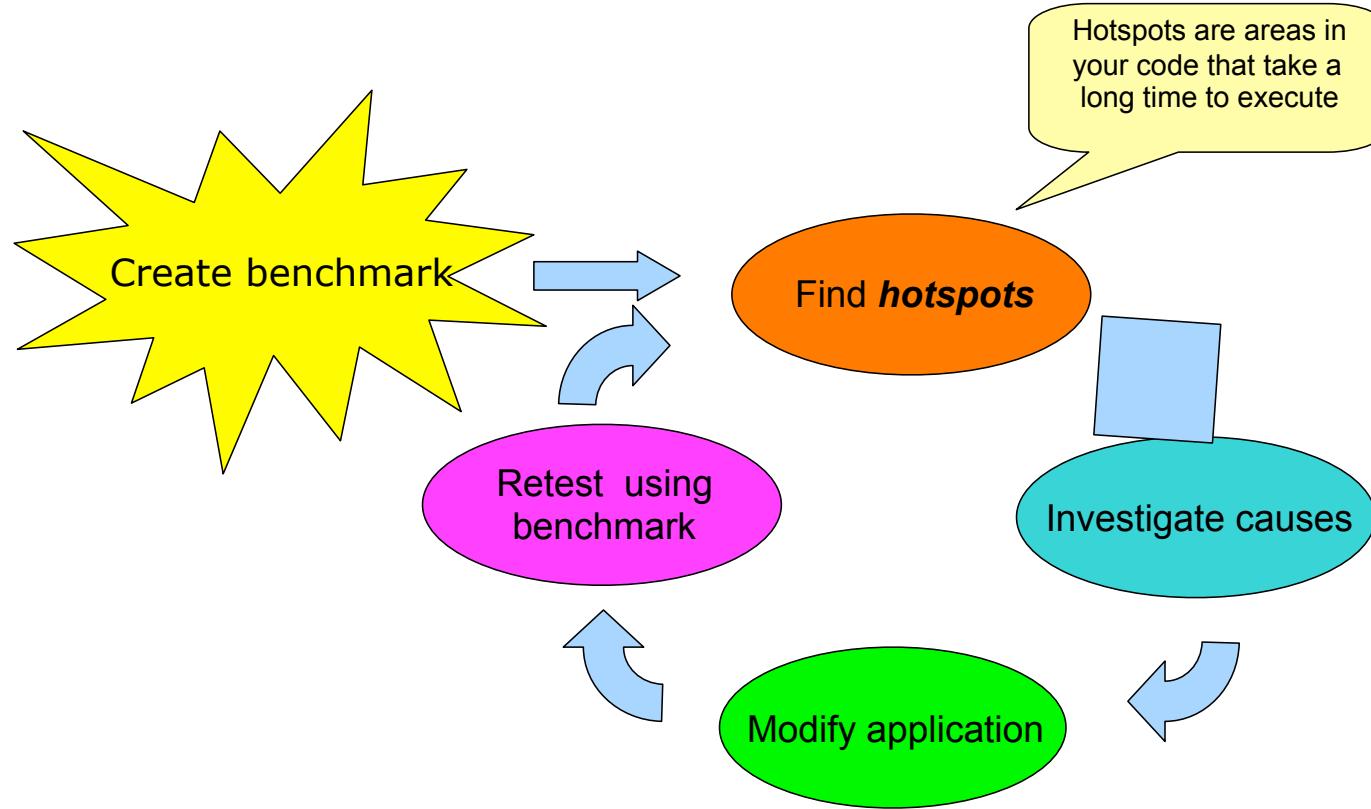
493

What are we optimizing?

- ▶ Optimization often degrades maintainability of the code
- ▶ Do not optimize, unless you have to
- ▶ Start by measuring performance
- ▶ Identify costs and optimize only when warranted by measurements
- ▶ Do not guess where the time goes, even experts can be wrong

Process

494



Some Optimization Challenges

495

- ▶ Must build the system first
- ▶ Must know on what hardware/software environment it will run (different from development environment)
- ▶ Use release builds (and not dev/debug)
- ▶ Measurements introduce noise (can we trust the tools)

- ▶ *Software optimization is an ongoing process that continues throughout the development of a system*

Benchmarks

496

Optimizations must be directed by measurements on benchmarks

Benchmarks should be:

- ▶ **Repeatable**

Must provide consistent measurements in the presence of other tasks in the system, of hardware variability, caching issues, et

- ▶ **Representative**

Must be typical of actual system load

Must exercise typical code paths (beware of slow paths and pathological cases)

- ▶ **Easy to run**

- ▶ **Verifiable**

Must have a checkable answer - to ensure that optimization do not break correctness

- ▶ **Deterministic**

Finding bottlenecks

497

Determine how system resources (memory and processor) are utilized to identify system-level bottlenecks

Measure the execution time for each component

Determine how the various components affect the performance of each other

Identify the most time-consuming code sequences

Determine how the program is executing at the processor level to identify microarchitecture-level performance problems

Tools overview

498

Timing mechanisms

- ▶ Stopwatch: time tool

System load monitors

- ▶ vmstat, iostat...

Software profiler

- ▶ Gprof...

Memory debugger/profiler

- ▶ Valgrind...

CPU load with top

499

top

- ▶ top gives an overview of processes running on the system
- ▶ A number of statistics provided covering CPU usage:

us:usermode

sy : system (kernel/)

ni:niced

id:idle

wa : I/O wait (blocking) – hi : irq handlers

si : softirq handlers

- ▶ Additionally, the following are displayed:

load average : 1,5,15-min load averages

system uptime info

total process counts (total|running|sleeping|stopped|zombie)

CPU load with top

500

Per-process stats

- ▶ PID:pid
- ▶ PR : priority
- ▶ NI : niceness
- ▶ TIME : total (system + user) spent since spent since startup
- ▶ COMMAND : command that started the process
- ▶ USER : username of running process
- ▶ VIRT : virtual image size
- ▶ RES : resident size
- ▶ SHR : shared mem size
- ▶ %CPU : CPU usage
- ▶ %MEM : Memory usage

CPU load: vmstat

501

vmstat

- ▶ Default output is averages
- ▶ Stats relevant to CPU:
 - in : interrupts
 - cs : context switches
 - us : total CPU in user (including "nice")
 - sy : total CPU in system (including irq and softirq) - wa : total CPU waiting
 - id : total CPU idle
- ▶ syntax: `vmstat [options] [delay [samples]]`

Profilers

502

Profiler show time elapsed in each function and its descendants

- ▶ number of calls, call-graph (some)

Profilers use instrumentation or sampling

	Sampling	Instrumentation
Overhead	Typically about 1%	High, may be 500% !
System-wide profiling	Yes, profiles all app, drivers, OS functions	Just application and instrumented DLLs
Detect unexpected events	Yes , can detect other programs using OS resources	No
Setup	None	Automatic ins. of data collection stubs required
Data collected	Counters, processor an OS state	Call graph , call times, critical path
Data granularity	Assembly level instr., with src line	Functions, sometimes statements
detects algorithmic issues	No, Limited to processes , threads	Yes – can see algorithm, call path is expensive

profiling with gprof

503

gprof

- ▶ the gnu profiler
 - ▶ requires instrumented binaries (compile time)
 - ▶ gprof used to parse output file
-
- ▶ gcov
coverage tool – uses the same instrumentation

Using *gprof* GNU profiler

504

Compile and link your program with profiling on

- ▶ `cc -g -c myprog.c utils.c -pg`
- ▶ `cc -o myprog myprog.o utils.o -pg`

Run your program to generate a profile data file

- ▶ Program will run normally (but slower) and will write the profile data into a file called `gmon.out` just before exiting
- ▶ Program should exit using `exit()` function

Run *gprof* to analyze the profile data

- ▶ `gprof a.out`

Example

505

```
#include <iostream>
#include <math.h>
#define NUM1 10000

void doit() { double x=0; for (int i=0;i<NUM1;i++) x+=sin(i);}

void f(){ for(int i=0;i<1000;i++) doit();}

void g(){ for(int i=0;i<5000;i++) doit();}

int main () {
    double s=0; for(int i=0;i< 1000*NUM1; i++) s+=sqrt(i);
    f();
    g();
    std::cout<<"Done "<<std::endl;
    exit (0);■
}
```

Understanding Flat Profile

506

The *flat profile* shows the total amount of time your program spent executing each function.

If a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called

Flat profile : %time

507

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
86.73	4.64	4.64	6000	0.00	0.00	doit()
13.27	5.35	0.71	1	0.71	5.35	main
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed to _Z4doitv
0.00	5.35	0.00	1	0.00	0.77	f()
0.00	5.35	0.00	1	0.00	3.87	g()
0.00	5.35	0.00	1			d_destruction_0(int, int)

Percentage of the total execution time your program spent in this function.
These should all add up to 100%.

Flat profile: Cumulative seconds

508

Each sample counts as 0.01 seconds.

%	cumulative	self				
time	seconds	seconds	calls	v.vv	v.vv	name()
86.73	4.64	4.64	6000	v.vv	v.vv	main()
13.27	5.35	0.71	1	0.71	5.35	main
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed to _Z4doitv
0.00	5.35	0.00	1	0.00	0.77	f()
0.00	5.35	0.00	1	0.00	3.87	g()
0.00	5.35	0.00	1	0.00	0.00	_static_initialization_and_destruction_0(int, int)

This is cumulative total number of seconds the spent in this functions, plus the time spent in all the functions above this one

Flat profile: Self seconds

509

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	time	function name
86.73	4.64	4.64	6000	0.00	0.00	doit()
13.27	5.35	0.71	1	0.71	5.35	main
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed to _Z4doitv
0.00	5.35	0.00	1	0.00	0.77	f()
0.00	5.35	0.00	1	0.00	3.87	g()
0.00	5.35	0.00	1	0.00	0.00	_static_initialization_and_destruction_0(int, int)

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self			total	Number of times was invoked
			calls	s/call	s/call		
86.73	4.64	4.64	6000	0.00	0.00	doit()	
13.27	5.35	0.71	1	0.71	5.35	main	
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed to _Z4doity	
0.00	5.35	0.00	1	0.00	0.77	f()	
0.00	5.35	0.00	1	0.00	3.87	g()	
0.00	5.35	0.00	1	0.00	0.00	_ static _ initialization_and_destruction_0(int, int)	

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total		
time	seconds	seconds	calls	s/call	s/call	name	
86.73	4.64	4.64	6000	0.00	0.00	doit()	
13.27	5.35	0.71	1	0.71	5.35	main	
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed	
to <code>_Z4doitv</code>							
0.00	5.35	0.00	1	0.00	0.77	f()	
0.00	5.35	0.00	1	0.00	3.87	g()	
0.00	5.35	0.00	1	0.00	0.00	<code>_static_initialization_and_destruction_0(int, int)</code>	

Average number of sec per call
Spent in this function alone

Average number of seconds spent
in this function and its descendants
per call

time	seconds	seconds	calls	self s/call	total s/call	name
86.73	4.64	4.64	6000	0.00	0.00	doit()
13.27	5.35	0.71	1	0.71	5.35	main
0.00	5.35	0.00	1	0.00	0.00	global constructors keyed to _Z4doitv
0.00	5.35	0.00	1	0.00	0.77	f()
0.00	5.35	0.00	1	0.00	3.87	g()
0.00	5.35	0.00	1	0.00	0.00	_static_initialization_ar d_destruction_0(int, int)

Call Graph : call tree of the program

		index	% time	self	children	called	name
[1]	100.0	0.71	4.64	0.71	4.64	1/1	_start [2]
		0.00	3.87	0.00	3.87	1/1	main [1]
		0.00	0.77	0.00	0.77	1/1	g() [4]
							f() [5]

[2]	100.0	0.00	5.35	0.00	5.35	1/1	<spontaneous>
		0.71	4.64	0.71	4.64	1/1	_start [2]
							main [1]

[3]	86.7	0.77	0.00	0.77	0.00	1000/6000	f() [5]
		3.87	0.00	3.87	0.00	5000/6000	g() [4]
		4.64	0.00	6000	0.00	6000	doit() [3]

[4]	72.3	0.00	3.87	0.00	3.87	1/1	main [1]
		3.87	0.00	3.87	0.00	1	g() [4]
		4.64	0.00	5000/6000	0.00	5000/6000	doit() [3]

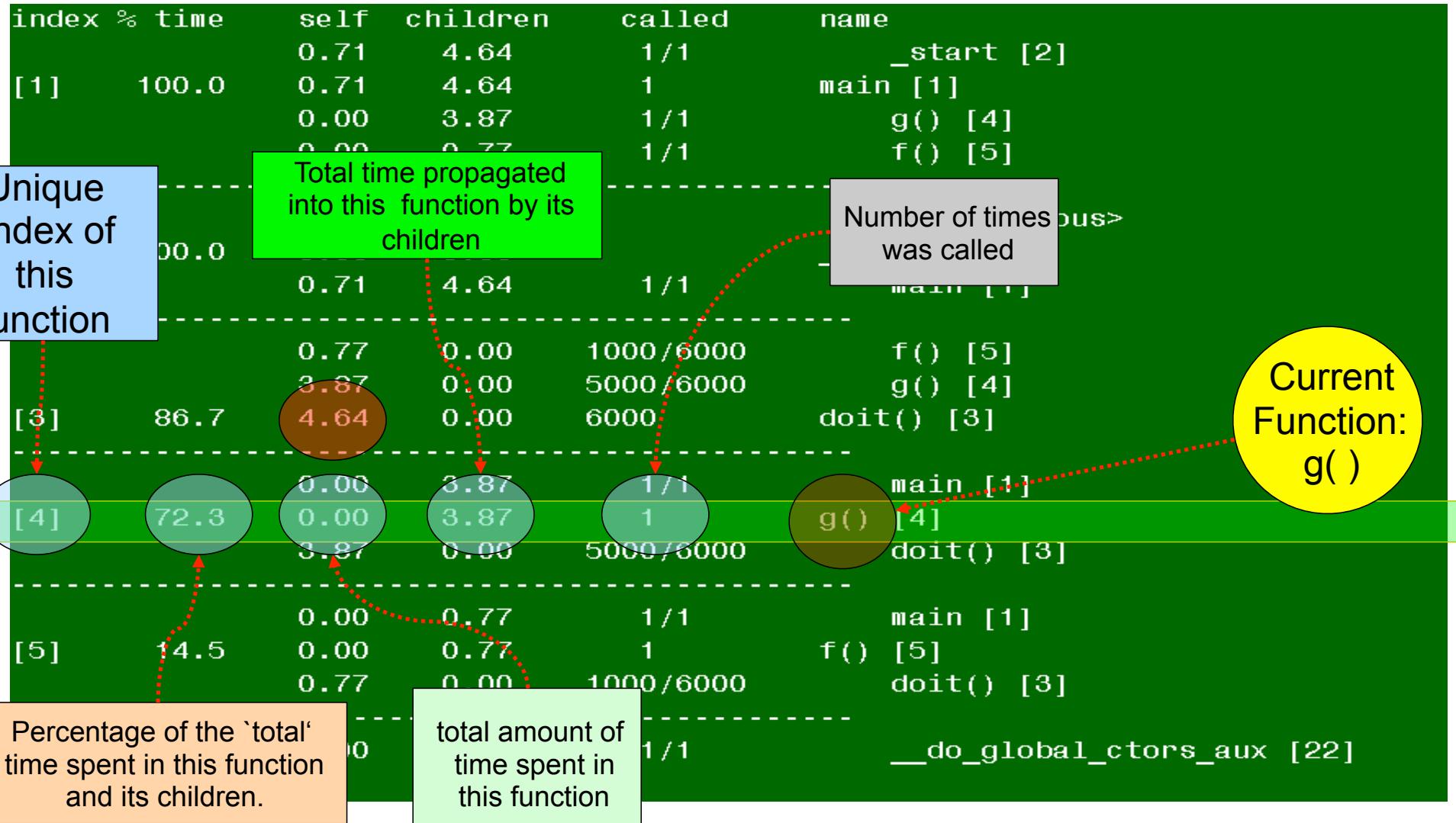
[5]	14.5	0.00	0.77	0.00	0.77	1/1	main [1]
		0.77	0.00	1	0.00	1000/6000	f() [5]
		0.00	0.00	1000/6000	0.00	1000/6000	doit() [3]

:		0.00	0.00	0.00	0.00	1/1	__do_global_c

Current Function:
g()

Called by :
main ()

Descendants:
doit ()



index % time self children called name

[1]	100.0	0.71	4.64	1/1	_start [2]
		0.71	4.64	1	main [1]
		0.00	3.87	1/1	g() [4]
		0.00	0.77		5]

[2]	100.0	0.00	5.35		taneous>
		0.71	4.64]
		0.77	0.00	1000/6000	f() [5]
		3.87	0.00	5000/6000	g() [4]
[3]	86.7	4.64	0.00	6000	doit() [3]

[4]	72.3	0.00	3.87	1/1	main [1]
		3.87	0.00	1	g() [4]
		3.87	3.87	5000/6000	doit() [3]

		0.00	0.77	1/1	main [1]
		0.00	0.77	1	f() [5]
		0.77	0.00	1000/6000	doit() [3]
		0.00	0.00	1/1	do_global_ctors_aux [22]

:					

The figure shows a call graph with nodes representing functions and edges representing calls. Nodes are colored by their current function: _start (light blue), main (light green), g() (orange), doit() (dark green), f() (yellow), and do_global_ctors_aux (purple). Edges are red dotted lines. A yellow circle labeled "Current Function: g()" highlights the orange node. Callouts provide details about the data in the table:

- Number of times this function called the child**: The count of calls from the current function to its children.
- total number of times this child was called**: The total count of calls to the current function from all its parents.
- Amount of time that was propagated directly from the child into function**: The time spent in the current function that was passed down from its children.
- Amount of time that was propagated from the child's children to the function**: The time spent in the current function that was passed up from its children's children.

How gprof works

516

Instruments program to count calls

Watches the program running, samples the PC every 0.01 sec

- ▶ Statistical inaccuracy: fast function may take 0 or 1 samples
- ▶ Run should be long enough comparing with sampling period
- ▶ Combine several gmon.out files into single report

The output from gprof gives no indication of parts of your program that are limited by I/O or swapping. Samples of the program counter are taken at fixed intervals of run time

number-of-calls figures are derived by counting, not sampling. They are accurate and do not vary if program is deterministic

Profiling with inlining and other optimizations needs care

Valgrind Toolkit

517

Memcheck is memory debugger

- ▶ detects memory-management problems

Cachegrind is a cache profiler

- ▶ performs detailed simulation of the L1, D1 and L2 caches in your CPU

Massif is a heap profiler

- ▶ performs detailed heap profiling by taking regular snapshots of a program's heap

Helgrind is a thread debugger

- ▶ finds data races in multithreaded programs

Memcheck Features

518

When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted

Memcheck can detect:

- ▶ Use of uninitialised memory
- ▶ Reading/writing memory after it has been free'd
- ▶ Reading/writing off the end of malloc'd blocks
- ▶ Reading/writing inappropriate areas on the stack
- ▶ Memory leaks – where pointers to malloc'd blocks are lost forever
- ▶ Passing of uninitialised and/or unaddressible memory to system calls
- ▶ Mismatched use of malloc/new/new [] vs free/delete/delete []
- ▶ Overlapping src and dst pointers in memcpy() and related functions
- ▶ Some misuses of the POSIX pthreads API

Memcheck Example

519

```
#include <iostream>

char * f() { char *cp=new char[17]; return cp; }

#define MM 100000
int main() {
    int *p= new int[10];
    p[10] = 6;

    int i,j;
    j= i+3;
    if (i>0) std::cout<<"Hi ";

    f();
    free (p);
    return 0;
}
```

Memcheck Example (Cont.)

520

Compile the program with -g flag:

- ▶ `g++ -c a.cc -g -o a.out`

Execute valgrind :

- ▶ `valgrind --tool=memcheck --leak-check=yes a.out > log`

View log

Memcheck report

Invalid write of size 4

at 0x80486CA: main (a.cc:8)

Address 0x1B92A050 is 0 bytes after a block of size 40 alloc'd

at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
by 0x80486BD: main (a.cc:7)

Conditional jump or move depends on uninitialized value(s)

at 0x80486DD: main (a.cc:12)

Mismatched free() / delete / delete []

at 0x1B904FA1: free (vg_replace_malloc.c:153)

by 0x8048703: main (a.cc:15)

Address 0x1B92A028 is 0 bytes inside a block of size 40 alloc'd

at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
by 0x80486BD: main (a.cc:7)

Memcheck report (cont.)

Leaks detected:

```
ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 15 from 1)
malloc/free: in use at exit: 17 bytes in 1 blocks.
malloc/free: 2 allocs, 1 frees, 57 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 1 not-freed blocks.
checked 2250336 bytes.
```

S
T
A
C
K

```
17 bytes in 1 blocks are definitely lost in loss record 1 of 1
at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
by 0x8048697: f() (a.cc:3)
by 0x80486F8: main (a.cc:14)
```

LEAK SUMMARY:

```
definitely lost: 17 bytes in 1 blocks.
```

Massif tool

523

Massif is a heap profiler - it measures how much heap memory programs use. It can give information about:

- ▶ Heap blocks
- ▶ Heap administration blocks
- ▶ Stack sizes

Help to reduce the amount of memory the program uses

- ▶ smaller programs interact better with caches, avoid paging

Detect leaks that aren't detected by traditional leak-checkers, such as Memcheck

- ▶ That's because the memory isn't ever actually lost - a pointer remains to it - but it's not in use anymore

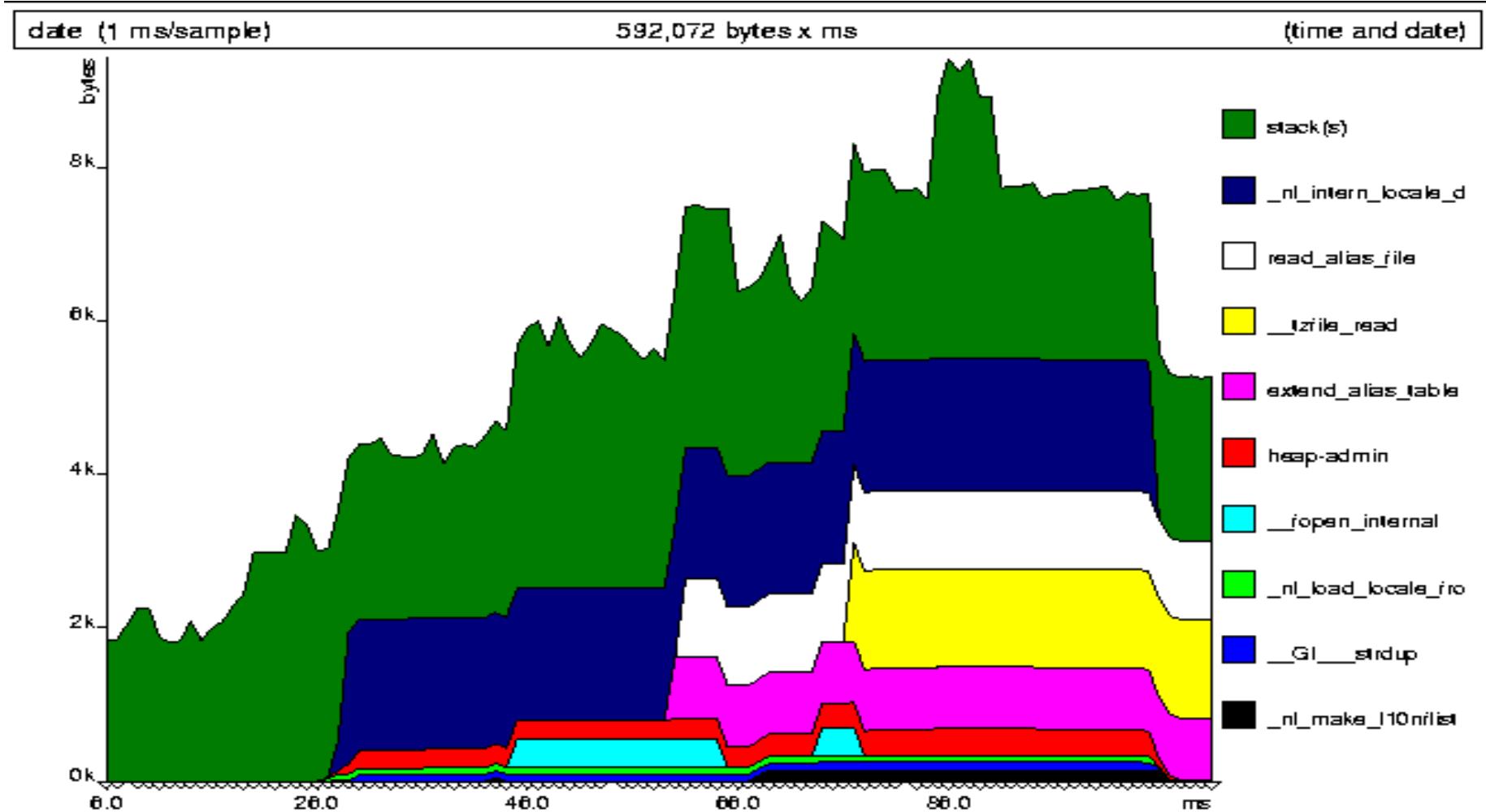
Executing Massif

- Run valgrind –tool=massif prog
 - Produces following:
 - Summary
 - Graph Picture
 - Report
- Summary will look like this:
 - Total spacetime: 2,258,106 ms.
 - Heap: 24.0%
 - Heap admin: 2.2%
 - Stack (s): 73.7%

Space (in bytes) multiplied by time (in milliseconds).

number of words allocated on heap, via malloc(), new and new[].

Spacetime Graphs



Spacetime Graph (Cont.)

526

Each band represents single line of source code

It's the height of a band that's important

Triangles on the x-axis show each point at which a memory census was taken

- ▶ Not necessarily evenly spread; Massif only takes a census when memory is allocated or de-allocated
- ▶ The time on the x-axis is wall-clock time
 - not ideal because can get different graphs for different executions of the same program, due to random OS delays

Text/HTML Report example

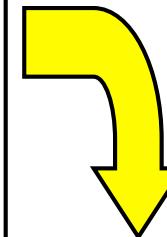
527

Contains a lot of extra information about heap allocations that you don't see in the graph.

Heap allocation functions accounted for 50.8% of measured spacetime

Called from:

- 22.1%: 0x401767D0: `_nl_intern_locale_data` (in `/lib/i686/libc-2.3.2.so`)
- 8.6%: 0x4017C393: `read_alias_file` (in `/lib/i686/libc-2.3.2.so`)
- (*several entries omitted*)
- and 6 other insignificant places



Shows places in the program where most memory was allocated

Context accounted for 22.1% of measured spacetime
0x401767D0: `_nl_intern_locale_data` (in `/lib/i686/libc-2.3.2.so`)

Called from:

- 22.1%: 0x40176F95: `_nl_load_locale_from_archive` (in `/lib/i686/libc-2.3.2.so`)