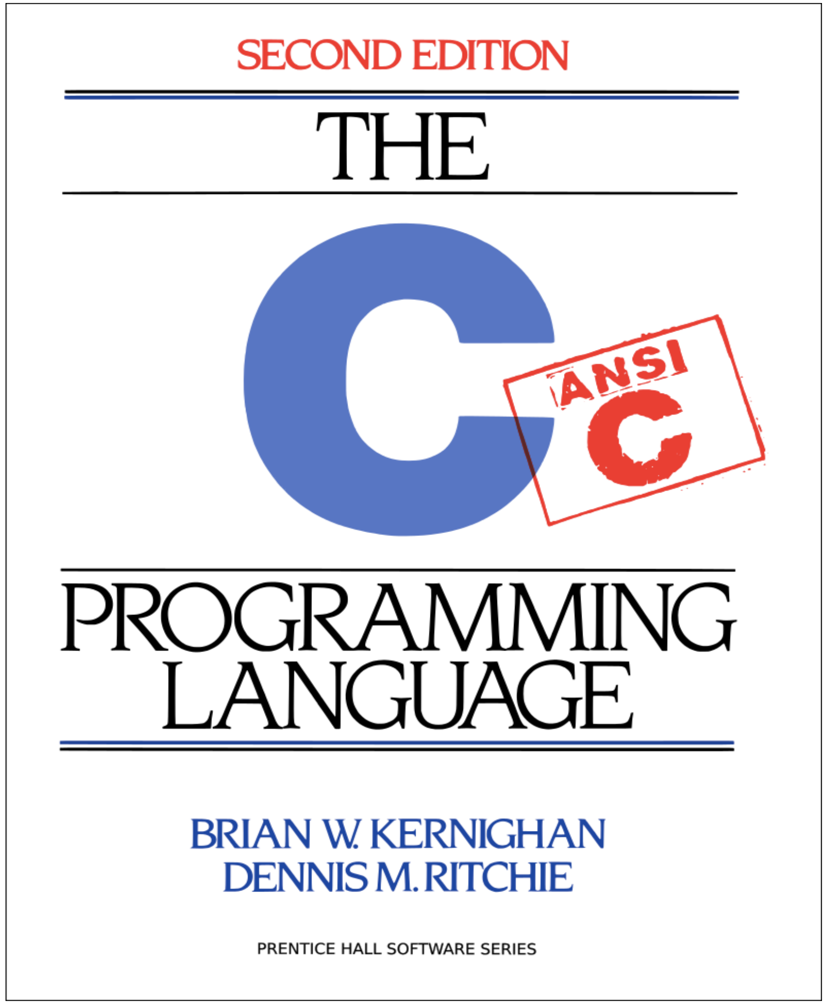
The C Programming Language

GEORGIOS PORTOKALIDIS

CS-392-A SYSTEMS PROGRAMMING


SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

Vs Java

Different philosophies!

C: “Make it efficient and simple, and let the programmer do whatever he wants”

Java: “Make it portable, provide a huge class library, and try to protect the programmer from doing stupid things.”

Vs C++

The syntax of C is almost identical to that of C++

Notable differences:

Comments can be both */* my comment */* or *// my comment*, but strictly speaking ANSI C only accepts the first

C is a procedural language

- You build your algorithms and programs around procedures/functions and functionalities
- Data are organized using structures instead of objects

Memory management is (even more) the job of the programmer

- With great power comes great responsibility
- No *new* or *delete* operators

Be Aware of Various Standards

ISO C

POSIX --- Portable Operating System Interface
- https://en.wikipedia.org/wiki/POSIX

Single UNIX Specification
- X/Open System Interfaces (XSI) enable additional interfaces in POSIX.1 systems

Be Av

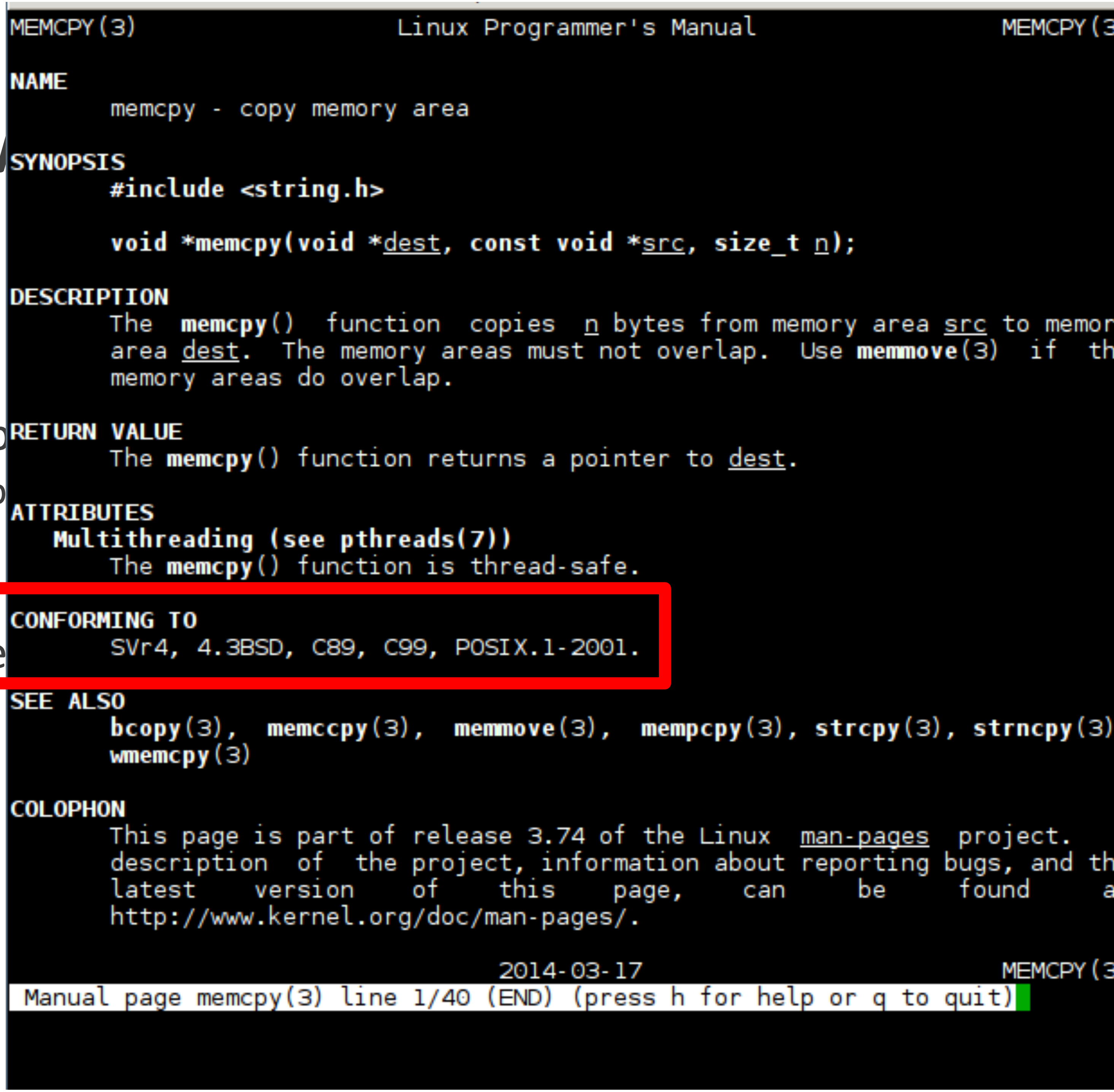
ISO C

POSIX --- Portab

- https://en.wikip

Single UNIX Spe

- X/Open System



```
MEMCPY(3)                 Linux Programmer's Manual                  MEMCPY(3

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

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

DESCRIPTION
       The  memcpy()  function  copies  n bytes from memory area src to memor
       area dest.  The memory areas must not overlap.  Use memmove(3)  if  th
       memory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.

ATTRIBUTES
   Multithreading (see pthreads(7))
       The memcpy() function is thread-safe.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99, POSIX.1-2001.

SEE ALSO
       bcopy(3),  memccpy(3),  memmove(3),  mempcpy(3), strcpy(3), strncpy(3)
       wmemcpy(3)

COLOPHON
       This page is part of release 3.74 of the Linux  man-pages  project.
       description  of  the project, information about reporting bugs, and th
       latest    version    of    this    page,    can    be    found    a
       http://www.kernel.org/doc/man-pages/.

                                  2014-03-17                         MEMCPY(3
 Manual page memcpy(3) line 1/40 (END) (press h for help or q to quit)
```

Coding Style

For systems programming I recommend the Linux kernel's style

https://www.kernel.org/doc/Documentation/CodingStyle

Many other options available

- May depend on preference, project you are contributing, the company you work at, etc.

Make sure that …

- Your code is readable
- You are consistent
- Your code looks the same independently of the editor your are using

Primitive Types

int – integer

float – floating point number

char -- character - a single byte (it is actually a number)

short -- short integer

long -- long integer

double -- double-precision floating point

Types can also by unsigned!

No bool type!

Pointers

Pointer types store an address of memory

- (Usually) an *unsigned long* integer

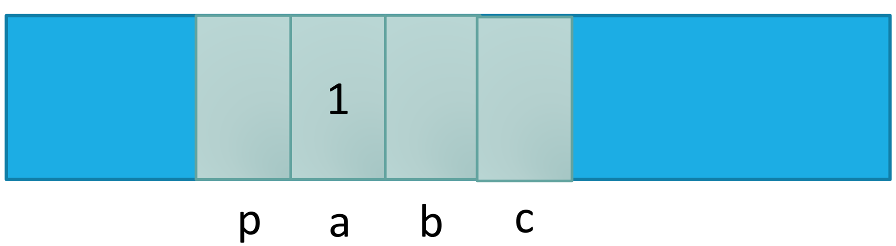
When dereferenced (operator *) they return the contents of what they point to according to the pointer type

- For example, an integer pointer returns an int
- A pointer <u>*type *ptr*</u> point to an area of ssize sizeof(type)
- Dereferencing NULL causes a runtime error

You can obtain a pointer to any variable using the & operator

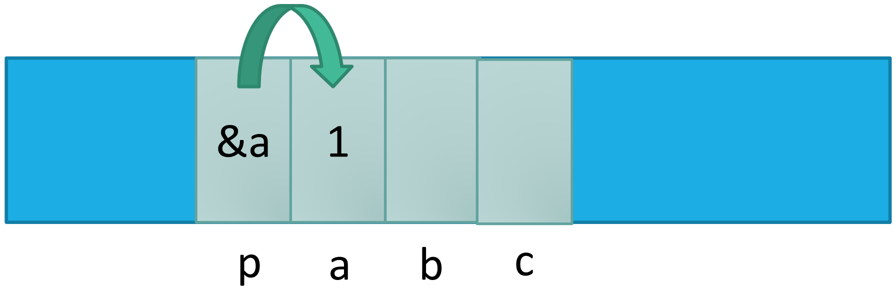
Multiple dereferences are possible

Pointers



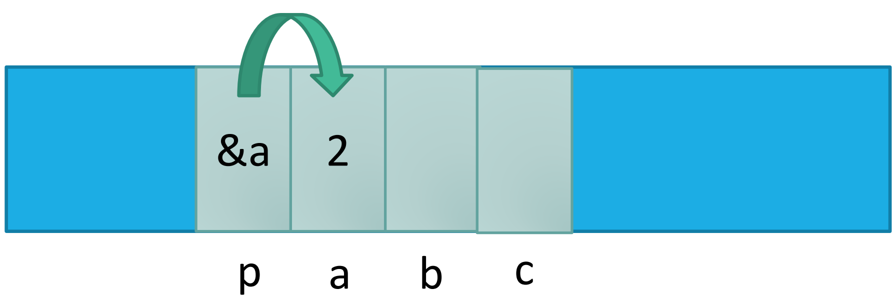

```
int a, b, c, *p;
a = 1;
```

Pointers



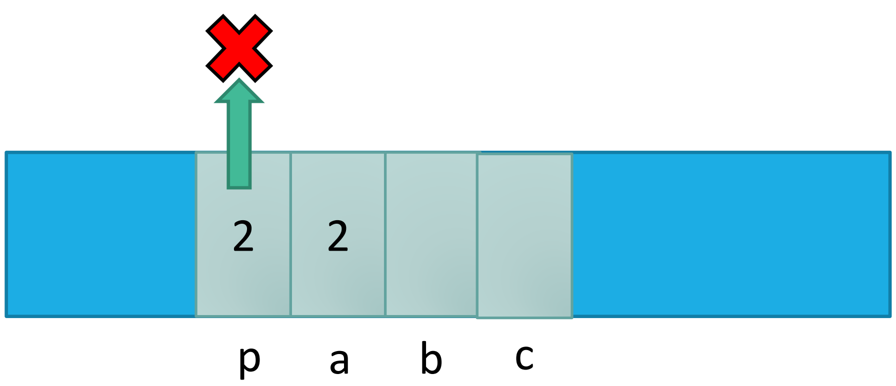

```
int a, b, c, *p;
a = 1;
p = &a;
```

Pointers




```
int a, b, c, *p;
a = 1;
p = &a;
*p = 2;
```

Pointers




```
int a, b, c, *p;
a = 1;
p = &a;
p = 2;
```

Casting

Can cast a variable to a different type

Integer Type Casting:

- signed <-> unsigned: change interpretation of most significant bit
- smaller signed -> larger signed: sign-extend (duplicate the sign bit)
- smaller unsigned -> larger unsigned: zero-extend (duplicate 0)

Cautions:

- cast explicitly, out of practice. C will cast operations involving different types implicitly, often leading to errors
- never cast to a smaller type; will truncate (lose) data
- never cast a pointer to a larger type and dereference it, this accesses memory with undefined contents

Pointer Arithmetic

Can add/subtract from an address to get a new address

- Only perform when necessary

A+i, where A is a pointer = 0x100, i is an int (x86-64)

- int* A: A+i = 0x100 + sizeof(int) * i = 0x100 + 4 * i
- char* A: A+i = 0x100 + sizeof(char) * i = 0x100 + i
- int** A: A + i = 0x100 + sizeof(int*) * i = 0x100 + 8 * I

Rule of thumb: cast pointer explicitly to avoid confusion

- Prefer (char*)(A) + i vs A + i, even if char* A

Generic Types

- void* type is C's provision for generic types
  - Raw pointer to some memory location (unknown type)
  - Can't dereference a `void*`
  - Must cast void* to another type in order to dereference it
- Can cast back and forth between `void*` and other pointer types

```
// stack implementation:

typedef void* elem;

stack stack_new();
void push(stack S, elem e);
elem pop(stack S);
```

```
// stack usage:

int x = 42; int y = 54;
stack S = stack_new():
push(S, &x);
push(S, &y);
int a = *(int*)pop(S);
int b = *(int*)pop(S);
```

Call-by-value Vs Call-by-reference

- Call-by-value: Changes made to arguments passed to a function *aren't* reflected in the calling function
- Call-by-reference: Changes made to arguments passed to a function *are* reflected in the calling function
- C is a *call-by-value* language
- To cause changes to values outside the function, use pointers
  - Do *not* assign the pointer to a different value (that won't be reflected!)
  - Instead, *dereference the pointer* and assign a value to that address

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

```
int x = 42;
int y = 54;
swap(&x, &y);
printf("%d\n", x); // 54
printf("%d\n", y); // 42
```

Macros

Fragment of code given a name; replace occurrence of name with contents of macro

- No function call overhead, type neutral

Uses:

- defining constants (INT_MAX, ARRAY_SIZE)
- defining simple operations (MAX(a, b))
- 122-style contracts (REQUIRES, ENSURES)

Warnings:

- Use parentheses around arguments/expressions, to avoid problems after substitution
- Do not pass expressions with side effects as arguments to macros

```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)
#define WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```

Enumerations

Organize multiple constants together

The first name has value 0, the next 1, and so on

- Unless explicitly specified

Examples:

enum boolean { NO, YES };

- Defines NO as 0 and YES as 1

enum months { JAN = 1, FEB, MAR, … };

- Start numbering from 1

enum escapes { BELL = ‘\a’, BACKSPACE = ‘\b’, … };

- Each constant is initialized with a separate value

Arrays/Strings

Arrays: fixed-size collection of elements of the same type

- Can allocate on the stack or on the heap
- int A[10]; // A is array of 10 int's on the stack

Strings: Null-character ('\0') terminated character arrays

- Null-character tells us where the string ends
- All standard C library functions on strings assume null-termination.

Odds and Ends

Prefix vs Postfix increment/decrement

- a++: use a in the expression, then increment a
- ++a: increment a, then use a in the expression

Switch Statements:

- remember break statements after every case, unless you want fall through (may be desirable in some cases)
- should probably use a default case

Typedefs

Creates an alias type name for a different type

Useful to simplify names of complex data types

```
struct list_node {
   int x;
};

typedef int pixel;
typedef struct list_node* node;
typedef int (*cmp)(int e1, int e2);

pixel x; // int type
node foo; // struct list_node* type
cmp int_cmp; // int (*cmp)(int e1, int e2) type
```

Structures

Collection of values placed under one name in a single block of memory

- Can put structs, arrays in other structs

Given a struct instance, access the fields using the '.' operator

Given a struct pointer, access the fields using the '->' operator

```
struct foo_s {
   int a;
   char b;
};
```

```
struct bar_s {
   char ar[10];
   foo_s baz;
};
```

```
bar_s biz; // bar_s instance
biz.ar[0] = 'a';
biz.baz.a = 42;
bar_s* boz = &biz; // bar_s ptr
boz->baz.b = 'b';
```

Unions

Declaring a union

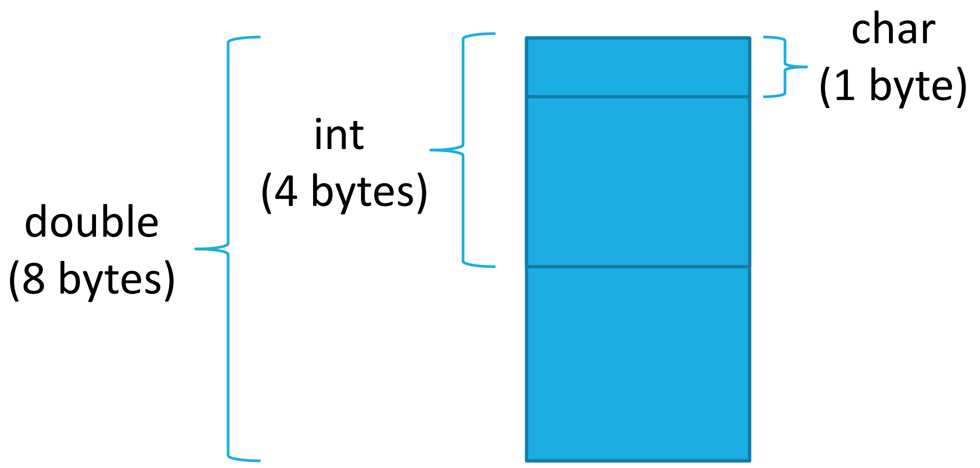
union my_union {

int i;

char c;

double d;

} ;



A union takes enough space to store the largest of its member

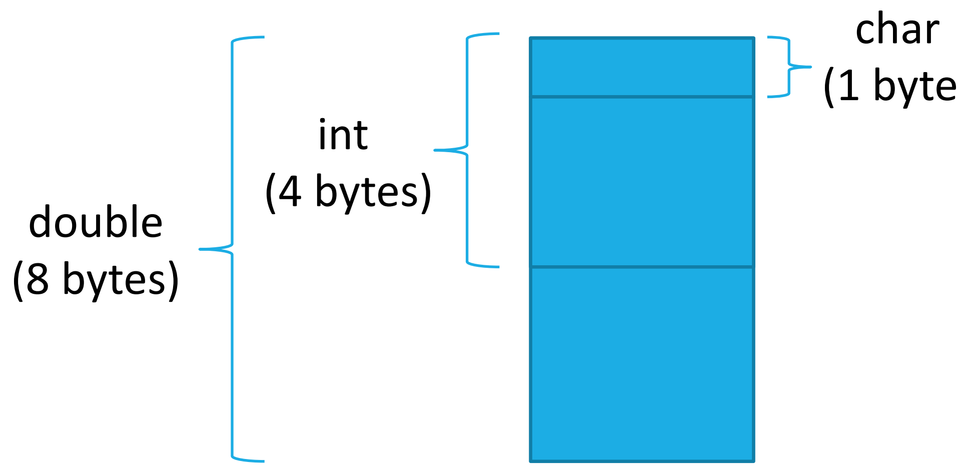
- But only holds one piece of data

Accessed the same way as structures

Unions

Declaring a union

```
union my_union {
        int i;
        char c;
        double d;
        char data[sizeof(double)];
} ;
```



A union takes enough space to store the largest of its member

- But only holds one piece of data

Accessed the same way as structures

Malloc, Free, Calloc

Handle dynamic memory

void* malloc (size_t size):

- allocate block of memory of size bytes
- does not initialize memory

void* calloc (size_t num, size_t size):

- allocate block of memory for array of num elements, each size bytes long
- initializes memory to zero values

void free(void* ptr):

- frees memory block, previously allocated by malloc, calloc, realloc, pointed by ptr
- use exactly once for each pointer you allocate

size argument:

- should be computed using the sizeof operator
- sizeof: takes a type and gives you its size
- e.g., sizeof(int), sizeof(int*)

Malloc Management Rules

Malloc what you free, free what you malloc

- client should free memory allocated by client code
- library should free memory allocated by library code

Number mallocs = Number frees

- Number mallocs > Number Frees: definitely a memory leak
- Number mallocs < Number Frees: definitely a double free

Free a malloced block exactly once

- Should not dereference a freed memory block

Programs in Multiple Files (1)

Each file is first compiled individually into an object file

How will the compiler know these exist?

```
int g_log_level = 0;

void log_to_file(int errorcode)
{
…
}
```

File (library) providing logging functionality

```
int main()
{
…
        g_log_level = 1;

        log_to_file(-1);
…
}
```

File providing logging functionality

Programs in Multiple Files (2)

The compiler needs to know about variables and functions not yet defined

Function prototype declared

```
int g_log_level =

void log_to_file(int er
{
…
}
```

File (library) providing logging functionality

Global variable declared as external

```
extern int g_log_level;
void log_to_file(int );

int main()
{
…
        g_log_level = 1;

        log_to_file(-1);
…
}
```

File providing logging functionality

Programs in Multiple Files (3)

The linker later links all object files into one executable

```
int g_log_level = 0;

void log_to_file(int errorcode)
{
…
}
```

File (library) providing logging functionality

```
extern int g_log_level;
void log_to_file(int );

int main()
{
…
        g_log_level = 1;

        log_to_file(-1);
…
}
```

File using logging functionality

Headers (1)

Include declaration of functions, global variables, macros, etc.

```
extern int g_log_level;
void log_to_file(int );
```

Header file “myheader.h”

```
int g_log_level = 0;

void log_to_file(int errorcode)
{
…
}
```

File (library) providing logging functionality

```
#include “myheader.h”

int main()
{
…
        g_log_level = 1;

        log_to_file(-1);
…
}
```

File using logging functionality

Headers (2)

#include <header.h> includes header.h in your source file

- System directories commonly including headers are searched for the header
  - Example: /usr/include, /usr/local/include , etc.

#include “header.h” searches for header.h in your local directory

It may also contain a pathname

- #include <sys/unistd.h> → File usually resides in /usr/include/sys/unistd.h

Intended for header files

Do not define variables in headers

Good place to define your own types

Headers (3)

A header included twice can lead to problems

- Can occur when source file includes header1.h and header2.h, and header1.h also includes header2.h

#ifdef and #ifndef to the rescue

```
#ifndef MYHEADER_H
#define MYHEADER_H

myheader.h contents

#endif
```

Headers (4)

Includes C declarations and macro definitions to be shared across multiple files

- Only include function prototypes/macros; no implementation code!

Usage: #include <header.h>

- #include <lib> for standard libraries (eg #include <string.h>)
- #include “file” for your source files (eg #include “header.h”)
- Never include .c files (bad practice)

```
// list.h
struct list_node {
   int data;
   struct list_node* next;
};
typedef struct list_node* node;

node new_list();
void add_node(int e, node l);
```

```
// list.c
#include "list.h"

node new_list() {
   // implementation
}

void add_node(int e, node l) {
   // implementation
}
```

```
// stacks.h
#include "list.h"
struct stack_head {
   node top;
   node bottom;
};
typedef struct stack_head* stack

stack new_stack();
void push(int e, stack S);
```

Variable Scope

Local – variables valid within functions

- Defined within the function before use
- Allocated in the program's stack by the compiler
- Prefer to define at the top of the function

```
int my_func(int c)
{
        int i;
…
```

Global – variables accessible by the entire program

- Defined outside functions
- Allocated in a special section in the produced executable

```
int g_debug_level;

int my_func(int c)
{

…
```

Static – variables accessible only within the file they were defined in

- Defined outside functions
- Allocated in a special section in the produced executable
- Note: they are only a programming abstraction. They can actually be accessed by the whole program, if it can find them. It is only programmatically that they are inaccessible to other files.

```
static int g_debug_level;
```

Stack Vs Heap Allocation

- Local variables and function arguments are placed on the *stack*
  - deallocated after the variable leaves scope
  - *do not* return a pointer to a stack-allocated variable!
  - *do not* reference the address of a variable outside its scope!
- Memory blocks allocated by calls to malloc/calloc are placed on the *heap*
- Globals, constants are placed elsewhere
- Example:
  - // a is a pointer on the *stack* to a memory block on the *heap*
  - int* a = malloc(sizeof(int));

Function Scope

By default functions can be called from other files

- That is, they are globally visible
- Function prototypes should be declared in the file where the call is made
  - E.g., through a header

Unless the static keyword is used, which restricts visibility to the file defining them

```
static int my_func(char arg1, int arg2, float arg3)
…
```

my_memcpy()

memcpy – copy memory area

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

```
void *memcpy(void * dst, void const * src, size_t len)
{
   char *dst_p = (char *) dst;
   char const * src_p = (char const *) src;

   while (len--)
   {
      *dst_p++ = *src_p++;
   }

   return (dst);
}
```

Slow byte-by-byte algorithm

my_memcpy()

memcpy – copy memory area

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

```
void *memcpy(void * dst, void const * src, size_t len)
{
   char *p = (char *) dst;
   int off = dst – src;

   while (len--)
   {
      *(p + off) = *p;
      p++;
}

   return p;
}
```

Slightly faster byte-by-byte algorithm