Programming in C and C++

Types, Variables, Expressions and Statements
Academic Year 2021/2022 — Michaelmas Term

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Course Structure

Basics of C:

- Types, variables, expressions and statements
- Functions, compilation and the pre-processor
- Pointers and structures

C Programming Techniques:

- Pointer manipulation: linked lists, trees, and graph algorithms
- Memory management strategies: ownership and lifetimes, reference counting, tracing, and arenas
- Cache-aware programming: array-of-struct to struct-of-array transformations, blocking loops, intrusive data structures
- Unsafe behaviour and mitigations: eg, valgrind, asan, ubsan

Course Structure, continued

Course organization:

- C lecture 1 is delivered in person (covid permitting).
- C lectures 2-9 are recorded and posted online.
- Meet in Intel Lab during lecture 2-9 slots for 8 programming workshops (unmarked, but useful for supervisions)
- There is a C/C++ assessed exercise (tick) that is compulsorary.

Introduction to C++ (with Prof Mycroft):

- C++ lectures 10-12 delivered in person (covid permitting).
- Similarities and differences from C.
- Extensions in C++: templates, classes, memory allocation.

Textbooks

Recommendations for C:

- *The C Programming Language*. Brian W. Kernighan and Dennis M. Ritchie.
- *C: A Reference Manual*. Samuel P. Harbison and Guy L. Steele.

The majority of the class will be on C, but here are two recommendations for C++ as well:

- *The C++ Programming Language*. Bjarne Stroustrup.
- *Thinking in C++: : Introduction to Standard C++*. Bruce Eckel.

The History of C++

- 1966: Martin Richards developed BCPL
- 1969: Ken Thompson designs B
- 1972: Dennis Ricthie designs C
- 1979: Bjarne Stroustrup designs C with Classes
- 1983: C with Classes becomes C++
- 1989: Original C90 ANSI C standard (ISO 1990)
- 1998: ISO C++ standard
- 1999: C99 standard (ISO 1999, ANSI 2000)
- 2011: C++11 ISO standard, C11 ISO standard
- 2014, 2017: C++ standard updates
- 2020: C++20 standard appeared December 2020.

C is a high-level language with exposed unsafe and low-level features.

- C's primitive types are characters, numbers and addresses
- Operators work on these types
- No primitives on composite types (eg, strings, arrays, sets)
- Only static definition and stack-based locals built in (the heap is implemented as a library)
- I/O and threading are also implemented as libraries (using OS primitives)
- The language is *unsafe*: many erroneous uses of C features are not checked (either statically or at runtime), so errors can silently cause memory corruption and arbitrary code execution

The Classic First Program

```c
#include <stdio.h>

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

Compile with

```
$ cc example1.c
```

Execute with:

```
$ ./a.out
Hello, world!
$
```

Generate assembly with

```
$ cc -S example1.c
```

Basic Types

- C has a small set of basic types

type	description
`char`	characters ($\geq$ 8 bits)
`int`	integers ($\geq$ 16 bits, usually 1 word)
`float`	single-precision floating point number
`double`	double-precision floating point number

- Precise size of types is architecture-dependent
- Various *type operators* alter meaning, including: `unsigned`, `short`, `long`, `const`, `volatile`
- This lets us make types like `long int` and `unsigned char`
- C99 added fixed-size types `int16_t`, `uint64_t` etc.

Constants

- Numeric literals can be written in many ways:

type	style	example
`char`	*none*	*none*
`int`	number, character or escape code	`12 'a' '\n'`
`long int`	num w/ suffix `l` or `L`	`1234L`
`float`	num with `'.'`, `'e'`, or `'E'` and suffix `'f'` or `'F'`	`1.234e3F 1234.0f`
`double`	num with `'.'`, `'e'`, or `'E'`	`1.234e3 1234.0`
`long double`	num with `'.'`, `'e'`, or `'E'` and suffix `'l'` or `'L'`	`1.23E3l 123.0L`

- Numbers can be expressed in octal with '0' prefix and hexadecimal with '0x' prefix: `52 = 064 = 0x34`

Defining Constant Values

- An *enumeration* can specify a set of constants:

  ```
  enum boolean {TRUE, FALSE}
  ```

- Enumeration default to allocating successive integers from 0
- It is possible to assign values to constants

  ```
  enum months {JAN=1, FEB, MAR};
  enum boolean {F,T,FALSE=0,TRUE, N=0, Y};
  ```

- *Names* in an enumeration must be distinct, but values need not be.

Variables

- Variables must be *declared* before use
- Variables must be *defined* (i.e., storage allocated) exactly once. (A definition counts as a declaration.)
- A variable name consists of letters, digits and underscores (_); a name must start with a letter or underscore
- Variables are defined with an adjacent type and name, and can optionally be initialised: `long int i = 28L;`
- Multiple variables of the same basic type can be declared or defined together: `char c,d,e;`

Operators

- All operators (including assignment) return a result
- Similar to those found in Java:

type	operators
arithmetic	`+ - * / ++ -- %`
logic	`== != > >= < <= || && !`
bitwise	`| & << >> ^ ~`
assignment	`= += -= *= /= <<= >>= &= ^= %=`
other	`sizeof`

Type Conversion

- Automatic type conversion may occur when two operands to a binary operator are of different type
- Generally, conversion "widens" a value (e.g., `short` → `int`)
- However, "narrowing" is possible and may not generate a warning:

  ```
  int i = 1234;
  char c;
  c = i+1; // i overflows c
  ```

- Type conversion can be forced via a *cast*, which is written as `(type) exp` — for example, `c = (char) 1234L;`

Expressions and Statements

- An expression is a literal, variable, function call or formed from expressions combined with operators: e.g. `x *= y - z`
- Every expression (even assignment) has a type and result
- Operator precedence gives an unambiguous parse for every expression
- An expression (e.g., `x = 0`) becomes a *statement* when followed by a semicolon (e.g. `x = 0;` or `ff(42);`
- Several expression can be separated using a comma ',' and expressions are then evaluated left-to-right: e.g., `x=0,y=1.0`
- The type and value of a comma-separated expression is the type and value of the result of the right-most expression

Blocks and Compound Statements

- A *block* or *compound statement* is formed when multiple statements are surrounded with braces (e.g. `{s1; s2; s3;}`)
- A block of statements is then equivalent to a single statement
- In C90, variables can only be declared or defined at the start of a block, but this restriction was lifted in C99
- Blocks are usually used in function definitions or control flow statements, but can appear anywhere a statement can

Variable Definition vs Declaration

- A variable can be *declared* without defining it using the `extern` keyword; for example `extern int a;`
- The declaration tells the compiler that storage has been allocated elsewhere (usually in another source file)
- If a variable is declared and used in a program, but not defined, this will result in a *link error* (more on this later)
- A *static* modifier prevents a declaration from being accessed elsewhere and, for a local variables, creates exactly one, persistent instance regardless of recursion or re-entrance by threads.

Scope and Type Example (very nasty)

```c
#include <stdio.h>

int a;                         /* what value does a have? */
unsigned char b = 'A';         /* safe to use this? */
extern int alpha;

int main(void) {
  extern unsigned char b;      /* is this needed? */
  double a = 3.4;
  {
    extern a;                  /* is this sloppy? */
    printf("%d %d\n",b,a+1);   /* what will this print? */
  }
  return 0;
}
```

Arrays and Strings

- One or more items of the same type can be grouped into an *array*; for example: `long int i[10];`
- The compiler will allocate a contiguous block of memory for the relevant number of values
- Array items are indexed from zero, and *there is no bounds checking*
- Strings in C are represented as an array of `char` terminated with the special character `'\0'`
- There is language support for this string representation in string contstants with double-quotes; for example `char s[]="two strings mer" "ged and terminated"` (note the implicit concatenation of string literals)
- String functions are in the `string.h` library

Control Flow

- Control flow constructs were ported to Java (so you might know them):
  - `exp ? exp : exp`
  - `if (exp) stmt1 else stmt2`
  -
    ```
    switch(exp) {
      case exp1 : stmt1
      ...
      case expn : stmtn
      default   : default_stmt
    }
    ```
  - `while (exp) stmt`
  - `for (exp1; exp2; exp3) stmt`
  - `do stmt while (exp);`
- The jump statements `break` and `continue` also exist

Control Flow and String Example

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

char s[]="University of Cambridge Computer Laboratory";

int main(void) {
  char c;
  int i, j;
  for (i=0,j=strlen(s)-1;i<j;i++,j--) { // strlen(s)-1 ?
    c=s[i], s[i]=s[j], s[j]=c;
  }
  printf("%s\n",s);
  return 0;
}
```

Goto (often considered harmful)

- The goto statement is never *required*
- It often results in difficult-to-understand code
- Exception handling (where you wish to exit from two or more loops) is one case where goto may be justified:

```
for (...) {
  for (...) {
    ...
    if (big_error) goto error;
  }
}
...
error: // handle error here
```

Programming in C and C++

Lecture 2: Functions and the Preprocessor

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Functions

- C does not have objects with methods, but does have functions
- A function definition has a return type, parameter specification, and a body or statement; for example:
  `int power(int base, int n) { stmt }`
- A function declaration has a return type and parameter specification followed by a semicolon; for example:
  `int power(int base, int n);`

Functions, continued

- Functions can be declared or defined extern or static.
- All arguments to a function are copied, i.e. passed-by-value; modification of the local value does not affect the original
- Just as for variables, a function must have exactly one definition and can have multiple declarations
- A function which is used but only has a declaration, and no definition, results in a link error (more on this later)
- Functions cannot be nested (no closures)

Function Type Gotchas

- A function declaration with no values (e.g. `int power();`) is not an empty parameter specification, rather it means that its arguments should not be type-checked! (luckily, this is not the case in C++)
- Instead, a function with no arguments is declared using void (e.g., `int power(void);`)
- An ellipsis (...) can be used for optional (or varying) parameter specification, for example:
  `int printf(char* fmt,...) { stmt }`
- The ellipsis is useful for defining functions with variable length arguments, but leaves a hole in the type system (`stdarg.h`)

Recursion

- Functions can call themselves recursively
- On each call, a new set of local variables is created
- Therefore, a function recursion of depth n has n sets of variables
- Recursion can be useful when dealing with recursively defined data structures, like trees (more on such data structures later)
- Recursion can also be used as you would in ML:

```c
unsigned int fact(unsigned int n) {
  return n ? n * fact(n-1) : 1;
}
```

Compilation

- A compiler transforms a C source file or execution unit into an object file
- An object file consists of machine code, and a list of:
    - defined or exported symbols representing defined function names and global variables
    - undefined or imported symbols for functions and global variables which are declared but not defined
- A linker combines several object files into an executable by:
    - combining all object code into a single file
    - adjusting the absolute addresses from each object file
    - resolving all undefined symbols

The Part 1b Compiler Course describes how to build a compiler and linker in more detail

Handling Code in Multiple Files in C

- C separates declaration from definition for both variables and functions
- This allows portions of code to be split across multiple files
- Code in different files can then be compiled at different times
  - This allows libraries to be compiled once, but used many times
  - It also allows companies to sell binary-only libraries
- In order to use code written in another file we still need a declaration
- A header file can be used to:
  - supply the declarations of function and variable definitions in another file
  - provide preprocessor macros (more on this later)
  - avoid duplication (and $\therefore$ errors) that would otherwise occur
- You might find the Unix tool `nm` useful for inspecting symbol tables

Multiple Source File Example

example4.h

```c
/* reverse s in place */
void reverse(char str[]);
```

example4a.c

```c
#include <string.h>
#include "example4.h"

void reverse(char s[]) {
  for (int i=0, j=strlen(s)-1;
       i < j; i++, j--) {
    char c=s[i];
    s[i]=s[j], s[j]=c;
  }
}
```

example4b.c

```c
#include <stdio.h>
#include "example4.h"

int main(void) {
  char s[] = "Reverse me";
  reverse(s);
  printf("%s\n", s);
  return 0;
}
```

Variable and Function Scope with static

- The `static` keyword limits the scope of a variable or function
- In the global scope, `static` does not export the function or variable symbol
  - This prevents the variable or function from being called externally
  - BEWARE: extern is the default, not `static` This is also the case for global variables.
- In the local scope, a `static` variable retains its value between function calls
  - A single `static` variable exists even if a function call is recursive
  - Note: `auto` is the default, not `static`

Address Space Layout

A typical x86 32-bit address-space layout:

Description	Address
Top of address space	`0xffff ffff`
...	
Stack (downwards-growing)	typical start `0x7fff ffff`
...	
Heap (upwards-growing)	typical start `0x0020 0000`
...	
Static variables	typical start `0x0010 0000`
C binary code	typical start `0x0000 8000`
...	
Null – often trapped	0x000 0000

(64 bit is messier, but not fundamentally different: see `layout.c`)

C Preprocessor

- The preprocessor executes before any compilation takes place
- It manipulates the text of the source file in a single pass
- Amongst other things, the preprocessor:
  - deletes each occurrence of a backslash followed by a newline;
  - replaces comments by a single space;
  - replaces definitions, obeys conditional preprocessing directives and expands macros; and
  - it replaces escaped sequences in character constants and string literals and concatenates adjacent string literals

Controlling the Preprocessor Programmatically

- The preprocessor can be used by the programmer to rewrite source code
- This is a powerful (and, at times, useful) feature, but can be hard to debug (more on this later)
- The preprocessor interprets lines starting with # with a special meaning
- Two text substitution directives: `#include` and `#define`
- Conditional directives: `#if` , `#elif` , `#else` and `#endif`

The #include Directive

- The `#include` directive performs text substitution
- It is written in one of two forms:

  ```
  #include "filename"
  #include <filename>
  ```
- Both forms replace the `#include` ... line in the source file with the contents of filename
- The quote (") form searches for the file in the same location as the source file, then searches a predefined set of directories
- The angle ($<$) form searches a predefined set of directories
- When a `#include`-d file is changed, all source files which depend on it should be recompiled (easily managed via a 'Makefile')

The `#define` Directive

- The `#define` directive has the form:
  `#define` *name replacement-text*
- The directive performs a direct text substitution of all future examples of *name* with the *replacement-text* for the remainder of the source file
- The *name* has the same constraints as a standard C variable name
- Replacement does not take place if *name* is found inside a quoted string
- By convention, *name* tends to be written in upper case to distinguish it from a normal variable name

Defining Macros

- The `#define` directive can be used to define macros; e.g.:
  `#define MAX(A,B)((A)>(B)?(A):(B))`
- In the body of the macro:
  - prefixing a parameter in the replacement text with ‘#’ places the parameter value inside string quotes (")
  - placing ‘##’ between two parameters in the replacement text removes any whitespace between the variables in generated output
- Remember: the preprocessor only performs text substitution!
  - Syntax analysis and type checking don’t occur until compilation
  - This can result in confusing compiler warnings on line numbers where the macro is used, rather than when it is defined; e.g.
    `#define JOIN(A,B) (A B))`
  - Beware:
    ```
    #define TWO 1+1
    #define WHAT TWO*TWO
    ```

Example

```c
#include <stdio.h>

#define PI 3.141592654
#define MAX(A,B) ((A)>(B)?(A):(B))
#define PERCENT(D) (100*D)                /* Wrong? */
#define DPRINT(D) printf(#D " = %g\n",D)
#define JOIN(A,B) (A ## B)

int main(void) {
  const unsigned int a1=3;
  const unsigned int i = JOIN(a,1);
  printf("%u %g\n",i, MAX(PI,3.14));
  DPRINT(MAX(PERCENT(0.32+0.16),PERCENT(0.15+0.48)));

  return 0;
}
```

Conditional Preprocessor Directives

Conditional directives: `#if` , `#ifdef` , `#ifndef` , `#elif` and `#endif`

- The preprocessor can use conditional statements to include or exclude code in later phases of compilation
- `#if` accepts an integer expression as an argument and retains the code between `#if` and `#endif` (or `#elif`) if it evaluates to a non-zero value; for example:
  `#if SOME_DEF > 8 && OTHER_DEF != THIRD_DEF`
- The preprocessor built-in `defined` takes a name as its argument and gives `1L` if it is `#define`-d; `0L` otherwise
- `#ifdef N` and `#ifndef N` are equivalent to `#if defined(N)` and `#if !defined(N)` respectively
- `#undef` can be used to remove a `#define`-d name from the preprocessor macro and variable namespace.

Preprocessor Example

Conditional directives have several uses, including preventing double definitions in header files and enabling code to function on several different architectures; for example:

```
#if SYSTEM_SYSV
#define HDR "sysv.h"
#elif SYSTEM_BSD
#define HDR "bsd.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

```
#ifndef MYHEADER_H
#define MYHEADER_H 1
...
/* declarations & defns */
...
#endif /* !MYHEADER_H */
```

Error control

- To help other compilers which generate C code (rather than machine code) as output, compiler line and filename warnings can be overridden with:

  ```
  #line constant "filename"
  ```

- The compiler then adjusts its internal value for the next line in the source file as *constant* and the current name of the file being processed as “filename” (“filename” may be omitted)
- The statement `#error` *some-text* causes the preprocessor to write a diagnostic message containing *some-text*
- There are several predefined identifiers that produce special information: `__LINE__` , `__FILE__` , `__DATE__` , and `__TIME__`

Programming in C and C++

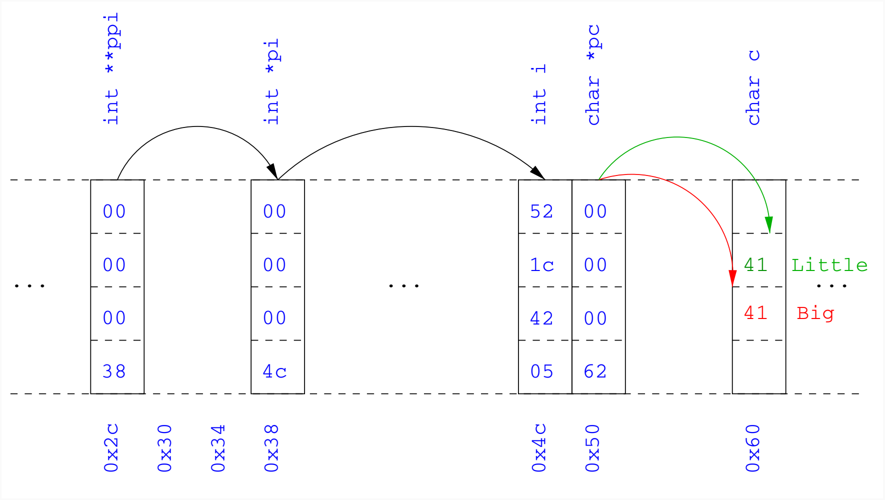
Lecture 3: Pointers and Structures

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Pointers

- Computer memory is often abstracted as a sequence of bytes, grouped into words
- Each byte has a unique address or index into this sequence
- The size of a word (and byte!) determines the size of addressable memory in the machine
- A pointer in C is a variable which contains the memory address of another variable (this can, itself, be a pointer)
- Pointers are declared or defined using an asterisk(`*`); for example: `char *pc;` or `int **ppi;`
- The asterisk binds to the variable name, not the type specifier; for example `char *pc,c;`
- A pointer does not necessarily take the same amount of storage space as the type it points to

Example



int **ppi
int *pi
int i
char *pc
char c
...
00
00
00
38
00
00
00
4c
...
52
1c
42
05
00
00
00
62
41
Little
41
Big
...
0x2c
0x30
0x34
0x38
0x4c
0x50
0x60

Manipulating pointers

- The value "pointed to" by a pointer can be "retrieved" or dereferenced by using the unary * operator; for example:

  ```
  int *p = ...
  int x = *p;
  ```

- The memory address of a variable is returned with the unary ampersand (&) operator; for example `int *p = &x;`
- Dereferenced pointer values can be used in normal expressions; for example: `*pi += 5;` or `(*pi)++`

Example

```c
#include <stdio.h>

int main(void) {
  int x=1,y=2;
  int *pi;
  int **ppi;

  pi = &x; ppi = &pi;
  printf("%p, %p, %d=%d=%d\n",ppi,pi,x,*pi,**ppi);
  pi = &y;
  printf("%p, %p, %d=%d=%d\n",ppi,pi,y,*pi,**ppi);

  return 0;
}
```

Pointers and arrays

- A C array uses consecutive memory addresses without padding to store data
- An array name (used in an expression without an index) represents the memory address of the first element of the array; for example:

```c
char c[10];
char *pc = c;       // This is the same
char *pc = &c[0];   // as this
```

- Pointers can be used to "index" into any element of an array; for example:

```c
int i[10];
int *pi = &i[5];
```

Pointer arithmetic

- Pointer arithmetic can be used to adjust where a pointer points; for example, if `pc` points to the first element of an array, after executing `pc+=3;` then `pc` points to the fourth element
- A pointer can even be dereferenced using array notation; for example `pc[2]` represents the value of the array element which is two elements beyond the array element currently pointed to by `pc`
- In summary, for an array `c` , `*(c+i) == c[i]` and `c+i == &c[i]`
- A pointer is a variable, but an array name is not; therefore `pc = c` and `pc++` are valid, but `c = pc` and `c++` are not

Pointer Arithmetic Example

```c
#include <stdio.h>

int main(void) {
  char str[] = "A string.";
  char *pc = str;

  printf("%c %c %c\n",str[0],*pc,pc[3]);
  pc += 2;
  printf("%c %c %c\n",*pc, pc[2], pc[5]);

  return 0;
}
```

Pointers as function arguments

- Recall that all arguments to a function are copied, i.e. passed-by-value; modification of the local value does not affect the original
- In the second lecture we defined functions which took an array as an argument; for example `void reverse(char s[])`
- Why, then, does reverse affect the values of the array after the function returns (i.e. the array values haven't been copied)?
- because `s` is re-written to `char *s` and the caller implicitly passes a pointer to the start of the array
- Pointers of any type can be passed as parameters and return types of functions
- Pointers allow a function to alter parameters passed to it

Example

Compare `swp1(a,b)` with `swp2(&a,&b)`:

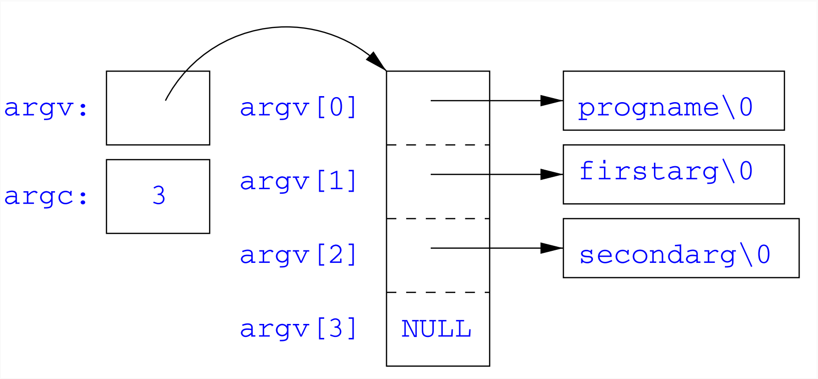
```c
void swp1(int x,int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

```c
void swp2(int *px,int *py)
{
  int temp = *px;
  *px = *py;
  *py = temp;
}
```

Arrays of pointers

- C allows the creation of arrays of pointers; for example
  `int *a[5];`
- Arrays of pointers are particularly useful with strings
- An example is C support of command line arguments:
  `int main(int argc, char *argv[]) { ... }`
- In this case `argv` is an array of character pointers, and `argc` tells the programmer the length of the array

Diagram of Argument List Layout



Multi-dimensional arrays

- Multi-dimensional arrays can be declared in C; for example: `int i[5][10];`
- Values of the array can be accessed using square brackets; for example: `i[3][2]`
- When passing a two dimensional array to a function, the first dimension is not needed; for example, the following are equivalent:

  ```
  void f(int i[5][10]) { ... }
  void f(int i[][10]) { ... }
  void f(int (*i)[10]) { ... }
  ```
- In arrays with higher dimensionality, all but the first dimension must be specified

Pointers to functions

- C allows the programmer to use pointers to functions
- This allows functions to be passed as arguments to functions
- For example, we may wish to parameterise a sort algorithm on different comparison operators (e.g. lexicographically or numerically)
- If the sort routine accepts a pointer to a function, the sort routine can call this function when deciding how to order values

Function Pointer Example

```c
void sort(int a[], const int len,
          int (*compare)(int, int)) {
  for(int i = 0; i < len-1; i++)
    for(int j = 0; j < len-1-i; j++)
      if ((*compare)(a[j],a[j+1])) {
        int tmp = a[j];
        a[j] = a[j+1], a[j+1] = tmp;
      }
}

int inc(int a, int b) { return a > b ? 1 : 0; }
```

Source of some confusion: either or both of the `*` s in `*compare` may be omitted due to language (over-)generosity.

Using a Higher-Order Function in C

```c
#include <stdio.h>
#include "example8.h"

int main(void) {
  int a[] = {1,4,3,2,5};
  unsigned int len = 5;
  sort(a,len,inc); //or sort(a,len,&inc);

  int *pa = a; //C99
  printf("[");
  while (len--) { printf("%d%s", *pa++, len?" ":""); }
  printf("]\n");

  return 0;
}
```

The `void *` pointer

- C has a “typeless” or “generic” pointer: `void *p`
- This can be a pointer to any object (but not legally to a function)
- This can be useful when dealing with dynamic memory
- Enables “polymorphic” code; for example:

  ```
  sort(void *p, const unsigned int len,
  int (*comp)(void *,void *));
  ```
- However this is also a big “hole” in the type system
- Therefore `void *` pointers should only be used where necessary

Structure declaration

- A structure is a collection of one or more members (fields)
- It provides a simple method of abstraction and grouping
- A structure may itself contain structures
- A structure can be assigned to, as well as passed to, and returned from functions
- We declare a structure using the keyword `struct`
- For example, to declare a structure circle we write
  `struct circle {int x; int y; unsigned int r;};`
- Declaring a structure creates a new type

Structure definition

- To define an instance of the structure circle we write
  `struct circle c;`
- A structure can also be initialised with values:
  ```
  struct circle c = {12, 23, 5};
  struct circle d = {.x = 12, .y = 23, .r = 5}; // C99
  ```
- An automatic, or local, structure variable can be initialised by function call: `struct circle c = circle_init();`
- A structure can be declared and several instances defined in one go:
  ```
  struct circle {int x; int y; unsigned int r;} a, b;
  ```

Member access

- A structure member can be accessed using '.' notation `structname.member`; for example: `vect.x`
- Comparison (e.g. `vect1 > vect2`) is undefined
- Pointers to structures may be defined; for example: `struct circle *pc;`
- When using a pointer to a struct, member access can be achieved with the '.' operator, but can look clumsy; for example: `(*pc).x`
- Equivalently, the '`->`" operator can be used; for example: `pc->x`

Self-referential structures

- A structure declaration cannot contain itself as a member, but it can contain a member which is a pointer whose type is the structure declaration itself
- This means we can build recursive data structures; for example:

```
struct tree {
  int val;
  struct tree *left;
  struct tree *right;
}
```

```
struct link {
  int val;
  struct link *next;
}
```

Unions

- A union variable is a single variable which can hold one of a number of different types
- A union variable is declared using a notation similar to structures; for example:
  `union u { int i; float f; char c;};`
- The size of a union variable is the size of its largest member
- The type held can change during program execution
- The type retrieved must be the type most recently stored
- Member access to unions is the same as for structures (‘`.`’ and ‘`->`’)
- Unions can be nested inside structures, and vice versa

Bit fields

- Bit fields allow low-level access to individual bits of a word
- Useful when memory is limited, or to interact with hardware
- A bit field is specified inside a struct by appending a declaration with a colon (`:`) and number of bits; e.g.: `struct fields { int f1 : 2; int f2 : 3;};`
- Members are accessed in the same way as for structs and unions
- A bit field member does not have an address (no & operator)
- Lots of details about bit fields are implementation specific:
    - word boundary overlap & alignment, assignment direction, etc.

Example (adapted from K&R)

```c
struct { /* a compiler symbol table */
  char *name;
  struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
  } flags;
  int utype;
  union {
    int ival; /* accessed as symtab[i].u.ival */
    float fval;
    char *sval;
  } u;
} symtab[NSYM];
```

Programming in C and C++

Lecture 4: Miscellaneous Features, Gotchas, Hints and Tips

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Uses of const and volatile

- Any declaration can be prefixed with `const` or `volatile`
- A `const` variable can only be assigned a value when it is defined
- The `const` declaration can also be used for parameters in a function definition
- The `volatile` keyword can be used to state that a variable may be changed by hardware or the kernel.
  - For example, the `volatile` keyword may prevent unsafe compiler optimisations for memory-mapped input/output

The use of pointers and the const keyword is quite subtle:

- `const int *p` is a pointer to a `const` int
- `int const *p` is also a pointer to a `const` int
- `int *const p` is a `const` pointer to an int
- `const int *const p` is a `const` pointer to a `const` int

Example

```c
int main(void) {
  int i = 42, j = 28;

  const int *pc = &i;        // Also: "int const *pc"
  *pc = 41;                  // Wrong
  pc = &j;

  int *const cp = &i;
  *cp = 41;
  cp = &j;                   // Wrong

  const int *const cpc = &i;
  *cpc = 41;                 // Wrong
  cpc = &j;                  // Wrong
  return 0;
}
```

Typedefs

- The `typedef` operator, creates a synonym for a data type; for example, `typedef unsigned int Radius;`
- Once a new data type has been created, it can be used in place of the usual type name in declarations and casts; for example, `Radius r = 5; ...; r = (Radius) rshort;`
- A `typedef` declaration does not create a new type
  - It just creates a synonym for an existing type
- A `typedef` is particularly useful with structures and unions:

```
typedef struct llist *llptr;
typedef struct llist {
  int val;
  llptr next;
} linklist;
```

Inline functions

- A function in C can be declared `inline`; for example:

```
inline int fact(unsigned int n) {
  return n ? n*fact(n-1) : 1;
}
```

- The compiler will then try to “inline” the function
- A clever compiler might generate 120 for fact(5)
- A compiler might not always be able to “inline” a function
- An inline function must be defined in the same execution unit as it is used
- The inline operator does not change function semantics
  - the inline function itself still has a unique address
  - static variables of an inline function still have a unique address
- Both `inline` and `register` are largely unnecessary with modern compilers and hardware

That's it!

- We have now explored most of the C language
- The language is quite subtle in places; especially beware of:
  - operator precedence
  - pointer assignment (particularly function pointers)
  - implicit casts between ints of different sizes and chars
- There is also extensive standard library support, including:
  - shell and file I/O (`stdio.h`)
  - dynamic memory allocation (`stdlib.h`)
  - string manipulation (`string.h`)
  - character class tests (`ctype.h`)
  - ...
  - (Read, for example, K&R Appendix B for a quick introduction)
  - (Or type "`man function`" at a Unix shell for details)

Library support: I/O

I/O is not managed directly by the compiler; support in `stdio.h`:

```
FILE *stdin, *stdout, *stderr;
int printf(const char *format, ...);
int sprintf(char *str, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...); // sscanf,fscanf
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

```c
#include <stdio.h>
#define BUFSIZE 1024

int main(void) {
  FILE *fp;
  char buffer[BUFSIZE];

  if ((fp=fopen("somefile.txt","rb")) == 0) {
    perror("fopen error:");
    return 1;
  }

  while(!feof(fp)) {
      int r = fread(buffer,sizeof(char),BUFSIZE,fp);
      fwrite(buffer,sizeof(char),r,stdout);
  }

  fclose(fp);
  return 0;
}
```

Library support: dynamic memory allocation

- Dynamic memory allocation is not managed directly by the C compiler
- Support is available in `stdlib.h`:
  - `void *malloc(size_t size)`
  - `void *calloc(size_t nobj, size_t size)`
  - `void *realloc(void *p, size_t size)`
  - `void free(void *p)`
- The C `sizeof` unary operator is handy when using malloc:
  `p = (char *) malloc(sizeof(char)*1000)`
- Any successfully allocated memory must be deallocated manually
  - Note: `free()` needs the pointer to the allocated memory
- Failure to deallocate will result in a memory leak

Gotchas: operator precedence

```c
#include <stdio.h>

struct test {int i;};
typedef struct test test_t;

int main(void) {

  test_t a,b;
  test_t *p[] = {&a,&b};
  p[0]->i=0;
  p[1]->i=0;
  test_t *q = p[0];

  printf("%d\n",++q->i); //What does this do?

  return 0;
}
```

Gotchas: Increment Expressions

```c
#include <stdio.h>

int main(void) {

  int i=2;
  int j=i++ + ++i;
  printf("%d %d\n",i,j); //What does this print?

  return 0;
}
```

Expressions like `i++ + ++i` are known as grey (or gray) expressions in that their meaning is compiler dependent in C (even if they are defined in Java)

Gotchas: local stack

```c
#include <stdio.h>

char *unary(unsigned short s) {
  char local[s+1];
  int i;
  for (i=0;i<s;i++) local[i]='1';
  local[s]='\0';
  return local;
}

int main(void) {

  printf("%s\n",unary(6)); //What does this print?

  return 0;
}
```

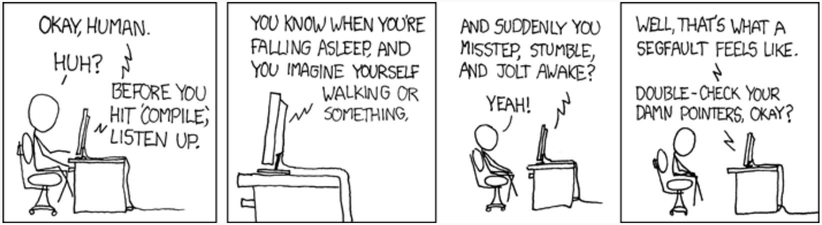
Gotchas: local stack (contd.)

```c
#include <stdio.h>

char global[10];

char *unary(unsigned short s) {
  char local[s+1];
  char *p = s%2 ? global : local;
  int i;
  for (i=0;i<s;i++) p[i]='1';
  p[s]='\0';
  return p;
}

int main(void) {
  printf("%s\n",unary(6)); //What does this print?
  return 0;
}
```

Gotchas: careful with pointers

```c
#include <stdio.h>

struct values { int a; int b; };

int main(void) {
 struct values  test2 = {2,3};
 struct values  test1 = {0,1};

 int *pi = &(test1.a);
 pi += 1; //Is this sensible?
 printf("%d\n",*pi);
 pi += 2; //What could this point at?
 printf("%d\n",*pi);

 return 0;
}
```

Gotchas: XKCD pointers


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Tricks: Duff's device

```c
send(int *to, int *from,
     int count)
{
  int n = (count+7)/8;
  switch(count%8) {
  case 0: do{ *to = *from++;
  case 7:     *to = *from++;
  case 6:     *to = *from++;
  case 5:     *to = *from++;
  case 4:     *to = *from++;
  case 3:     *to = *from++;
  case 2:     *to = *from++;
  case 1:     *to = *from++;
    } while(--n>0);
  }
}
```

```c
boring_send(int *to, int *from,
            int count) {
  do {
    *to = *from++;
  } while(--count > 0);
}
```

Programming in C and C++

Lecture 5: Tooling

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures. . .
- . . . leading to *undefined behaviour*
- Enormous number of possible sources of undefined behavior (See `https://blog.regehr.org/archives/1520`)
- What can we do about it?

Tooling and Instrumentation

Add instrumentation to detect unsafe behaviour!

We will look at 4 tools:

- ASan (Address Sanitizer)
- MSan (Memory Sanitizer)
- UBSan (Undefined Behaviour Sanitizer)
- Valgrind

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
    - Out-of-bounds array accesses
    - Use pointer after call to `free()`
    - Use stack variable after it is out of scope
    - Double-frees or other invalid frees
    - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing
- Built into gcc and clang!

ASan Example #1

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

#define N 10

int main(void) {
  char s[N]  = "123456789";
  for (int i = 0; i <= N; i++)
    printf ("%c", s[i]);
  printf("\n");
  return 0;
}
```

- Loop bound goes past the end of the array
- Undefined behaviour!
- Compile with `-fsanitize=address`

ASan Example #2

```c
#include <stdlib.h>

int main(void) {
  int *a =
    malloc(sizeof(int) * 100);
  free(a);
  return a[5]; // DOOM!
}
```

1. array is allocated
2. array is freed
3. array is dereferenced! (aka use-after-free)

ASan Example #3

```c
#include <stdlib.h>

int main(void) {
  char *s =
    malloc(sizeof(char) * 10);
  free(s);
  free(s);
  printf("%s", s);
  return 0;
}
```

1. array is allocated
2. array is freed
3. array is double-freed

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
    - Typical slowdown 2x
- Does not catch all memory errors
    - NEVER catches *uninitialized* memory accesses
- Still: a **must-use** tool during development

MSan: Memory Sanitizer

- Both local variable declarations and dynamic memory allocation via `malloc()` do not initialize memory:

```c
#include <stdio.h>

int main(void) {
  int x[10];
  printf("%d\n", x[0]); // uninitialized
  return 0;
}
```

- Accesses to uninitialized variables are undefined
  - This does *NOT* mean that you get some unspecified value
  - It means that the compiler is free to do *anything it likes*
- ASan does not catch *uninitialized memory accesses*

MSan: Memory Sanitizer

```c
#include <stdio.h>

int main(void) {
  int x[10];
  printf("%d\n", x[0]); // uninitialized
  return 0;
}
```

- Memory sanitizer (MSan) does check for uninitialized memory accesses
- Compile with `-fsanitize=memory`

MSan Example #1: Stack Allocation

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main(int argc, char** argv) {
 5    int a[10];
 6    a[2] = 0;
 7    if (a[argc])
 8      printf("print something\n");
 9    return 0;
10  }
```

1. Stack allocate array on line 5
2. Partially initialize it on line 6
3. Access it on line 7
4. This might or might not be initialized

MSan Example #2: Heap Allocation

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv) {
5    int *a = malloc(sizeof(int) * 10);
6    a[2] = 0;
7    if (a[argc])
8      printf("print something\n");
9    free(a);
10   return 0;
11 }
```

1. Heap allocate array on line 5
2. Partially initialize it on line 6
3. Access it on line 7
4. This might or might not be initialized

MSan Limitations

- MSan just checks for memory initialization errors
- It is very expensive
    - 2-3x slowdowns, on top of anything else
- Currently only available on clang, and not gcc

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
    - Signed integer overflow
    - Dereferencing null pointers
    - Pointer arithmetic overflow
    - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
    - Typical overhead of 20%
- Use it while developing, maybe even in production
- Built into gcc and clang!

UBSan Example #1

```c
#include <limits.h>

int main(void) {
  int n = INT_MAX;
  int m = n + 1;
  return 0;
}
```

1. Signed integer overflow is undefined
2. So value of `m` is undefined
3. Compile with `-fsanitize=undefined`

UBSan Example #2

```c
#include <limits.h>

int main(void) {
  int n = 65
  int m = n / (n - n);
  return 0;
}
```

1. Division-by-zero is undefined
2. So value of `m` is undefined
3. Any possible behaviour is legal!

UBSan Example #3

```c
#include <stdlib.h>

struct foo {
  int a, b;
};

int main(void) {
  struct foo *x = NULL;
  int m = x->a;
  return 0;
}
```

1. Accessing a null pointer is undefined
2. So accessing fields of `x` is undefined
3. Any possible behaviour is legal!

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead
- Does not catch all undefined behaviour
- Still: a **must-use** tool during development
- **Seriously consider** using it in production

Valgrind

- UBSan, MSan, and ASan require recompiling
- UBSan and ASan don't catch accesses to uninitialized memory
- Enter *Valgrind*!
- Instruments binaries to detect numerous errors

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of `s` is undefined
2. Program prints uninitialized memory
3. Any possible behaviour is legal!
4. Invoke `valgrind` with binary name

Valgrind Limitations

- Adds very substantial runtime overhead
- Not built into GCC/clang (plus or minus?)
- As usual, does not catch all undefined behaviour
- Still: a **must-use** tool during testing

Summary

Tool	Slowdown	Source/Binary	Tool
ASan	Big	Source	GCC/Clang
MSan	Big	Source	Clang
UBSan	Small	Source	GCC/Clang
Valgrind	Very big	Binary	Standalone

Programming in C and C++

Lecture 6: Aliasing, Graphs, and Deallocation

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

The C API for Dynamic Memory Allocation

- `void *malloc(size_t size)`

  Allocate a pointer to an object of size `size`
- `void free(void *ptr)`

  Deallocate the storage `ptr` points to
- Each allocated pointer must be deallocated exactly once along each execution path through the program.
- Once deallocated, the pointer must not be used any more.

One Deallocation Per Path

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

int main(void) {
  int *pi = malloc(sizeof(int));
  scanf("%d", pi);                   // Read an int
  if (*pi % 2) {
    printf("Odd!\n");
    free(pi);                        // WRONG!
  }
}
```

One Deallocation Per Path

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

    int main(void) {
      int *pi = malloc(sizeof(int));
      scanf("%d", pi);                   // Read an int
      if (*pi % 2) {
        printf("Odd!\n");
        free(pi);                        // WRONG!
      }
  }
```

- This code fails to deallocate pi if *pi is even

One Deallocation Per Path

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

int main(void) {
  int *pi = malloc(sizeof(int));
  scanf("%d", pi);                   // Read an int
  if (*pi % 2) {
    printf("Odd!\n");
  }
  free(pi);                          // OK!
}
```

- This code fails to deallocate `pi` if `*pi` is even
- Moving it ensures it always runs

A Tree Data Type

```c
struct node {
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Tree;
```

- This is the tree type from Lab 4.
- It has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.

A Tree Data Type

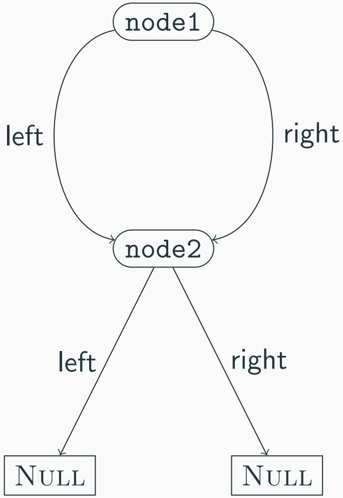
```c
Tree *node(int value, Tree *left, Tree *right) {
  Tree *t = malloc(sizeof(tree));
  t->value = value;
  t->right = right;
  t->left = left;
  return t;
}
void tree_free(Tree *tree) {
  if (tree != NULL) {
    tree_free(tree->left);
    tree_free(tree->right);
    free(tree);
  }
}
```

A Directed Acyclic Graph (DAG)

```
    // Initialize node2
    Tree *node2 = node(2, NULL, NULL);

    // Initialize node1
    Tree *node1 = node(1, node2, node2); // node2 repeated

    // note node1->left == node1->right == node2!
```

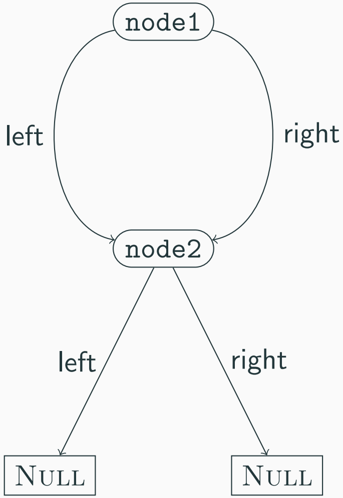
What kind of “tree” is this?

The shape of the graph



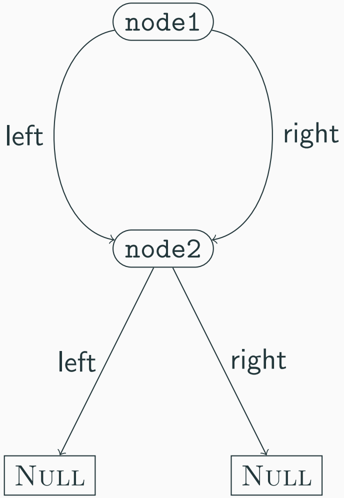
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- `tree_free(node1)` will call `tree_free(node2)` *twice*!

Evaluating `free(node1)`



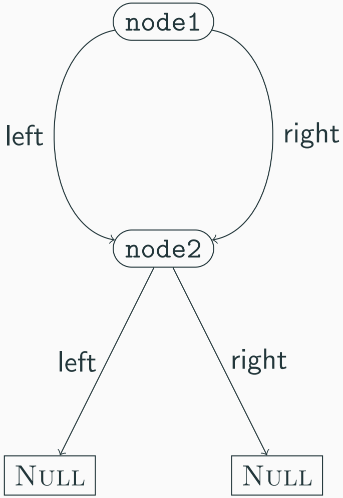
```
free(node1);
```

Evaluating `free(node1)`



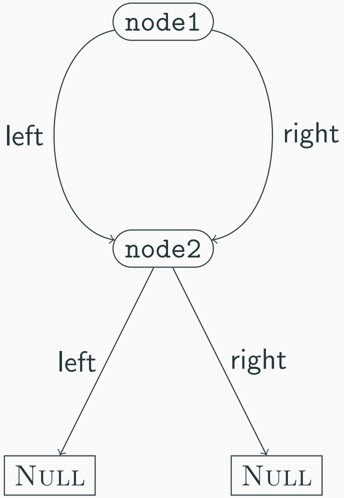
```c
if (node1 != NULL) {
  tree_free(node1->left);
  tree_free(node1->right);
  free(node1);
}
```

Evaluating `free(node1)`



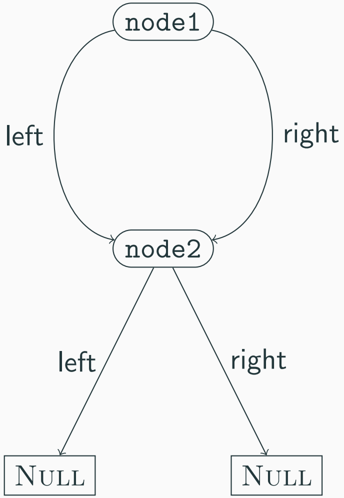
```
    tree_free(node1->left);
    tree_free(node1->right);
    free(node1);
```

Evaluating `free(node1)`



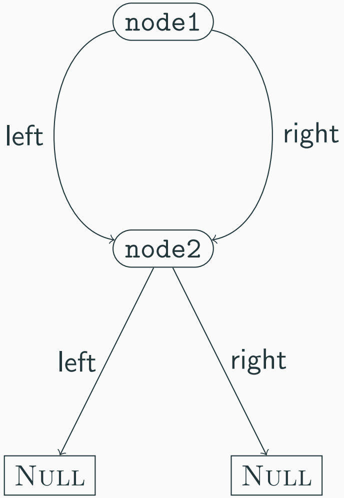
```
tree_free(node2);
tree_free(node2);
free(node1);
```

Evaluating `free(node1)`



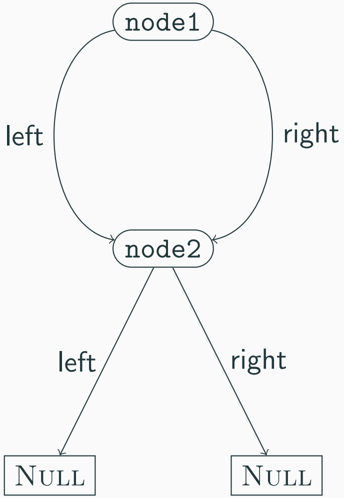
```c
if (node2 != NULL) {
  tree_free(node2->left);
  tree_free(node2->right);
  free(node2);
}
tree_free(node2);
free(node1);
```

Evaluating `free(node1)`



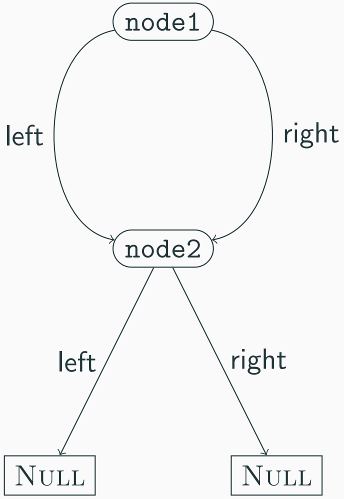
```
    tree_free(node2->left);
    tree_free(node2->right);
    free(node2);
    tree_free(node2);
    free(node1);
```

Evaluating `free(node1)`



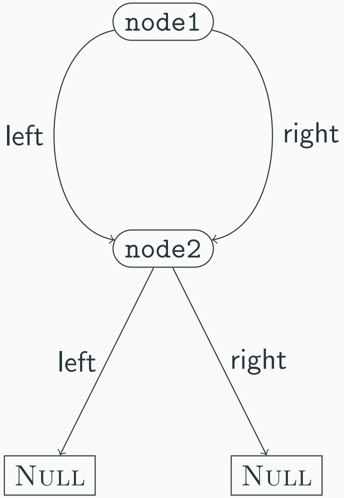
```
tree_free(NULL);
tree_free(NULL);
free(node2);
tree_free(node2);
free(node1);
```

Evaluating `free(node1)`



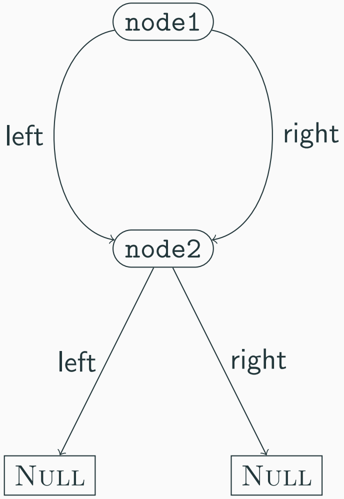
```
if (NULL != NULL) {
  tree_free(NULL->left);
  tree_free(NULL->right);
  free(node1);
}
tree_free(NULL);
free(node2);
tree_free(node2);
free(node1);
```

Evaluating `free(node1)`



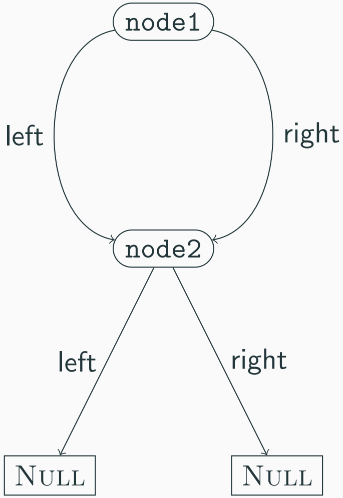
```
tree_free(NULL);
free(node2);
tree_free(node2);
free(node1);
```

Evaluating `free(node1)`



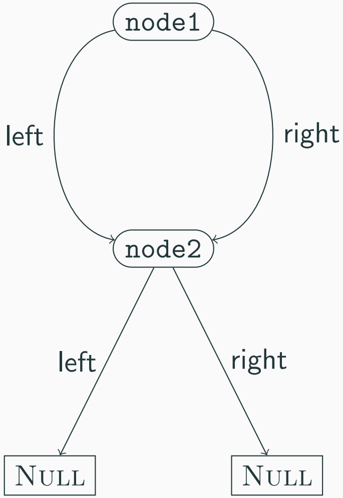
```
    free(node2);
    tree_free(node2);
    free(node1);
```

Evaluating `free(node1)`



```
    free(node2);
    free(node2);
    free(node1);
```

Evaluating `free(node1)`



```
free(node2);
free(node2);
free(node1);
```

`node2` is freed twice!

A Tree Data Type which Tracks Visits

```c
struct node {
  bool visited;
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Tree;
```

- This tree has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.
- it also has a *visited* field.

Creating Nodes of Tree Type

```
Tree *node(int value, Tree *left, Tree *right) {
  Tree *t = malloc(sizeof(tree));
  t->visited = false;
  t->value = value;
  t->right = right;
  t->left = left;
  return t;
}
```

1. Constructing a node sets the visited field to false
2. Otherwise returns the same fresh node as before

Freeing Nodes of Tree Type, Part 1

```
typedef struct TreeListCell TreeList;
struct TreeListCell {
  Tree *head;
  TreeList *tail;
}
TreeList *cons(Tree *head, TreeList *tail) {
  TreeList *result = malloc(TreeListCell);
  result->head = head;
  result->tail = tail;
  return result;
}
```

- This defines `TreeList` as a type of lists of tree nodes.
- `cons` dynamically allocates a new element of a list.

Freeing Nodes of Tree Type, Part 2

```
TreeList *getNodes(Tree *tree, TreeList *nodes) {
  if (tree == NULL || tree->visited) {
    return nodes;
  } else {
    tree->visited = true;
    nodes = cons(tree, nodes);
    nodes = getNodes(tree->right, nodes);
    nodes = getNodes(tree->left, nodes);
    return nodes;
  }
}
```

- Add the unvisited nodes of `tree` to `nodes`.
- Finish if the node is a leaf or already visited
- Otherwise, add the current node and recurse

Freeing Nodes of Tree Type, Part 3

```
void tree_free(Tree *tree) {
  NodeList *nodes = getNodes(tree, NULL);
  while (nodes != NULL) {
    Tree *head = nodes->head;
    NodeList *tail = nodes->tail;
    free(head);
    free(nodes);
    nodes = tail;
  }
}
```

- To free a tree, get all the unique nodes in a list
- Iterate over the list, freeing the nodes
- Don't forget to free the list!
- We're doing dynamic allocation to free some data. . .

Summary

- Freeing trees is relatively easy
- Freeing DAGs or general graphs is much harder
- Freeing objects at most once is harder if there are multiple paths to them.

Arenas

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Tree;
```

- This is the original tree data type
- Let's keep this type, but change the (de)allocation API

Arenas

```c
typedef struct arena *arena_t;
struct arena {
  int size;
  int current;
  Tree *elts;
};

arena_t make_arena(int size) {
  arena_t arena = malloc(sizeof(struct arena));
  arena->size = size;
  arena->current = 0;
  arena->elts = malloc(size * sizeof(Tree));
  return arena;
}
```

Arena allocation

```
Tree *node(int value, Tree *left, Tree *right,
           arena_t arena) {
  if (arena->current < arena->size) {
    Tree *t = arena->elts + arena->current;
    arena->current += 1;
    t->value = value, t->left = left, t->right = right;
    return t;
  } else
    return NULL;
}
```

To allocate a node from an arena:

1. Initialize `current` element
2. Increment `current`
3. Return the initialized node

Freeing an Arena

```c
void free_arena(arena_t arena) {
  free(arena->elts);
  free(arena);
}
```

- We no longer free trees individually
- Instead, free a whole arena at a time
- All tree nodes allocated from the arena are freed at once

Example

```c
arena_t a = make_arena(BIG_NUMBER);

Tree *node1 = node(0, NULL, NULL, a);
Tree *node2 = node(1, node1, node1, a); // it's a DAG now
// do something with the nodes...
free_arena(a);
```

- We allocate the arena
- We can build an arbitrary graph
- And free all the elements at once

Conclusion

- Correct memory deallocation in C requires thinking about control flow
- This can get tricky!
- Arenas are an idiom for (de)allocating big blocks at once
- Reduces need for thinking about control paths
- But can increase working set sizes

Programming in C and C++

Lecture 7: Reference Counting and Garbage Collection

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

The C API for Dynamic Memory Allocation

- In the previous lecture, we saw how to use arenas and ad-hoc graph traversals to manage memory when pointer graphs contain aliasing or cycles
- These are not the only idioms for memory management in C!
- Two more common patterns are *reference counting* and *type-specific garbage collectors*.

A Tree Data Type

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Tree;
```

- This is still the tree type from Lab 4.
- It has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.

Construct Nodes of a Tree

```c
Tree *node(int value, Tree *left, Tree *right) {
  Tree *t = malloc(sizeof(tree));
  t->value = value;
  t->right = right;
  t->left = left;
  return t;
}
```

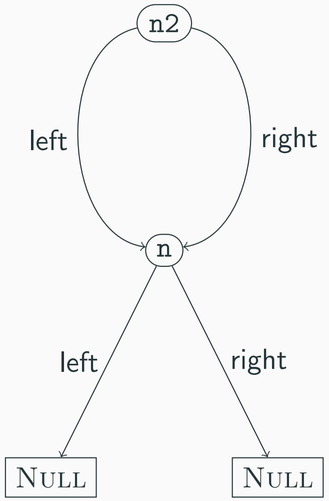
1. Allocate a pointer to a tree struct
2. Initialize the `value` field
3. Initialize the `right` field
4. Initialize the `left` field
5. Return the initialized pointer!

A Directed Acyclic Graph (DAG)

```
1     Tree *n = node(2, NULL, NULL);
2     Tree *n2 =
3       node(1, n, n); // n repeated!
```

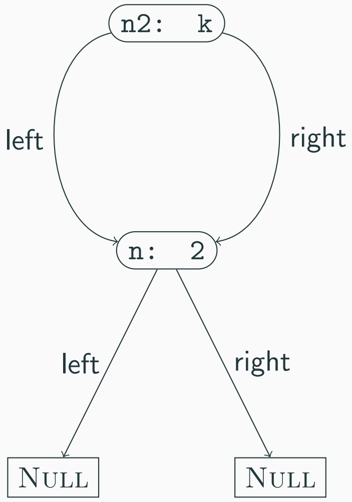
1. We allocate `n` on line 1
2. On line 2, we create `n2` whose `left` *and* `right` fields are `n`.
3. Hence `n2->left` and `n2->right` are said to *alias* – they are two pointers aimed at the same block of memory.

The shape of the graph



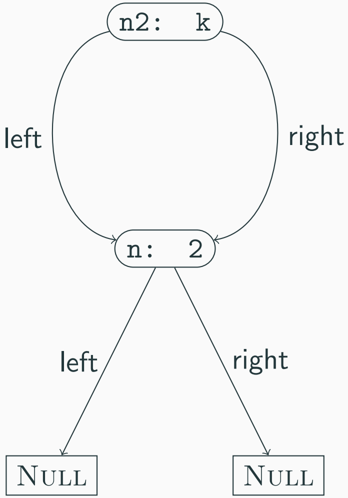
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- A recursive free of the tree `n2` will try to free `n` twice.

The Idea of Reference Counting



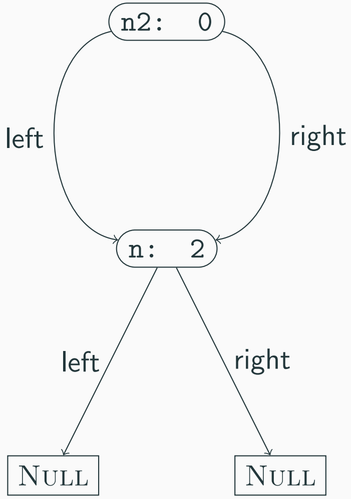
1. The problem: freeing things with two pointers to them twice
2. Solution: stop doing that
3. Keep track of the number of pointers to an object
4. Only free when the count reaches zero

How Reference Counting Works



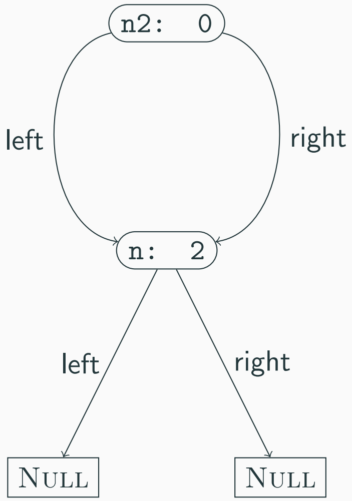
1. We start with k references to `n2`

How Reference Counting Works



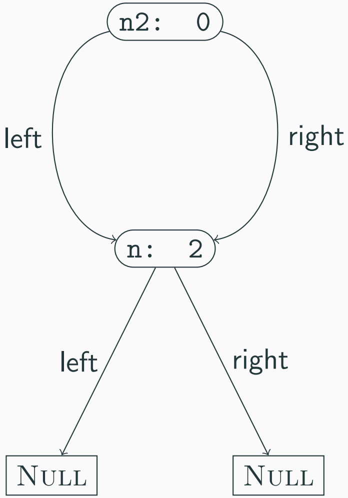
1. We start with *k* references to `n2`
2. Eventually *k* becomes 0

How Reference Counting Works



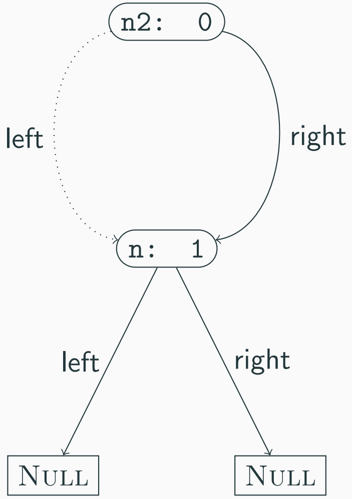
1. We start with *k* references to `n2`
2. Eventually *k* becomes 0
3. It's time to delete `n2`

How Reference Counting Works



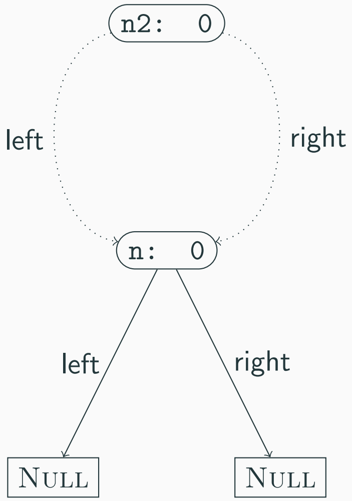
1. We start with *k* references to `n2`
2. Eventually *k* becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



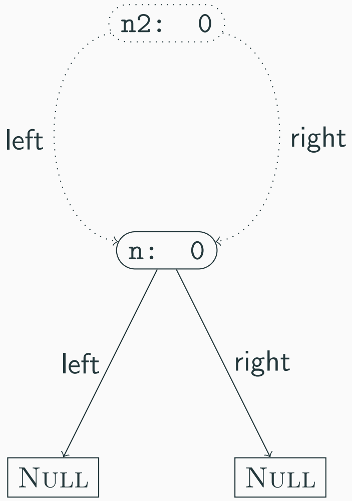
1. We start with *k* references to `n2`
2. Eventually *k* becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



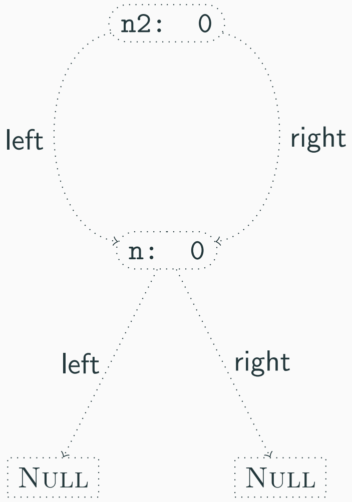
1. We start with *k* references to `n2`
2. Eventually *k* becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`

How Reference Counting Works



1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`

How Reference Counting Works



1. We start with *k* references to `n2`
2. Eventually *k* becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`
6. Recursively delete `n`

The Reference Counting API

```c
struct node {
  unsigned int rc;
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Node;

const Node *empty = NULL;
Node *node(int value,
           Node *left,
           Node *right);
void inc_ref(Node *node);
void dec_ref(Node *node);
```

- We add a field `rc` to keep track of the references.
- We keep the same `node` constructor interface.
- We add a procedure `inc_ref` to increment the reference count of a node.
- We add a procedure `dec_ref` to decrement the reference count of a node.

Reference Counting Implementation: `node()`

```
1  Node *node(int value,
2              Node *left,
3              Node *right) {
4    Node *r = malloc(sizeof(Node));
5    r->rc = 1;
6    r->value = value;
7
8    r->left = left;
9    inc_ref(left);
10
11   r->right = right;
12   inc_ref(right);
13   return r;
14 }
```

- On line 4, we initialize the `rc` field to 1. (Annoyingly, this is a rather delicate point!)
- On line 8-9, we set the `left` field, and increment the reference count of the pointed-to node.
- On line 11-12, we do the same to `right`

Reference Counting Implementation: `inc_ref()`

```
1  void inc_ref(Node *node) {
2    if (node != NULL) {
3      node->rc += 1;
4    }
5  }
```

- On line 3, we increment the `rc` field (if nonnull)
- That's it!

Reference Counting Implementation: `dec_ref()`

```
1  void dec_ref(Node *node) {
2    if (node != NULL) {
3      if (node->rc > 1) {
4        node->rc -= 1;
5      } else {
6        dec_ref(node->left);
7        dec_ref(node->right);
8        free(node);
9      }
10   }
11 }
```

- When we decrement a reference count, we check to see if we are the last reference (line 3)
- If not, we just decrement the reference count (line 4)
- If so, then decrement the reference counts of the children (lines 6-7)
- Then free the current object. (line 8)

Example 1

```
1   Node *complete(int n) {
2     if (n == 0) {
3       return empty;
4     } else {
5       Node *sub = complete(n-1);
6       Node *result =
7         node(n, sub, sub);
8       dec_ref(sub);
9       return result;
10    }
11  }
```

- `complete(n)` builds a complete binary tree of depth n
- Sharing makes memory usage $O(n)$
- On line 5, makes a recursive call to build subtree.
- On line 6, builds the tree
- On line 8, call `dec_ref(sub)` to drop the stack reference sub
- On line 9, *don't* call `dec_ref(result)`

Example 1 – mistake 1

```c
Node *complete(int n) {
  if (n == 0) {
    return empty;
  } else {
    Node *sub = complete(n-1);
    Node *result =
      node(n, sub, sub);
    // dec_ref(sub);
    return result;
  }
}
```

- If we forget to call `dec_ref(sub)`, we get a memory leak!
- `sub` begins with a refcount of 1
- `node(sub, sub)` bumps it to 3
- If we call `dec_ref(complete(n))`, the outer node will get freed
- But the children will end up with an `rc` field of 1

Example 1 – mistake 2

```
1  Node *complete(int n) {
2    if (n == 0) {
3      return empty;
4    } else {
5      return node(n,
6                  complete(n-1),
7                  complete(n-1));
8    }
9  }
```

- This still leaks memory!
- `complete(n-1)` begins with a refcount of 1
- The expression on lines 5-7 bumps each subtree to a refcount of 2
- If we call `free(complete(n))`, the outer node will get freed
- But the children will end up with an `rc` field of 1

Design Issues with Reference Counting APIs

- The key problem: *who is responsible for managing reference counts?*
- Two main options: sharing references vs transferring references
- Both choices work, but must be made consistently
- To make this work, API must be documented very carefully
    - Good example: Python C API
    - `https://docs.python.org/3/c-api/intro.html#objects-types-and-reference-counts`

Mitigations: Careful Use of Getters and Setters

```c
Node *get_left(Node *node) {
  inc_ref(node->left);
  return(node->left);
}

void set_left(Node *node,
              Node *newval) {
  inc_ref(newval);
  dec_ref(node->left);
  node->left = newval;
}
```

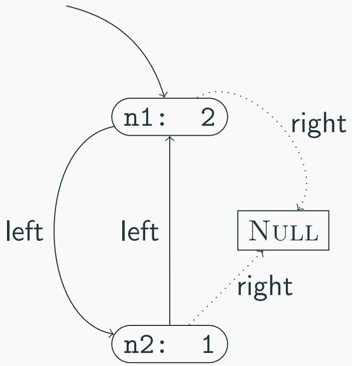
- The `get_left()` function returns the left subtree, but also increments the reference count
- The `set_left()` function updates the left subtree, incrementing the reference count to the new value and decrementing the reference

Cycles: A Fundamental Limitation on Reference Counting

```
Node *foo() {
  Node *n1 = node(1, NULL, NULL);
  Node *n2 = node(2, NULL, NULL);
  set_left(n1, n2);
  set_left(n2, n1);
  dec_ref(n2);
  return node1;
}
```

What does a call to `foo()` build?

A Cyclic Object Graph



- `n1->rc` is 2, since `n2` points to it
- `n2->rc` is 1, since `n1` points to it
- This is a cyclic graph
- Even though there is only 1 external reference to `n1`, `n1->rc` is 2.
- Hence `dec_ref(foo())` will not free memory!
- Reference counting *cannot collect cycles*

Garbage Collection: Dealing with Cycles

- In ML or Java, we don't have to worry about cycles or managing reference counts explicitly
- We rely on a *garbage collector* to manage memory automatically
- In C, we can *implement* garbage collection to manage memory

GC API – Data structures

```c
struct node {
  int value;
  struct node *left;
  struct node *right;
  bool mark;
  struct node *next;
};
typedef struct node Node;

struct root {
  Node *start;
  struct root *next;
};
typedef struct root Root;

struct alloc {
  Node *nodes;
  Root *roots;
};
typedef struct alloc Alloc;
```

- `Node *` are node objects, but augmented with a `mark` bit (Lab 5) and a `next` link connecting all allocated nodes
- A `Root *` is a node we don't want to garbage collect. Roots are also in a linked list
- An allocator `Alloc *` holds the head of the lists of nodes and roots

GC API – Procedures

```
Alloc *make_allocator(void);
Node *node(int value,
           Node *left,
           Node *right,
           Alloc *a);
Root *root(Node *node, Alloc *a);
void gc(Alloc *a);
```

- `make_allocator` creates a fresh allocator
- `node(n, l, r, a)` creates a fresh node in allocator `a` (as in the arena API)
- `root(n)` creates a new root object rooting the node `n`
- `gc(a)` frees all nodes unreachable from the roots

Creating a Fresh Allocator

```c
Alloc *make_allocator(void) {
  Alloc *a = malloc(sizeof(Alloc));
  a->roots = NULL;
  a->nodes = NULL;
  return a;
}
```

- Creates a fresh allocator with empty set of roots and nodes
- Invariant: no root or node is part of two allocators!
- (Could use global variables, but thread-unfriendly)

Creating a Node

```
1  Node *node(int value,
2             Node *left,
3             Node *right,
4             Alloc *a) {
5    Node *r = malloc(sizeof(Node));
6    r->value = value;
7    r->left = left;
8    r->right = right;
9    //
10   r->mark = false;
11   r->next = a->nodes;
12   a->nodes = r;
13   return r;
14 }
```

- Lines 5-9 perform familiar operations: allocate memory (line 5) and initialize data fields (6-8)
- Line 10 initializes `mark` to `false`
- Lines 11-12 add new node to `a->nodes`

Creating a Root

```
1  Root *root(Node *node,
2             Alloc *a) {
3    Root *g =
4      malloc(sizeof(Root));
5    g->start = node;
6    g->next = a->roots;
7    a->roots = g;
8    return g;
9  }
```

- On line 4, allocate a new `Root` struct `g`
- On line 5, set the `start` field to the `node` argument
- On lines 6-7, attach `g` to the `roots` of the allocator `a`
- Now the allocator knows to treat the root as always reachable

Implementing a Mark-and-Sweep GC

- Idea: split GC into two phases, *mark* and *sweep*
- In mark phase:
    - From each root, mark the nodes reachable from that root
    - I.e., set the `mark` field to true
    - So every reachable node will have a true mark bit, and every unreachable one will be set to false
- In sweep phase:
    - Iterate over every allocated node
    - If the node is unmarked, free it
    - If the node is marked, reset the mark bit to false

Marking

```
void mark_node(Node *node) {
  if (node != NULL && !node->mark) {
    node->mark = true;
    mark_node(node->left);
    mark_node(node->right);
  }
}

void mark(Alloc *a) {
  Root *g = a->roots;
  while (g != NULL) {
    mark_node(g->start);
    g = g->next;
  }
}
```

- `mark_node()` function marks a node if unmarked, and then recursively marks subnodes
- Just like in lab 6!
- `mark()` procedure iterates over the roots, marking the nodes reachable from it.
- If a node is not reachable from the `a->roots` pointer, it will stay `false`

Sweeping

```
1  void sweep(Alloc *a) {
2    Node *n = a->nodes;
3    Node *live = NULL;
4    while (n != NULL) {
5      Node *tl = n->next;
6      if (!(n->mark)) {
7        free(n);
8      } else {
9        n->mark = false;
10       n->next = live;
11       live = n;
12     }
13     n = tl;
14   }
15   a->nodes = live;
16 }
```

- On line 2, get a pointer to *all allocated nodes* via `a->nodes`
- On line 3, create a new empty list of `live` nodes
- On lines 4-14, iterate over each allocated node
- On line 6, check to see if the node is unmarked
- If unmarked, free it (line 8)
- If marked, reset the mark bit and add it to the `live` list (9-11)
- On line 15, update `a->nodes` to the still-live `live` nodes

The `gc()` routine

```
void gc(Alloc *a) {
  mark(a);
  sweep(a);
}
```

- `gc(a)` just marks and sweeps!
- To use the gc, we allocate nodes as normal
- Periodically, invoke `gc(a)` to clear out unused nodes
- That's it!

Design Considerations

- This kind of custom GC is quite slow relative to ML/Java gcs
- However, simple and easy to implement (only 50 lines of code!)
- No worries about cycles or managing reference counts
- Worth considering using the Boehm gc if gc in C/C++ is needed:
  - `https://www.hboehm.info/gc/`
  - Drop-in replacement for malloc!
- Still useful when dealing with interop between gc'd and manually-managed languages (eg, DOM nodes in web browsers)

Programming in C and C++

Lecture 8: The Memory Hierarchy and Cache Optimization

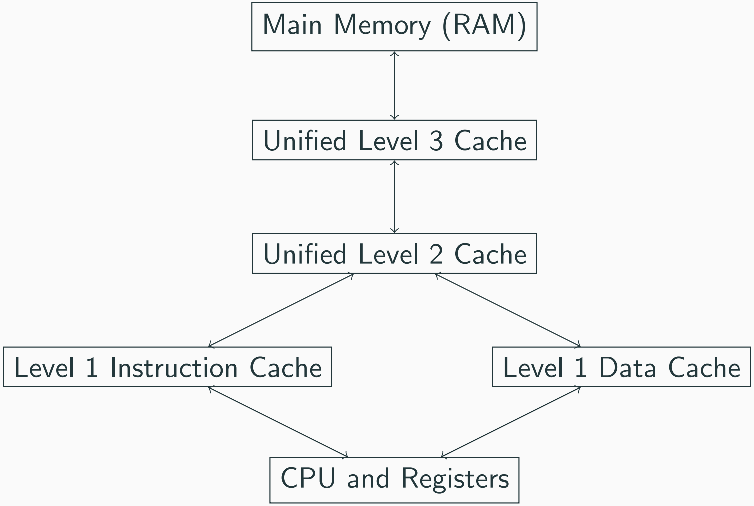
David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

Three Simple C Functions

```
void increment_every(int *array)
  for (int i = 0; i < BIG_NUMBER; i += 1) {
    array[i] = 0;
}
void increment_8th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 8)
    array[i] = 0;
}
void increment_16th(int *array) {
  for (int i = 0; i < BIG_NUMBER; i += 16)
    array[i] = 0;
}
```

- Which runs faster?
- …and by how much?

The Memory Hierarchy



Latencies in the Memory Hierarchy

Access Type	Cycles	Time	Human Scale
L1 cache reference	≈4	1.3 ns	1s
L2 cache reference	≈10	4 ns	3s
L3 cache reference, unshared	≈40	13 ns	10s
L3 cache reference, shared	≈65	20 ns	16s
Main memory reference	≈300	100 ns	80s

- Random accesses to main memory are *slow*
- This can dominate performance!

How Caches Work

When a CPU looks up an address. . . :

1. It looks up the address in the cache
2. If present, this is a *cache hit* (cheap!)
3. If absent, this is a *cache miss*
    3.1 The address is then looked up in main memory (expensive!)
    3.2 The address/value pair is then stored in the cache
    3.3 . . . along with the next 64 bytes (typically) of memory
    3.4 This is a *cache line* or *cache block*

Locality: Taking advantage of caching

Caching is most favorable:

- Each piece of data the program works on is near (in RAM) the address of the last piece of data the program worked on.
- This is the *principle of locality*
- Performance engineering involves redesigning data structures to take advantage of locality.

Pointers Are Expensive

Consider the following Java linked list implementation

```
class List<T> {
  public T head;
  public List<T> tail;

  public List(T head, List<T> tail) {
    this.head = head;
    this.tail = tail;
  }
}
```

Pointers Are Expensive in C, too

```
typedef struct List* list_t;
struct List {
  void *head;
  list_t tail;
};
list_t list_cons(void *head, list_t tail) {
  list_t result = malloc(sizeof(struct list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- C uses `void *` for genericity, but this introduces pointer indirections.
- This can get expensive!

Specializing the Representation

Suppose we use a list at a Data $*$ type:

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

struct List {
  Data *head;
  struct List *tail;
};
```

Technique #1: Intrusive Lists

We can try changing the list representation to:

```
typedef struct intrusive_list ilist_t;
struct intrusive_list {
  Data head;
  ilist_t tail;
};
ilist_t ilist_cons(Data head, ilist_t tail) {
  list_t result = malloc(sizeof(struct intrusive_list));
  r->head = head;
  r->tail = tail;
  return r;
}
```

- The indirection in the head is removed
- But we had to use a specialized representation
- Can no longer use generic linked list routines

Technique #2: Lists of Structs to Arrays of Structs

Linked lists are expensive:

1. Following a tail pointer can lead to *cache miss*
2. Cons cells requiring storing a tail pointer…
3. This reduces the number of data elements that fit in a cache line
4. This decreases data density, and increases *cache miss rate*
5. Replace `ilist_t` with `Data[]`!

Technique #2: Lists of Structs to Arrays of Structs

We can try changing the list representation to:

```
Data *iota_array(int n) {
  Data *a = malloc(n * sizeof(Data));
  for (int i = 0; i < n; i++) {
    a[i].i = i;
    a[i].d =  1.0;
    a[i].c = 'x';
  }
  return a;
}
```

- No longer store tail pointers
- Every element comes after previous element in memory
- Can no longer incrementally build lists
- Have to know size up-front

Technique #3: Arrays of Structs to Struct of Arrays

```
struct data {
  int i;
  double d;
  char c;
};
typedef struct data Data;

void traverse(int n, Data *a) {
  for (int i = 0; i < n; i++)
    a[i].c += 'y';
}
```

- Note that we are only modifying character field c.
- We have “hop over” the integer and double fields.
- So characters are at least 12, and probably 16 bytes apart.
- This means only 4 characters in each cache line. . .
- Optimally, 64 characters fit in each cache line. . .

Technique #3: Arrays of Structs to Struct of Arrays

```
typedef struct datavec *DataVec;
struct datavec {
  int *is;
  double *ds;
  char *cs;
};
```

- Instead of storing an array of structures...
- We store a struct of arrays
- Now traversing just the cs is easy

Technique #3: Traversing Struct of Arrays

```
void traverse_datavec(int n, DataVec d) {
  char *a = d->cs;
  for (int i = 0; i < n; i++) {
    a[i] += 'y';
  }
}
```

- To update the characters. . .
- Just iterate over the character. . .
- Higher cache efficiency!

Technique #4: Loop Blocking

```c
#define SIZE 8192
#define dim(i, j) (((i) * SIZE) + (j))

double *add_transpose(double *A,
                      double *B) {
  double *dest =
    malloc(sizeof(double)
           * SIZE * SIZE);
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      dest[dim(i,j)] =
        A[dim(i,j)] + B[dim(j,i)];
    }
  }
  return dest;
}
```

- The `add_transpose` function takes two square matrices A and B, and returns a new matrix equal to $A + B^T$.
- C stores arrays in row-major order.

How Matrices are Laid out in Memory

$$A \triangleq \left\{ \begin{array}{lll} 0 & 1 & 4 \\ 9 & 16 & 25 \\ 36 & 49 & 64 \\ 81 & 100 & 121 \end{array} \right\}$$

Address	0	1	2	3	4	5	6	7	8	9	10	11
Value	0	1	4	9	16	25	36	49	64	81	100	121

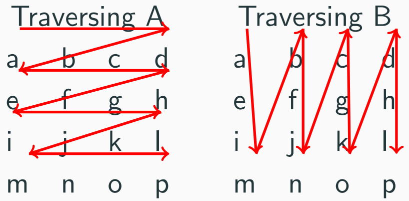
- A is a 3×4 array.
- $A(i, j)$ is at address $3 \times i + j$ (0 based!)
- E.g., $A(2, 1) = 49$, at address 7
- E.g., $A(3, 1) = 100$, at address 10

Loop Blocking

```c
#define SIZE 8192
#define dim(i, j) (((i) * SIZE) + (j))

double *add_transpose(double *A,
                      double *B) {
  double *dest =
    malloc(sizeof(double)
           * SIZE * SIZE);
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      dest[dim(i,j)] =
        A[dim(i,j)] + B[dim(j,i)];
    }
  }
  return dest;
}
```

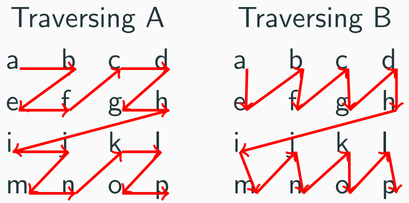
- The succesive accesses to $A(i, j)$ will go sequentially in memory
- The successive accesses to $B(j, i)$ will jump SIZE elements at a time

How to Block a Loop, Concept



- We can see that A has a favorable traversal, and B is “jumpy”
- Let’s change the traversal order!

How to Block a Loop, Concept



- Since each nested iteration is acting on the same $n \times n$ submatrix, a cache miss on one lookup will bring memory into cache for the other lookup
- This reduces the total number of cache misses

Loop Blocking

```c
double *add_transpose_blocked(double *m1,
                              double *m2,
                              int bsize) {
double *dest =
  malloc(sizeof(double) * SIZE * SIZE);
for (int i = 0; i < SIZE; i += bsize) {
  for (int j = 0; j < SIZE; j += bsize) {
    for (int ii = i; ii < i+bsize; ii++) {
      for (int jj = j; jj < j+bsize; jj++) {
        dest[dim(ii,jj)] =
          m1[dim(ii,jj)] + m2[dim(jj, ii)];
      }
    }
  }
}
return dest;
}
```

- Doubly-nested loop goes to quadruply-nested loop
- Increment `i` and `j` by `bsize` at a time
- Do a little iteration over the submatrix with `ii` and `jj`

Conclusion

- Memory is hierarchical, with each level slower than predecessors
- Caching make *locality assumption*
- Making this assumption true requires careful design
- Substantial code alterations can be needed
- But can lead to major performance gains

Programming in C and C++

Lecture 9: Debugging

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

What is Debugging?

Debugging is a methodical process of finding and reducing the number of bugs (or defects) in a computer program, thus making it behave as originally expected.

There are two main types of errors that need debugging:

- Compile-time: These occur due to misuse of language constructs, such as syntax errors. Normally fairly easy to find by using compiler tools and warnings to fix reported problems, e.g.: `gcc -Wall -pedantic -c main.c`
- Run-time: These are much harder to figure out, as they cause the program to generate incorrect output (or "crash") during execution. This lecture will examine how to methodically debug a run-time error in your C code.

The Runtime Debugging Process

A typical lifecycle for a C/C++ bug is:

- a program fails a *unit test* included with the source code, or a bug is reported by a user, or observed by the programmer.
- given the failing input conditions, a programmer *debugs* the program until the offending source code is located.
- the program is recompiled with a source code fix and a *regression test* is run to confirm that the behaviour is fixed.

Unit tests are short code fragments written to test code modules in isolation, typically written by the original developer.

Regression testing ensures that changes do not uncover new bugs, for example by causing unit tests to fail in an unrelated component.

What is a Bug?

The program contains a *defect* in the source code, either due to a design misunderstanding or an implementation mistake. This defect is manifested as a *runtime failure*. The program could:

- crash with a memory error or "segmentation fault"
- return an incorrect result
- have unintended side-effects like corrupting persistent storage

The art of debugging arises because defects *do not materialise predictably*.

- The program may require specific inputs to trigger a bug
- Undefined or implementation-defined behaviour may cause it to only crash on a particular OS or hardware architecture
- The issue may require separate runs and many hours (or months!) of execution time to manifest

Finding Defects

Defects are not necessarily located in the source code near a particular runtime failure.

- A variable might be set incorrectly that causes a timer to fire a few seconds too late. The *defect* is the assignment point, but the *failure* is later in time.
- A configuration file might be parsed incorrectly, causing the program to subsequently choose the wrong control path. The *defect* is the parse logic, but the *failure* is observing the program perform the wrong actions.
- A function `foo()` may corrupt a data structure, and then `bar()` tries to use it and crashes. The *defect* is in `foo()` but the *failure* is triggered at `bar()`.

The greater the distance between the original defect and the associated failure, the more difficult it is to debug.

Finding Defects (cont.)

Sometimes the defect directly causes the failure and is easy to debug. Consider this NULL pointer error:

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

int main(int argc, char **argv) {
  struct hostent *hp;
  hp = gethostbyname("doesntexist.abc");
  printf("%s\n", hp->h_name);
  return 0;
}
```

Just fix the code to add a NULL pointer check before printing hp.

Debugging via printing values

A very common approach to debugging is via printing values as the program executes.

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

int main(int argc, char **argv) {
  struct hostent *hp;
  printf("argc: %d\n", argc);
  for (int i=1; i<argc; i++) {
    hp = gethostbyname(argv[i]);
    printf("hp: %p\n", hp);
    if (hp)
     printf("%s\n", hp->h_name);
  }
  return 0;
}
```

Debugging via printing values (cont.)

Executing this will always show the output as the program runs.

```
./lookup google.org recoil.org
argc: 3
hp: 0x7fd87ae00420
google.org
hp: 0x7fd87ae00490
recoil.org
```

Some tips on debug printing:

- Put in as much debugging information as you can to help gather information in the future
- Make each entry as unique as possible so that you tie the output back to the source code
- Flush the debug output so that it reliably appears in the terminal

Debugging via printing values (cont.)

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

int main(int argc, char **argv) {
  struct hostent *hp;
  printf("%s:%2d argc: %d\n", __FILE__, __LINE__, argc);
  for (int i=1; i<argc; i++) {
    hp = gethostbyname(argv[i]);
    printf("%s:%2d hp: %p\n", __FILE__, __LINE__, hp);
    fflush(stdout);
    printf("%s:%2d %s\n", __FILE__, __LINE__, hp->h_name);
    fflush(stdout);
  }
  return 0;
}
```

Debugging via printing values (cont.)

The source code is now very ugly and littered with debugging statements. The C preprocessor comes to the rescue.

- Define a DEBUG parameter to compile your program with.
- `#define` a debug printf that only runs if DEBUG is non-zero.
- Disabling DEBUG means debugging calls will be optimised away at compile time.

```
#ifndef DEBUG
#define DEBUG 0
#endif
#define debug_printf(fmt, ...) \
  do { if (DEBUG) { \
    fprintf(stderr, fmt, __VA_ARGS__); \
    fflush(stderr); } } \
  while (0)
```

Debugging via printing values (cont.)

```c
#include <netdb.h>
#include <stdio.h>
#ifndef DEBUG
#define DEBUG 0
#endif
#define debug_printf(fmt, ...) \
  do { if (DEBUG) { fprintf(stderr, fmt, __VA_ARGS__); \
                    fflush(stderr); } } while (0)
int main(int argc, char **argv) {
  debug_printf("argc: %d\n", argc);
  for (int i=1; i<argc; i++) {
    struct hostent *hp = gethostbyname(argv[i]);
    debug_printf("hp: %p\n", hp);
    printf("%s\n", hp->h_name);
  }
  return 0;
}
```

Debugging via Assertions

Defects can be found more quickly than printing values by using assertions to encode invariants through the source code.

```c
#include <netdb.h>
#include <stdio.h>
#include <assert.h> // new header file

int main(int argc, char **argv) {
  struct hostent *hp;
  hp = gethostbyname("doesntexist.abc");
  assert(hp != NULL); // new invariant
  printf("%s\n", hp->h_name);
  return 0;
}
```

Debugging via Assertions (cont.)

The original program without assertions will crash:

```
cc -Wall debug-s6.c && ./lookup
Segmentation fault: 11
```

Running with assertions results in a much more friendly error message.

```
cc -Wall debug-s12.c && ./lookup
Assertion failed: (hp != NULL),
function main, file debug2.c, line 10.
```

Debugging via Assertions (cont.)

Using assert is a cheap way to ensure an invariant remains true.

- A failed assertion will immediately exit a program.
- Assertions can be disabled by defining the NDEBUG preprocessor flag.

Never cause side-effects in assertions as they may not be active!

```
cc -Wall -DNDEBUG debug-s12.c && ./lookup
Segmentation fault: 11
```

Fault Isolation

Debugging is the process of fault isolation to find the cause of the failure.

- Never try to guess the cause randomly. This is time consuming.
- Stop making code changes incrementally to fix the bug.
- Do think like a detective and find clues to the cause.

Remember that you are trying to:

- Reproduce the problem so you can observe the failure.
- Isolate the failure to some specific inputs.
- Fix the issue and confirm that there are no regressions.

Reproducing the Bug

Consider this revised program that performs an Internet name lookup from a command-line argument.

```c
#include <netdb.h>
#include <stdio.h>
#include <assert.h>
int main(int argc, char **argv) {
  struct hostent *hp;
  hp = gethostbyname(argv[1]);
  printf("%s\n", hp->h_name);
  return 0;
}
```

This program can crash in at least two ways. How can we reproduce both?

Reproducing the Bug (cont.)

This program can crash in at least two ways.

`cc -Wall -o lookup debug-s16.c`

First crash: if we do not provide a command-line argument:

```
./lookup
Segmentation fault: 11
```

Second crash: if we provide an invalid network hostname:

```
./lookup doesntexist.abc
Segmentation fault: 11
```

It does work if we provide a valid hostname:

```
./lookup www.recoil.org
bark.recoil.org
```

Both positive and negative results are important to give you more hints about how many distinct bugs there are, and where their source is.

Isolating the Bug

We now know of two failing inputs, but need to figure out where in the source code the defect is. From earlier, one solution is to put assert statements everywhere that we suspect could have a failure.

```c
#include <netdb.h>
#include <stdio.h>
#include <assert.h>
int main(int argc, char **argv) {
  struct hostent *hp;
  assert(argv[1] != NULL);
  hp = gethostbyname(argv[1]);
  assert(hp != NULL);
  printf("%s\n", hp->h_name);
  return 0;
}
```

Reproducing the Bug with Assertions

Recompile the program with the assertions enabled.

```
cc -Wall -o lookup debug-s18.c
```

First crash: if we do not provide a command-line argument:

```
./lookup
Assertion failed: (argv[1] != NULL),
function main, file debug-s18.c, line 7.
```

Second crash: if we provide an invalid network hostname:

```
./lookup doesntexist.abc
Assertion failed: (hp != NULL), function main,
file debug-s18.c, line 9.
```

Reproducing the Bug with Assertions (cont.)

It does work if we provide a valid hostname:

```
./lookup www.recoil.org
bark.recoil.org
```

The assertions show that there are two distinct failure points in application, triggered by two separate inputs.

Using Debugging Tools

While assertions are convenient, they do not scale to larger programs. It would be useful to:

- Observe the value of a C variable *during* a program execution
- Stop the execution of the program if an assertion is violated.
- Get a *trace* of the function calls leading up to the failure.

These features are provided by **debuggers**, which let a programmer to monitor the memory state of a program during its execution.

- *Interpretive* debuggers work by simulating program execution one statement at a time.
- More common for C code are *direct execution* debuggers that use hardware and operating system features to inspect the program memory and set "breakpoints" to pause execution.

Example: Using lldb from LLVM

Let's use the lldb debugger from LLVM to find the runtime failure without requiring assertions.

```
cc -Wall -o lookup -DNDEBUG -g debug-s18.c
```

Run the binary using lldb instead of executing it directly.

```
lldb ./lookup
(lldb) target create "./lookup"
Current executable set to './lookup' (x86_64).
```

At the (lldb) prompt use run to start execution.

```
(lldb) run www.recoil.org
Process 9515 launched: './lookup' (x86_64)
bark.recoil.org
Process 9515 exited with status = 0 (0x00000000)
```

Example: Using lldb from LLVM (cont.)

Now try running the program with inputs that trigger a crash:

```
(lldb) run doesntexist.abc
frame #0: 0x0000000100000f52 lookup
main(argc=2, argv=0x00007fff5fbff888) + 50 at debug-s18.c:12
    9      assert(argv[1] != NULL);
   10      hp = gethostbyname(argv[1]);
   11      assert(hp != NULL);
-> 12      printf("%s\n", hp->h\_name);
   13
return 0;
```

The program has halted at line 12 and lets us inspect the value of variables that are in scope, confirming that the hp pointer is NULL.

```
(lldb) print hp
(hostent *) $1 = 0x0000000000000000
```

Example: Using lldb from LLVM (cont.)

We do not have to wait for a crash to inspect variables.

Breakpoints allow us to halt execution at a function call.

```
(lldb) break set --name main
(lldb) run www.recoil.org
    7    {
    8      struct hostent *hp;
    9      assert(argv[1] != NULL);
-> 10      hp = gethostbyname(argv[1]);
```

The program has run until the main function is encountered, and stopped at the first statement.

Example: Using lldb from LLVM (cont.)

We can set a watchpoint to inspect when variables change state.

```
(lldb) watchpoint set variable hp
```

This will pause execution right after the hp variable is assigned to. We can now resume execution and see what happens:

```
(lldb) continue
Process 9661 resuming
Process 9661 stopped
* thread #1: tid = 0x3c2fd3 <..> stop reason = watchpoint 1
frame #0: 0x0000000100000f4e <...> debug-s18.c:12
    9    assert(argv[1] != NULL);
   10    hp = gethostbyname(argv[1]);
   11    assert(hp != NULL);
-> 12    printf("%s\n", hp->h_name);
   13    return 0;
   14  }
```

Example: Using lldb from LLVM (cont.)

When program execution is paused in lldb, we can inspect the local variables using the print command.

```
(lldb) print hp
(hostent *) $0 = 0x0000000100300460
(lldb) print hp->h_name
(char *) $1 = 0x0000000100300488 "bark.recoil.org"
```

We can thus:

- confirm that hp is non-NULL even when the program does not crash
- print the contents of the `hp->h` name value, since the debugger can follow pointers
- see the C types that the variables had (e.g. `hostent *`)

Debugging Symbols

How did the debugger find the source code in the compiled executable? Compile it without the -g flag to see what happens.

```
cc -Wall -DNDEBUG debug-s18.c
```

```
(lldb) run doesnotexist.abc
loader'main + 50:
-> 0x100000f52: movq  (%rax), %rsi
   0x100000f55: movb  $0x0, %al
   0x100000f57: callq 0x100000f72 ; symbol stub for: printf
   0x100000f5c: movl  $0x0, %ecx
```

We now only have assembly language backtrace, with some hints in the output about `printf`.

Debugging Symbols (cont.)

The compiler emits a *symbol table* that records the mapping between a program's variables and their locations in memory.

- Machine code uses memory addresses to reference memory, and has no notion of variable names. For example, `0x100000f72` is the address of printf earlier
- The symbol table records an association from `0x100000f72` and the printf function
- The -g compiler flag embeds additional debugging information into the symbol tables
- This debugging information also maps the source code to the program counter register, keeping track of the control flow

Debugging Symbols (cont.)

Debugging information is not included by default because:

- The additional table entries take up a significant amount of extra disk space, which would be a problem on embedded systems like a Raspberry Pi
- They are not essential to run most applications, unless they specifically need to modify their own code at runtime
- Advanced users can still use debuggers with just the default symbol table, although relying on the assembly language is more difficult
- Modern operating systems such as Linux, MacOS X and Windows support storing the debugging symbols in a separate file, making disk space and compilation time the only overhead to generating them.

Debugging Tools

`lldb` is just one of a suite of debugging tools that are useful in bug hunting.

- The LLVM compiler suite (`https://llvm.org`) also has the `clang-analyzer` static analysis engine that inspects your source code for errors.
- The GCC compiler (`https://gcc.gnu.org`) includes the `gdb` debugger, which has similar functionality to `lldb` but with a different command syntax.
- Valgrind (`http://valgrind.org`) (seen in an earlier lecture!) is a dynamic analysis framework that instruments binaries to detect many classes of memory management and threading bugs.

Unit and Regression Test

We have used many techniques to find our bugs, but it is equally important to make sure they do not return. Create a unit test:

```c
#include <stdio.h>
#include <netdb.h>
#include <assert.h>
void lookup(char *buf) {
  assert(buf != NULL);
  struct hostent *hp = gethostbyname(buf);
  printf("%s -> %s\n", buf, hp ? hp->h_name : "unknown");
}
void lookup_test(void) {
  lookup("google.com");
  lookup("doesntexist.abc");
  lookup("");
  lookup(NULL);
}
```

Unit and Regression Test (cont.)

We can now invoke `lookup()` for user code, or lookup `test()` to perform the self-test.

```c
#include <stdlib.h>

void lookup(char *buf);
void lookup_test(void);

int main(int argc, char **argv) {
  if (getenv("SELFTEST"))
    lookup_test ();
  else
    lookup(argv[1]);
}
```

Unit and Regression Test (cont.)

Can now run this code as a test case or for live lookups.

```
cc -Wall -g lookup_logic.c lookup_main.c -o lookup
./lookup google.com
# for live operation
env SELFTEST=1 ./lookup # for unit tests
```

Unit and Regression Tests (cont.)

Building effective unit tests requires methodical attention to detail:

- The use of `assert` in the lookup logic is a poor interface, since it terminates the entire program. How could this be improved?
- The unit tests in lookup test are manually written to enumerate the allowable inputs. How can we improve this coverage?
- C and C++ have many *open-source unit test frameworks* available. Using them gives you access to widely used conventions such as `xUnit` that helps you structure your tests.
- Take the opportunity to run your unit tests after every significant code change to spot unexpected failures (dubbed *regression testing*).
- *Continuous integration* runs unit tests against every single code commit. If using GitHub, then Travis CI (`https://travis-ci.org`) will be useful for your projects in any language.