Programming in C and C++

Academic Year 2023/2024 — Michaelmas Term

David J Greaves
(Materials by and Alan Mycroft, Neel Krishnaswami et al.)

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.
- 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).

Introduction to C++:

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

There is a compulsory Assessed Exercise (tick) for this course.

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:

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

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 (and `goto`)
    ```

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:

```
1  for (...) {
2    for (...) {
3      ...
4      if (big_error) goto error;
5    }
6  }
7  ...
8  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:

```
1        unsigned int fact(unsigned int n) {
2          return n ? n * fact(n-1) : 1;
3        }
```

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

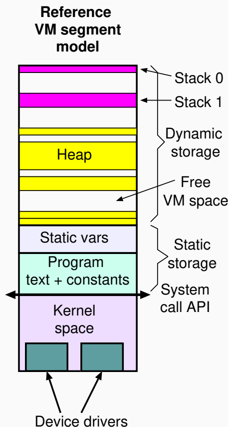
Typical segment structure:

C programs can run with or without virtual memory. The memory regions used remain the same. With VM, four principle segments are needed in user space. System calls are used for O/S interactions.

The program code and constants, such as string literals, can share one, read-only segment, shared over all instances of the running program.

Statically-allocated variables are given addresses at compile/link time. In C, these are globals and explicit statics. They require a mutable segment whose size is known at load time.

Dynamic storage consists of the shared heap and a stack per thread. These vary unpredictably in size from run to run. They must be positioned far apart in address space so they don't bump into each other. If each is a segment, unused gaps have very little cost in a VM system. But for embedded controllers without VM and a single thread, having the stack grow down while the heap generally growing upwards can help ensure that when they bump into each other the system has truely run out of space (but the heap with be striated).



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++

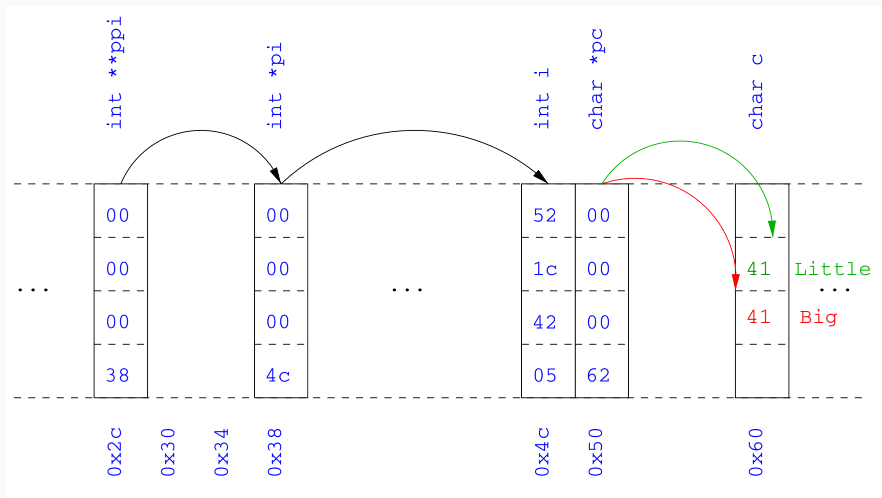
Lecture 3: Pointers and Structures

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



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

```
1  #include <stdio.h>
2
3  int main(void) {
4    int x=1,y=2;
5    int *pi;
6    int **ppi;
7
8    pi = &x; ppi = &pi;
9    printf("%p, %p, %d=%d=%d\n",ppi,pi,x,*pi,**ppi);
10   pi = &y;
11   printf("%p, %p, %d=%d=%d\n",ppi,pi,y,*pi,**ppi);
12
13   return 0;
14 }
```

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)`:

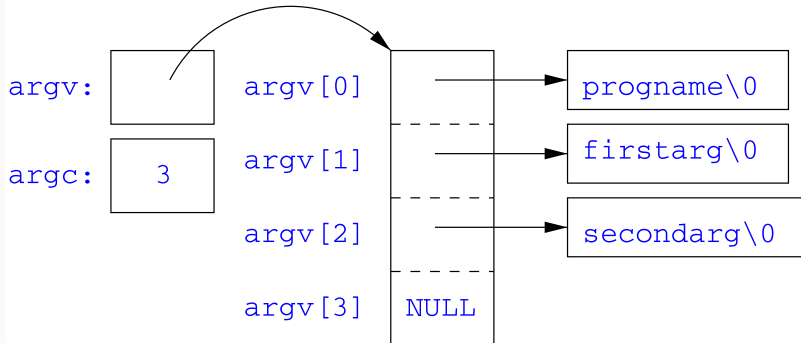
```
1  void swp1(int x,int y)
2  {
3    int temp = x;
4    x = y;
5    y = temp;
6  }
```

```
1  void swp2(int *px,int *py)
2  {
3    int temp = *px;
4    *px = *py;
5    *py = temp;
6  }
```

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

```
1  void sort(int a[], const int len,
2            int (*compare)(int, int)) {
3    for(int i = 0; i < len-1; i++)
4      for(int j = 0; j < len-1-i; j++)
5        if ((*compare)(a[j],a[j+1])) {
6          int tmp = a[j];
7          a[j] = a[j+1], a[j+1] = tmp;
8        }
9  }
10
11 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

```
1   #include <stdio.h>
2   #include "example8.h"
3
4   int main(void) {
5     int a[] = {1,4,3,2,5};
6     unsigned int len = 5;
7     sort(a,len,inc); //or sort(a,len,&inc);
8
9     int *pa = a; //C99
10    printf("[");
11    while (len--) { printf("%d%s", *pa++, len?" ":""); }
12    printf("]\n");
13
14    return 0;
15  }
```

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;
}
```

```
1 struct link {
2   int val;
3   struct link *next;
4 }
```

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

```
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:

```
1         typedef struct llist *llptr;
2         typedef struct llist {
3           int val;
4           llptr next;
5         } 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

```
1  #include <stdio.h>
2
3  char *unary(unsigned short s) {
4    char local[s+1];
5    int i;
6    for (i=0;i<s;i++) local[i]='1';
7    local[s]='\0';
8    return local;
9  }
10
11 int main(void) {
12
13   printf("%s\n",unary(6)); //What does this print?
14
15   return 0;
16 }
```

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

Tricks: Duff's device

```
1  send(int *to, int *from,
2       int count)
3  {
4    int n = (count+7)/8;
5    switch(count%8) {
6    case 0: do{ *to = *from++;
7    case 7:     *to = *from++;
8    case 6:     *to = *from++;
9    case 5:     *to = *from++;
10   case 4:     *to = *from++;
11   case 3:     *to = *from++;
12   case 2:     *to = *from++;
13   case 1:     *to = *from++;
14     } while(--n>0);
15   }
16 }
```

```
1  boring_send(int *to, int *from,
2              int count) {
3    do {
4      *to = *from++;
5    } while(--count > 0);
6  }
```

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

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

int main(int argc, char** argv) {
  int a[10];
  a[2] = 0;
  if (a[argc])
    printf("print something\n");
  return 0;
}
```

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

int main(int argc, char** argv) {
  int *a = malloc(sizeof(int) * 10);
  a[2] = 0;
  if (a[argc])
    printf("print something\n");
  free(a);
  return 0;
}
```

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

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = INT_MAX;
5    int m = n + 1;
6    return 0;
7  }
```

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

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5     int *pi = malloc(sizeof(int));
6     scanf("%d", pi);                 // Read an int
7     if (*pi % 2) {
8       printf("Odd!\n");
9       free(pi);                      // WRONG!
10    }
11  }
```

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

```
1   Tree *node(int value, Tree *left, Tree *right) {
2     Tree *t = malloc(sizeof(tree));
3     t->value = value;
4     t->right = right;
5     t->left = left;
6     return t;
7   }
8   void tree_free(Tree *tree) {
9     if (tree != NULL) {
10      tree_free(tree->left);
11      tree_free(tree->right);
12      free(tree);
13    }
14  }
```

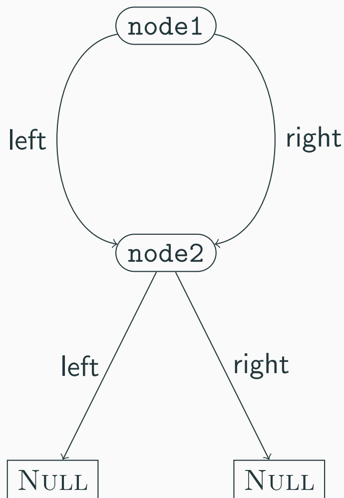
1. Free the left subtree

A Directed Acyclic Graph (DAG)

```
1     // Initialize node2
2     Tree *node2 = node(2, NULL, NULL);
3
4     // Initialize node1
5     Tree *node1 = node(1, node2, node2); // node2 repeated
6
7     // note node1->left == node1->right == node2!
```

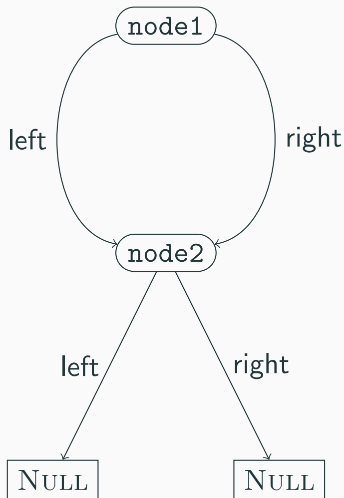
What kind of "tree" is this?

The shape of the graph



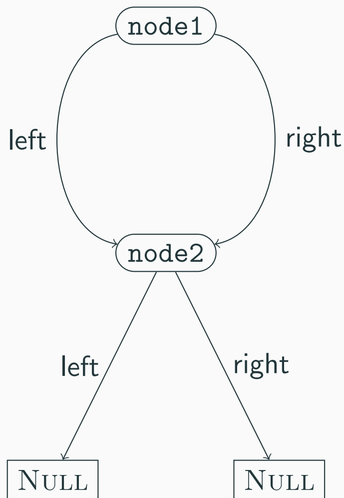
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- `tree_free(node1)` will call `tree_free(node2)` *twice*!

Evaluating `free(node1)`



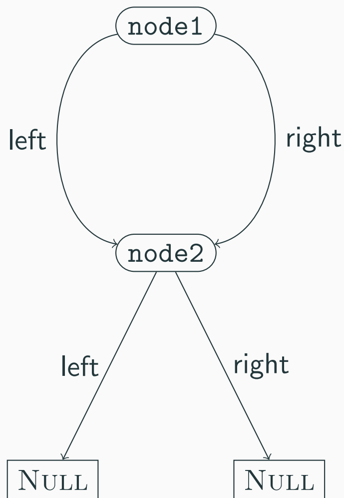

```
1    free(node1);
```

Evaluating `free(node1)`



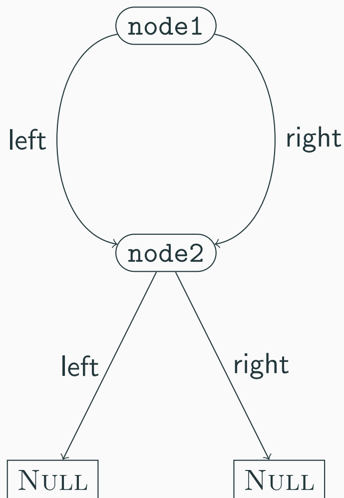

```
1  if (node1 != NULL) {
2    tree_free(node1->left);
3    tree_free(node1->right);
4    free(node1);
5  }
```

Evaluating `free(node1)`



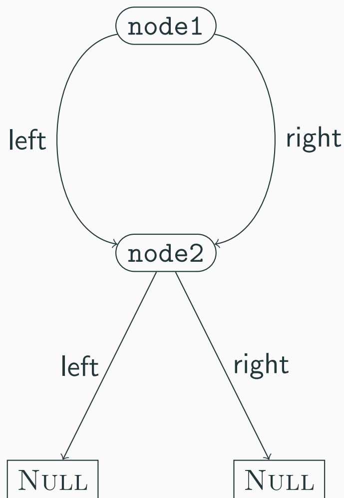

```
1    tree_free(node1->left);
2    tree_free(node1->right);
3    free(node1);
```

Evaluating `free(node1)`



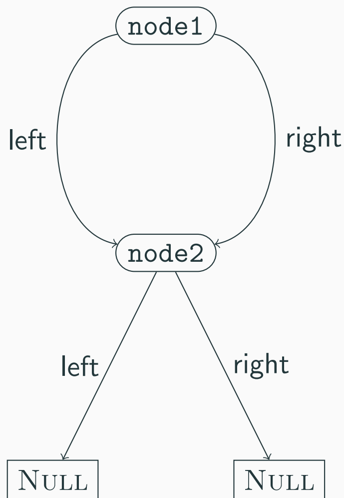

```
1    tree_free(node2);
2    tree_free(node2);
3    free(node1);
```

Evaluating `free(node1)`



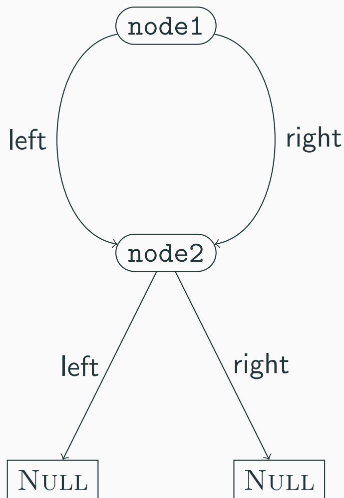

```
1   if (node2 != NULL) {
2     tree_free(node2->left);
3     tree_free(node2->right);
4     free(node2);
5   }
6   tree_free(node2);
7   free(node1);
```

Evaluating `free(node1)`



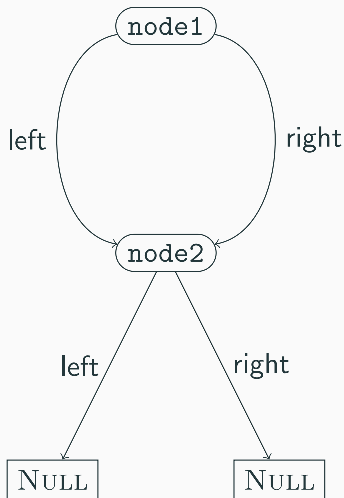

```
1      tree_free(node2->left);
2      tree_free(node2->right);
3      free(node2);
4      tree_free(node2);
5      free(node1);
```

Evaluating `free(node1)`



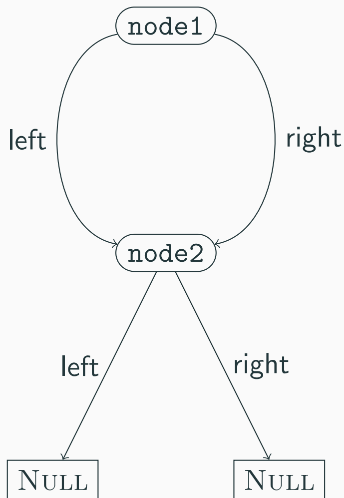

```
1    tree_free(NULL);
2    tree_free(NULL);
3    free(node2);
4    tree_free(node2);
5    free(node1);
```

Evaluating `free(node1)`



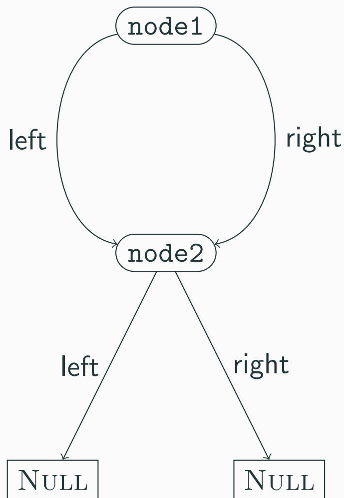

```
1    if (NULL != NULL) {
2      tree_free(NULL->left);
3      tree_free(NULL->right);
4      free(node1);
5    }
6    tree_free(NULL);
7    free(node2);
8    tree_free(node2);
9    free(node1);
```

Evaluating `free(node1)`



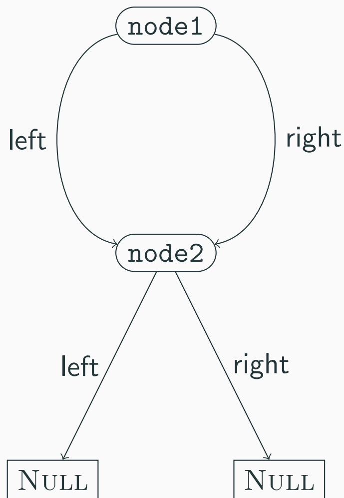

```
1   tree_free(NULL);
2   free(node2);
3   tree_free(node2);
4   free(node1);
```

Evaluating `free(node1)`



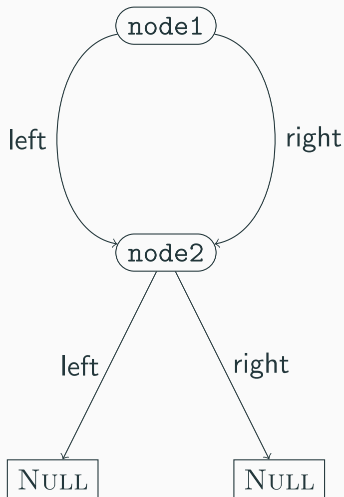

```
1    free(node2);
2    tree_free(node2);
3    free(node1);
```

Evaluating `free(node1)`




```
1    free(node2);
2    free(node2);
3    free(node1);
```

Evaluating `free(node1)`




```
1    free(node2);
2    free(node2);
3    free(node1);
```

`node2` is freed twice!

A Tree Data Type which Tracks Visits

```
1   struct node {
2     bool visited;
3     int value;
4     struct node *left;
5     struct node *right;
6   };
7   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

```
1    Tree *node(int value, Tree *left, Tree *right) {
2      Tree *t = malloc(sizeof(tree));
3      t->visited = false;
4      t->value = value;
5      t->right = right;
6      t->left = left;
7      return t;
8    }
```

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

```
1      typedef struct TreeListCell TreeList;
2      struct TreeListCell {
3        Tree *head;
4        TreeList *tail;
5      }
6      TreeList *cons(Tree *head, TreeList *tail) {
7        TreeList *result = malloc(TreeListCell);
8        result->head = head;
9        result->tail = tail;
10       return result;
11     }
```

- 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

```
1  TreeList *getNodes(Tree *tree, TreeList *nodes) {
2    if (tree == NULL || tree->visited) {
3      return nodes;
4    } else {
5      tree->visited = true;
6      nodes = cons(tree, nodes);
7      nodes = getNodes(tree->right, nodes);
8      nodes = getNodes(tree->left, nodes);
9      return nodes;
10   }
11 }
```

- 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

```
1     void tree_free(Tree *tree) {
2       NodeList *nodes = getNodes(tree, NULL);
3       while (nodes != NULL) {
4         Tree *head = nodes->head;
5         NodeList *tail = nodes->tail;
6         free(head);
7         free(nodes);
8         nodes = tail;
9       }
10    }
```

- 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

```c
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

```
1   typedef struct arena *arena_t;
2   struct arena {
3     int size;
4     int current;
5     Tree *elts;
6   };
7
8   arena_t make_arena(int size) {
9     arena_t arena = malloc(sizeof(struct arena));
10    arena->size = size;
11    arena->current = 0;
12    arena->elts = malloc(size * sizeof(Tree));
13    return arena;
14  }
```

- An *arena* is just a preallocated array of nodes

Arena allocation

```c
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

```
1   void free_arena(arena_t arena) {
2     free(arena->elts);
3     free(arena);
4   }
```

- 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

```
1 struct node {
2   int value;
3   struct node *left;
4   struct node *right;
5 };
6 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

```
1    Tree *node(int value, Tree *left, Tree *right) {
2      Tree *t = malloc(sizeof(tree));
3      t->value = value;
4      t->right = right;
5      t->left = left;
6      return t;
7    }
```

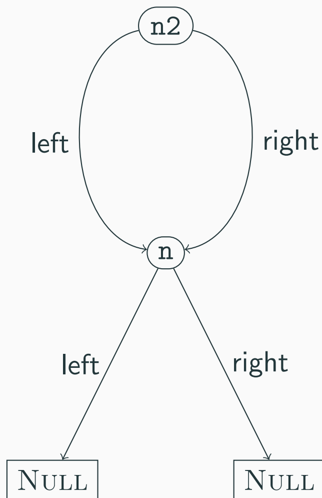
1. Allocate a pointer to a tree struct
2. Initialize the `value` field
3. Initialize the `right` field
4. Initialize the `left` field
5. Return the initialized pointer!

A Directed Acyclic Graph (DAG)

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

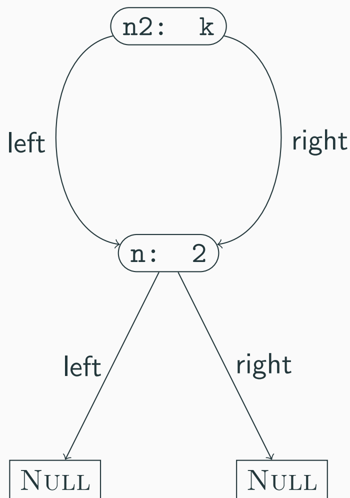
1. We allocate `n` on line 1
2. On line 2, we create `n2` whose `left` *and* `right` fields are `n`.
3. Hence `n2->left` and `n2->right` are said to *alias* – they are two pointers aimed at the same block of memory.

The shape of the graph



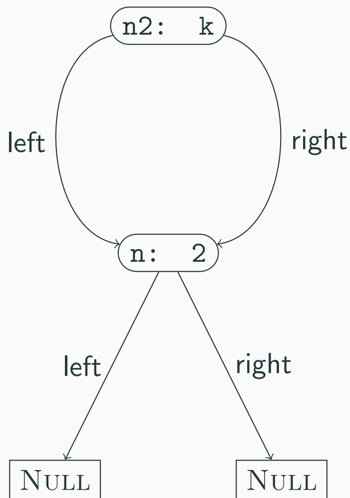
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- A recursive free of the tree `n2` will try to free `n` twice.

The Idea of Reference Counting



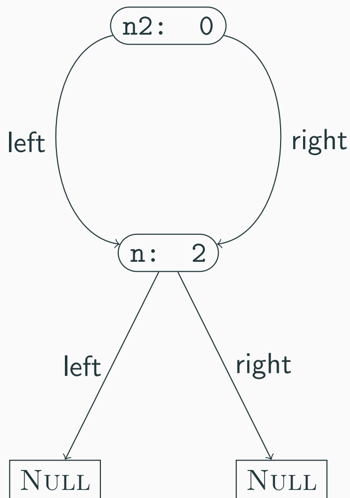
1. The problem: freeing things with two pointers to them twice
2. Solution: stop doing that
3. Keep track of the number of pointers to an object
4. Only free when the count reaches zero

How Reference Counting Works



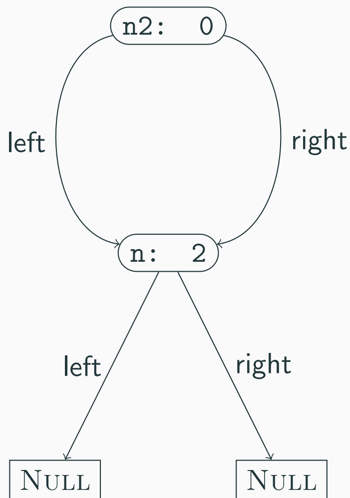
1. We start with k references to `n2`

How Reference Counting Works



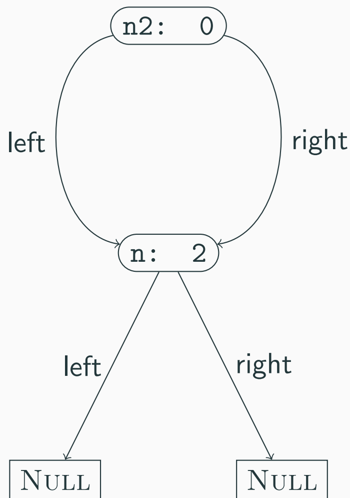
1. We start with k references to `n2`
2. Eventually k becomes 0

How Reference Counting Works



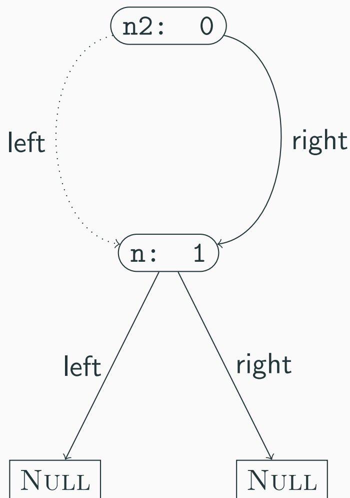
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`

How Reference Counting Works



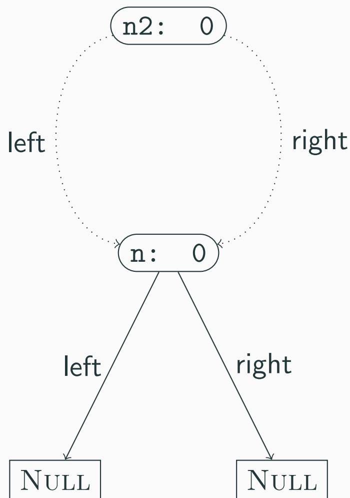
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



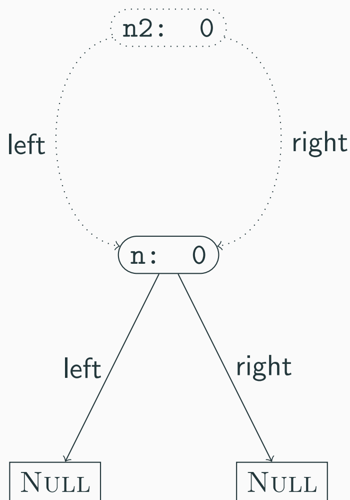
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



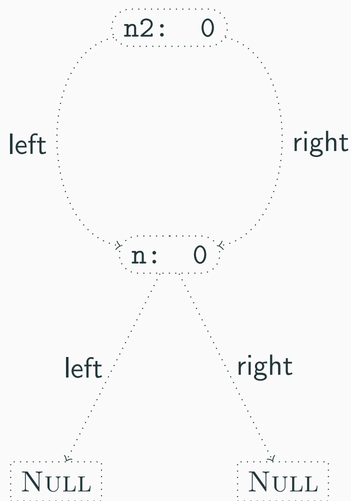
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`

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

```
1  struct node {
2    unsigned int rc;
3    int value;
4    struct node *left;
5    struct node *right;
6  };
7  typedef struct node Node;
8
9  const Node *empty = NULL;
10 Node *node(int value,
11            Node *left,
12            Node *right);
13 void inc_ref(Node *node);
14 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()`

```
Node *node(int value,
           Node *left,
           Node *right) {
  Node *r = malloc(sizeof(Node));
  r->rc = 1;
  r->value = value;

  r->left = left;
  inc_ref(left);

  r->right = right;
  inc_ref(right);
  return r;
}
```

- 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

```
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  }
```

- 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

```
1  Node *get_left(Node *node) {
2    inc_ref(node->left);
3    return(node->left);
4  }
5
6  void set_left(Node *node,
7                Node *newval) {
8    inc_ref(newval);
9    dec_ref(node->left);
10   node->left = newval;
11 }
```

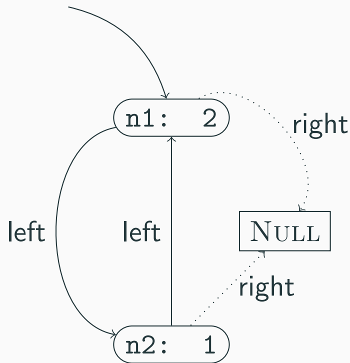
- The `get_left()` function returns the left subtree, but also increments the reference count
- The `set_left()` function updates the left subtree, incrementing the reference count to the new value and decrementing the reference

Cycles: A Fundamental Limitation on Reference Counting

```
1   Node *foo() {
2     Node *n1 = node(1, NULL, NULL);
3     Node *n2 = node(2, NULL, NULL);
4     set_left(n1, n2);
5     set_left(n2, n1);
6     dec_ref(n2);
7     return node1;
8   }
```

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

```
1   struct node {
2     int value;
3     struct node *left;
4     struct node *right;
5     bool mark;
6     struct node *next;
7   };
8   typedef struct node Node;
9
10  struct root {
11    Node *start;
12    struct root *next;
13  };
14  typedef struct root Root;
15
16  struct alloc {
17    Node *nodes;
18    Root *roots;
19  };
20  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

```
1   Alloc *make_allocator(void);
2   Node *node(int value,
3              Node *left,
4              Node *right,
5              Alloc *a);
6   Root *root(Node *node, Alloc *a);
7   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

```
1  void mark_node(Node *node) {
2    if (node != NULL && !node->mark) {
3      node->mark = true;
4      mark_node(node->left);
5      mark_node(node->right);
6    }
7  }
8
9  void mark(Alloc *a) {
10   Root *g = a->roots;
11   while (g != NULL) {
12     mark_node(g->start);
13     g = g->next;
14   }
15 }
```

- `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

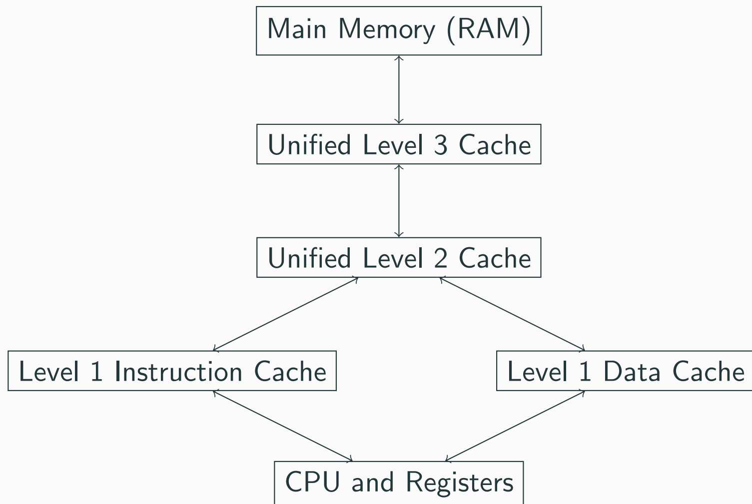
David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

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	$\approx$4	1.3 ns	1s
L2 cache reference	$\approx$10	4 ns	3s
L3 cache reference, unshared	$\approx$40	13 ns	10s
L3 cache reference, shared	$\approx$65	20 ns	16s
Main memory reference	$\approx$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

```c
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

```c
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

```c
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

```
1   #define SIZE 8192
2   #define dim(i, j) (((i) * SIZE) + (j))
3
4   double *add_transpose(double *A,
5                         double *B) {
6     double *dest =
7       malloc(sizeof(double)
8              * SIZE * SIZE);
9     for (int i = 0; i < SIZE; i++) {
10      for (int j = 0; j < SIZE; j++) {
11        dest[dim(i,j)] =
12          A[dim(i,j)] + B[dim(j,i)];
13      }
14    }
15    return dest;
16  }
```

- 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

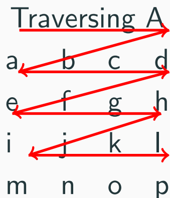
- A is a 3×4 array.
- $A(i, j)$ is at address $3 \times i + j$ (0 based!)
- E.g., $A(2, 1) = 49$, at address 7
- E.g., $A(3, 1) = 100$, at address 10

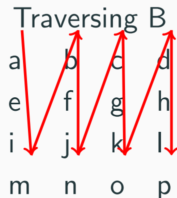
Loop Blocking

```
1   #define SIZE 8192
2   #define dim(i, j) (((i) * SIZE) + (j))
3
4   double *add_transpose(double *A,
5                         double *B) {
6     double *dest =
7       malloc(sizeof(double)
8              * SIZE * SIZE);
9     for (int i = 0; i < SIZE; i++) {
10      for (int j = 0; j < SIZE; j++) {
11        dest[dim(i,j)] =
12          A[dim(i,j)] + B[dim(j,i)];
13      }
14    }
15    return dest;
16  }
```

- 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

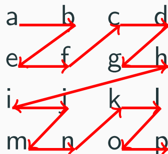
How to Block a Loop, Concept

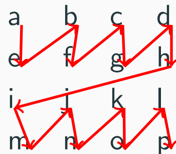




- We can see that A has a favorable traversal, and B is “jumpy”
- Let’s change the traversal order!

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

```
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:

```
1  #include <netdb.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5    struct hostent *hp;
6    hp = gethostbyname("doesntexist.abc");
7    printf("%s\n", hp->h_name);
8    return 0;
9  }
```

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.

```
1   #include <netdb.h>
2   #include <stdio.h>
3
4   int main(int argc, char **argv) {
5     struct hostent *hp;
6     printf("argc: %d\n", argc);
7     for (int i=1; i<argc; i++) {
8       hp = gethostbyname(argv[i]);
9       printf("hp: %p\n", hp);
10      if (hp)
11       printf("%s\n", hp->h_name);
12    }
13    return 0;
14  }
```

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.)

```
1   #include <netdb.h>
2   #include <stdio.h>
3
4   int main(int argc, char **argv) {
5     struct hostent *hp;
6     printf("%s:%2d argc: %d\n", __FILE__, __LINE__, argc);
7     for (int i=1; i<argc; i++) {
8       hp = gethostbyname(argv[i]);
9       printf("%s:%2d hp: %p\n", __FILE__, __LINE__, hp);
10      fflush(stdout);
11      printf("%s:%2d %s\n", __FILE__, __LINE__, hp->h_name);
12      fflush(stdout);
13    }
14    return 0;
15  }
```

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.

```
1 #ifndef DEBUG
2 #define DEBUG 0
3 #endif
4 #define debug_printf(fmt, ...) \
5   do { if (DEBUG) { \
6     fprintf(stderr, fmt, __VA_ARGS__); \
7     fflush(stderr); } } \
8   while (0)
```

Debugging via printing values (cont.)

```
1   #include <netdb.h>
2   #include <stdio.h>
3   #ifndef DEBUG
4   #define DEBUG 0
5   #endif
6   #define debug_printf(fmt, ...) \
7     do { if (DEBUG) { fprintf(stderr, fmt, __VA_ARGS__); \
8                       fflush(stderr); } } while (0)
9   int main(int argc, char **argv) {
10    debug_printf("argc: %d\n", argc);
11    for (int i=1; i<argc; i++) {
12      struct hostent *hp = gethostbyname(argv[i]);
13      debug_printf("hp: %p\n", hp);
14      printf("%s\n", hp->h_name);
15    }
16    return 0;
17  }
```

Debugging via Assertions

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

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #include <assert.h> // new header file
4
5  int main(int argc, char **argv) {
6    struct hostent *hp;
7    hp = gethostbyname("doesntexist.abc");
8    assert(hp != NULL); // new invariant
9    printf("%s\n", hp->h_name);
10   return 0;
11 }
```

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.

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #include <assert.h>
4  int main(int argc, char **argv) {
5    struct hostent *hp;
6    hp = gethostbyname(argv[1]);
7    printf("%s\n", hp->h_name);
8    return 0;
9  }
```

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.

```
1  #include <netdb.h>
2  #include <stdio.h>
3  #include <assert.h>
4  int main(int argc, char **argv) {
5    struct hostent *hp;
6    assert(argv[1] != NULL);
7    hp = gethostbyname(argv[1]);
8    assert(hp != NULL);
9    printf("%s\n", hp->h_name);
10   return 0;
11 }
```

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:

```
#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.

```
#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.

Programming in C and C++

Lectures 10–12: C++ for Java and C programmers

D. J. Greaves[1]

Computer Laboratory, University of Cambridge

Michaelmas Term 2023/24

[1] Notes based, with thanks, on slides due to Alan Mycroft, Alastair Beresford and Andrew Moore.

Aims of C++

To quote Bjarne Stroustrup:

"C++ is a general-purpose programming language with a bias towards systems programming that:

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming."

Alternatively: C++ is "an (almost upwards-compatible) extension of C with support for: classes and objects (including multiple inheritance), call-by-reference, operator overloading, exceptions and templates (a richer form of generics)".

Much is familiar from Java, but with many subtle differences.

What we'll cover

- Differences between C and C++
- References versus pointers
- Overloaded functions and operators
- Objects in C++; Classes and structs; Destructors; Virtual functions
- Multiple inheritance; Virtual base classes; Casts
- Exceptions
- Templates and metaprogramming

- For exam purposes, focus on 'big-picture' novelties and differences between features of C++ and those in C and Java.
- For coding, sorry but compilers insist you get it exactly right.

Reference sources

C++ is a big language with many subtleties. The current draft C++20 standard is 1841 pages (457 for the C++ language and 1152 for the C++ Standard Library; the grammar alone is 21 pages)!

https://isocpp.org/ The ISO standard. Published standards cost money but draft standards are free online, e.g. draft C++20 on `https://isocpp.org/files/papers/N4860.pdf`

https://cppreference.com Wiki-book attempt to track standard.

https://learncpp.com More-chatty tutorial-style articles.

https://www.stroustrup.com Entertaining and educational articles by the creator of C++.

These are useful when wanting to know more about exactly how things (e.g. lambdas, overloading resolution) work, they are not necessary for exam purposes!

How to follow these three lectures

- These slides try capture the core features of C++, so that afterwards you will be able to read C++ code, and tentatively modify it.
  The Main ISO C++ versions are: C++98, C++11, C++20; we'll focus on core features—those in C++98.
- But C++ is a very complex language, so these slides are incomplete, even if they uncomfortably large.
- For exam purposes the fine details don't matter, it's more important to get the big picture, which I'll try to emphasise in lectures.

Should I program my application in C or C++?

Or both or neither?

- One aim of these lectures is to help you decide.
- C and C++ both have very good run-time performance
- C++ has more facilities, but note Bjarne Stroustrup's quote:
  "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."
- Even if C++ is a superset of C then mixing code is risky, e.g.
  - you don't want two conflicting IO libraries being active,
  - you often program using different metaphors in C and C++
  - C functions may not expect an exception to bypass their tidy-up code
  - Using C-coded stand-alone libraries in C++ is fine.
- C++ vs. Java? Speed vs. safety? More vs. fewer features? Java is trying to follow C++ (and C#) by having value types (objects/structs as values not just references).

Decide C <u>or</u> C++ at the start of a project.

C++ Types [big picture]

C++ types are like C types, but additionally:

- character literals (e.g. `'a'`) are type `char` (but `int` in C)
- new type `bool` (values `true` and `false`)
- reference types: new type constructor `&`, so can have `int x, *y, &z;`
- `enum` types are distinct (not just synonyms for integers)
- new type constructor `class` (generalising `struct` in C)
- names for `enum`, `class`, `struct` and `union` can be used directly as types (C needs an additional `typedef`)
- member functions (methods) can specify `this` to be `const`.

Many of the above changes are 'just what you expect from programming in Java'.

C++ auto and thread_local

C's storage classes are `auto`, `extern`, `static`, `register`. In C++:

- `auto` is reused in initialised definitions to mean 'the type of the initialising expression', e.g. `auto x = foo(3);`
- `thread_local` is an additional storage class, e.g. `static int x = 4; thread_local int y = 5;`
- `register` is removed since C++17.

C++ booleans

- type `bool` has two values: `true` and `false`
- When cast to an integer, `true`$\rightarrow$`1` and `false`$\rightarrow$`0`
- When casting from an integer, non-zero values become `true` and zero becomes `false` (NB: differs from `enum`, see next slide).

C++ enumeration

- Unlike C, C++ enumerations define a new type; for example

```
enum flag {is_keyword=1, is_static=2, is_extern=4, ... }
```

- When defining storage for an instance of an enumeration, you use its name; for example: `flag f = is_keyword;`
- Implicit type conversion is not allowed:

```
f = 5; //wrong        f = flag(5); //right(!!)
```

- Subtlety: Why is 5 'right' (but 8 would be wrong)? Answer: C++ rules to ensure 'bitmaps' work:
  - The maximum valid value of an enumeration is the enumeration's largest value rounded up to the nearest larger binary power minus one
  - The minimum valid value of an enumeration with no negative values is zero
  - The minimum valid value of an enumeration with negative values is the nearest least negative binary power

References

C++ references provide an alternative name (alias) for a variable

- Generally used for specifying parameters to functions and return values as well as overloaded operators (more later)
- A reference is declared with the `&` operator; compare:
  `int i[] = {1,3}; int &refi = i[0]; int *ptri = &i[0];`
- A reference must be initialised when it is declared
- The connection between a reference and what it refers to cannot be changed after initialisation; for example:

  ```
  refi++; // increments value referenced to 2
  ptri++; // increments the pointer to &i[1]
  ```

Think of reference types as pointer types with implicit `*` at every use.
Subtlety (non-examinable): C++11 added 'rvalue references', e.g. `int &&lvr`, useful in copy constructors (see later).

References in function arguments

- When used as a function parameter, a referenced value is not copied; for example:
  `void inc(int& i) { i++;}`
- Declare a reference as `const` when no modification takes place
- It can be noticeably more efficient to pass a large struct by reference
- Implicit type conversion into a temporary takes place for a `const` reference but results in an error otherwise; for example:

```
1 float fun1(float&);
2 float fun2(const float&);
3 void test() {
4   double v=3.141592654;
5   fun1(v); // Wrong
6   fun2(v); // OK, but beware the temporary's lifetime.
7   fun1((float)v); /* Wrong: a non-const reference should
8   not reference a temporary. */
9 }
```

- Cf. Fortran call-by-reference

Overloaded functions

- Just like Java we can define two functions with the same name, but varying in argument types (for good style functions doing different things should have different names).
- Type conversion is used to find the "best" match
- A best match may not always be possible:

```
1 void f(double);
2 void f(long);
3 void test() {
4   f(1L);  // f(long)
5   f(1.0); // f(double)
6   f(1);   // Wrong: f(long(1)) or f(double(1)) ?
```

- Can also overload built-in operators, such as assignment and equality.

Applies both to top-level functions and member functions (methods).

Scoping and overloading

- Overloading does not apply to functions declared in different scopes; for example:

```
1 void f(int);
2
3 void example() {
4   void f(double);
5   f(1); //calls f(double);
6 }
```

Default function arguments

- A function can have default arguments; for example:
```
double log(double v, double base=10.0);
```
- A non-default argument cannot come after a default; for example:
```
double log(double base=10.0, double v); //wrong
```
- A declaration does not need to name the variable; for example:
```
double log(double v, double=10.0);
```
- Be careful of the lexical interaction between `*` and `=`; for example:
```
void f(char*=0); // Wrong: '*=' is assignment
```

Namespaces

Related data can be grouped together in a namespace. Can use `::` and `using` to access components. Think Java packages.

```
namespace Stack { //header file
  void push(char);
  char pop();

}
```

```
void f() { //usage
  ...
  Stack::push('c');
  ...
}
```

```
namespace Stack { //implementation
  const int max_size = 100;
  char s[max_size];
  int top = 0;

  void push(char c) { ... }
  char pop() { ... }
}
```

Example

```
1  namespace Module1 {int x;}
2
3  namespace Module2 {
4    inline int sqr(const int& i) {return i*i;}
5    inline int halve(const int& i) {return i/2;}
6  }
7
8  using namespace Module1; //"import" everything
9
10 int main() {
11   using Module2::halve;  //"import" the halve function
12   x = halve(x);
13   sqr(x);                //Wrong
14 }
```

(Non-examinable: C++20 adds `module` constructs giving more control over name visibility. Think Java 9 ‘modules’, while namespaces are more like Java ‘packages’.)

Using namespaces

- A namespace is a scope and expresses logical program structure
- It provides a way of collecting together related pieces of code
- A namespace without a name limits the scope of variables, functions and classes within it to the local execution unit
- The same namespace can be declared in several source files
- A namespace can be defined more than once
  - Allows, for example, internal and external library definitions
- The use of a variable or function name from a different namespace must be qualified with the appropriate namespace(s)
  - The keyword `using` allows this qualification to be stated once, thereby shortening names
  - Can also be used to generate a hybrid namespace
  - `typedef` can be used: `typedef Some::Thing thing;`
- The global function `main()` cannot be inside a namespace

Linking C and C++ code

- The directive `extern "C"` specifies that the following declaration or definition should be linked as C, not C++, code:
  `extern "C" int f();`
- Multiple declarations and definitions can be grouped in curly brackets:

```
1 extern "C" {
2   int globalvar; //definition
3   int f();
4   void g(int);
5 }
```

Why do we need this?

- 'Name mangling' for overloaded functions. A C compiler typically generates linker symbol '`_f`' for `f` above, but (in the absence of `extern "C"`) a C++ compiler typically generates '`__Z1fv`'.
- Function calling sequences may also differ (e.g. for exceptions).

Linking C and C++ code

- What if I want to write a library in C, and specify it via `mylib.h` which is importable into both C and C++?
- Use conditional compilation (`#ifdef`) in `mylib.h`, e.g.

```
1 #ifdef __cplusplus
2    extern "C" void myfn(int, bool);
3 #else
4 #  include <stdbool.h>  // Ensure type bool defined in C
5    extern void myfn(int, bool);
6 #endif
```

Linking C and C++ code

- Care must be taken with pointers to functions and linkage:

```
1 extern "C" void qsort(void* p, \
2                     size_t nmemb, size_t size, \
3                     int (*compar)(const void*, const void*));
4
5 int compare(const void*,const void*);
6
7 char s[] = "some chars";
8 qsort(s,9,1,compare); //Wrong
```

Big Picture

So far we've only done minor things.

- We've seen C++ extensions to C. But, apart from reference types, nothing really new has appeared that's beyond Java concepts.
- Now for classes and objects, which look the same, but aren't ...

Classes and objects in C++

C++ classes are somewhat like Java:

- Classes contain both data members and member functions (methods) which act on the data; they can extend (syntax ‘`:`’) other classes.
- Members can be `static` (i.e. per-class)
- Members have access control: `private`, `protected` and `public`
- Classes are created with `class` or `struct` keywords
  - `struct` members default to `public` access; `class` to `private`
- A member function with the same name as a class is called a constructor
- Can use overloading on constructors and member functions.

But also:

- A member function with the same name as the class, prefixed with a tilde (`~`), is called a destructor

Classes and objects: big differences from Java

- Values of class types are not references to objects, but the objects themselves. So we access members with C-style '`.`' (but using '`->`' is more convenient when we have pointers to objects).
- We can create an object of class `C`, either by:
  - on the stack (or globally) by declaring a variable: `C x;`
  - on the heap: `new C()` (returns a pointer to `C`)
- Member functions (methods) by default are statically resolved. For Java-like code declare them `virtual`
- Member functions can be declared inside a class but defined outside it using '`::`' (the scope-resolution operator)
- C++ uses `new` to allocate and `delete` to de-allocate. There is no garbage collector—users must de-allocate heap objects themselves.

Example (emphasising differences from Java)

```
1  class Complex {
2    double re, im;  // private by default
3   public:
4    Complex(double r=0.0, double i=0.0);
5  };
6
7  Complex::Complex(double r,double i) : re(r), im(i) {
8    // preferred form, necessary for const fields
9  }
10
11 Complex::Complex(double r,double i) {
12   re=r, im=i;  // deprecated initialisation-by-assignment
13 }
14
15 int main() {
16   Complex c(2.0), d(), e(1,5.0);
17   return 0;
18 } // local objects c,d,e are deallocated on scope exit
```

New behaviours w.r.t. Java

In Java constructors are only used to initialise heap storage, and the only way we can update a field of an object is by `x.f = e;`.

In C++ having object values as first-class citizens gives more behaviours. Consider the following, given `class C`

```
C x;      // how is x initialised? (default constructor)
C y = x; // how is y initialised? (copy constructor)
x = y;    // what does the assignment do? (assignment operator)
          // what happens to x,y on scope exit? (destructor)
```

For C `struct`s, these either perform bit copies or leave `x` uninitialised.

C++ class definitions may need to control the above behaviours to preserve class invariants and object encapsulation.

Constructors and destructors

- A default constructor is a function with no arguments (or only default arguments)
- The programmer can specify one or more constructors, but as in Java, only one is called when an object is created.
- If no constructors are specified, the compiler generates a default constructor (which does does as little initialisation as possible).
- To forbid users of a class from using a default constructor then define it explicitly and declare it `private`.
- There can only be one destructor
  - This is called when a stack-allocated object goes out of scope (including when an exception causes this to happen—see later) or when a heap-allocated object is deallocated with `delete`;
  - Stack-allocated objects with destructors are a useful way to release resources on scope exit (similar effect as Java try-finally) – "RAII: Resource Acquisition is Initialisation".
  - Make destructors virtual if class has subtypes or supertypes.

Copy constructor

- A new class instance can be defined by initialisation; for example:

```
1 Complex c(1,2); // note this C++ initialiser syntax;
2                 // it calls the two-argument constructor
3 Complex d = c;  // copy constructor called
```

- In the second case, by default object `d` is initialised with copies of all of the non-static member variables of `c`; no constructor is called
- If this behaviour is undesirable (e.g. consider a class with a pointer as a member variable) define your own copy constructor:
  - `Complex::Complex(const Complex&) { ... }`
- To forbid users of a class from copying objects, make the copy constructor a private member function, or in C++11 use `delete`.
- Note that assignment, e.g. `d = c;` differs differs from initialisation and does not use the copy constructor—see next slide.

Assignment operator

- By default a class is copied on assignment by over-writing all non-static member variables; for example:

```
1 Complex c(), d(1.0,2.3);
2 c = d; //assignment
```

- This behaviour may also not be desirable (e.g. you might want to tidy up the object being over-written).
- The assignment operator (`operator=`) can be defined explicitly:

```
1 Complex& Complex::operator=(const Complex& c) {
2   ...
3 }
```

- Note the result type of assignment, and the reference-type parameter (passing the argument by value would cause a copy constructor to be used before doing the assignment, and also be slower).

Constant member functions

- Member functions can be declared `const`
- Prevents object members being modified by the function:

```
1 double Complex::real() const {
2   // forbidden to modify 're' or `this->re' here
3   return re;
4 }
```

- The syntax might appear odd at first, but note that `const` above merely qualifies the (implicit/hidden) parameter '`this`'. So here `this` is effectively declared as `const Complex *this` instead of the usual `Complex *this`.
- Helpful to both programmer (maintenance) and compiler (efficiency).

Arrays and heap allocation

- An array of class objects can be defined if a class has a default constructor
- C++ has a `new` operator to place items on the heap:
  `Complex* c = new Complex(3.4);`
- Items on the heap exist until they are explicitly deleted:
  `delete c;`
- Since C++ (like C) doesn't distinguish between a pointer to a single object and a pointer to an the first element of an array of objects, array deletion needs different syntax:

```
1 Complex* c = new Complex[5];
2   ...
3 delete[] c; //Using "delete" is wrong here
```

- When an object is deleted, the object destructor is invoked
- When an array is deleted, the object destructor is invoked on each element

Exercises

1. Write an implementation of a class `LinkList` which stores zero or more positive integers internally as a linked list on the heap. The class should provide appropriate constructors and destructors and a method `pop()` to remove items from the head of the list. The method `pop()` should return -1 if there are no remaining items. Your implementation should override the copy constructor and assignment operator to copy the linked-list structure between class instances. You might like to test your implementation with the following:

```
int main() {
  int test[] = {1,2,3,4,5};
  LinkList l1(test+1,4), l2(test,5);
  LinkList l3=l2, l4;
  l4=l1;
  printf("%d %d %d\n",l1.pop(),l3.pop(),l4.pop());
return 0;
}
```

Hint: heap allocation & deallocation should occur exactly once!

Operators

- C++ allows the programmer to overload the built-in operators
- For example, a new test for equality:

```
1 bool operator==(Complex a, Complex b) {
2   return a.real()==b.real() && a.imag()==b.imag();
3   // presume real() is an accessor for field 're', etc.
4 }
```

- An operator can be defined or declared within the body of a class, and in this case one fewer argument is required; for example:

```
1 bool Complex::operator==(Complex b) {
2   return re==b.real() && im==b.imag();
3 }
```

- Almost all operators can be overloaded, including address-taking, assignment, array indexing and function application.
  It's probably bad practice to define `++x` and `x+=1` to have different meanings!

Streams

- Overloaded operators also work with built-in types
- Overloading is used to define `<<` (C++'s "`printf`"); for example:

```
1  #include <iostream>
2
3  int main() {
4    const char* s = "char array";
5
6    std::cout << s << std::endl;
7
8    //Unexpected output; prints &s[0]
9    std::cout.operator<<(s).operator<<(std::endl);
10
11   //Expected output; prints s
12   std::operator<<(std::cout,s);
13   std::cout.operator<<(std::endl);
14   return 0;
15 }
```

- Note `std::cin`, `std::cout`, `std::cerr`

The ‘this’ pointer

- If an operator is defined in the body of a class, it may need to return a reference to the current object
  - The keyword `this` can be used
- For example:

```
1 Complex& Complex::operator+=(Complex b) {
2   re += b.real();
3   this->im += b.imag();
4   return *this;
5 }
```

- In C (or assembler) terms `this` is an implicit argument to a method when seen as a function.

Class instances as member variables

- A class can have an instance of another class as a member variable
- How can we pass arguments to the class constructor?
- New C++ syntax for constructors:

```
1 class Z {
2   Complex c;
3   Complex d;
4   Z(double x, double y): c(x,y), d(y) {
5     ...
6   }
7 };
```

- This notation must be used to initialise const and reference members
- It can also be more efficient

Temporary objects

- Temporary objects are often created during execution
- A temporary which is not bound to a reference or named object exists only during evaluation of a full expression (BUGS BUGS BUGS!)
- Example: the C++ `string` class has a function `c_str()` which returns a pointer to a C representation of a string:

```
string a("A "), b("string");
const char *s1 = a.c_str();      //OK
const char *s2 = (a+b).c_str(); //Wrong
 ...
//s2 still in scope here, but the temporary holding
//"a+b" has been deallocated
 ...
string tmp = a+b;
const char *s3 = tmp.c_str();   //OK
```

[Non-examinable:] C++11 added rvalue references ‘`&&`’ to help address this issue.

Friend Classes and Functions

- If, within a class `C`, the declaration `friend class D;` appears, then `D` is allowed to access the private and protected members of `C`.
- A top-level function can be declared `friend` to allow it to access the private and protected members of the enclosing class, e.g.

```
1 class Matrix {
2   ...
3   friend Vector multiply(const Matrix&, const Vector&);
4    ...
5   };
6 }
```

  This code allows `multiply` to access the private fields of `Matrix`, even though it is defined elsewhere.
- Mental model: granting your lawyer rights to access your private papers. Note that friendship isn't symmetric.

Inheritance

- C++ allows a class to inherit features of another:

```
class vehicle {
  int wheels;
public:
  vehicle(int w=4):wheels(w) {}
};

class bicycle : public vehicle {
  bool panniers;
public:
  bicycle(bool p):vehicle(2),panniers(p) {}
};

int main() {
  bicycle(false);
}
```

Derived member function call

I.e. when we call a function overridden in a subclass.

- Default derived member function call semantics differ from Java:

```
1  // example13.hh
2
3  class vehicle {
4    int wheels;
5  public:
6    vehicle(int w=4):wheels(w) {}
7    int maxSpeed() {return 60;}
8  };
9
10 class bicycle : public vehicle {
11   bool panniers;
12 public:
13   bicycle(bool p=true):vehicle(2),panniers(p) {}
14   int maxSpeed() {return panniers ? 12 : 15;}
15 };
```

Example

```
1 #include <iostream>
2 #include "example13.hh"
3
4 void print_speed(vehicle &v, bicycle &b) {
5   std::cout << v.maxSpeed() << " ";
6   std::cout << b.maxSpeed() << std::endl;
7 }
8
9 int main() {
10   bicycle b = bicycle(true);
11   print_speed(b,b); //prints "60 12"
12 }
```

Virtual functions

- Non-virtual member functions are called depending on the static type of the variable, pointer or reference
- Since a pointer to a derived class can be cast to a pointer to a base class, calls at base class do not see the overridden function.
- To get polymorphic behaviour, declare the function `virtual` in the superclass:

```
1 class vehicle {
2   int wheels;
3  public:
4   vehicle(int w=4):wheels(w) {}
5   virtual int maxSpeed() {return 60;}
6 };
```

Virtual functions

- In general, for a virtual function, selecting the right function has to be run-time decision; for example:

```
1 bicycle b(true);
2 vehicle v;
3 vehicle* pv;
4
5 user_input() ? pv = &b : pv = &v;
6
7 std::cout << pv->maxSpeed() << std::endl;
8 }
```

Enabling virtual functions

- To enable virtual functions, the compiler generates a <u>virtual function table</u> or <u>vtable</u>
- A vtable contains a pointer to the correct function for each object instance
- Indirect (virtual) function calls are slower than direct function calls.
- Question: virtual function calls are compulsory in Java; is C++'s additional choice of virtual/non-virtual calls good for efficiency or bad for being an additional source of bugs?
- C++ vtables also contain an encoding of the class type: 'run-time type information' (RTTI). Syntax `typeid(e)` gives the type of `e` encoded as an object of `type_info` which is defined in standard header `<typeinfo>`.

Abstract classes

- Just like Java except for syntax.
- Sometimes a base class is an un-implementable concept
- In this case we can create an abstract class:

```
1 class shape {
2  public:
3   virtual void draw() = 0;
4 }
```

- It is forbidden to instantiate an abstract class:
  `shape s; //Wrong`
- A derived class can provide an implementation for some (or all) the abstract functions
- A derived class with no abstract functions can be instantiated
- C++ has no equivalent to Java '`implements interface`'.

Example

```
1 class shape {
2 public:
3   virtual void draw() = 0;
4 };
5
6 class circle : public shape {
7 public:
8   //...
9   void draw() { /* impl */ }
10 };
```

Multiple inheritance

- It is possible to inherit from multiple base classes; for example:

```
1 class ShapelyVehicle: public vehicle, public shape {
2  ...
3 }
```

- Members from <u>both</u> base classes exist in the derived class
- If there is a name clash, explicit naming is required
- This is done by specifying the class name; for example:
  `ShapelyVehicle sv;`
  `sv.vehicle::maxSpeed();`

Multiple instances of a base class

- With multiple inheritance, we can build:

```
1 class A { int var; };
2 class B : public A {};
3 class C : public A {};
4 class D : public B, public C {};
```

- This means we have two instances of `A` even though we only have a single instance of `D`
- This is legal C++, but means all accesses to members of `A` within a `D` must be stated explicitly:

```
1 D d;
2 d.B::var=3;
3 d.C::var=4;
```

Virtual base classes

- Alternatively, we can have a single instance of the base class
- Such a "virtual" base class is shared amongst all those deriving from it

```
1 class Vehicle {int VIN;};
2 class Boat : public virtual Vehicle { ... };
3 class Car : public virtual Vehicle { ... };
4 class JamesBondCar : public Boat, public Car { ... };
```

- Multiple inheritance is often regarded as problematic, and one of the reasons for Java creating `interface`.

Casts in C++

- In C, casts play multiple roles, e.g. given `double *p`

```
1 int i = (int)*p;     // well-defined, safe
2 int j = *(int *)p;   // undefined behaviour
```

- In C++ the role of constructors and casts overlap. Given `double x` consider (slide 25 defines `Complex`):

```
1 Complex c1(x,0);     // C++ initialisation syntax
2 Complex c2 = Complex(x); // beware (two constructors?)
3 Complex c3 = x;      // OK, but 'explicit' would forbid
4 int i0 = (int)x;     // classic C syntax
5 int i1(x);           // C++ initialisation syntax
6 int i2 = int(x);     // C++ constructor syntax for cast
7 int i3 = x;          // implicit cast
```

- `c3` is OK—the `Complex` constructor can take one argument. Declare the constructor `explicit` if you want to disallow `c3` (but not `c2`). Compare `i3`, some languages might forbid this.

Casts from a class type

What if I want to write either of the following:

```
1 Complex c;
2 double d1 = (double)c;  // explicit cast
3 double d2 = c;          // implicit cast
```

These are faulted by the type checker.

Answer: overload `operator double()` for class `Complex`:

```
1 Class Complex {
2   ...
3   operator double() const { return re; }
4 }
```

Adding qualifier `explicit` requires casts to be explicit, allowing `d1` but forbidding `d2`.

Casts in C++ (new forms)

Downsides of C-style casts:

- hard to find (and classify) using a text editor in C or Java.
- they do no checking (cf. Java downcasts)

C++ encourages the more-descriptive forms:

- `dynamic_cast<T>(e)`: like Java reference casts: run-time checks when casting pointers within an inheritance hierarchy. This uses RTTI.
- `static_cast<T>(e)`: nearest to C—best efforts at compile time, e.g. `static_cast<int>(3.14)`.
- `reinterpret_cast<T>(e)`: to explicitly flag re-use of bit patterns.
- `const_cast<T>(e)`: remove `const` (or `volatile`) from a type, to modify something the type says you can't!

Pointer casts and multiple inheritance

C-style casts `(C1 *)p` (and indeed `static_cast<C1 *>(p)`) are risky in an inheritance hierarchy when multiple inheritance or virtual bases are used; the compiler must be able to see the inheritance tree otherwise it might not compile the right operation (casting to a superclass might require an addition or indirection).

Java single inheritance means that storage for a base class is always at offset zero in any subclass, making casting between references a no-op (albeit with a run-time check for a downcast).

Exercises

1. If a function `f` has a static instance of a class as a local variable, when might the class constructor be called?
2. Write a class `Matrix` which allows a programmer to define 2×2 matrices. Overload the common operators (e.g. `+`, `-`, `*`, and `/`)
3. Write a class `Vector` which allows a programmer to define a vector of length two. Modify your `Matrix` and `Vector` classes so that they inter-operate correctly (e.g. `v2 = m*v1` should work as expected)
4. Why should destructors in an abstract class almost always be declared `virtual`?

Exceptions

Just like Java, but you normally throw an object value rather than an object reference:

- Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it
- C++ provides exceptions to allow an error to be communicated
- In C++ terminology, one portion of code throws an exception; another portion catches it.
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception
- If an exception is not caught, the program terminates

C++ has no try-finally (use local variables having destructors – RAII).

Throwing exceptions

- Exceptions in C++ are just normal values, matched by type
- A class is often used to define a particular error type:
  `class MyError {};`
- An instance of this can then be thrown, caught and possibly re-thrown:

```
1 void f() { ... throw MyError(); ... }
2 ...
3     try {
4       f();
5     }
6     catch (MyError) {
7       //handle error
8       throw; //re-throw error
9     }
```

Conveying information

- The “thrown” type can carry information:

```
1  struct MyError {
2    int errorcode;
3    MyError(i):errorcode(i) {}
4  };
5
6  void f() { ... throw MyError(5); ... }
7
8  try {
9    f();
10 }
11 catch (MyError x) {
12   //handle error (x.errorcode has the value 5)
13   ...
14 }
```

Handling multiple errors

- Multiple catch blocks can be used to catch different errors:

```
1 try {
2   ...
3 }
4 catch (MyError x) {
5   //handle MyError
6 }
7 catch (YourError x) {
8   //handle YourError
9 }
```

- The wildcard syntax `catch(...)` catches all exceptions but discouraged in practice (what have you caught?)
- Class hierarchies can be used to express exceptions. BUT, they need RTTI for the following code to work (the virtual function in `SomeError` causes it to have a vtable—and hence RTTI):

```cpp
#include <iostream>

struct SomeError {virtual void print() = 0;};
struct ThisError : public SomeError {
  virtual void print() {
    std::cout << "This Error" << std::endl;
  }
};
struct ThatError : public SomeError {
  virtual void print() {
    std::cout << "That Error" << std::endl;
  }
};
int main() {
  try { throw ThisError(); }
  catch (SomeError& e) { //reference, not value
    e.print();
  }
  return 0;
}
```

Exceptions and local variables [important]

- When an exception is thrown, the stack is unwound
- The destructors of any local variables are called as this process continues
- Therefore it is good C++ design practice to wrap any locks, open file handles, heap memory etc., inside stack-allocated object(s), with constructors doing allocation and destructors doing deallocation. This design pattern is analogous to Java's try-finally, and is often referred to as "RAII: Resource Acquisition is Initialisation".

Templates

- Templates support metaprogramming, where code can be evaluated at compile time rather than run time
- Templates support generic programming by allowing types to be parameters in a program
- Generic programming means we can write one set of algorithms and one set of data structures to work with objects of any type
- We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. `sort` routine presented earlier), but at the cost of losing static checking.
- The C++ Standard Library makes extensive use of templates
- C++ templates are similar to, but richer than, Java generics.

Templates – big-picture view (TL;DR)

- Templates are like Java generics, but can have both type and value parameters:
  `template <typename T, int max>class Buffer { T[max] v; int n;};`
- You can also specify 'template specialisations', special cases for certain types (think compile-time pattern matching).
- This gives lots of power (Turing-powerful) at compile time: 'metaprogramming'.
- Top-level functions can also be templated, with ML-style inference allowing template parameters to be omitted, given

```
1 template<typename T> void sort(T a[], const unsigned& len);
2 int a[] = {2,1,3};
```

  then `sort(a,3)` ≡ `sort<int>(a,3)`
- The rest of the slides explore the details.

An example: a generic stack [revision]

- The stack data structure is a useful data abstraction concept for objects of many different types
- In one program, we might like to store a stack of `int`s
- In another, a stack of `NetworkHeader` objects
- Templates allow us to write a single generic stack implementation for an unspecified type `T`
- What functionality would we like a stack to have?
  - `bool isEmpty();`
  - `void push(T item);`
  - `T pop();`
  - ...
- Many of these operations depend on the type `T`

[Just like Java so far.]

Template for Stack

- A class template is defined in the following manner:

  ```
  template<typename T> class Stack { ...  }
  ```

  or equivalently (using historical pre-ISO syntax)

  ```
  template<class T> class Stack { ...  }
  ```

- Instantiating such a `Stack` is syntactically like Java, so (e.g.) we can declare a variable by `Stack<int> intstack;`.
- Note that template parameter `T` can in principle be instantiated to any C++ type (here `int`). Java programmers: note Java forbids `List<int>` (generics cannot use primitive types); this is a good reason to prefer syntax `template <typename T>` over `template <class T>`.
- We can then use the object as normal: `intstack.push(3);`
- So, how do we implement `Stack`?
  - Write `T` whenever you would normally use a concrete type

```
// example16.hh

template<typename T> class Stack {

  struct Item { //class with all public members
    T val;
    Item* next;
    Item(T v) : val(v), next(0) {}
  };
  Item* head;
  // forbid users being able to copy stacks:
  Stack(const Stack& s) {}             //private
  Stack& operator=(const Stack& s) {} //private
public:
  Stack() : head(0) {}
  ~Stack();       // should generally be virtual
  T pop();
  void push(T val);
  void append(T val);
};
```

```
1 // sample implementation and use of template Stack:
2
3 #include "example16.hh"
4
5 template<typename T> void Stack<T>::append(T val) {
6   Item **pp = &head;
7   while(*pp) {pp = &((*pp)->next);}
8   *pp = new Item(val);
9 }
10
11 //Complete these as an exercise
12 template<typename T> void Stack<T>::push(T) {/* ... */}
13 template<typename T> T Stack<T>::pop() {/* ... */}
14 template<typename T> Stack<T>::~Stack() {/* ... */}
15
16 int main() {
17   Stack<char> s;
18   s.push('a'), s.append('b'), s.pop();
19 }
```

Template richer details

- A template parameter can take an integer value instead of a type: `template<int i> class Buf { int b[i]; ... };`
- A template can take several parameters: `template<typename T,int i> class Buf { T b[i]; ... };`
- A template parameter can be used to declare a subsequent parameter: `template<typename T, T val> class A { ... };`
- Template parameters may be given default values

```
1 template <typename T,int i=128> struct Buffer{
2   T buf[i];
3 };
4
5 int main() {
6   Buffer<int> B; //i=128
7   Buffer<int,256> C;
8 }
```

Templates behave like macros

- A templated class is not type checked until the template is instantiated:

```
template<typename T> class B {const static T a=3;};
```

  - B<int> b; is fine, but what about B<B<int> > bi;?

  Historically, template expansion behaved like macro expansion and could give rise to mysterious diagnostics for small errors; C++20 adds syntax for concept to help address this.
- Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Java programmers: in Java generics are implemented by “type erasure”. Every generic type parameter is replaced by Object so a generic class compiles to a single class definition. Each call to a generic method has casts to/from Object inserted—these can never fail at run-time.

Template specialisation

- The `typename T` template parameter will accept any type `T`
- We can define a specialisation for a particular type as well (effectively type comparison by pattern-matching at compile time)

```
1  #include <iostream>
2  class A {};
3  
4  template<typename T> struct B {
5    void print() { std::cout << "General" << std::endl;}
6  };
7  template<> struct B<A> {
8    void print() { std::cout << "Special" << std::endl;}
9  };
10 int  main() {
11   B<A> b1;
12   B<int> b2;
13   b1.print(); //Special
14   b2.print(); //General
15 }
```

Templated functions

- A top-level function definition can also be specified as a template; for example (think ML):

```
1 template<typename T> void sort(T a[],
2                                 const unsigned int& len);
```

- The type of the template is inferred from the argument types:
  `int a[] = {2,1,3}; sort(a,3);` $\Longrightarrow$ `T` is an `int`
- The type can also be expressed explicitly:
  `sort<int>(a,3)`
- There is no such type inference for templated classes
- Using templates in this way enables:
  - better type checking than using `void *`
  - potentially faster code (no function pointers in vtables)
  - larger binaries if `sort()` is used with data of many different types

```
1 #include <iostream>
2
3 template<typename T> void sort(T a[], const unsigned int& len) {
4   T tmp;
5   for(unsigned int i=0;i<len-1;i++)
6     for(unsigned int j=0;j<len-1-i;j++)
7       if (a[j] > a[j+1]) //type T must support "operator>"
8        tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
9 }
10
11 int main() {
12   const unsigned int len = 5;
13   int a[len] = {1,4,3,2,5};
14   float f[len] = {3.14,2.72,2.54,1.62,1.41};
15
16   sort(a,len), sort(f,len);
17   for(unsigned int i=0; i<len; i++)
18     std::cout << a[i] << "\t" << f[i] << std::endl;
19 }
```

Overloading templated functions

- Templated functions can be overloaded with templated and non-templated functions
- Resolving an overloaded function call uses the "most specialised" function call
- If this is ambiguous, then an error is given, and the programmer must fix by:
  - being explicit with template parameters (e.g. `sort<int>(...)`)
  - re-writing definitions of overloaded functions

Template specialisation enables metaprogramming

Template metaprogramming means separating compile-time and run-time evaluation (we use `enum` to ensure compile-time evaluation of `fact<7>`).

```
1 #include <iostream>
2
3 template<unsigned int n> struct fact {
4         enum { value = n * fact<n-1>::value };
5 };
6
7 template <> struct fact<0> {
8         enum { value = 1 };
9 };
10
11 int main() {
12   std::cout << "fact<7>::value = "
13             << (unsigned int)fact<7>::value << std::endl;
14 }
```

Templates are a Turing-complete compile-time programming language!

Exercises

1. Provide an implementation for:
   `template<typename T> T Stack<T>::pop();` and
   `template<typename T> Stack<T>::~Stack();`
2. Provide an implementation for:
   `Stack(const Stack& s);` and
   `Stack& operator=(const Stack& s);`
3. Using metaprogramming, write a templated class `prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.
4. How can you be sure that your implementation of class `prime` has been evaluated at compile time?

Miscellaneous things [non-examinable]

- C++ annotations `[[thing]]` – like Java `@thing`
- C++ lambdas: like Java, but lambda is spelt ‘`[]`’. E.g.

```
1  auto addone = [](int x){ return x+1; }
2  std::cout << addone(5);
```

  Lambdas have class type (like Java), and the combination of `auto` and overloading the ‘`operator()`’ makes everything just work. Placing variables between the ‘`[]`’ enables access to free variables: default by rvalue, prefix with ‘`&`’ for lvalue, e.g. ‘`[i,&j]`’
- C++20 lets programmers define operator ‘`<=>`’ (3-way compare) on a class, and get 6 binary comparisons (‘`==`, ‘`<`‘, ‘`<=`‘ etc.) for free.
- use keyword `constexpr` to require an expression to be compile-time evaluable—helps with template metaprogramming.
- use `nullptr` for new C++ code—instead of `NULL` or `0`, which still largely work.