2019

Programming in C and C++

CAMBRIDGE COMPUTER SCIENCE TRIPOS PART IB, PAPER 4

ASHWIN AHUJA

Table of Contents

C 3
Types 3
Variables 3
Operators 4
Type Conversion 4
Arrays 4
Strings 4
Expressions and Statements 4
Control Flow 5
Functions 5
Recursion 5
Compilation 5
Code in Multiple Files 6
Static 6
Address Space Layout 6
Pre-processor 7
Pointers and Pointer Manipulation 8
Arrays 8
Pointer Arithmetic 9
Collections 9
Miscellaneous 9
Const and Volatile 9
Typedef 10
Inline 10
Operator Precedence 10
Library Support 11
I/O 11
Dynamic Memory Allocation 11
Tooling 11
Address Sanitizer – ASan 11
Memory Sanitizer – MSan 12
Undefined Behaviour Sanitizer – UBSan 12
Valgrind 12
Graphs 13
Tree 13
Arenas 14
Reference Counting 15
Garbage Collection 17
Data Structures 17
Procedures 17
Design Considerations 19
Memory Hierarchy and Cache Optimization 19
Caches 19
Performance Engineering 20
Intrusive Lists 20
Array of Structs 21
Loop Blocking 22
Debugging 22
Bugs 22
Finding Defects 23
Debugging Tools 23
Undefined Behaviour 24
Implementation-defined behaviour 24
Unspecified behaviour 24

Undefined behaviour 24
Optimisation Trade-off 25
C Abstract Machine 25
C++ 26
Differences 26
Features 26
Linking C and C++ code 27
Objects 27
Differences from Java 28
Constructor 28
Destructor 28
Array of Class Object 28
Operators 28
This 29
Temporary Objects 29
Friends 29
Inheritance 29
Abstract Classes 30
Casts 30
Streams 30
Virtual Functions 30
Multiple Inheritance 31
Exceptions 31
Templates and Meta-Programming 32

C

Types

C is unsafe – erroneous uses of C features are not checked (either statically or at runtime), so errors can lead to memory corruption and arbitrary code execution.

Primitive types are: (1) characters, (2) numbers and (3) addresses – these are the types which are worked on by operators

- There are no primitives on composite types – strings, arrays, etc
- Size of types is architecture-dependent
- C99 adds fixed-size types: int16_t

type	description
char	Characters - ≥ 8 bits
Int	Integers - ≥ 16 bits – usually 1 word
Float	Single-precision floating point number
double	Double-precision floating point number

Type operators: unsigned, short, long, const, volatile

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 with suffix ‘l’ or ‘L’	1234L
Float	Num with ‘.’, ‘e’, or ‘E’ and suffix ‘f’ or ‘F’	1.234e3F
Double	Num with ‘.’, ‘e’ or ‘E’	1.234e3
Long double	Num with ‘.’, ‘e’, or ‘E’ and suffix ‘l’ or ‘L’	1.23E3L

- Numbers can be expressed in octal with ‘0’ prefix and hexadecimal with ‘0x’ prefix

Libraries: only static definition and stack-based locals built in

- Heap, I/O and threading implemented as libraries

Constants:

- enum boolean {TRUE, FALSE} – default to allocating successive integers from 0 – also possible to assign values to constants. Values do not need to be distinct, but names must be distinct
- const …

Variables

Must be **declared** before use – set the types of everything. Can be declared without defining it using ‘extern’ keyword – tells the compiler that storage has been allocated elsewhere.
Link Error: if variable declared and used in a program, but not defined
Defined: storage allocation - variables must be defined exactly once. A definition counts as a declaration.
Variable Names: composed of letters, digits and underscores – names must start with letter or underscore

Variables defined by prefixing name with a type and can be initialised
Can define multiple variables of the same basic type can be declared or defined together.

Operators

All operators (including assignment) return a result

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

Type Conversion

Automatic: when two operands to a binary operator are of different types

- Conversion widens a value: short -> int
- But narrowing is possible and may not generate a warning

Forced: can be forced via a cast – (type) exp

Arrays

Contiguous block of memory for the relevant number of values – indexed from zero and there is no bounds checking: *long int i[10]*

Can also have multi-dimensional arrays – 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

Strings

Represented in C are represented as an array of char terminated with '\0' – supported with double-quotes: *char s[]="two strings mer" "ged and terminated"*

- Has implicit concatenation of string literals

Functions are in the string.h library

Expressions and Statements

Expression: when one or more operators are combined – has a type and result – it becomes a **statement** when followed by a semicolon – several expressions can be separated by a comma, and expressions then evaluated left-to-right: **type and value is type and value of the right-most expression**

Block (Compound) Statement: formed when multiple statements surrounded with braces – a block is equivalent to a single statement.

- Originally in C90, variables only declared at start of the block, but restriction lifted in C99

Control Flow

```
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);
```

Goto: never required and often results in difficult-to-understand code – only exception is for exception handling

Functions

Function Definition: has a return type, parameter specification, and a body or statement
Function Declaration: has a return type and a parameter specification followed by a semicolon.

Properties:

- Can be declared or defined extern or static
- Pass-by-value
- Function must have exactly one definition and can have multiple declarations
- Cannot be nested
- Function declaration with no values (*int power()*) means that arguments should not be type-checked (may have any number of arguments)
    - Function with no arguments has a *void* instead of the arguments
    - Ellipsis can be used to define functions with variable length arguments

Recursion

Functions call themselves recursively – on each call, a new set of local variables created. Therefore, recursion of depth n has n sets of variables.

Compilation

Compiler transforms C source code or execution unit into **object file:** consists of

- Machine code
- 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

Linker: combines several object files into an executable by:
1. Combining all object code into a single file
2. Adjusting the absolute addresses from each object file
3. Resolving all undefined symbols

Symbol Table: 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
- Symbol table records an association from 0x100000f72 and the printf earlier
- -g embeds additional debugging information into symbol tables
    - Not included by default – non-essential
- Also maps source code to the program counter register, keeping track of the control flow

Code in Multiple Files

C separates declaration from definition from both variables and functions – allows portions of code to be split across multiple files.
- Code in different files can then by compiled at different times – allows libraries to be compiled once but used many times.
- Allows companies to sell binary-only libraries

But need to have **header file** to have the declarations (function and variables):
- Also offers pre-processor macros
- Avoid duplication and errors that would otherwise occur

Static

Static keyword limits the scope of a variable or function.
- In the global scope, static doesn't export the function or variable symbol – this prevents the variable or function from being called externally
- In local scope, static variable retains its value between function calls
    - Single static variable exists even if a function call is recursive

Address Space Layout

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

Pre-processor

Executes before any compilation takes place – manipulates the text of the source file in a single pass.

1. Deletes every occurrence of a backslash followed by newline
2. Replaces comments by a single space
3. Replaces definitions, obeys conditional preprocessing directives and expands macros
4. Replaces escaped sequences in character constants and string literals and concatenates adjacent string literals

Programming Pre-processor

Pre-processor can be used by programmer to rewrite the source code – this is very powerful but very hard to debug:

- Interprets lines starting with # with a special meaning
- **Text Substitution Directives:**
    - #include
        - include simply replaces the line in the source file with the contents of the file is it linked to (*#include "filename" OR #include <filename>*)
        - " searches for a file in the same location as the source file, then searches a predefined set of directories
        - < searches a predefined set of directories
    - #define *name replacement-text*
        - Provides a direct text substitution of all future examples of name with the replacement-text for the remainder of the source file
        - Replacement does not take place if name is found in quoted string – by convention tends to be in upper case to distinguish it from a normal variable name
        - **Macros**
            - In body of the macro, prefix parameter in replacement text with '#' places the parameter value inside string quotes ' " '
                - Placing '##' between two parameters in the replacement text removes whitespace between variables in generated output
            - Always use do() while…
- **Conditional Directives:** #if, #elif, #else and #endif
    - 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 if it evaluates to a non-zero value
    - Preprocessor built-in defined takes a name as its argument and gives 1L if it is #define-d; 0L otherwise
    - #undef can be used to remove pre-processor macro
    - #ifdef N = #ifdef defined(N)

Error Control

To help other compilers which generate C code as output, compiler line and filename warnings can be overridden with:
#line constant "filename"

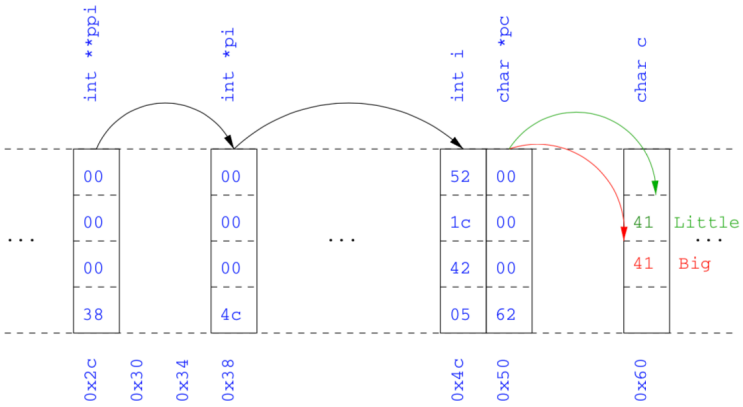
Compiler then adjusts its internal value for the next line in the source file as constant and the current name of the file being processes as "filename" – the statement *#error some-text* causes the pre-processor to write a diagnostic message containing *some-text*

Predefined Identifiers: __LINE__, __FILE__, __DATE__ and __TIME__

Pointers and Pointer Manipulation

Computer memory is often abstracted as a sequence of bytes, grouped into words – size of word determines the size of addressable memory in the machine. Pointer is a variable which contains the memory address of another variable.

*Char *pc:* declared using an asterisk – binds to the variable name, not the type specifier.



* operator: dereferences the pointer – gets the value pointed to by the pointer
& operator: gets the memory address of a variable

Can be passed as a parameter to functions – allow a function to alter parameters passed to it. Also, can have a pointer to functions – allowed to pass functions to be passed as arguments to functions

Void * Pointer: typeless or generic pointer – pointer to any object (not a function). This is really terrible for the type system, therefore should only be used where necessary.

Arrays

Array name represents the memory address of the first element of the array – pointers can be used to index any element of any array

```
char c[10];
char *pc = c;       // This is the same
char *pc = &c[0];   // as this
int i[10];
int *pi = &i[5];
```

Arrays of Pointers

Particularly useful with strings – used for C support of command line arguments

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

argv is an array of character pointers and argc tells the programmer the length of the array

Pointer Arithmetic

Can be used to adjust where a pointer points – if pc points to the first element of an array, after executing pc += 3, then pc points to the fourth element. Can also dereference using array notation.

For an array c, *(c+i) == c[i] and c+i == &c[i]

Collections

Struct: Collection of one or more members (fields) – provides a simple method abstraction and grouping.

- A structure can contain structures itself
- Can also be assigned to, as well as passed to and returned from functions
- Declared with keyword **struct** – declaring structure creates a new type
- **Initialising Structure**

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**
  - Access using . notation – structname.member
  - For pointer to struct can use *(*pc).member* or *pc -> member* interchangeable
- **Self-referential Structures**
  - 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

Unions: Union variable is a single variable which can hold one of a number of different types

- *union u { int I; float f; char c; }*
- Size of a union variable is the size of its largest member – type held can change during program execution.
  - Type retrieved must be the type most recently stored
- Same memory access as for structs

Bit Fields

- Allow low-level access to individual bits of a word – useful when memory limited
- Specified inside a struct by appending a declaration with a colon and a number of bits

Miscellaneous

Const and Volatile

Const: can only be assigned a value when it is defined – can also be used for parameters in a function definition

Volatile: used to state that a variable may be changed by hardware or the kernel – may prevent unsafe compiler optimisations for memory-mapped input / output

Pointers and Const

```
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
```

Typedef

Typedef operator creates a synonym for a data type – once it has been created, it can be used in the place of the usual type name in declarations and casts. It is particularly useful with structures and unions.

Inline

Function can be declared as inline – compiler with then try to inline the function (might not necessarily be able to). An inline function must be defined in the same execution unit as it is used. Both inline and register are largely unnecessary with modern compilers and hardware.

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of[note 1] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 2]	?:	Ternary conditional[note 3]	Right-to-Left
14	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

1. ↑ The operand of `sizeof` can't be a type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
2. ↑ Fictional precedence level, see Notes below
3. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to `?:` is ignored.

When parsing an expression, an operator which is listed on some row will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it. For example, the expression `*p++` is parsed as `*(p++)`, and not as `(*p)++`.

Operators that are in the same cell (there may be several rows of operators listed in a cell) are evaluated with the same precedence, in the given direction. For example, the expression `a=b=c` is parsed as `a=(b=c)`, and not as `(a=b)=c` because of right-to-left associativity.

Library Support

I/O

I/O is not managed directly by the compiler – supported 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);
```

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)
```

Any successfully allocated memory must be deallocated manually – failure to deallocate will result in a memory leak. Also, each allocated pointer must be deallocated exactly once along each execution path through the program.

Tooling

Address Sanitizer – ASan

AddressSanitizer checks for memory corruption:

- Out-of-bounds array accesses
- Use pointer after call to free()
- Use stack variable after out of scope
- Double-frees or other invalid frees
- Memory leaks

It adds (1) runtime overhead (2x), (2) needs a recompilation and (3) doesn't catch all memory errors – however it is useful to use while developing (it is built into gcc and clang)
Compile with: *-fsanitize=address*

Memory Sanitizer – MSan

Both local variable declarations and dynamic memory allocation via malloc() do not initialize memory. Accesses to unitialized variables are undefined – doesn't mean that you get some unspecified value, means compiler is free to do anything it likes.
Compile with: *-fsanitize=memory*

Issues

(1) Very expensive (2-3x slowdowns)
(2) Checks for memory initialisation errors
(3) Only available on clang (not gcc)

Undefined Behaviour Sanitizer – UBSan

Built into gcc and clang – can often even be used in production
Deals with:

- Signed integer overflow
- Dereferencing null pointers
- Pointer arithmetic overflow
- Dynamic array whose size is non-positive

Issues:

(1) Needs to recompile
(2) Adds runtime overhead – 20%
(3) Does not catch all undefined behaviour

Valgrind

UBSan, MSan and ASan require recompiling, UBSan and ASan don't catch accesses to unitialised memory. Valgrind is a combination tool with instrument binaries to detect numerous errors

Issues:

(1) Adds substantial runtime overhead
(2) Not built into GCC / Clang
(3) Does not catch all undefined behaviour

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

Graphs

Tree

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

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

This doesn't really work when you have a graph with repeated nodes – the node gets deallocated multiple times.

Corrected

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

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

typedef struct TreeListCell TreeList;
struct TreeListCell {
```

```
    Tree *head;
    TreeList *tail;
}

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

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

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

Arenas

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

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

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) Initialise current element
        (2) Increment current
        (3) Return the initialised node
*/

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

/*
    Free a whole arena at a time – all tree nodes are freed at once
*/
```

Reference Counting

Idea is to keep track of the number of pointers to an object – only free an object when the count reaches zero.

Algorithm:

(1) Start with k references to n2
(2) Eventually k becomes 0
(3) Decrement reference count of each thing n2 points to
(4) Then delete n2
(5) Recursively delete n

For tree structure:

```
struct node {
    unsigned int rc; // ADDED
    int value;
    struct node *left;
    struct node *right;
};
typedef struct node Node;
const Node *empty = NULL;
Node *node(int value, Node *left, Node *right);
void inc_ref(Node *node);
void dec_ref(Node *node);
```

```
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;
}

void inc_ref (Node *node) {
    if (node != NULL) {
        node->rc += 1;
    }
}

void dec_ref (Node *node) {
    if (node != NULL) {
        if (node->rc > 1) {
            node->rc -= 1;
        }
        else
        {
            dec_ref(node->left);
            dec_ref(node->right);
            free(node);
        }
    }
}

/*
The get_left() function returns the left subtree, but also increments the reference
count
*/
Node *get_left (Node *node) {
    inc_ref(node->left);
    return (node->left);
}

/*
The set_left() function updates the left subtree, incrementing the reference count
to the new value and decrementing the reference
*/
void set_left(Node *node, Node *newval) {
    inc_ref(newval);
    dec_ref(node->left);
    node->left = newval;
}
```

Sharing References vs Transferring References: who is responsible for managing reference counts?

Also, this clearly doesn't work if there are any cycles of references – garbage collection works with this.

Garbage Collection

Data Structures

```
// Node are node objects but augmented with a mark bit and a next link connecting
all allocated nodes
struct node {
    int value;
    struct node *left;
    struct node *right;
    bool mark;
    struct node *next;
};
typedef struct node Node;

// root is a node we don't want to garbage collection – in a linked list
struct root {
    Node *start;
    struct root *next;
};
typedef struct root Root;

// alloc holds the head of the lists of nodes and roots
struct alloc {
    Node *nodes;
    Root *roots;
};
typedef struct alloc Alloc;
```

Procedures

Mark-and-Sweep:

- **Mark**
  - From each root, mark the nodes reachable from that root – set mark = true
  - So, every reachable will have a true mark bit, and every unreachable one will be set to false
- **Sweep**
  - Iterate over every allocated node
  - If node is unmarked, free it and if node is market, reset the mark bit to false

```
// creates a fresh allocator
// Invariant: no root or node is part of two allocators
Alloc *make_allocator(void) {
    Alloc *a = malloc(sizeof(Alloc));
    a->roots = NULL;
    a->nodes = NULL;
```

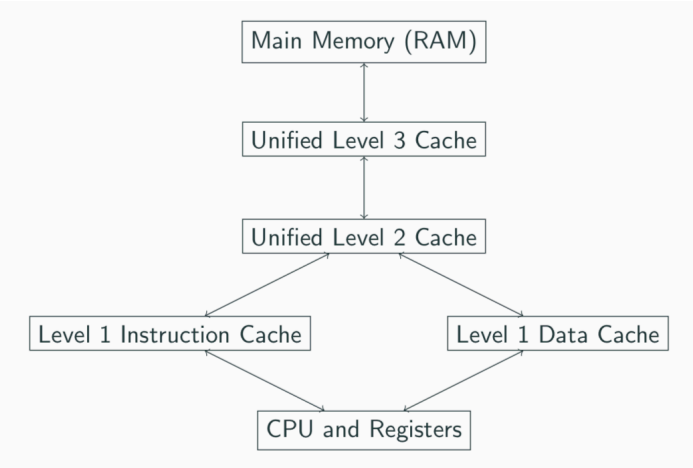
```
    return a;
}
// node(n, l, r, a) creates a fresh node in allocator a
Node *node(int value, Node *left, Node *right, Alloc *a)
{
    Node *r = malloc(sizeof(Node));
    r->value = value;
    r->left = left;
    r->right = right;
    r->mark = false;
    r->next = a->nodes;
    a->nodes = r;
    return r;
}

// root(n) creates a new root object rooting the node n
Root *root(Node *node, Alloc *a){
    Root *g = malloc(sizeof(root));
    g->start = node;
    g->next = a->roots;
    a->roots = g;
    return g;
}

// gc(a) frees all nodes unreachable from the roots using mark and sweep
void mark_node(Node *node){
    if(node != NULL && !node->mark){
        node->mark = true;
        mark_node(node->left);
        mark_node(node->right);
    }
}

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

void sweep(Alloc *a) {
    Node *n = a->nodes;
    Node *live = NULL;
    while(n != NULL) {
        Node *tl = n->next;
        if(!(n->mark)) {
            free(n);
        }
        else
        {
```

```
            n->mark = false;
            n->next = live;
            live = n;
        }
        n = tl;
    }
    a->nodes = live;
}

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

Design Considerations

This kind of custom GC is quite slow relative to Java GCs but is very simple to implement. Don't need to worry about cycles or managing reference counts. Worth considering Boehm gc – drop-in replacement to malloc.

Memory Hierarchy and Cache Optimization



Latencies in Memory Hierarchy

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

Caches

Process

- When CPU looks up an address
- If present – cache hit
- If not – cache miss
    - Address is then looked up in main memory
    - Address / value pair is then stored in the cache – and relevant cache line is brought in

Why caches?: Principle of Locality – temporal and spatial

Performance Engineering

Redesigning data structures to take advantage of locality.

Example: Lists - original structure uses either void pointers or has a lot of unnecessary space usage

```
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;
}
```

```
struct data {
    int i;
    double d;
    char c;
};
typedef struct data Data;
struct List {
    Data *head;
    struct List *tail;
};
```

Intrusive Lists

```
/*
    Loses indirection in the head is removed
    But we have to use a specialized representation – can no longer use generic
linked list routines
*/
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;
}
```

Array of Structs

- Following tail pointer can lead to cache miss
- Cons cells requiring storing a tail pointer
- => reduction in the number of data elements that can fit in a cache line
- => reduces data density and increases cache miss rate
- Therefore, we represent ilist_t with Data[] instead – but have to know size up-front

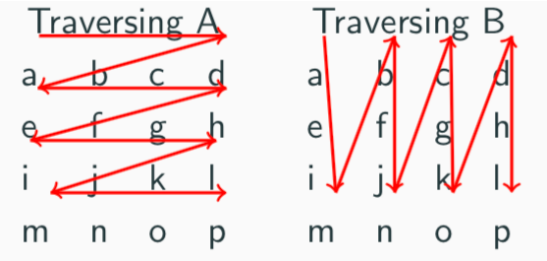
```
Data *iota_array(int n) {
    Data *a = malloc(n * sizeof(Data));
    for (int i = 0; i < n; i++) {
        a[i].i = i;
        a[i].d =  1.0;
        a[i].c = 'x';
    }
    return a;
}

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

// only modifying character field c – so have to hop over integer and double fields
therefore not maximising the number of characters in each cache line
void traverse(int n, Data *a) {
    for (int i = 0; i < n; i++)
        a[i].c += 'y';
}
```

Instead of storing an array of structures but instead of that, we store a struct of arrays – makes traversing far more efficient

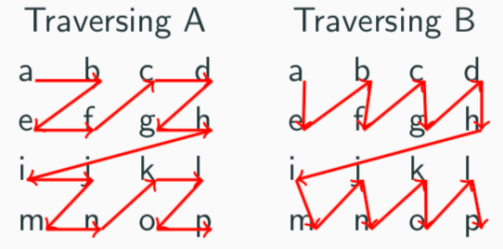
```
typedef struct datavec *DataVec;
struct datavec {
    int *is;
    double *ds;
    char *cs;
};

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

Loop Blocking

Traversing two arrays at the same time: items next to each other horizontally are next to each other in memory



Easy to see that A has a favourable traversal and B is 'jumpy' therefore this could be improved.



This reduces the total number of cache misses and therefore, speeds the system up – for full example, see lecture slides (Lecture 8, slide 16)

Debugging

Debugging is a methodical process of finding and reducing the number of bugs (**Fault Isolation:** find the cause of failure) in a computer program, thus making it behave as originally expected.

Errors:

- Compile-time: these occur due to misuse of language constructs, such as syntax errors. Fairly easy to find using compiler tools and warnings to fix reported problems.
- Run-time: Much harder to figure out, as they cause the program to generate incorrect output during execution

Bugs

Program contains a defect in the source code, either due to (1) design misunderstanding or (2) implementation mistake. Could lead to:

- Crash with memory error or seg fault
- Return incorrect result
- Unintended side-effects

Often hard to debug, because defects do not materialise predictably:

- May require specific inputs
- Bug may be specific to OS or architecture
- May be due to long run time

Lifestyle

(1) Program fails unit test or bug is reported by a user

   a. **Unit Tests:** short code fragments written to test code modules in isolation, generally written by the original develop
(2) Given failing input conditions, a programmer debugs the program until the offending source code is located
(3) Program is recompiled with source code fix and regression test is run to confirm that the behaviour is fixed.
   a. **Regression testing:** ensures that changes do not uncover new bugs, running unit, integration and full system testing.

Finding Defects

Import to note that defects are not necessarily located in the source code near a particular runtime failure – the greater the distance, the more difficult it is to debug.

Printing Values

Print values as the program executes:

- Put in as much debugging information as you can
- Make each entry as unique as possible
- Flush the debug output so it reliably appears in the terminal

C pre-processor:

- Define a DEBUG parameter to compile 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

Assertions

Defects can be found more quickly than printing values by using assertions to encode invariants through the source code – offers a more useful and specific error message.

Assertions can be disabled by defining the NDEBUG pre-processor flag – important to note that one should never cause side-effects in assertions as they may not be active

Reproducing the Bug

Try and reproduce the bug and can therefore attempt to trace the failure.

Debugging Tools

Purposes:

(1) Observe value of a variable during program execution
(2) Stop execution of program if assertion is violated
(3) Get trace of function calls leading up to failure

Types of Debuggers:

- **Interpretive Debugger:** simulate program execution one statement at a time
- **Direct Execution Debuggers:** use hardware and operating system features to inspect the program memory and set breakpoints to pause execution

LLDB from LLVM: run using *cc -Wall -o -lookup -DNDEBUG -g -debug-s18.c* and run the binary using *lldb ./lookup*

- Can print state of a variable once stopped
- Can set a breakpoint: stop at a specific line
- Can set a watchpoint: inspect when variables change state

xUnit: Unit testing framework for C – helps you structure your tests

Undefined Behaviour

Implementation-defined behaviour

Compiler must choose and document a consistent behaviour

- Set depending on the target hardware architecture and operating systems Application Binary Interface (ABI)
- Example of implementation-defined behaviour is:
    - (1) Number of bits in a byte
    - (2) sizeof(int)
    - (3) Results of some bitwise operations on signed integers
    - (4) Result of converting a pointer to an integer or vice versa

Unspecified behaviour

Means that from a given set of possibilities, the compiler can vary them within the same program to maximise effectiveness of its optimisations, examples include:

(1) Evaluation order of arguments in a function call
(2) Order in which side effects take place when not otherwise explicitly specified by standard
(3) Whether call is inline
(4) Memory layout of storage for function arguments
(5) Order and contiguity of storage allocated by successive calls to malloc, calloc or realloc

Undefined behaviour

Arbitrary behaviour from the compiler – needs to be avoided, examples include:

(1) Object modified more than once between two sequence points
(2) Conversion of pointer to an integer type produces a value outside of representation range
(3) Value of a pointer to an object whose lifetime is ended
(4) Use of uninitialized variable before accessing it
(5) Accessing out-of-bounds memory
(6) Dereferencing NULL pointer or wild pointer
(7) Signed integer arithmetic if results overflows
(8) Security issues – due to overflow

Compiler only has to execute statements where the behaviour is defined and can optimise the rest away – compiler categories functions into three kinds:

(1) Always Defined: no restrictions on inputs and defined for all possible inputs
    a. Inputs carefully checked to ensure that undefined operations invoke an error function

(2) Sometimes Defined: some restrictions on inputs, so can be either defined or undefined depending on the input value
   a. If compiler statically detects an input that would be undefined, it can skip the function call entirely

(3) Always Undefined: no valid inputs and are always undefined

Living with Undefined Behaviour: can use unsafe programming languages to build safer abstractions – i.e. use C and C++ to make bindings. In the short term, can use tools and techniques

- Compilers
- Static analysis tools (clang-analyser and Coverity) or dynamic analysis engines (Valgrind)
- Clang and GCC have undefined behaviour sanitizers that detect and generate errors for many classes of undefined behaviour via *-fsanitize=undefined*
- Always check all inputs – avoid sometimes-undefined
- Use high-quality third-party libraries that obey these rules

Optimisation Trade-off

Tension between:

- Programmers' ability to predict performance of code
- Compilers generating fast machine code for specific hardware
- Language's portability on present and future architectures
- Revisions to C must remain backwards compatible with existing code
- **Balance is achieved via ongoing language specification process:**
  - Ritchie and Kerrigan – 1978
  - ANSI C (C89) – 1989
  - C99: adding floating point support
  - C11: detailed memory model

C Abstract Machine

Definitions:

- (1) Semantic descriptions of language features describe the behaviour of an abstract machine in which issues of optimization are irrelevant
- (2) Defines side effects as accessing a volatile object, modifying an object or file, or calling a function that does any of those operations
  - Side effects change the state of the execution environment and are produced by expression evaluation
- (3) **Sequence Points:** at which point all side effects which have been seen so far are guaranteed to be complete

Sequence Points

If an asynchronous signal is received, only the values of objects from the previous sequence point can be relied on:

(1) Between left and right operands of '&&' and '||' operators
(2) Between evaluation of the first operand on the ternary (question mark) operator and the second and third operands
(3) At end of full expression, for example

   a. Return statements
   b. Control flow from if, switch, while or do/while
   c. All three expressions in a for statement
(4) Before function call is entered

Execution

In abstract machine, all expressions are evaluated as specified by language semantics in the standard. Implementation need not evaluate part of an expression if it can deduce that:

(1) Value not used
(2) No needed side effects produced
(3) Includes any caused by function calls or accessing a volatile object

C++

Aims: General-purpose programming language with a bias towards systems programming that:

- Is a better C
- Supports data abstraction
- Support object-oriented programming
- Support generic programming

Differences

- C and C++ both have very good run-time performance
- C++ has more facilities: 'C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off'
- C++ is a superset of C but have to be careful about it:
  - (1) don't want conflicting IO libraries
  - (2) often program using different metaphors in C and C++
  - (3) C functions don't expect an exception to bypass their tidy-up code

Features

(1) bool – new type
(2) class – new type constructor (generalises struct in C)
(3) & - new type constructor – **references**
   a. Provides an alias for a variable
   b. Generally used for specifying parameters to functions and return values as well as overloaded operators
   c. Reference must be initialised when it is declared – connection between a reference and what it refers to cannot be changed after initialisation
   d. Effectively pointer types with implicit * at every use
   e. **In Function Arguments**
      i. When used as a function parameter, referenced value is not copied
      ii. Allows us to change the value of actual object
(4) Enum types are distinct – not a synonym for integers
   a. **Names for enum, class, struct and union can be used directly for types** (C needed an additional typedef)

b. Enumerations define a new type – you use its name of an enumeration to define storage for an instance of an enumeration
c. Implicit type conversion is not allowed
d. Maximum valid value of an enumeration is the enumeration's largest value rounded up to the nearest largest binary power minus one
e. Minimum valid value of an enumeration with no negative values is zero
f. Minimum valid value of an enumeration with negative values is the nearest least negative binary power

(5) **Overloaded Functions**

a. Can define two functions with the same name, but varying in argument types
b. Function can also have default arguments
    i. But non-default argument cannot come after a default
    ii. Declaration does not need to name the variable

(6) **Namespaces**

a. Related data grouped in a namespace
b. Can use :: and using to access components

```
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() { ... }
}
```

c. Namespace is a scope – expressed logical program structure. It provides a way of collecting together related pieces of code.
d. A namespace without a name limits the scope of variables, functions and classes to the local execution unit
e. Same namespace can be declared in several source files
f. Global function main() cannot be inside a namespace

Linking C and C++ code

Extern "C' specifies that following declaration or definition should be linked as C, not C++ code – multiple declarations and definitions can be grouped in curly brackets. Care has to be taken with pointers to functions and linkage.

Objects

Somewhat like Java: contains both data and member functions – can extend other classes

- Members can be static
- Have access control: private, protected and public
- Classes created with class or struct keywords
    - Struct default to public, class to private

Differences from Java

- Values of class types not references to objects but objects themselves => access members with '.'
- Create object of class C on:
  - (1) stack – C x;
  - (2) heap – new C()
- Member functions statically resolved – otherwise can be declared as virtual.
- Functions can be declared inside a class, but defined outside it using '::'
- C++ uses *new* to allocate and *delete* to de-allocate – no in-built garbage collector

Constructor

Member function with same name as a class – defines what to do on creating new object

- Default constructor is a function with no arguments – constructor generated if none specified
- Can have multiple constructors

Copy Constructors: Can define your own copy constructors

- *class_name::class_name(const class_name&) { … }*
- Then, should overwrite the assignment operator explicitly
- *class_name& class_name::operator=(const class_name& c) { … }*

Destructor

Member function with the same name as the class, prefixed with a tilde (~) – only one destructor

- This is called when a stack-allocated object goes out of scope or when a heap-allocated object is deallocated with *delete* – also occurs for stack-allocated objects deallocated during exception handling
- Make destructors virtual if class has subtypes or supertypes

Constant Member Functions: prevents object members being modified by the function – this is helpful to both the programmer (for maintenance) and compiler (for efficiency)

Array of Class Object

Can only be defined if the class has a default constructor – since C++ doesn't distinguish between a pointer to a single object and a pointer to the first element of an array of object, array deletion needs new syntax: *delete[] c;* - when array is deleted, object destructor is invoked on each element

Operators

Almost all operators can be overloaded, with the following syntax:

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

```
// if you're inside the body of a class
bool Complex::operator==(Complex b) {
    return re==b.real() && im==b.imag();
}
```

This

'this' is a keyword which returns a reference to the current object – this is an implicit argument to a method when seen as a function.

Class instances as member variables: class can have an instance of another class as a member variable:

```
class X {
    Complex c;
    Complex d;
    X(double a, double b): c(a,b), d(b) {
        ...
    }
}
```

Temporary Objects

Often created during execution – temporary which is not bound to a reference or named object exists only during evaluation of a full expression.

Example: 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(); //Okay
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(); //Okay
```

Friends

In class: *Friend class class_name_of_friend;* - then friend is allowed to access the private and protected members of the class in which the statement occurs

In function: function can be declared friend to allow it to access the private and protected members of the enclosing class

Inheritance

Allows a class to inherit features of another – anything that was public or protected.

Derived Member Function Call:

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

Abstract Classes

Exactly the same as for Java, except for syntax – cannot be instantiated. Derived class can provide an implementation for some (or all) of the abstract functions. If derived class has no abstract function, it can be instantiated

Casts

(1) **Classical C-Style Casts**
   a. Issues when there is multiple inheritance or virtual bases
   b. Compiler must be able to see the inheritance tree otherwise might not compile the right operation

(2) **New C++ constructor syntax:** int('a') or C(expr)

(3) **New C++ more descriptive forms:** Close to Java object-reference casts and generates code to do run-time tests of compatibility
   a. Dynamic_cast<T>(e)
   b. Static_cast<T>(e)
   c. Reinterpret_cast<T>(e)
   d. Const_cast<T>(e)

(4) Typeid(e) gives the type of e encoded as an object of type_info defined in standard header <typeinfo>

Streams

Std::cin – take an input
Std::cout – output something to console
Std::cerr – throw an error message

Virtual Functions

Non-virtual member functions are called depending on the static type of the variable, pointer or reference. Since 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:

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

```
    virtual int maxSpeed() { return 60; }
};
```

Selecting the right function has to be a run-time decision – to enable this, compiler generates a **virtual function table (vtable):**

- Contains a pointer to the correct function for every object instance
- Allows for enabling virtual functions, but introduces run-time overhead

Multiple Inheritance

Possible to inherit from multiple base classes – members from both base classes exist in the derived class. If there is a name clash, explicit naming is required.

```
ShapelyVehicle sv;
sv.vehicle::maxSpeed();
```

Means we can inherit from the same class twice in a single class – therefore all references must be stated explicitly.

Virtual Base Classes: alternatively have a single instance of the base class – this is shared amongst all those deriving from it.

Exceptions

Exactly like Java but throw an object value rather than an object reference. If an exception is thrown, the call stack is unwound until a function is found which catches the exception – if this is not caught, the program terminates.

```
struct MyError {
    int errorcode;
    MyError(i):errorcode(i) {}
};

void f() { ... throw MyError(5); ... }
...
    try {
        f();
    }
    catch(MyError x) {
        // handle error (x.errorcode has the value 5)
        throw; // re-throw error
    }
```

When an exception is thrown, the stack is unwound – the destructors of any local variables are called as this process continues. Good C++ design practise to wrap any locks, open file handles, heap memory inside stack-allocated objects with constructors doing allocation and destructors doing deallocation – **Resource Allocation is Initialisation (RAII)**

Templates and Meta-Programming

Templates support **meta-programming:** where code can be evaluated at compile time rather than run time. Also supports **generic programming:** 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
- Achieve some of this flexibility in C, by casting everything to void *

Like Java generics, but can have both type and value parameters:

```
template <class T, int max> class Buffer { T[max] v; int n; };
```

Can also specify template specialisations, special cases for certain types – this gives lots of power at compile time.

```
template<class T> struct B {
    void print() { std::cout << "General" << std::endl;}
};
template<> struct B<A> {
    void print() { std::cout << "Special" << std::endl;}
};
```

Top level functions can also be templated (can have default values as well), with ML-style inference, allowing template parameters to be omitted, given:

```
template<class T> void sort(T a[], const unsigned& len);
// call sort<int>([2, 1, 3], 3)
```

Can also have one template parameter in the definition of a subsequent parameter:

```
template<class T, T val> class A { ... }
```

Templated class is not type checked until the template is instantiated – so therefore can have errors.