Freestanding C

Adam Boult (www.bou.lt)

November 2, 2023

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Unstructured integer programming with C

Chapter 1

Integer literals and variable assignment in C, and naming conventions

1.1 Variable initialisation

1.1.1 Variable initialisation

when doing eg "int myvar;" we are:

+ declaraing: saying the variable exists for the code + defining: allocates memory for it

distinction is important, can split up in eg "extern"

1.2 Variables

1.2.1 Assigning literals to variables

Variables rather than addresses within a scope memory location is equivalent to variable: in high level and variables

We can assign a value to a variable by eg:

```
int a = 10;
```

This does two different things. First it declares the variable a. This assigns part of the memory for the variable. Secondly it defines the value of the memory represented by the variable to 10.

These can be split out as follows.

```
int a;
a = 10;
```

If the variable is declared before it is defined it is an uninitialised variable, and its value is undefined.

In addition to decimal we can also set values using other literals.

```
int a = 10;
int b = 010;
int c = 0x10;
```

Here b is octal because of the first 0 in the literal. In decimal it is 8. c is hexidecimal because of the first $0x$ in the literal. In decimal it is 16.

1.2.2 Assigning variables from other variables

We can also assign variables from other variables.

```
int a = 10;
int b = a;
```

The following is valid syntax but unlikely to be what was intended.

```
int a = 10;
int b;
int c = b = a;
```

1.2.3 Note on lvalues

lvalue in c identifiable location in memory. not a constant. not a function. not a literal, not a calculation eg (a+b)

left has to be lvalue. right can be lvalue or not int a = 1; // OK. a is an lvalue. doesn't matter what right is. int b = a; // OK. b is an lvalue. right is also an lvalue, which is ok. (a+b) = 5; // not OK 5 = a; // Not OK left hand side has to be an lvalue. has to be an address we can set result of right hand side to.

1.3 Naming conventions

1.3.1 Camel case

CamelCase lowerCamelCase UpperCamelCase

1.3.2 Kebab case

Kebab-Case Upper-Kebab-Case lower-kebab-case

1.3.3 Snake case

Snake_Case lower_snake_case Upper_Snake_Case

1.4 Introduction

1.4.1 Reserved

_C where C is any capital as variables for C. reserved.

Chapter 2

Integer addition and subtraction in C

2.1 Introduction

2.1.1 Addition

```
int a = 1+2;
```

```
int a = 1;
int b = a+2;
```

```
int a = 1;
a+=1
```

2.1.2 Integer overflow

2.1.3 Subtraction

int a = 2 - 1;

```
int a = 2;
int b = a-1;
int c = -a;
```

```
-=
```

2.2 Increments and decrements

2.2.1 Increment

int a = 1; a++;

Ambiguity:

a=a++;

2.2.2 Decrement

int a = 1; a–;

Ambiguity:

a=a–;

Chapter 3

Integer multiplication and division in C

3.1 Introduction

3.1.1 Multiplication

*

*=

3.1.2 Division and modulo division

/

/=

Chapter 4

Evaluating mathematical expressions using abstract syntax trees, and -fix notation

4.1 Introduction

4.1.1 Introduction

Chapter 5

Types of integers in C, characters and casting

5.1 Introduction

5.1.1 Integer types in C

5.1.2 Standard (signed) integers

Following are equivalent.

Signed means includes +ve and -ve numbers.

These are at least 16-bit (ie could be 32, 64, 16, or something else - up to the implementaion)

In 16 bit this goes between -32,767 and 32,767.

```
int a = 1;
signed int a = 1;
signed a = 1;
```

5.1.3 Unsigned integers

If these are 16-bit these are between 0 and $65,535$.

```
unsigned int a = 1;
unsigned a = 1;
```

5.1.4 Short integers

These are at least 16 bit, and equal or lesser than standard integers in their bits.

```
short int a = 1;
signed short int a = 1;
signed short a = 1;
short a = 1;
```

Unsigned versions.

```
unsigned short int a = 1;
unsigned short a = 1;
```

5.1.5 32-bit

Guaranteed to be at least as big as int, and at least 32 bit.

```
long int a = 1;
signed long int a = 1;
signed long a = 1;
long a = 1;
```

Unsigned versions:

```
unsigned long int a = 1;
unsigned long a = 1;
```

5.1.6 64-bit

Guaranteed to be at least as big as long, and at least 64 bit.

```
long long int a = 1;
signed long long int a = 1;
signed long long a = 1;
long long a = 1;
```

Unsigned versions:

```
unsigned long long int a = 1;
unsigned long long a = 1;
```

5.1.7 char

At least 8 bit.

char can be unsigned or signed.

```
char a = 1;
```

If signed is between -127 and 128 (if 8 bit)

```
signed char a = 1;
```

If unsigned is between 0 and 255 (if 8 bit)

```
unsigned char a = 1;
```

5.1.8 Using American Standard Code for Information Interchange (ASCII)

```
char a = "a";
```

5.1.9 Casting

```
unsigned short int a = 1;
unsigned long long b = (long long) a;
```

Chapter 6

Goto and switch in C

6.1 Go To

6.1.1 Go to

```
label_name:
.
.
.
goto label_name;
```

6.2 Switch-case

6.2.1 Switch-Case

```
int x;
switch(a)
{
case 1:
  x = 1;
  break;
case 2:
  x = 0;
  break;
case 3:
  x = 2;
  break;
default:
  x = 5;
```

```
}
```

If break isn't used, all following cases will be run.

Chapter 7

const, register and volatile in C

7.1 Introduction

7.1.1 Introduction

const tells compiler value will not change.

volatile tells compiler that value is prone to change.

register hints that the variable should be stored in a register.

Generally only const is useful, as the others can be automated by the compiler. Using const allows errors to be detected if the const is not treated as a const in code.

Chapter 8

The C preprocessor

8.1 Introduction

8.1.1 define

use of #define

```
#define PI 3.149
x=PI*4;
```

can define functions using macros can define

```
#define TIMES_THREE(x) (x * 3)
y=TIMES_THREE(5);
```

can turn on DEBUG eg

```
#define DEBUG
```

or do with compiler eg

```
GCC -DDEBUG (D then macro name)
```

why use const over define? handled by compiler not preprocessor. means you can do type checking

8.1.2 include

8.1.3 undef

8.1.4 ifdef

ifdef thing: following prints "a" only if DEBUG is defined

```
#ifdef DEBUG
```

```
printf("a");
#endif
```

Chapter 9

enum and const

9.1 Introduction

9.1.1 enum

```
enum flag {const1, const2, ..., constN};
const1 = 0 by default
```

can do const1+1 etc

Changing default values of enum constants

```
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

can initialse from enum

```
enum week day;
```

or when defining

```
enum week{Mon, Tue, Wed}day;

day = Wed;
```

9.1.2 CONST and modifiable lvalues

use of "const" here? page? motivation? is it actually stored in memory or just compiled?

basically throws an error if you try to modify. still actually stored in memory. eg a function can take a const as an input. variable at compile time. const and run time.

distinction between unmodifiable lvalues and modifiable lvalues

```
const int a = 1;
a = 2; // bad, even though a is an lvalue.
```

Chapter 10

Bitwise operations in C

10.1 Introduction

10.1.1 Bitwise AND

```
int x = 12;
int y = 10;

z = x & y;
```

```
int x = 12;
int y = 10;

// Following two lines are equivalent.
x &= y;
x = x & y;
```

10.1.2 Bitwise OR

```
int x = 12;
int y = 10;

z = x | y;
```

```
int x = 12;
int y = 10;

// Following two lines are equivalent.
x |= y;
x = x | y;
```

10.1.3 Bitwise NOT

```
int x = 12;

int y =  ~x;
```

10.1.4 Bitwise XOR

```
int x = 12;
int y = 10;

z = x ^ y;
```

```
int x = 12;
int y = 10;

// Following two lines are equivalent.
x ^= y;
x = x ^ y;
```

10.1.5 Left bitshifts

Left shift shifts all bits to the left. In binary, left shift by 1 place is same as multiplying by 2, assuming no overflows.

We can left shift by any number of places.

The new bit on the furthest right is set to 0.

```
int x = 12;
int y = 2;

int z = x << y;
```

```
int x = 12;
int y = 2;

// Following two lines are equivalent.
x <<= y;
x = x << y;
```

10.1.6 Right bitshifts

Right shift shifts all bits to the right. In binary, right shift by 1 place is same as diving by 2 and dropping the remainder.

We can right shift by any number of places.

The new bit on the furthest left is set to 0.

```
int x = 12;
int y = 2;

int z = x >> y;
```

```
int x = 12;
int y = 2;

// Following two lines are equivalent.
x >>= y;
x = x >> y;
```

Part II

Other data types in C

Chapter 11

Pointers and arrays in C

11.1 Pointers

11.1.1 Pointers

Getting address of variable with &

variable pointers with & and * in c

```
int a = 1;
int * b = &a;
```

$*p$ assumes p is pointer and gets what it points at. $\&p$ assumes p is value and gets address of p.

$NULL$ is literal pointer to no valid data. same as 0.

pointer of zero doesn't work, reserved

11.1.2 Dangling pointers

```
int * p = NULL;
{
int a = 1;
p = &a;
}
```

p is dangling after because scope ends.

11.1.3 Void pointers

11.2 Arrays

11.2.1 Defining arrays

```
int vals[] = {1, 2, 3, 4, 5};
```

Can also define empty array:

11.2.2 Accessing values in arrays

The following are the same. Difference is just syntactic sugar.

```
a[i]
*(a+i)
```

11.2.3 Array shifting

Relevant for insertion in place.

11.2.4 sizeof()

This function gets the length of an array. It is determined at compile time.

11.2.5 Buffer overflows

11.2.6 Multi-dimensional arrays

The following are the same. The difference is just syntactic sugar.

```
a[i][j]
??
```

(Doesn't this need to know dimensions of all but last one? How does this work if size not known in eg a function?)

11.2.7 Sort

Regular arrays. If want to insert or remove can create new array with new size. Traversal of array is O(1)

array slices etc operations

c increase size of array. automatically creates array twice as big?? how does this work with stack??

size of array unknown at runtime unless provided

arrays if you have array of length 4 and you look at 5 what happens? Does compiler prevent? What about if just have pointer to array?

Chapter 12

Strings in C

12.1 Introduction

12.1.1 Introduction

```
char c[] = "c string";
```

can declare. eg

```
char s[6];
s = "hello";
```

Can over declare, givng spare space.

```
char c[50] = "hello";
```

12.1.2 Null character

Compiler adds null character automatically.
0 is null character.

```
char c[] = {"h", "e", "l", "l", "o", "\0"};
```

12.1.3 Strings and pointers

Strings are arrays of characters, and so we can use pointers to access.

$*c$ returns "h". $*(c+1)$ returns "e".

12.2 Strings

12.2.1 Encodings

Can represent same information in multiple ways?

12.2.2 American Standard Code for Information Interchange (ASCII)

12.2.3 UTF (Unicode)

12.2.4 Sort

strings as data type. similar to array? but diff functions. eg find "cat" in "concatenate"

can subinstr. upper. lower. proper. strpos

can regex. section on regex in data types?

character: number maps to character

string literals

array of chars

buffer overflows on strings (already discussed them in the context of arrays)

can use sizeof. size determined at compile time

null terminated string

string literals: escaping characters

Chapter 13

Floats in C

Chapter 14

Structs in C

14.1 Introduction

14.1.1 Introduction

```
struct myStructure {
  int myNum;
  char myLetter;
};

struct myStructure s1;

s1.myNum = 13;
s1.myLetter = 'B';
```

14.1.2 Accessing via pointers

The following two ways of accessing via a pointer are the same.

```
struct myStructure {
  int myNum;
  char myLetter;
};

struct myStructure s1;

s1.myNum = 13;
s1.myLetter = 'B';
```

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

(*p).myNum;
(*p).myLetter;

p->myNum;
p->myLetter;
```

Chapter 15

Booleans in C

15.1 Introduction

15.1.1 Introduction

_Bool is a variable type in base C.

```
_Bool x = 1;
```

Part III

Structured and procedural programming with C

Chapter 16

Global and local scope in C

16.1 Introduction

16.1.1 Blocks

blocks

16.1.2 Global

Global variables are allocated at compile time.

Initialised variables are set to their given value.

Uninitialised variables have space set aside for them.

Chapter 17

Relations and logical operators

17.1 Relations

17.1.1 Equals

```
int x = 1;
int y = 1;

int z = x == y;
```

17.1.2 Not equals

```
int x = 1;
int y = 1;

int z = x != y;
```

17.1.3 Greater than and less than

```
int x = 1;
int y = 1;

int z = x > y;
```

```
int x = 1;
int y = 1;

int z = x < y;
```

17.1.4 Greater or equal than and less or equal than

```
int x = 1;
int y = 1;

int z = x >= y;
```

```
int x = 1;
int y = 1;

int z = x <= y;
```

17.2 Logic

17.2.1 Logical AND, and short-circuiting for AND

Returns 1 if both positive. 0 otherwise.

If first is not truthy, the second is not evaluated.

```
int x = 1;
int y = 0;

int z = x && y;
```

17.2.2 Logical OR, and short-circuiting for OR

Returns 1 if either positive. 0 otherwise.

If first is truthy, the second is not evaluated.

```
int x = 1;
int y = 0;

int z = x || y;
```

17.2.3 Logical NOT

```
int x = 1;
int y = 0;

int z = !(x && y);
```

Chapter 18

Control flow and logic in C and the structured program theorem

18.1 If-Then-Else

18.1.1 If-then-else

`if () x else y`

can group multiple with block

`if () {} else {}`

block curly counts as single command

can pass varibales to logical evaluation

```
int x = 1;
if (x)
```

evaluating ints and longs. seems to be if ¿= 1 then true otherwise false? does something on char too?

18.1.2 Conditional operator

If we want to set the value of something based on a conditional, eg:

```
if (a > b) {
    result = x;
}
else {
```

```
    result = y;
}
```

We can instead do the following

```
result = a > b ? x : y;
```

If we have something of the form:

```
if (a) {
    result = a;
}
else {
    result = y;
}
```

We can write the following in some implementations of C.

```
result = a ? : y;
```

Known as the Elvis operator (because of "?:").

18.2 While loops

18.2.1 While

```
while (n<10) {;
  n++;
}
```

18.2.2 Do-While

do while in addition to while. do while means the loop is run at least once

```
do {
  // the body of the loop
}
while (testExpression);
```

18.3 For loops

18.3.1 For loops

```
int a = 1;
for (int n=10; n>0; n--) {
    a = a + 5;
}
```

18.3.2 Continue in for loops

```
int a = 1;
for (int n=10; n>0; n--) {
    if (n==5) continue;
    a = a + 2;
}
```

18.3.3 Break in for loops

```
int a = 1;
for (int n=10; n>0; n--) {
    if (n==5) break;
    a = a + 1;
}
```

18.3.4 Using multiple variables

for loops like this. can do multiple variables like below.

```
for (int n=0, i=100 ; n!=i ; n++, i--)
{
   // whatever here...
}
```

18.4 Structured program theorem

18.4.1 Introduction

Can do without gotos, and use structured loops instead.

Chapter 19

Functions and stack memory in C

19.1 Introduction

19.1.1 Basics

```
void foo() {
    int a = 0;
    a++;
    printf("%d%d\n", a);
}
```

calling this print 1, 1, 1, 1

c is staticly typed. functions can only take correct type

scoping (lexical scoping?) static binding: global variable. variable points to same address throughout dynamic binding: ones on stack (or heap). variable does not point to same address throughout allocated within functions. automatically freed when function finished.

variables inside functions are local, can't be accessed by other functions. even if called from within another function

c doesn't accept default parameters in function

stack memory: allocated at run time. data released when function finished

19.1.2 The main function

Can move main code into its own function, reducing global scope and risk of mistakes.

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

19.1.3 Splitting out function declaration and definition

splitting out declaration and definition of functions allow definitions to be placed more conveniently. eg main at top rather than bottom declarations of functions (or variables) must be made before referenced in functions

as with variables, defining functions takes two forms int increment (int a); int increment (int a) a = a + 1; return a (is this right? can we define functions after declaring without repeating whole of declaration?

declaration is also known as the function prototype

19.1.4 Static variables in functions

Static variables in functions are global variables, but only accessed via scope of function.

```
void foo() {
    static int a = 0;
    int b = 0;
    a++;
    b++;
    printf("%d%d\n", a, b);
}
```

calling this print 11, 21, 31, 41 etc

19.1.5 Pointers to functions

function pointers in c

passing functions as arguments in c. possible!

19.1.6 SORT

passing array length in functions. can we use sizeof? probably not because sizeof figured out at compile time?

when arrays passed as arguments, it;s the pointer to the first element.

c. stack pivot exploit

stack buffer overflow + point outside area reserved for stack, in particular if overwrite return address

call stack

stack pointer

stack memory in c literally a stack everytime allocatte an integer, move the stack allocator ahead however many needed, then return the memory address

can't define function inside function.

Chapter 20

Side effects and sequence points in C

20.1 Introduction

20.1.1 Introduction

static in function means can have side effects.

can use const to prevent side effects

```
int myfunction (const int * x)
```

means can't modify x in function.

concept of "pass by reference" in functions, side effects

can also have side effects if global variables exist

c allows you to put function calls inside if statements. means both won't necessarily be run

function side effects in c. + happen with pointers

Chapter 21

Typedef in C

21.1 Introduction

21.1.1 Introduction

```
typedef int my_score;
void my_function(my_score score) {}
```

allows you to write functions that accept int etc, but only specificly defined, not ints more generally. just allows for safer code writing, optional

If we are doing a typedef for a struct it will look like this:

```
typedef struct my_score_struct {
   int a;
   char b;
} my_score;
void my_function(my_score score) {}
```

We can also split out the struct definition and the alias.

```
struct my_score_struct{
   int a;
   char b;
};
typedef my_score_struct my_score;

void my_function(my_score score) {}
```

Chapter 22

Optimising compiled C code, including tail call optimisation and the restrict keyword

22.1 Introduction

22.1.1 Tail call optimisation

22.1.2 restrict

restrict keyword for pointers.

Indicates to compiler that no other pointer points to this variable.

Leads to more efficient code.

Part IV

Freestanding headers for C

Chapter 23

Macros and the C preprocessor

23.1 Macros

23.1.1 Macros

Not separate, code is replaced before it is run

```
#define PI 3.14
```

These are then expanded before compiling.

23.1.2 Parametised macros

We can also replace using functions.

For example we could do:

```
print(3 + 1)
print(2 + 1)
```

or

```
#define plusOne(x) (x+1)
print(plusOne(x))
print(plusOne(x))
```

Chapter 24

stddef.h - including size_t and the sizeof operator

24.1 Introduction

24.1.1 Introduction

imported via other libaries, so don't need to directly call

24.1.2 size_t and the sizeof operator

```
int x = 1;
size_t x = sizeof(x);

size_t y = sizeof(int);
```

sizeof operator is calculated at compile time. *sizeof* is in base C, but the data type *size_t* is not, and requires the stddef.h header.

24.1.3 NULL

Chapter 25

Macros for boolean variables using stdbool.h

25.1 Introduction

25.1.1 Introduction

Introduces a macro "bool" for "_Bool".

Also "true" for 1 and "false" for 0.

```
_Bool x = 1;
bool y = true;
```

Chapter 26

Allowing indefinite function arguments using stdarg.h

26.1 Introduction

26.1.1 Introduction

The following declares the new type va_list at the start.

Then between va_start and va_end we can access arguments using va_args repeatedly, while giving it the type to take out.

```
void my_function(int count, ...) {

    va_list args;
    int i;
    va_start(args, count);
    for (i = 0; i < count; i++) {
        int num = va_arg(args, int);
    }
    va_end(args);
}
```

When we traverse va_arg we can't go back. Another approach is to use va_copy to take a copy of the args before going through them.

```
void my_function(int count, ...) {

    va_list args;
```

```
    int i;
    va_start(args, count);
    va_copy(args_copy, args);
    for (i = 0; i < count; i++) {
        int num = va_arg(args, int);
    }
    for (i = 0; i < count; i++) {
        int num = va_arg(args_copy, int);
    }
    va_end(args);
    va_end(args_copy);
}
```

Chapter 27

Adding fixed width integers with stdint.h

27.1 Introduction

27.1.1 Introduction

This adds new types of fixed width integers which won't vary by computer.

To recap, existing variables, such as int have a lower bound but not an upper bound, as such the behaviour can vary machine to machine.

The types are:

+ int8_t: Signed 8 bit + uint8_t: Unsigned 8 bit + Equivalents for 16, 32 and 64 bit (eg uint64).

The header also provides macros to indicate the min and max of these types, including eg INT8_MIN, UINT64_MAX.

Chapter 28

Documenting non-returning functions using stdnoreturn.h

28.1 Introduction

28.1.1 Introduction

Adds ability to prefix functions with noreturn.

This hints to the compiler and reader that the function will not return. Eg if there is an infinite loop.

```
noreturn void loop_forever() {
    int x = 1;
    while(1) {
        x = 2;
    }

}
```

Chapter 29

Getting and setting type width using stdalign.h

29.1 Introduction

29.1.1 Introduction

Get number of bytes:

```
alignof(int);
alignof(x);
```

Set number of bytes:

```
alignas(double) int x;
```

Also adds macros to check if this is available:

```
__alignas_is_defined
__alignof_is_defined
```

Chapter 30

Getting upper and lower integer limits using limits.h

30.1 Introduction

30.1.1 Introduction

Provides macros for bits and stuff

CHAR_BIT: Bits in byte MB_LEN_MAX: Max bytes in multi-byte character

Also does upper and lower limits of variable types SCHAR_MIN: Signed character min CHAR_MAX: Unsigned character max.

These are available for CHAR, UCHAR, SHRT, USHRT, INT, UINT, LONG, ULONG.

Chapter 31

Macros for operations (eg ”and”, ”bitand”) using iso646.h

31.1 Introduction

31.1.1 Macros for logic

Adds ”and”, ”or”, ”not” and ”not_eq”.

```
int x = 1;
int y = 0;

// These are the same.
int z = x && y;
int z = x and y;
```

```
int x = 1;

// These are the same.
int z = ! x;
int z = not x;
```

```
int x = 1;
int y = 0;
```

```
// These are the same.
int z = x || y;
int z = x or y;


int x = 1;
int y = 0;

// These are the same.
int z = x != y;
int z = x not_eq y;
```

31.1.2 Macros for bitwise operations

```
int x = 12;
int y = 10;

// These are the same.
int z = x & y;
int z = x bitand y;

// These are the same.
int z = x | y;
int z = x bitor y;

// These are the same.
int z = x ^ y;
int z = x xor y;
```

Also eq versions

```
int x = 12;
int y = 10;

// These are the same.
int y &= x;
int y and_eq x;

// These are the same.
int y |= x;
int y or_eq x;
```

```
// These are the same.
int y ^= x;
int y xor_eq x;
```

And also bitwise not

```
int x = 10;

// These are the same.
int y = ~x;
int y = compl x;
```

Chapter 32

float.h

32.1 Introduction

32.1.1 Introduction

Part V

Compiling C code

Chapter 33

Static single-assignment form

33.1 Introduction

33.1.1 Introduction