Notes 3.0: A partial overview of C

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2014 session 1

An example program

The programs puzzle_1.c and puzzle_2.c find all sets of positive integers $\{x, y, z\}$ such that

- x, y and z have no occurrence of 0,
- every nonzero digit occurs exactly once in one of x, y or z, and
- x, y and z are perfect squares.

We use them to provide an overview of the C language.

The first variant is simpler but the second one makes a better use of memory. It is worth comparing them and appreciate their respective merits.

Preprocessor commands (1)

Before it is compiled, the source code of a program is modified by the preprocessor, that in particular, processes preprocessor commands, such as `#define` and `#include`.

A line of the form `#define` *`MACRO_NAME expression`* defines `MACRO_NAME` to be a user-defined macro for *`expression`*, and requests the preprocessor to replace all occurrences of *`MACRO_NAME`* in the source code by *`expression`*.

So the only occurrence of `MAX` in both variants of the program will be replaced by `9876532`, and the only occurrence of `TEN_ONES` in the second variant will be replaced by `1023`.

A common convention is that all letters in the name of user-defined macro be written in upper-case.

Preprocessor commands (2)

A line of the form `#include <file_name.h>` requests the preprocessor to replace that line with the contents of *file_name*.h, a **h**eader file (whose extension ends in .h by convention) of the standard library (as indicated by the angle brackets). We request to include

- the stdio.h—where stdio stands for **st**an**d**ard **i**nput/**o**utput—and math.h header files, because they contain information needed by the compiler about the two library functions we use, namely, printf()—where f stands for **f**ormatted, and sqrt()—which stands for **sq**uare **r**oo**t**.
- the stdlib.h—where stdlib stands for **st**an**d**ard **lib**rary—and stdbool.h—where stdbool stands for **st**an**d**ard **bool**ean— header files, because we use the macros EXIT_SUCCESS and EXIT_FAILURE (defined in stdlib.h), true, false and bool (defined in stdbool.h) all of whose occurrences in the source code will be replaced by the preprocessor by 0, 1, 1, 0 and _Bool, respectively.

Functions (1)

A function possibly has parameters and possibly returns a value, all of a particular type. Both programs define two functions:

- `main()` that has no parameter, and returns a value of type `int`;
- `test()` that has two parameters, a first parameter of type `int` and a second parameter of type pointer to `int`, qualified with the `const` keyword, and returns a value of type `_Bool`.

A pointer is an address, that of a memory location where a value is stored; so a pointer parameter gives access to an address *and* a value, whereas a nonpointer parameter gives access to a value only.

`bool test(int, int *const);` is a function prototype. A function prototype gives the compiler information about how that function is meant to be used and what it is meant to do, and that information is needed by the compiler to do its job.

Functions (2)

More generally, a function prototype indicates

- whether the function has parameters, and if it does, how many and of which types;
- whether the function returns a value—using a return statement—, and if it does, the type of the latter, which is also the type of the function, whereas the type of the function is `void` if no value is to be returned.

The `stdio.h` and `math.h` header files contain the prototypes of the functions `printf()` and `sqrt()`, respectively.

- `printf()` is of type `int` and has a variable number of parameters: a format string as a first parameter, followed by as many parameters as format specifiers, that all start with `%`, in the format string; here we use format specifiers that all end in `d`, to print **d**ecimal numbers.
- `sqrt()` is of type `double` and has a single parameter of type `double`.

Functions (3)

Every program contains the definition of a `main()` function, of type `int`, that either has no parameter or has parameters of some kind.

A program starts by executing `main()`, which should eventually return a value that can indicate to the operating system how it seemed to have run: `0` if seemingly successfully, and a nonero value if seemingly unsuccessfully, for reasons that can be indicated by the actual value being returned.

Here `main()` can call or summon other functions (`test()`, `sqrt()` and `printf()`), thanks to statements where the called function is invoked with specific arguments providing specific values to the function parameters.

- When `sqrt()` is called, it is passed an argument of type `int` that is automatically converted to a `double` in accordance with its prototype.
- When `printf()` is called, the value it returns is ignored.
- Function `test()` returns `0` or `1` and as a side effect, possibly modifies the value stored at the address passed as second argument by `main()`.

Declarations and assignments

A number of variables are declared (`nb_of_solutions`, `max`, `i`, `j`, `k`, `i_square`, `j_square`, `k_square`, `i_digits`, `i_and_j_digits` and `i_and_j_and_k_digits` in `main()`, `dig` in `test()` and in `main()` of the first version) with a mention of their type, and most of them are initialised at declaration.

The variables differ by their scope: they are visible and can be used from where they are declared, only in the enclosing block.

Variable names, like function names, must be sequences of (lower case or upper case) letters, underscores and digits, and not start with a digit.

Assignments allow one to modify the value of a variable. Function `test()` contains 3 assignment statements.

The value of a variable can also be modified with an increment or decrement operator: for instance, the statements `++j` in `main()` increments the value of `j` (by `1`).

Tests

`main()` can perform a number of tests thanks to a number of if statements and one if-else statement, and `test()` can also perform tests thanks to one if statement.

The result of a test is either true or false. If true, the statement or statements that make up the body of the if statement or if clause (in the case of an if-else statement) are executed, otherwise they are skipped and in the case of an if-else statement, the statement or statements that make up the body of the else clause are executed.

A value of `0` is interpreted as *false* and causes a test to fail; any other value is interpreted as *true* and causes a test to pass.

The unary boolean operator `!` changes a value of `0` to `1` and a nonzero value to `0`. For instance, in `main()`, if `test(..., ...)` returns `true` (that is, `1`) then `!test(..., ...)` evaluates to `0` and the test `if (!test(..., ...))` fails.

Repetitions (1)

`main()` and `test()` can execute a sequence of statements repeatedly thanks to some for statements or for loops, and thanks to some while statement or while loop.

A for loop has an initialisation, a test, an update, and a body.

- The first iteration of a for loop starts by executing the initialisation part and then passing the test, while every other iteration starts by executing the update part and then passing the test.
- Never failing the test of an executing for statement causes the program to loop, in which case it might be necessary to send this running process a signal (often, Control C) to kill it.

For instance, `for (int i = 1; i <= max; ++i)` is the beginning of a for statement whose body is expected to be executed `max` times, that is, 3,142 times, because it contains no statement that can modify the value of `i` or cause the program to break out of the loop.

Repetitions (2)

A while loop only has a test and a body.

In `test()` the test of the while loop is passed as long as `i` evaluates to a nonzero value. For the program not to loop, it is necessary that the value of `i` be changed in the body of the loop, which is the case thanks to the last statement: the number of times the loop is expected to be executed is at most equal to the number of digits in the decimal representation of `i` (that is, the integral part of the log of base 10 of `i` plus 1).

Being caught in an infinite while loop is as undesirable as being caught in an infinite for loop and is prone to the same effects, with the same potential remedies.

The flow of control of a repetition structure can be altered with `break` statements to exit out the loop, and with `continue` statements to jump back to the top of the loop without executing the remaining statements.

Arrays

The first version of the program makes use of 3 arrays of 10 `int`s. The first of this array is initialised at declaration time with the statement `int i_digits[10] = {1};` that sets its first element to `1` and all others to `0`. The second and third arrays have all their elements initialised with a `for` loop after they have been declared.

The elements that make up an array have to be all of the same type, and they are stored contiguously in memory.

When `i_digits`, `i_and_j_digits` or `i_and_j_and_k_digits` is passed as second argument to the function `test()`, the function actually receives as second argument the address where the first element of the array is stored, which is why its second parameter is declared to be of type pointer to `int`.

Operators (1)

The programs make use of:

- four of the five binary arithmetic operators: `*` for multiplication, `+` for addition, `/` for division, and `%` for remainder (`-` denotes subtraction);
- two of the four bitwise operators: `|` for binary (inclusive) or and `&` for binary and (`^` denotes binary exclusive or and `~`, unary not);
- one of the two binary shift operators: `<<` for left shift (`>>` denotes right shift);
- two of the four binary relational operators, which are `<`, `<=`, `>=` and `>` to denote the properties of being smaller than, smaller than or equal to, greater than or equal to or greater than, respectively;
- one of the two binary equality operators, which are `==` and `!=` to denote the properties of being equal or different, respectively;
- one of the boolean operators, namely, the unary `!` (negation), the others being the binary `&&` (conjunction) and `||` (disjunction).

Operators (2)

All those binary operators, as well as the assignment operator, namely `=`, are characterized by associativity rules, and precedence rules give some operators priority over others. For instance, `%` and `*` have the same priority, which is higher than the priority of `+`, which is higher than the priority of `<<`, which is higher than the priority of `=`.

Parentheses are used to write expressions that have to be parsed differently to the (unique) way imposed by those rules. For instance,

- `x = 1 + 2 * 3` would add `1` to the result of the multiplication of `2` by `3`, and assign the resulting value, namely `7`, to `x`, whereas `x = (1 + 2) * 3` would multiply the result of the addition of `1` and `2` by `3`, and assign the resulting value, namely `9`, to `x`;
- as `%` associates to the left, `x = 12 % 5 % 3` would assign `2` to `x`, whereas `x = 12 % (5 % 3)` would assign `0` to `x`.

Operators (3)

The expression `dig = 1 << i % 10` causes `1` to be “shifted to the left” `i % 10` times, with `0`s filling the gaps, and the resulting sequence of bits to become the new binary value of `dig`. For instance, if `i` was equal to `314` then `dig = 1 << i % 10` would assign `16` to `dig` (because the binary representation of `16` is `10000`).

There are assignment shortcuts for all binary arithmetical, bitwise and shift operators. Two of them are used in the programs:

- `i /= 10` as a shortcut for `i = i / 10`;
- `*pt_to_digits |= dig` as a shortcut for `*pt_to_digits = *pt_to_digits | dig`.

The expression `i /= 10` causes `i` to be divided by `10` and the integral part of the result to become the new value of `i`. For instance, if `i` was equal to `654` then `i /= 10` would change the value of `i` to `65`.

Operators (4)

The expression `*pt_to_digits |= dig` uses the dereference operator `*` to get the value *v* stored at address `pt_to_digits`, and causes the sequence of bits that makes up the binary representation of *v* to be "or"ed with the sequence of bits that makes up the binary representation of `dig` and *v* to be changed to the value whose binary representation is the resulting sequence of bits. For instance, if *v* was equal to `52` and `dig` to `17` then `*pt_to_digits |= dig` would change *v* to `53` at location `pt_to_digits` (because the binary representations of `52`, `17` and `53` are `110100`, `10001` and `110101`, respectively).

The address operator `&` is the dual of the dereference operator `*`: applied to a variable, it returns the address where that variable stores its value. In both programs, `main()` passes as second argument to `test()` the address where `i_digits`, `i_and_j_digits` or `i_and_j_and_k_digits` store its value (10 `int`s for the first version, a single `int` for the second version).

The const keyword (1)

The intuitive interpretation of `const` is read-only. It is intended to give a function least privilege and not allow it to modify a value that is not meant to be modified.

The type of the second parameter of `test()` reads as constant pointer to int. This means that the compiler would not accept that in the definition of `test()`, some statement would try and change `pt_to_digits`'s value, with for instance an assignment such as `pt_to_digits = &dig` whose purpose would be to change the value of `pt_to_digits` to the address where variable `dig` stores its value.

The const keyword (2)

If the second parameter of `test()` was of type `const int *` or `int const *`, which reads as pointer to a constant int, then the compiler would not allow any statement in the definition of `test()` to change the value stored at the location passed as second argument. Hence the compiler would fail to produce an executable since the statement `*pt_to_digits |= dig;` is precisely meant to change that value.

A type of `const int *const` or `int const *const` would read as constant pointer to a constant int and would not allow address *and* value to be modified.

Whitespace (1)

Whitespace is used to separate tokens. The source code is parsed by "eating up" as many characters as possible to form a valid token, stopping when a space character is encountered, if ever.

For instance, the expression `i <= max` could be written `i<=max` as no valid token starts with `i<` nor with `<=m`.

There is a postfix decrement operator. By the previous principle, the compiler interprets the expression `a---b` as `a-- - b`, not as the equally syntactically valid (though semantically different) expression `a - --b`. Also by the same principle, the syntactically valid expression `a-- - --b` could not be written `a-----b` because the compiler would parse the latter as `a -- -- - b`, which is not a valid expression (the decrement operator cannot be applied twice), despite the fact that `a-----b` can be unambiguously decoded into a single valid expression.

Whitespace (2)

The amount of whitespace that separates tokens is irrelevant, and can consist of simple spaces, tabs or newline characters.

From a stylistic point of view, it is essential that whitespace be used *wisely* and *consistently* to maximise readability, in particular so that the code be properly indented.

Spaces are interpreted literally when they occur in string literals, that is, sequences of characters enclosed between double quotes, as illustrated in end_of_line_and_literal_strings.c, which also illustrates how a newline character can be escaped in the source file so that it be ignored.

Comments

Wise use of comments is essential, following common sense principles. A block of code should be commented if and only if a comment can simplify the task of understanding that block of code; if that is the case, the comment should be as concise and precise as possible.

Good choice of identifiers (variable names, macro names, function names) can make the code more readable and reduce the need for comments.

There are two kinds of comments, which together with the rules that govern their use, is illustrated in comments.c

Types

Both programs make use of a number of (sometimes qualified) types:

- `int`
- `_Bool`
- array of 10 `int`s
- pointer to `int`
- function returning an `int`
- function returning a `_Bool`

not to mention the implicit types of functions from the standard library and the implicit types of their arguments, such a *function returning* a `double` as type of `sqrt()`, whose argument is of type `double`.

A fundamental topic in the study of C is the study of its type system, with types.pdf providing the big picture.