COMP10002

Semester One, 2017

More C

Overview

Chapter 4

Chapter 5

Chapter 6

Chapter 4 – Concepts

Loops should have an invariant that is trivially satisfied by the initialization; is refined at each iteration; and when combined with the negation of the guard, represents the desired outcome.

Where a loop has alternative exit conditions, or multiple exit points, the subsequent computation paths will also differ.

Count from zero whenever possible, up to (but not including) `n`.

Loops can iterate over the input, either value by value, or character by character.

Chapter 4 – Program examples

- `forloop1.c`
- `daynumber.c`
- `forloop2.c`
- `forloop3.c`
- `savings.c`
- `daynumber-squash.c`
- `threen.c`
- `isprime.c`
- `readloop1.c`
- `fortcomm.c`

Chapter 4 – Exercise

Exercise

Write a nesting of loops that reads numbers from `stdin`, and for each value read, computes and prints the sum of the primes that are less than or equal to it, and then whether or not that sum is itself prime.

Chapter 4 – Summary

Key messages:

- `for` loops and `while` loops *feel* different, but are almost identical
- Loop guards are integer-valued expressions; loop initializers and iterators are expressions
- Loops can iterate over input data through the use of the return value from `scanf`
- Avoid `do` loops
- Use a consistent layout and style to make your programs readable.

Chapter 5 – Concepts

Functions provide abstraction, to complement calculation, selection, and iteration.

Like all variables and constants, functions have a type signature that is declared in advance of their use.

Functions can be separately compiled to make modules and libraries.

There is a wide range of standard C function libraries, for mathematical computation, character processing, string handling, etc.

Chapter 5 – Program examples

- `savingsfunc.c`
- `isprimefunc.c`
- `usemathlib.c`
- `savingsfuncgen.c`
- `triangle.c`
- `hanoi.c`
- `croot.c`
- `evenodd.c`

Chapter 5 – Calling a function

Argument expressions are evaluated in the calling context.

Argument values are copied into local variables in function.

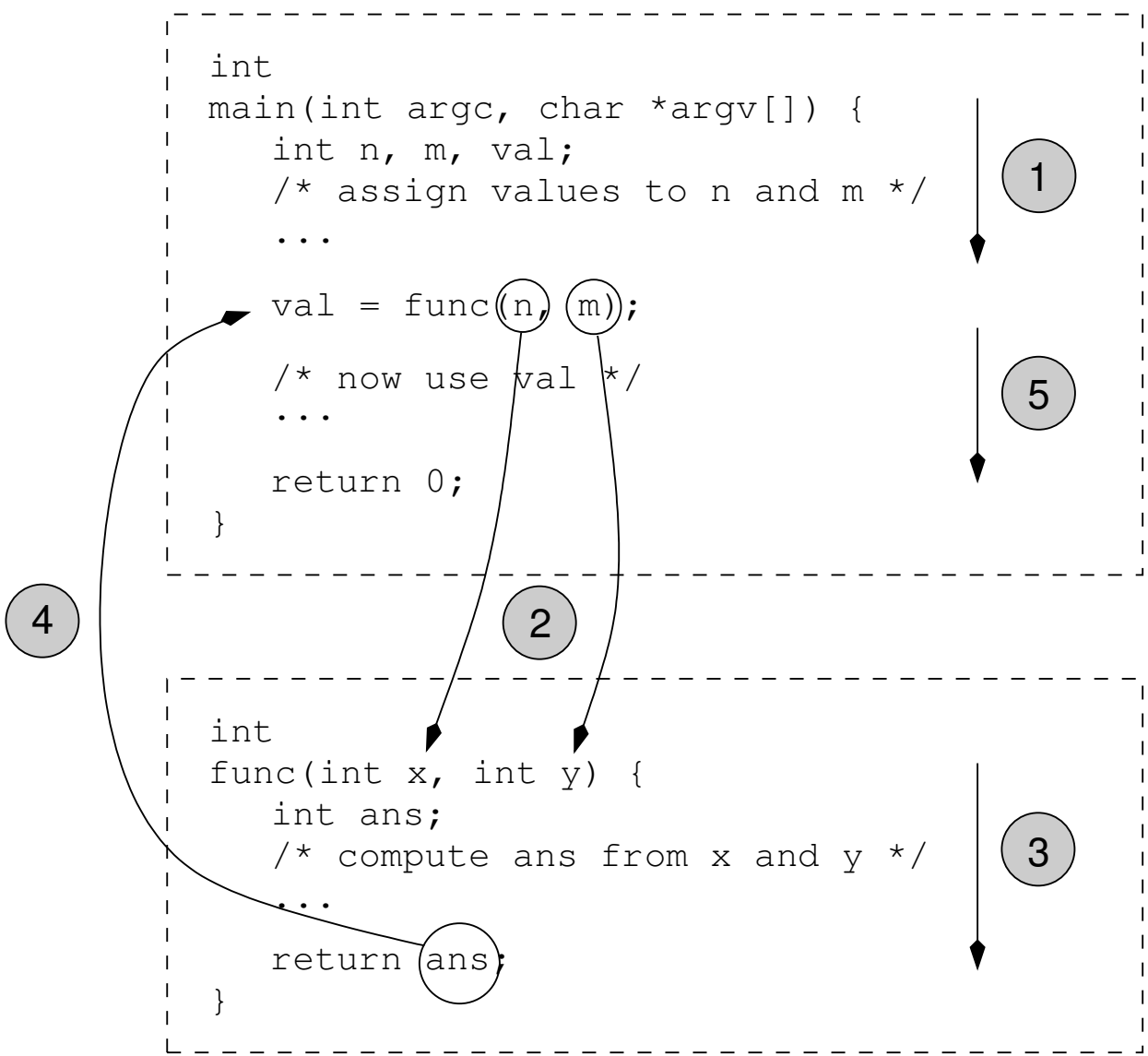
Function executes until `return` or end reached.

Return expression, if any is computed in context of function.

Function exits, all local variables destroyed.

Return value is made available in calling context.

Chapter 5 – Calling a function



Chapter 5 – Parameter passing

Variables (and expressions) are passed as values, copied into local variables.

When pointers are required in a function, they are constructed as pointer expressions, then copied into local variables (Chapter 6).

Arrays are passed as pointers to the first element in the array (Chapter 7); `struct`'s are passed as values, and copied into local variables (Chapter 8).

All non-`static` variables are destroyed when the function returns. Best to avoid `static` variables if possible.

Chapter 5 – Exercise

Exercise:

Write a function that takes three `int` arguments and returns the median (middle) one.

Chapter 5 – Summary

Key messages:

- Functions provide a mechanism for abstraction
- The values of arguments to functions are copied in to local variables at the time the function commences
- Changes in the function to arguments do *not* affect variables in the calling context
- Recursion provides another form of iteration.

Summary of Chapters 2 to 5

C and Python are alike, because:

- They are imperative languages
- They offer a range of arithmetic and logical operations
- They offer a range of control structures, including selection, iteration, and recursion
- Function arguments are received as initial values of local variables
- Libraries are available for a wide range of other operations.

Summary of Chapters 2 to 5

C and Python are different, because:

- C program structure is indicated by semicolons and braces, Python program structure by layout
- C integer arithmetic is bounded, and silently overflows
- C does not have an explicit `bool` type, and uses `int`
- C has static typing and requires declarations, Python has dynamic typing
- C is usually compiled, Python is usually interpreted
- Python provides in-built list, set, and dictionary structures, and operations on them
- C provides explicit pointer variables and pointer operations.

Chapter 6 – Concepts

All variables and compound structures are mapped to addresses in memory via execution-time pointer values.

C provides operations that manipulate pointer values, including `&`, `*`, `+`, and (Chapter 8) `->`.

Pointer variables and expressions derive their types from the underlying variables. So `int*` is type "pointer to `int`".

Functions that need to alter their arguments must receive pointers; the corresponding call must provide addresses of variables of the same type.

The declaration `void*` allows untyped pointers.

Chapter 6 – Concepts

The scope rules determine which variables can be accessed at each point in a program.

Variables declared in a function are local, or automatic; variables declared outside any function are global.

Argument variables are considered to be local to the function, but can also be shadowed by local variables declared within the function.

Local and global variables can also be declared with the modifier `static`. Static variables are initialized once, and thereafter retain their value through the execution.

Chapter 6 – Program examples

- `void.c`
- `scope1.c`
- `scope2.c`
- `scope3.c`
- `scope4.c`
- `pointer1.c`
- `pointer2.c`
- `pointer3.c`
- `readnum.c`

Chapter 6 – Exercise 1

Case Study

Write a function that reads integers until it obtains one in the range given by its first two arguments. When a suitable value is read, it stores that value using its third argument, and returns the predefined constant `READ_OK`. If no suitable value is located, the predefined constant `READ_ERROR` should be returned.

- `readnum.c`

Chapter 6 – Exercise 2

Exercise

Write a function that orders its three `int` arguments from smallest to largest.

Chapter 6 – Summary

The scope rules determine which variables can be accessed at each point of a program.

Pointer arguments allow functions to to make changes to variables in the calling environment.

This facility is sometimes called call by reference; the alternative is call by value (which in fact is what C always does).

Pointers provide a mechanism for aliasing. It is a flexibility that is extremely useful, not just in functions, but needs to be treated with respect.

In C, pointers and arrays go hand in hand.