Programming, Problem Solving, and Abstraction

Chapter Five

Functions

Chapter 5

Chapter 5 – Concepts

- Calculation, selection, iteration, and abstraction.
- Functions as a way of hiding details and allowing reusing of software components.
- Function libraries.
- Recursion.
- Program development as a collection of functions.
- Testing.

5.1 Abstraction

Four programming techniques are provided in almost all languages:

- Calculation: doing arithmetic to compute new values
- Selection: choosing between alternative execution paths
- Iteration: repeating a computation until a desire goal is arrived at
- Abstraction: creating units which can be reused, and in which internal detail is hidden from outside inspection

5.1 Abstraction

If the a computation used at one place is also required at other places, it can be abstracted into a function.

Functions allow computations to be reused.

- `savingsfunc.c`
- `isprimefunc.c`

5.1 Abstraction

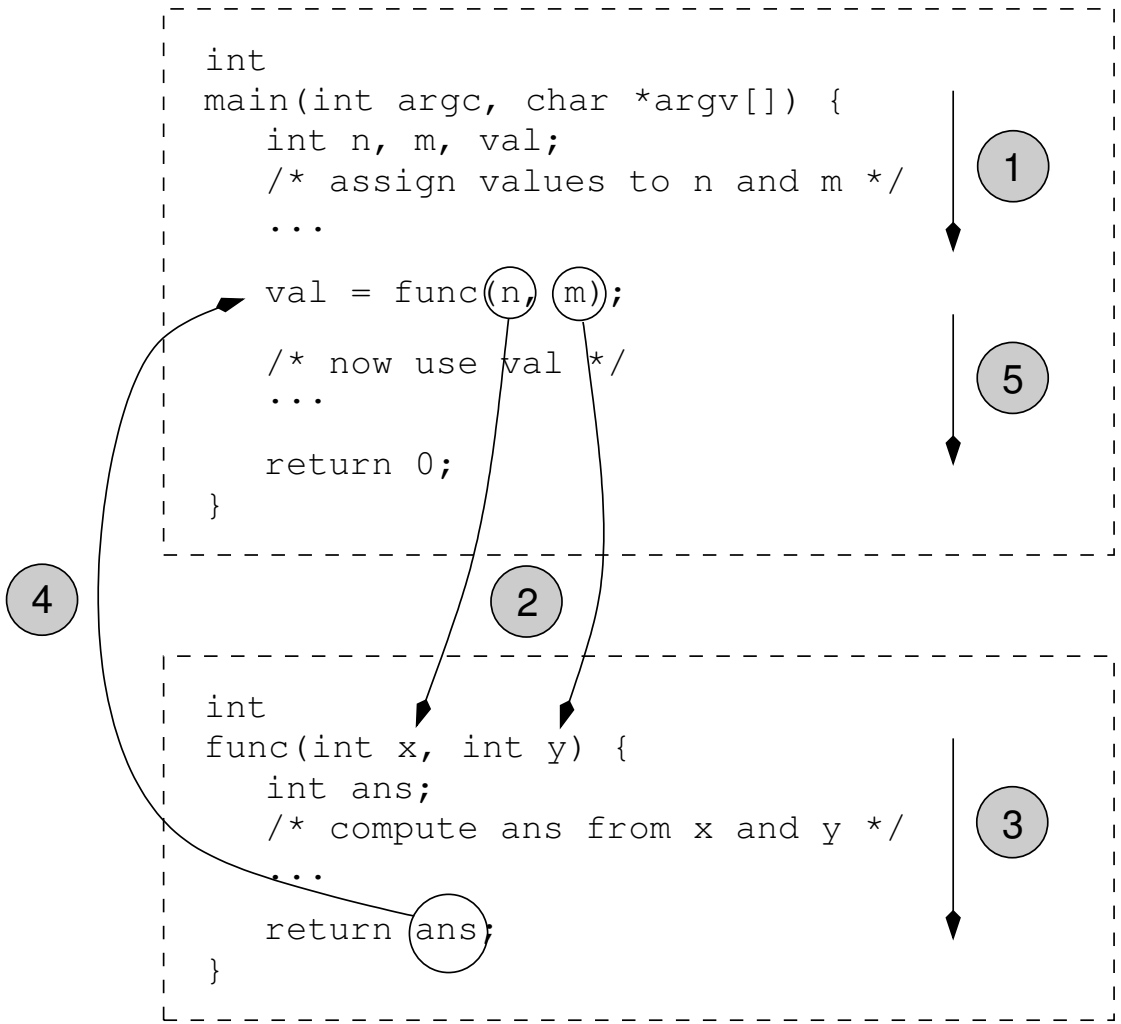
Each function takes arguments, and has a type.

The values of the arguments, plus any local variables that it declares, are used in the computation.

A value of the indicated type is then passed back via a `return` statement.

The function is called as part of an expression, and passed suitable argument values. It can be passed different arguments each time it is called.

5.1 Abstraction



5.1 Abstraction

In detail:

- The values of the argument expressions are evaluated using the context applicable at the point of call.
- Those values are assigned as the initial values of the corresponding argument variables, with any necessary assignment type conversions carried out.
- The body of the function is executed, through until a `return`.

5.1 Abstraction

- The expression associated with the `return` statement is evaluated in the context of the function.
- That value is passed back to the point at which the call was made.
- All local and argument variables in the function are destroyed.

Functions also help us think about programs – the various parts of the task to be performed are naturally implemented as separate functions.

5.1 Persistence

The argument variables store local copies of the argument expressions, and are discarded when the function returns.

Argument variables can be changed within the function, even if the corresponding argument expressions are not simple variables.

But the changes made are always lost.

5.1 Exercise 1

Write a function `int_max_3` that takes three `int` arguments and returns the largest of them as the value of the function.

5.2 Compilation with functions, option 1

Include function in the same source file as main program.

Typical structure:

- symbolic constants;
- prototypes for all functions;
- main function; then
- function definitions.

Compiler builds a single executable.

Execution commences with the `main` function.

5.2 Compilation with functions, option 2

Put function into separate source file.

Structure:

- Use `#include "func.c"` to bring function text from file `func.c` into main program file.
- Plus, use `#include "func.h"` to include a prototype.
- Combination of `func.h` and `func.c` form a module.

Having just one version of the function, makes reuse and maintenance easier.

5.2 Compilation with functions, option 3

Make use of separate compilation.

Structure:

- Include prototype from `func.h` in all modules or files needing to make use of that collection of functions.
- Compile using "`-c`" option to create "`.o`" object file.
- Use `gcc` to link together required object files and generate executable.
- Tool called `make` allows file dependencies to be specified, and minimal recompilations to be requested.

Avoids recompilation of "finished" or "standard" modules.

5.3 Library functions

Collections of related functions may be standardized and brought together into a library. C has many standard libraries.

The library described by `stdio.h` includes functions for input and output, and constants like `EOF`.

The library described by `stdlib.h` covers a range of general-purpose functions, and constants like `EXIT_FAILURE`.

5.3 Library functions

The library described by `math.h` covers mathematical functions, such as `sqrt`, `sin`, and `pow`.

Constants like `M_PI` are often provided in `math.h`, but are not part of the 1989 ANSI C standard, and may not be portable.

- `usemathlib.c`

The `-lm` flag tells the compiler to draw compiled functions from the maths library.

5.3 Library functions

Other useful libraries:

- `ctype.h` – character-level functions, such as `isalpha` and `tolower`
- `string.h` – functions on strings such as `strlen` and `strcpy`

5.3 Exercise 2

The function `islower` returns true if its `int` argument is a lowercase character. Write your own version of `islower` called `myislower`.

Then write your own version of `toupper`, also in `ctype.h`, which converts lowercase letters to uppercase letters, and leaves all other characters unchanged.

5.4 Generalizing the abstraction

To broaden the usefulness of a function, further arguments might be added, including ones not required at first.

Designing a function is a compromise between generality of purpose and simplicity of use.

If not required, fixed values can be passed in as the initial values of the additional arguments.

- `savingsfuncgen.c`

5.4 A common mistake

"Input" to a function is via the arguments; "output" via the return value.

Don't need to worry about how the argument values are being generated; just write the function as a stand-alone component that assumes that the arguments have values.

Normally no need for `scanf` or `printf` calls in a function. Exception is when (a) the task of the function is to perform explicit input or output operations; or (b) in error situations.

5.5 Recursion

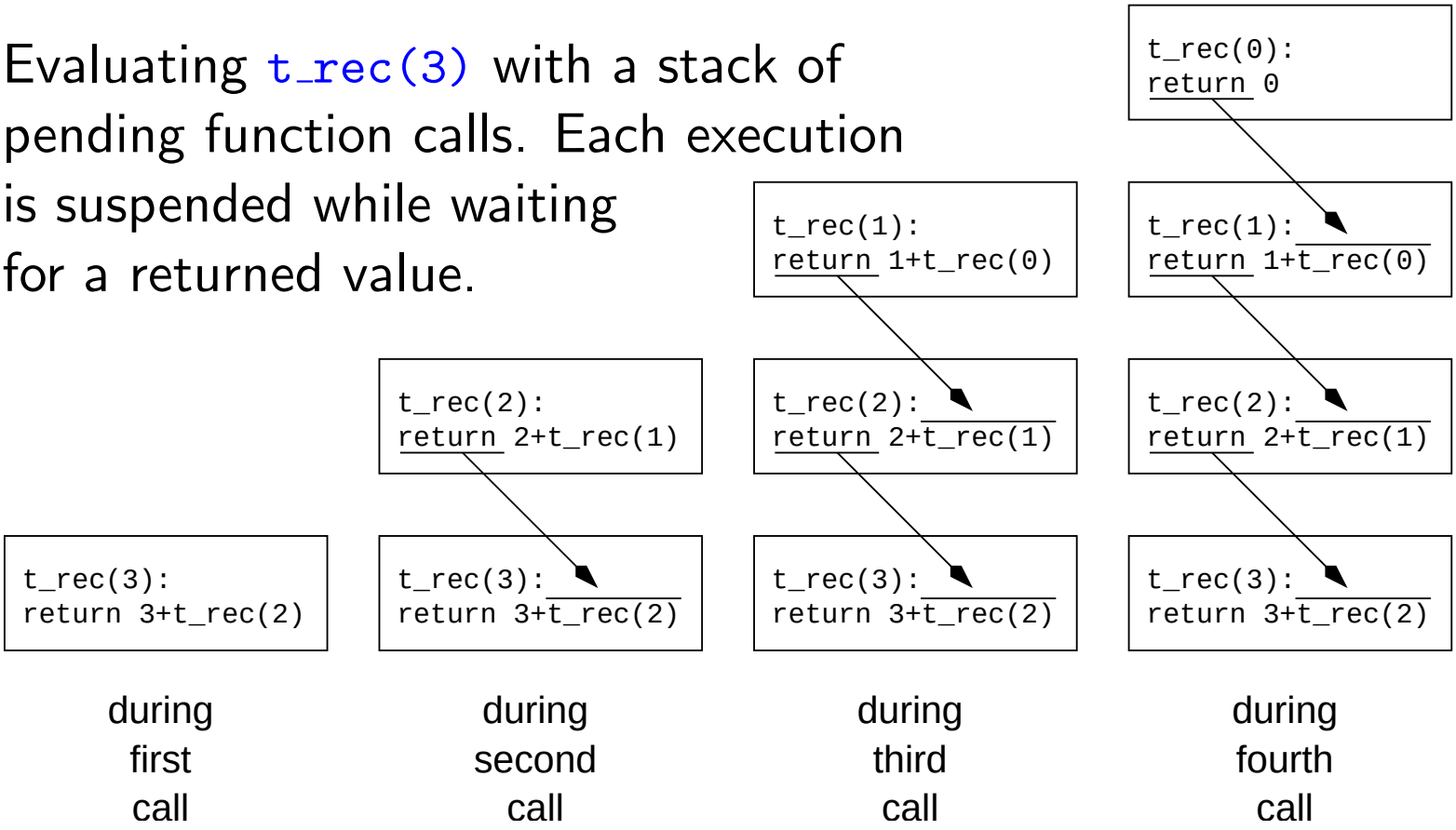
In some situations it is appropriate for a function to call "itself". This is recursion.

A base case must be provided if the recursion is not to be endless.

- `triangle.c`

5.5 Recursion

Evaluating `t_rec(3)` with a stack of pending function calls. Each execution is suspended while waiting for a returned value.



5.5 Recursion

Pending function executions are recorded by allocating each function call a frame on a stack. The frame contains local variables (including arguments), and a return address.

When an instance of a function returns, its frame is popped off the stack.

In some languages recursion replaces iterative control structures.

5.5 Exercise 5.14

The function $\log^*$ is defined by:

$$\log^* x = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^*(\log_2 x) & \text{otherwise.} \end{cases}$$

Write a C function `int logstar(double)`.

(What is the smallest number x for which $\log^* x \geq 4$?)

5.6 Case study

If x is an approximation of the cube root of v, then $x' = (2x + v/x^2)/3$ is a better approximation. For v between $10^{-6} \leq |v| \leq 10^6$, a total of 25 iterations of this formula is enough, starting from $x = 1.0$.

Write a function `cube_root` that receives a `double` argument and calculates and returns an approximate cube root for it. Then write a main program that to test it.

- `croot.c`

5.7 Testing functions and programs

- Design the functional decomposition – how to break up the task into smaller parts.
- Create stubs for the corresponding functions, and scaffolding that allows the first function to be written.
- When first function has been implemented and tested, change the scaffolding, and move to second function.
- If any "finished" function requires modification, be sure to fully test it all over again.

5.7 Testing functions and programs

Functions should check their arguments – the person or program calling the function may not understand its interface.

Invalid arguments should be reported, and then `exit` used to terminate program execution. Values read from files should be similarly tested.

Programs that silently continue their computations based upon erroneous values are dangerous.

5.7 Testing functions and programs

Functions and programs need to be exhaustively tested before being relied upon.

Tests should cover simple cases, complex cases, absurd cases, inputs just inside the design boundaries of the software, and inputs that lie outside the design parameters.

Careful design, evaluation, and recording of appropriate test cases is an integral part of the original software design.

Don't regard testing as an afterthought.

Chapter 5 – Summary

- Functions provide the ability to reuse "proven" code. In general, the more arguments, the greater the flexibility.
- Recursion is an important programming tool.
- Non-trivial programs are designed as a collection of functions.
- Then developed incrementally, one function at a time.
- Rigorous testing is required of all critical software. But remember that testing only ever shows the presence of errors, and never their absence.