Part V

Functions

Basics | Function Definition | Definition vs Declaration | Variadic Arguments | Return Values | Recursion

Declaration & Definition | External Variables | `inline`

Basics

Functions are used to break complex program into smaller pieces

- increase re-use of existing functionality
- better maintenance of the system
- hiding details that are not required to expose

Design principles in C (and UNIX as well)

- favor smaller functions (programs) that do one job but do it well
- composite functions (programs) to achieve higher goals

Function Definition

```
specifiers return-type function-name ( argument-declaration )
{
    compound-statement
}
```

Functions in C are defined by assigning and identifier (function-name) and list of arguments with a function body (compound-statement)

- functions definition must be in file scope

`specifiers` (optional)

- **storage class** specifiers (`static` , `extern`), and/or
- **function** specific **properties** (like `inline`)

Function Definition

```
specifiers return-type function-name ( argument-declaration )
{
  compound-statement
}
```

Functions in C are defined by assigning and identifier (function-name) and list of arguments with a function body (compound-statement)

- functions definition must be in file scope

`return-type`

- a basic or derived type specifier that states the result type of the function (like `int`, `unsigned long`, `char`, structs, pointers, ...)
- or void

Function Definition

```
specifiers return-type function-name ( argument-declaration )
{
   compound-statement
}
```

Functions in C are defined by assigning and identifier (function-name) and list of arguments with a function body (compound-statement)

- functions definition must be in file scope

`function-name`

- an identifier

Function Definition

```
specifiers return-type function-name ( argument-declaration )
{
  compound-statement
}
```

Functions in C are defined by assigning and identifier (function-name) and list of arguments with a function body (compound-statement)

- functions definition must be in file scope

`argument-declaration` (optional)

- either **`void`** , or a comma-separated list of parameters (type specifier and identifier)
- passing parameters to a function can be done by

call-by-value	*or*	call-by-reference
• *function gets copy of parameter value* • *original parameter values are unmodified*		• *function accesses original value of parameter* • *original parameter values might be modified*

C only support call-by-value and fakes call-by-references via parameters that are pointer types. Modification is done at the object the pointer points to but not at the original parameter value (i.e., the pointer parameter itself). Actually, that's pass-by-address.

Function Definition

```
specifiers return-type function-name ( argument-declaration )
{
    compound-statement
}
```

Functions in C are defined by assigning and identifier (function-name) and list of arguments with a function body (compound-statement)

- functions definition must be in file scope

`compound-statement`

- a sequence of declarations and statements

Basics (2)

Example (inspired by *The C Programming Language*)
A program that prints its input if this input contains a user-defined substring `needle`.

Split into two functions

- `readline(buffer)`
  - returns false iff input is „`:exit`“
    - prints input character (`$`)
    - gets an input line from std input & stores into string buffer
    - removed newline character
    - compares buffer with „`:exit`“
- `index_of(str, needle)`
  - returns index of needle in str if contained, -1 otherwise
    1. scans str character by character
    2. if character equals first one in needle
       I. test if all subsequent characters of str and needle are equal as well
       II. if so, return index of character in (2)

```
SELECT … FROM
WHERE column LIKE '%needle%';
```

Standard library function

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX_LINE_LENGTH 1024

static bool readline(char buffer[]);
static int  index_of(char str[], char needle[]);

int main(void)
{
    char instr[MAX_LINE_LENGTH], needle[MAX_LINE_LENGTH];
    printf("This echos input that contains a user-defined needle\n"
           "Type :exit to terminate.\nEnter search string:\n");
    if (readline(needle)) {
        printf("Enter strings\n");
        register bool matches;
        while (readline(instr)) {
            matches = (index_of(instr, needle) >= 0);
            printf("%s", matches ? instr : "");
            printf("%s", matches ? "\n" : "");
        }
    }
    return EXIT_SUCCESS;
}

static bool readline(char buffer[])
{
    printf("$ ");
    fgets(buffer, MAX_LINE_LENGTH, stdin);
    buffer[strcspn(buffer, "\n")] = 0;
    return (strcmp(buffer, „:exit\n") != 0);
}

static int index_of(char str[], char needle[])
{
    register char str_c, str_scan_c, needle_c;
    register int  match_idx;
    register bool match_found;

    for (match_idx = 0, match_found = false;
         str_c = str[match_idx], str_c != '\0' && !match_found;
         match_idx++) {
        for (unsigned j = 0; needle_c = needle[j], str_scan_c = str[match_idx + j],
             str_c == needle[0] && needle_c != '\0' && str_scan_c != '\0' && str_scan_c == needle_c;
             j++) {
            match_found |= (needle[j + 1] == '\0');
        }
    }
    return match_found ? match_idx : -1;
}
```

Function Definition vs Declaration

Function

Declaration

Declaration introduces a **particular identifier to *name* a function**. Optionally, the parameter list can also be stated.

Technically: Provide basic information on a *symbol*: its type and its name. Reuse the same symbol/function across multiple files.

Informal: Stating that there will be a function *f*, with return type *t* and with the parameters $p_1, p_2, \ldots, p_n$ but it is not stated how this function is actually implemented.

Effect: Enables the *compiler* to **understand code** that is not completely defined (i.e., **implementation details are left out**)

Example

```
static bool readline(char buffer[]);
```

Function

Definition

Definition is assignment of a **function body** with the **function name** and **parameter list**.

Technically: Provides the implementation behind a function name.

Effect: Enables the *linker* to **link** references to the actual implementation.

Example

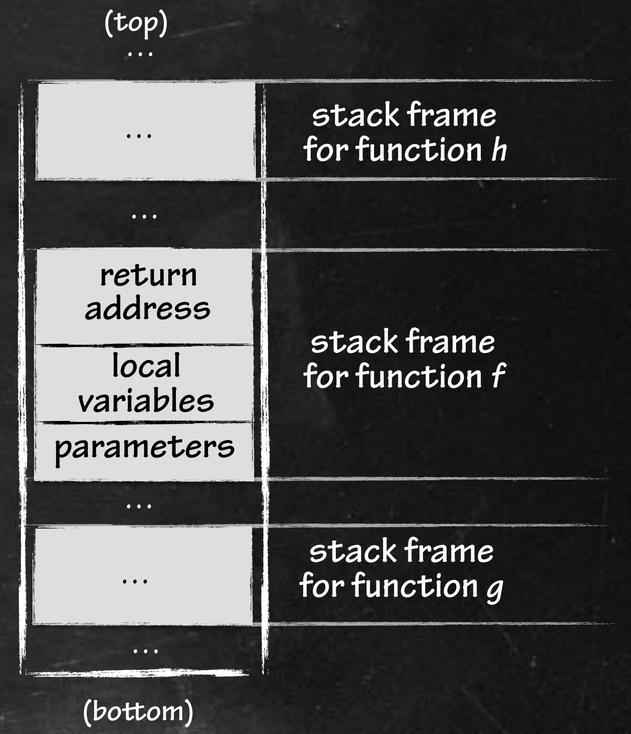
```
static bool readline(char buffer[])
{
    printf("$ ");
    fgets(buffer, MAX_LINE_LENGTH, stdin);
    buffer[strcspn(buffer, "\n")] = 0;
    return (strcmp(buffer, ":exit\n") != 0);
}
```

a definition is enough for the compiler to know everything needed. Hence, declarations are optional if definitions are exposed. However, modularized programs typically rely heavily on declarations.

if *declaration* is done in *header files* (*.h) and *definition* is done in *implementation files* (*.c), then this enables to **re-use compilation units** (if they are not out-of-date)

Function Calling Internals

Call Stack

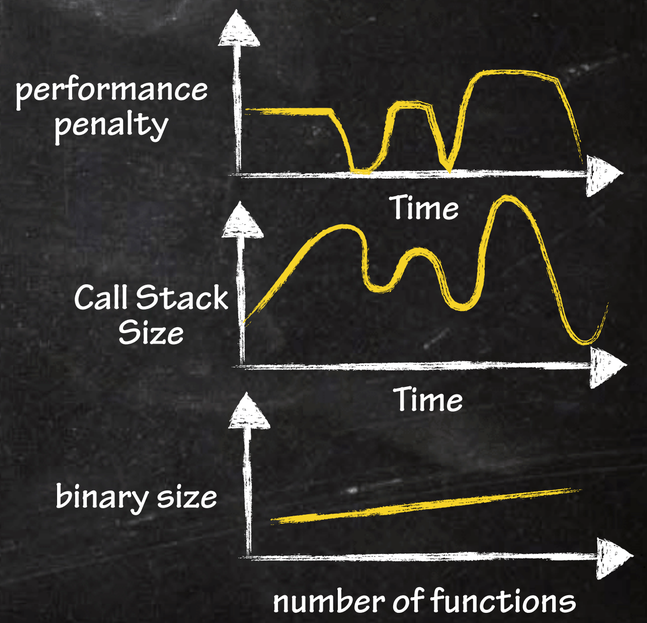


Call stack: a call stack in the machine is a stack that **stores information on active subroutines** (functions).

- Manages jumping into and returning from subroutines
- Provides parameter values for subroutines
- Provides local variables for subroutines (independent of where these variable occur in the subroutine)

Function call overhead: calling a function comes with overhead

- Parameter setup for the stack frame
- Setting/getting the return value of the function
- Function local memory and CPU register value management
- Compiler is unable to optimize between functions
- Non-optimal caching due to non linearity in execution code



Inlined Functions

Inlining a function is an approach to avoid function call overhead by replacing the function call by the function body („inlining“) where the function is called.

C provides two way for function inlining

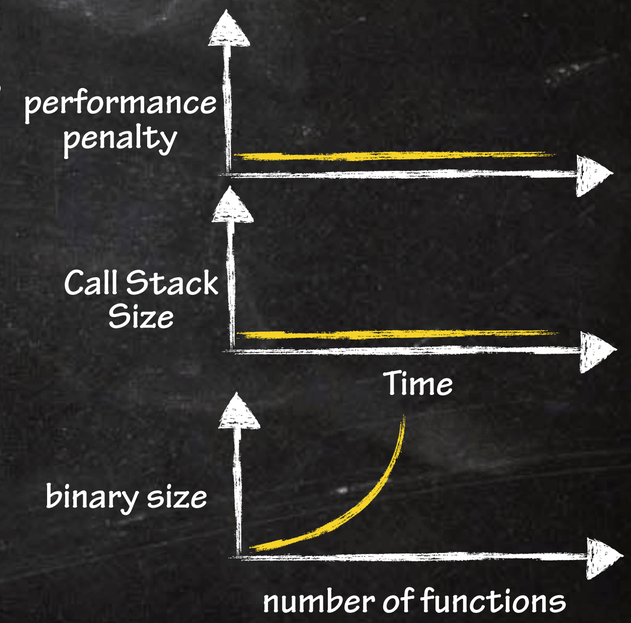
1. *Hint* the compiler that the function is intended for inlining by marking function by the `inline` keyword
   - not guaranteed that inlining is performed
   - declaring `static inline` functions has no restrictions
     - others do: e.g, non-static inline function cannot access static file scope variables
2. Write function as C preprocessor macros
   - guaranteed that inlining is performed
   - things get more complicated (see later in this chapter)

Benefit of inlining: avoid function call overhead and (notably) increase the execution performance.
Drawback of inlining: when inlined function is used at *n* places in the program, the function body is „copied“ to these *n* places. Result is a bloated binary size!

```c
inline int plus(int x, int y)
{
    return x + y;
}

/* ... */
int a = 5, b = 42;
int r = plus(23, plus(a, b));
/* ... */
```


```c
/* ... */
int a = 5, b = 42;
int r = (23 + (a + b);
/* ... */
```



Variadic Arguments

```
specifiers return-type function-name ( argument-declaration , ... )
{

}
```

A variadic function is a function that can be called with different number of arguments.

- indicated by a parameter of the form `...`
  - must be the last parameter
  - must follow at least one named parameter
- within function body, values of these arguments are accessed via `<stdarg.h>` library functionality:
  - `va_start` starts access to arguments
  - `va_arg` access next argument
  - `va_copy` copies the current argument
  - `va_end` ends traversal of arguments

Example

```c
#include <stdio.h>

float int_mean(int count, ...)
{
    float result = 0;
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; i++) {
        result += va_arg(args, int);
    }
    va_end(args);
    return (result / count);
}

int main(void)
{
    printf("%f", int_mean(3, -11, 0, 10));
    return 0;
}
```

taken from: http://en.cppreference.com/w/c/language/variadic

Notes on Function Naming, and Style

more importantly: it must be consistent in the code base!

function naming

cases

- `MyFunction` (pascal case)
- `myFunction` (camel case)
- `my_function` (snake case) ✓
- `MYFUNCTION`
- `MY_FUNCTION`

prefixes and suffixes

(for external linkage functions)

`prefixnamesuffix`

- (none)
- modul name, e.g. gecko_ ✓
- (others)
- (none) ✓
- (others)

formatting

```
void
gecko_my_function(
int arg1,
int arg2)
{
    /* ... */
}
```

```
void
gecko_my_function(int arg1, int arg2)
{
    /* ... */
}
```

```
void gecko_my_function(int arg1, int arg2) {
    /* ... */
}
```

```
void gecko_my_function(int arg1, int arg2)
{
    /* ... */
}
```

...

✓ convention in this lecture

Notes on Function Naming, and Style (2)

more importantly: it must be consistent in the code base!

Input / Output Parameters

Text-book version of C function call result is typically done via the `return` statement.

```
result_t foo(/* ... */)
{
    /* ... */
    return result;
}
```

However, more complex functions need often **returning more than one result** and may use the `return` statement to express status states (such as success or failure)

```
status_e foo(inout1, inout2,... , inoutn, in1, in2,... ,inm)
{
    /* ... */
    return status_okay;
}
```

Typical convention: pass parameters $inout_1$, $inout_2$,... , $inout_n$ that are intended for results (e.g., creation of certain values or modification) by call-by-reference/pass-by-address on the left of the function parameter list. Other call-by-reference/pass-by-address parameters that are not modified are stored at the right and are annotated with const.

Notes on Function Naming, and Style (3)

more importantly: it must be consistent in the code base!

Input / Output Parameters

Example