C Syntax and the GCC Compiler

(what does mean: '`void (*(*f[])())()`' ?)

Emmanuel Fleury
<emmanuel.fleury@labri.fr>

LaBRI, Université de Bordeaux, France

September 1, 2017

Overview

Overview

C Programming Language

C Language:

- Started in 1969 by Dennis Ritchie in the Bell Labs.
- The C language is **imperative** (describes computation in terms of statements that change a program state).
- Simplicity and portability.
- Not so efficient for modular and extensible programs.

C Language Standards:

- **1978**: K&R (Kernighan & Ritchie)
- **1989**: ANSI or C89
- **1999**: **ISO/IEC 9899:1999** or **C99** (most used now)
- **2011**: ISO/IEC 9899:2011 or C11

Primitive Data Types

Integral Types

Types

- `char` (1 byte)
- `short int` (2 bytes)
- `int` (2/4 bytes, depends on CPU)
- `long int` (4 bytes)
- `long long int` (8 bytes)

Type Qualifiers

- `signed`
- `unsigned`

Module `inttypes.h` (C99)

Syntax

- <type>`_t` (signed: `int32_t`)
- `u`<type>`_t` (unsigned: `uint32_t`)

Types

- `int8_t`/`uint8_t` (1 byte)
- `int16_t`/`uint16_t` (2 bytes)
- `int32_t`/`uint32_t` (4 bytes)
- `int64_t`/`uint64_t` (8 bytes)
- `uintptr_t` (hold a pointer)

Floating point Types

- `float` (single-precision)
- `double` (double precision)
- `long double` (double extended precision)

Module `stdbool.h` (C99)

- `bool` (boolean type)
- `true` (1)
- `false` (0)

Primitive Data Types (part two)

Enumerated Types

Defines a list of **integer constants** that can be called by their name:

```
enum colors {RED, BLUE=5};
```

Type definition

Allow to define its own custom types:

```
typedef struct {int x; int y;} myt_t;
typedef enum {MONDAY, SUNDAY} days_t;
```

Structures

Glue **non-overlapping** data fields:

```
struct mystructure {
    int   x;
    float y;
    char *z;
};
```

Accessing 'x' with `mstr.x`.

Unions

Glue **overlapping** data fields:

```
union myunion {
    int   x;
    float y;
    char *z;
};
```

Accessing 'x' with `myuni.x`.

Void

The `void` type means that the variable has no value at compile time.

Variable Qualifiers

`auto`

Defines a local variable as having a local lifetime. Almost never used explicitly as this is the default.

`register`

Tells the compiler to store the variable being declared in a CPU register.

`static`

Preserves variable value to survive after its scope ends all over the module but no more.

`const` (C89)

Makes variable value or pointer parameter unmodifiable.

`extern`

Indicates that the variable is defined in a separate source code module.

`volatile` (C89)

Indicates that a variable can be changed by a background routine.

Arrays

Declaration

```
float array1[5][6];
int array2[5] = { 10, 32, 34, 41, 35 };
int matrix[3][3] = {
    { 1, 0, 0 },
    { 0, 1, 0 },
    { 0, 0, 1 }
};
```

Usage

```
i = matrix[0][2] + 1;
```

Good to know...

- Array index start at zero and end at $n-1$.
- In C99 we can define parametrized arrays within a block.

Strings

Declaration

```
char string1[128];
char string2[5] = { 'A', 'B', 'c', 'e', '\0' };
char string3[];
```

Usage

```
printf ("The␣string␣is␣%s\n", string2);
```

Good to know...

- Strings are one dimensional vectors of `char`.
- All strings **MUST** be ended by a '\0' character.
- In C, strings are the weakest point of your program.
  Be extremely careful with it.

Pointers & Addresses

Address, Indirection

Get the address of variable 'i': &i
Get the content of pointer 'ptr': *ptr

Pointer arithmetic

Point to the next memory cell of size 'sizeof(ptr)': ptr+1
Number of cells between 'ptr1' and 'ptr2': ptr2-ptr1

Exercises

```
int i, *ptr;
i = 0;
ptr = i;
*ptr = i;
ptr = &i;
*(ptr + 1) = i;
```

Control Structures

if... else...

```
if ((i <= 10) && (j > 5))
   i += 1;
else
   j = 6;
```

switch... case...

```
switch (color) {
  case 0:
       rgb.red += 1;
       break;
  case 1:
       rgb.blue += 5;
  /* FALLTRHOUGH */
  case 2:
       rgb.green += 10;
       break;
  default:
       puts("Error␣!");
}
```

while...

```
while (!i)
  i += 1;
```

do... while...

```
do
{
  i += 1;
} while (!i);
```

for... (C99)

```
for (int i=0; i<MAX_ITER; i++)
{
  array[i][i] = 1;
}
```

Functions

Declaration

You need to define a **return type**, an **identificator** and a list of **arguments** and then the set of instructions:

```
double
norm (double x, double y) {
  return sqrt (x * x + y * y);
}
```

Value passing arguments

Arguments are passed by value !
They can't be modified by functions.
You must pass arguments through pointers when you need to modify it.

```
void swap(int *i, int *j);
```

`inline` (C99) (Qualifier)

The compiler will substitute the code of the function into its caller.

`return` (Func)

Leave the current function and returns to the caller function.

`exit` (Func)

Stop current process at once, no other function can be called after.

`static` (Qualifier)

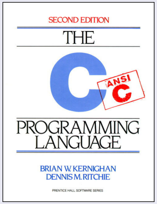
Preserves the function to be accessible all over the module and only the module.

`extern` (Qualifier)

Indicates that the function is defined in a separate source code module.

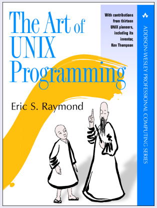
Further Readings

The C Programming Language



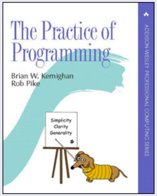
Brian W. Kernighan, Dennis M. Ritchie, Prentice Hall Software Series, 1988.

The Art of UNIX Programming



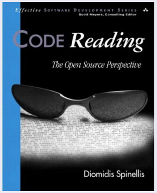
Eric S. Raymond, Addison-Wesley Professional, 2003.

The Practice of Programming



Brian W. Kernighan, Rob Pike, Addison-Wesley Professional, 1999.

Code Reading: The Open-Source Perspective



Diomidis Spinellis, Addison-Wesley Professional, 2003.

Overview

GCC: GNU Compiler Collection

Started in 1985 (first release in 1987 as the 'GNU C Compiler'), this project intended to produce a **high quality compiler** within the Free Software community.

Nowadays, GCC 4.x is multi-language able and can compile:

- **C** (`gcc`),
- **C++** (`g++`),
- **Fortran** (`gfortran`),
- **Pascal** (`gpc`),
- **Objective-C** (`gobjc`),
- **Java** (`gcj`),
- **Ada** (`gnat`),
- and few others.

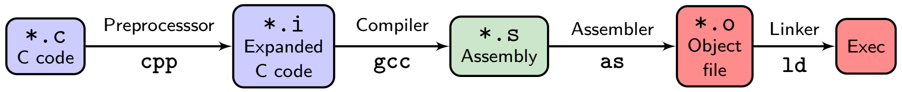
Moreover, GCC can target many architectures such as Alpha, ARM, PowerPC, IA-32, IA-64, MIPS, . . . and many others.

GCC is probably the most used compiler in the UNIX world. . .

Associated with it, we can find debugging and profiling tools such as `gdb` (debugger), `gprof` (profiler), `gcov` (covering tool), valgrind (memory checker), . . .

First rule of programming: KNOW YOUR TOOLS !!!

Compiling Chain



- **Preprocessor**: Expand macros embedded in the code.

```
gcc -E hello.c > hello.i
```

- **Compiler**: Translate C code into assembly code.

```
gcc -S -o hello.s hello.c
```

- **Assembler**: Translate human readable assembly code into machine readable code.

```
gcc -c hello.c
```

- **Linker**: Build the executable and compute offsets.

Preprocessor (`cpp`)

All preprocessor directives starts with a #.

Unconditional directives:

- `#include <file.h>`: Inserts a header file which is in the include paths;
- `#include "file.h"`: Inserts a header file which is in the same directory;
- `#define MACRO definition`: Defines a preprocessor macro;
- `#undef MACRO`: Undefines a preprocessor macro.

Conditional directives:

- `#ifdef MACRO`: If `MACRO` is defined;
- `#ifndef` If this macro is not defined;
- `#if` Test if a compile time condition is true;
- `#else`: The alternative for `#if`;
- `#elif`: `#else` an `#if` in one statement;
- `#endif`: End preprocessor conditional.

Compiling Options

- `-g[level]`: Set debug level 0-3 (default: 2)
- `-o exe`: Set executable file name (default: `a.out`).
- `-I include_path`: Add extra include paths.
- `-L library_path`: Add extra library paths.
- `-lmylibrary`: Link the executable with `libmylibrary.so`.
- `-D`**`FLAG`**: Set the variable `FLAG` as defined for the preprocessor.
- `-O<level>`: Set the optimization level:
  - `0`: No optimization
  - `1`: Basic optimization without expansion.
  - `2`: All optimizations that do not involve a space-speed tradeoff.
  - `3`: Full optimization (dangerous).
  - `s`: Optimize size (for embedded software).
- `-Wall -Wextra`: Set the warning level up to the maximum.
- `-std=c11`: Set the compiler standard to C11.

Usual Compilation Process

Producing Object files

```
gcc -std=c11 -Wall -Wextra -O2 -I../include -c main.c
gcc -std=c11 -Wall -Wextra -O2 -I../include -c io.c
gcc -std=c11 -Wall -Wextra -O2 -I../include -c sets.c
```

Building Static Libraries

```
ar rcs libsets.a sets.o
```

Building Dynamic Libraries

One of the two should work (PIC = Position-Independant Code):

```
gcc -std=c11 -Wall -Wextra -O2 -c -fpic sets.c
gcc -std=c11 -Wall -Wextra -O2 -c -fPIC sets.c
```

Then: `gcc -shared -o libsets.so sets.o`

Linking Object files

```
gcc -std=c11 -Wall -Wextra -O2 -L../libs -o mysoft main.o io.o -lsets
```