AIMS Embedded Systems Programming
MT 2016
The Basics of the C Programming Language

Daniel Kroening

University of Oxford, Computer Science Department

Version 1.0, 2014
Outline

Motivation
C Compilers
Tokens
Basic Data Types
Declartions and Definitions
Statements
Expressions
Pointers
Data Structures
Compilation Units and Modularisation
Memory-mapped I/O, Interrupts, Threads
The C Programming Language

- Origin: K&R C

- ANSI X3.159-1989 (C89)

- 1990: ISO/IEC 9899 (same as C89), followed by C90, C95, C99, C11

- Gazillions of flavours and vendor-specific variants and extensions
Why do we care?

![TIOBE Programming Community Index](source: www.tiobe.com)

It’s the most popular programming language
Why do we care?

- Near-monopoly for embedded systems
  - Memory-constrained
  - Real-time requirements
  - Focus on I/O
  - Focus on power consumption

- Compilers exist for basically any target
- Many targets only have C compilers

→ It’s an intermediate representation for many other tools (e.g., ADA, Simulink, Labview)
C Compilers

- C is a compiled language (vs. interpreted languages)

- The compiler reads the C program and generates an executable
  (more detail later)

- The executable contains the machine code for the target CPU
Compiling C

C Program (text file) → Pre-processor → Tokenizer → Parser → Frontend

CFG Gen. → SSA Gen. → Optimizer → Code Generator → Backend

Executable
Assigning Meaning to Programs

- There are many programs that have proper syntax but without any meaning given by the standard

- The standard distinguishes
  - implementation-defined and
  - undefined behaviour.

- Both should be avoided
Implementation-defined Behaviour

Why? Say Java is more specific.

- C is designed to target a vast array of architectures

- Fixing too much means that execution on some targets will be inefficient

- Some implementation-defined or undefined behaviour is critical for performance on any reasonable target
Preprocessor

- Expands `#include`, `#define` and `#ifdef` directives

- `#ifdef` is mostly used for variant management

- Many library functions are really `#define` macros
Tokens

The C program is a sequence of characters, which are grouped into *tokens*:

- keyword
- identifier
- constant
- string-literal
- punctuator
Keywords

- `auto`, `break`, `case`, `char`, `const` and so on.

- case-sensitive
Constants

- integer constant:
  1               type int
  1u              type unsigned int
  10000000000000  type long long int
  0xffffffff     type unsigned int
  1lu             type unsigned long int
  1llu            type unsigned long long int

  The type is the smallest signed bigger or equal int that fits the number. If it’s hex or octal, include unsigned types.

- floating constant:
  1.23 = 123e-2   type double
  1.23l           type long double
  1.23f           type float
Constants

- enumeration constant:
  These are enum identifiers, and have type `int`

- character constant:
  `'a' = 97,  'ab' = 98 + 97 \cdot 256,  '\'xff' = -1`

These have type `int`!
Punctuators

- These are
  
  \[ \] ( ) { } . ->
  ++ -- & * + - ~ !
  / % << >> < > <= >= == != ^ | && ||
  ? : ; . .
  = *= /= %= += -= <<= >>= &= ^= |=
  , # ##

- The meaning depends on context; we will see this later.

- There are 6 aliases (digraphs), which we avoid.
Identifiers

- Almost anything that doesn’t start with a digit, and isn’t a keyword, constant or punctuator

- E.g., unicode-characters are ok

- case-sensitive
Basic Data Types: Integers

- _Bool: 0 or 1  use stdbool.h instead
- char
- signed char
- unsigned char
- short = signed short = signed short int
- unsigned short = unsigned short int
- signed int = signed = int
- unsigned int = unsigned
- signed long int = signed long = long int = long
- unsigned long int = unsigned long
- signed long long int = signed long long = long long int = long long
- unsigned long long int = unsigned long long
Basic Data Types: Integers

- Note that `char` is signed or unsigned, but is not the same type as `signed char` or `unsigned char`.

- Note that the range gets bigger, but not strictly.

- E.g., `long int` has the same range as `int` on many architectures.

- If you need some fixed number of bits, use `intN_t` and `uintN_t`, which are defined in `stdint.h`.

- Use `size_t` (`stddef.h`) for things such as string lengths.
Basic Data Types: Enumerated Types

```c
enum my_booleans { my_false, my_true };

enum months {
    January=1, February, March, ...
};
```

The identifier after the keyword `enum` is called the enum tag. Tags use a separate namespace, and are always preceded by the *aggregate key* in C.
Basic Data Types: Floating-Point

- **float**: IEEE single-precision floating-point
- **double**: IEEE double-precision floating-point
- **long double**: something bigger than double
- Some targets have `float _Complex`, `double _Complex`, `long double _Complex`
Basic Data Types

- **void**: nothing

Beyond the basic types, there are *derived types*:

- arrays
- structures
- unions
- pointers

We will see these later.
Type Qualifiers

- **const**: the object is read-only

- **volatile**: someone else may modify the object

- **restrict**: will cover later
Variable Declarations and Definitions

```c
int some_integer;

const int some_other_integer = 200;

int a, b, c;

long double some_important_value;

enum months begin_of_spring = May;
```
Function Declarations and Definitions

Functions should first be declared:

```c
void do_something(int a, int b);
```

And then defined:

```c
void do_something(int a, int b)
{
    some_other_function(a+1);
}
```

Many bugs will happen when forgetting the declaration, or when declaration and definition are inconsistent.
Variables in a Block Scope

```c
int a;

void do_something(int a) {
    int a=3;

    {
        char a=4;
    }

    printf("%d\n", a);
}

int main() {
    a=1;
    do_something(2);
}

Variables have *scope*: same name, different variable!
```
Storage Classes

- **extern**: external linkage (default)
- **static**: variables get stored in global data segment, functions/variables get internal linkage
- **_Thread_local**: one copy per thread
- **auto**: default, says nothing
- **register**: obsolete; the compiler decides these things now
void do_something(void)
{
    static int a = 1;

    a = a + 1;

    printf("%d\n", a);
}

int main()
{
    do_something();
    do_something();
}
Variables with static storage duration are *zero initialized*. Variables with with temporary storage duration have an *indeterminate* initial value.
Puzzle

...  

```c
void some_function()
{
    _Bool my_boolean;

    if(my_boolean)
        printf("It’s true :-)
    else
        printf("It’s false :-(

}
```

What’s wrong?
**Type Definitions**

```c
typedef unsigned int uint32_t;

typedef signed short wchar_t;

uint32_t some_32_bit_integer;

const uint32_t some_const_32_bit_integer;
```

These are short-hands, and not separate types. They can’t be modified, e.g., “signed uint32_t”, but qualifiers can be added.
Statements

Overview:

- Expression statements (this includes assignment)
- Iteration statements: `while`, `do while`, `for`
- Jump statement: `goto label;`
- Labeled statements: `label: ...`
- Compound statements:
  ```c
  { x=1; y=2; }
  ```
- Selection statements:
  ```c
  switch(...) { case ...: .... }
  if(...) ... else ...
  ```
Expressions

- Obvious things: +, − (binary and unary), /, and so on
- Shifts: << and >>
- Function applications: \( f(a_1, a_2, \ldots) \)
- Relations: <, ≥, =, !=, == and so on. Type of result is int.
- Boolean operators:
  \&\&, ||, !: these operate on 0/1
  &, |, ^: these operate bit-wise
- Ternary if-then-else: c?t:f
- Size of types and expressions:
  \texttt{sizeof} 20 vs. \texttt{sizeof(int)}

Watch out for operator precedence!
In particular, &, | and ^ bind very weakly.
Arithmetic Promotion

- Operands of arithmetic operators are *promoted* before evaluation.

- Everything is promoted to at least int, then more as needed until both sides have the same type.

- Danger: `unsigned int` is considered larger than `int`. Cast to larger signed type.
Puzzle

-1 < 0xffffffff?
You can write \((T)x\) for a type \(T\).

This converts the value of \(x\) to type \(T\).

This is a semantic conversion; the resulting bit-pattern may differ.

Conversion from floating-point to integer truncates, e.g., \((\text{int})1.9\) is 1.

Conversion to \_Bool maps non-zero values to 1.
Bit-vector Semantics

Arithmetic over unsigned integers is modulo $2^n$

- The standard in essence hard-wires the binary representation
- Programmers are allowed to rely on this; but it may be considered bad style or violate coding guidelines
Arithmetic over signed integers must remain in range $-2^{n-1} \ldots 2^{n-1} - 1$

- The standard explicitly warns that behaviour is undefined
- May crash, may produce odd numbers; this violates all relevant safety standards
- Many programmers ignore this anyway
- Remember arithmetic promotion rules
- Remember that char may be signed
Expressions with Assignment

- The basic assignment $x = y$ is an expression. Associativity is right-to-left, e.g., you can write $x = y = z$;

- But beware
  ```c
  int x, z;
  char y;
  
  x = y = z;
  ```
Expressions with Assignment

- $x += y$ is short for $x = x + y$
- But note that the expression $x$ is evaluated only once.

- Same for $-=, *=, \text{etc}$.

- $++x$ is $x += 1$.

- $x++$ is the value of $x$ before incrementing
char some_char;

while((some_char=getchar())!=EOF)
{
   /* do something with some_char */
}

Puzzle
Order of Evaluation

- Most evaluation orders are implementation-defined.

- `&&` and `||` are ordered left-to-right, and evaluation aborts early

- `c?t:f`: the expression `c` is evaluated first, and then according to outcome `t` or `f` only
Simultaneous Assignment

- var1 = var1++; obviously has an ordering problem

- But it also performs a *simultaneous assignment*, which is forbidden
Pointers

- Pointers hold the address of a variable or a function
- Syntax:
  ```
  int *p;
  ```
- Warning: The declaration splits up into a type specifier and a declarator. The * is part of the declarator.
- Thus,
  ```
  int *p, i, **z;
  ```
  is a pointer to an integer, an integer and a pointer to a pointer to an integer.
Pointers

- You get the address of something using the unary & operator. (Don’t confuse with binary & operator).
  ```
  int *p=&i;
  ```

- You get the object that is pointed to using the unary * operator. (Don’t confuse with binary * operator).
  ```
  *p=123;
  ```

- What is *p++?

Call-by-reference

Pointers let you define “output arguments” for functions.

```c
void some_func(int *x, int y) {
    *x=1;
    y=2;
}

void other_func()
{
    int a, b;
    some_func(&a, b);
}
```
Arrays

- These are for vectors or matrices of values

- Syntax:

  ```
  int some_array[10], some_2d_array[20][30];
  ```

- The suffix parses left-to-right, and binds weaker than the prefix.

- Get an element by writing `some_array[5]` and `some_2d_array[19][29]`

- Numbering of elements always begins with 0

- `int *a[10]` is an array of 10 pointers to int, `int (*a)[10]` is a pointer to an array of 10 ints.
Pointer Arithmetic

- This is a distinct feature of C/C++.
- `int array[10], *p;`  
  `p=array;`  
  `p=p+3;`
- `p` will point to the 4th element of the array.
- `p[i]` is in fact just `*(p+i)`
- Stay within the bounds *plus one* at the end
- There is no gap in multi-dimensional arrays!
- You can also compare pointers and compute the difference if they point into the same object.
Define a *compound type*:

```c
struct some_tag {
    int first_member, second_member;
    char third_member;
};
```

```c
struct some_tag some_variable;
```

Recall that tags have their own name space

Members are accessed with the “dot”:

```c
some_variable.third_member
```

May be nested, also with arrays
Linked Lists

- These can refer to themselves!

```c
struct some_tag {
    int some_data;
    struct some_tag *next;
};

struct some_tag *list_head;
```

- Use `malloc(sizeof(struct some_tag))` to produce a new list element
struct some_tag *new_list()
{
    struct some_tag *head, *tail;

    head=tail=malloc(sizeof(struct some_tag));

    for(int i=1; i<10; i++)
    {
        tail->next=malloc(sizeof(struct some_tag));
        tail=tail->next;
    }

    tail->next=NULL;
    return head;
}
void traverse_list(struct some_tag *list) {
    while (list != NULL) {
        /* do something with the list element */
        list = list->next;
    }
}
Compilation Units

- You can have multiple source files, which are compiled separately.

- Only symbols with external linkage are exported to other units.

- This means that the others are invisible!
Modularisation

- Large software project suffer from name clashes

- You want access control to retain maintainability

- Use internal linkage to make internals of small libraries inaccessible

- Expose only opaque types to the outside
Interrupts

▶ Interrupts are hardware-generated diversions of the control-flow to an **Interrupt Service Routine**

▶ There is some abuse of the interrupt mechanism to do calls to an OS; this is distracting.

▶ Most relevant for us to react in real time to outside stimulus

▶ There could be a low-priority tasks that is interrupted

▶ Get an RTOS (e.g., \(\mu C\) OS) to set up the interrupt for you

▶ Some people layer this (using interrupts with priorities)
Threads

- Essential to make use of multi-core CPUs

- Useful in a real-time context even for just one core (this is like an interrupt that just “sleeps”)

- Many APIs; e.g., POSIX pthread on the Mac

- The trick is communication between threads

- Locks acceptable for low-performance jobs, but inacceptable for high-end and problematic for real-time
Threads, Volatile and Clobbers

There are two options to write concurrent code:

1. The `volatile` type qualifier tells the compiler that some other thread might tweak some data

2. The alternative are compiler-specific memory clobbers, which typically sit in the lock/unlock functions

Really look for a library for concurrent data structures