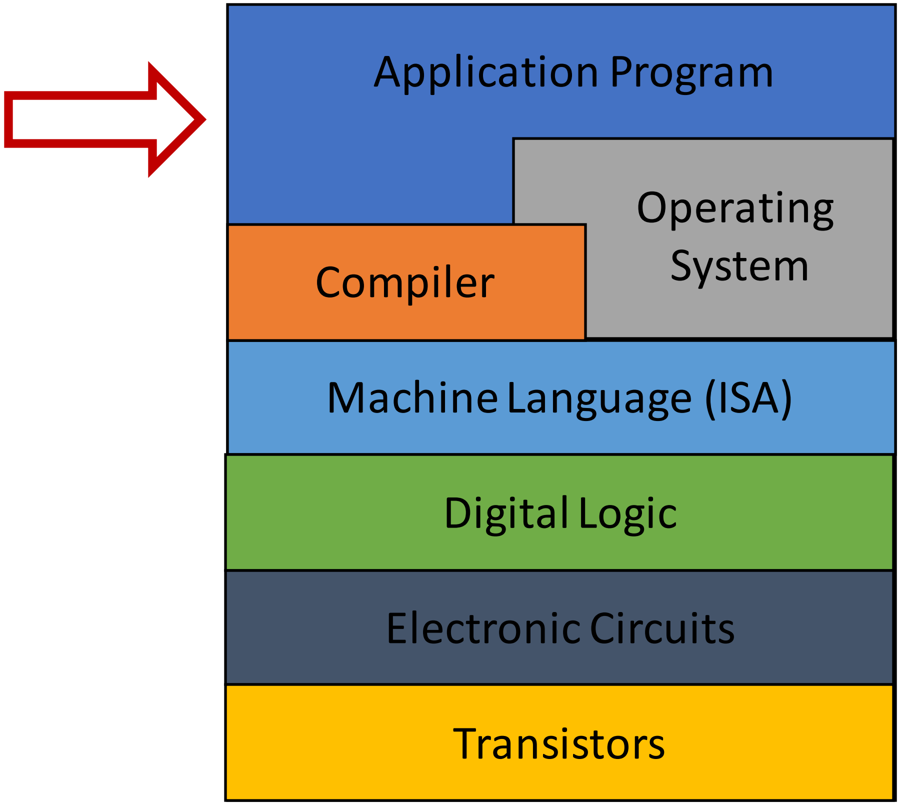
C Programming Refresher

EECS388 Fall 2022

Lecture notes based in part on slides created by Alex Fosdick and Heechul Yun

Context



Application Program

Operating System

Compiler

Machine Language (ISA)

Digital Logic

Electronic Circuits

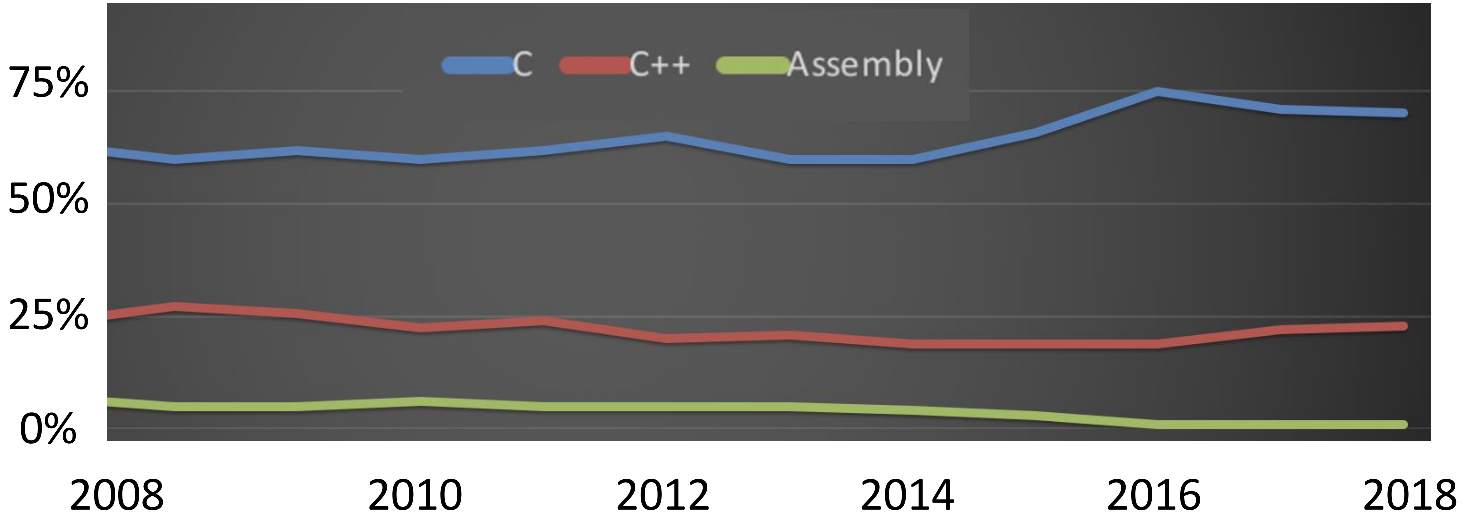
Transistors

Embedded Software Market Share

C: 70%

C++: 25%

Assembly, Java, etc: 5%



https://embeddedgurus.com/barr-code/2018/02/c-the-immortal-programming-language/

Embedded systems primarily uses C programming. But WHY?

- High-level enough that programmer doesn’t need to know every details
- Low level enough for efficient memory management, direct memory mapped hardware and IO control, and optimized execution/timing

Declaring Variables

<type qualifier(s)> <type modifier> <data-type> <variable name> = <initial value>;

Example:

int var;

const unsigned int var = 7;

var = 13

<type qualifier(s)> <type modifier> **<data-type>** <variable name> = <initial value>;

Data Types

Describe a specific variable

- Integer: e.g., int, char
- Floating point: e.g., float, double
- Void
    - Function return type. Function only has side effects
    - Universal pointer type (all other types are subtypes)
- Enumerated
- Derived: e.g., arrays, pointers

```
void func () {
        printf(“Hi EECS388!”);
}
```

<type qualifier(s)> **<type modifier>** <data-type> <variable name> = <initial value>;

Type Modifiers

Increase the size of data types or change their properties

- Short
- Long
- Unsigned
- Signed

Data type	Storage	Range
char	8 bits	[-128,+127]
unsigned char	8 bits	[0, 255]
short int	16 bits	[-32768,+32767]
unsigned short int	16 bits	[0, 65535]
int	16 or 32 bits	$[-2^{15}, 2^{15}-1]$ or $[-2^{31}, 2^{31}-1]$
long int	32 or 64 bits	$[-2^{31}, 2^{31}-1]$ or $[-2^{63}, 2^{63}-1]$
long long int	64 bits	$[-2^{63}, 2^{63}-1]$

<type qualifier(s)> <type modifier> <data-type> <variable name> = <initial value>;

Type Qualifiers

Provide specific instructions to the compiler on how the variable should be managed

- Const
  - The value of the variable cannot be modified
- Volatile
  - Re-load register from memory on every access
- Restrict

Data Storage in Memory

$$\overbrace{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}^{\text{8 bits == 1 byte}}$$

Most
significant bit
(MSB)

Least
significant bit
(LSB)

Number Systems

- Decimal (base 10)
  - Symbols: 0,1,…,9
  - E.g., $123_{10} = 1x10^2 + 2x10^1 + 3x10^0$
- Binary (base 2)
  - Symbols: 0,1
  - E.g., 1011_2 = **0b1011** = $1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

*Assuming unsigned number

- Hexadecimal (base 16)
  - Symbols: 0,1,…,9,A,B,…,F
  - E.g., 123_{16} = **0x123** = $1x16^2 + 2x16^1 + 3x16^0$

*Assuming unsigned number

Number Systems

Decimal	Hexadecimal	Binary
0	0x0	0b0
2		
9		
	0xA	
	0xF	
	0x1F	
		0b1000 0000
		0b1000 0011
		0b1000 0000 0000 0000

Number Systems

Decimal	Hexadecimal	Binary
0	0x0	0b0
2	0x2	0b10
9	0x9	0b1001
10	0xA	0b1010
15	0xF	0b1111
31	0x1F	0b1 1111
128	0x80	0b1000 0000
131	0x83	0b1000 0011
32768	0x8000	0b1000 0000 0000 0000

Categories of Numbers

- Unsigned
- Signed
- Fractional

Unsigned Numbers

- n-bit binary number:

$$\underset{\substack{\uparrow \\ \text{MSB}}}{b_{n-1}}\ b_{n-2}\ \dots\ b_1\ \underset{\substack{\uparrow \\ \text{LSB}}}{b_0}$$

- Decimal value:

$$b_{n-1} * 2^{n-1} + b_{n-2} * 2^{n-2} + \dots + b_1 * 2^1 + b_0 * 2^0$$

- Example:

$0b10011 = 2^4 + 2^1 + 2^0 = 19$ (d)

Signed – 2’s Complement Numbers

- n-bit binary number:

$$\underset{\substack{\uparrow \\ \text{MSB}}}{b_{n-1}}\ b_{n-2}\ \ldots\ b_1\ \underset{\substack{\uparrow \\ \text{LSB}}}{b_0}$$

- Decimal value:

$$- (b_{n-1} * 2^{n-1}) + b_{n-2} * 2^{n-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$$

- Example:

$0b10011 = - (2^4) + 2^1 + 2^0 = -13$ (d)

Fractional Numbers: Float and double

Float

- IEEE 754 single precision floating point numbers
- 1-bit sign, 8-bits exponent, 23-bits fraction
- 6 significant decimal digits of precision

Double

- 1-bit sign, 11-bits exponent, 52-bits fraction
- 15-17 significant decimal digits of precision

Operators

Used to manipulate data

- Logical: ||, &&, !
- Bitwise: <<, >>, |, &, ^, ~
- Arithmetic: +, -, /, *, ++, --, %
- Relational: <, <=, >, =>, ==, !=

Logical Operators

- || = logical OR
- && = logical AND
- ! = logical NOT

Boolean condition:
True: non-zero condition
False: zero condition

```
if (cond0 || cond1)
if (cond0 && cond1)
if (!cond0)
```

How about signed types??

Bitwise Operators

Zero shifts in for “unsigned” types

- << = left shift
- >> = right shift
- | = bitwise OR
- & = bitwise AND
- ^ = bitwise EXOR
- ~ = bitwise one’s complement

```
  10011011 << 2
= 01101100
```

```
  10011011 >> 2
= 00100110
```

```
  10011011 &
  01101100
= 00001000
```

```
  10011011 ^
  01101100
= 11110111
```

```
~ 10011011
= 01100100
```

Arithmetic Operators

Perform math operations

- + = add
- - = subtract
- / = divide
- * = multiply
- ++ = increment
- -- = decrement
- % = modulus (remainder)

var++ → var = var + 1
var-- → var = var - 1

10 % 2 = 0
10 % 3 = 1

Increment and decrement (++, --)

var++

- First use the value and then increase it by one

++var

- Increment the value and then use it

```
a = 1; b = 2; c = 3;
x = a-- + b++ - ++c;
x →  -1
```

Arithmetic and bitwise operators can be combined with an assignment for simplified expressions

- var0 += var1; → var0 = var0 + var1;
- var0 >>= 5; → var0 = var0 >> 5;

Relational Operators

Used for Boolean expressions in conditional blocks

- <   = less than
- <=   = less than or equal
- >   = greater than
- =>   = greater than or equal
- ==   = equal
- !=   = not equal

Program Flow Control

```
if (condition) {
  // code
}

if (condition) {
  // code
} else {
  // code
}

if (condition) {
  // code
} else if (condition) {
  // code
} else {
  // code
}
```

```
switch (expression) {
  case const-exp1:
    // code
    break;
  case const-exp2:
    // code
    break;
  …
  default:
    // code
    break;
}
```

Loops

```
while (condition) {
  // code
}

do {
  // code
} while (condition);

for (init; condition; expression) {
  // code
}
```

Break and Continue

```
while ( condition ) {

  // code 0
  if ( condition1 ) {
     break;
  }

  // code 1
  if ( condition2 ) {
     continue;
  }

  // code 2
}

// code 3
```

Exit while loop and start executing “code 3”

Skip this iteration and start executing the next iteration which can be “code 0” or “code 3”

Functions

main.c

```
#include <stdio.h>

int add(int a, int b);

void main()
{
   int c = add(1, 1);
   printf("%d\n", c);
}
```

mylib.c

```
int add(int a, int b)
{
  return a + b;
}
```

Function definition

Function declaration

func_type func_name (param_type1 param_name1, …)

- Header files (.h files) contain public function declaration
- Implementation files (.c files) contain private function declaration and function definitions

Pointers

Derived data types that hold addresses

int * ptr;

Pointer declaration operator

int var = 10;

ptr = & var;

“Address-of” operator

*ptr = 20;

Dereference operator

var → 20;

Passing by Value vs. Pointer

Sending a copy or pointer of the variable to the function

```
void add_val(int a) {
   ++a;
}

void main()
{
   int var = 10;
   add_val(var);
   printf(“%d\n”, var);
}
```

```
void add_point(int *a) {
   ++*a;
}

void main()
{
   int var = 10;
   add_point(&var);
   printf(“%d\n”, var);
}
```

Memory Addresses

1 byte

256-bytes,
byte addressable
memory

Address	**data**
0x00	0x00
0x01	0xF1
0x02	0x11
…	…
0xFE	0x20
0xFF	0x34

Maximum addressable memory?

- Depends on the CPU architecture and platform

Pointers and Memory Addresses

```
1: int * ptr;
2: int var = 0x0A;
3: ptr = & var;
4: *ptr = 0xF5;
```

Address	data
0x00	0xFE
0x01	0xE1
0x02	0x1C

Pointers and Memory Addresses

```
1: int * ptr;
2: int var = 0x0A;
3: ptr = & var;
4: *ptr = 0xF5;
```

①

Address	data
0x00 (**ptr**)	0xFE
0x01	0xE1
0x02	0x1C

int *ptr

Pointers and Memory Addresses

```
1: int * ptr;
2: int var = 0x0A;
3: ptr = & var;
4: *ptr = 0xF5;
```

①

Address	data
0x00 (**ptr**)	0xFE
0x01	0xE1
0x02	0x1C

int *ptr

②

Address	data
0x00 (ptr)	0xFE
0x01 (**var**)	**0x0A**
0x02	0x1C

int var = 0x0A

Pointers and Memory Addresses

```
1: int * ptr;
2: int var = 0x0A;
3: ptr = & var;
4: *ptr = 0xF5;
```

①

Address	data
0x00 (**ptr**)	0xFE
0x01	0xE1
0x02	0x1C

int *ptr

Variable ptr holds the address of var

②

Address	data
0x00 (ptr)	0xFE
0x01 (**var**)	**0x0A**
0x02	0x1C

int var = 0x0A

③

Address	data
0x00 (ptr)	0xFE
0x01 (var)	0x0A
0x02	0x1C

ptr = &var

Pointers and Memory Addresses

```
1: int * ptr;
2: int var = 0x0A;
3: ptr = & var;
4: *ptr = 0xF5;
```

①

Address	data
0x00 (**ptr**)	0xFE
0x01	0xE1
0x02	0x1C

int *ptr

Variable ptr holds the address of var

②

Address	data
0x00 (ptr)	0xFE
0x01 (**var**)	**0x0A**
0x02	0x1C

int var = 0x0A

③

Address	data
0x00 (ptr)	**0x01**
0x01 (var)	0x0A
0x02	0x1C

ptr = &var

④

Address	data
0x00 (ptr)	0x01
0x01 (var)	**0xF5**
0x02	0x1C

*ptr = 0xF5

Recap

- Why C?
- C
  - Declaring variables
  - Number systems
  - Operators
  - Flow control
  - Functions
  - Pointers