Coding in C
Basics, Basic Types and Operators

Antonio Barbalace
antonio.barbalace@ed.ac.uk

Basics

- Comments
- Curly Braces {}
- Variables

Comments

Comments in C are: enclosed by:

- delimited by slash/star pairs, possibly across multiple lines

```
/* this is a comment */
/* this is also
a comment */
```

- or started by two slashes and extending to the end of the line

```
// comment until the line end
// this is a comment
   but this is not a line of comment
```

If you like **javadoc**, C has something similar called **doxygen**.

Curly Braces {}

- C uses curly braces { } to group multiple statements together
- Statements execute in order
- In C variables may be declared within the body of a function, but must follow a '{'
- C99 standard permits mixed declaration and code
- Curly braces imply variables scope

Variables

Variables scope

- Global
- Local

- Basic types
  - fixed point arithmetic
  - floating point arithmetic
- Arrays
- User Defined
- Pointers

Variables scope

```
global_variable1
global_variable2
{
  local_variable1
  {
     local_variable2
    {
      local_variable3
      /* access to global_variable1
   global_variable2 local_variable1
   local_variable2 local_variable3
   */
    }
    /* access to global_variable1
   global_variable2 local_variable1
   local_variable2 */
  }
  /* access to global_variable1
   global_variable2 local_variable1
   */
}
```

Basic Types

- Fixed Point Arithmetic
- Floating Point Arithmetic
- Declaring a variable
- Declaring a constant
- #define

Basic Types

Fixed Point Arithmetic

- `char` ASCII character (at least 8 bits). As a practical matter `char` is always a byte (8bits). `char` is also the "smallest addressable unit"
- `short` [`int`] small integer (at least 16bits)
- `int` an integer (at least 16bits): defined to be the "most comfortable" size for the computer
- `long` [`int`] large integer (at least 32bits)
- `long long` [`int`] (introduced with C99 standard)

Basic Types

Fixed Point Arithmetic **modifiers**

- `unsigned` variables represent only natural numbers (≥ 0)
- `signed` variables are saved in 2's complement and can hold positive and negative values

Literals and Suffixes

- `-20` (decimal) `024` (octal) `0x14` (hexadecimal)
- `128u` (unsigned)
- `1024UL` (unsigned long)
- `3047LL` (long long)

type	size (x86_32)
char	8bit
short	16bit
int	32bit
long	32bit
long long	64bit

Basic Types

Floating Point Arithmetic (IEEE754)

- `float` single precision floating point number
- `double` double precision floating point number
- `long double` extended precision number

Suffixs

- `3.5F` (float)
- `-5.6L` (double)
- `1.0e-35`

type	size (x86_32)
float	32bit
double	64bit
long double	80bit

Basic Types

Variable Modifiers

- `const`
- `static`
- `extern`
- `register`
- `auto`

Declaring a variable

[<modifiers>] <type> <variable name>;

```
int i;
static const unsigned short US;
long int l;
long long ll;
double d;
```

examples

Basic Types

Declaring a constant

`const [<modifiers>] <type> <const name> = <const value>;`

examples

```
const char CHAR_A = 'A';
const unsigned short int MAX = 128;
const double PI = 3.14159L;
```

In this way a constant (i.e. **a variable with the `const` modifier**) is:

- a really allocated piece of memory initialized with a predefined value
- the value may not be modified run-time

University of Edinburgh

#define

Another way of defining a constant is by using the preprocessing directive `#define`

```
#define LENGTH 100
int a = LENGTH;
```

During the preprocessing phase the above code line is replaced by the next one

```
int a = 100;
```

Operators

- Assignment
- Arithmetic
- Bitwise
- Other Assignment
- Unary Increment
- Relational
- Logical

Assignment Operator

The assignment Operator is the single equals sign =

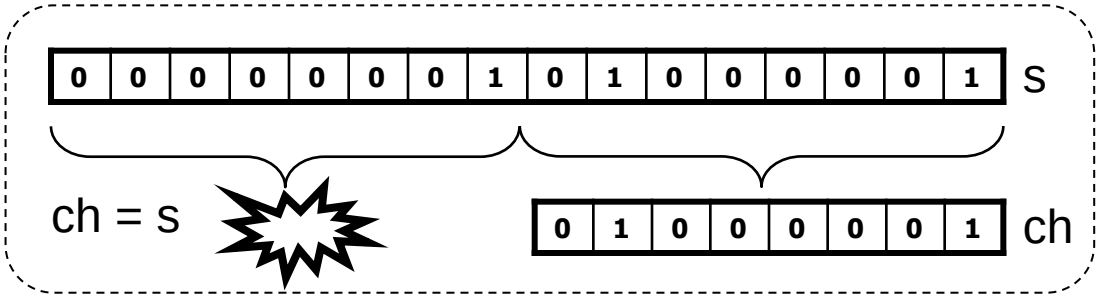
```
int i;
i = 6;
i = i + 6;
// i is now 12
```

examples

About Truncation

- Truncation moves a value from a type to a smaller type
- It may generate a loss of information
- The compiler just drops the extra bits on fixed arithmetic
- Assigning a floating point to a fixed point number drops the fractional part

```
char ch;
short s;
s = 321;
ch = s;
// ch is now 65
```



University of Edinburgh

Arithmetic Operators

Operation Name	Arithmetic Operator	Algebric Expression	C Expression
Addition	**+**	**f + 7**	`f + 7`
Subtraction	**-**	**p – c**	`p - c`
Multiplication	**x**	**bm**	`b * m`
Division	**:**	**x / y**	`x / y`
Remainder (mod)	**mod**	**r mod s**	`r % s`

Operators	Operations	Precedence
`()`	Parentheses	Evaluated first. If parentheses are nested, the expression in the innermost pair is evaluated first. In case of several pairs of parentheses "on the same level" they are evaluated left to right.
`*, /, %`	Multiplication, Division, Remainder	Evaluated second. If there are several, they are evaluated left to right.
`+, -`	Addition, Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Bitwise Operators

Operation Name	C Operator	Algebric example	C Expression
AND	**&**	01b AND 10b = 00b	`a & b`
OR	**|**	01b OR 10b = 11b	`a | b`
XOR	**^**	01b XOR 10b = 11b	`a ^ b`
NOT	**~**	NOT(01b) = 10b	`~ a`
SHIFT Left	**<<**	10b SHIFTl(2) = 1000b	`a << 2`
SHIFT Right	**>>**	1000b SHIFTr(2) = 10b	`a >> 2`

- Bitwise operator NOT is unary operators;
- **SHIFT Right** is the equivalent of **dividing** by a power of 2;
- **SHIFT Left** is the equivalent of **multiplying** by a power of 2;
- Shifting is far more efficient than multiplying and dividing.

Other Assignement Operators

In addition to the plain = operator C includes many shorthand operators which represent variations on the basic =.

Operation name	C operator	C example	C equivalent
Increment by RHS	`+=`	`a += b;`	`a = a + b;`
Decrement by RHS	`-=`	`a -= b;`	`a = a - b;`
Multiply by RHS	`*=`	`a *= b;`	`a = a * b;`
Divide by RHS	`/=`	`a /= b;`	`a = a / b;`
Mod by RHS	`%=`	`a %= b;`	`a = a % b;`
Bitwise AND by RHS	`&=`	`a &= b;`	`a = a & b;`
Bitwise OR by RHS	`|=`	`a |= b;`	`a = a | b;`
Bitwise XOR by RHS	`^=`	`a ^= b;`	`a = a ^ b;`
Bitwise left shift by RHS	`<<=`	`a <<= b;`	`a = a << b;`
Bitwise right shift by RHS	`>>=`	`a >>= b;`	`a = a >> b;`

(RHS = Right Hand Side)

Unary Increment Operators

- The unary operator ++ increments the value of a variable;
- The unary operator -- decrements the value of a variable;
- There are **pre** and **post** variants for both operators.

- `var++` post-increment
- `++var` pre-increment
- `var--` post-decrement
- `--var` pre-decrement

examples

```
int i = 42;
int j;

j = (i++ + 10);
// i is now 43 j is now 52 (NOT 53!)

j = (++i + 10)
// i is now 44 j is now 54
```

Relational Operators

Operator name	C Operator	Return **0** Example	Return **1** Example
Equal	==	`55 == 23`	`55 == 55`
Not Equal	!=	`55 != 55`	`55 != 23`
Greater Than	>	`23 > 55`	`55 > 23`
Less Then	<	`55 < 23`	`23 < 55`
Greater or Equal	>=	`23 >= 55`	`23 >= 23`
Less or Equal	<=	`55 <= 23`	`55 <= 55`

- They operate on integer and floating point arithmetic and return 0 (FALSE) or 1 (TRUE);
- An absolutely classic **pitfall** is to write **assignment** (=) when you mean **comparison**(==); this is not a compiler's problem.

Logical Operators

- The `!` is the unary boolean **not** operator;
- The `&&` is the boolean **and** operator;
- The `||` is the boolean **or** operator.

For these operators

- The value **0 is FALSE**;
- **Anything else is TRUE**;
- The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced.

examples

```
int i = 42;
int j = 1;

j = !i;
// j is now 0

j = i && j;
// j is still 0
```

Exercises

- Implement a C equivalent to the pseudocode of slide 5 in which you define and access a mixed of variables within different scopes
- Write a small program that employs constants also defined as MACROs (#define)
- Try to assign fixed point variables to floating point variables and the contrary, explore truncation
- Try out the examples in slides 18 and 20