C Programming

kv5002

Dr Alun Moon

Computer Science

Lecture 02.1

Text editor Programs are just text! I happen to use `vi`/`vim`, others like `nano` work equally well.

C Compiler Translates source code into executable code. On Linux systems this is usually `gcc`, though I type `cc` out of habit!

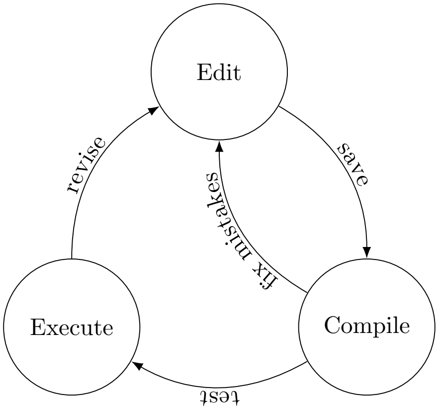
Terminal shell in which to run the tools. I usually have several terminals open.

man pages Most Unix systems should have the "man pages" installed. These document all the shell commands and the c-functions in all the libraries.

Version control such as `git`. Invaluable for keeping track of changes, going back to earlier points,...

Build system such as `make` make the process of building complex projects easier by having a description of the build process.

Writing a program in C on Unix systems goes through a familiar cycle. **Edit** the code, *save* the changes, **Compile** the program and *fix errors* that occur at compilation stage. Then *test* the program by **Executing** it, and revise the program in the light of test results



Editing is done in the editor.

```
$ vi hello.c
$ nano hello.c
```

Compiling is done with the GNU C Compiler (gcc) or the default C compiler (cc). The command line option `-o` sets the output filename to `hello` in this case.

```
$ cc -o hello  hello.c
$ gcc -o hello  hello.c
```

To Execute the program, use the name of the file it is compiled into as the command. Depending on how the `PATH` variable is set you may have to use a `./` prefix.

```
$ hello
$ ./hello
```

hello.c

```c
#include <stdio.h>

int main ( int argc, char *argv[] )
{
        printf("Hello world\n");
}
```

```
$ vi hello.c
$ cc -o hello hello.c
$ ./hello
Hello World
$
```

Sometimes you will make a mistake, and the compiler will fail.

```
$ cc -o hello hello.c
hello.c: In function main:
hello.c:6:1: error: expected ; before } token
 }
  ^
$
```

The compiler will tell you where the error is and what it thinks is wrong. First it tells you which function the compiler is working on. The error message is prefixed with the filename, the line-number, and the character-position. In the example the line number is 6 and the character position 1.

! Beware! The compiler only reports where *it* found an error. *Your* error may be at some point *before* that.

! Beware! The compiler only reports where *it* found an error. *Your* error may be at some point *before* that.

```
1  #include <stdio.h>
2
3  int main ( int argc, char *argv[] )
4  {
5          puts("Hello world")
6  }
```

You can see that the missing semi-colon should be at the end of line 5. With the `vi` and `nano` editors, you can invoke them on the command-line with the line number, and the editor will position the cursor on that line.

```
$ vi +6 hello.c
$ nano +6 hello.c
```

Sometimes a single mistake cam cause a whole series of reported errors! In these cases it is best to fix the first error reported and see how many of the other errors disappear.

```
$ cc -o hello hello.c
hello.c: In function main:
hello.c:5:7: warning: missing terminating " character
  puts("Hello world);
       ^
hello.c:5:7: error: missing terminating " character
  puts("Hello world);
       ^~~~~~~~~~~~~~
hello.c:6:1: error: expected expression before } token
 }
 ^
hello.c:6:1: error: expected ; before } token
$
```

C provides a basic set of data-types, the exact size in bits and the range of values is specified in the standard as a minimum.

> ! Do not rely in types having a specific size in number of bits.

The arithmetic integer types `short`, `int`, and `long`, may be declared with a qualifier of `signed` (the default) or `unsigned`, the sizes remain the same and the ranges are shown in

type		size	lower	upper
`char`	character – a single byte	$\geq$ 8 bit	Null `\0`	Del (`7F` hex)
`short`	short integer	$\geq$ 16 bit	-32676	32676
`int`	integer		-32676	32676
`long`	long integer	$\geq$ 32 bit	$-2^{31}-1$	$2^{31}-1$
`float`	floating point			
`double`	double-precision			

type			lower	upper
`unsigned short`	short integer	$\geq$ 16 bit	0	65535
`unsigned int`	integer		0	65535
`unsigned long`	long integer	$\geq$ 32 bit	0	$2^{32}-1$

Reserve memory for the value and assign an identifier(name), then may include an initial value.

```
char c;
char delim=':';
int i,j;
short n=30000;
long epoc;
unsigned int mask=65535;
unsigned long particles=0
float f, q, y;
double errtol=1e-3;
```

Characters

for single character values, the character is enclosed in single quote marks ' (ASCII 39).

`\0`	Null	ASCII value zero
`\b`	backspace	ASCII value 8
`\n`	line feed	ASCII value 10
`\r`	carrage return	ASCII value 13
`\\`	backslash character	
`\'`	single quote	
`\"`	double quote	

Strings are enclosed in double quote marks " (ASCII 34). Escaped characters can be used within the string.

Integer values

are written out as expected. The compiler tries to use it's best guess as to which data-type the value is. You can explicitly tell the compiler that a value in `long` or `unsigned` by using a suffix letter.

```
short ctr=5;
long  mask = 3472L;
unisgned int delta=34U;
```

Numbers can be given in hexadecimal if the value starts with the characters `0x`

```
int label = 0xABC0F34;
```

Floating point numbers

have to include a decimal point, or else the compiler thinks it is an integer. The compiler assumes that floating point numbers are `double` unless the value has a suffix `f`.

```
double epsilon=0.001;
float half=.5f;
float cube=3.f;
double mega=1e+6;
```

Numbers can be written in *exponential notation* where `1e+6` stands for 1×10^6, `6.67e-11` is 6.67×10^{-11}, and `2.99e8` is 2.99×10^8.

Variables can be declared as constants with the `const` qualifier. This marks the variable as read-only and the compiler complains if you try to write a value to the variable. These have to be given an initial value, as they cannot be subsequently modified.

```
const double pi = 3.141592;
const double  c = 2.99e8;

const unsigned long mask = 0x452facbeUL;
```

The fundamental function for output is `printf` given in the `stdio.h` header.

The `printf` function is unusual in that it can be called with a varying number of parameters. The first parameter *must* be a string, the contents of this string specify what additional values and their types follow.

The Format string for `printf` follows some simple rules. I'll present the simple version here, there is more complexity and options that can be applied.

1. Characters (other than percent % (ASCII 37)) are copied straight to the output.
2. The percent character % (ASCII 37), introduces a placeholder for a value, the following characters specify the type of data to expect as the next parameter to the function.
3. A single character follows the percent, this specifies the type (and format) of the value to be written.

specifier	type	
i	signed integer	`int`
u	unsigned integer	`unsigned int`
f	floating point number	`double`
s	character array (string)	`char*`

```
int n=5;
double f = 0.332;

printf("count %i fraction %f \n", n, f );
```

produces as output

```
count 5 fraction 0.332000
```

problem

We want a program that will read in a shopping list of items, the quantity of each and the item price. As output we want a till receipt showing subtotals and total cost. All neatly formated of course.

printf

Starting with the output we need `printf`.

```
printf("%s\t%d@£%f\t£%f\n",item,qty,ppi,subt);
```

gives

```
flour	2@£5.990000	£11.980000
eggs	12@£0.060000	 £0.720000
milk	1@£1.020000	£1.020000
sugar	3@£3.450000	£10.350000
```

This is not neat!

issues

```
flour    2@£5.990000      £11.980000
eggs     12@£0.060000      £0.720000
milk     1@£1.020000      £1.020000
sugar    3@£3.450000      £10.350000
```

issues

```
flour   2@£5.990000     £11.980000
eggs    12@£0.060000     £0.720000
milk    1@£1.020000     £1.020000
sugar   3@£3.450000     £10.350000
```

- generally prices are in pounds and pence, not micro-pounds μ£

issues

```
flour   2@£5.990000     £11.980000
eggs    12@£0.060000     £0.720000
milk    1@£1.020000     £1.020000
sugar   3@£3.450000     £10.350000
```

- generally prices are in pounds and pence, not micro-pounds $\mu£$
- for neatness the decimal point should line up

issues

```
flour   2@£5.990000     £11.980000
eggs    12@£0.060000     £0.720000
milk    1@£1.020000     £1.020000
sugar   3@£3.450000     £10.350000
```

- generally prices are in pounds and pence, not micro-pounds μ£
- for neatness the decimal point should line up
- the quantities should also line up neatly

How printf works

Two modes

How printf works

Two modes

1. copies ordinary characters from the format string to the output

How printf works

Two modes

1. copies ordinary characters from the format string to the output
2. conversion specifiers

How printf works

Two modes

1. copies ordinary characters from the format string to the output
2. conversion specifiers
   - begins with a %

How printf works

Two modes

1. copies ordinary characters from the format string to the output
2. conversion specifiers
    - begins with a %
    - ends with a conversion character (see next slide)

Conversion characters

`d i`	`int`	decimal number
`x`	`int`	hexadecimal number
`c`	`int`	single character
`s`	`char*`	string (until terminating \0)
`f`	`double`	floating point number

Table: some conversion characters

The conversion consumes the next data item in the parameter list.

Conversion characters

`d i`	`int`	decimal number
`x`	`int`	hexadecimal number
`c`	`int`	single character
`s`	`char*`	string (until terminating \0)
`f`	`double`	floating point number

Table: some conversion characters

The conversion consumes the next data item in the parameter list.

There may be no type-checking, beware of mismatches

Conversion characters

`d i`	`int`	decimal number
`x`	`int`	hexadecimal number
`c`	`int`	single character
`s`	`char*`	string (until terminating \0)
`f`	`double`	floating point number

Table: some conversion characters

The conversion consumes the next data item in the parameter list.

There may be no type-checking, beware of mismatches

```
printf("%d\n",3.14159);
```

produces

Conversion characters

d i	`int`	decimal number
x	`int`	hexadecimal number
c	`int`	single character
s	`char*`	string (until terminating \0)
f	`double`	floating point number

Table: some conversion characters

The conversion consumes the next data item in the parameter list.

There may be no type-checking, beware of mismatches

```
printf("%d\n",3.14159);
```

produces

```
-266631570
```

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.
- a period '.'

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.
- a period '.'
- a number, the precision.

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.
- a period '.'
- a number, the precision.

  string the maximum number of characters to print

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.
- a period '.'
- a number, the precision.

string	the maximum number of characters to print
float	the number of digits after the decimal point

there are others, see the manual page for printf (`man 3 printf`)

Conversion modifers

There are a number of modifiers that can appear between the % and the conversion character. These are *in order*

- a minus sign '-', which specifies left allignment in the field
- a number, the field width. The field is printed in at least this many characters.
- a period '.'
- a number, the precision.

string	the maximum number of characters to print
float	the number of digits after the decimal point
int	the minimum number of digits printed

there are others, see the manual page for printf (`man 3 printf`)

shopping list

revisited

How do we fix our format?

shopping list
revisited

How do we fix our format?

pence use a precision of .2

shopping list

revisited

How do we fix our format?

- **pence** use a precision of .2
- **alignment** right justify in a field (item name is left justified)

shopping list

revisited

How do we fix our format?

- pence: use a precision of .2
- alignment: right justify in a field (item name is left justified)

shopping list

revisited

How do we fix our format?

- **pence** use a precision of .2
- **alignment** right justify in a field (item name is left justified)

```
printf("%-12s\t%-2d@£%5.2f\t£%6.2f\n",
        item,qty,ppi,subt);
```

gives

shopping list

revisited

How do we fix our format?

pence use a precision of .2

alignment right justify in a field (item name is left justified)

```
printf("%-12s\t%-2d@£%5.2f\t£%6.2f\n",
        item,qty,ppi,subt);
```

gives

```
flour        2 @£ 5.99      £ 11.98
eggs         12@£ 0.06      £  0.72
milk         1 @£ 1.02      £  1.02
sugar        3 @£ 3.45      £ 10.35
```

scanf

- Scanf works very similarly to printf

scanf

- Scanf works very similarly to printf
- input rather than output

scanf

- Scanf works very similarly to printf
- input rather than output
- needs pointers for somewhere to write data

scanf

- Scanf works very similarly to printf
- input rather than output
- needs pointers for somewhere to write data
- consumes input differently

How scanf works

like scanf it has a conversion string

How scanf works

like scanf it has a conversion string

- blanks, spaces, tabs are ignored (in effect match any amount of whitespace)

How scanf works

like scanf it has a conversion string

- blanks, spaces, tabs are ignored (in effect match any amount of whitespace)
- ordinary characters, are expected to match the next non-whitespace character on input.

How scanf works

like scanf it has a conversion string

- blanks, spaces, tabs are ignored (in effect match any amount of whitespace)
- ordinary characters, are expected to match the next non-whitespace character on input.
- conversion specification %. . .

consuming input

- automatically skips over whitespace

consuming input

- automatically skips over whitespace
- scanf stops processing when

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string
  - fails to match the input to the next expected character from the format string.

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string
  - fails to match the input to the next expected character from the format string.
- input fields (conversions)

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string
  - fails to match the input to the next expected character from the format string.
- input fields (conversions)
  - sequence of non-whitespace characters

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string
  - fails to match the input to the next expected character from the format string.
- input fields (conversions)
  - sequence of non-whitespace characters
  - capable of conversion to the type specified

consuming input

- automatically skips over whitespace
- scanf stops processing when
  - exhausts the format string
  - fails to match the input to the next expected character from the format string.
- input fields (conversions)
  - sequence of non-whitespace characters
  - capable of conversion to the type specified
  - written into address given by next pointer in the parameter list, which is also consumed.

shopping list

to read the list

```
scanf("%s %d £%f £%f",name,&qty,&ppi,&subt);
```

note

- no & for string, an array name is a pointer
- & for int and float, to obtain the pointer
- no need for format widths

words not strings

the %s specifier will read a sequence of non-whitespace characters, so can only read single words. Spaces separate fields and words so "hello world" is read as two words "hello" and "world"

- experiment with format strings, field widths, precision and allignment
- write a program to write out a times table, properly alligning the columns
- write a program that reads in a file which contains modulecode, weight and mark.

  ```
  cg057 .5 45
  cg107 .2 70
  cm711 .3 90
  ```

  print out a table of modulecode, mark, weight and weighted mark
- if scanf fails to finish the format string (unmatched input), what is the return value? What does it mean? How can it be used?
- what use could the format string `"%d%c"` be for input?
- what does %% do?

Quick Summary

Declaration and access

Array declaration allocates memory for the number of elements specified. These are uninitialised and contain some random value.

declaration

```
int scores[30];
double resistance[100];
```

Arrays are accessed using an index which is an offset into the array. Remember arrays start at 0.

access

```
scores[2] = 29;
factor = resistance[3]*10;
```

How big is the array

no bounds checks on index

When declaring an array you need to make sure there are enough elements to use. In some cases this is obvious, in others less so

bounds

Remember there is no bounds checking. Any integer value can be used as an index.
If the value falls outside the declared array, the program just uses whatever location corresponds to the indexed position.

How big is the array

solutions

- always check the index for bounds. Explicitly using `if` statements.
- make sure `for` loops always terminate within the declared size.

use a constant for array declaration and checking

```
const int points = 56;
float data[points];

if(n>=0 && n<points) data[n] *= 2.0;

for(i=0 ; i<points ; i++)
    sum += data[i];
```

How big is the array

choosing a size

Any array has to be big enough for its needs. Where it is declared has an impact on this.

Globally an array needs a constant for its size. For current versions of gcc[1] this is

- an explicit number `int x[5]` not recommended
- an explicit number via a define'd symbol. better

```
#define N 23
float y[N];
```

- an enumerated constant best

```
enum { space=65 };
float z[space];
```

the second option is by far the most common in practice

[1] 3.4.5

How big is the array

choosing a size

Locally

- the array size can be a variable passed to the function.

```
int sum(int n)
{
    float data[n];
}
```

- a constant integer variable can be used.

```
const int capacity=100;
double heights[capacity];
```

The array is only valid within the function and exists for the lifetime of the function.

Once an array has been declared its size is **fixed**. Its size cannot be changed

How big is the array

choosing a size

Since an array has a fixed size, how big should it be.

- declare a sufficiantly large one and make sure bounds are checked.
  - make sure the size is bigger than any likely needed, say 10000?
  - memory is relativly cheap, but do some quick calculations
- determine the value from the problem
  - for assignment marks there are only numbers up to 100, so for a frequency list use 101 in the size.

> use a named value, defined once at the top of the program to make changing the size easy to do.

Populating an array

Given an aray of floating point numbers

```
enum { n_max = 1000 };
float marks[nmax];
```

To read from standard input, checking the bounds

```
int n_mark = 0;
float x;

while(scanf("%f",&x)!=EOF)
{
    marks[n_mark++] = x;
    if(n_mark==n_max) break;
}
```