Overview: Fundamentals of C, part 1

- **Some simple C programs**
- **C program syntax: identifiers, datatypes, punctuation**
- **Arithmetic operators**

A simple C program

What happens when you *compile* this program?
What happens when you *execute* (or *run*) this program?

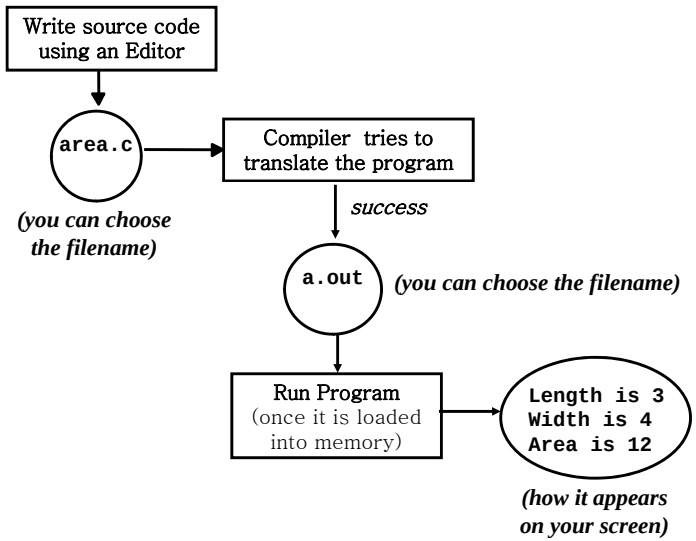
```
/* Date: 18/01/2014       Version: 1.0   */
/* Program to calculate the area of a rectangle, given its length and width */

#include <stdio.h>                      /* definition of printf and scanf */

int main()
{
   int length=3;                        /* declare and initialise length to 3 */
   int width=4;                         /* declare and initialise width to 4 */
   int area;                            /* declare area, no initialisation */

   /* Calculate the area of the given rectangle */
   area = length * width;               /* now area has the value 12 */

 /* Output the data and results to the screen */
   printf("Length is %d\n",length);     /* output: Length is 3  */
   printf("Width is %d\n",width);       /* output: Width is 4   */
   printf("Area is %d\n",area);         /* output: Area is 12   */
return 0;
}
```

A simple C program – variables

```
int length=3;
```

• This is the **declaration** of a **variable** called **`length`**, of type **`int`** (meaning: only holds integer values), and at the same time initialising **`length`** with the value 3

• Variables are a very important concept in C. In order to process data, we need a way of *storing* and *referring to* the data. Using variables, we can store data values in memory, and then use them in subsequent statements when the data is to be processed.

• A variable declaration reserves a memory location of the *appropriate size* in which the variable's data can be stored. In order to know how much memory to reserve, the computer needs to know what type of data (integer, floating-point, …) will be stored.

A simple C program – variables (contd.)

• recall: a variable in C has a name (called its **identifier**), a **memory location** where it is stored, a **type**, and a **value**:
  • identifier should be meaningful
    e.g. **`length`** instead of **`data1`**, **`x`**, etc.
  • identifier, type, and value -- you decide
  • memory location -- the computer decides

• general syntax of a variable declaration: **`type name;`**
  • **`type`** can be **`int`**, **`float`**, **`char`**, …

• optional (but a good idea): **initialise variable at declaration**
  • general syntax: **`type name = value;`**
  • **`value`** can be a constant, e.g. **`int length = 3;`**
  • **`value`** can refer to earlier variables, e.g. could have had

```
int area = length * width;
```


A simple C program – assignment statement

```
area = length * width;
```

• the interpretation of this statement is: get the current values of the variables on the ***right-hand*** side, multiply these values, and store the result in the variable on the ***left-hand*** side.

• general syntax of an assignment statement:
`varname = expression;`
  • **`varname`** is an already-declared variable
  • **`expression`** is a piece of code that evaluates to some value

→ the computer evaluates the current value of **`expression`** and stores the result in **`varname`**, overwriting the current value of **`varname`**

Assignment statement

Warning: in C, the assignment operator **`=`** does **NOT** mean equals (in the mathematical sense). It makes perfect sense to write

```
x = x * 2;
```

which is interpreted as: get the current value of **`x`**, multiply it by 2, and store the result back in **`x`**. ***The effect is to double the value of*** **`x`**

What are the final values of **`x`**, **`y`**, and **`z`** in the following?

```
int x = 4;
int y = 10;
int z = x;
x = y;
y = x;
```

```
int x = 4;
int y = x;
int z;
x = 2;
y = x;
z = y * 3;
```


A simple C program – output to the screen

• how did the code
```
printf("Length is %d\n",length);
printf("Width is %d\n",width);
printf("Area is %d\n",area);
```
produce the screen output
```
Length is 3
Width is 4
Area is 12
```
?

• usually, everything inside the double quotes **`" "`** is written as-is to the screen (the double quotes themselves are not).

• however, **`%d`** is a special character sequence which tells the computer to **replace** the **`%d`** with the value of the **corresponding integer variable** when writing to the screen.

• **`\n`** is an *escape sequence* which tells the computer to start the next output on a new line.

• in order to use **`printf`**, need: **`#include <stdio.h>`**

A simple C program – `main()`

• recall: execution of a C program ***always*** starts with **`main()`**

• **`main()`** is an example of a **function** in C. In general, programs can contain a number of functions, each of which may be responsible for implementing a small part of the overall algorithm.

- Example: write one function to get data input, one function for data output, and one or more functions to process the data. Then activate functions in the correct order to solve the problem!
- the function **`main()`** is activated by the computer's ***Operating System (OS)*** when the program is run.
- **`void main(void)`** means: **`main()`** receives no input data from the OS before the start of program execution, and returns no output data to the OS at program termination.

• the statements that make up a function are called the ***body*** of the function, and are enclosed in braces **`{ }`**

Another simple C program

```
/* Date: 18/01/2014       Version: 1.0    */
/* Program to calculate the area of a circle, given its radius */

#include <stdio.h>                    /* definition of printf and scanf */
#define PI 3.1415927                  /* definition of constant value for π */

int main(void)
{
  int diameter = 3;                   /* declare and initialise diameter to 3 */
  float radius = diameter/2.0;        /* declare and initialise radius to 1.5 */
  float area = PI * radius * radius;  /* declare and calculate area */

  /* Output the data and results to the screen */
  printf("Diameter is %d\n", diameter);        /* output: Diameter is 3  */
  printf("Radius is %.2f\n", radius);          /* output: Radius is 1.50   */
  printf("Area is %.4f\n",area);               /* output: Area is 7.0686   */

return 0;
}
```


Another simple C program – a more detailed look

• since **`radius`** could in general have a fractional part, it is declared as a **`float`**

• **`area`** must also be declared as a **`float`**, since in general it is also a floating-point number.

- What happens if you declare them as **`int`** instead? Answer: their fractional parts are ***truncated*** (not rounded).

• PI is defined as a constant, so program can't change it. Makes the program more readable for us – computer doesn't care!

• **`printf("Area is %.4f\n",area);`** means that the floating-point variable **`area`** is to be written to the screen with 4 places of decimals (***rounding*** is used if needed – as in this case).

- **`%.4f`** is a ***placeholder*** which specifies the format in which the corresponding variable **`area`** will be output.

Another simple C program – importance of variables

• could have written the previous program like this:

```
int main(void)
{
  float radius = 3 / 2.0;              /* declare and initialise radius to 1.5 */
  float area = PI * radius * radius;   /* declare and calculate area */
  (rest as before)
```

(i.e. dropped diameter and replaced it with its value 3). This results in more compact code and gives the same result as before. However, the above code is more difficult to understand.

• variables make programs easier to **read**, **debug**, & **understand**.

• a program can **change the value of a variable while the program is running**. The programs we've examined so far haven't done so, but we'll see examples of this later…

Another simple C program – integer division

• `float radius=diameter/2.0;` **why 2.0 and not 2?**

• answer: `diameter` and 2 are both integers, and according to C, ***the result of dividing one integer by another is also an integer*** – even if that means truncating the result (i.e. dropping the fractional part).

• if we had `radius=diameter/2` , the value of `radius` would be 1 (3/2 = 1.5, then drop the fractional part).

• but if **any** of the quantities involved are floating-point, then the **entire calculation** is done using floating-point arithmetic. We can "force" the computer to do this by making one of the quantities of type `float`: this is why we had 2.0 instead of 2

C program syntax

• when viewed as a typed document, a natural language (e.g. English) document consists of:

words
punctuation
special symbols
spaces and blank lines

• when viewed as a typed document, a C program consists of:

keywords
identifiers
punctuation
comments
spaces and blank lines

C keywords and identifiers

• C keywords: words which have special meaning to the compiler and are essentially the 'building blocks' of the language. These include

`break`, `continue`, `else`, `enum`, `float`, `int`, `do`, `case` …
(this is only a partial list – see any C text for more keywords)

• when choosing an identifier (e.g. a variable name), you can only use uppercase and lowercase letters, the digits 0—9, and the underscore _
• further restrictions on choosing an identifier:
  • **must** start with a letter or _
  • followed by any mixture of letters, digits, and _
  • **cannot** be a C keyword
    • also: maximum recommended identifier length = 31 characters
• examples of ***invalid*** identifiers: `3letter`, `int`, `Hello!`, `air-temp`
• examples of ***valid*** identifiers: `letter3`, `letter_3`, `_letter3`
  • remember: C is case-sensitive – `letter` and `Letter` are different

Datatypes in C

- a datatype defines a set of values and a set of operations on those values
  - operations: comparison of values, arithmetic operations, …
- some standard C datatypes are predefined
  e.g. `float`, `double`, `int`, `long`, `short`, `char`
  - `float` and `double` are for real numbers
  - `int`, `long`, and `short` are for integers (i.e. whole numbers)
  - `char` is used for characters
- a constant in C is simply a specific value:

7	(the integer value 7)
2345	(the integer value 2345)
4.2	(the floating-point value 4.2)
`'a'`	(the character 'a')
`"abcdef"`	(the character string "abcdef")

Datatypes in C (contd.)

Data Type: `int`, `long`, `short`

- not all integers can be represented, due to finite memory constraints

  e.g. **`int`**: 2 bytes wide, ranging from -2^{15} to $(2^{15}-1)$

  `long`: 4 bytes wide, ranging from -2^{31} to $(2^{31}-1)$

  `short`: 1 byte wide, ranging from −128 to 127

(on some computers, **`int`** gets 4 bytes, and/or **`short`** gets 2 bytes)

Data Type: `float`, `double`

- represents real numbers
- integral part and fractional part separated by decimal point

  `float`: for single-precision floating point numbers

  `double`: for double-precision floating point numbers
- **`float`** and **`double`** are datatypes for real numbers but do not include all of them, again due to memory constraints

Datatypes in C (contd.)

- floating-point can represent both integer and non-integer values

  e.g. −1.0, 0.05, 91.1

Scientific notation: -1.0×10^{0}, 5.0×10^{-2}, 9.11×10^{1} *(mantissa × power of 10)*

Exponential notation: −1.0e0, 5.0e–2, 9.11e1

- number of digits the computer allows for the decimal part of the mantissa determines the ***precision*** of the floating-point representation.
- number of digits the computer allows for the exponent determines the ***range*** of the floating-point representation.

Datatypes in C (contd.)

- **datatype hierarchy:**

```
double   ↑ high
float    |
long     |
int      |
short    ↓ low
```

- no information is lost if a value is moved to a higher-order datatype, e.g.

```
double x;
x=10;
```

After this assignment, **`x`** has the value 10.0

- information may be lost if a value is moved to a lower-order datatype, e.g.

```
int a;
a=12.8;
```

After this assignment, **`a`** has the value 12 (due to truncation).

The `char` datatype

- represents individual character value: letter, digit, special symbol e.g. + $. , @
- A single character constant is enclosed in single quotes e.g. **`'A'`**, **`'b'`**, **`'1'`**, **`' '`**

Character variables are typically assigned one byte of storage. They are stored using a standard "encoding", such as **ASCII** (American Standard Code for Information Interchange) or **EBCDIC** (Extended Binary Coded Decimal Interchange Code). Both are ways of assigning each character to a unique byte sequence of 1's and 0's. Examples: ASCII for 'Z' is 01011010, ASCII for '4' is 00110100. In decimal: ASCII for 'Z' is 90, ASCII for '4' is 52.

Characters can be treated as integers! Each **`char`** is stored using these binary codes just like integers, and so we can manipulate them in the same way that we can manipulate other integers:

```
char c = 'a'; /* OR: char c = 97 */
while (c <= 'z') { /* OR: while (c <= 122) */
   printf("%c", c);
   c++;
}
```

Produces the screen output: **`abcdefghijklmnopqrstuvwxyz`**

This works because in ASCII, a-z follow each other in sequence (so do 0-9 and A-Z)

Punctuation in C

- Semi-colon **`;`** --**every C statement must end with `;`**
- Full-stop or "dot" **`.`** --used as the **decimal point** for real numbers.
- Comma **`,`** --used to separate elements of a **list**. For example we can declare a number of variables at the same time (if they are all of the same type):

```
int celsius,fahrenheit,count;
```

- Single quote **`'`** --used to indicate single characters.
- Double quote **`"`** --used to indicate **strings** (sequences of characters).
- Braces **`{ }`** --used to group a number of statements into a single, compound statement.
- Parentheses **`( )`** --used to group certain items such as arguments to functions, or to indicate the order of calculation in arithmetic expressions.

C program layout

• the C compiler doesn't care about spaces, indentation, identifiers, line breaks, and other features of program layout…

```
#include <stdio.h>
#define CON1 0.88288
main() {float v1,v2;printf("Enter the value of the product in Sterling: ");scanf("%f",&v1);
v2=v1/CON1;printf("That equals %.2f Euro.\n",v2);}
```

• … but people do! ***equivalent***

```
#include <stdio.h>
#define CONVERSION_RATE 0.88288

main()
{
  float value_sterling,  value_euro;

  printf("Enter the value of the product in Sterling: ");
  scanf("%f",&value_sterling);

  value_euro=value_sterling/CONVERSION_RATE;

  printf("That equals %.2f Euro.\n",value_euro);
}
```


Arithmetic operators in C

- **`+`** (addition)
- **`-`** (subtraction)
- **`*`** (multiplication)
- **`/`** (division – value of **`3/2.0`** is 1.5, value of **`3/2`** is 1
  Also: value of **`2/7`** is 0)
- **`%`** (remainder – value of **`5%2`** is 1, value of **`4%2`** is 0,
  value of **`2%7`** is 2)

<u>Warning:</u> **`a/b`** or **`a%b`** will cause a **run-time error** if b is 0

• meaning: C compiler won't see an error, but during program execution, if the value of **`b`** is 0 when the above division or remainder operations are applied, the program will fail.

Arithmetic operators in C (contd.)

• Examples:

```
rectarea = length * width;
circarea = radius * radius * PI;
root1 = (-b + sqrt(b*b-4*a*c))/(2.0*a);
```

• parentheses used to explicitly order arithmetic:

```
w = x - y / z;
w = (x - y)/z;
w = x - (y/z);
```

expression in parentheses is evaluated first. nested parentheses: evaluated from the inside out.

• without parentheses: **`*`**, **`/`** before **`+`**, **`-`** and left-to-right between operators with same ***precedence*** (e.g. **`*`** and **`/`**)

Arithmetic operators in C – Exercise

What is the assigned (left-hand side) value in each case?

```
int s, m=3, n=5, r, t;
float x=3.0, y;

t = n/m;
r = n%m;
y = n/m;
t = x*y-m/2;
x = x*2.0;
s = (m+n)/r;
y = -n; /* unary operator */
```

(unary **+**, **–** have higher ***precedence*** than *****, **/**, **%**, which in turn have higher ***precedence*** than binary **+**, **–**)

Arithmetic operators in C – operator precedence

Given the radius of a sphere R, the volume of the sphere is **$(4/3)(\pi)(R^3)$**
Suppose we have a statement **`float radius;`**
Which of the following C statements calculates this volume correctly, and why?

float vol = 4/3 * PI * radius * radius * radius;	***NO***
float vol = PI * radius * radius * radius * 4/3;	***YES***
float vol = (4/3) * PI * radius * radius * radius;	***NO***
float vol = PI * radius * radius * radius * (4/3);	***NO***
float vol = (4.0/3) * PI * radius * radius * radius;	***YES***
float vol = 4.0/3 * PI * radius * radius * radius;	***YES***
float vol = PI * radius * radius * radius * 4/3.0;	***YES***

Operators and expressions

What is the value of each of these C expressions?

```
5 * (3 / 5.0)

5 * (3 / 5)

5 * 3 / 5
```

As seen before, we can "force" the computer to use floating-point arithmetic by using **`5.0`** instead of **`5`** (if this is the desired behaviour). But what if the quantities involved in the expression are variables, rather than constants? For example, how would you ensure floating-point calculations in this case:

```
int i=3, j=5;
float k;
k = j*(i/j); /* want k to be assigned the value 3.0 */
```

We can't just write **`i.0`** or **`j.0`**…

Type casting in C

To "temporarily" force the computer to regard a variable of a particular type as having a ***different*** type, put the desired type in parentheses before the variable name.

Examples:
- **`(float)i`** forces the computer to temporarily regard **`i`** as a **`float`**
- **`(int)x`** forces the computer to temporarily regard **`x`** as an **`int`**

Note: the type of the variable being cast is NOT changed (e.g. if **`i`** had been declared as an **`int`**, it is still an **`int`** – it is just treated as a **`float`** for the duration of the statement in which it is cast to type **`float`**).

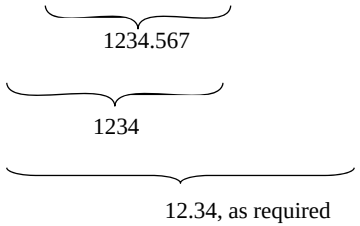
Type casting is not just for variables – can also be applied to constants, e.g.

`(int)6.5`	is the integer "6"
`(float)5`	is the floating point number "5.0"

Type casting in C – an example

Strip all but 2 decimal points from a float (***not rounding***): **12.34567 => 12.34**

```
float trimmed = ((int)(100*number))/100.0 ;
```



(this can be generalised to stripping all but N decimal points, for any value of N – details later)

Arithmetic operators in C – other operators

Increment and decrement operators:

`x++ ;`	is equivalent to	`x = x + 1;`
`y-- ;`	is equivalent to	`y = y – 1;`

Compound (or abbreviated) assignment operators:

`x += 3;`	is equivalent to	`x = x + 3;`
`sum –= y;`	is equivalent to	`sum = sum – y;`
`product *= z;`	is equivalent to	`product = product * z;`
`d /= 4.5;`	is equivalent to	`d = d / 4.5 ;`
`r %= 2;`	is equivalent to	`r = r % 2 ;`

In general: `identifier operator= expression;`
means `identifier=identifier operator (expression);`

(Note: `expression` is evaluated first in these compound assignments)