3 First Steps in C Programming

The following is the typical structure of an executable C program:

```c
/* C Program Structure */

/* This is a
   multline
   comment */

/* These are pre-processor directives
   to include declarations of library functions */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* declarations; */

int main(int argc, char *argv[]) {
  /* declarations; */
  /* statements; */
}

/* other user defined functions */
double sum(double a, double b) {
  /* declarations; */
  /* statements; */
}
```

If the program is to be a stand alone executable then it most have a function named `main`. Most compilers will accept a sloppy version of `main`

```c
main() {
  /* declarations; */
  /* statements; */
}
```

which takes no arguments.

Note that the body of the the function `main` (as well as any other block construction in C) is delimited by a pair of braces (`{ }`).

Comments are delimited by `/*` and `*/` and can span more than one line. Comments are a very important component of any well written program. They contribute nothing to the executable code but make a program readable and understandable to a human. This facilitates collaboration, debugging, documentation, etc.

C does not provide any command to handle complex tasks. All complex functionality is provided in the form of libraries of functions. Some of these libraries are now standardized and became *part* of the C language standard. In order to use functions from these libraries they should be *declared* to the compiler, typically by inputing a so-called *header file* through the *pre-processor* directive `#include`. For example, the lines

```
#include <stdio.h>
#include <math.h>
```

declare many functions for handling character input and output and usual mathematical functions.

All C declarations and statements must end with a semicolon (`;`). Blocks delimited by braces (`{ }`) do not need to be terminated by a semicolon.

The lines starting with the sharp symbol (#) need not end with a semicolon either, even if they seem to contain declarations or statements. Strictly speaking, these lines are not used by the C compiler but by a *pre-processor* that performs some text manipulation prior to program compilation.

3.1 Basic formatted output

One of the most used library function in `stdio` is `printf`. It is used to print on the terminal screen. The call

```
  printf("Hello world!\n");
```

prints *Hello world!* and starts a new line on the terminal screen. Note that the argument of the function `printf` was a sequence of characters in double quotes, that is a *character string*.

Characters preceded by a backslash (\) have a special meaning in C and are translated during compilation. For example `\n` stands for *new line* and `\t` stands for a *tab*. If you need to type the backslash character '\' use `\\`.
The following statement:

```
printf("First line \\\n\n\tSecond line \\\\\n\t\tThird line \\\\\\\n");
```

will print something similar to

```
First line \

        Second line \\
                Third line \\\
```

on your terminal screen. Sometimes it is easier to break up statements so that it is easier to read. The following statements are equivalent to the one above:

```
printf("First line \\\n");
printf("\n");
printf("\tSecond line \\\\\n");
printf("\t\tThird line \\\\\\\n");
```

but are much easier to read.

3.2 Basic data types

All four primary data types in C are listed in the following table

Data Type	Keyword
character	char
integer number	int
floating point	float
double floating point	double
undefined	void

In addition most types accept further qualifiers that affect their storage size and range as follows:

Keyword	Size (bits)	Range	Precision
char	8	[-128, 127]	
unsigned char	8	[0, 255]	
int	16	[-32768, 32767]	
unsigned int	16	[0, 65535]	
long	32	[-2147483648, 2147483647]	
unsigned long	32	[0, 4294967295]	
float	32	[-3.402823e+38, 3.402823e+38]*	$\approx$ 6 digits
double	64	[-1.797693e+308, 1.797693e+308]*	$\approx$ 10 digits
long double	128	[-1.189731e+4932, 1.189731e+4932]*	$\approx$ 19 digits

*: Be careful! Not all numbers in the range are represented! For example, only { -1.401298e-45, 0, 1.401298e-45 } are represented in the range [-1.401298e-45, 1.401298e-45]!

We will discuss more on that later.

Variables are created in a C program by simply declaring their type. For

example:

```
int n, ijk;
char letter = 'A';
double val = 3.7;
```

creates, i.e. allocates memory space and declare `n`, `ijk`, `letter` and `val` as valid variables in a C program. Note that more than one variable can be declared at a time separated by comma. The variable `letter` has been *initialized* with the character `A` and the variable `val` with the number 3.7.

Identifiers

Variable names must be valid *identifiers*, that is a sequence of *letters* and *digits* that obey the following rules (ANSI-C):

1. The first character must be a letter;
2. The underscore '_' counts as a letter;
3. Upper and lower case letters are different;
4. Have any length but only the first 31 characters are significant.

The following are examples of valid identifiers:

```
n
alpha33
a3ldp22
x_0
ThisIsAnIdentifier
long_names_help
```

The following identifiers are reserved (AKA *keywords*), and may not be used as identifiers:

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  short     union
const     float   signed    unsigned
continue  for     sizeof    void
default   goto    static    volatile
do        if      while
```

3.3 Simple programs with integers

Let's create a program which declares an integer identifier, `num1`, assigns the number 100 to it and prints the content of `num1` by using `printf` on the screen.

```
/* Program w2-1.c */

#include <stdio.h>

main() {

  /* declare integer variable num1 */
  int num1;

  /* assign value 140 to num1 */
  num1 = 140;

  /* print value on screen */
  printf("The first value of num1 is %d.\n", num1);

  /* assign value -30 to num1 */
  num1 = -30;

  /* print value on screen */
  printf("The second value of num1 is %d.\n", num1);

}
```

Two novelties here. First the assignment

```
  num1 = 140;
```

which is performed with the help of the *operator* '='; second the use of `printf` to print the value of an integer variable. Note that the `printf` now takes two arguments: a string specifying a *format* followed by the variable `num1`. In this example, `printf` will replace the '%d' by the value of variable `num1`. We place '%d in the format string at the location where the integer has to be printed and list the name of the identifier after the format string. We will study other format specifications later. The output of the above program is

```
The first value of num1 is 140.
The second value of num1 is -30.
```

The assignment operator ('=') means *replace the current value of the left hand side argument with the value of the right hand side argument* and is different than the equal operator ('==') which we will study later. In particular, '='

returns the assigned value so that its result can be use by other operators or functions. This is illustrated in the next example

```
/* Program w2-2.c */

#include <stdio.h>

main() {

  /* declare and initialize integer variables num1 and num2 */
  int num1 = 2, num2 = 6;

  /* print values on screen */
  printf("1. num1 = %+3d,\tnum2 = %03d\n", num1, num2);

  /* assign value 1 to both variables */
  num2 = num1 = 1;

  /* print values on screen */
  printf("2. num1 = %+3d,\tnum2 = %03d\n", num1, num2);

  /* assign value 1 to both variables */
  num2 = 1 + (num1 = -1);

  /* print values on screen */
  printf("3. num1 = %+3d,\tnum2 = %03d\n", num1, num2);


}
```

in which we have two integer variables, `num1` and `num2`. The nested assignment should be interpreted

```
  num2 = (num1 = 1);
```

that is, `num2` is assigned to the result of the rightmost assignment `num1 = 1` which in this case is the integer `1`. In the second assignment

```
  num2 = 1 + (num1 = -1);
```

one is added to the result of the first assignment before being assigned to `num2`. The operator plus (+) is used to add two integers in this case.

Note also the presence of two '%d's in the format string and calls to `printf` with both `num1` and `num2` as arguments. The above program generates the following output:

```
1. num1 =  +2,  num2 = 006
2. num1 =  +1,  num2 = 001
```

```
3. num1 =   -1,   num2 = 000
```

The tab character '\t' is used to align the columns in the output. In the above example the decimal format specifier '%d' has been modified to account for spacing and padding by adding modifiers between the '%' and the 'd'. For example '%+3d' forces the integers to be printed taking up at least 3 spaces always with a plus or minus sign even when the number is positive and the specifier '%03d' prints an integer using three spaces padded with zeros on the left as needed.

3.3.1 Integer input with `scanf`

We will now use another function from the `stdio` library, `scanf`, to read an integer value store it in an integer variable. For example, the statement

```
scanf("%d", &num1);
```

interrupts until the <return> key has been pressed and then assigns, if possible, the contents to the integer variable `num1`. Note that the specifier '%d' is the same used with `printf` to specify that the input is a decimal integer. An essential part for the correct execution of the statement is the use of the *address-of operator* '&'. It passes to `scanf` the memory address where the variable `num1` is stored so that `scanf` can put the input directly into that memory address.

3.3.2 Important note on the usage of the address-of operator

As we will learn later, C can only pass arguments *by value* to functions. Therefore, in order to `scanf` known which variable to copy the terminal input to we need to provide the variable memory addresses rather then the value of the variable as its argument. That is the reason for the use of the address-of operator '&'. Note that memory address are essentially integers so that the compiler has no reason to deem the following statement

```
scanf("%d", num1);
```

as invalid. It *will not generate any compilation errors or warnings*! The program will compile and run, but will not perform as you would expect, possibly trowing a *run-time error* that will interrupt your program.

The following is a complete example of using `scanf` to read an integers from the terminal.

```c
/* Program w2-3.c */

#include <stdio.h>

main() {

  /* declare integer variable num1 */
  int num1;

  /* prompt for input */
  printf("Enter an integer: ");

  /* read integer from terminal */
  scanf("%d", &num1);

  /* print value on screen */
  printf("num1 = %d\n", num1);

}
```

You should try to "break" this code by providing invalid integer inputs to get a better feeling on how `scanf` behaves! For example, type letters in between your inputs (e.g. '123ab34'), or large numbers that you know will overflow (e.g. '2147483648'). A computer program that does not know how to handle invalid input is still one of the major sources of security breaches on computer software at all levels, from the operating system to all sorts of applications.

The following program will read and add two integers:

```c
/* Program w2-4.c */

#include <stdio.h>

main() {

  /* declare integer variables num1, num2 and num3 */
  int  num1, num2, num3;

  /* prompt for input */
  printf("Enter two integers: ");

  /* read integers from terminal */
  scanf("%d %d", &num1, &num2);

  /* add the two numbers */
  num3 = num1 + num2;
```

```
  /* print sum on screen */
  printf("%d + %d = %d\n", num1, num2, num3);

}
```

Apart from the predictable behavior of the *plus operator* '+', the other point to notice is the whitespace between the two '%d's in `scanf`. This will tell `scanf` to continue reading after the first valid integer ignoring all whitespace in between. Whitespaces includes regular spaces, tabs and new lines.

3.3.3 Binary arithmetic operators

In C, arithmetic addition, subtraction, multiplication, division, and modulus are expressed by the operators +, −, *, /, %, respectively. These operators take *two operands*, therefore the name *binary operators*. We have already seen + in action. The other operators work similarly. For example:

```
  c = a + b; /* addition */
  d = a - b; /* subtraction */
  e = a * b; /* multiplication */
  f = a / b; /* division */
  g = a % b; /* modulus */
```

These operators, with the exception of the modulus operator '%', work on integers as well as doubles and mixed numeric types. Arithmetic properties and precedence rules are taken in the usual mathematical sense.

The modulus operator % is used to compute the remainder in a division between two positive integers:

```
  num1 = 5 % 2;    /* num1 =  1 */
  num2 = 6 % 3;    /* num2 =  0 */
  num3 = 7 % 3;    /* num3 =  1 */
  num4 = 7 % 11;   /* num4 =  7 */
  num5 = 7 % 5;    /* num5 =  2 */
  num6 = -5 % 2;   /* num6 = -1 */
```

WARNING: The last expression above, which has a negative dividend, does not conform to the mathematical definition of modulus of having a positive remainder!

Also be careful with integer division. For example, the result of

```
  num1 = 1 + 1 / 2 + 1 / 2;    /* num1 = 1 != 2 */
```

because the result of the integer division `1/2 = 0`! Division will throw a *run-time* error when the dividend is zero.

3.4 Simple programs with doubles

A variable of type `float` or `double` is a *floating-point number*. These variables are essentially rational numbers and there is only a finite number of rational numbers that can be represented exactly. All other number will be rounded or truncated to fit a `float` or `double`. How the numbers are actually stored and processed in operations is beyond the scope of this course. You can set a floating point variable in C by using either a standard floating-point notation or scientific notation. The following are valid statements in C using floating-point notation:

```
double a, b, c, d = -1.;
a = 25.6;
b = -0.00000012345;
c = -2345600000.;
```

and their equivalent scientific notation:

```
double a, b, c, d = -1e0;
a = 2.56e1;
b = -1.2345e-7;
c = -2.3456e9;
```

Note the use of `e` to denote a decimal exponent as usual and that variable `d` is directly initialized at its declaration. One can also use a capital '`E`' instead of a lower-case '`e`'.

Floating-point numbers can be printed using `printf` with the following format specifiers '`%f`' or '`%e`' where using the first specifier prints a number using floating-point notation and using the second prints a number in scientific notation. The specifier '`%E`' prints a number in scientific notation using a capital '`E`' for the exponent. A third specifier is the '`%g`' which will print a number using '`%f`' or '`%e`', whichever is shorter. '`%G`' does the same but uses '`%E`'. As with integers you can add modifiers to control the number of decimals being printed with both `%f` and `%e`. For example `%10.8f` will print a number in floating point notation taking a total of 10 spaces with 8 decimals.

The following program illustrate the use of `printf` with floating point numbers:

```
/* Program w2-5.c */
```

```c
#include <stdio.h>

main() {

  double x = 1e-6, y = 2310;

  printf("x = %20.18e\n", x);
  printf("y = %20.18e\n\n", y);

  printf("x + y = %f + %f = %f\n", x, y, x + y);
  printf("x + y = %e + %e = %e\n", x, y, x + y);
  printf("x + y = %g + %g = %g\n", x, y, x + y);

}
```

It produces the following output:

```
x = 9.999999999999999547e-07
y = 2.310000000000000000e+03

x + y = 0.000001 + 2310.000000 = 2310.000001
x + y = 1.000000e-06 + 2.310000e+03 = 2.310000e+03
x + y = 1e-06 + 2310 = 2310
```

The output also illustrates the interesting fact that the number 1e-6 does not have an exact *binary* floating-point representation even though it has an exact *decimal* floating-point representation!

3.4.1 A program with a mathematical function

The next program will use doubles and a call to the mathematical function `sqrt`, which computes square roots, in order to compute the distance between two points in the plane:

```c
/* Program w2-6.c */

/* This program computes the distance between two points in the
   plane. The x and y coordinates of two points, (x1, y1), (x2, y2)
   are defined in the declaration. */

#include <stdio.h>

/* include math.h so that we can use sqrt ()
   you will need to instruct the compiler to link with the
   math library as
```

```
        gcc w2-6.c -lm

 */
#include <math.h>

main() {

  /* declare double variables with the coordinates */
  double
    x1 = 2., y1 = 2.3,
    x2 = 5., y2 = 7.;

  double dx, dy, distance;

  /* computes coordinate differences */
  dx = x2 - x1;
  dy = y2 - y1;

  /* computes distance
     note that C does not have a power operator */
  distance = sqrt(dx*dx + dy*dy);

  /* print points and distance */
  printf("The distance between the points (%g, %g) "
         "and (%g, %g) is %g \n",
         x1, y1, x2, y2, distance);

}
```

A number of new things here. First note that the pre-processor directive `#include <math.h>` to include the standard C mathematical library and the later use of the function `sqrt`, which takes a positive double and returns a double (no complex numbers here!). Also note that squares are computed by taking products as C does not have a native power operator. `math.h` does have a power function, `pow(a,b)`, which computes a^b for arbitrary doubles a and b. A cosmetic and useful feature is the automatic concatenation of strings which allows one to break the format string in the `printf` function in two lines.

Finally, as we intend to use a function declared in `math.h` we need to tell the compiler to link with the actual library file. As we will learn later `math.h` is *not the actual library* but simply a *header file* that declares the functions available in the library. The actual library file is located in a special directory in the server[1]

[1] Usually `/usr/lib/`.

and in the case of `math.h` is called[2] `libm.a`. The use of the flag `-l`*name* will make the compiler link a program with the library file `lib`*name*`.a`. Hence:

```
iacs5.ucsd.edu% gcc w2-6.c -lm
```

will link our program with the mathematical library. Note that an error will be generated by the compiler if you do not tell the linker which file to link with. So at `iacs5.ucsd.edu` we get the following error after trying to compile

```
iacs5.ucsd.edu% gcc w2-6.c
/var/tmp//ccGt1EcB.o(.text+0xa8): In function `main':
: undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

The message is telling you that the function `sqrt` has been properly declared and is been used by your program but could not be found in any of the standard library files. Note that this error is not issued by the *compiler* (`gcc`) but the *linker* (`ld`) which is implicitly invoked by the compiler. Indeed, one can break up this process in two by instructing the compiler not to invoke the linker and then invoking the linker separately:

```
iacs5.ucsd.edu% gcc -c w2-6.c
iacs5.ucsd.edu% gcc w2-6.o
w2-6.o(.text+0xa8): In function `main':
: undefined reference to `sqrt'
collect2: ld returned 1 exit status
iacs5.ucsd.edu% gcc w2-6.o -lm
```

The flag `-c` instructs the compiler to stop right after generating the machine code, which is stored in the file named `w2-6.o`, before invoking the linker. As you can see, no compilation errors were generated! The header file properly declared the function `sqrt`. However, when we invoke the linker (here again by calling `gcc`) this time with the machine code file `w2-6.o` and not the source code `w2-6.c` we see the *undefined reference* error appear once again. Passing the appropriate library file flag fixes the error.

3.5 A word of caution if mixing integer and floating-point numbers

Integers and floating-point numbers are not interchangeable even when they may represent the same mathematical number! For example, in C, the integer `1` and the floating-point `1.0` *are not the same number*! In particular the equal

[2] This is if a *static library* is to be used. We will talk more about static versus dynamic libraries later in the course.

operator (==) returns false if forced to compare `1` and `1.0`! This can be a source of confusion and programming errors.

Another common mistake is the use of fractions with integers. The expression `1/2` is interpreted by C as the integer division of the integer `1` by the integer `2` which is equal to the integer `0` (we have seen an example of that before). The following are other examples of operations involving integers (and their expected results):

```
 2/3  (=0)    -5/2  (=-2)  199/800  (=0)
-6/2  (=-3)  199/3  (=66)   -203/1  (=-203)
```

If you want to use fractions to express a rational number you have to have at least one of the arguments as a floating-point number. For example, `1./2` will be equal to floating-point number `0.5`. This last expression takes advantage of C's *mixed-type* feature. The compiler automatically *promotes* integers to floating-points based on the involved operands. The same examples as before now involving floating-points and mixed types yield:

```
  2./3  (=0.666666)    -5./2  (=-2.5)       199./800  (=0.24875)
-6./2.  (=-3.)       199./3.  (=66.333333)  -203./1.  (=-203.)
```

It is a good idea to avoid mixed operations by explicitly *casting* integers and floating-points to the desired target type. This can be done with the use of *casting operators*, e.g. `(int)` to integer and `(double)` to double. For example, `(double)3` is the same as `3.0` and `(int)2.5` the same as `2`. Here are some examples involving explicit type-casting

```
      2/(double)3  (=0.666666)  -(double)5/2  (=-2.5)
(int)2./(int)3.  (=0)          (int)sqrt(4)  (=2)
```

The last example illustrates casting the return value of a function returning a double. Note that `sqrt` takes a double as an argument so the compiler automatically converts the integer `4` into a double before calling it. The instruction is equivalent to `(int) sqrt((double)4)`.

3.6 Integer and floating-point operators

Arithmetic and logic operations in C are performed using *unary* and *binary* operators. *Unary* and *binary* refer to the number of arguments involved in the

operation.

Examples of *unary operators* are:

1. Plus and minus signs: `+`, `-`;
2. Increment and decrement: `++`, `--`;
3. Type casting: `(int)`, `(double)`, `(unsigned int)`;

Examples of *binary operators* are:

1. Addition and subtraction: `+`, `-`;
2. Multiplication and division: `+`, `-`, `*`, `/`;
3. Modulus (for integers): `%`

Arithmetic operators take precedence as expected, with `*`, `/`, and `%` being of higher precedence than `+` and `-`. Use parentheses to alter the order of evaluation.

For example, the expression `15 - 3 * 5` produces the value `0`, while `(15 -3)*5` yields the value `60`. A slightly more complex example is:

```
num1 = a * b + c / d * d;
```

which evaluates the mathematical operation

$$ab + \frac{c}{d}d$$

When two of the above binary operators are at the same precedence level then they are evaluated from *left-to-right*. That is how the compiler effectively resolves the expression '`c / d * d`' as $(c/d)d$ and not $c/(dd) = c/d^2$.

The increment and decrement operators '`++`' and '`--`' can be applied to both integers and floating-point numbers and increment by 1 or decrement by 1:

```
int x = 3, y = 6;
y++; /* y = y + 1 */
x--; /* x = x - 1 */
```

At the end of the execution of the above excerpt `y` and `x` have the values 7 and 2, respectively.

There are two versions of the increment and decrement operators based on their position: as a *prefix* (e.g. `++count`, `--k`) or as a *postfix* (e.g. `count++`, `k--`). The *prefix* version increments or decrements *before* its value is used and the *postfix* version increments or decrements *after* its value is used. It is easier to see in the examples:

```
int n = 5, x;
x = n++;          /* x = 5 and n = 6 */
```

and

```
int n = 5, x;
x = ++n;          /* x = 6 and n = 6 */
```

Also when involving other operators as in:

```
int x = 2, y = 1, w;
w = ++x - y;    /* x = 3 and w = 2 */
```

and

```
int x = 2, y = 1, w;
w = x++ - y;    /* x = 3 and w = 1 */
```

Expressions such as '`x = x + 3`' in which the variable on the left-hand side is repeated immediately on the right, can be written in a compact form as `x += 3`. The binary operators +, -, *, / and % have the corresponding abbreviated operators +=, -=, *=, /=, and %=. These have lower precedence than all standard binary operators. Here are some examples:

```
x += 3;          /* x = x + 3 */
sum += x;        /* sum = sum + x */
d /= 3.5;        /* d = d / 3.5 */
r %= 2;          /* r = r % 2 */
x *= y + 1;      /* x = x * (y + 1); */
(x *= y) + 1;    /* x = x * y + 1; */
```

3.7 Mathematical functions

These are some of the mathematical functions defined in `math.h`:

`double fabs(double x)`	$\lvert x \rvert$
`double sqrt(double x)`	$\sqrt{x}$
`double pow(double x, double y)`	x^y
`double exp(double x)`	e^x
`double log(double x)`	$\log x$
`double log10(double x)`	$\log_{10} x$
`double sin(double x)`	$\sin x$
`double cos(double x)`	$\cos x$
`double tan(double x)`	$\tan x$
`double asin(double x)`	$\arcsin x \in [-\pi/2, \pi/2]$
`double acos(double x)`	$\arccos x \in [0, \pi]$
`double atan(double x)`	$\arctan x \in [-\pi/2, \pi/2]$
`double atan2(double y, double x)`	$\arctan(y/x) \in [-\pi, \pi]$
`double sinh(double x)`	$\sinh x$
`double cosh(double x)`	$\cosh x$
`double tanh(double x)`	$\tanh x$

Other functions in `math.h` deal with integer division, rounding and truncation. The above functions assume that the types of arguments are double, and the functions all return a double.

The function `atan` always returns an angle in the first and fourth quadrants, whereas `atan2` returns an angle that can be in any quadrant depending on the signs of x and y. Thus, in many applications, the `atan2` function is preferred over the `atan` function.

Angles for trigonometric functions are expressed in *radians*. Conversion between radians and degrees can be done as in the next program which computes the area of circle:

```
/* Program w2-7.c */

#include <stdio.h>

/* The next preprocessor directive defines a value to be used */
#define PI 3.141592659

main() {

  /* declare variables */
  double area, radius = 2.5;
```

```
  /* compute the area */
  area = PI * radius * radius;

  /* display radius and area */
  printf("radius = %f\narea   = %f\n", radius, area);

}
```

Note the use of the pre-processor directive `#define` in order to define a symbol for later use in the program. It turns out that π is already defined in `math.h` so the next program will include `math.h` so as to have access to π:

```
/* Program w2-7a.c */

#include <stdio.h>

/* we include math.h in order to access the constant M_PI */
#include <math.h>

main() {

  /* declare variables */
  double area, radius = 2.5;

  /* compute the area */
  area = M_PI * radius * radius;

  /* display radius and area */
  printf("radius = %f\narea   = %f\n", radius, area);

}
```

The next program does the exact same thing as the above two program, this time using a more "modern" C construction in order to declare a constant by preceding a declaration variable by the word `const`:

```
/* Program w2-7b.c */

#include <stdio.h>

main() {

  /* declare variables */
  double area, radius = 2.5;

  /* we now declare PI using a C construction */
  const double PI = 3.141592659;
```

```
  /* compute the area */
  area = PI * radius * radius;

  /* display radius and area */
  printf("radius = %f\narea   = %f\n", radius, area);

}
```

Finally, this last example illustrates how `scanf` can be used to input floating-point numbers:

```
/* Program w2-7c.c */

#include <stdio.h>

main() {

  /* declare variables */
  double area, radius;

  /* we now declare PI using a C construction */
  const double PI = 3.141592659;

  /* read input */
  printf("Enter circle radius: ");
  scanf("%lf", &radius);

  /* compute the area */
  area = PI * radius * radius;

  /* display radius and area */
  printf("radius = %f\narea   = %f\n", radius, area);

}
```

The format specifier `%f` copies the input to a `float` whereas the specifier `%lf` copies the input to a `double`.

3.8 Standard input and output

As you already know, we use `printf` to output on the screen and `scanf` to capture input from the terminal. Do not forget to include:

```
  #include <stdio.h>
```

in order to have these function declared.

3.8.1 The function `printf`

The format tag follows the prototype

```
%[flags][width][.precision][length]specifier
```

where `specifier` and the optional `length` parameter is as follows:

Specifier	Output
%c	character
%s	string
%d	signed decimal
%u	unsigned decimal
%e, %E	float in scientific notation
%f	float in decimal notation
%g	shorter of %f or %e

Length	Argument
%l	`long int`
%L	`long double`

the `width` controls the number of characters to be printed and `precision` the number of decimals in the case on floating-point numbers, as explained before.

Finally the optional `flag` parameter can take

Flag	Effect
-	Field is left justified
+	Always prints a number with '+' or '-'
' ' (space)	Blank space inserted if no sign
0	Left pad a number with zeros

Here are some examples with integers:

Number	Specifier	Output
-135	%d	-135
-135	%4d	-135
-135	%6d	-135
-135	%-6d	-135
135	%+d	+135
135	%06d	000135

And here are some examples with floating-point numbers:

Number	Specifier	Output
147.9797	%f	147.979700
147.9797	%+f	+147.979700
147.9797	% f	147.979700
147.9797	%6.2f	147.98
147.9797	%8.2f	147.98
147.9797	%-8.2f	147.98
147.9797	%e	1.479797e+02
147.9797	%.3e	1.480e+02
147.9797	%g	147.980

The next program illustrates these formats:

```
/* Program w2-13.c */

#include <stdio.h>

main () {

  /* declare char, int, double and long double variables */
  char ch;
  int i, j;
  double h, u;
  double g, d;
  float fh, fu;

  /* assign values */
  ch = 'A';

  i = -31768;
  j = 31767;

  h = 1004235678901212.123455689;
  u = 1.00423567890121212345568 9;

  g = 100.423567;
  d = 1.054e-10;

  h = 1234567891123456789.1123456789;
  u = 1.2345678911234567891123456789;

  fh = 1234567891123456789.1123456789;
  fu = 1.2345678911234567891123456789;

  /* print char */
```

```
  printf("> char\n");
  printf("  ch is %c\n", ch);

  /* print integers */
  printf("> int\n");
  printf("  i is %d\n", i);
  printf("  i is %u\n", i); /* this will interpret the - sign wrongly */

  printf("  j is %d\n", j);
  printf("  j is %u\n", j); /* the + sign is just fine */

  /* print doubles */
  printf("> double\n");
  printf("  g is %f\n", g);
  printf("  g is %.2f\n", g); /* rounded to two digits */

  printf("  d is %e\n", d);
  printf("  d is %.4e\n", d); /* rounded to four digits */

  printf("  h is %f\n", h); /* double can handle up to 16-17 digits */
  printf("  u is %.17f\n", u);

  /* print floats */
  printf("> float\n");
  printf("  fh is %f\n", fh); /* float can handle up to 7-9 digits */
  printf("  fu is %.17f\n", fu);

}
```

It produces the following output:

```
> char
  ch is A
> int
  i is -31768
  i is 4294935528
  j is 31767
  j is 31767
> double
  g is 100.423567
  g is 100.42
  d is 1.054000e-10
  d is 1.0540e-10
  h is 1234567891123456768.000000
  u is 1.23456789112345677
> float
  fh is 1234567939550609408.000000
  fu is 1.23456788063049316
```

Finally you can input some special character through the use of *escape sequences*. These are character sequences preceded by a backslash (\). Some common escape sequences are given in the next table.

Escape Sequence	Character
\n	new line
\b	backspace
\f	form feed (new page)
\r	carriage return
\?	question mark
\t	horizontal tab
\v	vertical tab
\'	single quote
\"	double quote
\\	backslash
\a	bell character

The next program illustrates the use of escape sequences:

```
/* Program w2-15a.c */

#include <stdio.h>

main() {

  printf("\'Hello\' \"World\"\?\n");
  printf("Horizontal Tabulation:\n");
  printf("1\t2\t3\t4\n");
  printf("Vertical Tabulation:\n");
  printf("1\v2\v3\v4\n");
  printf("\a");

}
```

Its output is as follows:

```
'Hello' "World"?
Horizontal Tabulation:
1       2       3       4
Vertical Tabulation:
1
 2
  3
   4
```

3.8.2 The function `scanf`

`scanf` reads input from the terminal. The following excerpt reads the values of year, x and y from the terminal:

```
int year;
double x, y;

/* &year means the memory address of year */
scanf("%d", &year);

/* &x and &y mean the memory addresses of x and y */
scanf("%lf %lf", &x, &y);
```

Recall the use of the *address-of operator* (`&`) in order to pass the address of the variable rather than its value to *scanf*. The variable type is controlled using format specifiers just as in the case of `printf`.

Specifier	Output
`%d`	int
`%ld`	long int
`%u`	unsigned int
`%lu`	long unsigned int
`%f`	float
`%lf`	double
`%Lf`	long double

Note the use of `%lf` to read `doubles`.

We can use the function `getchar()` if we needs to read a single character from the terminal. The use of this function will be illustrated later.