Programming with C

Terry Marris November 2010

6 Selections

Previously we looked at arithmetic with numbers of type *integer*. Now we start our study of the *if ...* statement.

6.1 The Equality Operators

The equality operators are:

==	equals, is-the-same-as
!=	not equals

If we have

```
int a = 2;
int b = 2;
int c = 3;
```

then *a* == *b* and *a* != *c*.

If we have

```
char a = 'X';
char b = 'X';
char c = 'Z';
```

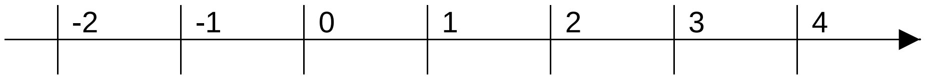
then *a* == *b* and *a* != *c*.

Notice the difference between = and ==. = is the assignment operator; it copies from right to left. == is the equality operator; it determines whether two values are identical.

The *!* is the negation operator; it negates whatever immediately follows it.

6.2 The Relational Operators

Think of an integer number line:



We can see that 3 is more than 2. We write 3 > 2.

We can see that 2 is less than 3. We write 2 < 3.

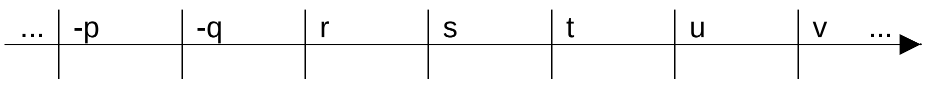
Less than has its point on the **L**eft. Mo**R**e than has its point on the **R**ight.

The relational operators are:

<	less than
<=	less than or equal to
>	more than
>=	more than or equal to

So, 2 <= 2 and 2 >= 2. Notice that the = sign is always written after the < or the >. and there is no space between < and =, between > and =.

Look at the character line shown below.



We can see that s < t, t > s, and s <= s and s >= s.

The ordering of letters of the alphabet is made possible by the ASCII collating sequence. ASCII is a table that maps characters to numbers. For example, A = 65, Z = 90, a = 97, z = 122, space = 32. These numbers are used to represent characters in a computer. ASCII stands for American Standard Code for Information Interchange.

Notice that *!*< is equivalent to >=, and *!*> is equivalent to <=.

6.3 Precedence

We remember that precedence is the order in which operators in an expression are evaluated. We see how the equality and relational operators fit in with the familiar mantra brackets first, then multiply and divide, then add and subtract.

Operator	*Description*	*Precedence*
( )	brackets	highest priority
++ --	increment and decrement operators	
* / %	times, divide, mod	
+ -	add, subract	
< <= > >=	relational operators	
== !-	equality operators	
=	assignment operator	lowest priority

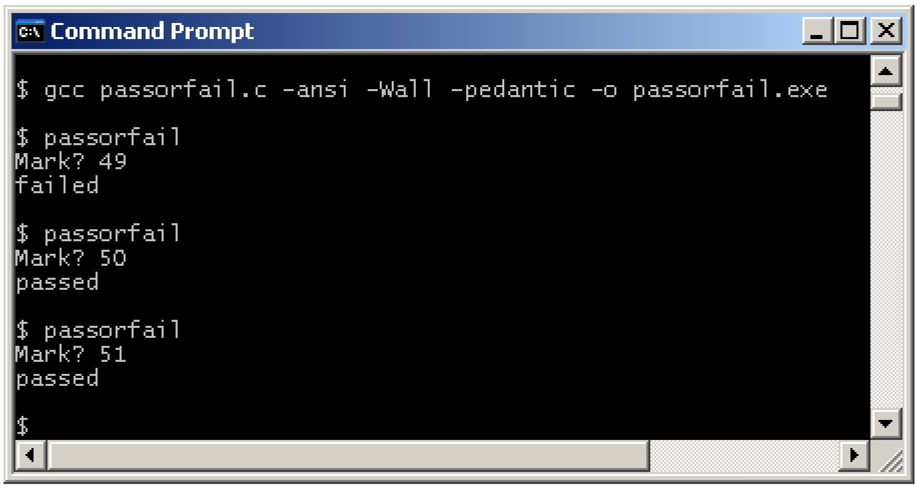
6.4 if ... else ...

If the program is broken, then fix it; otherwise, leave it alone.

If introduces a decision. *the program is broken* is an example of a *Boolean* expression: its value is either true or false. Either the program is broken, or it is not. *fix it* is the action to be taken if the program is broken. *leave it alone* is the action to be taken if the program is not broken. We refine these ideas into a C program.

The pass mark for an exam is 50. The next program, shown below, inputs a mark and outputs whether the student has passed or failed.

```
/* passorfail.c: displays whether a student has passed */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int mark;
  char string[BUFSIZ];

  printf("Mark? ");
  gets(string);
  mark = atoi(string);
  if (mark >= 50)
    printf("passed\n");
  else
    printf("failed\n");
  return 0;
}
```



```
$ gcc passorfail.c -ansi -Wall -pedantic -o passorfail.exe

$ passorfail
Mark? 49
failed

$ passorfail
Mark? 50
passed

$ passorfail
Mark? 51
passed

$
```

Here, we have run the program three times: once with a mark input of 49, once with a mark of 50, and once with a mark of 51.

mark >= 50 is an example of a *boundary*. A boundary occurs when a small change in a variable causes a large change in behaviour. *50* is a boundary point because a mark less than *50* causes *fail* to be displayed, a mark more than *50* causes *pass* to be displayed.

The essence of the program is the selection statement

```
if (mark >= 50)
  printf("passed\n");
else
  printf("failed\n");
```

(mark >= 50) is an example of a *Boolean* expression. A Boolean is either true or false.

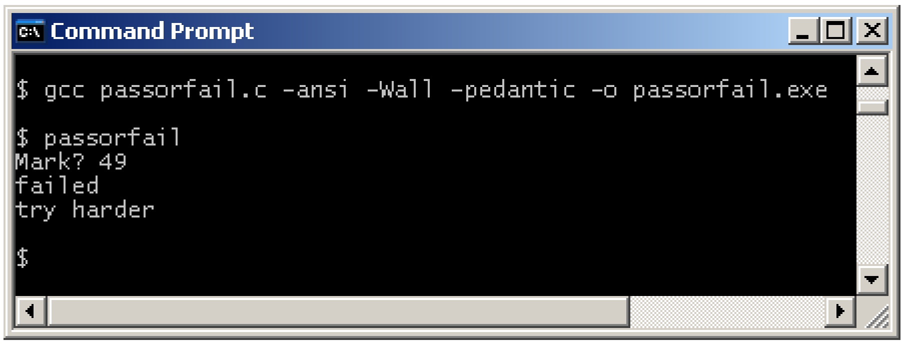
If *(mark >= 50)* is true then *passed* is displayed.

If *(mark >= 50)* is false, then *failed* is displayed.

Notice the layout. *else* is aligned directly under the *if.* The *printf()* statements are indented by two spaces.

If there is more than one statement subject to the Boolean expression then we need to include them within a statement block, as shown in the next program.

```
/* passorfail.c: displays whether a student has passed */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int mark;
  char string[BUFSIZ];

  printf("Mark? ");
  gets(string);
  mark = atoi(string);



  if (mark >= 50){
    printf("passed\n");
    printf("congratulations\n");
  }
  else {
    printf("failed\n");
    printf("try harder\n");
  }
  return 0;
}
```

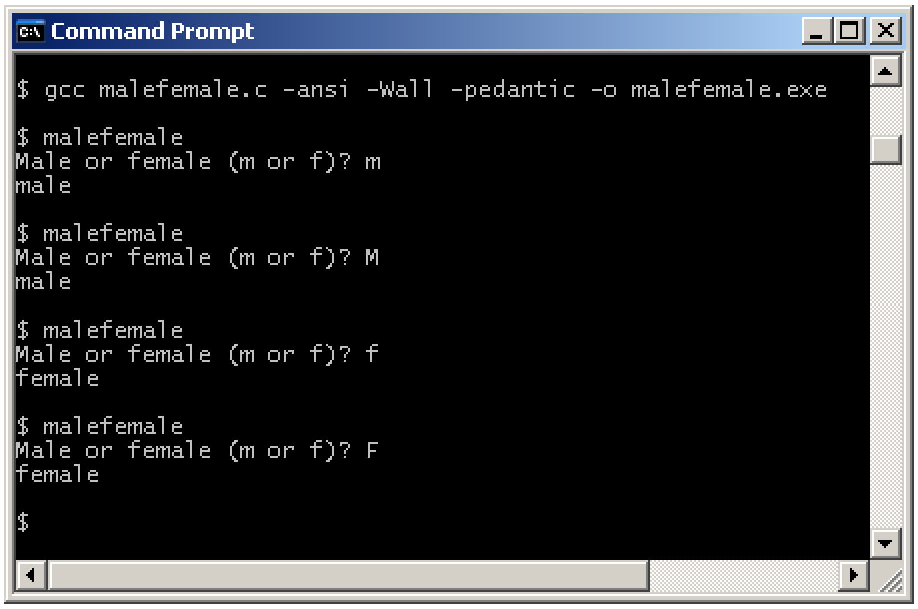


```
$ gcc passorfail.c -ansi -Wall -pedantic -o passorfail.exe

$ passorfail
Mark? 49
failed
try harder

$
```

Again, notice the layout. The *{* that marks the beginning of a statement block is alongside the *if* ... or the *else* .... The *}* that marks the end of the block is on its own line directly under the *if*. Paying attention to layout like this enables you to see at a glance whether your braces are balanced, or not. Whatever layout standard you adopt, be consistent and helpful, not only to yourself but also to the person who has to read your coding or fix your errors.

Here is another example.   The program invites the user to enter a single character to represent a gender, then displays that gender in full.

```
/* malefemale.c: displays either male or female */

#include <stdio.h>
#include <ctype.h>

int main()
{
  char string[BUFSIZ];
  char gender;

  printf("Male or female (m or f)? ");
  gets(string);
  gender = string[0];
  gender = tolower(gender);
  if (gender == 'm')
    printf("male\n");
  else
    printf("female\n");
  return 0;
}
```



```
$ gcc malefemale.c -ansi -Wall -pedantic -o malefemale.exe

$ malefemale
Male or female (m or f)? m
male

$ malefemale
Male or female (m or f)? M
male

$ malefemale
Male or female (m or f)? f
female

$ malefemale
Male or female (m or f)? F
female

$
```

First, we prompt for an input and store the first character entered in *gender*.

```
printf("Male or female (m or f)? ");
gets(string);
gender = string[0];
```

Then we convert the character entered to lower case with *tolower()*. Why? Well, *M* is not the same as *m* and *F* is not the same as *f*.

```
gender = tolower(gender);
```

Then the selection statement is straightforward.

```
if (gender == 'm')
  printf("male\n");
else
  printf("female\n");
```

But what if the first character entered was neither an *m*, *f*, *M* or *F*? We are coming to that.

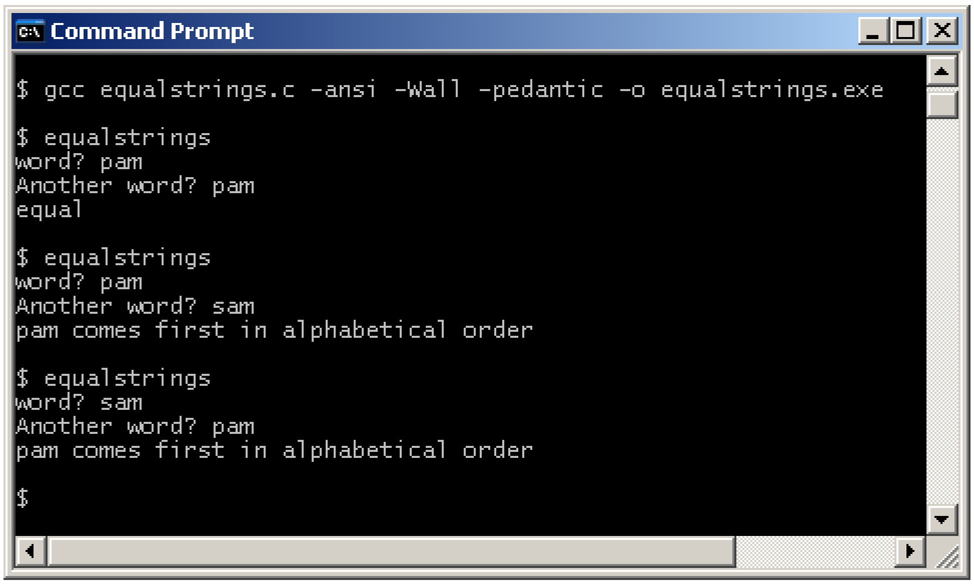
tolower() converts a character to lower case. *toupper()* converts a character to upper case. *isupper()* returns true if its character is an upper case character. *islower()* returns true if its character is a lower case character. *tolower()*, *toupper()*, *islower()* and *isupper()* are all defined in *ctype.h*.

What is truth? In C, true is a non-zero integer value.

6.5 strcmp()

The *strcmp()* function compares two strings. *strcmp(string1, string2)* returns zero if they are identical, a negative integer if the *string1* precedes *string2* in alphabetical (or dictionary) order, and a positive integer if *string2* precedes *string1*. *strcmp()* is used in the program shown below.

```
/* equalstring.c: inputs two strings, determines their alpha order */

#include <stdio.h>
#include <string.h>

int main()
{
  char string1[BUFSIZ];
  char string2[BUFSIZ];

  printf("word? ");
  gets(string1);
  printf("Another word? ");
  gets(string2);
  if (strcmp(string1, string2) == 0)
    printf("equal\n");
  if (strcmp(string1, string2) < 0)
    printf("%s comes first in alphabetical order\n", string1);
  if (strcmp(string1, string2) > 0)
    printf("%s comes first in alphabetical order\n", string2);
  return 0;
}
```



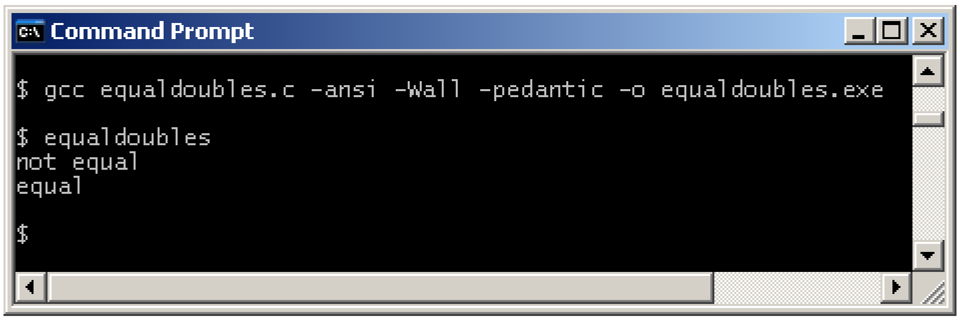
```
$ gcc equalstrings.c -ansi -Wall -pedantic -o equalstrings.exe

$ equalstrings
word? pam
Another word? pam
equal

$ equalstrings
word? pam
Another word? sam
pam comes first in alphabetical order

$ equalstrings
word? sam
Another word? pam
pam comes first in alphabetical order

$
```

The two string values must be of the same case i.e. all lower case or all upper case. We see how to change all the characters in a string to upper case or to lower case in the chapter on arrays.

6.6 Equal Doubles

Not all numbers of type *double* can be held exactly. First, numbers are converted into binary before storage, and there might not be an exact representation of a real number in binary. A computer's memory is finite and so there is a limit to the precision i.e. number of digits that a computer can hold. Also, some numbers are irrational, i.e. the sequence of digits after the decimal point goes on without end. So, we have to agree that two real numbers are identical if their difference is sufficiently small. What is sufficiently small? Well if we are dealing with, say, values with two digits after the decimal point, then a reasonable measure of smallness might be 0.005.

```
/* equaldoubles.c: equality of numbers of type double */

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

int main()
{
  const double epsilon = 0.005;
  double a = 0.01;
  double b = 0.1 * 0.1;
  double difference = a - b;

  if (a == b)
    printf("equal\n");
  else
    printf("not equal\n");

  if (fabs(difference) < epsilon)
    printf("equal\n");
  else
    printf("not equal\n");

  return 0;
}
```


Command Prompt

```
$ gcc equaldoubles.c -ansi -Wall -pedantic -o equaldoubles.exe

$ equaldoubles
not equal
equal

$
```

Check. We have

```
double a = 0.01;
double b = 0.1 * 0.1;

if (a == b)
  printf("equal\n");
else
  printf("not equal\n");
```

which results in *not equal* being displayed. Not equal? Check the maths. 0.1 x 0.1 is 0.01. That is just my point. Computers do not always get maths involving a decimal point exactly right. Now look at this code fragment.

```
const double epsilon = 0.005;
double a = 0.01;
double b = 0.1 * 0.1;
double difference = a - b;

if (fabs(difference) < epsilon)
  printf("equal\n");
else
  printf("not equal\n");
```

does result in *equal* being displayed. Basically, we are saying that two numbers are equal if their difference is sufficiently small. We define sufficiently small in *epsilon* (the Greek letter ε often used by mathematicians to represent an arbitrarily small positive value).

fabs(difference) just remove the minus sign if there is one. Then all we need to do is to check whether this *difference* is less than *epsilon* for two numbers to be considered equal.

We always consider using this method when comparing two numbers of type *double* for equality, especially in safety-critical systems. Best of all, we avoid comparing numbers of type *double*, if we reasonably can.

Exercise 6.1

1. A car park has spaces for fifty cars. Write and test a program that inputs the number of cars parked, and displays *space available* if the number of cars parked is less than 50, or *car park full* if the number of cars parked is 50.

2. Write and test a program that invites a yes/no response to the question *Are you a householder?* If the response is *y* or *Y*, the program should display *go to question 5*, otherwise, the program should display *go to question 10*.

3. Write and test a program that prompts for a password to be entered, and compares the password entered with one held within the program. The program should output either *access denied* or *you're in*.

4. The roots of the quadratic equation $ax^2 + bx + c = 0$ are given by $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ provided $a \neq 0$ and $b^2 \geq 4ac$. Write and test a program that inputs the coefficients a, b and c, and outputs the two real roots of a quadratic equation.

5. Write and test a program that outputs whether a person is clinically obese. A person is clinically obese if their body mass index, that is, if their weight (in Kg) divided by their height (in metres) squared is 30.0 or more.

We have seen how to compare two values for equality, how to use the equality and relational operators, and how to write basic selection statements.

Next we see look at sequences of if ... statements and the logical operators.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988
http://catless.ncl.ac.uk/risks accessed Nov 2010.