2 Arithmetic

2.1 The Natural Numbers

We have all used natural counting numbers such as 1, 2 and 3 in simple arithmetic calculations. In C, the natural counting numbers between 0 and 65535 inclusive are known as *unsigned int* values. (*unsigned* because the values are never negative, *int* because they are integers, that is, whole numbers.)

Just as in ordinary arithmetic, we can add, subtract, multiply and divide *unsigned int* values. And we can find the remainder after dividing one *unsigned int* by another.

in normal arithmetic			in C		
7 + 2	=	9	7 + 2	==	9
7 - 2	=	5	7 - 2	==	5
7 x 2	=	24	7 * 2	==	14
7 ÷ 2	=	3 remainder 1	7 / 2	==	3
			7 % 2	==	1

The two equal symbols == together mean *is the same as*, the * means *multiply* and the *%* means *the remainder after division*.

An *unsigned int* value can never be negative (less than zero). So, as far as *unsigned ints* are concerned, the expression *2 - 3* has no meaning; 2 - 3 cannot equal -1. Similarly, if the largest *unsigned int* is *65535*, then the result of *65535 + 1* cannot be an *unsigned int*. (With some C programming systems the largest *unsigned int* value may be more than *65535*.) The result of any calculation involving *unsigned int* values must remain within the range from *0* up to *65535* if the answer is to be sensible.

Let us write a simple C program which will print the result of adding two *unsigned int* numbers.

```
/* program 2.1 - prints the sum of two numbers. */
#include <stdio.h>

int main()
{
  printf("%u", 7u + 2u);
  printf("\n");
  return 0;
}
```

The result of running this program is that *9* is displayed on the screen.

In ordinary arithmetic, we would write the number seven, for example, like this: 7. But in C we indicate that the number is an *unsigned int* value. We do this by attaching the letter *u* (or *U*) to the number thus: *7u*.

Now look at the *printf("%u", 7u + 2u)* statement. The *%u* specifies that the result of *7u + 2u* is to be printed in unsigned decimal format. In this context, decimal means base 10 - the base used for normal everyday arithmetic. And the format is the normal, everyday way in which positive whole numbers are written.

%u is an example of a conversion specification. A conversion specification specifies the format in which a value is to be shown on the screen. Notice that the conversion specification is enclosed within quotation marks.

We have seen the letter *u* used for two different purposes. First, it is used in a conversion specification. Second, it is used to specify that a number is an *unsigned int* value.

Program 2.1 is not particularly useful. The program has to be changed if it is to print the sum of two different numbers. This may not be a problem to people who are programmers. But people who use programs are not usually programmers. We cannot expect non-programmers to make changes to programs. So what we want the next program to do is

- allow the user to enter a number at the keyboard
- store the number in a suitable place in the computers memory
- enter another number and store it in another location in memory
- sum the contents of these two stored values and display the result on the screen

But first we follow a small but necessary diversion.

2.2 Variables

A computer's memory is perhaps best regarded as a store. Data, such as numbers, can be held in the store. Processes, such as arithmetic, can be applied to data held in the store.

The store comprises a sequence of storage locations. Each storage location is numbered from zero up to (say) 65535 and every location has its own number. The number which identifies a storage location is known as its address.

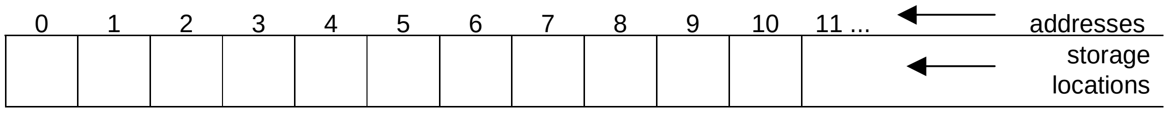

Figure 2.1 - store is a set of sequentially numbered storage locations.

A storage location is a container in which a value may be stored.

Suppose the value *26* is held in the store. Then we can imagine a storage location at address *65508* say, which contains this value

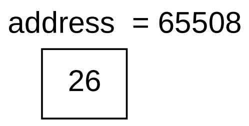

Figure 2.2 - a location in store contains a value

A storage location can hold a value of a specified kind or type. An example of a type is an *unsigned int*. A value of type *unsigned int* is a whole number, either zero or positive. So, a value such as 3.1416, which is not a whole number, cannot be stored in a location specified to hold an *unsigned int* value. A location in store designated to hold a value of some specified type is known as a *variable*. How can we define a variable in C? To define a variable which can store a value of type *unsigned int,* and to name it *aNumber*, we write

```
unsigned int aNumber;
```

At this point, the value stored in *aNumber* is not defined. But the address at which the variable is located is determined automatically by the C programming system.

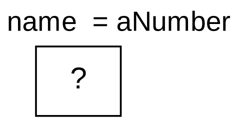

Figure 2.3 - the contents of a variable are undefined - unless a value is explicitly stored there

We could allocate a value, *28* say, to be stored in this variable by writing

```
unsigned int aNumber = 28u;
```

The symbol = is known as the *assignment operator*. It copies the value of whatever is written on its right into a variable written on its left. In our example, it places the value *28* into the variable *aNumber*. Now we can picture the variable thus:

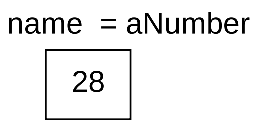

Figure 2.4 - the variable named aNumber contains the value 28.

Program 2.2 shown below prints the address at which the variable *aNumber* is located, together with the value it contains.

```
/* program 2.2 - displays the address at which a variable is
               located, together with the value stored at that
               address. */
#include <stdio.h>

int main()
{
  unsigned int aNumber = 28u;

  printf("The variable is located at address %u", &aNumber);
  printf("\n");
  printf("It contains the value %u", aNumber);
  printf("\n");
  return 0;
}
```

The result of running this program on my computer system is that

```
The variable is located at address 65508
It contains the value 28
```

is displayed on the screen.

Every variable must be defined before it is used. And every variable must be given a name. A variable name is also known as an identifier. Identifiers must not contain spaces or symbols such as & and -. We use the convention that identifiers chosen by ourselves always begin with a lower case letter and are always written entirely in lower case, except the first letter in a word - this is always written in upper case. In our example shown above, the variable identifier is *aNumber*. Any identifier chosen by ourselves should be descriptive, that is, its name should suggest its purpose. So, for example, the identifier *aNumber* is chosen because the variable's purpose is to store a number value.

The statement

```
printf("It contains the value %u", aNumber);
```

is straightforward. Even though the conversion specification *%u* is part of the text between the quotation marks, it is not printed (the *%* symbol sees to that). But the text *It contains the value* is printed.

The ampersand symbol in *&aNumber* specifies that the address at which the variable *aNumber* is located and is the value to be printed. And since an address is a positive number, the *%u* conversion specification is used. (Strictly, the conversion specification should be *%p* but this usually prints an address in hexadecimal format, which is not so easy to understand.)

2.3 Interactive Input

Program 2.1 - to display the result of adding two numbers - is limited because, in order to find the sum of a different pair of numbers we have to change the program.

If a non-programmer was using our program, then it would be unreasonable to expect him or her to amend it. Our strategy now is to obtain input from the keyboard while the program is running. How can that be done?

First, we prompt the user with something like

```
printf("Please enter a number: ");
```

Then we read the keys pressed by the user and store them. Any key pressed on the keyboard generates a character. In C, a character is known a *char*. To define a variable to store a sequence of characters we write

```
char string[BUFSIZ];
```

Here, *char* is a data type. *string* is a variable identifer chosen to suggest a sequence of characters - in computing, a sequence of characters is known as a *string*. *BUFSIZ* specifies the maximum number of characters to be stored. Its value, typically *512*, is defined in *stdio.h*. To store the character values entered at the keyboard we write

```
gets(string);
```

gets gets a string of characters from the keyboard and places them in the named variable. Here, the named variable is *string*. *gets* waits for the user to press the return or enter key; this key indicates that there are no more characters to be entered. The characters may be letters of the alphabet, but in this case we expect them to be digits because we are dealing with numbers.

Finally, we convert the character digits into a single number in *unsigned int* format. This is achieved by the function *sscanf*. *sscanf* scans a string and converts its contents into a specified format. To convert the contents of *string* into *unsigned int* format, and to store the result in a variable named *aNumber* we write

```
sscanf(string, "%u", &aNumber);
```

The conversion specification, *%u*, is the same as the one used in the *printf* function. Notice the use of the address operator, *&*, which prefixes the variable's identifier; this is compulsory. Both *gets* and *sscanf* are defined in the *stdio* library.

We need to read natural numbers entered at the keyboard on different occassions for different purposes. So it would be handy if we wrote a single function to do the job in most circumstances. Here is one version of the function. Do not worry if you find the function strange; functions are explained in more detail in chapter four. You do not need to understand the function fully in order to use it.

```
unsigned int unsignedNumberRead()
{
  char string[BUFSIZ];
  unsigned int aNumber = 0u;

  printf("Number? ");
  gets(string);
  sscanf(string, "%u", &aNumber);
  return aNumber;
}
```

For example, given the variable definition

```
unsigned int ageInYears;
```

we can use the function to input a number from the keyboard by writing

```
ageInYears = unsignedNumberRead();
```

Let us look at some of the details not yet explained. The last statement in the function is

```
return aNumber;
```

The effect of this is to send the value stored in *aNumber* back to the point where the function was called. In our example shown above, this value is placed in the variable *ageInYears*. Here is another example:

```
unsigned int numberOfStudentsInThisRoom;
numberOfStudentsInThisRoom = unsignedNumberRead();
```

The value contained in the function's variable *aNumber* is returned and stored in *numberOfStudentsInThisRoom*.

The prompt *Number?* is perhaps not appropriate for every situation. Since we would like to use the same function but in different cirmcumstances, we want to be able to specify the prompt to suit the occassion. For example:

```
ageInYears=unsignedNumberRead("What is your age in years? ");
```

or

```
numberOfStudents = unsignedNumberRead("How many students? ");
```

Here, we have specified the required prompt at the point of call. Now we modify the function to match.

```
unsigned int unsignedNumberRead(char prompt[])
{
  char string[BUFSIZ];
  unsigned int aNumber = 0u;

  printf("%s", prompt);
  gets(string);
  sscanf(string, "%u", &aNumber);
  return aNumber;
}
```

The function heading is

```
unsigned int unsignedNumberRead(char prompt[])
```

unsigned int tells you that the function returns an unsigned integer value. The function identifier is *unsignedNumberRead*. The function requires a string to be given to it. This is specified by the expression

```
char prompt[]
```

The maximum number of characters to be contained in the string variable *prompt* is not specified here; it is specified before the function is called. The function is supplied with a value to be stored in *prompt* at the point where the function is called.

So, for example, if the function call is

```
ageInYears=unsignedNumberRead("What is your age in years? ");
```

then *prompt* would contain *What is your age in years?*.

The contents of prompt are displayed by the statement

```
printf("%s", prompt);
```

Here, the conversion specification is *%s*. The *s* specifies that the contents of *prompt* are to be printed in the format of a string, that is, as a sequence of characters.

Finally, we come to the program itself. The program, shown below, asks the user to enter two numbers. The program then displays the result of adding these two numbers together.

```
/* program 2.3 - inputs two numbers, outputs their sum. */
#include <stdio.h>

unsigned int unsignedNumberRead(char []);     ←   function prototype

int main()                                    ←   main function
{
  unsigned int aNumber, anotherNumber;

  aNumber = unsignedNumberRead("Number? ");
  anotherNumber = unsignedNumberRead("Another number? ");
  printf("Their sum is %u", aNumber + anotherNumber);
  printf("\n");
  return 0;
}

unsigned int unsignedNumberRead(char prompt[])
{
  char string[BUFSIZ];                  ⎫
  unsigned int aNumber = 0u;            ⎪
                                        ⎪    function
  printf("%s", prompt);                 ⎬ ←  implementation
  gets(string);                         ⎪
  sscanf(string, "%u", &aNumber);       ⎪
  return aNumber;                       ⎭
}
```

An example of a program run is

```
Number? 3
Another number? 4
Their sum is 7
```

The values chosen and entered by the user are shown in bold type.

Notice the layout. First we have the function prototype. This announces to the compiler that we have a function named *unsignedNumberRead* that receives a string (signified by the *char []*) and returns an unsigned integer. Then we have the *main* function. Then we have the *unsignedNumberRead* function itself.

Exercise 2.1

1 Run program 2.2, shown in section 2.2 above, on your computer system.

2 Write a program to the following specification. The program is to ask the user to enter two numbers. The numbers entered should be stored in variables. Then the program should display the result of multiplying the two numbers together.

3 Write a program which will print the result of dividing one number by another.

4 Write a program which will print the remainder obtained after dividing one number by another. Remember that division involving *unsigned int* values always truncates, that is, cuts off, any fractional part. So, for example, 7 / 3 == 2 and 2 / 3 == 0.

5 Write a program which will input a number of days and calculate and output the number of weeks and days it contains. For example, if the user entered 17 days, then the program should display 2 weeks 3 days. Hint: you could use *days / 7* to give you the number of weeks, and *days % 7* to give you the number of days in the part week.

2.4 The Large Natural Numbers

The maximum value of type *unsigned int* is typically *65535*, hardly big enough to deal with astronomical distances or figures involving millions of pounds. C provides the *unsigned long int* data type whose largest value is at least *4,294,967,295* that is, over four thousand million. The following program prints the largest value that can be stored in a variable of type *unsigned long int*.

```
/* program 2.4 - prints the largest unsigned long int value.*/
#include <stdio.h>
#include <limits.h>

int main()
{
  printf("%lu", ULONG_MAX);
  printf("\n");
  return 0;
}
```

The value represented by *ULONG_MAX* (unsigned long maximum) is defined in *limits.h*. This value is printed by the line

```
printf("%lu", ULONG_MAX);
```

Here, the conversion specification is *%lu* for *long unsigned int*. Notice that it is the letter ell which follows the *%* symbol and not the number 1.

The usual arithmetic operators, +, -, *, / and % can be used with values of type *unsigned long int*. Let us look at a simple example which uses values of type *unsigned long int*.

Suppose that every week the area covered by green algae on a pond doubles in size. This week's algae population size is two times last week's. We express this by writing

algaePopulationSize' = algaePopulationSize x 2

The expression means the current *algaePopulationSize* is equal to the previous *algaePopulationSize* times *2*. Here, the prime symbol (') means the value after the calculation has taken place. The equivalent in C is

```
algaePopulationSize = algaePopulationSize * 2;
```

This means that the value of two times the old *algaePopulationSize* is assigned to, or stored in, *algaePopulationSize*; the old value is overwritten with the new value. The following diagram illustrates the point.

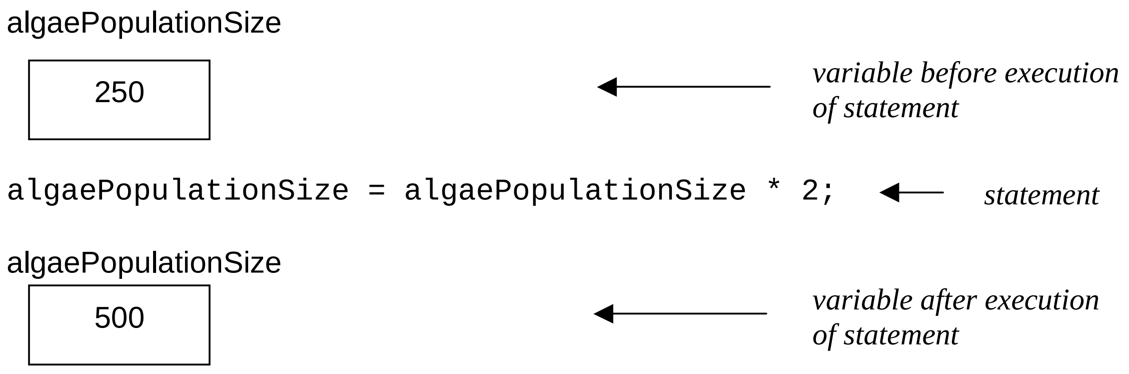
algaePopulationSize
250
variable before execution of statement
algaePopulationSize = algaePopulationSize * 2;
statement
algaePopulationSize
500
variable after execution of statement

Figure 2.5 - shows the contents of the variable before the calculation has taken place, and afterwards.

The following program inputs this weeks area covered by algae and outputs the area which will be covered next week.

```
/* program 2.5 - calculates the new area of algae which */
/*               doubles in size every week.            */
#include <stdio.h>

unsigned long int unsignedLongRead(char []);

int main()
{
  unsigned long int algaePopSize;

  algaePopSize =
     unsignedLongRead("This weeks area covered by algae? ");
  algaePopSize = algaePopSize * 2ul;
  printf("The area covered next week will be %lu",
               algaePopSize);
  printf("\n");
  return 0;
}

unsigned long int unsignedLongRead(char prompt[])
{
  char string[BUFSIZ];
  unsigned long int aNumber = 0ul;

  printf("%s", prompt);
  gets(string);
  sscanf(string, "%lu", &aNumber);
  return aNumber;
}
```

An example of a program run is

```
This weeks area covered by algae? 64000
The area covered next week will be 128000
```

The function *unsignedLongRead* is much the same as the function *unsignedNumberRead* described in section 2.3 above, the essential difference being the line

```
sscanf(string, "%lu", &aNumber);
```

Here, the conversion specification is *%lu*, just as it is in the *printf()* function.

Now, the conversion characters for values of type *unsigned long int* is the letter l followed by u, but specific values of type *unsigned long int* have the letters *ul* (or UL) attached to them - the order in which the letters are written is reversed. This is illustrated in the following example which prints the value of *128000ul*.

```
/* program 2.6 - prints an unsigned long constant value. */
#include <stdio.h>

int main()
{
  printf("%lu", 128000ul);
  printf("\n");
  return 0;
}
```

When run, this program prints *128000* on the screen.

Specific, explicitly stated numeric values such as *128000ul* and *65000u* are known as *constants*.

Exercise 2.2

1 Run program 2.4. Then amend the program so that it displays the value of *UINT_MAX*. *UINT_MAX* represents the largest unsigned integer value on your computer system; its value is defined in *limits.h*.

2.5 The Signed Integers

Sometimes we need to use negative numbers. For example, a temperature of -10 degrees celsius represents 10 degrees below zero, thefreezing point of water.

A whole number value between, typically, -32768 and +32767 is known in C as an *int* value. *int* stands for integer; it means a whole number such as -3, -2, -1, 0, 1, 2, 3 and 4. An *int* constant does not have have a suffix (whereas *unsigned int* constants end with *u* or *U*). The usual arithmetic operators can be used with *int* values.

Program 2.7 shown below prints the least and the largest int value.

```
/* program 2.7 - prints the least and the largest int value.*/
#include <stdio.h>
#include <limits.h>

int main()
{
  printf("Least int value is %d \n", INT_MIN);
  printf("largest int value is %d \n", INT_MAX);
  return 0;
}
```

Notice here that the newline character, *\n*, has been included with the text to be displayed.

%d is the conversion specification for decimal (that is, base 10) integers. And *%d* is used in the conversion of a string to an *int* value as shown in the function *intNumberRead*.

```
int intNumberRead(char prompt[])
{
  char string[BUFSIZ];
  int aNumber = 0;

  printf("%s", prompt);
  gets(string);
  sscanf(string, "%d", &aNumber);
  return aNumber;
}
```

Just as we have *unsigned int* and *unsigned long int* data types, so we have the *long int* type. The largest long *int* value is represented by *LONG_MAX*. *LONG_MAX* is defined in *limits.h* and its value is, typically, *2,147,483,647*. The value of *LONG_MIN* is usually -*2,147,483,647*. The conversion specifier for *long int* values is *%ld* (the letter ell followed by d). And a *long in*t constant ends with the letter *l* (or *L*), for example: *123456789L*.

Exercise 2.3

1 Write a program which will print the least and the largest long int value.

2 Write a function named *longIntRead* which will input a number from the keyboard as a *long int*. Then write a program to test the function. Hint: you could use the *intNumberRead* function to guide you.

2.6 THE REAL NUMBERS

We use the *float* data type when we want to maintain some accuracy in calculations involving division, for example, when we want 2 divided by 3 to equal 0.666667 (and not 0 as would be the case of we were dividing the integer 2 by the integer 3). A constant value of type *float* contains a decimal point and the suffix *f* or *F*.

We use the *double* data type when we want even more accuracy in calculations, such as those that might be used for scientific purposes or for company accounting systems. *double* stands for *double precision float*.

The usual arithmetic operators, +, -, * and / can all be used with values of type *float* or *double*.

The following program, which features the *double* data type, asks the user to enter a number representing a value in degrees celsius, and then it calculates and outputs the corresponding value in degrees fahrenheit.

```
/* program 2.8 - converts a temperature in degrees celsius */
/*               into fahrenheit.                           */

#include <stdio.h>

double doubleNumberRead(char []);

int main()
{
  double celsius, fahrenheit;

  celsius = doubleNumberRead("Celsius? ");
  fahrenheit = 9.0 * celsius / 5.0 + 32.0;
  printf("Fahrenheit equivalent is %0.2f \n", fahrenheit);
  return 0;
}

double doubleNumberRead(char prompt[])
{
  char string[BUFSIZ];
  double aNumber = 0.0;

  printf("%s", prompt);
  gets(string);
  sscanf(string, "%lf", &aNumber);
  return aNumber;
}
```

An example of a program run is

```
Celsius? 36.9
Fahrenheit equivalent is 98.42
```

The variables *celsius* and *fahrenheit* are defined to be of type *double* in the line

```
double celsius, fahrenheit;
```

The effect of the statement

```
celsius = doubleNumberRead("Celsius? ");
```

is to place the value entered by the user into *celsius*. The next statement calculates the equivalent value in fahrenheit.

```
fahrenheit = 9.0 * celsius / 5.0 + 32.0;
```

Notice that the constant values, namely 9.0, 5.0 and 32.0, all contain a decimal point (and no suffix). This identifies them as being constants of type *double*.

The value stored in fahrenheit is displayed by the line

```
printf("Fahrenheit equivalent is %0.2f \n", fahrenheit);
```

The conversion specification is *%0.2*f. It specifies that the value stored in *fahrenheit* is to be displayed with two digits after the decimal point. To display a *float* (or *double*) value with, say, four digits after the decimal point we would write *%0.4f* as the conversion specification. The number following the decimal point in the conversion specification specifies the number of digits to be printed after the decimal point.

Now look at the *sscanf* statement in the *doubleNumberRead* function.

```
sscanf(string, "%lf", &aNumber);
```

Here, the conversion specification is *%lf* (letter ell followed by f). The letter *l* specifies that the value in string is to be converted to a *double* rather than a *float*. This conversion specification is not the same as the one used in *printf*; in *printf* we would use *%f* to print a *double* number value.

We can divide one *double* number by another and find the remainder by using the *fmod* function. For example, to find the remainder obtained when 7.00 is divided by 4.00

```
fmod(7.00, 4.00) == 3.00
```

fmod is defined in the *math* library.

We can divide one *double* number by another, to give an answer without a remainder and without a fractional decimal part, by using the *floor* function. For example

```
floor(7.00 / 4.00) == 1.00
```

The floor of a number of type *double* is the largest integer (expressed as a *double*) which is less than the number. For example,

`floor(16.2)=16.0`, `floor(16.8)=16.0` and `floor(-16.2)=-17.0`.

The following program shows how *fmod* and *floor* may be used.

```
/* program 2.9 - converts pence into pounds and pence. */

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

double doubleNumberRead(char promp[]);

int main()
{
  double pounds, pence;
  pence = doubleNumberRead("Pence? ");
  pounds = floor(pence / 100.00);
  pence = fmod(pence, 100.00);
  printf("That is %0.0f pounds and %0.0f pence.\n",
               pounds, pence);
  return 0;
}

double doubleNumberRead(char prompt[])
{
  char string[BUFSIZ];
  double aNumber = 0.0;
  printf("%s", prompt);
  gets(string);
  sscanf(string, "%lf", &aNumber);
  return aNumber;
}
```

An example of a program run is

```
Pence? 204
That is 2 pounds and 4 pence
```

2.7 Mixed Number Type Arithmetic

In each calculation shown so far, only variables and constants of the same type have been used. For example, in program 2.8 we find

fahrenheit = 9.0 * celsius / 5.0 + 32.0;

fahrenheit and *celsius* are variables of type *double*, and the constants 9.0, 5.0 and 32.0 are also of type *double*. This is not always convenient. We often want to use mixed number types in calculations. For example, suppose we want to calculate the number of times a heart beats in a lifetime. We multiply the number of times it beats in a minute by 60 to obtain the number of times it beats in an hour. We multiply the beats per hour by 24 to obtain the number of times it beats in a day. We multiply beats per day by 365.25 to obtain the number of times it beats in a year. (The 0.25 takes leap years into account.) We multiply beats per year by the number of years in a lifetime. The calculation involves *long unsigned ints*, *unsigned ints* and a *double precision float*.

beatsInALifetime	=	beatsPerMinute	*	60u	*	24u	*	365.25	*	years
long unsigned int		unsigned int		unsigned int		unsigned int		double		unsigned int

Each arithmetic type has an upper limit to the value that can be stored. If we list the arithmetic data types introduced so far in the order of their maximum values (largest first) we obtain

double	largest
float	↕
unsigned long int	
unsigned int	
int	smallest

Figure 2.6 - the relative sizes of the number types.

We can promote a value of a smaller type (lower down in the list) to a larger type value (higher up in the list) with no problem. For example, an unsigned value such as 56 can fit into a variable of type *double*; here, the value stored would be 56.0000000000.

But if we demote a value of a larger type to a smaller type, then information is lost. For example, if we try to store a double value such as 38000.75 in an *int* variable then we loose information because the largest value that can be stored there is 32767; in this example, we have no way of knowing exactly what value will be stored. And even if a *double* value was not too big to be stored in an *int* variable, the digits after the decimal point would be lost.

Looking at the calculation to work out the number of heartbeats in a lifetime, we can promote each unsigned value to *double* (to match the 365.25) so that the result of the calculation is a *double* value. But if we assign a *double* value to the *long unsigned int* variable *beatsInALifetime* some accuracy in the result will be lost. In this case, we can live with the inaccuracy. But there may well be circumstances where such inaccuracy cannot be tolerated. A solution would be to perform all calculations using values of type *double*. But this wasts space in memory. A larger data type value requires more space for storage than a smaller one and this is not always convenient.

We can rely on C to automatically promote a smaller type value to a larger type in an arithmetic calculation. But we have to explicitly coerce a larger type value into a smaller type. We do this with a cast operator.

In the following example the cast operator is *(long unsigned int)*.

```
beatsInALifetime =
(long unsigned int)beatsPerMinute * 60U * 24U * 365.25 * years
```

A cast operator is a named type enclosed within brackets. It temporarily changes the type of the expression which follows it to the named type. Why should this be done? Because the presence of the *double* constant 365.25 causes the other terms to be promoted to *double* before the calculation takes place and we need to store the result of the calculation in a variable of type *unsigned long int*.

Here is the complete program which inputs the number of times a heart beats in one minute and the number of years in a lifetime, and then calculates and outputs the number of times it beats in that lifetime.

```
/* program 2.10 - calculates the number of times a heart beats
                  in a lifetime.
*/
#include <stdio.h>

unsigned int unsignedNumberRead(char prompt[]);

int main()
{
  unsigned int beatsPerMinute, years;
  long unsigned int beatsInALifetime;
  beatsPerMinute = unsignedNumberRead("Beats per minute? ");
  years = unsignedNumberRead("For how many years? ");
  beatsInALifetime =
         (long unsigned int)beatsPerMinute * 60U * 24U *
          365.25 * years;
  printf("The number of beats in a lifetime is %lu \n",
          beatsInALifetime);
  return 0;
}

unsigned int unsignedNumberRead(char prompt[])
{
  char string[BUFSIZ];
  unsigned aNumber = 0u;
  printf("%s", prompt);
  gets(string);
  sscanf(string, "%u", &aNumber);
  return aNumber;
}
```

An example of a program run is

```
Beats per minute? 70
For how many years? 70
The number of beats in a lifetime is 2577204000
```

2.12 Documentation

The names chosen for variables should be descriptive; they should reflect the purpose of the object they name. Descriptive variable names help to make programs easier for (human) readers to understand. Variable names should be written in lower case letters and digits, except the first letter in each word which makes up the name; these should be written in upper case. However, the first letter in a variable name should be a lower case letter.

2.13 Programming Principles

- remember that a variable is a container; it is a location in memory which has a name and which contains a value
- choose appropriate types for the values to be used in a calculation
- work out each each step of a calculation using pencil and paper (and not a calculator - why?) so that you can check whether your program is giving you the correct answers
- take extra care when using values of different types in a calculation
- remember that dividing by zero is NEVER ALLOWED; ensure that division by zero cannot happen.

Exercise 2.4

1 Write a program which will convert a whole number weight in pounds to a weight in kilograms. (One pound weight equals 0.45 kilograms.)

2 Write a program which will input an amount of money (that is, a bill) for goods or services, add VAT at 17.5% and output the bill including VAT.

3 The maximum mortgage usually allowed on a house is 95% of its value. Write a program which will input the value of a house and output 95% of that value.

4 Write a program which will input hours worked and hourly rate of pay, calculate and output gross pay, tax at 33% of gross pay and net pay.

5 Write a program which will input a whole number of hours and convert it into days and hours eg 27 hours input converts to one day and three hours output.

6 The owner of a restaurant needs a simple program to calculate running costs. The cost of feeding a group of people depends on how many there are (the greater the number of people, the greater the cost of the food they consume) and on the overheads (such as cooks wages, rent and insurance) which have to be paid even if there are no people to feed. The formula is

costOfFeeding = numberOfPeople x costOfFoodPerPerson + overheads

Write a program which will allow the user to enter number of people, the cost of food per person and the overheads, and which will calculate and display the total cost of feeding the group.

7 Write a program which will input an amount of money and output the maximum number of fifty pound notes that may be contained in the amount of money input. For example, £175 contains at most three fifty-pound notes.

8 Write a program which will input an amount of money and output the money left over when the maximum number of twenty-pound notes have been removed from the amount input. For example, £3 remains after removing four twenty-pound notes from £83.

9 A program is required which will tell a clerk how to make up an employee's pay packet. Fifty, twenty, ten and five pound notes are to be used together with pound coins. Pence are to be held over and accumulated until they make up over one pound. For example, if an employee's wage is £193.56 then the pay packet would contain three fifty- pound notes, two twenty-pound notes and three one-pound coins; the 56 pence will be added to next week's pay. Write a program which will input an employee's wage as a whole number of pounds, and output the number of notes of each denomination, together with pound coins, to make up the employee's pay packet.

10 A run-time error occurs during program execution when, for example, an attempt is made to divide by zero or an attempt is made to store a value in a variable which is too small to contain it. Write and run a program which demonstrates a run-time error.

11 A flea weighs about 0.00005 kilograms and, when jumping, exerts a force of about 150 times its own weight in moving its body upwards a distance of about 0.0015 metres before its legs lose contact with the surface on which it was resting. Write a program which will calculate the height jumped according to the formula

$$\text{HeightJumped} = \frac{\text{Force x Distance}}{\text{Weight x 10}}$$