Short C Tutorial

Why would you want to learn C rather than COBOL (gag me) or FORTRAN?

1. C is the most widely used programming language.
2. Even though FORTRAN is still in heavy use by Physicists, it is debatable whether it is dominant.
3. If you want to get a computer-related job outside the field of Physics, you really need to know C.
4. C is the first language ported to any new computer architecture. In fact, FORTRAN compilers often emit C code!
5. C will most likely be the language of choice for future generations of multiprocessor computers (and hence if you are in the business of writing numerically intensive programs, you will need to know C).
6. The C compiler will find errors in your source code that FORTRAN will ignore. This leads to a shorter program development time, and a greater likelyhood of obtaining a correct program.
7. Many scientific instruments are most easily programmed in C (e.g., National Instruments PC cards come with C library interfaces).
8. C is closer to assembly language, so you can have finer control over what the computer is doing, and thereby make faster programs.
9. There is a free C compiler available (GNU C, gcc), that is of very high quality and that has been ported to numerous machines. (Stop press: GNU now have a FORTRAN compiler, although it is still in the early stages of development).
10. UNIX is written in C, so it is easier to interface with UNIX operating systems if you write in C.
11. Once you have mastered C, you will find PERL easy to learn, and PERL is an extremely useful language to know, but that is another story...

Some problems with C

- The language was designed with writing operating systems in mind, and so it was not purpose-built for numerical work (e.g., until recently, all floating point arithmetic in C was done in double precision). There are still major problems, e.g., the ANSI standard only requires trig functions to be provided in double-precision versions (although many compilers, including cc on newt, do provide them in single-precision as well).
- There is no support for complex numbers.
- Handling multi-dimensional arrays, and understanding pointers, are difficult when you first encounter them.
- It is easier to write completely opaque code in C than it is in FORTRAN (although, one rarely sees example of spaghetti-C, whereas spaghetti-FORTRAN is the norm).
- You have to explicitly declare every variable that you use. This is actually not a problem at all! It is a feature. You should have been using IMPLICIT NONE in your FORTRAN programs anyway, and if you haven't, I wouldn't trust any of your results!
- It is possible to bypass many of the protective features of the language, and get yourself into trouble (alternatively, this can be seen as an advantage, since the language imposes few limitations on what you can do).

A brief history of C

C evolved from a language called B, written by Ken Thompson at Bell Labs in 1970. Ken used B to write one of the first implementations of UNIX. B in turn was a decendant of the language BCPL (developed at Cambridge (UK) in 1967), with most of its instructions removed.

So many instructions were removed in going from BCPL to B, that Dennis Ritchie of Bell Labs put some back in (in 1972), and called the language C.

The famous book *The C Programming Language* was written by Kernighan and Ritchie in 1978, and was the definitive reference book on C for almost a decade.

The original C was still too limiting, and not standardized, and so in 1983 an ANSI committee was established to formalise the language definition.

It has taken until now (ten years later) for the ANSI standard to become well accepted and almost universally supported by compilers.

Recommended books on C

This section is incomplete...

The simplest possible C program

Here it is (it doesn't do anything useful):

```
main () {
}
```

Hello world

Here is the generic ''Hello world'' program:

```
#include <stdio.h>
main () {
  printf ("hello world\n");
}
```

Notes:

- `#include` is a *pre-processor directive*, i.e., it is interpreted by a program call the 'pre-processor', cpp (any line starting with the # character is interpreted by cpp). In this case, `#include` simply *includes* the file /usr/include/stdio.h into the source code, before passing it on to the C compiler. This is similar to the `INCLUDE` statement in FORTRAN.
- The angle brackets around `stdio.h` indicate that this is an include-file provided by the C compiler itself (if you wanted to include your own file, then use double quotes instead).
- The reason that we include `/usr/include/stdio.h` is that it contains a *function prototype* for the function `printf` which we call at line 3. Function prototypes are used by ANSI-standard C to check that you are calling the function with the right number and type of arguments. This is a huge advantage over FORTRAN, which does no checking, and eliminates lots of errors.
- `main ()` is a *function*. Every C program must contain at least one function definition called `main`. The fact that `main ()` is followed by a curly bracket means that this is the *definition* of `main`. (To call a function, you give its name, its arguments (in parentheses), and then a semicolon).
- `printf ("hello world\n");` is an example of a function being called. `"hello world\n"` is the argument (in this case a character string).
- A semicolon is used to delimit statements in C. This is one of the most confusing things about C for the FORTRAN programmer. We will soon learn some rules for when and where to use semicolons.

Note a fundamental difference with FORTRAN: C uses a *function* (printf) to produce output on the terminal, whereas FORTRAN has specific statements (WRITE, FORMAT) to accomplish the same thing.

Note also the strange character sequence '\n' in the string constant `"hello world\n"`. '\n' is called a *newline character*, and is regarded as a single character in C (i.e., it occupies one byte of storage).

Character constants, escape sequences, and string constants

A *character constant* in C is a single character (or an escape sequence such as \n) enclosed in single quotes, e.g., 'A'. (In fact, you can use multiple characters within the quotes, but the result is not defined in the ANSI standard).

The value of a character constant is the numeric value of the character in the computer's character set (e.g., 'A'

has the value 65). In 99.99% of cases this is the ASCII character set, but this is not defined by the standard!

But how do you represent a character such as a single quote itself? The answer is to use an *escape sequence*.

For reference, here is a program to print out all the special *escape sequences*:

```
/* A program to print out all the special C escape sequences  */

/* Michael Ashley / UNSW / 04-May-1994                        */

#include <stdio.h>      /* for printf definition       */

main () {

  printf ("audible alert (bell) BEL  \\a   %d\n" , '\a');
  printf ("backspace            BS   \\b   %d\n" , '\b');
  printf ("horizontal tab       HT   \\t   %d\n" , '\t');
  printf ("newline              LF   \\n  %d\n"  , '\n');
  printf ("vertical tab         VT   \\v  %d\n"  , '\v');
  printf ("formfeed             FF   \\f  %d\n"  , '\f');
  printf ("carriage return      CR   \\r  %d\n"  , '\r');
  printf ("double quote         \"    \\\"  %d\n", '\"');
  printf ("single quote         \'    \\\'  %d\n", '\'');
  printf ("question mark        ?    \\?  %d\n"  , '\?');
  printf ("backslash            \\    \\\\  %d\n", '\\');
}
```

And here is the output it produces, when compiled with gcc on newt:

```
audible alert (bell) BEL  \a   7
backspace            BS   \b   8
horizontal tab       HT   \t   9
newline              LF   \n  10
vertical tab         VT   \v  11
formfeed             FF   \f  12
carriage return      CR   \r  13
double quote         "    \"  34
single quote         '    \'  39
question mark        ?    \?  63
backslash            \    \\  92
```

Note: this program actually produces the wrong output when used with cc on newt!

EXERCISE: *try compiling the program with gcc and cc and determine which of the compilers produce the correct result.*

In addition, you can specify any 8-bit ASCII character using either \ooo or \xhh where 'ooo' is an octal number (with from 1 to 3 digits), and 'xhh' is a hexadecimal number (with 1 or 2 digits). For example, \x20 is the ASCII character for SPACE.

The above program also shows how to add *comments* to your C program.

String constants are a sequence of zero or more characters, enclosed in double quotes. For example, "test", "", "this is an invalid string" are all valid strings (you can't always believe what a string tells you!). String constants are stored in memory as a sequence of numbers (usually from the ASCII character set), and are *terminated by a null byte (\0)*. So, "test" would appear in memory as the numbers 116, 110, 115, 116, 0.

We have already used some examples of strings in our programs, e.g, "hello world\n" was a null-terminated character string that we sent to the `printf` function above.

Comments

Comments are delimited by '/*' and '*/'. Any text between the first '/*' and the next '*/' is ignored by the compiler, irrespective of the position of line breaks. Note that this means that **comments do not nest.**. Here are some examples of the valid use of comments:

```
/* This is a comment */
                     /* here is another one
                                that spans two lines */
  i = /* a big number */ 123456;
```

Here are some problems that can arise when using comments:

```
  i = 123456; /* a comment starts here
  i = i + 2;     this statement is also part of the comment */
  /* this is a comment /* and so is this */ but this will generate an error */
```

The fact that comments don't nest is a real nuisance if you want to comment-out a whole section of code. In this case, it is best to use pre-processor directives to do the job. For example:

```
  i = 123456;
#if 0
  i = i + 2;
  j = i * i;
#endif
```

Makefiles - the easy way to automate compilation

When developing software, a great deal of time (both your own, and CPU time) can be save by using the UNIX utility `make` (there are similar utilities available on other platforms, e.g., Microsoft has `nmake` for the PC).

The idea behind `make` is that you should be able to compile/link a program simply by typing `make` (followed by a carriage-return). For this to work, you have to tell the computer how to build your program, by storing instructions in a *Makefile*.

While this step takes some time, you are amply repaid by the time savings later on. `make` is particularly useful for large programs that consist of numerous source files: `make` only recompiles the file that need recompiling (it works out which ones to process by looking at the dates of last modification).

Here is an example of a simple `Makefile`:

```
test: test.o; cc test.o -o test
```

Notes:

1. A `Makefile` consists of a series of lines containing *targets*, *dependencies*, and *shell commands* for updating the targets.
2. In the above example,
   - `test` is the target,
   - `test.o` is the dependency (i.e., the file that `test` depends on), and
   - `cc test.o -o test` is the shell command for making `test` from its dependencies.
3. Important note: `test` is probably a bad name for a program since it conflicts with the UNIX shell command of the same name, hence if you type "test" it will run the UNIX command rather than your program. To get around this problem, simply use "./test" (if "test" is in your current directory), or rename the program. It is always a good idea to use the leading "./" since it avoids a lot of subtle problems that can be hard to track down. Also, don't put a "." in your PATH, this is a security risk.

Makefiles can get *much* more complicated than this simple example. Here is the next step in complexity, showing the use of a macro definition and a program that depends on multiple files.

```
OBJS = main.o sub1.o sub2.o
main: $(OBJS); cc $(OBJS) -o main
```

It is well worth learning a little bit about `make` since it can save a lot of time!

How to compile and run a C program on newt

1. Create a new directory for the program (not essential, but it helps to be organised).
2. Use whatever editor you want to generate the program (make sure the program file has a suffix of '.c').
3. Construct a Makefile for the program (not essential, but worth doing).
4. Type `make` to compile the program (if you don't have a Makefile, you will need to type the compiler command yourself).

5. Run the program by typing its name.
6. Locate and fix bugs, then go to step 4.

Here is a complete example of the above process for the ‘‘Hello world’’ program:

```
cd
mkdir hello
cd hello
cat > hello.c
#include <stdio.h>
main () {
  printf ("hello world\n");
}
^D
cat > Makefile
hello: hello.o; cc hello.o -o hello
^D
make
./hello
```

EXERCISE: *try doing this yourself with some of the example programs later in these notes.*

C compilers available on newt

cc
: The MIPS C compiler (recommended, used for this course).

gcc
: The GNU C compiler (recommended).

A program to give information on C data types

Like other programming languages, C has a variety of different data types. The ANSI standard defines the data types that must be supported by a compiler, but it doesn’t tell you details such as the range of numbers that each type can represent, or the number of bytes of storage occupied by each type. These details are implementation dependent, and defined in the two system include-files "limits.h" and "float.h". To find out what the limits are, try running the following program with your favourite C compiler.

```
/* A program to print out various machine-dependent constants */
/* Michael Ashley / UNSW / 04-May-1994                        */

#include <stdio.h>      /* for printf definition       */
#include <limits.h>     /* for CHAR_MIN, CHAR_MAX, etc */
#include <float.h>      /* for FLT_DIG, DBL_DIG, etc   */

main () {

  printf ("char           %d bytes %d to %d \n",   sizeof(char          ), CHAR_MIN, CHAR_MAX  );
  printf ("unsigned char  %d bytes %d to %d \n",   sizeof(unsigned char ), 0       , UCHAR_MAX );
  printf ("short          %d bytes %hi to %hi \n", sizeof(short         ), SHRT_MIN, SHRT_MAX  );
  printf ("unsigned short %d bytes %hu to %hu \n", sizeof(unsigned short), 0       , USHRT_MAX );
  printf ("int            %d bytes %i to %i \n",   sizeof(int           ), INT_MIN , INT_MAX   );
  printf ("unsigned int   %d bytes %u to %u \n",   sizeof(unsigned int  ), 0       , UINT_MAX  );
  printf ("long           %d bytes %li to %li \n", sizeof(long          ), LONG_MIN, LONG_MAX  );
  printf ("unsigned long  %d bytes %lu to %lu \n", sizeof(unsigned long ), 0       , ULONG_MAX );
  printf ("float          %d bytes %e to %e \n",   sizeof(float         ), FLT_MIN , FLT_MAX   );
  printf ("double         %d bytes %e to %e \n",   sizeof(double        ), DBL_MIN , DBL_MAX   );
  printf ("precision of float   %d digits\n", FLT_DIG);
  printf ("precision of double %d digits\n",  DBL_DIG);
}
```

Notes:

- `sizeof` looks like a function, but it is actually a built-in C *operator* (i.e., just like +,-,*). The compiler replaces `sizeof(data-type-name)` (or, in fact, `sizeof(variable)`) with the number of bytes of storage allocated to the data-type or variable.
- The ’unsigned’ data types are useful when you are refering to things which are naturally positive, such as the number of bytes in a file. They also give you a factor of two increase in the largest number that you can represent.

Most of the time you don’t need to worry about how many bytes are in each data type, since the limits are usually OK for normal programs. However, a common problem is that the "int" type is only 2-bytes long on most PC compilers, whereas on UNIX machines it is usually 4-bytes.

Here is the output of the preceeding program when run on newt, using cc:

```
char           1 bytes -128 to 127
unsigned char  1 bytes 0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           4 bytes -2147483648 to 2147483647
unsigned long  4 bytes 0 to 4294967295
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
precision of float   6 digits
precision of double 15 digits
```

Constants

We have already seen how to write character constants and strings. Let’s now look at other types of constants:

```
int            123, -1, 2147483647, 040 (octal), 0xab (hexadecimal)
unsigned int   123u, 2147483648, 040U (octal), 0X02 (hexadecimal)
long           123L, 0XFFFFl (hexadecimal)
unsigned long  123ul, 0777UL (octal)
float          1.23F, 3.14e+0f
double         1.23, 2.718281828
long double    1.23L, 9.99E-9L
```

Note:

- Integers are automatically assumed to be of the smallest type that can represent them (but at least an ’int’). For example, 2147483648 was assumed by the compiler to be an ’unsigned int’ since this number is too big for an ’int’. Numbers too big to be an ’unsigned int’ are promoted to ’long’ or ’unsigned long’ as appropriate, although on the DECstation these types do not hold larger numbers, so an error will result.
- An integer starting with a zero is assumed to be octal, unless the character after the zero is an ’x’ or ’X’, in which case the number in hexadecimal.
- Unsigned numbers have a suffix of ’u’ or ’U’.
- Long ’int’s or ’double’s have a suffix of ’l’ or ’L’ (it is probably better to use ’L’ so that it isn’t mistaken for the number one).
- Floating-point numbers are assumed to be ’double’s unless they have a suffix of ’f’ or ’F’ for ’float’, or ’l’ or ’L’ for ’long double’.

It pays to be very careful when specifying numbers, to make sure that you do it correctly, particularly when dealing with issues of precision. This is often neglected in FORTRAN. For example, consider the following program:

```
      real*8 r
      r = 1.0 + 0.2
      r = r - 1.2
      write (*,*) r
      end
```

When compiled with ’f77’ on newt, this gives the result ’4.7683715864721423E-08’, not zero as you might expect. The reason for this is that ’1.0’ and ’0.2’ are single precision numbers by default, and so the addition is only done to this precision. The number ’1.2’ is converted into binary with double precision accuracy, and so is a different number from ’1.0 + 0.2’.

Interestingly, the above program gives the result ’0.0000000000000000E+00’ when compiled with ’f772.1’ on newt, and ’4.768371586472142E-08’ when compiled with ’f77’ on the CANCES HP cluster.

The correct way to write this program is as follows:

```
      real*8 r
      r = 1.0D0 + 0.2D0
      r = r - 1.2D0
      write (*,*) r
      end
```

Here is the equivalent program written in C:

```
#include <stdio.h>
main () {
  double r;
  r = 1.0 + 0.2;
  r = r - 1.2;
  printf ("%22.16e\n", r);
}
```

In this case the result is '0.0000000000000000e+00', but this isn't really a fair comparison with our original FORTRAN program since floating point numbers are assumed to be 'double' in C, not 'real*4' as in FORTRAN. So let's go back to using 'float's instead:

```
#include <stdio.h>
main () {
  double r;
  r = 1.0F + 0.2F;
  r = r - 1.2F;
  printf ("%22.16e\n", r);
}
```

Now the program generates '-4.4703483581542969e-08' when compiled with 'cc' on newt, and yet it gives '0.0000000000000000e+00' when compiled with 'gcc', interesting...

The lesson to be learnt here is *when writing constants, always think carefully about what type you want them to be, and use the suffixes 'U', 'L', and 'F' to be explicit about it.* It is not a good idea to rely on the compiler to do what you expect. Don't be surprised if different machines give different answers if you program sloppily.

Conversion between integers and floating point numbers

In C, as in FORTRAN, there are rules that the compiler uses when a program mixes integers and floating point numbers in the same expression. Let's look at what happens if you assign a floating point number to an integer variable:

```
#include <stdio.h>
main () {
  int i, j;
  i =  1.99;
  j = -1.99;
  printf ("i = %d; j = %d\n", i, j);
}
```

This program produces the result 'i = 1; j = -1'. Note that the floating point numbers have been *truncated* when converted to integers (FORTRAN does the same thing).

When converting integers to floating-point, be aware that a 'float' has fewer digits of precision than an 'int', even though they both use 4 bytes of storage (on newt). This can result in some strange behaviour, e.g.,

```
#include <stdio.h>
main () {
  unsigned int i;
  float f;
  i = 4294967295;       /* the largest unsigned int */
  f = i;                /* convert it to a float    */
  printf ("%u %20.13e %20.13e\n", i, f, f - i);
}
```

This program produces the following output when compiled with 'cc':

4294967295 4.2949672960000e+09 1.0000000000000e+00

and this output when compiled with 'gcc':

4294967295 4.2949672960000e+09 0.0000000000000e+00

Curiouser and curiouser... It appears that what is happening is that 'cc' is doing the calculation 'f - i' as a 'double', i.e., 'f' and 'i' are converted to type 'double', and then subtracted. Whereas 'gcc' is converting 'i' to type 'float' (just as was done with 'f = i'), and hence the subtraction results in zero. To test this hypothesis, you can force 'cc' to use a 'float' conversion by putting a *type-cast operator* before 'i'. Here it is

```
#include <stdio.h>
main () {
  unsigned int i;
  float f;
  i = 4294967295;       /* the largest unsigned int */
  f = i;                /* convert it to a float    */
  printf ("%u %20.13e %20.13e\n", i, f, f - (float)i);
}
```

This program now gives the same results when used with 'cc' or 'gcc' (i.e., zero). Incidentally, 'gcc's behaviour without the '(float)' agrees with the ANSI standard.

Note the use of the type-cast operator '(float)'. This converts the number or variable or parethesised expression immediately to its right, to the indicated type. It is a good idea to use type-casting to ensure that you leave nothing to chance.

Operators

C has a rich set of *operators* (i.e., things like + - * /), which allow you to write complicated expression quite compactly (unlike FORTRAN which requires function calls to duplicate many of the C operators).

Unary operators

Unary operators are operators that only take one argument. The following list will be confusing when you first see it, so just keep it mind for reference later on.

Some of the operators we have already seen (e.g., 'sizeof()'), others are very simple (e.g., +, -), others are really neat (e.g., ~, !), others are useful for adding/subtracting 1 automatically (e.g., ++i, --i, i++, i--), and the rest involve pointers and addressing, which will be covered in detail later.

```
sizeof(i)       the number of bytes of storage allocated to i
+123            positive 123
-123            negative 123
~i              one's complement (bitwise complement)
!i              logical negation (i.e., 1 if i is zero, 0 otherwise)
*i              returns the value stored at the address pointed to by i
&i              returns the address in memory of i
++i             adds one to i, and returns the new value of i
--i             subtracts one from i, and returns the new value of i
i++             adds one to i, and returns the old value of i
i--             subtracts one from i, and returns the old value of i
i[j]            array indexing
i (j)           calling the function i with argument j
i.j             returns member j of structure i
i->j            returns member j of structure pointed to by i
```

Binary operators

Binary operators work on two operands ('binary' here means 2 operands, not in the sense of base-2 arithmetic).

Here is a list. All the usual operators that you would expect are there, with a whole bunch of interesting new ones.

```
+          addition
-          subtraction
*          multiplication
/          division
```

```
%         remainder  (e.g., 2%3 is 2), also called 'modulo'
<<        left-shift (e.g., i<<j is i shifted to the left by j bits)
>>        right-shift
&         bitwise AND
|         bitwise OR
^         bitwise exclusive-OR
&&        logical AND (returns 1 if both operands are non-zero; else 0)
||        logical OR (returns 1 if either operand is non-zero; else 0)
<         less than (e.g., i<j  returns 1 iff i is less than j)
>         greater than
<=        less than or equal
>=        greater than or equal
==        equals
!=        does not equal
?         conditional operator, explained later...
```

Note:

- Truth and falsity in C is represented by numbers being non-zero and zero respectively (although logical operators always return 1 or 0).
- Don't make the mistake of using '=' when you meant '=='!
- Note the distinction between bitwise operators and logical operators.

Assignment operators

Assignment operators are really just binary operators. The simplest example is '=', which takes the value on the right and places it into the variable on the left. But C provides you with a host of other assignment operators which make life easier for the programmer.

```
=      assignment
+=     addition assignment
-=     subtraction assignment
*=     multiplication assignment
/=     division assignment
%=     remainder/modulus assignment
&=     bitwise AND assignment
|=     bitwise OR assignment
^=     bitwise exclusive OR assignment
<<=    left shift assignment
>>=    right shift assignment
```

So, for example, 'i += j' is equivalent to 'i = i + j'. The advantage of the assignment operators is that they can reduce the amount of typing that you have to do, and make the meaning clearer. This is particularly noticeable when, instead of a simple variable such as 'i', you have something complicated such as 'position[wavelength + correction_factor * 2]';

The thing to the left of the assignment operator has to be something where a result can be stored, and is known as an 'lvalue' (i.e., something that can be on the left of an '='). Valid 'lvalues' include variables such as 'i', and array expressions. It doesn't make sense, however, to use constants or expressions to the left of an equals sign, so these are not 'lvalues'.

The comma operator

C allows you to put multiple expression in the same statement, separated by a comma. The expressions are evaluated in left-to-right order. The value of the overall expression is then equal to that of the rightmost expression.

For example,

```
Example                          Equivalent to
-------                          -------------
i = ((j = 2), 3);                i = 3; j = 2;
myfunct (i, (j = 2, j + 1), 1);  j = 2; myfunct (i, 3, 1);
```

The comma operator has the lowest precedence, so it is always executed last when evaluating an expression. Note that in the example given comma is used in two distinct ways inside an argument list for a function.

Both the above examples are artifical, and not very useful. The comma operator *can* be useful when used in

'for' and 'while' loops as we will see later.

Precedence and associativity of operators

The *precedence* of an operator gives the order in which operators are applied in expressions: the highest precedence operator is applied first, followed by the next highest, and so on.

The *associativity* of an operator gives the order in which expressions involving operators of the same precedence are evaluated.

The following table lists all the operators, in order of precedence, with their associativity:

```
Operator                                  Associativity
--------                                  -------------

() [] ->> .                               left-to-right
- + ++ -- ! ~ * & sizeof (type)           right-to-left
* / %                                     left-to-right
+ -                                       left-to-right
<< >>                                     left-to-right
< <= > >=                                 left-to-right
== !=                                     left-to-right
&                                         left-to-right
^                                         left-to-right
|                                         left-to-right
&&                                        left-to-right
||                                        left-to-right
?:                                        right-to-left
= += -= *= /= %= &= ^= |= <<= >>=         right-to-left
,                                         left-to-right
```

Note: the + - and * operators appear twice in the above table. The unary forms (on the second line) have higher precedence that the binary forms.

Operators on the same line have the same precedence, and are evaluated in the order given by the associativity.

To specify a different order of evaluation you can use parentheses. In fact, it is often good practice to use parentheses to guard against making mistakes in difficult cases, or to make your meaning clear.

Side effects in evaluating expressions

It is possible to write C expressions that give different answers on different machines, since some aspects of expression-evaluation are not defined by the ANSI standard. This is deliberate since it gives the compiler writers the ability to choose different evaluation orders depending on the underlying machine architecture. You, the programmer, should avoid writing expressions with side effects.

Here are some examples:

```
myfunc (j, ++j); /* the arguments may be the same, or differ by one */
array[j] = j++;  /* is j incremented before being used as an index? */
i = f1() + f2(); /* the order of evaluation of the two functions
                    is not defined. If one function affects the
                    results of the other, then side effects will
                    result */
```

Evaluation of logical AND and OR

A useful aspect of C is that it guarantees the order of evaluation of expressions containing the logical AND (&&) and OR (||) operators: it is always left-to-right, and stops when the outcome is known. For example, in the expression '1 || f()', the function 'f()' will not be called since the truth of the expression is known regardless of the value returned by 'f()'.

It is worth keeping this in mind. You can often speed up programs by rearranging logical tests so that the outcome of the test can be predicted as soon as possible.

Another good example is an expression such as ‘i >= 0 && i <n && array[i] == 0’. The compiler will guarantee that the index into ‘array’ is within legal bounds (assuming the array has ‘n’ elements).

What is a C identifier?

An identifier is the name used for a variable, function, data definition, etc.

Rules for identifiers:

- Legal characters are a-z, A-Z, 0-9, and _.
- Case is significant.
- The first character must be a letter or _.
- Identifiers can be of any length (although only the first 31 characters are *guaranteed* to be significant).
- The following are reserved keywords, and may not be used as identifiers:

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

- Here are some examples of legal identifiers:

```
i
count
NumberOfAardvarks
number_of_aardvarks
MAX_LENGTH
```

Mathematical functions

Calling mathematical functions in C is very similar to FORTRAN, although C doesn’t have FORTRAN’s ability to use generic function calls that are converted to the right type during compilation (e.g., the FORTRAN compiler will select the right version of the SIN routine to match the argument, C requires you to use a different routine for single/double precision).

Prototypes for the math functions are in the system include-file "math.h", so you should put the line

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

in any C source file that calls one of them.

Here is a list of the math functions defined by the ANI standard:

```
sin(x)              sine of x
cos(x)              cosine of x
tan(x)              tan of x
asin(x)             arcsine of x, result between -pi/2 and +pi/2
acos(x)             arccosine of x, result between 0 and +pi
atan(x)             arctan of x, result between -pi/2 and +pi/2
atan2(y,x)          arctan of (y/x), result between -pi and +pi
hsin(x)             hyperbolic sine of x
hcos(x)             hyperbolic cosine of x
htan(x)             hyperbolic tan of x
exp(x)              exponential function
log(x)              natural logarithm
log10(x)            logarithm to base 10
pow(x,y)            x to the power of y  (x**y in FORTRAN)
sqrt(x)             the square root of x (x must not be negative)
ceil(x)             ceiling; the smallest integer not less than x
floor(x)            floor; the largest integer not greater than x
fabs(x)             absolute value of x
ldexp(x,n)          x times 2**n
frexp(x, int *exp)  returns x normalized between 0.5 and 1; the exponent of 2 is in *exp
modf(x, double *ip) returns the fractional part of x; the integral part goes to *ip
```



```
fmod(x,y)           returns the floating-point remainder of x/y, with the sign of x
```

In the above table, ’x’, and ’y’ are of type ’double’, and ’n’ is an ’int’. All the above functions return ’double’ results.

C libraries may also include ’float’ versions of the above. For example, ’fsin(x)’ on newt takes a float argument and returns a float result. Microsoft C does not provide ’float’ versions (presumably because the floating-point accelerator chips do all their work in double precision).

The ‘for’ loop

The basic looping construct in C is the ‘for’ loop.

Here is the syntax of the ‘for’ statement:

```
for (initial_expression; loop_condition; loop_expression) statement;
```

An example will clear this up:

```
for (i = 0; i < 100; i++) printf ("%i\n", i);
```

which simply prints the first 100 integers onto ‘stdout’. If you want to include more that one statement in the loop, use curly brackets to delimit the body of the loop, e.g.,

```
for (i = 0; i < 100; i++) {
  j = i * i;
  printf ("i = %i; j = %i\n", i, j);
}
```

Topics left over from last lecture

How to link with the math library

```
cc -o prog prog.c -lm
```

The ‘-l’ switch stands for ‘library’, which means that the specified library of pre-compiled C routines is searched in order to satisfy any external references from yoru program ‘prog.c’. The library that is searched in this case is ‘libm.a’ and the path that is used for the search is the default library search path, which include ‘/usr/lib’ where ’libm.a’ is found.

To use a library in a directory that is not part of the default library search path, you use the ’-L’ switch. For example, to search the library ’/usr/users/smith/libastro.a’, you would use

```
cc -o prog prog.c -L/usr/users/smith -lastro
```

Note: the order of the switches is important. External references are only searched for in libraries to the right of the reference. So, if you have two libraries that call each other, then you need to do something like the following:

```
cc -o prog prog.c -L/usr/users/smith -llib1 -llib2 -llib1
```

Here is a simple example of calling the math library:

```
#include <stdio.h>
#include <math.h>
main () {
 const double pi = 3.1415926535;
 double e, d = pi/2;
 e = sin(d);
 printf ("The sine of %f is %f\n", d, e);
}
```

This program produces the result:

```
The sine of 1.570796 is 1.000000
```

However, if you leave off the '#include <math.h>' line, you will get

```
The sine of 1.570796 is 4.000000
```

Why, because the default type for an undefined function is 'extern int function();'

Semicolons in compound statements

The last statement in the body of statements in a 'for' loop (or, in fact, in any other compound statement) must be terminated with a semicolon.

For example,

```
for (i = 0; i < 10; i++) {
  x = i * i;
  x += 2;               /* the semicolon is required here */
}                       /* do not use a semicolon here */
```

#include <stdio.h> in the example programs

The example programs I showed last time didn't always have '#include <stdio.h>' at the top. They should have had this line (although they will work without it), since it defines the prototypes of the I/O functions, thereby guarding against errors.

Variables defined within compound statements

You can create variables that are local to a compound statement by declaring the variables immediately after the leading curly bracket.

Variable storage classes

Variables in C belong to one of two fundamental storage classes: 'static' or 'automatic'.

A static variable is stored at a fixed memory location in the computer, and is created and initialised once when the program is first started. Such a variable maintains its value between calls to the block (a function, or compound statement) in which it is defined.

An automatic variable is created, and initialised, each time the block is entered (if you jump in half-way through a block, the creation still works, but the variable is not initialised). The variable is destroyed when the block is exited.

Variables can be explicitly declared as 'static' or 'auto' by using these keywords before the data-type definition. If you don't use one of these keywords, the default is 'static' for variables defined outside any block, and 'auto' for those inside a block.

Actually, there is another storage class: 'register'. This is like 'auto' except that it asks the compiler to try and store the variable in one of the CPU's fast internal registers. In practice, it is usually best not to use the 'register' type since compilers are now so smart that they can do a better job of deciding which variables to place in fast storage than you can.

Const - volatile

Variables can be qualified as 'const' to indicate that they are really constants, that can be initialised, but not altered.

Variables can also be termed 'volatile' to indicate that their value may change unexpectedly during the execution of the program (e.g., they may be hardware registers on a PC, able to be altered by external events).

By using the 'volatile' qualifier, you prevent the compiler from optimising the variable out of loops.

Extern

Variables (and functions) can also be classified as 'extern', which means that they are defined external to the current block (or even to the current source file). An 'extern' variable must be defined once (and only once) without the 'extern' qualifier.

As an example of an 'extern' function, all the functions in 'libm.a' (the math library) are external to the source file that calls them.

An example showing storage class and variable scope

```
#include <stdio.h>
int i;          /* i is static, and visible to the entire program */
extern j;       /* j is static, and visible to the entire program */
static int j;   /* k is static, and visible to the routines in this source
                   file */

void func (void) {          /* i.e., a function that takes no arguments, and
                               doesn't return a value */
  int m = 1;          /* m is automatic, local to this function, and initialised
                         each time the function is called */
  auto int n = 2;     /* n is automatic, local to this function, and initialised
                         each time the function is called */
  static int p = 3;   /* p is static, local to this function, and initialised
                         once when the program is first started up */
  extern int q;       /* q is static, and defined in some external module */

  for (i = 0; i < 10; i++) {
    int m = 10;       /* m is automatic, local to this block, and initialised
                         each time the block is entered */
    printf ("m = %i\n", m);
  }
}
```

Initialisation of variables

A variable is initialised by equating it to a constant expression on the line in which it is defined. For example

```
  int i = 0;
```

'static' variables are initialised once (to zero if not explicitly initialised), 'automatic' variables are initialised when the block in which they are defined is entered (and to an undefined value if not explicitly initialised).

The 'constant expression' can contain combinations of any type of constant, and any operator (except assignment, incrementing, decrementing, function call, or the comma operator), including the ability to use the unary & operator to find the address of static variables.

Here are some valid examples:

```
#include <stdio.h>
#include <math.h>
int i = 0;
int j = 2 + 2;
int k = 2 * (3 << 8) / 3;
int m = (int)(&i + 2);
int p = sizeof(float) * 2;
int q = sizeof(p);
float r = (float)(2 * 3);
main () {
  printf ("i = %i\n", i);
  printf ("j = %i\n", j);
  printf ("k = %i\n", k);
  printf ("m = %i\n", m);
  printf ("p = %i\n", p);
  printf ("q = %i\n", q);
  printf ("r = %f\n", r);
  for (r = 0.0; r < 1.0; r += 0.1) {
```

```
    double s = sin(r);
    printf ("The sine of %f is %f\n", r, s);
  }
}
```

Notes:

- An ‘automatic’ variable can be initialised to any expression, even one using other variables, since the initialisation is done at run-time.
- It is good style to put all your ‘extern’ variables at the top of each source file, or even to put them into a header file.
- Don’t overuse ‘extern’ variables! It is usually better to pass lots of arguments to functions rather than to rely on hidden variables being passed (since you end up with clearer code, and reusable functions).

If statements

```
if (expression)
  statement
else if (expression)
  statement
else if (expression)
  statement
else
  statement
```

Where ‘statement’ is a simple C statement ending in a semicolon, or a compound statement ending in a curly bracket.

Some examples will help:

```
if (i == 6) x = 3;
```

```
if (i == 6) {
  x = 3;
}
```

```
if (i) x = 3;
```

```
if (i)
  x = 3;
else if (i ==1)
  if (j)
    y = 2;
  else              /* NOTE: possible ambiguity here, the compiler uses  */
    y = 3;          /*       the closest if statement that does not have */
else {              /*       an else clause                              */
  x = 4;
  y = 5;
}
```

Break and continue

These two statements are used in loop control.

‘break’ exits the innermost current loop.

‘continue’ starts the next iteration of the loop.

Infinite loops

```
for (;;;) {
  statement;
}

while (1) {
  statement;
}

do {
  statement;
```



```
} while (1);
```

Infinite loops can be useful. They are normally terminated using a conditional test with a ‘break’ or ‘return’ statement.

goto

C does have a ‘goto’ statement, but you don’t need it. Using ‘goto’ is almost always a result of bad programming.

Formatted output: printf description

The format string given to the ‘printf’ function may contain both ordinary characters (which are simply printed out) and conversion characters (beginning with a percent symbol, %, these define how the value of an internal variable is to be converted into a character string for output).

Here is the syntax of a conversion specification:

```
%{flags: - + space 0 #}{minimum field width}{.}{precision}{length modifier}{conversion character}
```

- Flags: ‘-’ means left-justify (default is right-justify), ‘+’ means that a sign will always be used, ‘ ’ prefix a space, ‘0’ pad to the field width with zeroes, ’#’ specifies an alternate form (for details, see the manual!). Flags can be concatenated in any order, or left off altogether.
- Minimum field width: the output field will be at least this wide, *and wider if necessary.*
- ’.’ separates the field width from the precision.
- Precision: its meaning depends on the type of object being printed:
  - Character: the maximum number of characters to be printed.
  - Integer: the minimum number of digits to be printed.
  - Floating point: the number of digits after the decimal point.
  - Exponential format: the number of significant digits.
- Length modifer: ‘h’ means short, or unsigned short; ‘l’ means long or unsigned long; ‘L’ means long double.
- Conversion character: a single character specifying the type of object being printed, and the manner in which it will be printed, according to the following table:

```
Character Type    Result

  d,i     int     signed decimal integer
   o      int     unsigned octal (no leading zero)
  x, X    int     unsigned hex (no leading 0x or 0X)
   u      int     unsigned decimal integer
   c      int     single character
   s      char *  characters from a string
   f      double  floating point [-]dddd.pppp
  e, E    double  exponential [-]dddd.pppp e[=/-]xx
  g, G    double  floating is exponent less than -4, or >= precision
                  else exponential
   p      void *  pointer
   n      int *   the number of characters written so far by printf
                  is stored into the argument (i.e., not printed)
   %              print %
```

Here is an example program to show some of these effects:

```
#include <stdio.h>
main () {
 int i = 123;
 double f = 3.1415926535;
 printf ("i = %i\n", i);
 printf ("i = %o\n", i);
 printf ("i = %x\n", i);
 printf ("i = %X\n", i);
 printf ("i = %+i\n", i);
 printf ("i = %8i\n", i);
 printf ("i = %08i\n", i);
 printf ("i = %+08i\n", i);
 printf ("f = %f\n", f);
 printf ("f = %10.3f\n", f);
```

```
 printf ("f = %+10.3f\n", f);
 printf ("f = %g\n", f);
 printf ("f = %10.6g\n", f);
 printf ("f = %10.6e\n", f);
}
```

Notes:

- One big difference is that FORTRAN will print asterisks in fields that are too small to fit the number, whereas C will expand the field. Be careful here, particularly when reading output generated by another program.
- There are differences between the various implementations of printf. The above should be a subset of the available options. Consult the manual for your particular C compiler to be sure.
- printf is actually an integer function, which returns the number of characters written (or a negative number if an error occurred).

Formatted input: scanf description

Input in C is similar to output: the same conversion characters are used. The main difference is that you use a routine called 'scanf', and you must pass the *addresses of the variables you want to change, not their values*.

For example:

```
  scanf ("%d", &i);         /* reads an integer into 'i' */
  scanf ("%i", &i);         /* reads an integer (or octal, or hex) into 'i' */
  scanf ("%f %i", &f, &i);  /* reads a double followed by an integer */
```

scanf is actually an integer function, which returns the number of input items assigned (or EOF if the end-of-file is reached or an error occurred).

The ampersand character '&' is a unary operator that returns the address of the thing to its right. Recall that a C function can not alter the value of its arguments (but there is nothing stopping it altering that value that is pointed to by one of its arguments!).

User input, a real program example

```
/* interest.c, by Tom Boutell, 6/27/93.
   Updated 6/29/93 to support user input. */

/* This program calculates the balance of a savings or loan account after
   a number of years specified by the user, with an interest rate,
   monthly payment, initial balance and rate of compounding specified
   by the user. */

/* Get standard input and output functions */
#include <stdio.h>

/* Get standard math functions */
#include <math.h>

int main() {
  /* Initial balance (money in account). Since this value can
    have a fractional part, we declare a float (floating point)
    variable to store it. */
  float initial_balance;

  /* Rate of interest, per year (also a floating point value) */
  float interest;

  /* Number of times interest is compounded each year (interest periods)
    (thus 1.0 is annually, 365.0 is daily) */
  float frequency;

  /* Time in years */
  float years;

  /* Total interest periods. This cannot have a fractional part,
    so we declare an integer (no fractional part) variable to store it. */
  int interest_periods;

  /* Current balance. (We store this in a separate place form the
    initial balance, so we will still be able to tell how much money
```



```
    we started with when the calculation is finished.) */

  float balance;

  /* Counter of interest periods */
  int i;

  /* Monthly deposit (negative values are permitted) */
  float deposit;

  /* Flag: when this is set true (nonzero), the user is finished */

  int done;

  /* User input: analyze again? */

  int again;

  /* Initially, of course, we are *not* finished. (C does NOT automatically
    set variables to zero. Making this assumption is a common mistake
    among new programmers.) */

  done = 0;

  /* Loop until done. */

  while (!done) {
    /* Fetch starting values from user */
    printf("Initial balance: ");
    scanf("%f", &initial_balance);

    printf("Interest rate (example: .05 for 5 percent): ");
    scanf("%f", &interest);

    printf("Number of compoundings per year (12 = monthly, 365 = daily): ");
    scanf("%f", &frequency);

    printf("Monthly deposit (enter negative value for loan payment): ");
    scanf("%f", &deposit);

    printf("Number of years (examples: 1, 5, .5): ");
    scanf("%f", &years);

    /* Actual logic begins here. */

    /* Calculate number of interest periods. */
    interest_periods = frequency * years;

    /* Set working balance to begin at initial balance. */
    balance = initial_balance;

    /* Loop through interest periods, increasing balance */
    for (i=0; (i
```

EXERCISE: *write a program in C to make a nicely-formatted table of sines, cosines, and tangents, of all the integral degree values between two numbers entered by the user.*

The on-line manual pages

man, xman, dxbook.

The simple debugger 'dbx'

```
cc -g -O0 main.c

dbx a.out



quit[!]                         - quit dbx
run arg1 arg2 ... { f1 }& f2    - begin execution of the program
stop at {line}                  - suspend execution at the line
[n] cont {signal}               - continue with signal
return                          - continue until the current procedure returns
```

```
print {exp} ...                 - print the value of the expressions
printf "string", exp, ...      - print expressions using format string(C)
where [n]                       - print currently active procedures (stack trace)
status                          - print trace/stop/record's in effect
func {proc}                     - move to activation level of {proc}
{exp}[/ | ?]{count}{format}     - display count number of formatted memory items
file {file}                     - change current file to file
list {exp1},{exp2}              - list source lines from {exp1} to {exp2}
list {exp}:{int}                - list source lines at {exp} for {int} lines
sh {shell command}              - perform shell command
```

Arrays

Arrays are declared in C as follows (for example):

```
int counts[100];
float temperature[1024];
```

In this example, 'count' is an array that can hold 100 integers, and 'temperature' is an array that can hold 1024 floats.

So far so good. The major departure from FORTRAN is that the first element in a C array is element number 0 (rather than 1 as in FORTRAN). While this may be confusing to die-hard FORTRAN programmers, it is really a more natural choice. A side-effect of this choice is that the last element in an array has an index that is *one less that the declared size of the array*. This is a source of some confusion, and something to watch out for.

Initialising arrays

To initialise an array, specify the initial values in a list within curly brackets. For example:

```
int primes[100] =
  {2,   3,   5,   7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47,  53,  59,  61,
   67,  71,  73,  79,  83,  89,  97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
  149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
  227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,
  307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383,
  389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
  467, 479, 487, 491, 499, 503, 509, 521, 523, 541};

float temp[1024] = {5.0F, 2.3F};

double trouble[] = {1.0, 2.0, 3.0};
```

In this example, 'primes' is initialised with the values of the first 100 primes (check them!), and the first two elements (temp[0] and temp[1]) of 'temp' are initialised to 5.0F and 2.3F respectively. The remaining elements of 'temp' are set to 0.0F. Note that we use the trailing 'F' on these numbers to indicate that they are floats, not doubles.

The array 'trouble' in the above example contains three double numbers. Note that its length is not explicitly declared, C is smart enough to work the length out.

Static arrays that are not explicitly initialised are set to zero. Automatic arrays that are not explicitly initialised, have undefined values.

Multidimensional arrays

Multidimensional arrays are declared and referenced in C by using multiple sets of square brackets. For example,

```
int table[2][3][4];
```

'table' is a 2x3x4 array, with the rightmost array subscript changing most rapidly as you move through memory (the first element is 'table[0][0][0]', the next element is 'table[0][0][1]', and the last element is 'table[1][2][3]'.

When writing programs that use huge arrays (say, more than a few megabytes), you should be very careful to

ensure that array references are as close as possible to being consecutive (otherwise you may get severe swapping problems).

Initialisation of multidimensional arrays

This is best demonstrated with an example:

```
int mat[3][4] = {
                  { 0, 1, 2},
                  { 3, 4, 5},
                  { 6, 7, 8}
                }
```

Note that I have only initialised a 3x3 subset of the 3x4 array. The last column of each row will have the default initialisation of zero.

Character arrays

Arrays can be of any type. Character arrays hold a single character in each element. In C, you manipulate character strings as arrays of characters, and operations on the strings (such as concatenation, searching) are done by calling special libary functions (e.g., strcat, strcmp).

Note that when calling a string-manipulation function, the end of the string is taken as the position of the first NUL character (0) in the string.

Character arrays can be initialised in the following ways:

```
char str[] = {'a', 'b', 'c'};
char prompt[] = "please enter a number";
```

In the example, 'str' has length 3 bytes, and 'prompt' has length 22 bytes, which is one more than the number of characters in "please enter a number", the extra character is used to store a NUL character (zero) as an indication of the end of the string. 'str' does not having a trailing NUL.

Pointers

If there is one thing that sets C apart from most other languages, it is the use of pointers. A 'pointer' is a variable containing the address of a memory location.

Suppose 'p' is a pointer, then '*p' is the thing which 'p' points to.

Suppose 'i' is being pointed at by 'p', then 'i' and '*p' are the same thing, and 'p', being equal to the address of 'i', is equal to '&i' (remember, '&' is the unary address operator).

Pointers are declared by specifying the type of thing they point at. For example,

```
int *p;
```

defines 'p' as a pointer to an int (so, therefore, '*p' is an int, hence the form of the declaration).

Note carefully, then by declaring 'p' as in the above example, the compiler simply allocates space for the pointer (4 bytes in most cases), *not for the variable that the pointer points to!* This is a very important point, and is often overlooked. Before using '*p' in an expression, you have to ensure that 'p' is set to point to a valid int. This 'int' must have had space allocated for it, either statically, automatically, or by dynamically allocating memory at run-time (using a 'malloc' function).

Here is an example showing some of the uses of pointers:

```
#include <stdio.h>
void main (void) {
  int m = 0, n = 1, k = 2;
  int *p;
```

```
  char msg[] = "hello world";
  char *cp;

  p   = &m;       /* p now points to m                        */
  *p  = 1;        /* m now equals 1                           */
  k   = *p;       /* k now equals 1                           */
  cp  = msg;      /* cp points to the first character of msg */
  *cp = 'H';      /* change the case of the 'h' in msg        */
  cp  = &msg[6];  /* cp points to the 'w'                     */
  *cp = 'W';      /* change its case                          */

  printf ("m = %d, n = %d, k = %d\nmsg = \"%s\"\n", m, n, k, msg);
}
```

Note the very important point that the name of an array ('msg' in the above example), if used without an index, is considered to be a pointer to the first element of the array. In fact, an array name followed by an index is exactly equivalent to a pointer followed by an offset. For example,

```
#include <stdio.h>
void main (void) {
  char msg[] = "hello world";
  char *cp;

  cp = msg;

  cp[0]     = 'H';
  *(msg+6)  = 'W';

  printf ("%s\n", msg);
  printf ("%s\n", &msg[0]);
  printf ("%s\n", cp);
  printf ("%s\n", &cp[0]);
}
```

Note, however, that 'cp' is a variable, and can be changed, whereas 'msg' is a constant, and is not an lvalue.

Pointers used as arguments to functions

We have already seen that C functions can not alter their arguments. They can, however, alter the variables that their arguments point to. Hence, by passing a pointer to a variable, one can get the effect of 'call by reference'.

Here is an example of a function that swaps the values of its two arguments:

```
void swap_args (int *pi, int *pj) {
  int temp;

  temp = *pi;
  *pi = *pj;
  *pj = temp;
}
```

If all the asterisks were left out of this routine, then it would still compile OK, but its only effect would be to swap local copies of 'pi' and 'pj' around.

String functions

The standard C libraries include a bunch of functions for manipulating strings (i.e., arrays of chars).

Note that before using any of these functions, you should include the line "#include <string.h>" in your program. This include-file defines prototypes for the following functions, as well as defining special types such as 'size_t', which are operating-system dependent.

```
char *strcat (s1, s2)          Concatenates s2 onto s1. null-terminates s1.
                               Returns s1.

char *strchr (s, c)            Searches string s for character c. Returns a
                               pointer to the first occurence, or null pointer
                               if not.

int strcmp (s1, s2)            Compares strings s1 and s2. Returns an integer
                               that is less than 0 is s1 < s2; zero if s1 = s2;
```



```
                               and greater than zero is s1 > s2.

char *strcpy (s1, s2)          Copies s2 to s1, returning s1.

size_t strlen (s)              Returns the number of characters in s, excluding
                               the terminating null.

char *strncat (s1, s2, n)      Concatenates s2 onto s1, stopping after 'n'
                               characters or the end of s2, whichever occurs
                               first. Returns s1.

int strncmp (s1, s2, n)        Like strcmp, except at most 'n' characters are
                               compared.

char *strncpy (s1, s2)         Like strcpy, except at most 'n' characters are
                               copied.

char *strrchr (s, c)           Like strchr, except searches from the end of
                               the string.

int strstr (s1, s2)            Searches for the string s2 in s1. Returns a
                               pointer if found, otherwise the null-pointer.

size_r strspn (s1, s2)         Returns the number of consecutive characters
                               in s1, starting at the beginning of the string,
                               that are contained within s2.

size_r strcspn (s1, s2)        Returns the number of consecutive characters
                               in s1, starting at the beginning of the string,
                               that are not contained within s2.

char *strpbrk (s1, s2)         Returns a pointer to the first character in s1
                               that is present in s2 (or NULL if none).

char *strerror (n)             Returns a pointer to a string describing the
                               error code 'n'.

char *strtok (s1, s2)          Searches s1 for tokens delimited by characters
                               from s1. The first time it is called with a
                               non-null s1, it writes a NULL into s1 at the
                               first position of a character from s2, and
                               returns a pointer to the beginning of s1. When
                               strtok is then called with a null s1, it finds
                               the next token in s1, starting at one position
                               beyond the previous null.
```

The following program illustrates the use of 'strtok', the most complex of the string functions.

```
#include <string.h>
#include <stdio.h>
int main (void) {
  char s1[80];
  char s2[80];
  char *cp;

  if (gets(s1) == (char *)NULL) { /* gets returns the next line of input from
                                     stdio, a null if there isn't one */

    printf ("end of file\n");
    return 1;
  }
  if (gets(s2) == (char *)NULL) {
    printf ("end of file\n");
    return 1;
  }

  cp = strtok (s1, s2);
  do {
    printf ("<%s>\n", cp);
  } while ((cp = strtok ((char *)NULL, s2)) != (char *)NULL);
  return 0;
}
```

File operations

<stdio> defines a number of functions that are used for accessing files. Before using a file, you have to declare a pointer to it, so that it can be referred to with the functions. You do this with, for example,

```
FILE *in_file;
FILE *out_file;
```

where ‘FILE’ is a type defined in <stdio> (it is usually a complicated structure of some sort). To open a file, you do, for example:

```
in_file  = fopen ("input_file.dat",  "r");
out_file = fopen ("output_file.dat", "w");
```

Note the use of "r" to indicate read access, and "w" to indicate write access. The following modes are available:

```
"r"   read
"w"   write (destroys any existing file with the same name)
"rb"  read a binary file
"wb"  write a binary file (overwriting any existing file)
"r+"  opens an existing file for random read access, and
      writing to its end
"w+"  opens a new file for random read access, and writing to
      its end (destroys any existing file with the same name)
```

Once the file is opened, you can use the following functions to read/write it:

```
int getc (FILE *fp)             Returns the next character from ‘fp’, or
                                EOF on error or end-of-file.

int putc (int c, FILE *fp)      Write the character c to the file ‘fp’,
                                returning the character written, or EOF on
                                error.

int fscanf (FILE *fp, char *format, ...) Like scanf, except input is taken
                                from the file fp.

int fprintf (FILE *fp, char *format, ...) Like printf, except output is written
                                to the file fp.

char *fgets (char *line, int n, FILE *fp) Gets the next line of input from the
                                file fp, up to ‘n-1’ characters in length.
                                The newline character is included at the end
                                of the string, and a null is appended. Returns
                                ‘line’ if successful, else NULL if end-of-file
                                or other error.

int fputs (char *line, FILE *fp) Outputs the string ‘line’ to the file fp.
                                Returns zero is successful, EOF if not.
```

When you have finished with a file, you should close it with ’fclose’:

```
int fclose (FILE *fp)           Closes the file ’fp’, after flushing any
                                buffers. This function is automatically called
                                for any open files by the operating
                                system at the end of a program.
```

It can also be useful to *flush* any buffers associated with a file, to guarantee that the characters that you have written have actually been sent to the file:

```
int fflush (FILE *fp)           Flushes any buffers associated with the
                                file ’fp’.
```

To check the status of a file, the following functions can be called:

```
int feof (FILE *fp)             Returns non-zero when an end-of-file is read.

int ferror (FILE *fp)           Returns non-zero when an error has occurred,
                                unless cleared by clearerr.

void clearerr (FILE *fp)        Resets the error and end-of-file statuses.

int fileno (FILE *fp)           Returns the integer file descriptor associated
                                with the file (useful for low-level I/O).
```

Note that the above ‘functions’ are actually defined as pre-processor macros in C. This is quite a common thing to do.

Structures

Any programming language worth its salt has to allow you to manipulate more complex data types that simply ints, and arrays. You need to have structures of some sort. This is best illustrated by example:

To define a structure, use the following syntax:

```
struct time {
  int hour;
  int minute;
  float second;
};
```

This *defines* a new data type, called ‘time’, which contains 3 elements (packed consecutively in memory). To declare a variable of type ‘time’, do the following:

```
struct time t1[10], t2 = {12, 0, 0.0F};
```

To refer to an individual element of a structure, you use the following syntax:

```
t1[0].hour = 12;
t1[0].minutes = 0;
t1[0].second = t2.second;
t1[1] = t2;
```

Structures can contain any type, including arrays and other structures. A common use is to create a linked list by having a structure contain a pointer to a structure of the same type, for example,

```
struct person {
  char *name[80];
  char *address[256];
  struct person *next_person;
};
```

Multiple precision arithmetic

As an example of how to program in C, let’s explore the topic of multiple precision arithmetic. All the hard work will be done by GNU’s Multiple Precision Arithmetic Library (GNU MP), written by Torbjorn Granlund. The version I will be using here is 1.3.2, obtained from archie.au on 9 August 1994.

Multiple precision arithmetic allows you to perform calculations to a greater precision than the host computer will normally allow. The penalty you pay is that the operations are slower, and that you have to call subroutines to do all the calculations.

GNU MP can perform operations on integers and rational numbers. It uses preprocessor macros (defined in gmp.h) to define special data types for storing these numbers. MP_INT is an integer, and MP_RAT is a rational number. However, since a multiple precision number may occupy an arbitrarily large amount of memory, it is not sufficient to allocate memory for each number at compile time. GNU MP copes with this problem by dynamically allocating more memory when necessary.

To begin with you need to initialise each variable that you are going to use. When you are finished using a variable, you should free the space it uses by calling a special function.

```
  MP_INT x, y;
  mpz_init (&x);
  mpz_init (&y);

  /* operations on x and y */

  mpz_clear (&x);
  mpz_clear (&y);
```

Let’s now try a full program. For example, calculating the square root of two to about 200 decimal places:

```
#include <gmp.h>
#include <stdio.h>
void main (void) {
```

```
  char two[450], guess[225];
  int i;
  MP_INT c, x, temp, diff;

  two[0] = '2';
  for (i = 1; i < sizeof(two)-1; i++) {
    two[i] = '0';
  }
  two[i] = 0;
  mpz_init_set_str (&c, two, 10);

  guess[0] = '1';
  for (i = 1; i < sizeof(guess)-1; i++) {
    guess[i] = '0';
  }
  guess[i] = 0;
  mpz_init_set_str (&x, guess, 10);

  mpz_init (&temp);
  mpz_init (&diff);

  do {
    mpz_div (&temp, &c, &x);
    mpz_sub (&diff, &x, &temp);
    mpz_abs (&diff, &diff);
    mpz_add (&x, &temp, &x);
    mpz_div_ui (&x, &x, 2U);
  } while (mpz_cmp_ui (&diff, 10U) > 0);

  printf ("the square root of two is approximately ");
  mpz_out_str (stdout, 10, &x);
}
```

To compile, link, and run the program, use

```
cc -o two two.c -I/usr/local/include -L/usr/local/lib -lgmp
./two
```

PREPROCESSOR

```
#define PI 3.1415926535
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)

#include "filename"             /* from the current directory */
#include <filename>             /* from the system directories (modified by -I) */

#define DEBUG
                                /* defines the symbol DEBUG */
#ifdef DEBUG
  /* code here is compiled if DEBUG is defined */
#elif defined UNIX
  /* code here is compiled if DEBUG is not defined, and UNIX is defined */
#else
  /* code here is compiled if neither DEBUG or UNIX are defined */
#endif

#if 0
  /* code here is never compiled */
#endif
```

COMMAND-LINE ARGS, and returning a status to the shell

```
#include <stdio.h>
void main (int argc, char *argv[]) {
  printf ("this program is called '%s'\n", argv[0]);
  if (argc == 1) {
    printf ("it was called without any arguments\n");
  } else {
    int i;
    printf ("it was called with %d arguments\n", argc - 1);
    for (i = 1; i < argc; i++) {
      printf ("argument number %d was <%s>\n", i, argv[i]);
    }
  }
  exit (argc);
}
```


echo $status

MALLOC:

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

/* Generates a two dimensional matrix, and fills it randomly with
zeroes and ones. */

void main (void) {
  int xdim, ydim;
  int i, j;
  int *p, *q;

  printf ("x dimension of matrix? > ");
  scanf ("%d", &xdim);
  printf ("y dimension of matrix? > ");
  scanf ("%d", &ydim);

  p = (int *) malloc (xdim * ydim * sizeof(int));
  if (p == NULL) {
    printf ("malloc failed!\n");
    return;
  }

  for (i = 0; i < xdim * ydim; i++) {
    if (rand() > RAND_MAX/2) {
      *(p+i) = 1;
    } else {
      *(p+i) = 0;
    }
  }

  for (i = 0; i < xdim; i++) {
    q = p + i * ydim;
    for (j = 0; j < ydim; j++) {
      printf ("%d ", *(q++));
    }
    printf ("\n");
  }
  free ((void *)p);
}
```

ASSIGNMENTS:

Everyone should attempt the following assignment:

Write a C program that will accept an arbitrarily long list of real numbers from stdin, one per line, and will output the list sorted into ascending order. You should use "malloc" to obtain space to store the numbers. You do not know in advance how many numbers to expect, so you will have to "malloc" on-the-fly as necessary.

In addition, you will each be assigned one of the following exercises:

---

(1) Write a program that reads STDIN and outputs each line in reverse order to STDOUT, i.e., given the input

```
  this is a test
  of the program
```

It should return

```
  tset a si siht
  margorp eht fo
```

You are not allowed to use the C library function for reversing strings!

---

(2) Write a program that reads STDIN and replaces multiple consecutive blanks with single blanks and removes trailing blanks, i.e., given the input

```
  this   is  a test
  of the    program
```

It should return

```
  this is a test
  of the program
```

---

(3) Write a program that reads STDIN and returns the frequency of occurence of the lengths of words (delimited by spaces or end-of-line characters), i.e., given the input

```
  this   is  a test
  of the    program
```

It should return

```
  1 2 1 2 0 0 1
```

---

(4) Write a program that reads STDIN and returns the frequency of occurence of all the ASCII characters in the input.

---

(5) Write a program that takes two command-line arguments: a token, and a filename, and then prints out all lines in the file that contain the token.

---

(6) Write a program that takes two command-line arguments: a token, and a filename, and then prints out all lines in the file that don't contain the token, regardless of case.

---

(7) Write a program that reads STDIN and prints out every token (delimited by spaces or an end-of-line character) that is a valid integer.

---

(8) Write a program that reads STDIN and prints out every token (delimited by spaces or an end-of-line character) that begins with a capital letter and that only contains characters from the set A-Za-z.

---

(9) Write a program that reads an arbitrary number of filenames from the command line, and write out the number of bytes in each of the files. If a file doesn't exist, give a warning message.

The following examples are taken from CORONADO ENTERPRISES C TUTOR - Ver 2.00

```
/* This is an example of a "while" loop */
#include <stdio.h>

main() {
   int count;

   count = 0;
   while (count < 6) {
      printf ("The value of count is %d\n", count);
      count = count + 1;
   }
}
```

```
/* This is an example of a do-while loop */
#include <stdio.h>
```



```
main()
{
int i;

   i = 0;
   do {
      printf("The value of i is now %d\n",i);
      i = i + 1;
   } while (i < 5);
}
```

```
/* This is an example of a for loop */
#include <stdio.h>

main()
{
int index;

   for (index = 0; index < 6; index++)
     printf ("The value of the index is %d\n",index);
}
```

```
/* This is an example of the if and the if-else statements */
#include <stdio.h>

main()
{
int data;

   for (data = 0; data < 10; data++) {

      if (data == 2)
         printf("Data is now equal to %d\n",data);

      if (data < 5)
         printf("Data is now %d, which is less than 5\n",data);
      else
         printf("Data is now %d, which is greater than 4\n",data);

   }  /* end of for loop */
}
```

```
#include <stdio.h>
main()
{
int xx;

   for (xx = 5; xx < 15; xx++){
      if (xx == 8)
         break;
      printf("In the break loop, xx is now %d\n",xx);
   }

   for(xx = 5;xx < 15;xx = xx + 1){
      if (xx == 8)
         continue;
      printf("In the continue loop, xx is now %d\n",xx);
   }
}
```

```
#include <stdio.h>
main()
{
int truck;

   for (truck = 3;truck < 13;truck = truck + 1) {

      switch (truck) {
         case 3  : printf("The value is three\n");
                   break;
```

```
         case 4  : printf("The value is four\n");
                   break;
         case 5  :
         case 6  :
         case 7  :
         case 8  : printf("The value is between 5 and 8\n");
                   break;
         case 11 : printf("The value is eleven\n");
                   break;
         default : printf("It is one of the undefined values\n");
                   break;
      } /* end of switch */

   } /* end of for loop */
}
```

```
#include <stdio.h>
main()
{
int dog,cat,pig;

   goto real_start;

   some_where:
   printf("This is another line of the mess.\n");
   goto stop_it;

/* the following section is the only section with a useable goto */
   real_start:
   for(dog = 1;dog < 6;dog = dog + 1) {
      for(cat = 1;cat < 6;cat = cat + 1) {
         for(pig = 1;pig < 4;pig = pig + 1) {
            printf("Dog = %d  Cat = %d  Pig = %d\n",dog,cat,pig);
            if ((dog + cat + pig) > 8 ) goto enough;
         };
      };
   };
   enough: printf("Those are enough animals for now.\n");
/* this is the end of the section with a useable goto statement */

   printf("\nThis is the first line out of the spaghetti code.\n");
   goto there;

   where:
   printf("This is the third line of spaghetti.\n");
   goto some_where;

   there:
   printf("This is the second line of the spaghetti code.\n");
   goto where;

   stop_it:
   printf("This is the last line of this mess.\n");

}
```

```
/*************************************************************/
/*                                                           */
/*     This is a temperature conversion program written in   */
/*      the C programming language. This program generates   */
/*      and displays a table of farenheit and centigrade     */
/*      temperatures, and lists the freezing and boiling     */
/*      of water.                                            */
/*                                                           */
/*************************************************************/
#include <stdio.h>

main()
{
int count;        /* a loop control variable              */
int farenheit;    /* the temperature in farenheit degrees  */
int centigrade;   /* the temperature in centigrade degrees */

   printf("Centigrade to Farenheit temperature table\n\n");

   for(count = -2;count <= 12;count = count + 1){
```



```
      centigrade = 10 * count;
      farenheit = 32 + (centigrade * 9)/5;
      printf("  C =%4d   F =%4d  ",centigrade,farenheit);
      if (centigrade == 0)
         printf(" Freezing point of water");
      if (centigrade == 100)
         printf(" Boiling point of water");
      printf("\n");
   } /* end of for loop */
}
```

C programming course, School of Physics, UNSW / Michael Ashley / mcba@newt.phys.unsw.edu.au