ANSI C for Programmers on UNIX Systems

Tim Love
Cambridge University engineering Department
tpl@eng.cam.ac.uk

December 16, 1993

This document aims to:-

- Introduce C by providing and explaining examples of common programming tasks.
- Enable the reader to learn from available source code by clarifying common causes of incomprehension.

Coverage is not meant to be uniform: pedantry will be selective, aimed at describing aspects of C which are not present in other languages or are different to what a programmer from another language might expect. For a full description of C refer to one of the many books in the bibliography.

The first part of the document is an informal introduction to C. After the 1st set of exercises a more comprehensive description is given. After the 2nd set of exercises selected topics are covered.

The exercises and examples form an integral part of the course, containing information not duplicated elsewhere.

Carole Klein and Nick McLaren (Cambridge Computer Lab), Andy Piper, Campbell Middleton and James Matheson (CUED), and the contributors to `comp.lang.c` have all helped with this document.

All suggestions and corrections should go to Tim Love, CUED (`tpl@eng.cam.ac.uk`).

Contents

List of Demo Programs

List of Figures

The newest version of this document (Postscript and LaTeX) is available

- By gopher

  `gopher.eng.cam.ac.uk:CUED Published/Computer Service/teaching_C.ps`

  and

  `gopher.eng.cam.ac.uk:CUED Published/Computer Service/LaTeX/`

- By ftp

```
unix> ftp svr-ftp.eng.cam.ac.uk
Name: anonymous
Password: (send userid)
ftp> cd misc
ftp> binary
ftp> get love_C.ps.Z
ftp> get love_C.shar
ftp> quit
```

1 Introduction

C's popularity has increased as Unix has become more widespread. It is a flexible, concise and small language, with a mix of low-level assembler-style commands and high-level commands. It's much used with the $\mathcal{X}$ graphics system and increasingly for numerical analysis. The first *de facto* standard C was as described in [6] and is often known as *K&R C* . The current standard is ANSI C [11] in which the source contained in this document is written. It can be compiled on the HP workstations using `cc -Aa`.

To those who have programmed before, simple C programs shouldn't be too hard to read. Suppose you call this program `basics.c`

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

int mean(int a,int b)
{
  return (a + b)/2;
}

int main()
{
int i, j;
int answer;
   /* comments are done like this */
   i = 7;
   j = 9;

   answer = mean(i,j);
   printf("The mean of %d and %d is %d\n", i, j, answer);
   exit (0);
}
```

Note that the source is free-format and case matters.

All C programs need a `main` function where execution begins. In this example some variables local to `main` are created and assigned (using '`=`' rather than '`:=`'. Also note that '`;`' is a statement terminator rather than a separator as it is in `Pascal`). Then a function `mean` is called that calculates the mean of the arguments given it. The types of the formal parameters of the function (in this case `a` and `b`) should be compatible with the actual parameters in the call. The initial values of `a` and `b` are copied from the variables mentioned in the call (`i and j`).

The function `mean` returns the answer (an integer, hence the '`int`' before the function name), which is printed out using `printf`. The on-line manual page describes `printf` fully. For now, just note that the 1st argument to `printf` is a string in which is embedded format strings; `%d` for integers, `%f` for floats and `%s` for strings. The variables that these format strings refer to are added to the argument list of `printf`. The '`\n`' character causes a carriage return.

C programs stop when

- The end of `main` is reached.
- An `exit()` call is reached.
- The program is interrupted in some way.

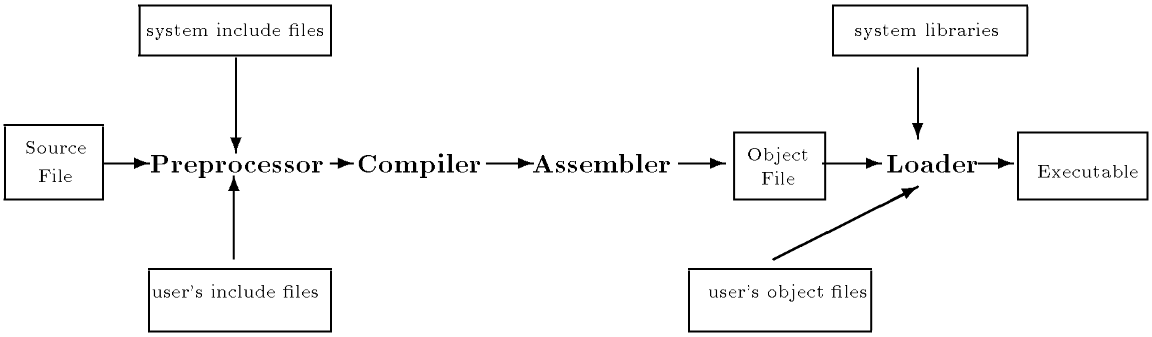

Figure 1: Compilation Stages

- The program crashes

This program can be compiled using '`cc -Aa -o basics basics.c`'. The '`-o`' option renames the resulting file `basics` rather than the default `a.out`. Run it by typing '`basics`'. A common mistake that beginners make is to call their executable '`test`'. Typing `test` is likely to run the `test` facility built into the shell, producing no input, rather than the user's program. This can be circumvented by typing `./test` but one might just as well void program names that might be names of unix facilities. If you're using the `ksh` shell then typing '`whence` *program_name*' will tell you whether there's already a facility with that name. ⚠

2 Compilation Stages

First the *Preprocessor* `cpp` is run. This strips out comments and interprets directives (lines with a '`#`' character in the first column). The '`#include`' directive inserts the named file into the source code, looking for the file in the directory `/usr/include`. Include files have a '`.h`' suffix by convention, and shouldn't contain executable code, only definitions and declarations. `/usr/include/stdio.h` and `/usr/include/stdlib.h` should always be included. Other useful include files are `/usr/include/limits.h` and `/usr/include/math.h` which define characteristics of the machine and the maths implementation.

Further preprocessor directives to do with macros, etc, will be introduced later. If you want to see how the code looks after pre-processing, try typing

```
cc -Aa -E basics.c
```

After the preprocessor comes the compiler, which after a pass or 2 produces assembler code. This is assembled to produce an object file, in this case `basics.o`.

Finally the link-loader produces the executable from the object file and any other object you mention. The standard C library is automatically consulted. Any other libraries that your code needs have to be mentioned on the compile line. *E.g.*, ending the line with '`-lm`' links in the maths library, '`-lX11`' links in X graphics functions. Note that it's not sufficient to just have '`#include <math.h>`' at the top of the file if you're doing maths. This just supplies enough information so that the compiler can do its work correctly. It doesn't tell the link-loader where the maths routines actually are. You need to have '`-lm`' on the command line before an executable can be produced.

The stages of compilation can be seen if a '`-v`' flag is given to the compiler.

3 Variables and Literals

Variable names can't begin with a digit. Nor can they contain operators (like '`.`' or '`-`'). They have to be declared before being used. The available scalar types are `char`, `short`, `int`, `long`, `float` and `double`. `char`s and the various lengths of integers can be `signed` (the default) or `unsigned`.

Often you don't explicitly have to convert types. If you operate on variables of different types, type conversion takes place auomatically. *E.g.* if you add an `int` and a `float` then the `int` will be converted to a `float` and the result will be a `float`.

```
unsigned int i;
float f = 3.14;
i = f;
```

will automatically set `i` to 3, truncating the real value. To explicitly convert types use *casting*; *E.g.*

```
i = (unsigned int) f;
```

Most conversions preserve the numerical value but occasionally this is not possible and overflow or loss of precision results. C won't warn you if this happens.

The length of an integer isn't the same on all machines. '`sizeof (int)`' will return the number of bytes used by an integer.

The scope of a variable is the block (the function, or `{ }` pair) it's declared in, or the remainder of the file it's declared in. Variables declared outside of a block and functions can be accessed from other files unless they're declared to be `static`.

Variables declared in functions can preserve their value between calls if they are defined as `static`, otherwise they're `automatic`, getting recreated for each call of the function.

Character values can written in various ways :- *E.g.* on machines that use `ASCII`

```
'A'   '\101'   '\x41'
```

all represent 65; the first using the ASCII `A` character, the second using octal and the third hexadecimal. The newline character is `'\n'`.

Integer and real values can be variously expressed:-

`15L`	long integer 15
`015`	octal integer 15
`0xF7`	Hex (base 16) number F7
`15.3e3`	15.3×10^3, a float
`15.3e3L`	15.3×10^3, a double

4 Aggregates

Variables of the same type can be put into arrays.

```
char letters[50];
```

defines an array of 50 characters, `letter[0]` being the 1st and `letter[49]` being the last character. C has no subscript checking; if you go off the end of an array C won't warn you.

Multidimensional arrays can be defined too. *E.g.*

```
char values[50][30][10];
```

defines a 3D array. Note that you can't access an element using `values[3,6,1]`; you have to type `values[3][6][1]`.

Variables of different types can be grouped into a *structure* (like a `record` in `Pascal`).

```
struct person {
     int age;
     int height;
     char surname[20];
} fred, jane;
```

defines 2 structures of type `person` each of 3 fields. Fields are accessed using the '`.`' operator. For example, `fred.age` is an integer which can be used in assignments just as a simple variable can.

`typedef` creates a new type. *E.g.*

```
typedef struct{
     int age;
     int height;
     char surname[20];
} person;
```

create a type called `person` and

```
typedef struct{
  double real;
  double imaginary;
} complex;
```

creates a `complex` type. Note that `typedef` creates new variable types but doesn't create any new variables. These are created just as variables of the predefined types are:-

```
person fred, jane;
```

As you see, new variable types are easily produced. What you *can't* do (but can in `C++` and `Algol68`) is extend the meaning of an existing operator ('overload' it) so that it works with the new variable type: you have to write a specific function instead.

A `union` is like a `struct` except that the fields occupy the same memory location with enough memory allocated to hold the largest item. The programmer has to keep a note of what the `union` is being used for. *E.g.* after this code fragment runs fred will be 163 years old.

```
...
union  person {
     int age;
     int height;
     char surname[20];
} fred;

fred.age = 23;
fred.height = 163;
printf("Fred is %d years old\n", fred.age);
...
```

5 Constructions

C has the following loop and selection constructs:-

Selection

```
...
if (i==3) /* checking for equality; `!=' tests for inequality */
   /* no braces needed for a single statement */
   j=4;
else{
  /*the braces are necessary if the
    clause has more than one statement
   */
   j=5;
   k=6;
}
...
```

```
...
/* switch is like the case statement in pascal.
   The values that the switching variable is compared with
   have to be constants, or `default'.
 */

switch(i){
case 1: printf("i is one\n");
        break; /* if break wasn't here, this case will
                  fall through into the next.
                */
case 2: printf("i is two\n");
        break;
default: printf("i is neither  one nor two\n");
        break;
}
...
```

Loops

```
...
while(i<30){    /* test at top of loop */
  something();
...
}
```

```
...
do {
   something();
} while (i<30); /* test at bottom of loop */
...
```

The '`for`' construction in C is very general. In its most common form it's much like `for` in other languages.

```
...
```

```
for(i=0; i<5; i=i+1){
   something();
}
...
```

The general form of '`for`' is

```
for ([expression1]; [expression2]; [expression3])
     something();
```

where all the expressions are optional. The default value for `expression2` is `1` (`true`). Essentially, the `for` loop is a `while` loop. The above `for` loop is equivalent to

```
...
expression1; /* initialisation */
while (expression2){ /* condition */
   something();
   expression3;      /* code done each iteration */
};
...
```

E.g. the 2 fragments below are equivalent. '`i`' is set to 3, the loop is run once for `i=3` and once for `i=4`, then iteration finishes when `i=5`.

```
for (i = 3; i < 5; i=i+1)
   total = total + i;
```

```
i = 3;
while(i < 5){
  total = total + i;
  i=i+1;
}
```

Within any of the above loop constructions, `continue` stops the current iteration and goes to the next and `break` stops the iterations altogether. *E.g.* in the following fragment `0` and `2` will be printed out.

```
...
i=0;
while (i<5){
   if (i==1){
     i = i+1;
     continue;
   }
   if (i==3)
     break;
   printf("i = %d\n", i);
   i=i+1;
}
...
```

If you want a loop which only ends when `break` is done, you can use '`while(1)`' (because `1` being non-zero, counts as being true) or '`for(;;)`'.

The `{ }` symbols are used to compound statements. You can declare variables at the start of any compound statement. For instance, if you're worried about the scope of an index variable in a `for` loop, you could do the following.

```
{int i;
 for (i=i;i<5;i++)
    printf("i is %d\n",i);
}
```

6 Exercises 1

1. Write a routine called `base` that does the following

   ```
   void base( int number, int base){
   /* print 'number' to the given 'base' */
   ...
   }
   ```

   Then write a `main` routine to test it.

2. `pascal` has a function called `odd`, that given an integer returns 1 if the number is odd, 0 otherwise. Write an `odd` function for C and write a `main` routine to test it. (hint – You can use the fact that in C, if `i` is an integer then `(i/2)*2` equals `i` only if `i` is even).

3. Print a table of all the primes less than 1000. Use any method you want. The sieve method is described here:- aim to create an array '`number`' such that if `numbers[i] == PRIME` then `i` is a prime number. First mark them all as being prime. Then repeatedly pick the smallest prime you haven't dealt with and mark all its multiples as being non prime. Print out the primes at the end. Here's a skeleton:-

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #define PRIME 1     /* Create aliases for 0 and 1 */
   #define NONPRIME 0

   int numbers[1000];


   void mark_multiples(int num){
   /* TODO: Set all elements which represent multiples of num to NONPRIME. */
   }

   int get_next_prime(int num){
   /* find the next prime number after 'num' */
   int answer;
     answer = num+1;
     while (numbers[answer] == NONPRIME){
        answer= answer +1;
        if (answer == 1000)
           break;
     }
     return answer;
   }

   main(){
   int i;
   int next_prime;

   /* TODO: Set all the elements to PRIME. Remember, the 1st element is
          numbers[0] and the last is numbers[999] */

   /* TODO: 0 and 1 aren't prime, so set numbers[0] and numbers[1]
   ```

```
   to NONPRIME */

   next_prime = 2;
   do{
     mark_multiples(next_prime);
     next_prime = get_next_prime(next_prime);
   } while(next_prime < 1000);

/* TODO: Print out the indices of elements which are still set to PRIME */

   exit(0);
}
```

The 'TODO' lines describe what code you need to add in.

You can speed up this program considerably by replacing 1000 where appropriate by something smaller. See page 42 for details.

7 Contractions

C veterans use abbreviated forms for expressions. Some are natural and widely adopted, others merely lead to obscurity – even if they produce faster code (and often they don't) they waste future programmers' time.

- `i++` is equivalent to `i=i+1`. This (and the `i--` decrementing operator) is a common contraction. The operation can be done after the variable is used, or (by using `--i`, `++i`) before, so

  ```
  ...
  i = 4;
  printf("i = %d\n", i++)
  ...
  ```

  and

  ```
  ...
  i = 4;
  printf("i = %d\n", ++i)
  ...
  ```

  will both leave `i` as `5`; but in the 1st fragment `4` will be printed out while in the 2nd `5` will.

- `i+=6` is equivalent to `i=i+6`. This style of contraction isn't so common, but can be used with most of the binary operators.

- Assignment statements have a value; the final value of the left-hand-side, so `j = (i=3+4)` will set `i` then `j` to `7`, and `i = j = k = 0` will set `k`, then `j`, then `i` to zero.

  This feature should be used with caution.

- The '`,`' operator is used between 2 expressions if the value of the 1st expression can be ignored. It's a way to put 2 or more statements where normally only one would go. *E.g.*

  ```
  for(init(3),i=0,j+0; i<100; i++,j++)
  ```

This feature is often over-used too.

- Expressions with comparison operators return `1` if the comparison is true, `0` if false, so `while(i!=0)` and `while(i)` are equivalent.
- The '`if (cond) exp1; else exp2`' construction can be abbreviated using '`(cond)?exp1: exp2`'. The following fragments are equivalent.

```
...
if (a==6)
  j=7;
else
  j=5;
...
(a==6)?j=7:j=5;
...
```

This notation should be used with discretion.

8 Functions

C has no procedures, only functions. Their definitions can't be nested but all except `main` can be called recursively. In `ANSI C` the form of a function definition is

<function type> *<function name>* (*<formal argument list>*)
{
<local variables>
<body>
}

E.g.

```
int mean(int x, int y)
{
int tmp;

tmp = (x + y)/2;
return tmp;
}
```

In *K&R C* the same function would be written as

```
int mean(x,y)
int x;
int y;
{
int tmp;

tmp = (x + y)/2;
return tmp;
}
```

Note that the formal argument declarations are differently placed. This are the most visible difference between `ANSI C` and *K&R C* . Programs exist to convert between the 2 forms of formal argument declaration (see the '`comp.lang.c`' newsgroup for details).

The default function type is '`extern int`' and the default type for the formal arguments is '`int`' but depending on these defaults is asking for trouble; they should be explicitly declared.

Functions end when

- execution reaches the closing '}' of the function. If the function is supposed to return something, the return value will be undefined.
- a '`return`' statement is reached, returning control to the calling function.
- an '`exit`' statement is reached, ending execution of the whole program.

Just as `return` can return a value to the calling routine, so `exit` returns a value to the `Unix` environment. By convention, returning a zero means that the program has run successfully.

All parameters in C are 'passed by value'. To performed the equivalent of `Pascal`'s 'pass by reference' you need to know about pointers.

9 Pointers

Even if you don't use pointers yourself, the code you'll learn from will have them. Suppose `i` is an integer. To find the address of `i` the `&` operator is used (`&i`). Setting a pointer to this value lets you refer indirectly to the variable `i`. If you have the address of a pointer variable and want to find the variable's value, then the dereferencing operator `*` is used.

```
...
int i;
/* The next statement declares i_ptr to be a pointer at
   an integer. The declaration says that if i_ptr is
   dereferenced, one gets an int.
 */
int *i_ptr;
i_ptr = &i; /* initialise i_ptr to point to i */

/* The following 2 lines each set i to 5 */
i = 5;
*iptr = 5;
```

A pointer-to-an-`int` has type `int*` and is of a different type to a pointer-to-a-`char` (which has type `char*`). The difference matters especially when the pointer is being incremented; the value of the pointer is increased by the size of the object it points to. In the following fragment, the pointer steps through an array.

```
...
int numbers[10];
int *iptr;
int i;

numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;

iptr = &numbers[0]; /* Point iptr to the first element in numbers[] */
```

```
/* now increment iptr to point to successive elements */
for (i=0; i<3; i++){
   printf("*iptr is %d\n", *iptr);
   iptr= iptr+1;
}
...
```

Pointers are especially useful when functions operate on structures. Using a pointer avoids copies of potentially big structures being made.

```
typedef struct {
     int age;
     int height;
     char surname[20];
} person;

person fred, jane;

int sum_of_ages(person *person1, person *person2){
int sum;  /* a variable local to this function. */

    /* Dereference the pointers, then use the `.' operator to get the
       fields */
    sum = (*person1).age + (*person2).age;
    return sum;
}
```

Operations like `(*person1).age` are so common that there's a special, more natural notation for it: `person1->age`.

To further illustrate the use of pointers let's suppose that in the first example on page 4 we wanted to pass to the function `mean` the variable where we wanted the answer stored. Since we no longer need `mean` to return a value, we can make it return `void`. Let's first try:-

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

void mean(int a, int b, int return_val )
{
  return_val = (a + b)/2;
  printf("return_val in mean in %d\n",return_val);
}

main()
{
int i, j;
int answer;
   i = 7;
   j = 9;

   mean(i,j, answer);

   printf("The mean of %d and %d is %d\n", i, j, answer);
}
```

This won't work. Although `return_val` is set to the right value in `mean`, `answer` isn't. Remember, `return_val` and `answer` are separate variables. The value of `answer` is copied into `return_val` when `mean` is called. The `mean` function doesn't know where `answer` is stored, so it can't change it. A pointer to `answer` has to be given to `mean()`.

```
#include <stdio.h>
#include <stdlib.h>
/* Note the form of the ptr_to_answer declaration below. It
   says that if you dereference ptr_to_answer you get an
   int. i.e. ptr_to_answer is a pointer to an int.
 */
void mean(int a,int b, int *ptr_to_answer)
{
  *ptr_to_answer = (a + b)/2;
}

main()
{
int i, j;
int answer;
   i = 7;
   j = 9;

   mean(i,j, &answer); /* Note that now we're passing a pointer
                        * to 'answer'
                        */

   printf("The mean of %d and %d is %d\n", i, j, answer);
}
```

There's a special value for null pointers (`NULL`) and a special type for generic pointers (`void*`). In *K&R C*, casting a pointer from one type to another didn't change its value. In `ANSI C` however, alignment is taken into account. If a `long` can only begin at an even memory location, then a pointer of type `char*` pointing to an odd location will have its value changed if cast into a `long*`.

10 Strings

In C a string is just an array of characters. The end of the string is denoted by a zero byte. The various string manipulation functions are described in the online manual page called '`string`', and declared in the `string.h` include file. The following piece of code illustrates their use and highlights some problems

```
/* strings.c */
#include <stdio.h>
#include <string.h>

char str1[10]; /* This reserves space for 10 characters */
char str2[10];
char str3[]= "initial text"; /* str3 is set to the right size for you
                              * and automatically terminated with a 0
                              * byte. Note that you can only initialise
                              * strings this way when defining
```

```
                              */
char *c_ptr;   /* declares a pointer, but doesn't initialise it. */

unsigned int len;

main()
{
 /* copy "hello" into str1. If str1 isn't big enough, hard luck */
 strcpy(str1,"hello");
 /* if you looked at memory location str1 you'd see these byte
    values:  'h','e','l','l','o','\0'
  */

 /* concatenate " sir" onto str1. If str1 isn't big enough, hard luck */
 strcat(str1," sir");
 /* values at str1 :  'h','e','l','l','o',' ','s','i','r','\0'
  */

 len = strlen(str1);  /* find the number of characters */
 printf("Length of <%s> is %d characters\n", str1, len);

 if(strcmp(str1, str3))
    printf("<%s> and <%s> are different\n", str1, str3);
 else
    printf("<%s> and <%s> are the same\n", str1, str3);

 if (strstr(str1, "boy") == (char*) NULL)
    printf("The string <boy> isn't in <%s>\n", str1);
 else
    printf("The string <boy> is in <%s>\n", str1);

 /* find the first `o' in str1 */
 c_ptr = strchr(str1,'o');

 if (c_ptr == (char*) NULL)
   printf("There is no o in <%s>\n", c_ptr);
 else{
   printf("<%s> is from the first o in <%s> to the end of the string\n",
                c_ptr, str1);
   /* Now copy this part of str1 into str2 */
   strcpy(str2, c_ptr);
 }
}
```

Usually '`str1`' would be used instead of '`&str1[0]`' to refer to the address of the first element of the character array, since C defines the value of an array name to be the location of the first element. In fact, once you've set `c_ptr` to `str`, the 2 variables behave similarly in *most* circumstances.

- There is not really any difference in the behaviour of the *array subscripting* operator `[]` as it applies to arrays and pointers. The expressions `str[i]` and `c_ptr[i]` are both processed internally using pointers. For instance, `str[i]` is equivalent to `*((str)+(i))`.

- Array and pointer declarations are interchangeable as function formal parameters. Since arrays decay immediately into pointers, an array is never actually passed to a function. Therefore, any parameter declarations which 'look like' arrays, e.g.

  ```
  int f(char a[])
  {
    ...
  }
  ```

  are treated by the compiler as if they were pointers, so '`char a[]`' could be replaced by '`char* a`'. This conversion holds *only* within function formal parameter declarations, nowhere else. If this conversion bothers you, avoid it.

Because the distinction between pointers and arrays often doesn't seem to matter, programmers get surprised when it does. Arrays are not pointers. The array declaration '`char str1[10];`' requests that space for ten characters be set aside. The pointer declaration '`char *c_ptr;`' on the other hand, requests a place which holds a pointer. The pointer is to be known by the name `c_ptr`, and can point to any `char` (or contiguous array of `char`s) anywhere. `str1` can't be changed: it's where the array begins and where it will always stay.

You can't pass whole arrays to functions, only pointers to them. To declare such pointers correctly you need to be aware of the different ways that multi-demensional arrays can be stored in memory. Suppose you created a 2D array of characters as follows:-

```
fruits[3][10] = {"apple", "banana", "orange"};
```

This creates space for 3 strings each 10 bytes long. Let's say that '`fruits`' gets stored at memory location `6000`. Then this will be the layout in memory:

```
6000 a  p  p  l  e \0 .  .  .  .
6010 b  a  n  a  n  a \0 .  .  .
6020 o  r  a  n  g  e \0 .  .  .
```

If you wanted to write a function that printed these strings out so you could do '`list_names(fruits)`', the following routine will work

```
void list_names(char names[][10] ){
  int i;
  for (i=0; i<3; i++){
    printf("%s\n", names[i]);
  }
}
```

The routine *has* to be told the size of the things that `names` points to, otherwise it won't be able to calculate `names[i]` correctly. So the '`10`' needs to be provided in the declaration. It doesn't care about how many things there are in the array, so the first pair of brackets might just as well be empty. An equivalent declation is

```
void list_names(char (*names)[10]
```

saying that '`names`' is a pointer to an array each of whose elements is 10 chars.

The above method wastes a lot of space if the strings differ greatly in length. An alternative way to initialise names is as follows:-

```
char *veg[] =  {"artichoke", "beetroot", "carrot"};
```

Here '`veg`' is set up as an array of `pointer-to-chars`. The layout in memory is different too. A possible layout is:-

```
   Address Value
      6000 9000
      6004 9600
      6008 9700
      ...

      9000  a  r  t  i  c  h  o  k  e \0
      9600  b  e  e  t  r  o  o  t  \0
      9700  c  a  r  r  o  t  \0
```

Note that '`veg`' is the start of an array of pointers. The actual characters are stored somewhere else. If we wanted a function that would print out these strings, then the '`list_names()`' routine above wouldn't do, since this time the argument '`names`' wouldn't be pointing to things that are 10 bytes long, but 4 (the size of a `pointer-to-char`). The declaration needs to say that '`names`' points to a character pointer.

```
void list_names(char **names){
  int i;
  for (i=0; i<3; i++){
    printf("%s\n", names[i]);
  }
}
```

The following declaration would also work:-

```
void list_names(char *names[]){
```

Using `cdecl` (see page 35) will help clarify the above declarations.

The program below shows the 2 types of array in action. The functions to print the names out are like the above except that

- The arrays are endstopped so that the functions needn't know beforehand how many elements are in the arrays.
- The `for` loop uses some common contractions: *E.g.* `list_names2` stops when `*names` is non-zero (*i.e.* non-`NULL`).

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

void list_names(char (*names)[10] ){
  for (; names[0][0]; names++){
    printf("%s\n", *names);
  }
}

void list_names2(char *names[] ){
  for (; *names; names++){
    printf("%s\n",*names);
  }
}
```

```
int main(int argc, char *argv[]){
char fruits[4][10] = {"apple", "banana", "orange", ""};
char *veg[] =  {"artichoke", "beetroot", "carrot", (char*) NULL};

  list_names(fruits);
  list_names2(veg);
  exit(0);
}
```

11 Exercises 2

To answer these exercises you'll need to be able to get keyboard input from the user. For the moment, use the following fragment to get a string from the user. `str` needs to point to the start of a character array.

```
char * get_string(char str[])
{
   printf("Input a string\n");
   return gets(str);
}
```

1. The following code fragment uses many of the contractions mentioned earlier. It comes from `ghostscript`. Re-write it to make it more legible.

   ```
   int ccase = (skew >= 0 ? copy_right :
                            ((bptr += chunk_bytes), copy_left))
                            + (int)function;
   ```

2. Write a program that invites the user to type in a string and prints the string out backwards (The answer to this is in section 17).

3. Write your own version of `strchr` (see the manual page for a description).

4. Write a program which reads in a string like "20C" or "15F" and outputs the temperature to the nearest degree using the other scale. The easiest way to parse the input string is to use `sscanf` to scan the input string for a number and a character. It will return the number of items successfully read in.

   ```
   ...
   int degrees;
   char scale;
   int return_value;
   ...
      return_value = sscanf(str,"%d%c",&degrees, &scale);
   ...
   ```

5. The following program will be developed later in the handout. Suppose you have a situation where you need to process a stream of things (they might be scanned character images, chess positions or as in this example, strings), some of which might be duplicates. The processing might be CPU-intensive, so you'd rather use the previously calculated values than re-process duplicate entries. What's needed is a *look-up table*.

   Each entry in the look-up table needs to have a record of the original string and the result of the processing. A structure of type `Entry`

```
typedef struct {
   char str[64];
   int value;
} Entry;
```

will do for now. For our purposes it doesn't matter much what the processing routine is. Let's use the following, multiplying all the characters' values together.

```
int process(char *str){
  int val = 1;
  while (*str){
     val = val * (*str);
     str++;
  }
  return val;
}
```

To get strings into the program you can use the `get_string` function. Now write a program that reads strings from the keyboard. If the string is new, then it's processed, otherwise its value is looked up in a table. The program should stop when '`end`' is typed. Here's a skeleton program to get you started.

```
/* hash1.c */
/* TODO include standard include files */

/* The following 2 lines use the preprocessor to create aliases.
   Note that these lines DON'T end with a ';'
 */
#define TABLE_SIZE 50
#define MAX_STR_LEN 64

typedef struct {
   char str[MAX_STR_LEN];
   int value;
} Entry;

char str[MAX_STR_LEN];

/* TODO Create an array of TABLE_SIZE elements of type Entry */

int process(char *str){
  int val = 1;
  while (*str){
     val = val * (*str);
     str++;
  }
  return val;
}

char * get_string(char str[])
{
   printf("Input a string\n");
```

```
   return gets(str);
}

main(){
       int num_of_entries = 0;
/* TODO Use get_string repeatedly. For each string:-
           If the string says 'end', then exit.
           If the str is already in the table,
              print the associated value
           else
              calculate the value, add a new
              entry to the table, then print the value.
 */
}
```

6. The method used above can be improved upon. Firstly, it will go wrong if there are too many strings. By choosing an arbitrarily large value for `TABLE_SIZE` you could overcome this problem, but the method of searching the table to see whether an entry is new becomes very inefficient as the table grows.

   A technique called *hashing* copes with this. First we need a *hash function* which given a string produces a number in the range `0..TABLE_SIZE`. The following function just adds up the value of the characters in the string and gets the remainder after dividing by `TABLE_SIZE`.

```
int hashfn(char *str){
  int total = 0;
  int i;
  while (i = *str++)
      total += i;
  return total % TABLE_SIZE;
}
```

   Now, whenever a string is to be processed, its hash value is calculated and that is used as an index into the table. If that entry is empty then the string is new and has to be processed. If the entry is occupied, then the associated value can be accessed. This method is flawed, but we'll deal with that problem later.

```
/* hash2.c */
/* TODO include standard include files */
#define TABLE_SIZE 50
#define MAX_STR_LEN 64
#define EMPTY -1
typedef struct {
   char str[MAX_STR_LEN];
   int value;
} Entry;

char str[MAX_STR_LEN];

/* TODO Create an array of TABLE_SIZE elements of type Entry */

int process(char *str){  /* Same as hash1.c */
```

```
  int val = 1;
  while (*str){
     val = val * (*str);
     str++;
  }
  return val;
}

char * get_string(char str[])  /* Same as hash1.c */
{
   printf("Input a string\n");
   return gets(str);
}

int hashfn(char *str){
  int total = 0;
  int i;
  while (i = *str++)
      total += i;
  return total % TABLE_SIZE;
}


void set_table_values(void){
/* TODO set all the value entries in the table to EMPTY
       (We'll assume that the process() routine doesn't
        produce -1)
*/
}

int find_entry(char *str, int bucket){
/* TODO
   if the entry is EMPTY then fill the entry in
      and return the string's processed value,
   else return the value of the entry.
*/
}

main(){
int bucket;
int val;
       set_table_values();
/* TODO Use get_a_string repeatedly. For each string:-
          use the hash function to find the string's entry
          in the table, then do the following
*/
       bucket = hashfn(str)
       val = find_entry(str,bucket);
       printf("Value of <%s> is %d\n",str, val);


}
```

7. The problem with this method is that the *hash function* may produce the same value for different strings (for example, '`act`' and '`cat`' will both map into the same entry). A simple way of coping with such '*collisions*' is the following:- If a table entry is occupied, check the string there to see if it's the one being searched for. If it is, then return the associated value. If it isn't the right string, then look at subsequent entries until either

   - an entry for the string is found
   - an empty entry is found.
   - It's been shown that all entries are full up.

   You'll have to add just a few lines to the `find_entry` routine of the previous exercise. Remember to cycle round when the bottom of the table is reached.

   A more robust method (and the answer to the exercise here) is in the next set of exercises (see section 17).

12 Keywords, Operators and Declarations

12.1 Keywords

You can't use the following reserved words for variable names, etc.

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

A few of these haven't yet been described.

auto :- This is the default *Storage Class* for variables so it's not explicitly used. `static`, which you've already met, is an alternative class.

const :- If a variable isn't meant to change you can define it as `const`. *E.g.*, If you create an integer using '`const int i = 6;`' then a later '`i = 7;`' will be illegal. However, if you create a pointer to `i` and use this to change the value, you'll probably get away with it. The main purpose of `const` is to help optimisers. `volatile` is the opposite of `const`.

enum :- C has enumerated types, like `pascal`. *E.g.*

```
enum color {Red, Green, Blue};
```

They're not as useful as in `pascal` because C doesn't check if you set an enumerated type to a valid value.

register :- You can suggest to the compiler that a variable should be kept in a register for faster access. *E.g.* '`register int i`' might help if `i` is a much-used indexing variable. An optimising compiler should use registers efficienty anyway. You can't use the '`&`' operator on a `register` variable.

12.2 Operators

At last, here is a table of operators and precedence.

The lines of the table are in order of precedence, so '`a * b + 6`' is interpreted as '`(a * b) + 6`'. When in doubt put brackets in!

The **Associativity** column shows how the operators group. *E.g.*'<' groups left to right, meaning that $a < b < c$ is equivalent to $(a < b) < c$ rather than $a < (b < c)$. Both are pretty useless expressions.

Associativity	Operator
left to right	`() [], ->, .`
right to left	the unary operators `!` (negation), `~` (bit-not), `++, --, -, *, &, sizeof`
right to left	cast `(type)`
left to right	`*, /, % (modulus)`
left to right	`- +`
left to right	`<<, >>`
left to right	`<, <=, >, >=`
left to right	`==, !=`
left to right	`&` (bit-and), `|` (bit-or)
left to right	`^` (bit-xor)
left to right	`&&` (logical and)
left to right	`||` (logical or)
right to left	`?:`
right to left	`=, +=, -=, /=, %=, >>=, &=`
left to right	`,`

Bit operations

C can be used to operate on bits. This is useful for low-level programming though the operations are also used when writing $\mathcal{X}$ graphics applications.

Setting a bit :- Suppose you wanted to set bit 6 of `i` (a `long`, say) to `1`. First you need to create a *mask* that has a `1` in the 6th bit and `0` elsewhere by doing '`1L<<6`' which shifts all the bits of the `long` `1` left 6 bits. Then you need to do a bit-wise `OR` using '`i = i | (1L<<6)`'.

Unsetting a bit :- Suppose you wanted to set bit 6 of `i` (a `long`, say) to `0`. First you need to create a *mask* that has a `0` in the 6th bit and `1` elsewhere by doing '`1L<<6`' then inverting the bits using the `~` operator. Then you need to do a bit-wise `AND` using the `&` operator. The whole operation is '`i =i & ~(1<<6)`' which can be contracted to '`i &= ~(1<<6)`'.

Creating a mask for an $\mathcal{X}$ call :- In $\mathcal{X}$ graphics, masks are often created each of whose bits represent a option that is to be selected in some way. Each bit can be referred to using an alias that has been set up in an include file. *E.g.* a mask which could be used in a call to make a window sensitive to keypresses, key releases and buttonpresses could be set up by doing

```
unsigned int mask = KeyPressMask | KeyReleaseMask | ButtonPressMask;
```

12.3 Declarations

First, a note on terminology. A variable is *defined* when it is created, and space is made for it. A variable is *declared* when it already exists but needs to be re-described to the compiler (perhaps because it was *defined* in another source file).

Think of *declaring* in C like *declaring* at customs – admitting to the existence of something.

C declarations are not easy to read. Any good book on C should explain how to read complicated C declarations "inside out" to understand them, starting at the variable name and working outwards back to the base type. You shouldn't need to use complicated declarations so don't worry to much if you can't 'decode' them. Keep a cribsheet of useful `typedef`s and play with `cdecl` (see section 16.1).

`ANSI C` introduced the use of the '`void`' keyword in various contexts.

- '`routine(void)`' – the routine takes no arguments.
- '`void routine (int i)`' – the routine returns no arguments.
- '`void *ptr`' – `ptr` is a generic pointer which should be cast into a specific form before use.

The following examples show common declarations.

`int *p`	pointer to an int
`int x[10]`	an array of 10 ints
`int (*x)[10]`	a pointer to an array of 10 ints
`int *x[10]`	array of 10 pointers to ints
`int (*f)(int)`	pointer to a function taking and returning an int
`void (*f)(void)`	pointer to a function taking no args and returning nothing
`int (*f[])(int)`	An array of pointers to a functions taking and returning an int

Note the importance of the brackets in these declarations. If a declaration gets too complex it should be broken down. For example, the last example could be rewritten as

```
typedef int (*PFI)(int) /* declare PFI as pointer to function that
                           takes and returns an int.*/
PFI f[];
```

13 Memory Allocation

Space is automatically set aside for variables when they are defined, but sometimes you don't know beforehand how many variables you'll need or just how long an array might need to be. The `malloc` command creates space, returning a pointer to this new area. To illustrate its use and dangers, here's a sequence of attempts at writing a string reverser program.

```
#include <stdio.h>
#include <stdlib.h>
print_reverse(char *str)
{
int i;
unsigned int len;

len = strlen(str) - 1; /* Why the -1? Because if a string has n chars, the
                          last char will be at position  n-1
                        */
for (i=len; i>=0; i--)
    putchar(str[i]);
```

```
}

void main()
{
char input_str[100] /* why 100? */
  printf("Input a string\n");
  gets(input_str);  /* should check return value */
  printf("String was %s\n", input_str);
  print_reverse(input_str);
}
```

This works, but is a bit 'loose' (suppose the user types more than 100 characters?) and doesn't keep a copy of the reversed string should it be needed later. The next example shows a wrong (but not uncommon) attempt to solve this limitation.

```
#include <stdio.h>
/* WRONG! */
char* make_reverse(char *str)
{
int i, j;
unsigned int len;
char newstr[100];
len = strlen(str) - 1;
j=0;

for (i=len; i>=0; i--;)
    newstr[j] = str[i];
    j++;
/* now return a pointer to this new string */
return newstr;
}

void main()
{
char input_str[100]; /* why 100? */
char *c_ptr;
  printf("Input a string\n");
  gets(input_str);  /* should check return value */
  c_ptr = make_reverse(input_str);
  printf("String was %s\n", input_str);
  printf("Reversed string is %s\n", c_ptr);
}
```

Like many flawed C programs this will work much of the time, especially if it's not part of a bigger program. The problems are that :-

- The memory allocated for `newstr` when it was declared as an 'automatic' variable in `make_reverse` isn't permanent – its lifetime is just the time `make_reverse()` takes to execute. However, the array's contents aren't erased, they're just freed for later use, so if you access the array from outside `make_reverse` you might still get away with it for a while. Making `newstr` a `static` will preserve the data but only until it's overwritten by a subsequent call.

- The newly created array of characters, **newstr**, isn't terminated with a zero character, '\0', so trying to print the characters out as a string may be disastrous. 'Luckily' the memory location that should have been set to zero is likely to be zero anyway.

Let's try again.

```
/* mallocing.c */
#include <stdio.h>
#include <stdlib.h>
char* make_reverse(char *str)
{
int i;
unsigned int len;
char *ret_str, *c_ptr;
len = strlen(str);

 /* Create enough space for the string AND the final \0.
    malloc returns a void*, so the results needs to be
    cast into the right form.
  */
ret_str = (char*) malloc(len +1);
 /*
    Now ret_str points to a 'permanent' area of memory.
  */

/* Point c_ptr to where the final  '\0' goes and put it in */
c_ptr = ret_str + len;
*c_ptr = '\0';

/* now copy characters from str into the newly created space.
   The str pointer will be advanced a char at a time,
   the cptr pointer will be decremented a char at a time.
 */
while(*str !=0){ /* while str isn't pointing to the  last '\0' */
  c_ptr--;
  *c_ptr = *str;
  str++; /* increment the pointer so that it points to each
            character in turn. */
}
return ret_str;
}

void main()
{
char input_str[100]; /* why 100? */
char *c_ptr;
  printf("Input a string\n");
  gets(input_str);  /* Should check return value */
  c_ptr = make_reverse(input_str);
  printf("String was %s\n", input_str);
  printf("Reversed string is %s\n", c_ptr);
}
```

The `malloc`'ed space will be preserved until it is explicitly freed (in this case by doing '`free(c_ptr)`'). Note that the pointer to the `malloc`'ed space is the only way you have to access that memory: lose it and the memory will be inaccessible.

`malloc` is often used to create tree and list structures, since one often doesn't know beforehand how many items will be needed. See section A.4 for an example.

14 Input/Output

14.1 File I/O under Unix

Some file operations work on *file pointers* (buffered) and some lower level ones use small integers called *file descriptors* (an index into a table of information about opened files).

The following code doesn't do anything useful but it does use most of the file handling routines. The manual pages describe how each routine reports errors. If `errnum` is set on error then `perror` can be called to print out the error string corresponding to the error number, and a string the programmer provides as the argument to `perror`.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>    /* the man pages of the commands say which
                         include files need to be mentioned */
#define TRUE 1
int bytes_read;
size_t  fp_bytes_read;
int fd;   /* File descriptors */
int fd2;
FILE *fp; /* File pointers */
FILE *fp2;
char buffer[BUFSIZ]; /* BUFSIZ is set up in stdio.h */

main(){

  /* Use File descriptors - unbuffered */
  fd = open ("/etc/group", O_RDONLY);
  if (fd == -1){
     perror("Opening /etc/group");
     exit(1);
  }

  while (TRUE){
     bytes_read = read (fd, buffer,BUFSIZ);
     if (bytes_read>0)
        printf("%d bytes read from /etc/group.\n", bytes_read);
     else{
        if (bytes_read==0){
          printf("End of file /etc/group reached\n");
          close(fd);
          break;
        }
        else if (bytes_read == -1){
```

```
          perror("Reading /etc/group");
          exit(1);
        }
      }
    }


 /* now use buffering */
 fp = fopen("/etc/passwd","r");
 if (fp == NULL){
    printf("fopen failed to open /etc/passwd\n");
    exit(1);
  }

 while(TRUE){
    fp_bytes_read= fread (buffer, 1, BUFSIZ, fp);
        printf("%d bytes read from /etc/passwd.\n", fp_bytes_read);
    if (fp_bytes_read==0)
       break;
 }

 rewind(fp); /* go back to the start of the file */

/* Find the descriptor associated with a stream */
  fd2 = fileno (fp);
  if (fd2 == -1)
    printf("fileno failed\n");

/* Find the stream associated with a descriptor */
  fp2 = fdopen (fd2, "r");
  if (fp2 == NULL)
    printf("fdopen failed\n");
  fclose(fp2);
 }
```

To take advantage of unix's I/O redirection it's often useful to write filters: programs that can read from `stdin` and write to `stdout`. In Unix, processes have `stdin`, `stdout` and `stderr` channels. In `stdio.h`, these names have been associated with file pointers. The following program reads lines from `stdin` and writes them to `stdout` prepending each line by a line number. Errors are printed on `stderr`. `fprintf` takes the same arguments as `printf` except that you also specify a file pointer. `fprintf(stdout,....)` is equivalent to `printf(....)`.

```
/* line_nums.c
   Sample Usage :    line_nums < /etc/group
 */
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
int lineno = 0;
int error_flag = 0;
char buf[BUFSIZ]; /* BUFSIZ is defined in stdio.h */

main(){
  while(TRUE){
```

```
    if (fgets(buf,BUFSIZ, stdin) == NULL){
       if (ferror(stdin) != 0){
          fprintf(stderr,"Error during reading\n");
          error_flag = 1;
       }
       if (feof(stdin) != 0)
          fprintf(stderr,"File ended\n");
       clearerr(stdin);
       break; /* exit the while loop */
    }
    else{
       lineno++;
       /* in the next line, "%3d" is used to restrict the
          number to 3 digits.
       */
       fprintf(stdout,"%3d: ", lineno);
       fputs(buf, stdout);
    }
  }

  fprintf(stderr,"%d lines written\n", lineno);
  exit(error_flag);
}
```

`ferror()` and `feof()` are intended to clarify ambiguous return values. Here that's not a problem since a `NULL` return value from `fgets()` can only mean end-of-file, but with for instance `getw()` such double checking is necessary.

14.2 Interactive

14.2.1 Output

For efficiency, writing to files under Unix is usually buffered, so `printf(....)` might not immediately produce bytes at `stdout`. If you want to force synchronous output you can

- Use `stderr` (which is usually unbuffered) instead of `stdout`.
- Use `fflush(stdout)` to flush out the standard output buffer.
- Use `setbuf(stdout,NULL)` to stop standard output being buffered.

14.2.2 Input

`scanf` is a useful-looking routine for getting input. It looks for input of the format described in its 1st argument and puts the input into the variables pointed to by the succeeding arguments. It returns the number of arguments successfully read.

Suppose you wanted the user to type their surname then their age in. You could do this:-

```
int age;
char name[50];
int return_val;
main(){
  printf("Type in your surname and age, then hit the Return key\n");
```

```
  while(TRUE){
    return_val= scanf("%s %d", name, &age);
    if (return_val == 2)
       break;
    else
       printf("Sorry. Try Again\n");
  }
}
```

If you use `scanf` in this way to directly get user input, and the user types in something different to what `scanf()` is expecting, it goes wrong because `scanf` keeps reading until its entire input list is fulfilled or `EOF` is reached. It treats a newline as *white space*. Thus users can become very frustrated in this example if, say, they keep typing their name, then hitting *Return*. A better scheme is to store user input in an intermediate string and use `sscanf()`, which is like `scanf()` except that its first argument is the string which is to be scanned. *E.g.* in

```
...
int ret, x, y, z;
ret = sscanf(str,"x=%d y=%d z=%d", &x, &y, &z);
...
```

`sscanf`, given a string '`x=3 y=7 z=89`', will set the `x`, `y`, and `z` values accordingly and `ret` will be set to `3` showing that 3 values have been scanned. If `str` is '`x+1 y=4`', `sscanf` will return 2 and won't hang.

To read the original string in, `fgets()` is a safer routine to use than `gets()` since with `gets()` one can't check to see if the input line is too large for the buffer. This still leaves the problem that the string may contain a newline character (not just whitespace) when using `fgets`. One must make annoying provisions for ends of lines that are not necessary when input is treated as a continuous stream of characters.

15 Source File organisation

The needs of large-scale organisation and support for many platforms may make modules incomprehensible unless some understanding of the overall structure is gained first.

15.1 Preprocesser Facilities

The preprocessor has some useful options.

sourcefile inclusion :-

```
#include "defines.h"
...
#include <defines.h>
```

The difference between these two variants is that with the included file in quotes, it is first looked for in the directory of the source file. In each case, the standard include directories on the system are searched as well as any directories mentioned on the command line after the '`-I`' flag. See the '`cpp`' man page for more details.

macro replacement :-

Note that these macros are expanded before the compiler is called. They aid legibility. In the first example below, a simple substitution is done. In the second, an in-line macro is defined, whose execution should be faster than the equivalent function.

```
#define ARRAY_SIZE 1000
char str[ARRAY_SIZE];
...
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int max_num;
    max_num = MAX(i,j);
...
```

conditional inclusion :-

Blocks of code can be conditionally compiled according to the existence or value of a preprocessor variable. A variable can be created using the '`#define`' preprocessor directive or using the '`-D`' option at compilation time.

```
#define DEBUG
#ifdef DEBUG
  printf("got here\n");
#else
  something();
#endif /*DEBUG*/
...
#if 0
 /* this code won't reach the compiler */
 printf("got here\n");
#endif
```

15.2 Multiple Source Files

Modularisation not only makes the source more easy to manage but it speeds up re-compilation: you need only recompile the changed source files. Also, by keeping the `I/O` components in one file (and perhaps the text to be printed into another) one can more easily convert the software to run on other machines and in other natural languages.

By default, functions and variables defined outside of functions can be accessed from other files, where they should be declared using the `extern` keyword. If however the variable is defined as `static`, it can't be accessed from other files. In the following example, '`i`', '`j`' and the function '`mean`' are created in `file1.c` but only '`i`' can be accessed from `file2.c`.

```
/* file1.c */                        /* file2.c */
int i;                               extern int i;
static int j;
static int mean(int a, int b){
...
```

Names of external variables should be kept short; only the first 6 initial characters are guaranteed to be significant (though in practise the first 255 character often are).

You should keep to a minimum the number of global variables. You can use `include` files to manage your global variables.

1. Construct a '`globals.h`' file with all of your `#define`s and variable declarations in it. Make sure all variables are defined as `extern`s. Include this file in all the relevant source files.

2. In the file that contains your `main()`, you again have all the variable definitions, minus the `extern`s. This is important – if they are all defined `extern`, the linker will not be able to allocate memory for them.

You can achieve this with the help of the pre-processor if your `globals.h` looks like this:-

```
#ifdef LOCAL
#define EXTERN
#else
#define EXTERN extern
#endif

EXTERN int num_of_files;
..
```

In this way, the '`EXTERN`' becomes '`extern`' in every file that includes `globals.h`. The trick is then to have

```
#define LOCAL
#include "globals.h"
```

in the file containing the `main` routine.

If you're calling a routine in one file from another file it's all the more important for the formal parameters to be declared correctly. Note especially that the declaration '`extern char *x`' is not the same as '`extern char x[]`' – one is of type '`pointer-to-char`' and the other is '`array-of-type-char`' (see section 10).

15.3 Make

If you have many source files you don't need to recompile them all if you only change one of them. By writing a `makefile` that describes how the executable is produced from the source files, the `make` command will do all the work for you. The following makefile says that `pgm` depends on two files `a.o` and `b.o`, and that they in turn depend on their corresponding source files (`a.c` and `b.c`) and a common file `incl.h`:

```
pgm: a.o b.o
	cc -Aa a.o b.o -o pgm
a.o: incl.h a.c
	cc -Aa -c a.c
b.o: incl.h b.c
	cc -Aa -c b.c
```

Lines with a '`:`' are of the form

```
target : dependencies
```

`make` updates a target only if it's older than a file it depends on. The way that the target should be updated is described on the line following the dependency line (Note: this line needs to begin with a `TAB` character).

Here's a more complex example of a `makefile` for a program called `dtree`. First some variables are created and assigned. In this case typing '`make`' will attempt to recompile the `dtree` program (because the default target is the first target mentioned). If any of the object files it depends on are older than their corresponding source file, then these object files are recreated.

The targets needn't be programs. In this example, typing '`make clean`' will remove any files created during the compilation process.

```
# Makefile for dtree
DEFS =  -Aa -DSYSV
CFLAGS = $(DEFS) -O
LDFLAGS =
LIBS = -lmalloc -lXm -lXt -lX11 -lm

BINDIR = /usr/local/bin/X11
MANDIR = /usr/local/man/man1

OBJECTS_A = dtree.o Arc.o Graph.o  #using XmGraph

ARCH_FILES = dtree.1 dtree.c Makefile Dtree Tree.h TreeP.h \
             dtree-i.h Tree.c Arc.c Arc.h ArcP.h Graph.c Graph.h GraphP.h

dtree: $(OBJECTS_A)
        $(CC) -o dtree $(LDFLAGS) $(OBJECTS_A) $(LIBS)

Arc.o: Arc.c
        $(CC) -c $(CFLAGS) Arc.c

Graph.o: Graph.c
        $(CC) -c $(CFLAGS) Graph.c

dtree.o: dtree.c
        $(CC) -o dtree.o -c $(CFLAGS) -DTREE dtree.c

install: dtree dtree.1
        cp dtree $(BINDIR)
        cp dtree.1 $(MANDIR)

clean:
        rm -f dtree *.o core tags a.out
```

16 Debugging

16.1 Utilities

lint :- is a program which gives the sort of warning messages about 'unused variables' and 'wrong number of arguments' that non-C compilers usually give. `lint` takes most of the same arguments as the compiler. It needs special libraries which are already provided.

cflow :- To show which functions call which, use `cflow`. This produces an indented output which also gives an idea of function call nesting.

warnings :- Some compilers have flags to turn on extra checking. `gcc` for example has a `-Wall` option which gives a list of suspicious constructions as well as the usual compile errors.

cb :- To standardise the indentation of your program, send it through `cb`, a C beautifier;

```
cb ugly.c > lovely.c
```

cxrefs :- tells you where variables and functions are mentioned. It's especially useful with multi-file sources.

adb :- I only use `adb` to see why a core dump happened. If '`myprog`' causes a core dump then

```
adb myprog
$c
```

will show you what function's return addresses were on the stack when the crash happened, and what hex arguments they were called with. Quit using `$q`

Symbolic Debuggers :- `dbx`, `xdb`, or `gdbx` may be available to you. They are symbolic source-level debuggers under which you can run a program in trace mode allowing you to use breakpoints, query values, etc. To use this you will have to first compile your program with the `-g` flag.

perror() :- When a system call fails it generally sets an external variable called `errno` to indicate the reason for failure. Using `perror()` (which takes a string as an argument) will print the string out and print the error message corresponding to the value of `errno`

assert() :- This is useful for putting diagnostics into programs. When it is executed, if the expression it takes as an argument is false (zero), `assert` prints the expression's value and the location of the assert call. See the on-line manual page for more details.

cdecl :- This program can help with C declarations. See man page for details. Some examples:-

```
tw012/tpl:   cdecl declare fptab as array of pointer to function returning int
int *fptab[]()
tw012/tpl:   cdecl explain int '*fptab[]()'
declare fptab as array of pointer to function returning int
```

`cdecl` is available from archives in `comp.sources.unix/volume6`.

16.2 Some Common mistakes

C is based on the principle that programmers know what they're doing, so it lets them get on with it and doesn't get in their way. Throughout this document common errors have a Warning sign in the margin. A checklist of more errors is given here.

16.2.1 Miscellaneous

- A common mistake is to type '`=`' instead of '`==`'.

```
if (i=3)
  return 1;
else
  return 0;
```

will always return `1` because the assignment '`i=3`' has the value `3` and `3` is true!

- Comments in C can't be nested. Use the preprocessor directives to temporarily 'comment out' blocks of code. Suppose you had the following code.

  ```
  if (i=6)
    z=mean(x,y); /* get the xy mean */
  mean(z,y);
  ```

  If you decided not to risk running `mean` you might do

  ```
  /* comment this fragment out
  if (i=6)
    z=mean(x,y); /* get the xy mean */
  mean(z,y);
  */
  ```

  but it wouldn't work because the first '`/*`' would be matched by the '`*/`' on the '`mean(x,y)`' line (the '`/*`' on that line being ignored), and '`mean(z,y);`' wouldn't be commented out at all. In this case the final '`*/`' would be flagged as an error, but you won't always be so lucky.

- ```
  ...
  i = 3;
  j = 10;
  while (i<100);
    i = i+j;
  ...
  ```

  This `while` loop will go on for ever. The semicolon after the while condition is a null statement which forms the body of the loop so `i` will always be `3`. Take away that semicolon and `i = i+j` becomes the body, which is probably what was intended.

- When you have an if-else statement nested in another if statement, always put braces around the if-else. Thus, never write like this:

  ```
      if (foo)
        if (bar)
           win ();
      else
        lose ();
  ```

  (the `else` matches the closest `if`), always like this:

  ```
      if (foo)
        {
          if (bar)
              win ();
  ```
  ```

```
        else
            lose ();
    }
```

- Don't be fooled by indentation. In the following fragment only the execution of the '`j = 7;`' statement is conditional upon the value of `i`.

```
...
if (i==7)
  j = 7;
  k = 7;
...
```

- The order of operations in an expression isn't guaranteed to be left-to-right. A line like

```
  a[i++] = b[i++];
```

  will have different results according to whether or not the `i` on the left-hand side is calculated before the right-hand side is evaluated.

- The order of operator precedence sometimes surprises people.

```
 ...
 FILE *fp;
 ...
  if (fp=fopen(filename, "r") == NULL)
    return (NULL);
```

  Here the intention is to try opening a file, then compare the resulting `fp` to `NULL` to see if `fopen` failed. Unfortunately, what actually happens first is the test `(fopen(filename, "r") == NULL)` which has an integer result (non-zero if the statement is true). This result is then assigned to `fp`. The compiler should warn you about this problem. The code should have some extra brackets:-

```
 ...
 FILE *fp;
 ...
  if ((fp=fopen(filename, "r")) == NULL)
    return (NULL);
```

- The following won't work as expected because the '`~`' character needs to be interpreted by the shell.

```
  if ((fp=fopen("~/data", "r")) == NULL)
    return (NULL);
```

  You'll have to find out your home directory (use `getenv("HOME")`) and append to it.

- `scanf` takes *pointers* to the variables that are going to be set. The following fragment will cause a crash

```
...
int i;
scanf("%d",i); /* this should be scanf("%d",&i) */
```

- The most uncomfortable bugs are those that seem to move as you hunt them down. Put in some `printf()` statements and they just disappear – or seem to. This probably means that you're writing off the end of an array or that one of your pointers has gone astray.

  You can avoid the use of naked fixed arrays by wrapping them in a structure with pre and post sentinels, e.g.

```
struct fooby {
int prefix;
char myarray[N];
int postfix;
};
```

  and adding code to initialize the sentinels and check whether they have been tampered with. More simply, you could do something like

```
#define BUFLEN 10
int x[BUFLEN], y;
...
if (y >= BUFLEN || y<0)
  [error code here]
else
  x[y] = 255;
...
```

- There's a big difference between `'\0'` and `"\0"`. Suppose you had

```
char str[100];
char *str_ptr;
     str_ptr = str;
```

  then `str_ptr` and `str` would both point to the first element in the array. Suppose you wanted to initialise this string by making the first element a zero byte. You could do

```
     strcpy(str_ptr, "\0") /* or strcpy(str_ptr, "") */
```

  or

```
     *str_ptr = '\0';
```

  but

```
     str_ptr = "\0";
```

  would do something quite different. It would create a string in your executable (namely `"\0"`) and set `str_ptr` to point to it with potentially disastrous effects.

- Turning on optimisation may change the behaviour of your program, especially if the program isn't perfect. For instance, if optimisation re-positions a variable into a register it's less likely to be `0` initially, so if you've not initialised variables before use you might get a surprize.

16.2.2 declaration mismatch

- `getchar` returns an integer, not a `char` as you might expect. If the integer value returned is stored into a character variable and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent. *Read the manual page* before using a function.

- Suppose a function `reverse` takes a string. If the *K&R C* programmer accidentally writes

  ```
  reverse (str)
  {
   char *str;
   ...
  }
  ```

  rather than

  ```
  reverse (str)
   char *str;
  {
   ...
  }
  ```

  the compiler might not warn the programmer that the formal parameter `str` has the default type `int` and a local variable `str` is created which isn't initialised.

- In the next example, it looks as if the programmer meant to define 2 pointers to integers. In fact, `ptr2` is being defined as an integer.

  ```
  int*  ptr1, ptr2;
  ```

- In *K&R C* the following code would crash; (`ANSI C` does automatic type conversion)

  ```
  int mean(num1, num2)
  int num1, num2;
  {
   ...
  }

  int i, answer;
  float f;
  /* deliberate mistake! */
  answer = mean(f,j);
  printf("The mean of %f and %d is %d\n", f, j, answer);
  ```

  C functions usually get given arguments via the stack. Calling functions put values on the stack then peel the same number of bytes off when returned to, so it doesn't matter to *K&R C* if the subfunction doesn't use or declare all the arguments that it is given. It doesn't even matter if it declares more arguments than given by the caling function as long as it doesn't write to these values. Were it to do so, it might well overwrite the address that the called function should return to. Such a problem might not be recognised for quite a while, and isn't easy to track down. This is where '`lint`' (see 16.1) becomes useful

- If in one source file you have `int array[100]` and you want to use this array from another source file, you mustn't declare as `extern int *array` but as `extern int array[]`. The clearest explanation of why this is so comes from Chris Volpe (`volpecr@crd.ge.com`)

  When you declare `int array[100];` the compiler sets aside storage for 100 ints, at say, address 500. The compiler knows that `array` is an array, and when it tries to generate code for an expression like `array[3]`, it does the following: It takes the starting address of the array (500), and adds to that an offset equal to the index (3) times the size of an int (typically 4) to get an address of `500+3*4=512`. It looks at the int stored at address 512 and there's the int.

  When you give an external declaration in another file like `extern int *array;`, the compiler takes your word for it that `array` is a pointer. The linker resolves the symbol for you as an object that resides at address 500. But since you lied to the compiler, the compiler thinks there's a pointer variable stored at address 500. So now, when the compiler sees an expression like `array[3]`, it generates code for it like this: It takes the address of the pointer (500) and, assuming there's a pointer there, reads the value of the pointer stored there. The pointer will typpically reside at address 500 through 503. What's actually in there is indeterminate. There could be a garbage value stored there, say 1687. The compiler gets this value, 1687, as the address of the first int to which the pointer points. It then adds the scaled index offset (12) to this value, to get 1699, and tries to read the integer stored at address 1699, which will likely result in a bus error or segmentation violation.

  The thing to remember about all this is that even though `array[index]` and `pointer[index]` can be used interchangeably in your source code, the compiler generates very different object code depending on whether you are indexing off an array identifier or a pointer identifier.

16.2.3 malloc

`malloc()` allocates memory dynamically. The standard `malloc()` and `free()` functions need to be efficient and can't check the integrity of the heap on every call. Therefore, if the heap gets corrupted, seemingly random behaviour can occur. The following code won't work.

```
char *answer;
printf("Type something:\n");
gets(answer);
printf("You typed \"%s\"\n", answer);
```

The pointer variable `answer`, which is handed to the `gets` function as the location into which the response should be stored, has not been set to point to any valid storage. That is, we cannot say where the pointer `answer` points. Since local variables are not initialized, and typically contain garbage, it is not even guaranteed that `answer` starts out as a null pointer.

The simplest way to correct the question-asking program is to use a local array, instead of a pointer, and let the compiler worry about allocation:

```
#include <string.h>

char answer[100], *p;
main(){
```

```
        printf("Type something:\n");
        fgets(answer, 100, stdin);
        if((p = strchr(answer, '\n')) != NULL)
              *p = '\0';
        printf("You typed \"%s\"\n", answer);
      }
```

Note that this example also uses `fgets` instead of `gets` (always a good idea), so that the size of the array can be specified and `fgets` will not overwrite the end of the array if the user types an overly-long line, though unfortunately for this example, `fgets` does not automatically delete the trailing `\n`, as `gets` would.

Type `man dbmalloc` to find out about a malloc library that does extensive error checking.

Alignment problems can arise if `malloc` is used carelessly. Processors have different rules about (for instance) whether a `long` can be stored starting at an odd memory location. If you try to break these rules, your program will crash giving little or no clue why. The `HP RISC` chips only permit a `double` to start at an address divisible by 8, so trying something like

```
char *block = (char*) malloc(sizeof(double));
double d = 1.2;
     * (double*)block = d;
```

is likely to crash.

16.2.4 Find the bug

What looks wrong with these programs?

- ```
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {
  int i;
  for (i=0; i<10; i=i+1);
    printf("i is %d\n",i);
  }
  ```

- ```
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {
  int i;
  for (i=0; i<10; i=i+1)
    if (i=2)
      printf("i is 2\n");
    else
      printf("i is not 2\n");
  }
  ```

- ```
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {
  int i;
  for (i=0; i<10; i=i+1)
  ```
  ```

```
  if (i<2)
     printf("i is less than, but\n");
     printf("but not equal to, 2\n");
}
```

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

  main()
  {
  int i;
   i = 0;
   while (i < 10);
     i = i + 1;
  }
  ```

- ```
  #include <stdio.h>
  #include <stdlib.h>
  main()
  {
  int i;
  for (i=0; i<10; i=i+1)
      switch(i){

      case 0: printf("i is 0\n");
      case 1: printf("i is 1\n");
      default: printf("i is more than 1\n");
      }
  }
  ```

- the following code works on some machines but crashes on others ...

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

  typedef struct {
     double *par;
     double *pos;
     double *vel;
   } ajoint;

  main()
  {
    ajoint *joint;
    joint = (ajoint *) malloc(sizeof(ajoint) + sizeof(double));
    joint->pos = (double*) (joint +1);
    *(joint->pos) = 0;
  }
  ```

17 Exercises 3

1. Improve your `primes` program so that

   - It stops searching for primes in the range 0 to `n` once it has marked all the multiples of primes $\leq \sqrt{n}$

- It can take as an argument a number to show the upper bound of the primes to print out.

2. Put 10 numbers (either floats or integers) in a file, one per line. Write a program that reads the numbers then prints their sum and and average.

3. Read the 1st 10 uids from `/etc/passwd`, save them in an array of strings and sort them using `qsort`.

4. Take a simple program (`demo3.c` will be ok) and break it up into 2 or 3 source files. See if you can compile them into an executable. Try adding `static` to variable and function definitions to see what difference it makes. Write a makefile for it.

5. Write a program to count the number of ways that 8 queens can be placed on a chess board without any 2 of them being on the same row, column or diagonal.

6. Hashing - First a solution to the last hash exercise.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 50
#define MAX_STR_LEN 64
#define EMPTY  -1

typedef struct {
   char str[MAX_STR_LEN];
   int value;
} Entry;

char str[MAX_STR_LEN];

/* Create an array of elements of type Entry */
Entry table[TABLE_SIZE];

int process(char *str){
  int val = 1;
  while (*str){
     val = val * (*str);
     str++;
  }
  return val;
}

char * get_string(char str[])
{
   printf("Input a string\n");
   return gets(str);
}

int hashfn(char *str){
  int total = 0;
  int i;
  while (i = *str++)
      total += i;
```

```
  return total % TABLE_SIZE;
}


void set_table_values(void){
/* set all the value entries in the table to EMPTY
   (here we assume that the process() routine doesn't
    produce -1)
*/
int i;
  for (i =0;i<TABLE_SIZE;i++)
      table[i].value= EMPTY;
}

int find_entry(char *str, int bucket){
   if (table[bucket].value == EMPTY){
      strcpy(table[bucket].str,str);
      table[bucket].value = process(str);
   }
   else{
      if (strcmp(table[bucket].str,str)){
         bucket = (bucket +1)% TABLE_SIZE;
         return find_entry(str, bucket);
      }
   }

   return table[bucket].value;
}


main(){
int bucket;
int val;
       set_table_values();

/* Use get_string repeatedly. For each string:-
       use the hash function to find the string's entry
       in the table.
*/
       while(get_string(str)){
         if (! strcmp(str,"end")){
             printf("Program ended\n");
             exit(0);
         }

         bucket = hashfn(str);
         val = find_entry(str,bucket);
         printf("Value of <%s> is %d\n",str,val);
       }
}
```

Another approach to collisions is for each entry in the hash table to be the beginning of a linked list of items that produce the same hash function value. First we need to alter the `Entry` structure so that it includes pointer to another `Entry`. There's a slight complication here in that we can't define a pointer to something which isn't defined yet, so we introduce a *tag name* to the structure.

```
typedef struct _entry {
 int value;
 struct _entry *next;
 char str[20];
} Entry;
```

New entry structures can be generated using the following routine.

```
Entry *create_an_entry(void){
Entry *entry;
  entry = (Entry*) malloc(sizeof (Entry));
  return entry;
}
```

`find_entry` needs to be re-written.

```
int find_entry(Entry ** entry, char *str){
 if (*entry == NULL){
    *entry =  create_an_entry();
    set_entry(*entry,str);
    return (*entry)->value;
 }
 else{
    if ((*entry) -> value != EMPTY){
       if (!strcmp ((*entry) ->str, str)){
          printf("Valueue for <%s> already calculated\n",str);
          return (*entry) -> value;
       }
       else{
          printf("There's a collision: <%s> and <%s> share\n",
                    (*entry) ->str, str);
          printf("the same hashfn valueue\n");

          find_entry(&((*entry)->next),str);

       }
    }
    else{
       printf("<%s> is a new string\n",str);
       set_entry((*entry),str);
       return (*entry)->value;
    }
 }
}
```

The initial table can now be

```
/* Create an array of elements of type Entry */
Entry *table[TABLE_SIZE];
```

These entries need to be initialised to `NULL`.

Now write a program with the following `main` routine to test all this out.

```
main(){
int bucket;
int value;
       set_table_values();

/* Use get_string repeatedly. For each string:-
       use the hash function to find the string's entry
       in the table.
*/
       while(get_string(str)){
         if (! strcmp(str,"end")){
             printf("Program ended\n");
             exit(0);
         }

         bucket = hashfn(str);
         value = find_entry(&(table[bucket]), str);
         printf("Valueue of <%s> is %d\n",str,value);
       }
}
```

This program could be further elaborated

- At the moment, if a string is long enough it will be too big for the array. Change the `Entry` definition to:-

  ```
  typedef _entry {
   int val;
   Entry *entry;
   char *str;
  }
  ```

  and change the code so that correctly sized space for each string is created using `malloc`.
- A hash function should be quick to calculate and provide an even spread of values to minimize collisions. Add some diagnostics to the program and improve the hash function.

18 More information

- The `comp.lang.c` newsgroup has a great deal of information. Read the `Frequently Asked Questions` if nothing else.
- Tutorials :- a set of tutorials written by Christopher Sawtell is available from
  - The `gopher.eng.cam.ac.uk` gopher in `CUED help/languages/C/Tutorials`.
  - `paris7.jussieu.fr:/contributions/docs` by `anon-ftp`.
- Style Guides and Portability Guides appear on the net from time to time. Read `comp.lang.c` for details.

A Examples

A.1 Command Line arguments

```
#include <stdio.h>
#include <stdlib.h>
/* This shows how args can be read from the Unix command line */
int main(int argc, char *argv[]){
int i;
   printf("The arguments are\n", argc);
   for (i=1; i<argc; i++)
       printf("%d %s\n",i, argv[i]);
   exit (0);
}
```

A.2 Using qsort, random numbers and the clock

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
* compile on HPs using c89 -D_HPUX_SOURCE -o filename filename.c */

#define NUM 10

int comp(const void *a, const void *b )
{
    return *(int *)a - * (int *)b;
}

int main(int argc, char *argv[])
{
    int numbers[NUM];
    int i;

    srand48((long)time(NULL));

    printf("\nUnsorted numbers are:-\n");
    for (i=0; i< NUM; i++){
        numbers[i]= 1000 * drand48();
        printf("%d: %3d\n", i, numbers[i]);
    }

    /* See the qsort man page for an explanation of the following */
    qsort((void*) numbers, (size_t) NUM, sizeof(int), comp);

    printf("\nSorted numbers are:-\n");

    for (i=0; i< NUM; i++)
        printf("%d:%3d\n", i, numbers[i]);

    exit(0);
}
```

A.3 Calling other programs

The commands used from the command line can be called from C.

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

```
int main(int argc, char *argv[]){
FILE *popen();
FILE *fp;
char string[32];
  /* First use the system() call. Output will go to stdout. */
  system("date");

  /* Now `capture' the output of date using popen() */
  fp = popen("date","r");
  if (fp == NULL)
    fprintf(stderr,"Cannot run date\n");
  else{
    fgets(string, 32, fp);
    printf("The date command returns [%s]\n", string);
    pclose(fp);
  }
}
```

A.4 Linked Lists

The following program creates a singly linked list. Pointers are maintained to the head and tail of the list.

```
#include <stdio.h>

typedef struct _list_item {
  int val;
  struct _list_item *next;
} list_item;

/* prototypes */
list_item *add_list_item(list_item *entry, int value);
void print_list_items(void);

list_item *head=NULL;
list_item *tail=NULL;

main(int argc, char *argv[])
{
  tail=add_list_item(tail,5);
  tail=add_list_item(tail,7);
  tail=add_list_item(tail,2);

  print_list_items();
}

list_item *add_list_item(list_item *entry, int value)
{
  list_item *new_list_item;

  new_list_item=(list_item*)malloc(sizeof(list_item));
  if (entry==NULL){
    head=new_list_item;
    printf("First list_item in list\n");
  }
  else {
```

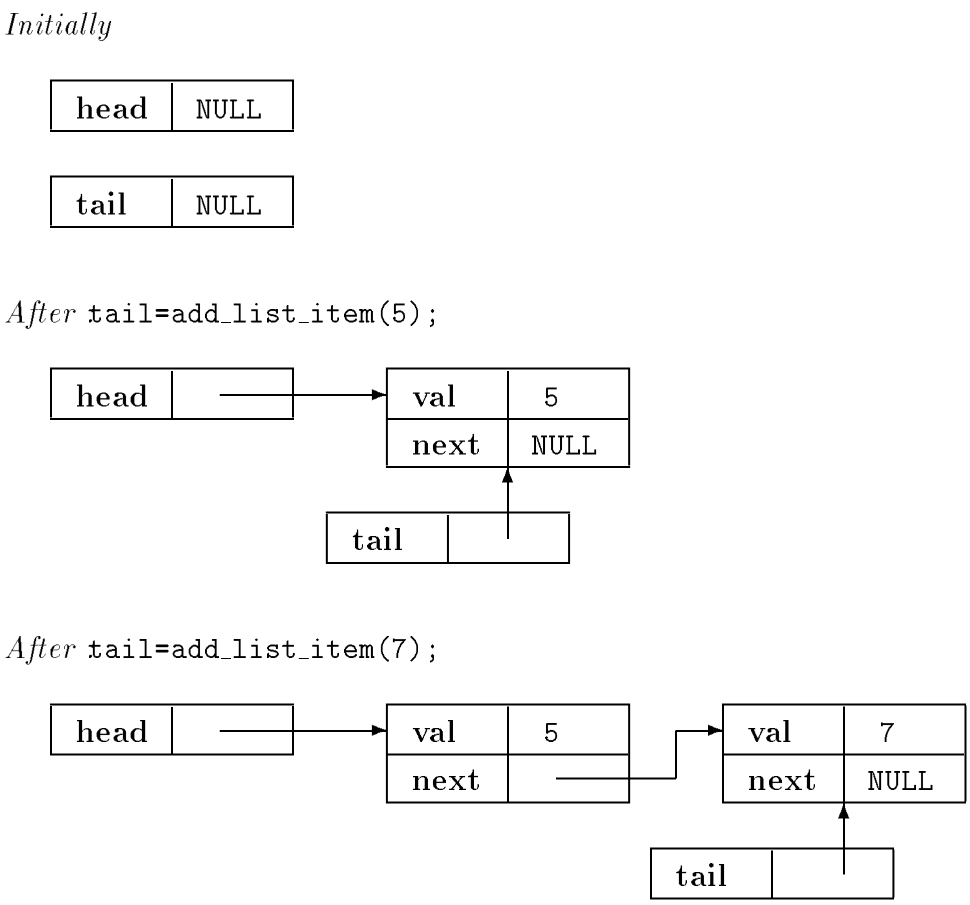

Figure 2: Linked List

```
    entry->next = new_list_item;
    printf("Adding %d to list. Last value was %d \n",value,entry->val);
  }
  new_list_item->val  = value;
  new_list_item->next = NULL;
  return new_list_item;
}

void print_list_items(void)
{
  list_item *ptr_to_list_item;

  for (ptr_to_list_item= head;ptr_to_list_item!= NULL;
           ptr_to_list_item=ptr_to_list_item->next) {
    printf("Value is %d \n", ptr_to_list_item->val);
  }
}
```

A.5 Using pointers instead of arrays

```
#include "stdio.h"
char *words[]={"apple","belt","corpus","daffodil","epicycle","floppy",
```

```
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
       "glands","handles","interfere","jumble","kick","lustiness",
"mangleworsel","nefarious","oleangeous","parsimonious",NULL};


void slow(void)
{
int i,j,count=0;
for (i=0; words[i] != NULL ; i=i+1)
     for (j=0; j <= strlen(words[i]) ; j=j+1)
         if(words[i][j] == words[i][j+1])
            count= count+1;
 printf("count %d\n",count);
}

void fast(void)
{
register char **cpp; /* cpp is an array of pointers to chars */
register char *cp;
register int count=0;

for (cpp= words; *cpp ; cpp++)     /* loop through words. The final
                                      NULL pointer terminates the loop */
     for (cp = *cpp ; *cp ; cp++)  /* loop through letters of a word.
                                      The final '\0' terminates the loop */
        if(*cp == *(cp+1))
          count++;
 printf("count %d\n",count);
}


/*count the number of double letters, first using arrays, then pointers */
void main(int argc, char *argv[]){
   slow();
   fast();
}
```

A.6 A data filter

The program reads from `stdin` an `ASCII` file containing values of a variable y for integral values of x running from 0 to n-1 where n is the number of values in the file. There may be several values on each line. The program outputs the x, y pairs, one pair per line, the y values scaled and translated.

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

int answer;
float offset;
float scale;
char buf[BUFSIZ];
int xcoord = 0;
```

```
char *cptr;

int transform(int a)
{
  return a * scale + offset + 0.5;
}


char*  eat_space(char *cptr){
/* This while loop skips to the nonspace after spaces.
   If this is the end of the line, return NULL
   `While a space, keep going'
 */

 while (*cptr ==' '){
     if (*cptr == '\0')
        return NULL;
     else
        cptr++;
   }
 return cptr;
}


char * next_next_num(char *cptr){
/* This while loop skips to the 1st space after a number.
   If this is the end of the line, return NULL
   `While NOT a space, keep going'
 */
   while (*cptr !=' '){
     if (*cptr == '\0')
        return NULL;
     else
        cptr++;
   }

   /* Now move to the start of the next number */
   return eat_space(cptr);
}

int main(int argc, char *argv[])
{
   offset  = 2.3;
   scale = 7.5;

   while(1){
      /* if we haven't reached the end of the file ...*/
      if(fgets(buf, BUFSIZ,stdin)!= NULL){
        /* initialise cptr to point to the first number ...*/
        cptr = eat_space(buf);
        do{
           /* convert the representation of the num into an int */
           sscanf(cptr,"%d", &num);
           /* print x and y to stdout */
           printf("%d %d\n",xcoord, tranform(num));
           /* skip to the start of the next number on the line */
           cptr=next_next_num(cptr);
           xcoord++;
```

```
        }while ( cptr!=NULL);
      }
      else{
        exit(0);
      }
    }
}
```

A.7 Reading Directories

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

#define REQUEST_DIR "/"
int main(int argc, char *argv[]){

FILE *fp;
DIR *dirp;
struct dirent *dp;
struct stat buf;

     dirp = opendir(REQUEST_DIR);
     chdir(REQUEST_DIR);

     /* Look at each entry in turn */
     while ((dp = readdir(dirp)) != NULL) {

       /* Now stat the file to get more information */
       if (stat(dp->d_name, &buf) == -1)
          perror("stat\n");

       if (S_ISDIR(buf.st_mode))
          printf("%s is a directory\n", dp->d_name);
       else if  (S_ISREG(buf.st_mode))
          printf("%s is a regular file\n", dp->d_name);
     }

     (void) closedir(dirp);
}
```

A.8 Queens: recursion and bit arithmetic

This program counts the number of ways that 8 queens can be placed on a chess board without any 2 of them being on the same row, column or diagonal. It was written by M. Richards at `cam.cl`

```
#include <stdio.h>
int count;

void try(int row, int ld, int rd){

  if (row == 0xFF)
    count++;
  else{
    int poss = 0xFF & ~(row | ld | rd);
    while (poss){
      int p = poss& -poss;
```

```
      poss = poss -p;
      try(row+p, (ld+p)<<1, (rd+p)>>1);
    }
  }
}

int main(int argc, char *argv[]){
  printf("Eight Queens\n");
  count = 0;
  try(0,0,0);
  printf("Number of solutions is %d\n", count);
  exit(0);
}
```

B More on Multidimensional Arrays and Pointers

The elements of `aai[4][2]` are stored in memory in the following order.
`aai[0][0]aai[0][1]aai[1][0]aai[1][1]`
`aai[2][0]aai[2][1]aai[3][0]aai[3][1]`

`*aai` is of type `int[]`. Note that:-

```
aai[1][2] == *( (aai[1])+2) == *(*(aai+1)+2)
```

and that numerically

```
aai == aai[0] == &aai[0][0]
```

`*aai` can be used as a pointer to the first element even though it is of type `'array 4 of int'` because it becomes `'pointer to int'` when used where a value is needed.

But `*aai` is not equivalent to a pointer. For example, you can't change its value. This distinction can easily and dangerously be blurred in multi-file situations illustrated in the following example. In

```
extern int *foo;
```

`foo` is a variable of type `pointer to int`. `foo`'s type is complete, `(sizeof foo)` is allowed. You can assign to foo. But given

```
extern int baz[];
```

`baz` is a variable of type '`array UNKNOWN-SIZE of int`'. This is an 'incomplete' type, you can't take `(sizeof baz)`. You cannot assign to `baz`, and although `baz` will decay into a pointer in most contexts, it is not possible for `(baz == NULL)` ever to be true.

The compiler will allow you to mix the array/pointer notation and will get it right, but it needs to know what the reality is. Once you declare the array/pointer correctly, you can then access it either way.

Suppose we have a simple array, and a subfunction for adding items to it:

```
#define MAXELTS 100

int array[MAXELTS];
int num_of_elements = 0;

install(x)
```

```
int x;
{
 if(num_of_elements >= MAXELTS){
    fprintf(stderr, "too many elements (max %d)\n", MAXELTS);
    exit(1);
 }

 array[num_of_elements++] = x;
}
```

Let's see how easy it is to remove the arbitrary limitation in this code, by dynamically allocating the array:

```
#include <stdlib.h>

int *array = NULL;
int nalloc = 0;
int num_of_elements = 0;

install(x)
int x;
{
  if(num_of_elements >= nalloc){
  /* We've run out of space. Reallocate with space for 10 more ints */
    nalloc += 10;
    array = (int *)realloc((char *)array, nalloc * sizeof(int));
    if(array == NULL){
       fprintf(stderr, "out of memory with %d elements\n", num_of_elements);
       exit(1);
    }
  }

  array[num_of_elements++] = x;
}
```

Be sure that realloc is declared correctly. If you don't have `<stdlib.h>` yet, use '`extern char *realloc();`' . If you want to be true-blue ANSI, use `size_t` for `nalloc` and `num_of_elements`.

When dynamically allocating a multidimensional array, it is usually best to allocate an array of pointers, and then initialize each pointer to a dynamically-allocated "row". The resulting "ragged" array can save space, although it is not necessarily contiguous in memory as a real array would be. Here is a two-dimensional example:

```
     /* create an array of pointers */
     int **array = (int **)malloc(nrows * sizeof(int *));
     if (array == NULL){
        fprintf(stderr,"Out of memory\n");
        exit(1);
     }

     for(i = 0; i < nrows; i++){
         /* create space for an array of ints */
         array[i] = (int *)malloc(ncolumns * sizeof(int));
           if (array[i] == NULL){
```

```
                fprintf(stderr,"Out of memory\n");
                exit(1);
            }
    }
```

You can keep the array's contents contiguous, while making later reallocation of individual rows difficult, with a bit of explicit pointer arithmetic:

```
int **array = (int **)malloc(nrows * sizeof(int *));
if (array == NULL){
   fprintf(stderr,"Out of memory\n");
   exit(1);
}

array[0] = (int *)malloc(nrows * ncolumns * sizeof(int));
if (array[0] == NULL){
   fprintf(stderr,"Out of memory\n");
   exit(1);
}

for(i = 1; i < nrows; i++)
   array[i] = array[0] + i * ncolumns;
```

In either case, the elements of the dynamic array can be accessed with normal-looking array subscripts: `array[i][j]`.

If the double indirection implied by the above schemes is for some reason unacceptable, you can simulate a two-dimensional array with a single, dynamically-allocated one-dimensional array:

```
int *array = (int *)malloc(nrows * ncolumns * sizeof(int));
```

However, you must now perform subscript calculations manually, accessing the `i,j` th element with `array[i * ncolumns + j]`.

C Signals and error handling

Various signals (interrupts) can be received by your program. See the `signal.h` include file for a list. You can trap them if you wish, or simply ignore them. *E.g.*

```
#include <signal.h>
...
/* this will ignore control-C */
signal(SIGINT, SIG_IGN);
```

The following code sets a 'timebomb'. After `Timer` is called, the program will continue execution until '`n`' milliseconds have passed, then normal execution will be interrupted and '`onalarm()`' will be called before normal execution is resumed.

```
#include <signal.h>


/*******/
static void onalarm()
```

```
/*******/
{
  something();
  signal(SIGALRM,SIG_DFL);
}

...
/*******/
void Timer(n)   /* waits for 'n' milliseconds */
 int  n;
/*******/
{
  long usec;
  struct itimerval it;

  if (!n) return;

  usec = (long) n * 1000;

  memset(&it, 0, sizeof(it));
  if (usec>=1000000L) {  /* more than 1 second */
    it.it_value.tv_sec = usec / 1000000L;
    usec %= 1000000L;
  }

  it.it_value.tv_usec = usec;
  signal(SIGALRM,onalarm);
  setitimer(ITIMER_REAL, &it, (struct itimerval *)0);
}
```

This same method can be used to catch emergency signals like `SIGBUS` (bus error) too.

D ANSI C

In 1983, the American National Standards Institute commissioned a committee, X3J11, to standardize the C language. After a long, arduous process, including several widespread public reviews, the committee's work was finally ratified as an American National Standard, X3.159-1989, on December 14, 1989, and published in the spring of 1990. For the most part, `ANSI C` standardizes existing practice, with a few additions from `C++` (most notably function prototypes) and support for multinational character sets (including the much-lambasted trigraph sequences). The `ANSI C` standard also formalizes the C run-time library support functions.

The published Standard includes a "Rationale," which explains many of its decisions, and discusses a number of subtle points, including several of those covered here. (The Rationale is "not part of ANSI Standard X3.159-1989, but is included for information only.") The Standard has been adopted as an international standard, ISO/IEC 9899:1990, although the Rationale is currently not included.

D.1 Converting to ANSI C

Many *K&R C* programs compile with an **ANSI C** compiler without changes. Where changes are required, the compiler will nearly always tell you. A list of differences between *K&R C* and **ANSI C** is in [6]. The most important are

Function prototyping :- Function prototypes aren't mandatory in **ANSI C**, but they improve error checking. Their use enables certain **ANSI C** features which otherwise, for backward compatibility, are suppressed.

Parameter Passing :-

- Floats are passed as floats (in *K&R C* floats are converted to doubles when passed to a function)
- Arguments are automatically cast into the right form for the called function. Without the function prototyping the following program wouldn't work because 'mean' is expecting 2 integers.

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

int mean(int a,int b)
{
  return a + b;
}

main()
{
int i;
float f;
int answer;

   i = 7;
   f= 5.3;

   /* deliberate mistake! */
   answer = mean(f,j);
   printf("%f + %d = %d\n", f, j, answer);
}
```

Standardisation :- The standard include files for **ANSI C** are

`assert.h`	Assertions
`ctype.h`	Character identification
`errno.h`	Error handling
`float.h`	Max and Min values for floats
`limits.h`	limits for integral types
`locale.h`	Internationalisation info
`math.h`	Advanced math functions
`setjmp.h`	Non-local jump
`signal.h`	Exception handling
`stdarg.h`	Variable numbers of arguments
`stddef.h`	Standard definitions
`stdio.h`	Input/Output
`stdlib.h`	General Utilities
`string.h`	String Manipulation
`time.h`	Date and Time functions

If you want to support both `ANSI C` and *K&R C* , you can use the following construction

```
#ifdef __STDC__
 /* ANSI code */
#else
 /* K and R code */
#endif
```

E Numerical Analysis

One problem when writing numerical algorithms is obtaining machine constants. On Sun's they can be obtained in `<values.h>`. The `ANSI C` standard recommends that such constants be defined in the header file `<float.h>`. Sun's and standards apart, these values are not always readily available,

The NCEG (Numerical C Extensions Group) is working on proposing standard extensions to C for numerical work, but nothing's ready yet, so before you do any heavy computation, especially with real numbers, I suggest that you browse through a `Numerical Analysis` book. Common problems that you might face are :-

Testing for equality :- Real numbers are handled in ways that don't guarantee expressions to yield exact results. It's risky to test for equality. Better is to use something like

```
d = max(1.0, fabs(a), fabs(b))
```

and then test `fabs(a - b) / d` against a relative error margin. Useful constants in `float.h` are `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`, defined to be the smallest numbers such that

```
1.0f + FLT_EPSILON  != 1.0f
1.0  + DBL_EPSILON  != 1.0
1.0L + LDBL_EPSILON != 1.0L
```

respectively.

Avoiding over- and underflow :- You can test the operands before performing an operation in order to check whether the operation would work. You should always avoid dividing by zero. For other checks, split up the numbers into fractional and exponent part using the `frexp()` and `ldexp()` library functions and compare the resulting values against `HUGE` (all in `<math.h>`).

Floats and Doubles :- *K&R C* encouraged the interchangeable use of `float` and `double` since all expressions with such data types where always evaluated using the `double` representation – a real nightmare for those implementing efficient numerical algorithms in C. This rule applied, in particular, to floating-point arguments and for most compilers around it does not matter whether one defines the argument as `float` or `double`.

According to the `ANSI C` standard such programs will continue to exhibit the same behavior *as long as one does not prototype the function.* Therefore, when prototyping functions make sure the prototype is included when the function definition is compiled so the compiler can check if the arguments match.

- Keep in mind that the `double` representation does not necessarily increase the *precision*. Actually, in most implementations the worst-case precision decreases but the *range* increases.
- Do not use `double` unnecessarily since in most cases there is a large performance penalty. Furthermore, there is no point in using higher precision if the additional bits which will be computed are garbage anyway. The precision one needs depends mostly on the precision of the input data and the numerical method used.

Infinity :- The IEEE standard for floating-point recommends a set of functions to be made available. Among these are functions to classify a value as `NaN`, `Infinity`, `Zero`, `Denormalized`, `Normalized`, and so on. Most implementations provide this functionality, although there are no standard names for the functions. Such implementations often provide predefined identifiers (such as `_NaN`, `_Infinity`, etc) to allow you to generate these values.

If `x` is a floating point variable, then `(x != x)` will be TRUE if and only if `x` has the value `NaN`. Many C implementations claim to be IEEE 748 conformant, but if you try the `(x!=x)` test above with `x` being a `NaN`, you'll find that they aren't.

In the mean time, you can write your own 'standard' functions and macros, and provide versions of them for each system you use. If the system provides the functions you need, you `#define` your 'standard' functions to be the system functions. Otherwise, you write your function as an interface to what the system provides, or write your own from scratch.

See `matherr(3)` for details on how to cope with errors once they've happened. See the `CUED help/languages/C/math_routines` file in the `gopher.eng.cam.ac.uk` gopher for a long list of available resources.

F Fortran and C

Here are some opinions that experienced programmers have given for why **fortran** has not been replaced by C:

- "C is definitely for wizards, not beginners or casual programmers. Usually people who are heavily into numerical work are not hacker types. They are mathematicians, scientists, or engineers. They want to do calculations, not tricky pointer manipulations. **fortran**'s constructs are more obvious to use, while even simple programs in C tend to be filled with tricks. Even the fundamental operation of reading input can be tricky in C."
- "**fortran** is dangerous to use, but not as dangerous as C. For instance, most **fortran** compilers have subscript checking as an option, while I have never encountered a C compiler with this feature. The ANSI standard for function prototypes will give C an edge over **fortran** in parameter mismatch error."
- "There is a large body of well tested mathematical packages available for **fortran**, that are not yet available in C. For example the IMSL package. However, this situation is improving for C."
- "In studies done at Cray Research, they found it took *significantly* longer for their programmers to learn C and the number of errors generated in coding in C (as opposed to **fortran**) was much higher."

- "C is hard to optimize, especially if the programmer makes full use of C's expressivity. Newer C features (like the `const` keyword etc) and new software technology are improving the situation."
- "C has too many system dependent aspects (e.g. round up or down when dividing negative integers)."

Whether or not the switch to C is worthwhile will depend on whether its quirks outweigh the benefits of having "more modern" data typing and control structures. `ANSI C` goes a long way to removing the quirks but for the time being **fortran** is probably more portable and will run faster on supercomputers without tweaking. On the other hand **fortran** may be harder to maintain, and it is a poor fit to algorithms that are best expressed with types more involved than n-dimensional arrays. When `Fortran9X` becomes commonplace, perhaps the decision will be easier to make.

G Calling Fortran from C

See the *Portability Guide* manual if you intend doing this on HP machines. Rudi Vankemmel (`ankemme@imec.be`) mentions some general points that are worth noting :-

1. Fortran uses a column wise storage of matrices while C stores them row wise. This means that when you want to parse a matrix from your C-program to the fortran routine you must transpose the matrix in your program before entering the routine. Of course, any output from such a routine must be transposed again.

   If you omit this step, then probably your program will run (because it has data to compute on) but it will generate wrong answers.

   If you have the Fortran source code (of any routine) then on some platforms you may use compiler directives specifying that the Fortran compiler must use row wise storage. Some platforms support these directives. However watch out with this if you call the same routine from another Fortran routine/program.

2. Your Fortran compiler may add an underscore to the routine name in the symbol table. Hence in the calling C-program/routine you must add a trailing underscore ! Otherwise the loader will complain about an undefined symbol. However, check your compiler for this. For example the Fortran compiler on VAX-VMS systems does NOT add a trailing underscore (there watch out with the fact that the VAX-Fortran compiler translates everything in uppercase).

3. Fortran passes its variables by reference. This means that you MUST give adresses in your calling C-program.

4. Watch out especially with `float`s and `double`s. Make sure that the size of the variable in the calling program is identical to the size in the Fortran routine. This is extremely important on machines with little endian byte ordering. Parsing a `float` (C-routine) to a `real*8` (Fortran) number will not generate SEGV but give wrong results as the data is parsed wrongly.

References

[1] Banahan, Brady, and Doran. *The C Book.* Addison Wesley, 1988.

[2] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual.* Prentice Hall, 1987.

[3] Allen I. Hollub. *The C Companion.* Prentice Hall, 1987.

[4] Mark Horton. *Portable C Software.* Prentice-Hall, 1990.

[5] Brian W. Kernighan and P.J. Plauger. *The Elements of Programming Style.* McGraw-Hill, 1978.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition.* Prentice-Hall, Inc., 1988.

[7] Andrew Koenig. *C Traps and Pitfalls.* Addison-Wesley, 1989.

[8] J.F.. Korsh and L.J. Garrett. *Data Structures, Algorithms, and Program Style using C.* PSW-Kent, 1988.

[9] Steve Oualline. *Practical C Programming.* O'Reilly & Associates, Inc.

[10] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *NUMERICAL RECIPES in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[11] X3J11. Draft Proposed American National Standard for Information Systems — Programming Language C. Technical Report X3J11/88–158, ANSI Accredited Standards Committee, X3 Information Processing Systems, December 1988.