

Standard Library: Useful Functions

- **Objectives**
 - Learn about the Standard Library
- **Contents**
 - Variable argument processing
 - The C and Pascal calling conventions
 - `stdarg.h`
 - Examples
 - Date and time types
 - Date and time functions
 - Examples
 - Random numbers
- **Practical**
- **Summary**



In this chapter, we shall examine a number of miscellaneous functions from the standard C library and show how they may be used in practical situations.

The chapter begins with a look at functions such as `printf` and `scanf` that take an unknown number of arguments. These functions are often referred to as *variadic functions*. They require special techniques in order to extract the parameters from the argument list. These techniques are illustrated in this chapter, and a number of examples are presented to highlight the various modes of operation of variadic functions.

The standard C library supplies a number of functions to determine the current date and time, as well as the elapsed time for which a process has been active. These date and time functions return the information in a number of different formats such as `time_t` and `struct tm`, and there are functions available to convert between the various formats. The chapter shows how these functions and data types fit together.

The next topic is random number generation, a technique used extensively in simulation and arcade games. Different compilers use different techniques for generating random numbers, but in each case the random number generator has to be "seeded" with an unpredictable initial value before random number generation can commence. This chapter discusses the standard functions `srand()` and `rand()` that are used for generating random numbers.

Variable argument functions

- **Functions which accept variable numbers of parameters**
 - Like *printf()* and *scanf()*
 - You can write your own variadic functions
 - This is considered a strength of C
- **<stdarg.h> contains macros to aid writing such functions**
 - *stdarg.h* (where shipped) contains an incompatible and obsolete set of macros invented on Berkeley enhanced versions of UNIX
- **Must declare at least one fixed parameter**
 - Fixed parameters are passed using Standard C convention
- **The variadic parameters are declared using ellipsis, "..."**
 - Variadic parameters are not type checked
 - Variadic parameters are passed using K&R convention
 - *char* passed as *int*
 - *short* passed as *int*
 - *float* passed as *double*

Most functions a programmer writes will take a fixed number of parameters. Every time the function is called, it will receive exactly the same number and type of arguments. Occasionally, however, it is necessary for a function to be flexible enough to take a variable number of parameters. A classic example is the `printf` function; `printf` takes a format string as its first argument, followed by a list of parameters of variable length and variable data types.

Functions such as `printf` are known as variadic because they take a variable number of parameters. In order to implement functions like these macros are provided in the Standard header file `stdarg.h`.

The `exec1` and `spawn1` family of functions already met are variadic. They have to be since it cannot be predicted in advance how many parameters there will be passed to the child process.

```
int  execlp(const char * cmdname, const char * arg0, ...);
int  spawnlp(int mode, const char* cmdname, const char* arg0, ...);
```

The macros in `stdarg.h` are closely related to those defined in the header file `stdarg.h`. This was developed at the University of Berkeley in California when they were enhancing the UNIX operating system. The similarity between the two sets of macros is very great, however they are not compatible.

Variadic functions must declare at least one fixed parameter. This is not unreasonable, for something needs to tell the function about the remaining parameters. With `printf` this single fixed parameter is the format string. This tells it about the number and type of the parameters.

The C calling convention

- **The C calling convention (rules for how parameters are passed on the stack) is central to the workings of variadic functions**
- **The convention is as follows:**
 - The *calling* function pushes the parameters from *right* to *left*
 - The function is called
 - The function returns
 - The *calling* function removes the parameters from the stack
- **The convention guarantees that `printf`, for example, is left with the format string at the top of the stack**

Each language has its own calling convention. This is a set of rules specifying how one function should pass parameters to another. The C language is no exception and has its own set of rules.

These rules are designed specifically to allow any variadic function to work.

The right to left order in which parameters are pushed onto the stack is vital for functions like `printf`. This guarantees that the thing left on the top of the stack is a pointer to the format string. If the parameters were pushed in reverse order, `printf` would have to grub around the stack guessing where the format string was located.

The Pascal calling convention

- **The Pascal calling convention is somewhat different and is provided as a contrast and because it is widely used in Windows programming**
- **The convention is:**
 - The *calling* function pushes the parameters from *left to right*
 - The function is called
 - The *called* function pops the parameters
 - The function returns
- **This convention is excellent for functions with fixed numbers of parameters**
 - (Pascal does not support variadic functions other than those built in to the language)

I think, therefore I am

The Pascal calling convention is designed for functions with fixed numbers of parameters – the language does not support variadic functions.

The first rule, in which parameters are passed from left to right, would be sufficient to break routines like `printf`. With the format string buried somewhere on the stack, `printf` would have to guess where it was. Also, `printf` would be responsible for popping (removing) the parameters. If it guessed incorrectly as to how many there were, it would remove the wrong amount and leave the stack corrupted.

In Windows, the Pascal calling convention is usually implemented using the `__stdcall` convention (also defined as `WINAPI` and `CALLBACK`). For example:

```
unsigned int WINAPI ThreadFunc (void *vNotUsed)
```

The `cdecl` calling convention is the default (as on Unix), but may be specified:

```
void __cdecl cdeclFunc ()
```

C vs. Pascal convention

- **Variadic functions**
 - Supported by C convention
 - Not supported by Pascal convention
- **Integrity**
 - C convention is (reasonably) proof against functions which access parameters which are not supplied and also functions which do not access parameters which are supplied
- **Efficiency**
 - Pascal convention is more efficient, since the "pop" instruction is placed once in the called function
 - C convention is slightly less efficient, since the "pop" instruction is repeated for each call
 - Some compilers provide keywords to specify the calling convention to be used
 - In the future the compiler itself may choose; fixed parameters - Pascal convention, variadic parameters - C convention

The C calling convention is more flexible than that used in Pascal. However, it is slightly less efficient in terms of code size. To understand why, consider the following code:

```
func_a(1, 2, 3, 4);
func_b(1, 2);
func_c(1);
func_d(1, 2);
```

C calling convention:

```
push parameters 1, 2, 3, 4
call func_a
pop parameters
push parameters 1, 2
call func_b
pop parameters
push parameter 1
call func_c
pop parameters
push parameters 1, 2
call func_d
pop parameters
```

Pascal calling convention:

```
push parameters 1, 2, 3, 4
call func_a

push parameters 1, 2
call func_b

push parameter 1
call func_c

push parameters 1, 2
call func_d
```

With the Pascal calling convention the "pops" occur in the functions `func_a`, `func_b`, `func_c` and `func_d`. Thus the code in the calling function is smaller.

The <stdarg.h> header file

- **A new type is defined, `va_list`**
 - Used to declare stack pointers, pointers that reference the stack
- **`va_start()`**
 - A macro, initialises a stack pointer with the last of the fixed parameters
- **`va_arg()`**
 - A macro, retrieves each data item from the stack via a stack pointer
 - The type of each data item must be known
 - `va_arg` may be called repeatedly
 - There must be some mechanism to determine when the last item has been retrieved
- **`va_end()`**
 - A macro, must be called when the stack processing is completed



The header file `stdarg.h` creates a new type called `va_list` which should be used to declare a pointer to the stack. This is done simply as follows:

```
va_list myPointer;
```

The pointer is initialised to point just under the last of the fixed parameters. Since all variadic functions must have fixed parameters, this is no real problem.

```
va_start(myPointer, last_fixed_parameter);
```

The `va_arg` macro is called for each parameter to be retrieved. For example, let us say we know there is an `int`, followed by a `double`, followed by an `int`:

```
i1 = va_arg(myPointer, int);
d  = va_arg(myPointer, double);
i  = va_arg(myPointer, int);
```

The thing to notice here is "let us say we know". The fact is, we don't know. The macros present the stack to us in exactly the format we ask. If the code was:

```
f1 = va_arg(myPointer, float);
i1 = va_arg(myPointer, int);
i2 = va_arg(myPointer, int);
f2 = va_arg(myPointer, float);
```

Then the same number of bytes would have been retrieved from the stack, just in the wrong order and the wrong format. Retrieving floats from the stack is a mistake since the K&R convention is used and the float is passed as double.

When the stack manipulation is finished the `va_end` macro is called:

```
va_end(myPointer);
```

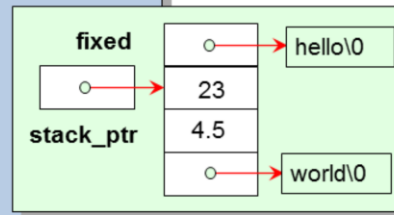

<stdarg.h> example

```
#include <stdarg.h>

void va_example(const char * fixed, ...)
{
    va_list stack_ptr;

    va_start(stack_ptr, fixed);
    {
        int i    = va_arg(stack_ptr, int);
        double d = va_arg(stack_ptr, double);
        char *pc = va_arg(stack_ptr, char *);
        /* some other code . . . */
    }
    va_end(stack_ptr);
}

int main(void)
{
    va_example("hello", 23, 4.5, "world");
    return 0;
}
```



The code above is an illustration of how the "va_" macros work. The va_start macro basically boils down to:

```
stack_ptr = &fixed + sizeof(fixed);
```

Thus the pointer is set to the item below "fixed" (the last of the fixed parameters and hence the start of the chaos), i.e. the first of the variadic parameters. The first invocation of the va_arg macro boils down to something like:

```
*(int*)((stack_ptr += sizeof(int)) - sizeof(int))
```

This one is a bit more complicated. Basically, it moves the stack pointer forwards by the size of an integer. Thus it moves beyond the current item onto the next. However, moving the pointer forwards is only half of what needs to be done. We also need to retrieve the value that **was** pointed to before the move. Thus, having added sizeof(int) and hence permanently changed the pointer sizeof(int) is subtracted temporarily generating the previous value of the pointer.

The second and third invocations of va_arg become:

```
*(double*)((stack_ptr += sizeof(double)) - sizeof(double))
```

```
*(char*)((stack_ptr += sizeof(char*)) - sizeof(char*))
```

Notice that the last invocation should not read:

```
va_arg(stack_ptr, (char*))
```

with additional parentheses around char* since this would effect the cast at the start of the expression.

In practice, of course, you would never write the function in this way. It would be much better to pass these values into fixed parameters as these would be checked by the compiler.


Another example

```
char d[100];
ManyStrCopy(d, "hello ", "world", (char*)0);
ManyStrCopy(d, "he", "llo", " wo", "rld", (char*)0);

void ManyStrCopy(char * dest, const char * src, ...)
{
    const char * ps = src;
    va_list stack_ptr;

    va_start(stack_ptr, src);
    *dest = '\0';

    while (ps)
    {
        strcat(dest, ps);
        ps = va_arg(stack_ptr, const char *);
    }
    va_end(stack_ptr);
}
```



The example above shows how a variadic function may be used to concatenate a variable number of strings into a single destination buffer, `dest`. The strings are passed as a list of character pointers. The end of the string is marked by a NULL pointer, which must be provided by the calling function. If the calling function fails to provide this special terminator value, the variadic function will continue to loop until it stumbles upon a NULL value somewhere on the stack!

This is an example of an implicit terminator value appearing in the variable-argument list. In general, this is perhaps the simplest and most widely used technique for implementing variadic functions, but note that it relies on each of the variadic arguments being of the same type (char pointers in this case).

The `ManyStrCopy()` function takes two fixed arguments: `dest` and `src`. The function insists on these two arguments being supplied, along with any additional strings that are to be concatenated at the end of `dest`. The function declares a `va_list` variable `stack_ptr` to step along the list of variable arguments until a NULL pointer is found. A working pointer variable `ps` is also declared to point to the next string to be concatenated to `dest`.

Before entering the main loop, the destination buffer is initialised with a null terminator at position zero. This is necessary because of the way `strcat()` works: it always appends the next string after the terminator of the destination buffer. Placing a null terminator at the beginning of the destination buffer ensures that the first string will be placed right at the beginning of the buffer.

Once the final string has been appended to `dest`, the `va_end` macro is invoked to tidy up `stack_ptr`, and the function terminates.

vprintf, vsprintf, vfprintf

- Instead of taking a variable argument list, these functions take a *va_list* parameter

```
/* stdio.h */
int vprintf(const char * format, va_list arg_ptr);
```

```
void AtPrint(int row, int col, const char * format, ...)
{
    va_list arg_ptr;
    Goto(row, col);
    va_start(arg_ptr, format);
    vprintf(format, arg_ptr);
    va_end(arg_ptr);
}
```

They offer a convenient way of calling printf-style routines from your variadic function

```
AtPrintf(10, 5, "%s, %f", "Radio 1", 99.50);
```

- C99 added **vscanf, vsscanf, vfscanf**

The example uses the function `vprintf` to write new versions of `printf`. `AtPrint()` function takes three fixed parameters, followed by a variable-argument list:

`int row, col` - column and row at which text should be printed
`const char * format` - printf-style format string
`... argument list` - printf-style variable-argument list

`AtPrint()` positions the cursor at text position (row, col) by calling a function such as `Goto()` which is supplied with the some compilers.

Having moved the cursor to the appropriate position, `AtPrint()` then calls `vprintf()` to print the list of variadic arguments. The `vprintf()` function is similar to `printf`, but takes a single argument of type `va_list` instead of a variadic list of arguments. The format argument has the same form and meaning as the format argument for `printf()`.

There are corresponding functions to print a formatted string to a FILE or to a character buffer:

```
int vfprintf(FILE * fp, const char * format, va_list arg_ptr);
int vsprintf(char * buffer, const char * format, va_list arg_ptr);
```

C99 added library support for the `v*scanf` functions. These had been implemented in some libraries for many years, and the standard has finally formalised them.

Dates and times

- **Date and time calculations in the Standard C library were derived from routines developed under UNIX**
 - These UNIX routines were developed by amateur astronomers who measured everything by Greenwich Mean Time
 - GMT is now called "Coordinated Universal Time" or "UTC"
- **The C Standard says a system need only provide its "best approximation" to the current time and date**
 - In practise this means a program should never take times too seriously
 - It is possible to enquire as to the current time and to display it in a number of attractive formats. It is not possible to know whether this is actually meaningful



Several developers of the UNIX operating system were amateur astronomers who used to thinking in Greenwich Mean Time (GMT). GMT itself is now known as UTC supposedly standing for a rather bewildering "Universal Coordinated Time". Cynics say that the ANSI committee, consisting mainly of Americans didn't have a clue where Greenwich was, or even in which continent it was to be found; hence the name change.

These amateur astronomers also recognised the need for representing times over a span of decades rather than just years.

The C Standard contains a number of "weasel words" in that an implementation need only provide a "best approximation" to the current time. Vendors could therefore write their routines to return "10:29am Thursday 17th December 1987" and justify this as a best approximation. This would still conform to the Standard.

Date and time data types

- **time.h provides the following data types:**
 - `time_t` an arithmetic type capable of representing times
 - usually implemented as an unsigned long int
 - `clock_t` an arithmetic type capable of representing times
 - usually implemented as an unsigned int
 - `struct tm` a structure containing the broken down time
- **The data type `time_t` usually represents the number of seconds since a particular date and time.**
 - Midnight 1st Jan 1970 is popular
 - Midnight 1st Jan 1980 is too
 - Midnight 1st Jan 1900 is as well !

Calculating from 1st Jan 1900 and estimating 32 bits in a `time_t` allows representation of dates and times until 2036

The Standard C header file `time.h` contains a number of types which are fundamental to the processing of times. These types are fairly nebulously defined by the Standard. "An arithmetic type capable of representing times" could mean anything. The type (whatever it is) is used to represent the number of seconds past a certain fixed point in time. Unfortunately, the certain "fixed point" in time is not fixed by the Standard. Different implementations use different fixed points.

Most implementations use long integers for the `time_t` type, because with 32 bits, numbers up to 4000 million may be represented.

Potentially nothing stops an implementation defining a `time_t` as a floating point type. Using a double, with its maximum value of around 10^{309} would take us up to year 3×10^{310} . That is roughly the year 3 million, with the word million repeated 51 times. Expressed in British billions (one million millions, rather than the American one thousand million), then that is 3 with the word billion repeated 25 times. That sort of time limit would see out the life of the sun.

The `struct tm` members are as follows. They may occur in any order and are all of int type:

member	description	range
<code>tm_sec</code>	seconds after minute	0 .. 61 (allows for "leap seconds")
<code>tm_min</code>	minutes after hour	0 .. 59
<code>tm_hour</code>	hours since midnight	0 .. 23
<code>tm_mday</code>	day of the month	1 .. 31
<code>tm_mon</code>	month since January	0 .. 11 (note: does not start at 1)
<code>tm_year</code>	years since 1900	0 upwards
<code>tm_wday</code>	days since Sunday	0 .. 6
<code>tm_yday</code>	day since January 1st	0 .. 365 (allows for leap years)
<code>tm_isdst</code>	daylight savings flag	any

`tm_isdst` is positive if daylight savings time is in force, 0 if it is not in force and negative if the information is not available.

Measuring processor time

```
clock_t clock(void);
```

According to the Standard, `clock()` "returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation".

- **To time a region of code**
 - Call `clock()` before and after the code
 - Calculate the difference
- **If the function is not available**
 - The value `(clock_t)-1` is returned
- **The number of clock ticks per second**
 - Is defined by the value `CLOCKS_PER_SEC`



The `clock()` function can be used to approximate the number of processor clock ticks that have elapsed between two points in the process. It is a useful function for benchmark comparisons of alternative algorithms to perform some task.


The `clock()` function is a standard routine and is prototyped in `time.h`. The return value is of type `clock_t`, which is usually typedef'd as an unsigned `int`. Note that `clock()` returns -1 if the processor elapsed-time cannot be established on the target environment.

The symbol `CLOCKS_PER_SEC` specifies the number of clock ticks per second and can be used in conjunction with `clock()` to evaluate the number of seconds that the process has taken to transfer control between two points in the code. On some older compilers, an alternative symbol `CLK_TCK` was provided, but this is now considered to be obsolete and should not be used if possible.

On most compilers (including Microsoft and GNU) the `clock()` function returns the total number of processor ticks that have occurred since the process was started. In other words, if you call `clock()` close to the start of a program, you will get a number close to zero. This behaviour, although sensible, is not required by the standard and should not be depended upon.

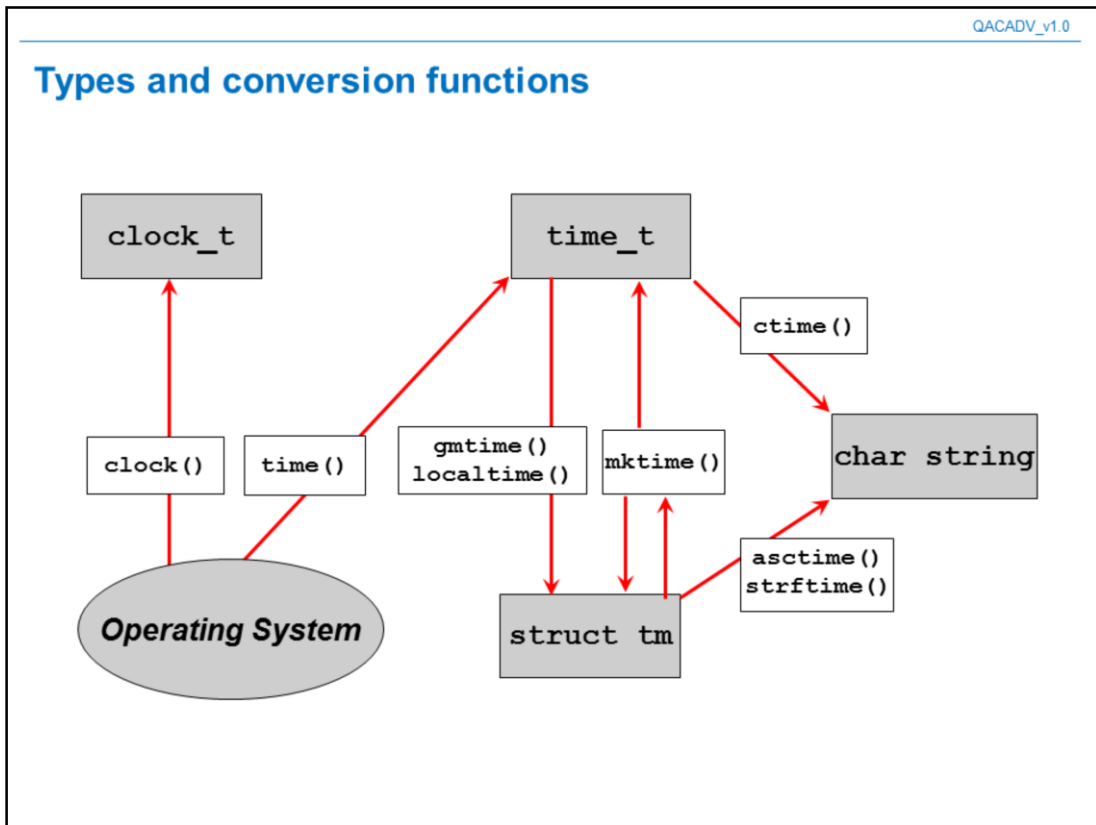
clock() example

```
#include <stdio.h>
#include <time.h>
void timed(void);
int main(void)
{
    clock_t start, end;
    start = clock();
    timed();
    end = clock();
    if (start == (clock_t)-1)
        printf("clock() does not work");
    else
        printf("Function 'timed' took %ld ticks "
               "(%0.2lf seconds at %ld ticks/sec)\n",
               (long)(end - start),
               (double)(end - start) / CLOCKS_PER_SEC,
               (long)CLOCKS_PER_SEC);
    return 0;
}
```



The example shown above illustrates how the `clock()` function may be used to calculate the processor time required to perform some action, which here is localised to the function called "timed".

The `clock()` function is called and the return value is stored in `start`, a variable of type `clock_t`. Note that a value of -1 indicates that the elapsed processor time could not be determined. The processing is done. The clock function is called a second time to initialise `end`. The two values are subtracted to give the elapsed time. This elapsed time is divided by `CLOCKS_PER_SEC` which will then indicate the number of elapsed seconds.



This slide summarises the various date and time functions provided by the ISO standard.

```
clock_t clock(void);
```

This function returns a number related to the processes start time.

```
time_t time(time_t * tod);
```

This is the starting point when you wish to determine the current date or time from the operating system. This function returns the current calendar time in seconds, or -1 if the target environment cannot determine the current time. If `tod` is not a NULL pointer, the calendar time is also written to `*tod`.

```
struct tm * gmtime (const time_t * tod);
```

```
struct tm * localtime(const time_t * tod);
```

These functions are similar; you would typically call one function or the other, but not both. Both functions take a `time_t` variable by reference (this value must have been set up by a previous call to the `time()` function). Both functions return a pointer to a `struct tm` holding the current date and time in a structured format. The difference between `gmtime()` and `localtime()` is that `gmtime()` converts the calendar time in `*tod` to GMT. The `localtime()` function, on the other hand, returns the local time (more on this later).

```
time_t mktime(struct tm * pt);
```

`mktime()` converts the time structure so that all members are normalised, e.g. `0 ≤ tm_min ≤ 59`. It then determines the values for the members `tm_mday`, `tm_wday` and `tm_yday` from the values of the other members. Finally, the function returns the corresponding calendar time in seconds.

```
char * ctime (const time_t * tod);
```

```
char * asctime (const struct tm * tptr);
```

These functions take a calendar time and return a formatted character string with the following format:

```
Wed Jan 02 02:03:55 1980\n\0
```


The calendar time can be obtained by a prior call to `gmtime()` or `localtime()`.

Example 1 - determining weekdays

What day of the week will Christmas day 2025 fall upon?

```
#include <stdio.h>
#include <time.h>
const char * weekday[]
    = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
int main(void)
{
    struct tm xmas = { 0 }; /* years are relative to 1900 */
    xmas.tm_year = 2025 - 1900; /* month range is 0..11 */
    xmas.tm_mon = 12 - 1;
    xmas.tm_mday = 25;
    if (mktime(&xmas) == -1)
        printf("mktime() failed\n");
    else
        printf("%s\n", weekday[xmas.tm_wday]);

    return 0;
}
```




The `mktime` function determines the `tm_wday` (weekday number) and `tm_yday` (day of the year) members from the other fields. Thus putting a date into the `tm_mday`, `tm_mon` and `tm_year` fields and calling `mktime` causes the weekday (amongst other things) to be determined.

Example 2 - date validation

- **mktime()** can be used for date validation

```
#include <stdio.h>
#include <time.h>
int main(void)
{
    struct tm t = { 0 };
    int dd, mm, cyy;
    printf("Enter date dd/mm/ccyy ");
    scanf("%d/%d/%d", &dd, &mm, &cyy);
    cyy -= 1900; /* year counts from 1900 */
    mm--; /* month counts from zero */
    t.tm_year = cyy;
    t.tm_mon = mm;
    t.tm_mday = dd;
    if (mktime(&t) == -1)
        printf("mktime() failed\n");
    else if (dd != t.tm_mday || mm != t.tm_mon || cyy != t.tm_year)
        printf("an invalid date\n");
    else
        printf("a valid date\n");
    return 0;
}
```



The `mktime()` function can be used to determine the validity of dates.

If a date such as 32/1/2012 is input, it is normalised into 1/2/2012. Since the day and month fields are now different from those originally input we can tell the original date was invalid.

Calling this prewritten function is far more convenient than all the tedious mucking around with the number of days in a month, the leap year calculations, the wrapping of December to January etc etc.

The one problem with the use of this function, is that it is not possible to predict the range of years that will be supported by a particular implementation. Dates from 1980 through to 2036 should be ok as most implementations will cover these. It is possible to write a small program to find out what date range `mktime()` will support.

Presumably as the year 2036 approaches different implementations of `mktime()` will appear. Perhaps by then 64 bit, or 128 bit variables will be in common use. The code will not need to change, just be relinked with the newer version of the Standard library.

Example 3 - formatting dates

```
#include <stdio.h>
#include <time.h>
enum { max_buf = 128 };
int main(void)
{
    time_t now;
    struct tm * ptime;
    char buf[max_buf + 1];

    time(&now);
    ptime = gmtime(&now);
    strftime(buf, max_buf, "%H:%M:%S", ptime);
    printf("GMT/UTC: %s\n", buf);

    ptime = localtime(&now);
    strftime(buf, max_buf, "%H:%M:%S", ptime);
    printf("local time: %s\n", buf);
    return 0;
}
```



**Print
GMT/UTC
time**



**Print local
time**

* C99 added some extensions to strftime()

In the example above, the current time is determined via a call to `time()`. `gmtime()` is then called to convert this value from seconds into a `struct tm` value representing GMT. The `strftime()` function is then called to build a formatted string holding the current time in GMT. `localtime()` is then called to convert the current time to local time, and `strftime()` is called again to build a string holding the local time. The C89 format specifiers for `strftime()` are listed below.

%	Description of <code>strftime()</code> format specifier	Example
%a	abbreviated day of the week	Fri
%A	full day of the week	Friday
%b	abbreviated month name	Nov
%B	full month name	November
%c	date and time	Nov 02 07:55:45 1981
%d	day of month (01 - 31)	31
%H	hour in 24-hour format (00 - 23)	23
%I	hour in 12-hour format (01 - 12)	12
%j	day of the year (001 - 366)	366
%m	month of the year (01 - 12)	01
%M	minutes after the hour (00 - 59)	59
%p	AM/PM indicator	AM
%S	seconds after the minute (00 - 59)	59
%U	week of year, Sunday as first day of week	51
%w	day of the week (0 - 6, where 0 = Sunday)	6
%W	week of year, Monday as first day of week	51
%x	date	Nov 02 1981
%X	time	07:55:45
%y	year of the century (00 - 99)	81
%Y	full year number	1981
%z	time zone name, if any	EST
%%	percent sign	%

Time zones before the standard

- **The time routines use UTC**
- **Users around the world want to see their own local time**
 - In summer and winter
- **The environment variable TZ holds the following :**
 - 3 character name of winter (daylight savings) time zone
 - Number of hours shifted from UTC (may be positive or negative)
 - 3 character name of summer time zone

		Winter time zone name			Offset from UTC in hours	Summer time zone name		
Default	P	S	T		8	P	D	T
England	G	M	T		0	B	S	T

```
void print_time(void) /* Our function to print GMT and local time */
{
    time_t t = time(0); /* Current time in seconds */
    struct tm * pm = localtime(&t); /* Pointer to static time buffer */
    /* Compute local time */
    printf("Local Time: %s", asctime(pm));
    pm = gmtime(&t); /* Next, compute GMT */
    printf("GMT: %s", asctime(pm));
}

int main(void) // Example:tzset and various global timezone-related variables
{
    // tzname[0] Timezone name --> e.g. PST Pacific time
    // tzname[1] DST (Daylight Saving Timezone) --> e.g. PDT Pacific daylight
    // timezone Difference from GMT in seconds --> e.g. 28800, i.e. 8 hours
    // daylight DST enabled flag (1 if enabled) --> e.g. 1, ie DST is enabled

    putenv("TZ=PST8PDT"); /* Set up timezone environment variable */
    tzset(); /* tzset accounts for timezone/daylight */

    printf("\nTimezone name = %s", tzname[0]);
    printf("\nDifference from GMT = %ld seconds", timezone);

    if (daylight == 0)
        printf("\nDaylight savings not in force\n\n");
    else
        printf("\nDaylight savings in force\n\n");

    print_time();
    return 0;
}
```

Time zones after the standard

- **The TZ mechanism has a number of failings:**
 - Assumes time zones are shifted by a whole number of hours from UTC
 - Assumes that summer and winter time zones differ by one hour
 - Contains no information when conversion from summer to winter time takes place
- **The Standard uses "locales" to provide the information above and more besides**
- **Unfortunately it dictates that only a "C" locale (with no information at all related to times) need be defined.**
- **Locale names are not standard**
 - Windows uses german, french-canadian
 - Linux uses de_DE, fr_CA

The TZ environment variable idea was invented in the US. Unfortunately, the idea does not work as well as everyone counting in seconds from a fixed point in time. It assumes that all timezones are shifted by a whole number of hours from GMT. Although this is the case in the US, it is not so everywhere in the world.

There is also an assumption that there is a summer and winter time regime, and that these times will differ by one (and not two, or one half, hours).

A large amount of guesswork is made by the routines as to *when* the changeover is made from summer to winter time and back again. Various enhancements (hacks) were made to the TZ idea. Some manufacturers would tack extra information onto the end. Since it then became 200 yards long the ANSI/ISO committees decided the best place for the information was in a file.

The file contains not only time information, but also collating information. This gives details of how and where, for example, à, á and ä come in the alphabet with relation to "a".

The book "The Standard C Library" by P. L. Plauger (someone heavily involved in the formation of the Standard) gives all the code required to implement a working Standard C library. Although the code is not in the public domain, it may be used providing its use is acknowledged.

Problems with sprintf?

- Standard string formatting can cause problems:

```
#include <string.h> /* sprintf */
#include <unistd.h> /* getpid */
#define NSIZE 15
...
char szFile[NSIZE] = "output_file";

sprintf (szFile, "%s.%d", szFile, getpid());
```



- Avoid toasted memory with C99 function **snprintf()**

```
#include <string.h> // snprintf
#include <unistd.h> // getpid
#define NSIZE 15
...
char szFile[NSIZE] = "output_file";

snprintf (szFile, NSIZE, "%s.%d", szFile, getpid());
```

A common criticism of C strings is that they are inherently unsafe. The string library function `sprintf()` has been singled out as a particular trap for the unwary.

In the first example, we create a character array of 15 bytes containing a file name. The idea is that we then use `sprintf()` to add our process id onto the end (`getpid()` is a Unix library routine). Unfortunately, we only have 3 bytes spare on the end of our `szFile` array to put the 'dot' and the process id. This might work with small process id's, but would stomp on memory eventually. It might cause a crash, or might just overwrite data.

Just as safer versions of `strcat()` (`strncat()`) and `strcpy()` (`strncpy()`) have been introduced, so has a safer version of `sprintf()` - it just took longer for the standard to catch up. C99 have taken `snprintf()` from BSD 4.4, and about time too!

The second argument is the maximum number of bytes to copy, minus one for the zero terminator. In our second example only the first two characters of the process id will be appended to "output_file".

Of course another way of doing this is to use a C99 Variable Length Array.

Random numbers

- **The Standard defines two functions for generating pseudo-random numbers**
 - `srand()` - sets the seed for pseudo-random numbers returned by subsequent calls to...
 - `rand()` - computes a sequence of pseudo-random numbers in the range 0 .. `RAND_MAX`

```
#include <stdlib.h>
static unsigned long int next = 1;
int rand(void)
{
    /* RAND_MAX assumed to be 32767 */
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}
void srand(unsigned int seed)
{
    next = seed;
}
```

Two functions are provided in the Standard for generating pseudo-random numbers. Random numbers form the heart of many arcade games and also fulfil a useful purpose in data encryption and simulation software. The `rand()` function actually generates the next number in the sequence, while `srand()` seeds the generator.

The numbers generated are in the range 0 to `RAND_MAX`. This constant is defined in `stdlib.h` and is guaranteed to be at least 32767.

The Standard actually suggests the implementations shown above for these two functions. For a given seed, the same sequence of numbers will be generated, thus the next value is always predictable. A common mistake is to forget to call `srand()`. In this case, the program acts as if the system had called `srand(1)` at program startup, and the subsequent "random" numbers always follow exactly the same sequence.

Another common error is to call `srand()` with a fixed value, such as `srand(1000)`. Once again, if a fixed seed value is given, the next value will always follow the same sequence.

The best solution is to seed the random numbers from an unpredictable source, such as the system time. Since the system time will be different each time the program is run, the pattern of random numbers will indeed approximate very closely to a purely random sequence.

The implementation of `rand()` above is decidedly "16 bit", ie. it returns a number in the range 0 to 32767 (even on 32-bit systems).

Random numbers example

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
enum { die_size = 6 };
int main(void)
{
    int die_1st, die_2nd;
    time_t now = time(0);
    srand((unsigned int)now);
    do
    {
        die_1st = rand() % die_size + 1;
        die_2nd = rand() % die_size + 1;
        printf("Roll: %d %d\n", die_1st, die_2nd);
    }
    while (die_1st == die_2nd);
    return 0;
}

```

DICE.C

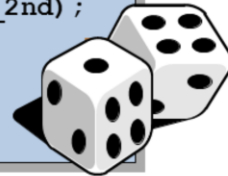
```

C:\>DICE
Roll: 4 1

C:\>DICE
Roll: 1 1
Roll: 2 5

C:\>

```



The example given above shows how `srand()` and `rand()` may be used to generate a sequence of pseudo-random integers in the range 1 to 6 inclusive.

The sequence of random numbers is initially seeded by calling `srand()` with the current system time. The current time is obtained by calling `time()`, and the resultant value is converted to an unsigned integer and passed to `srand()`. The cast is necessary because the `time_t` type is usually typedef'd as a long int.

The `rand()` function generates a random number in the range `0..RAND_MAX`, a standard symbol defined in `stdlib.h`. The code fragment

```
die_1st = rand() % die_size + 1;
```

generates a random number in the range 1 ... 6 to simulate a random throw of the die. The program keeps iterating while the two dice show the same value.

One way of testing the random number generator to see if it is truly random is to throw two dice a large number of times and see how many times a double six is achieved. If the test is repeated often enough, a double six should occur on average once in every 36 throws of the dice.

Summary

- **Variable argument processing: stdarg.h**
 - *va_list*, *va_start()*, *va_arg()*, *va_end()*
- **Date and time constants**
 - *CLOCKS_PER_SEC*
- **Date and time types**
 - *time_t*, *clock_t*, *struct tm*
- **Date and time functions**
 - *time()*, *clock()*, *mktime()*, *gmtime()*, *localtime()*, *strftime()*, *asctime()*
- **TZ vs. Locales**
- **Pseudo-random number generation**
 - *srand()* and *rand()*

