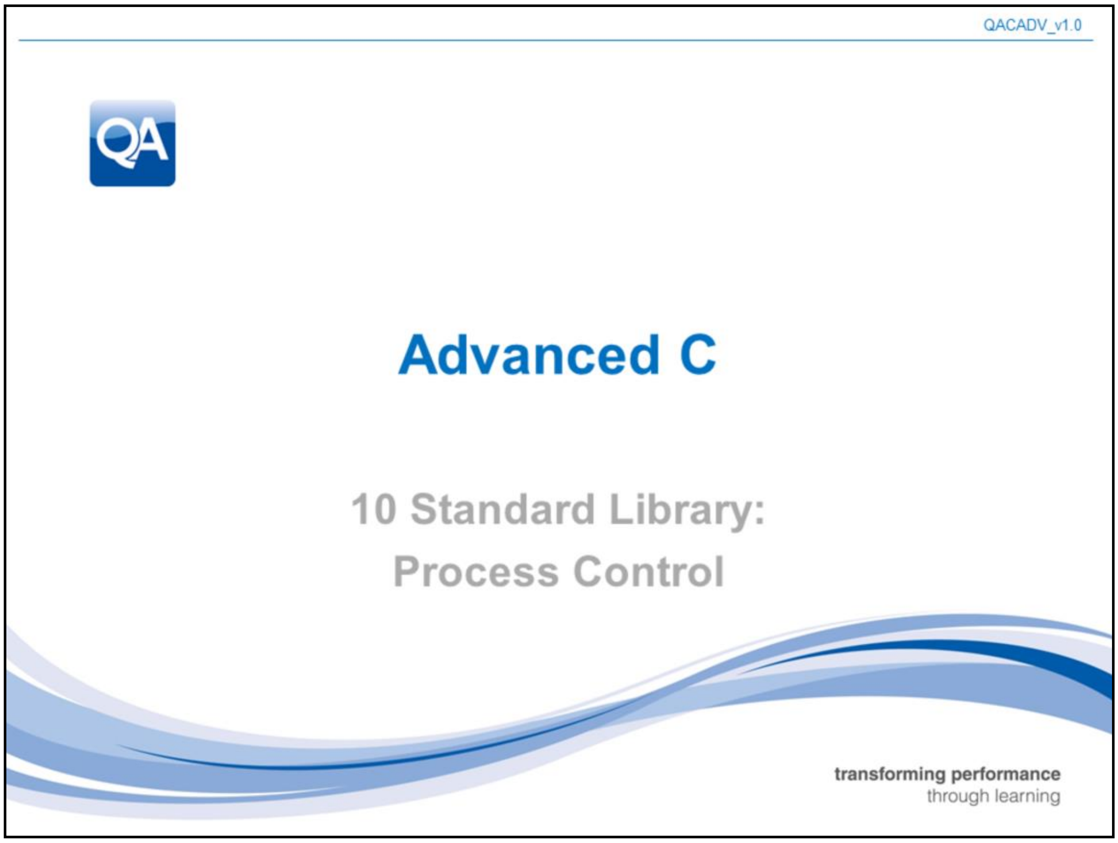
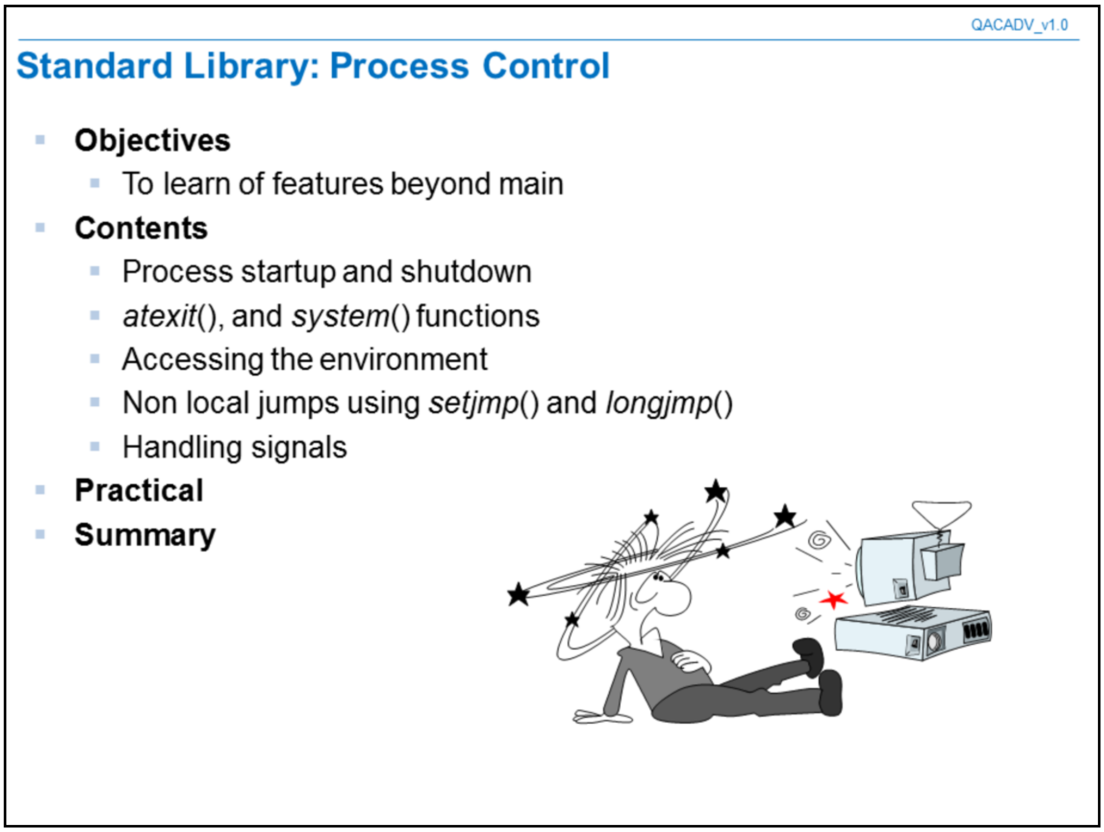


QACADV_v1.0

Advanced C

10 Standard Library: Process Control

transforming performance
through learning



QACADV_v1.0

Standard Library: Process Control

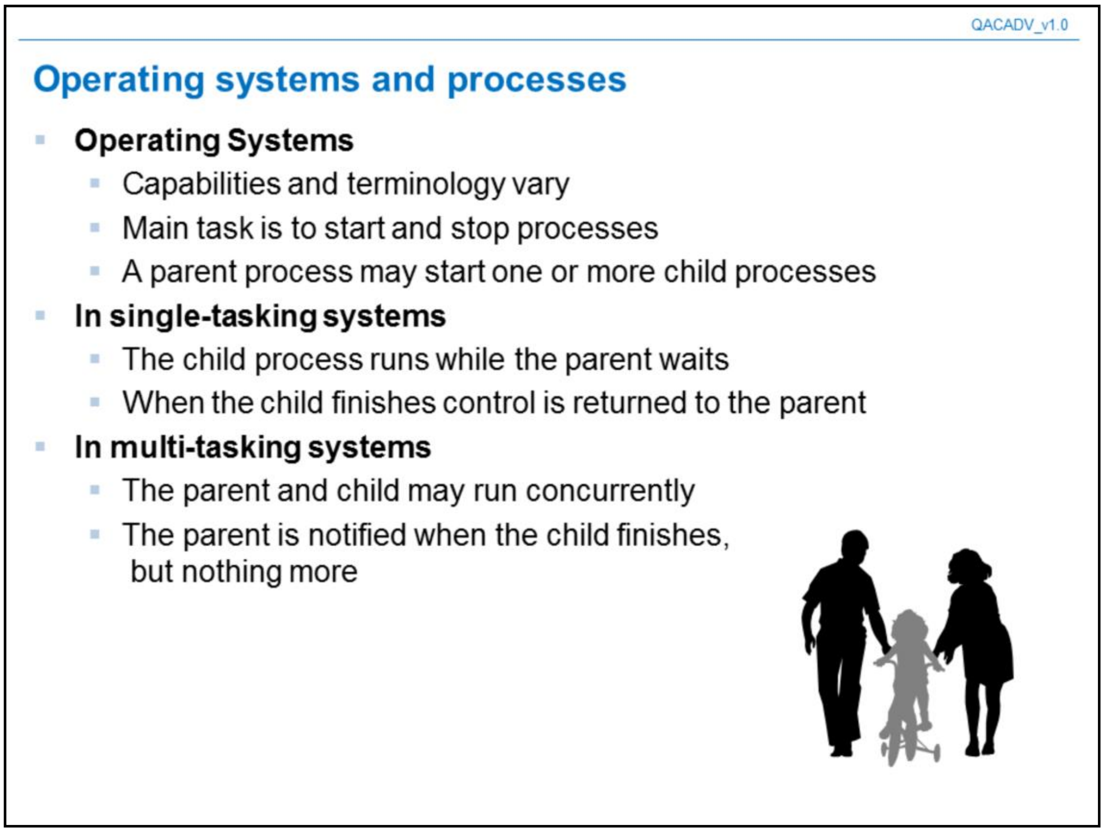
- **Objectives**
  - To learn of features beyond main
- **Contents**
  - Process startup and shutdown
  - *atexit*(), and *system*() functions
  - Accessing the environment
  - Non local jumps using *setjmp*() and *longjmp*()
  - Handling signals
- **Practical**
- **Summary**

In this chapter we show how to achieve process control in C. There are a number of functions in the standard library which may be used to start and terminate a process, including `system()`, `exit()` and `abort()`.

Environment variables offer one way of communicating information from one process to another. We shall look at `getenv()` in this chapter, a standard function for retrieving the current value of an environment variable. We shall also look at `putenv()`, a non-standard function which allows you to change the value of a particular environment variable during your process.

The chapter also shows how "exception handling" may be emulated in C using inter-function jumps via `setjmp()` and `longjmp()`, two standard C functions prototyped in the `stdarg.h` header file. There are important design issues involved here and we shall examine these in some detail.

The chapter closes with a discussion on signal handling. Firstly we explain what signals mean and how they are generated, then we look at how they can be intercepted in your programs and what you should do to recover from the event.



QACADV_v1.0

Operating systems and processes

- **Operating Systems**
  - Capabilities and terminology vary
  - Main task is to start and stop processes
  - A parent process may start one or more child processes
- **In single-tasking systems**
  - The child process runs while the parent waits
  - When the child finishes control is returned to the parent
- **In multi-tasking systems**
  - The parent and child may run concurrently
  - The parent is notified when the child finishes, but nothing more

One of the most basic services that an operating system must provide is the capability of starting and stopping processes. The term "process" is usually interpreted as meaning a "program in execution". It was first used in the 1960s when the Multics operating system was being designed.

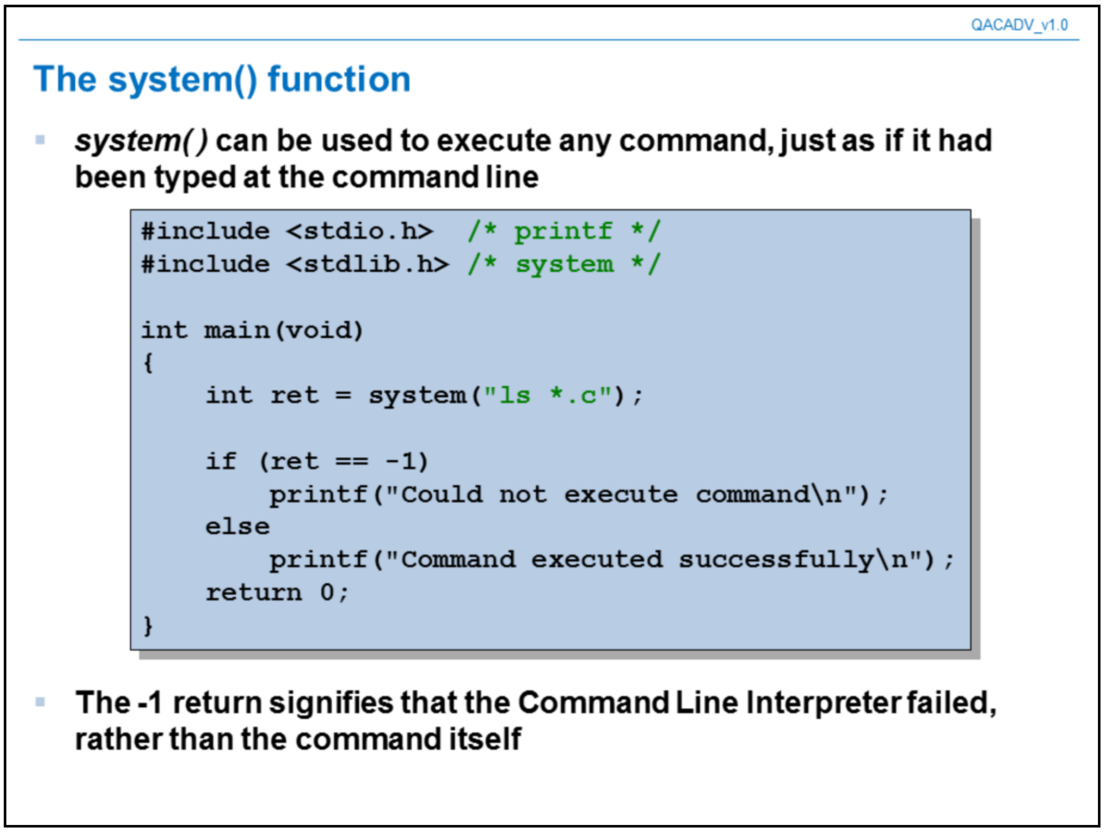
Operating systems allow a variety of operations to be performed on processes. Details vary from one operating system to another, but the most common operations are discussed below.

One process (the parent) may start another process (the child) by loading the instructions and data for the child process into memory and adding the new process to the list of current processes.

Only one parent is needed to create a child. This leads to a hierarchical process structure in which each child has one parent, but a parent may have many children.

In single-tasking systems such as MS-DOS, creating a new child process blocks the parent process until the child has finished. Only when the child has terminated will the parent resume its processing.

However, when a child process is created in a multi-tasking system such as Microsoft Windows, Unix, Linux or VMS, it is possible for the parent and child process to run concurrently.



The system() function

- *system()* can be used to execute any command, just as if it had been typed at the command line

```
#include <stdio.h>  /* printf */
#include <stdlib.h> /* system */

int main(void)
{
    int ret = system("ls *.c");

    if (ret == -1)
        printf("Could not execute command\n");
    else
        printf("Command executed successfully\n");
    return 0;
}
```

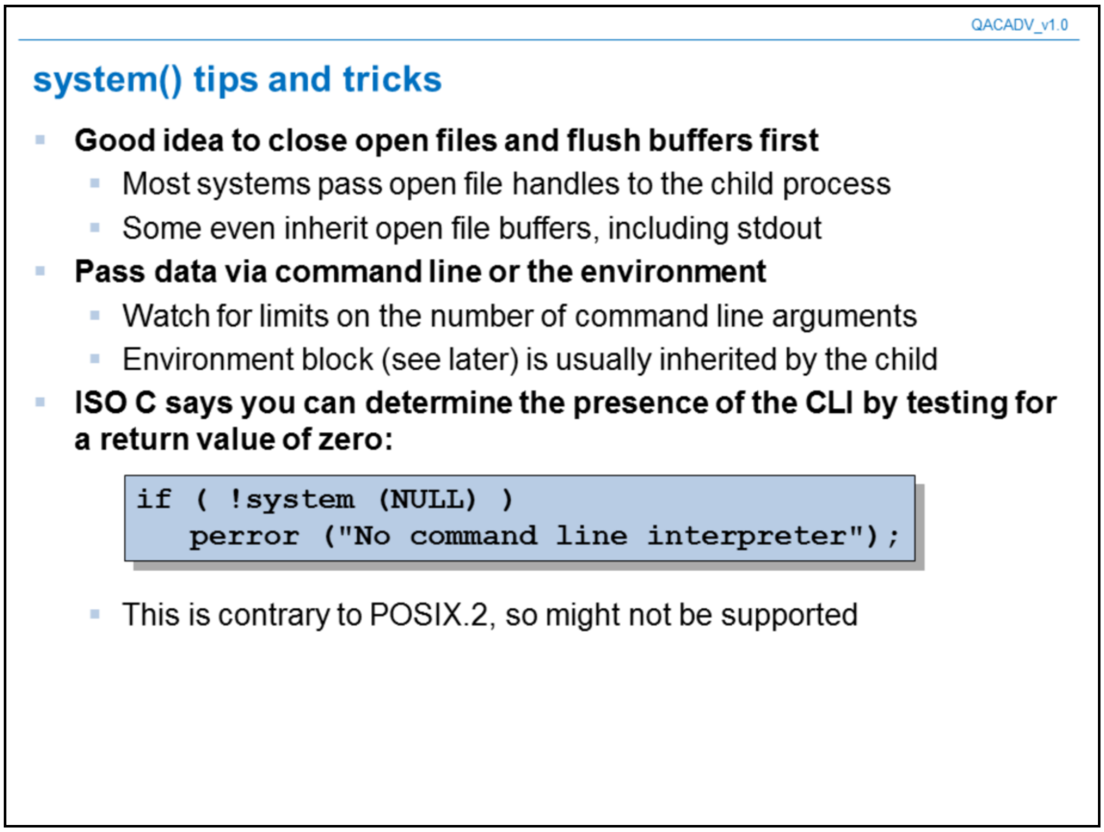
- The -1 return signifies that the Command Line Interpreter failed, rather than the command itself

The easiest way to execute one program from another is by using the ISO standard library function **`system()`**. This function takes one parameter, a character string, and passes it to the operating system's command line interpreter. A secondary copy of the command line interpreter (or shell) is loaded into memory to process the string as an operating system command. Under some multi-tasking systems the command may be executed concurrently.

The formal ISO definition of **`system()`** states that the return value is the status reported by the command line interpreter, and that this will vary from one environment to another. In general, `system()` returns the exit code of the command processor, which may or may not be that of the last command run.

The example given above shows how to use **`system()`** to invoke a simple Unix shell command to get a listing of the ".c" files in the current directory. Although **`system()`** is a standard function, the string that you pass to it might depend on which operating system you are using.

Generally the use of this function is considered a dirty cheat! It is very easy to use, but the load and execution of the shell can be a considerable overhead.



QACADV_v1.0

system() tips and tricks

- **Good idea to close open files and flush buffers first**
  - Most systems pass open file handles to the child process
  - Some even inherit open file buffers, including stdout
- **Pass data via command line or the environment**
  - Watch for limits on the number of command line arguments
  - Environment block (see later) is usually inherited by the child
- **ISO C says you can determine the presence of the CLI by testing for a return value of zero:**

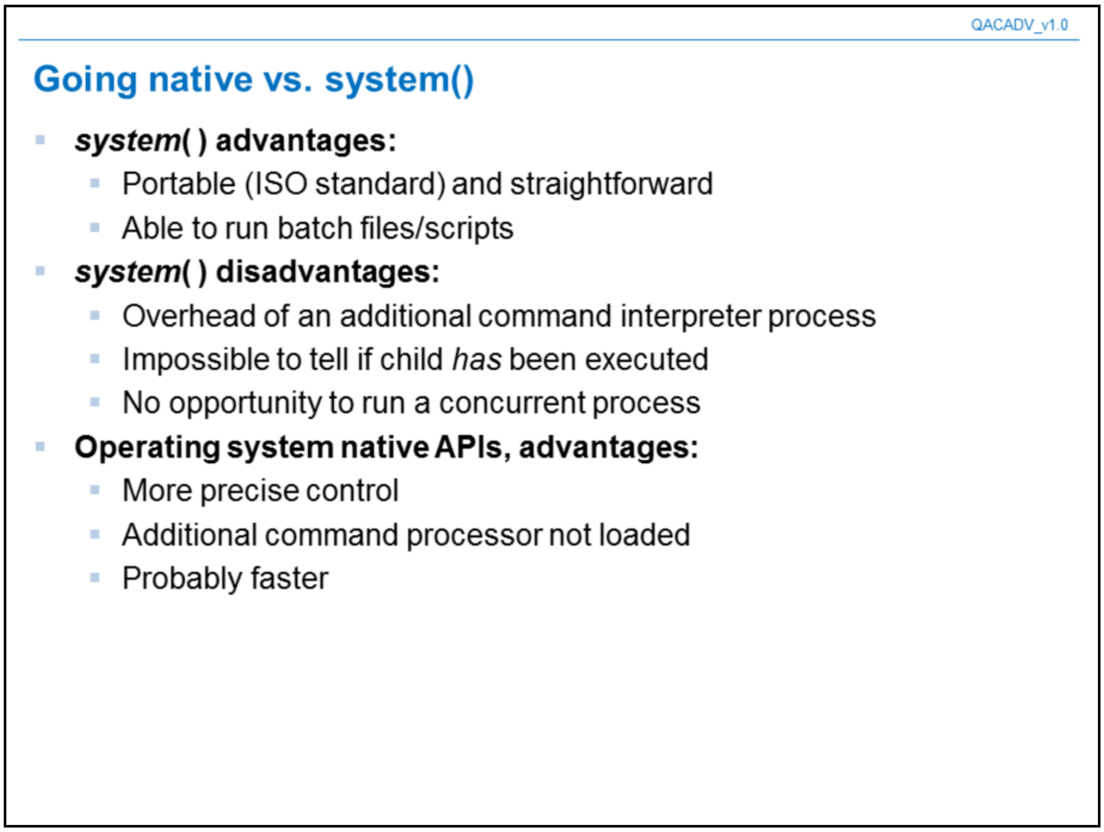
```
if ( !system (NULL) )
   perror ("No command line interpreter");
```

  - This is contrary to POSIX.2, so might not be supported

When a process creates another using the **`system()`** command then certain attributes are said to be *inherited* by the child process. Exactly what get inherited will depend on the operating system, but on many the child will get a copy of the parent's environment block, and any open file handles (or file descriptors). VMS is an exception, child processes do not inherit file handles on that OS. Some systems specifically instruct the programmer to **`fflush()`** all buffers before calling **`system()`**, and this applies to Microsoft Windows.

Data can be passed to the child process using command line arguments or environment variables, depending on the application. Notice that on many systems there is a limit on the number (or size) of the command line, a maximum of 255 items is common. We shall be discussing environment variables shortly.

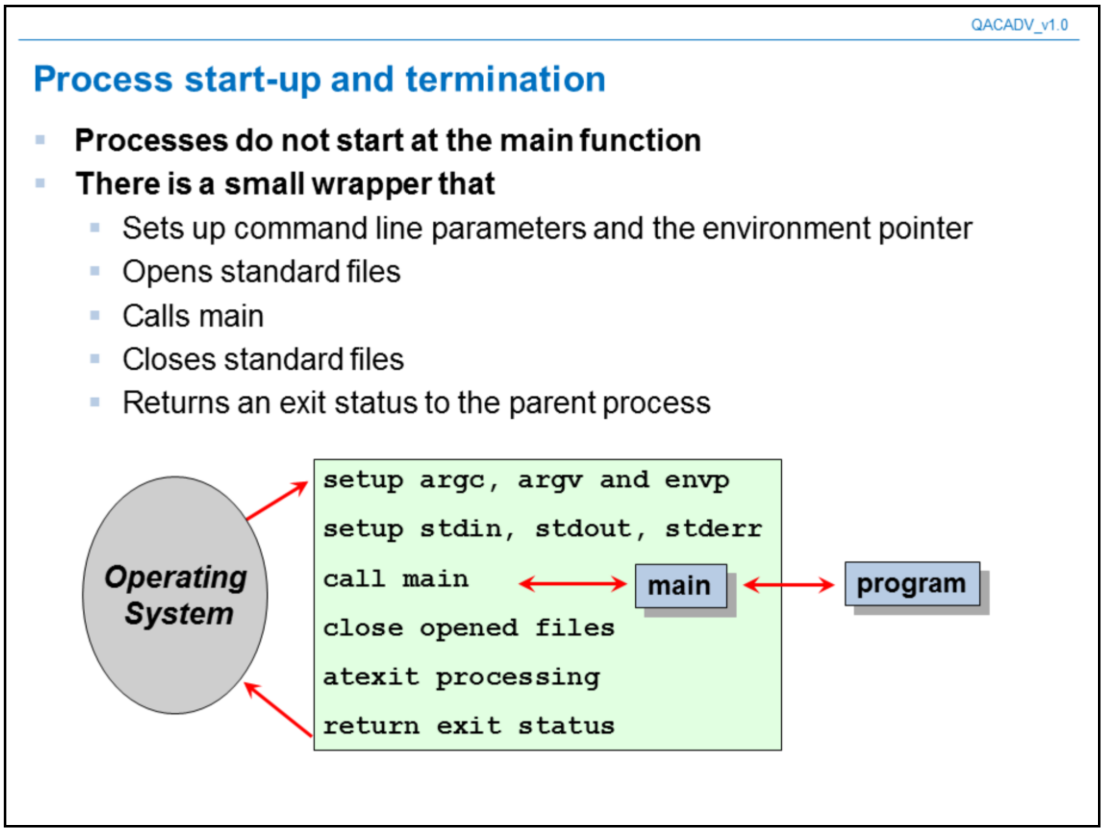
Note that `system()` may be invoked with a `NULL` parameter. In this case, a non-zero return value indicates that a command processor is available. Using the Microsoft run-time as an example, **`system()`** returns 0 if it is unable to run the command processor, and sets `errno` to `ENOENT`. POSIX.2, however, specifies that the return value must always be non-zero, so most Unix's and Linux return -1.


QACADV_v1.0

Going native vs. system()

- ***system*( ) advantages:**
  - Portable (ISO standard) and straightforward
  - Able to run batch files/scripts
- ***system*( ) disadvantages:**
  - Overhead of an additional command interpreter process
  - Impossible to tell if child *has* been executed
  - No opportunity to run a concurrent process
- **Operating system native APIs, advantages:**
  - More precise control
  - Additional command processor not loaded
  - Probably faster

This slide summarises the pros and cons of **`system()`** versus the native operating system's own process creation functions. By and large, the choice is quite straightforward: `system` is portable but native APIs are usually much faster and more powerful. Furthermore, they do not incur the overhead of having to load a secondary copy of the command processor.



QACADV_v1.0

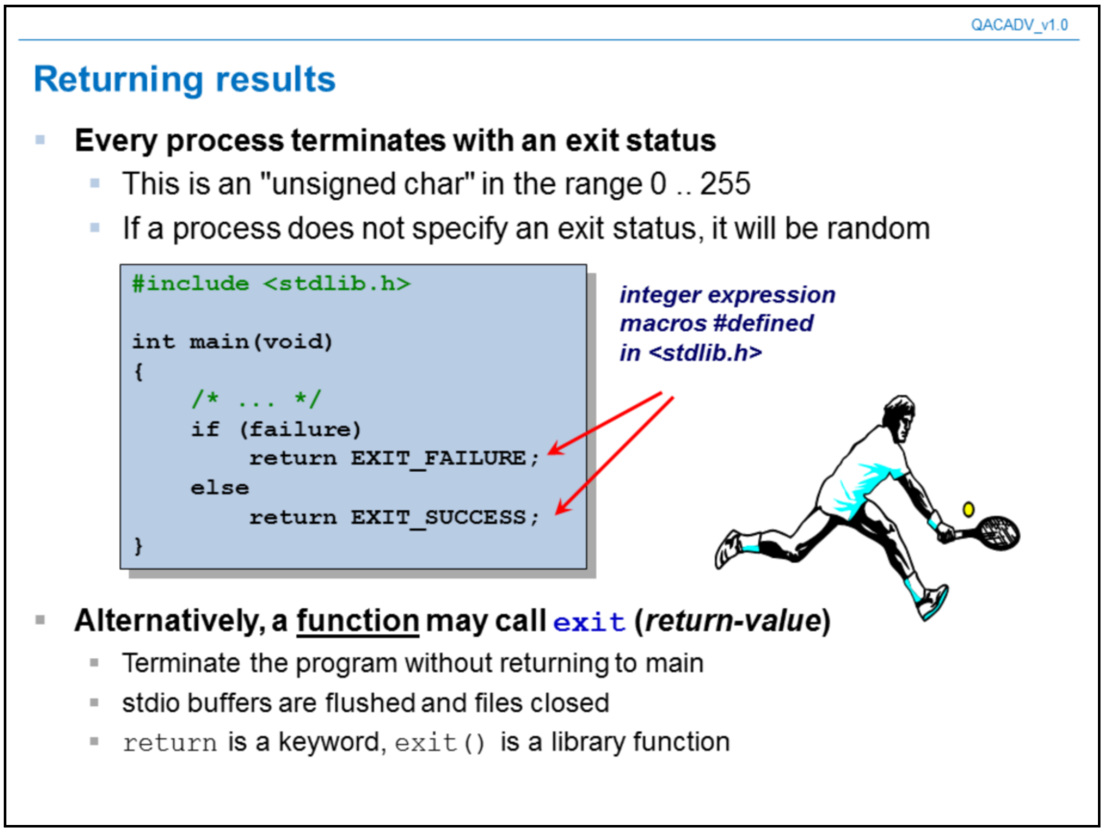
Process start-up and termination

- **Processes do not start at the main function**
- **There is a small wrapper that**
  - Sets up command line parameters and the environment pointer
  - Opens standard files
  - Calls main
  - Closes standard files
  - Returns an exit status to the parent process

Before the `main()` function starts in a program, a number of start-up tasks have already been performed. For example, the command-line arguments are parsed and used to build up `argv` and `argc` ready for `main()` to use. The standard input and output streams `stdin`, `stdout` and `stderr` are opened, and various default signal handlers are installed (see later in the chapter).

Similarly, when `main()` terminates, there are various clean-up tasks that must be performed before the program can terminate completely. Any open files must be flushed and closed; memory allocated dynamically must be released; signal and interrupt handlers must be restored to their original values, and so on.

These duties are carried out automatically via library code linked into the program. With some compilers alternative start-up routines may be chosen.



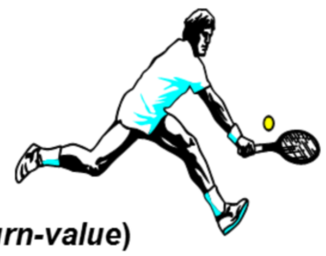
QACADV_v1.0

Returning results

- **Every process terminates with an exit status**
  - This is an "unsigned char" in the range 0 .. 255
  - If a process does not specify an exit status, it will be random

```
#include <stdlib.h>

int main(void)
{
    /* ... */
    if (failure)
        return EXIT_FAILURE;
    else
        return EXIT_SUCCESS;
}
```

integer expression macros #defined in <stdlib.h>



- **Alternatively, a <u>function</u> may call `exit` (*return-value*)**
  - Terminate the program without returning to main
  - stdio buffers are flushed and files closed
  - `return` is a keyword, `exit()` is a library function

Every process that runs is able to return an exit status – a number – to its parent. This number, which is used to indicate whether the child succeeded or failed, is ALWAYS sent. If a process does not specify an exit status, a random one will be returned. This number can only be in the range 0 to 255 inclusive, since operating systems only make one byte available. This range is further subdivided into 0 indicating success and 1 through to 255 indicating failure.

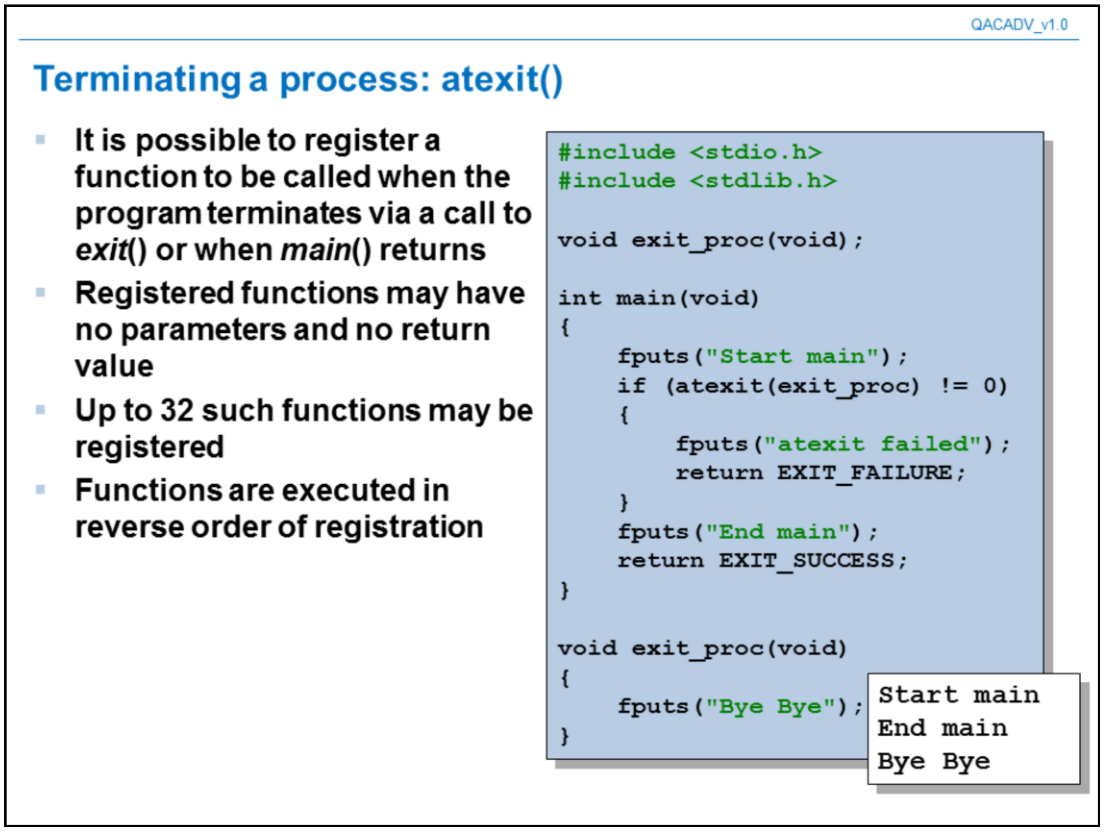
In C terms, returning a value of 0 to indicate success seems strange. The value of 0 in C indicates "false". This value was chosen there is only one way a program may succeed. It can find the file, open the database, allocate the memory, read the records and print them out. However, there are many ways a program can fail. The user might have spelled the name of the file wrong. The database might be corrupt, the memory allocation may fail, etc. etc.

The single value of 0 indicates the single possibility of success. The remaining 1 to 255 values indicate the many possible ways a program may fail.

Two values defined in the stdlib.h header file allow the distinction to be made between success and failure. There are no other standard exit status values.

In Windows .BAT files the exit status is placed in the ERRORLEVEL variable. Under most UNIX shells the exit status is placed in the $? variable (in csh it is placed in $status).

The **`exit()`** function can be called at any time, rather than returning a value from main. The integer parameter to exit is the return value from the program. People like to use: exit(-1); however the value returned will not actually be -1, but whatever the low byte of -1 looks like (usually 255).



QACADV_v1.0

Terminating a process: atexit()

- It is possible to register a function to be called when the program terminates via a call to *exit*() or when *main*() returns
- Registered functions may have no parameters and no return value
- Up to 32 such functions may be registered
- Functions are executed in reverse order of registration

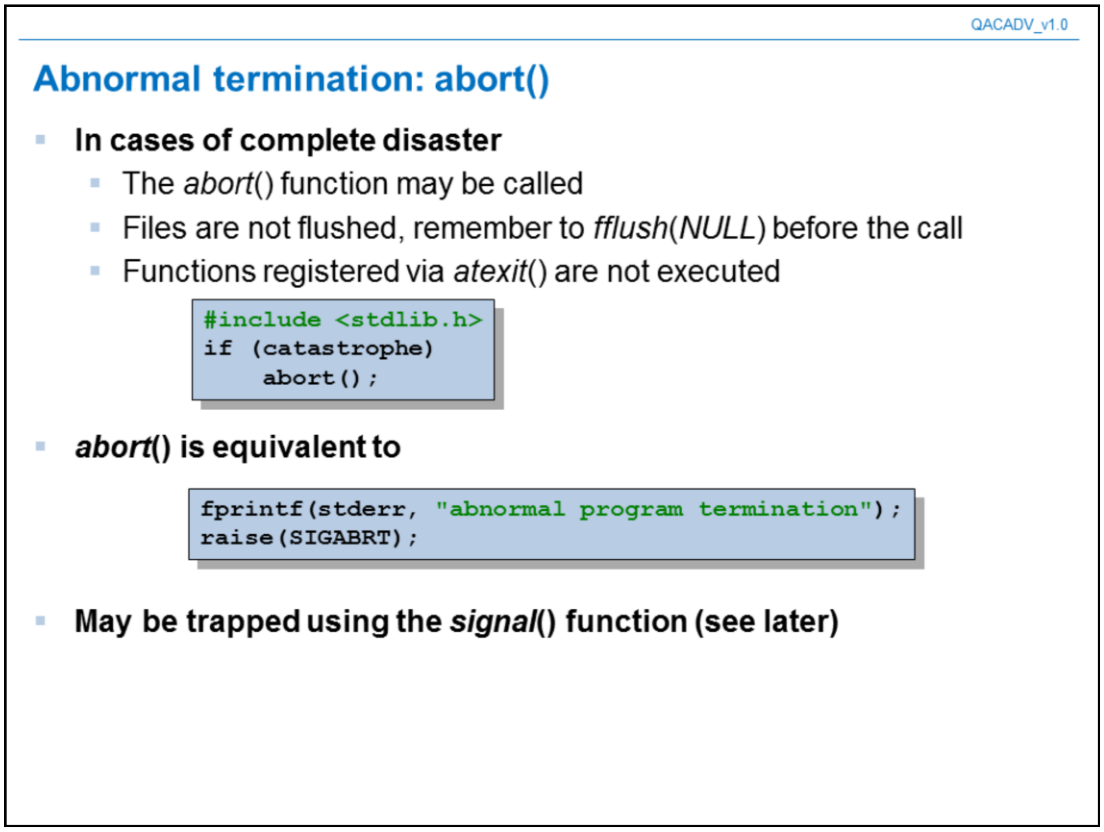
```
#include <stdio.h>
#include <stdlib.h>

void exit_proc(void);

int main(void)
{
    fputs("Start main");
    if (atexit(exit_proc) != 0)
    {
        fputs("atexit failed");
        return EXIT_FAILURE;
    }
    fputs("End main");
    return EXIT_SUCCESS;
}

void exit_proc(void)
{
    fputs("Bye Bye");
}
```

```
Start main
End main
Bye Bye
```

The **`atexit()`** function is prototyped in `stdlib.h` as follows:

```
    int atexit( void (*pfunc)(void) );
```

`atexit()` registers the function whose address is `pfunc`, so that **`exit()`** calls the function during program termination. The C standard states that up to 32 functions can be registered in this way, however some C run-times (e.g. Microsoft) limit this by the available memory on the heap (which may give greater or less that 32 functions). They will be called in reverse order of registration. This technique is useful if there are specific actions that need to be performed before the program terminates, such as closing down a serial communication link gracefully, or seeking confirmation from the user before saving any open files.



QACADV_v1.0

Abnormal termination: abort()

- **In cases of complete disaster**
  - The *abort*() function may be called
  - Files are not flushed, remember to *fflush*(*NULL*) before the call
  - Functions registered via *atexit*() are not executed

```
#include <stdlib.h>
if (catastrophe)
     abort();
```

- ***abort*() is equivalent to**

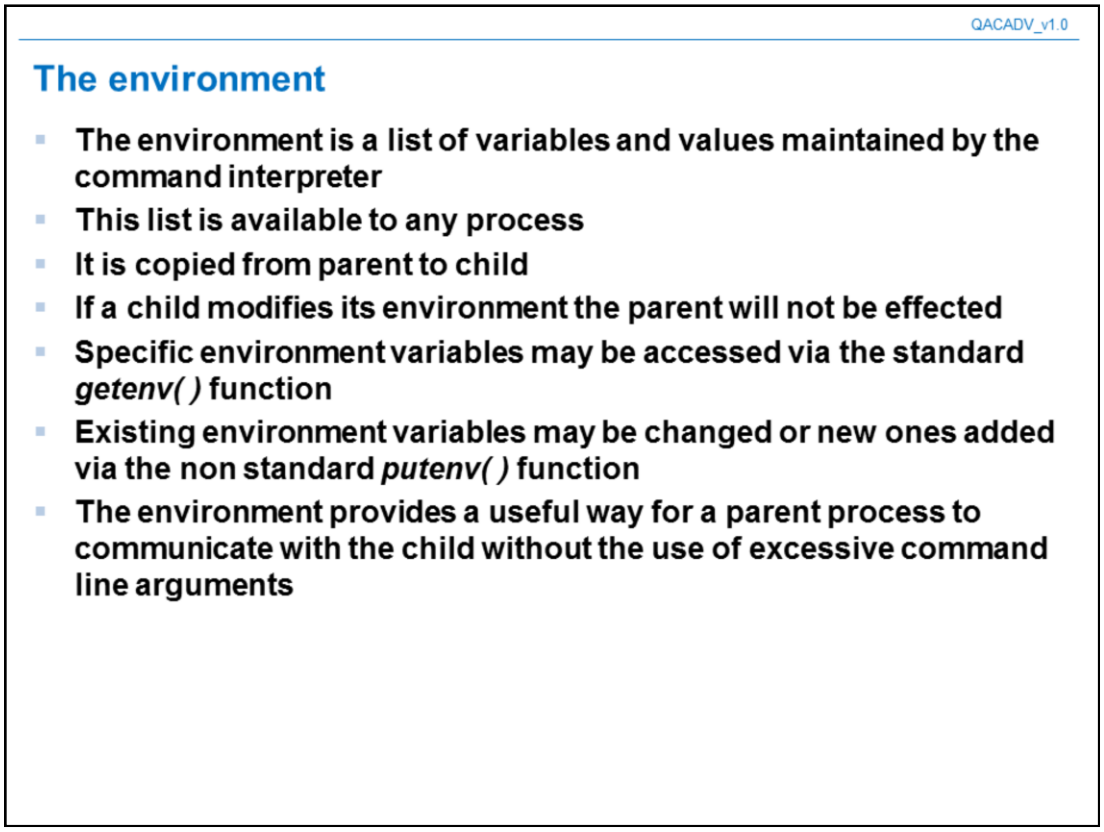
```
fprintf(stderr, "abnormal program termination");
raise(SIGABRT);
```

- **May be trapped using the *signal*() function (see later)**

`abort()` is useful in extreme circumstances when you need to terminate the program quickly without performing the standard shutdown procedures. For example, if you registered a function with **`atexit()`**, what should you do if you detected a critical error within this function? If you called `exit()` again, this would invoke all the functions registered via **`atexit()`** and you end up calling the same function over and over again! A call to **`abort()`** will avoid this problem.

The actual method of termination varies with the platform, for example with Microsoft Visual C++ **`abort()`** will generate a Debug exception if running in Debug, and an "abnormal program termination" (with exit code 3) when running in Release. Most Unix platforms produce a core dump.

If you do not want to the diagnostics associated with abort(), use _Exit() instead. This is an ISO C standard routine (stdlib.h) which exits the process at once without calling **`atexit()`** routines, without flushing IO buffers, and with no signal trapping.


QACADV_v1.0

The environment

- The environment is a list of variables and values maintained by the command interpreter
- This list is available to any process
- It is copied from parent to child
- If a child modifies its environment the parent will not be effected
- Specific environment variables may be accessed via the standard *getenv( )* function
- Existing environment variables may be changed or new ones added via the non standard *putenv( )* function
- The environment provides a useful way for a parent process to communicate with the child without the use of excessive command line arguments

When a process is created, it inherits a copy of its parent's environment. This is one way of communicating information from the parent process to the child. Since the parent is often a shell, most have commands to adjust environment variables:
Microsoft Windows: Environment variables are usually stored in the Registry. There are two set of environment variables, System and User, however these appear as one contiguous block to a C program. They may also be set from a command prompt, for example:

```
set PATH=C:\;C:\WINDOWS;C:\MSVC\BIN
set LIB=C:\MSVC\LIB
```

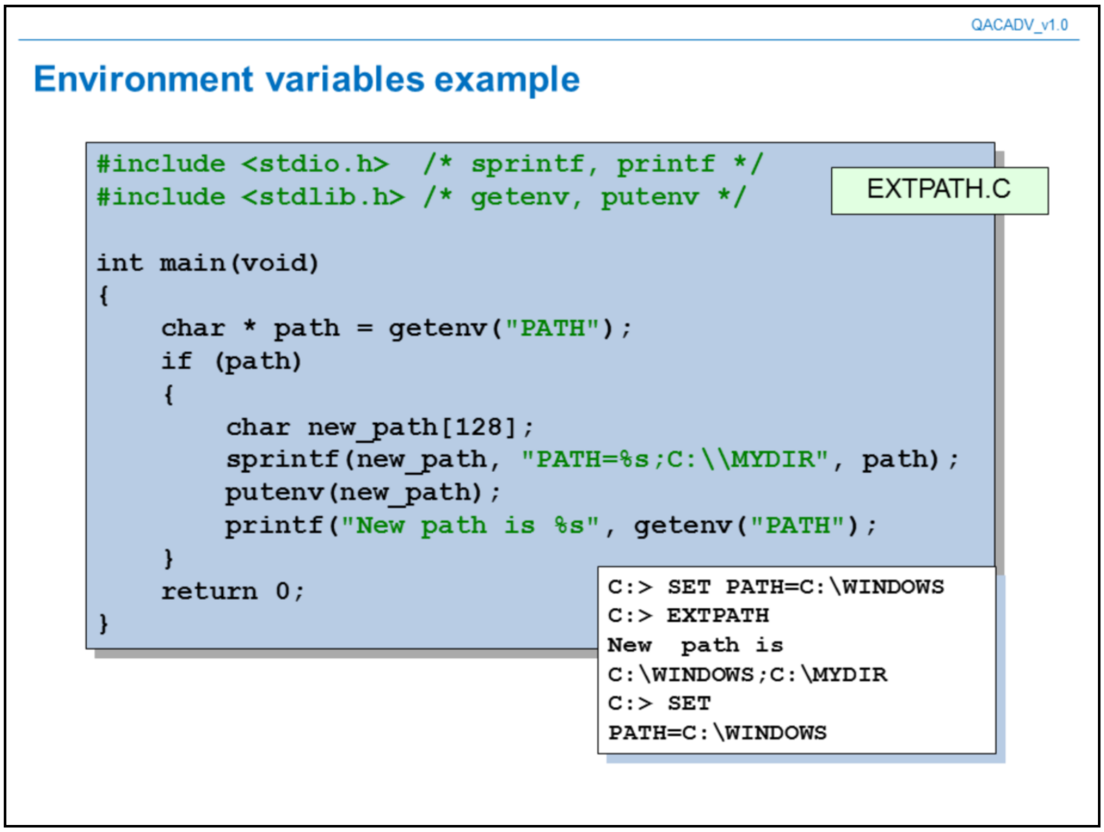
Unix Bourne/Korn/Bash shells

```
export PATH=${PATH}:/home/fred/bin
export LD_LIBRARY_PATH=${ORACLE_HOME}/lib
```

Unix C shell

```
setenv PATH ${PATH}:/home/fred/bin
setenv LD_LIBRARY_PATH ${ORACLE_HOME}/lib
```

A process may access a particular variable in its environment by calling the **getenv** function. Altering the environment is a little more tricky since the **putenv** function is not standard. Certain operating systems do not allow a process to modify its environment, these do not provide versions of putenv. Any changes made by a process to its environment effect only that process and its children. Since the parent COPIED its environment to the child, its own environment is uneffected. If the child needs to pass information back to the parent, normal Inter-Process Communication (IPC) mechanisms may be used, for example shared memory, pipes, or messages.



QACADV_v1.0

Environment variables example

EXTPATH.C

```
#include <stdio.h>  /* sprintf, printf */
#include <stdlib.h> /* getenv, putenv */

int main(void)
{
    char * path = getenv("PATH");
    if (path)
    {
        char new_path[128];
        sprintf(new_path, "PATH=%s;C:\\MYDIR", path);
        putenv(new_path);
        printf("New path is %s", getenv("PATH"));
    }
    return 0;
}
```

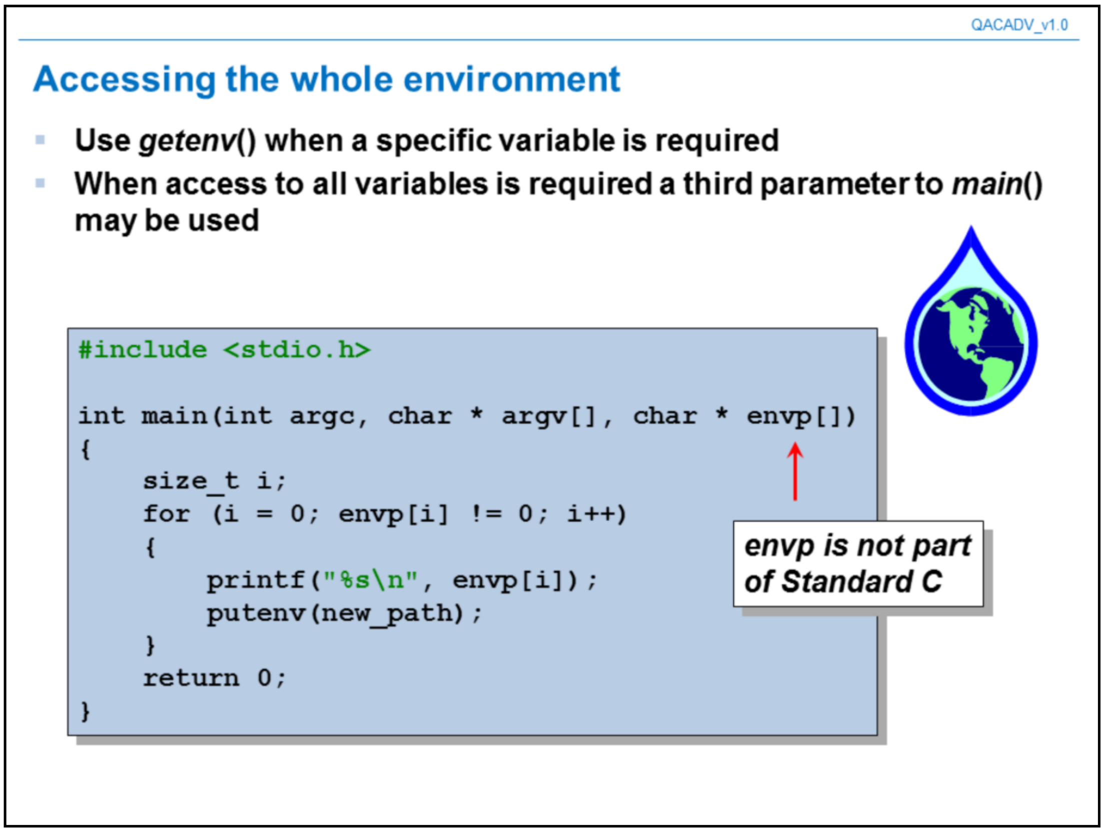
```
C:> SET PATH=C:\WINDOWS
C:> EXTPATH
New  path is
C:\WINDOWS;C:\MYDIR
C:> SET
PATH=C:\WINDOWS
```

The child process can use the ISO standard function `getenv()` to access individual environment variables. `getenv()` searches the environment list for the specified environment variable. A `NULL` return value indicates that the environment variable could not be found. Otherwise, `getenv()` returns a pointer to a static duration object containing the required environment variable setting. *Do not change this string directly!*

Some compilers offer the separate (and non-standard) function `putenv()` to modify an existing environment variable or to add a new one to the list. Use this function if you need to change an environment variable. Remember that these changes are local to your child process and will be lost when the process terminates. There is no way of permanently changing a setting in your parent's environment.

How do `getenv()` and `putenv()` work? Detailed implementation varies from one compiler to the next, but the Microsoft compiler is fairly typical. Under Microsoft C, `getenv()` and `putenv()` use the global variable `environ` to access the environment table. `putenv()` modifies this variable, but it is local to the program and is therefore lost when the program terminates.

In the code fragment shown in the example above, `getenv()` is used to determine the current `PATH`. The `sprintf()` function is then used to build up an extended `PATH` setting and `putenv()` is called to enforce the new `PATH` setting.



QACADV_v1.0

Accessing the whole environment

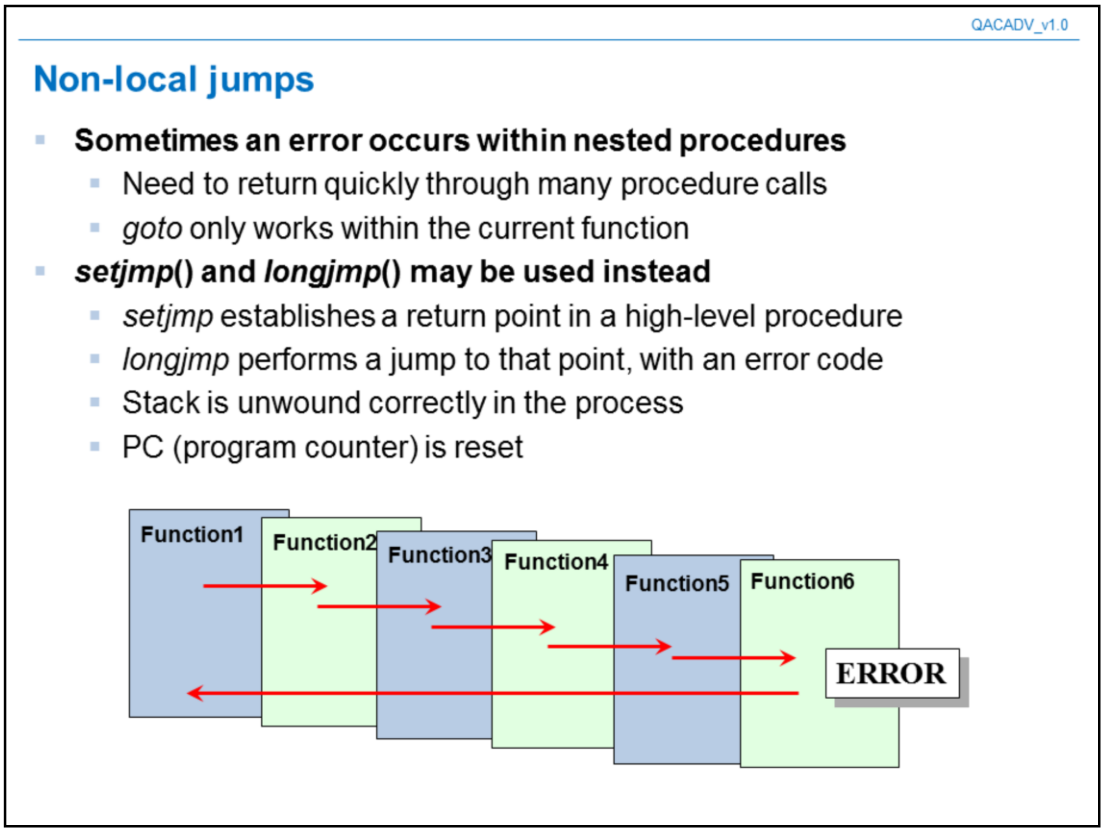
- Use *getenv*() when a specific variable is required
- When access to all variables is required a third parameter to *main*() may be used

```
#include <stdio.h>

int main(int argc, char * argv[], char * envp[])
{
    size_t i;
    for (i = 0; envp[i] != 0; i++)
    {
        printf("%s\n", envp[i]);
        putenv(new_path);
    }
    return 0;
}
```

envp is not part of Standard C

The `envp` pointer is set up just like the `argv` pointer. It too, is a pointer to an array of "string" and is used to scan through the individual environment strings.

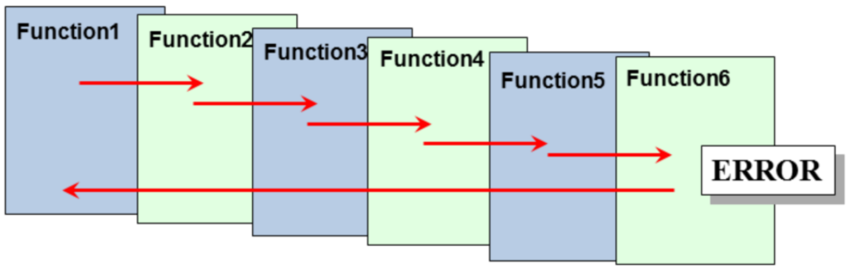
Note that the implementation of environment variable names is not portable. For example on Microsoft Windows environment variable names are not case sensitive (`getenv("Path")` gives the same as `getenv("PATH")`), whereas on Unix they are.



QACADV_v1.0

Non-local jumps

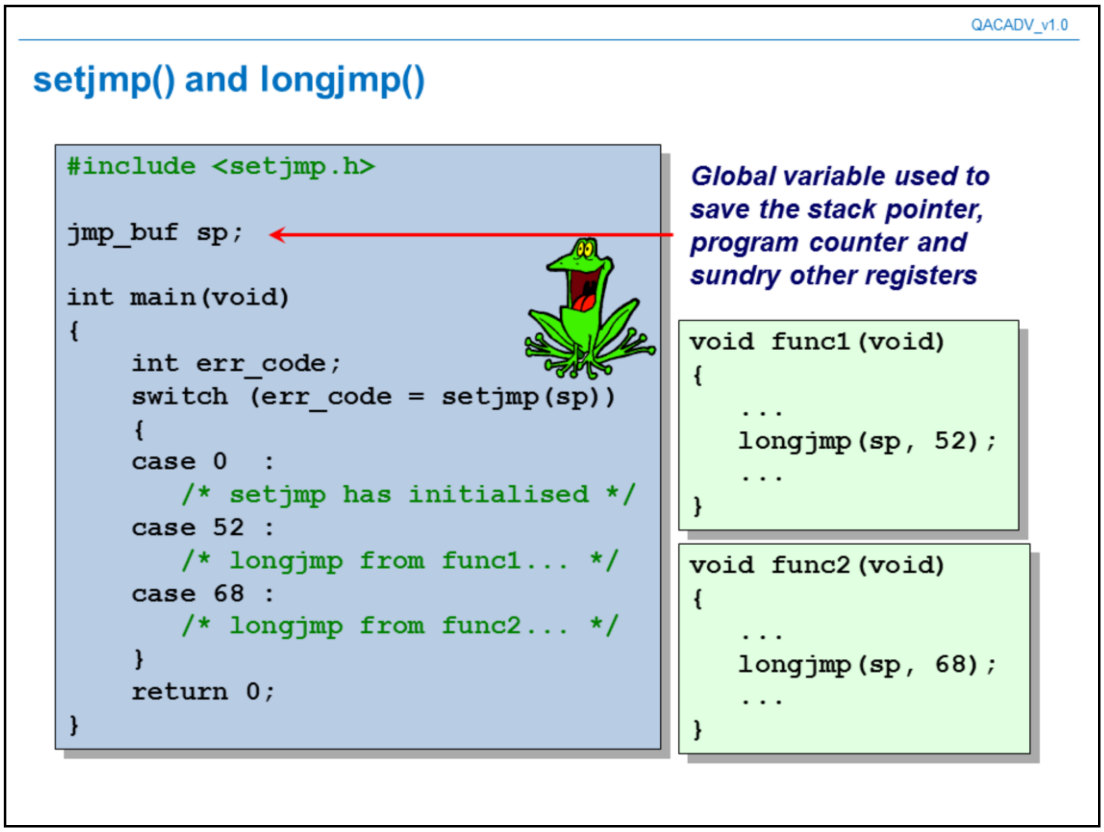
- **Sometimes an error occurs within nested procedures**
  - Need to return quickly through many procedure calls
  - *goto* only works within the current function
- ***setjmp*() and *longjmp*() may be used instead**
  - *setjmp* establishes a return point in a high-level procedure
  - *longjmp* performs a jump to that point, with an error code
  - Stack is unwound correctly in the process
  - PC (program counter) is reset



One of the most demanding tasks in many programs is that of recovering from errors that occur deep within nested functions. This situation frequently arises during file handling, where a great deal of file processing might already have taken place when a non-recoverable error occurs. In some situations it is often desirable to return quickly and directly to some higher-level function, without having to unwind through all the intermediate functions manually.

At first glance, the (dreaded!) `goto` statement seems like the ideal candidate, but closer inspection reveals that `goto` cannot be used to jump to a label in a different function, only to a label in the current function.

In fact, the ISO definition of C provides two standard functions `setjmp()` and `longjmp()` that fit the bill exactly. `setjmp()` establishes a recovery point in some high-level function. Subsequent lower-level functions are then called in the usual manner to perform some detailed processing. If one of these functions detects a serious error, the function can simply call `longjmp()` to abandon processing and return immediately to the original high-level function. Returning is done by resetting the program counter and stack pointer ("unwinding" the stack in the process).



QACADV_v1.0

setjmp() and longjmp()

```
#include <setjmp.h>

jmp_buf sp;

int main(void)
{
    int err_code;
    switch (err_code = setjmp(sp))
    {
    case 0  :
        /* setjmp has initialised */
    case 52 :
        /* longjmp from func1... */
    case 68 :
        /* longjmp from func2... */
    }
    return 0;
}
```

Global variable used to save the stack pointer, program counter and sundry other registers

```
void func1(void)
{
    ...
    longjmp(sp, 52);
    ...
}
```

```
void func2(void)
{
    ...
    longjmp(sp, 68);
    ...
}
```

The first step is to include the header file `setjmp.h`. This not only prototypes the two functions, `setmp()` and `longjmp()` but typedefs a `jmp_buf` as an architecture specific data structure. This will vary between machines, operating systems and compilers.

Just in case you are wondering how `setjmp()` can alter its `jmp_buf` parameter, it is usually declared as some form of array. Thus call by reference is achieved.

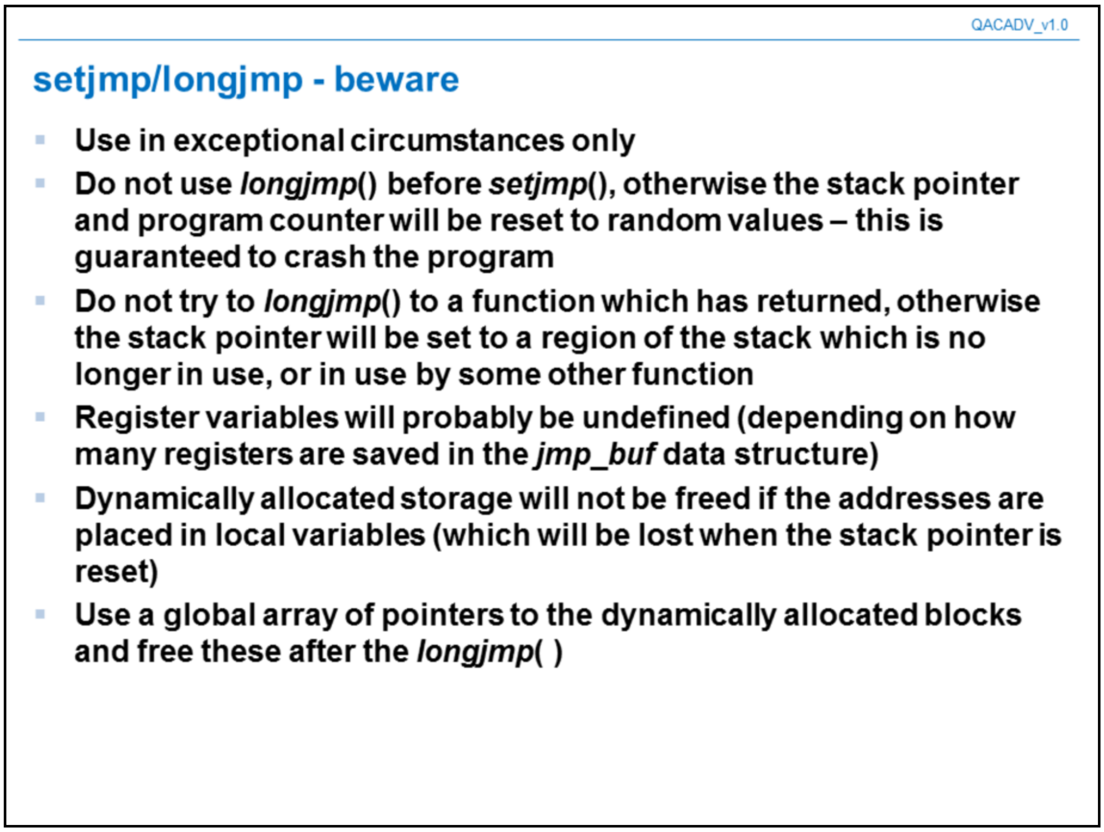
When `setjmp()` is called, the current stack pointer, current instruction address and various registers are stored in the data structure. All this information is needed to reinstate the current execution status if a longjump occurs back to this point at some later stage in the program's execution.

When `setjmp()` initialises the jump buffer it returns 0.

If some lower-level function calls `longjmp()` the program counter and stack pointer are reset. This has the effect of making it look as though `setjmp()` has just been called. The difference between the *actual* call and the jump back is the "return" value. When initialising `setjmp()` gives a value of 0, when jumping back, `setjmp()` gives the second parameter to `longjmp()`.

If zero is passed as the second parameter to `longjmp()`, it is changed to 1 to avoid "returning" 0 and fooling the main function into thinking the jump buffer had just been initialised.

What you do in response to a long jump will clearly depend on your application. For example, if you are in the middle of some complex file handling, you might choose to "roll back" the current operation, ignore any changes, and revert to the files you started with before the file operation began.

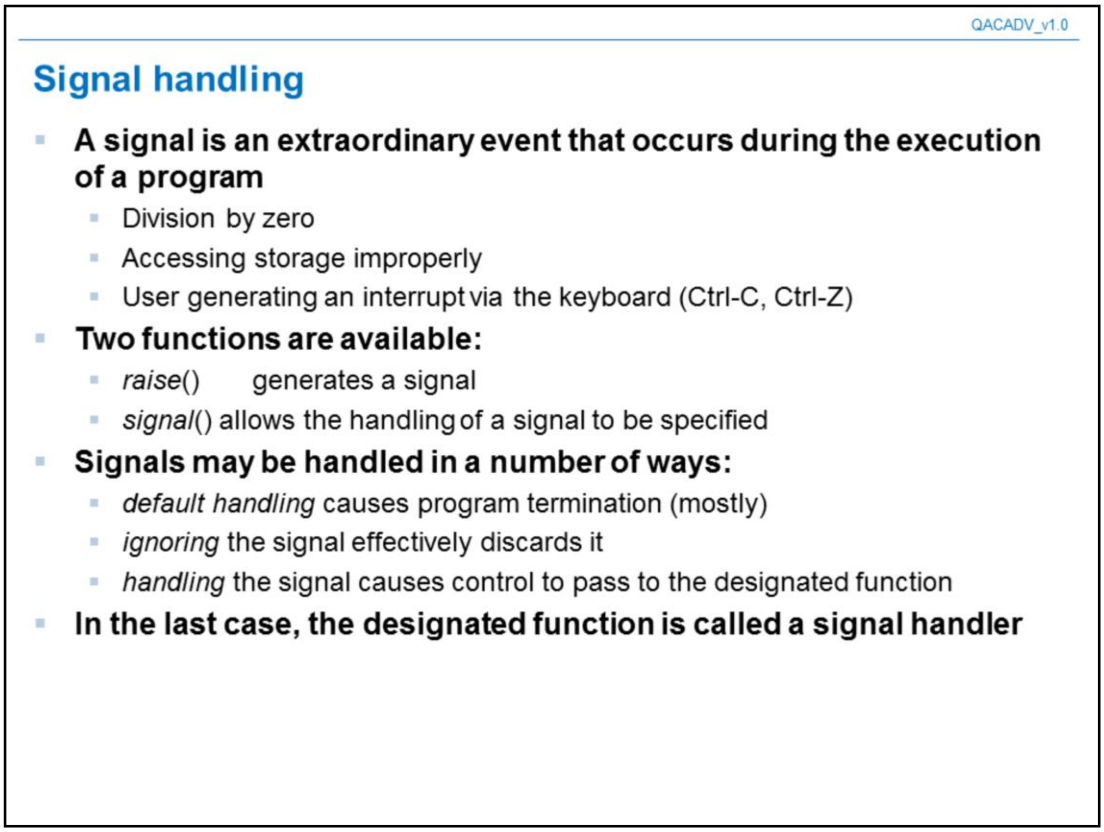

QACADV_v1.0

setjmp/longjmp - beware

- Use in exceptional circumstances only
- Do not use *longjmp*() before *setjmp*(), otherwise the stack pointer and program counter will be reset to random values – this is guaranteed to crash the program
- Do not try to *longjmp*() to a function which has returned, otherwise the stack pointer will be set to a region of the stack which is no longer in use, or in use by some other function
- Register variables will probably be undefined (depending on how many registers are saved in the *jmp_buf* data structure)
- Dynamically allocated storage will not be freed if the addresses are placed in local variables (which will be lost when the stack pointer is reset)
- Use a global array of pointers to the dynamically allocated blocks and free these after the *longjmp*( )

Use setjmp and longjmp in exceptional circumstances only. Otherwise a program can quickly lose its structure and become extremely difficult to maintain and update in the future.

Remember you are (more or less) directly manipulating the program counter and the stack pointer. This is not to be undertaken lightly, if mistakes are made the program is unlikely to recover.



QACADV_v1.0

Signal handling

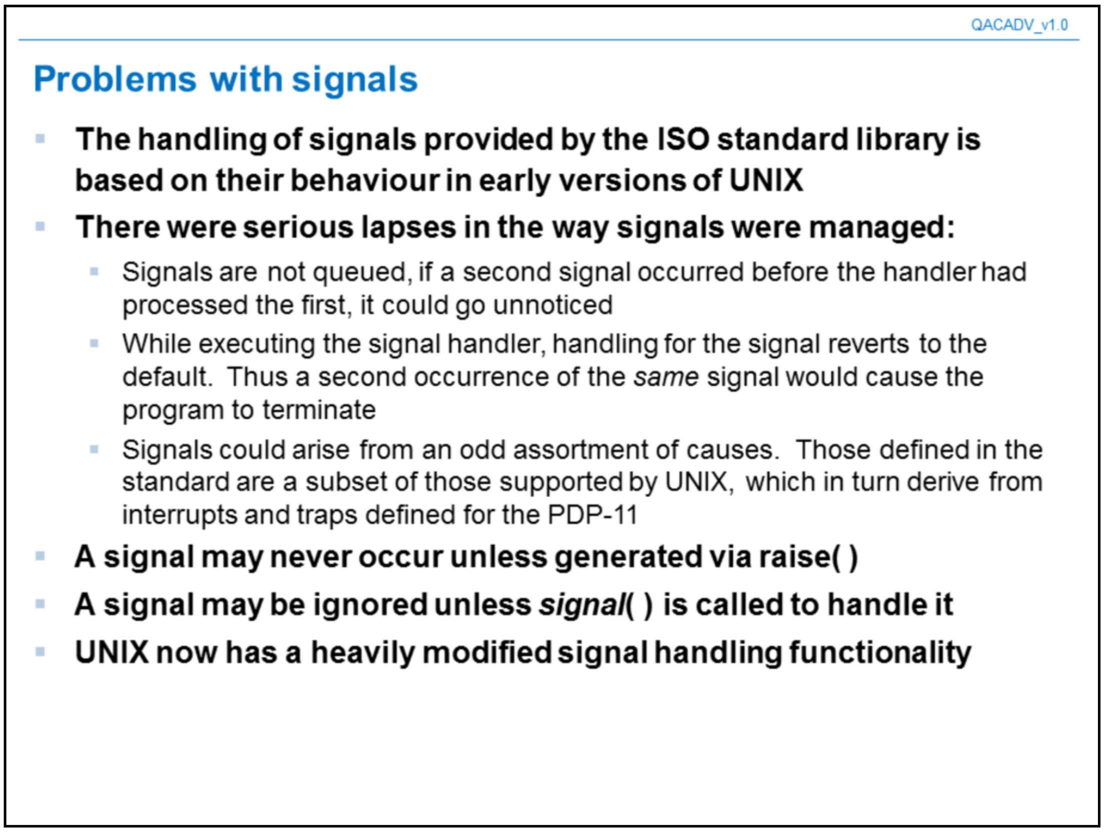
- **A signal is an extraordinary event that occurs during the execution of a program**
  - Division by zero
  - Accessing storage improperly
  - User generating an interrupt via the keyboard (Ctrl-C, Ctrl-Z)
- **Two functions are available:**
  - *raise*() generates a signal
  - *signal*() allows the handling of a signal to be specified
- **Signals may be handled in a number of ways:**
  - *default handling* causes program termination (mostly)
  - *ignoring* the signal effectively discards it
  - *handling* the signal causes control to pass to the designated function
- **In the last case, the designated function is called a signal handler**

Signals are generated by the operating system when something extraordinary happens to a process. Dividing a number by zero may not seem that extraordinary but it is something you don't want to do that often. If a signal is generated for which special handling has not been specified, the process receiving the signal will be killed.

If a process does not take account of signals, then the default handling will occur. As described above, this causes the death of the process. A process may choose to ignore a signal or group of signals, in this case the operating system effectively throws the signals away.

If handling the signal is desired, the process must nominate a function which will take control whenever a signal is received. This special function is called the signal handler. Normal execution of the process is suspended while the signal handler executes. If the function does not terminate the process (by calling exit for example) then control is returned to the point at which the signal was raised. If a signal handler does return then, aside from the extra time taken to execute it, the program continues as before.

If this all sounds very elegant, it is not. Many issues regarding synchronisation are raised. Further problems occur within the standard C library itself. Say, for example, printf is in the middle of printing a string when a signal occurs. Control is immediately passed to the signal handler. Should this handler print a message the output stream could end up in a rather confused state. There is no way from within a signal handler to determine whether or not the standard library is in a safe or an unsafe state.

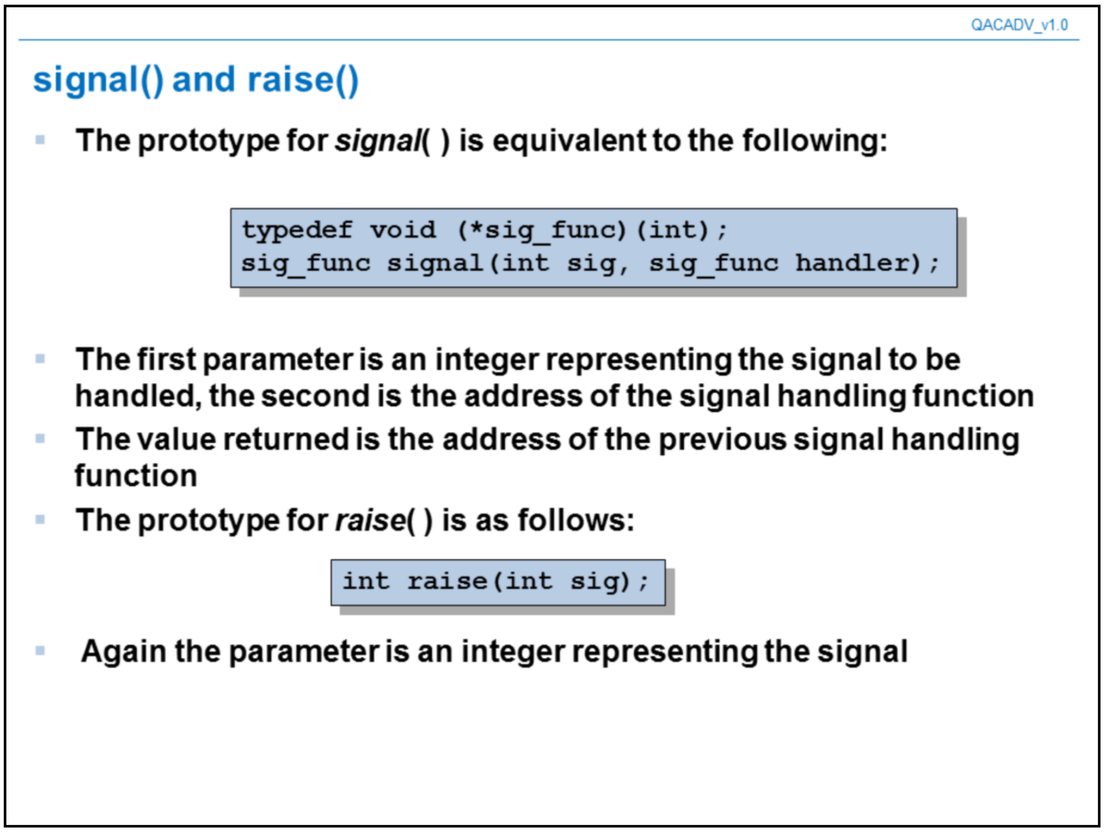

QACADV_v1.0

Problems with signals

- **The handling of signals provided by the ISO standard library is based on their behaviour in early versions of UNIX**
- **There were serious lapses in the way signals were managed:**
  - Signals are not queued, if a second signal occurred before the handler had processed the first, it could go unnoticed
  - While executing the signal handler, handling for the signal reverts to the default. Thus a second occurrence of the *same* signal would cause the program to terminate
  - Signals could arise from an odd assortment of causes. Those defined in the standard are a subset of those supported by UNIX, which in turn derive from interrupts and traps defined for the PDP-11
- **A signal may never occur unless generated via raise( )**
- **A signal may be ignored unless *signal*( ) is called to handle it**
- **UNIX now has a heavily modified signal handling functionality**

There are a number of problems with the signals in ISO C. They are derived from a very old architecture. They are not well serviced by Microsoft Windows (there are plenty of alternatives) and should no longer be used on Unix.

On System V Unix there are now signal masks, and a signal handler is set-up using the `sigaction()` function. The programmer should be careful what is used inside a signal handler, since almost anything from the C run-time library is ***not*** signal safe. For example, when a call to `malloc()` is interrupted by a signal, the heap is in an indeterminate state. Should the signal handler also use `malloc()` then there is a high possibility that the heap chaining will be corrupt on return. Likewise, buffered stdio functions like fopen, printf, fread, etc., must *not* be used, use the kernel calls `open()`, `write()`, `read()`, which return EINTR when interrupted so they be retried.



QACADV_v1.0

signal() and raise()

- The prototype for *signal*( ) is equivalent to the following:

```
typedef void (*sig_func)(int);
sig_func signal(int sig, sig_func handler);
```

- The first parameter is an integer representing the signal to be handled, the second is the address of the signal handling function
- The value returned is the address of the previous signal handling function
- The prototype for *raise*( ) is as follows:

```
int raise(int sig);
```

- Again the parameter is an integer representing the signal

The signal() function allows the programmer to install a handler function that will be called automatically when a certain signal occurs. The second parameter to signal() is the address of the new handler function. The return value from signal() is the address of the previous handler function, which should be stored somewhere so that it can be reinstated later if desired.

There are six signal types defined in the C standard:

SIGINT	-	keyboard interrupt (such as CTRL-C)
SIGABRT	-	abnormal termination
SIGFPE	-	an erroneous arithmetic operation, such as divide by zero or an operation resulting in an overflow
SIGILL	-	illegal instruction
SIGSEGV	-	an invalid access to storage
SIGTERM	-	termination request sent to the program

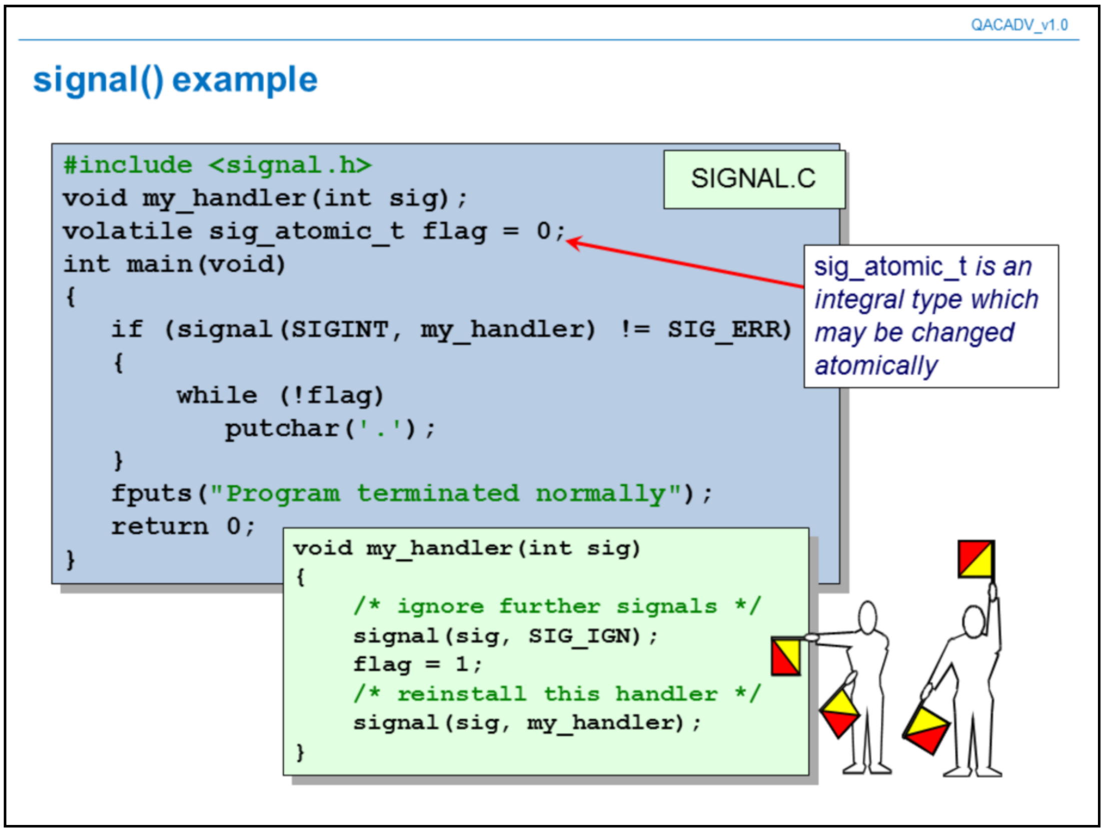
An implementation does not have to generate any of these signals, except via explicit calls to raise. For example, Windows does not generate SIGINT, ILL, SEGV, or TERM, but they can be used with raise() (in the same thread only). Additional signals may be declared, they will all be defined as SIG????.

The following macros are #defined to be compatible with the second parameter to signal:

SIG_DFL	-	take the default action for a particular signal
SIG_IGN	-	ignore the signal

One more value is defined indicating the occurrence of an error:

SIG_ERR	-	a value indicating an error



QACADV_v1.0

signal() example

SIGNAL.C

```
#include <signal.h>
void my_handler(int sig);
volatile sig_atomic_t flag = 0;
int main(void)
{
   if (signal(SIGINT, my_handler) != SIG_ERR)
   {
       while (!flag)
          putchar('.');
   }
   fputs("Program terminated normally");
   return 0;
}
```

sig_atomic_t is an integral type which may be changed atomically

```
void my_handler(int sig)
{
    /* ignore further signals */
    signal(sig, SIG_IGN);
    flag = 1;
    /* reinstall this handler */
    signal(sig, my_handler);
}
```

The example given above shows how the `signal()` function can be used to install a new CTRL-C handler which intercepts the `SIGINT` signal. The following points should be considered:

- For maximum portability, an asynchronous signal handler should only make calls to the function `signal()`, assign values to a data object of type `volatile sig_atomic_t`, and then return. The type `sig_atomic_t` is an standard typedef (often for `int`) for communication between signal handlers and the rest of the program. It is an "atomic" type in the sense that you can modify its value in a single transaction, without worrying about another signal occurring in the middle of the assignment.

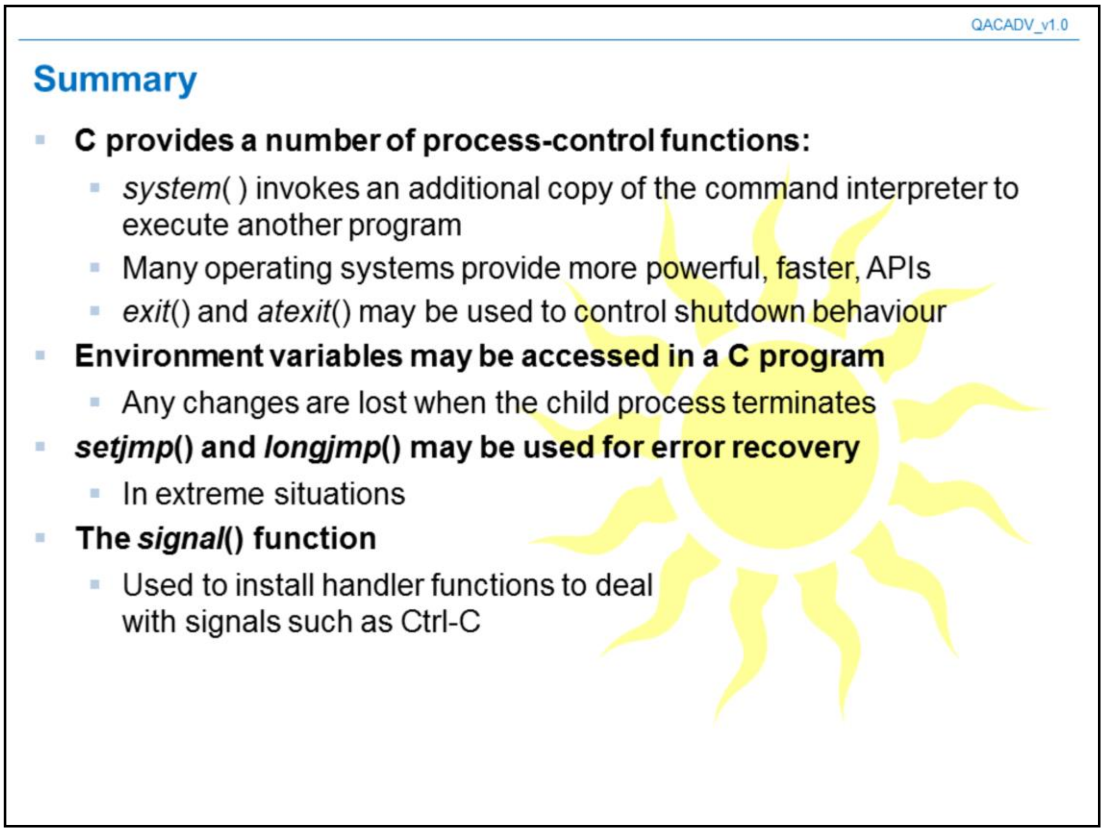
- On entry to a signal handler, the `SIGINT` interrupt handler should be disabled to prevent the signal handler itself from being interrupted by CTRL-C.

- The original signal handler should be reinstated on exit from the signal handler, in readiness for any subsequent CTRL-Cs that might occur.

On Windows a CTRL-C generates a SIGINT by default, but the application can override this with `SetConsoleMode()`. On NT/2000/XP use the `SetCtrlHandler()` API instead of signal handling for CTRL-C.

On Unix the user can alter the key sequence required to generate a SIGINT using:

```
stty intr key-sequence
```



QACADV_v1.0

Summary

- **C provides a number of process-control functions:**
  - *system*( ) invokes an additional copy of the command interpreter to execute another program
  - Many operating systems provide more powerful, faster, APIs
  - *exit*() and *atexit*() may be used to control shutdown behaviour
- **Environment variables may be accessed in a C program**
  - Any changes are lost when the child process terminates
- ***setjmp*() and *longjmp*() may be used for error recovery**
  - In extreme situations
- **The *signal*() function**
  - Used to install handler functions to deal with signals such as Ctrl-C

In this chapter, we have looked at a number of functions and techniques for starting child processes, terminating them via `exit()` or `abort()`, and communicating information using environment variables.

We have also discussed inter-segment jumps and shown how they may be used to achieve an application-wide strategy for error recovery. Finally, we looked at signal handling via the standard function, `signal()`.

You should also consult your own compiler documentation to see if there are any non-standard functions available. Frequently, these functions are more powerful and useful than the standard functions, although portability issues obviously need to be taken into account.