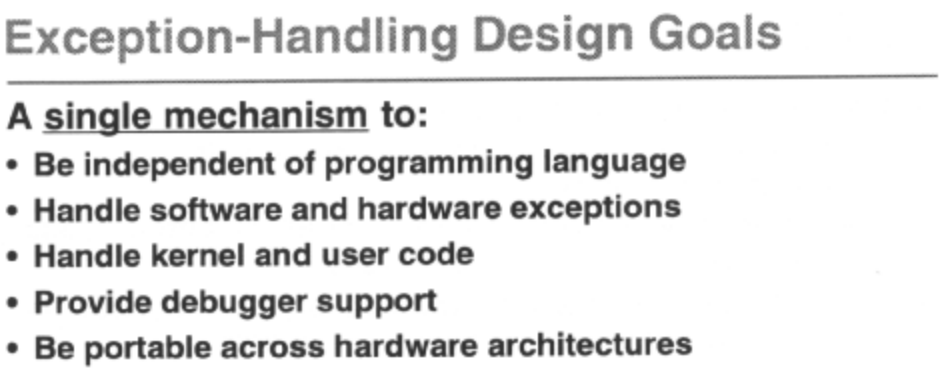


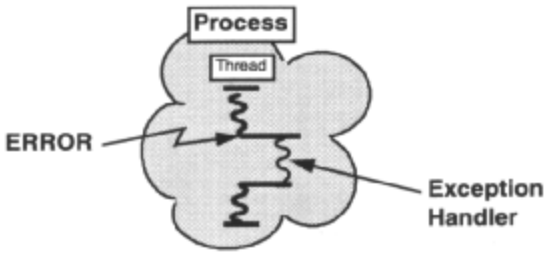
The design goals of Win32 exception handling are to provide a single mechanism for the handling or exceptions that:

- Is usable across all languages.
- Allows the handling of hardware-generated, as well as software-generated, exceptions.
- Can be used by privileged and non-privileged software.
- Gives necessary support to sophisticated debuggers.

Is portable, separating machine-dependent from machine-independent information.

Exception Handlers

- **A function to specifically handle exceptions**
  - **Frame based and associated with a certain scope of code**
  - **Language-specific syntax**
- **System provides default '*handler*' for all exception types**
  - **In most cases this terminates the current process**
- **Application needing to attempt recovery from a particular exception can provide an exception handler**



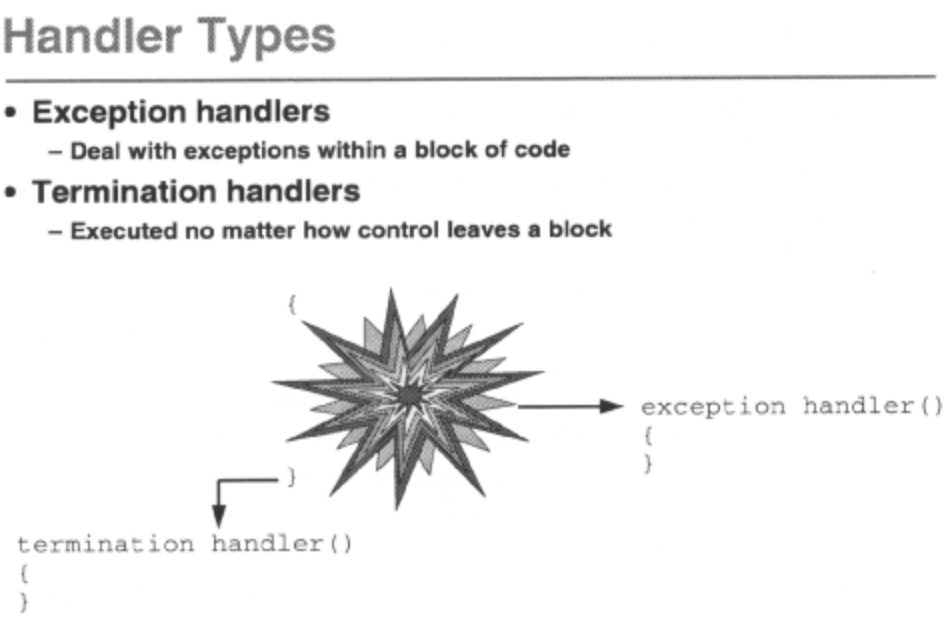
When an exception is raised, the operating system performs a systematic search in order to find an appropriate exception *'handler'*. An exception handler is a function written to explicitly deal with the possibility that an exception may occur in a certain sequence of code.

Exception handlers are declared in a language-specific syntax, and can be associated with a specific block of code (i.e. a set of curly braces). This is referred to as a *'guarded block'*. Exception handlers are *frame-based'*, which means they are associated with the current *'stack frame'* or *'call frame'*. A stack frame is an area on the stack that encompasses all the data needed by a particular block of code; a block of code may be a function. Stack frames are linked together to enable a 'call and return' mechanism to work. In a typical piece of code, where many function calls and blocks of code are nested, the stack will contain many stack frames.

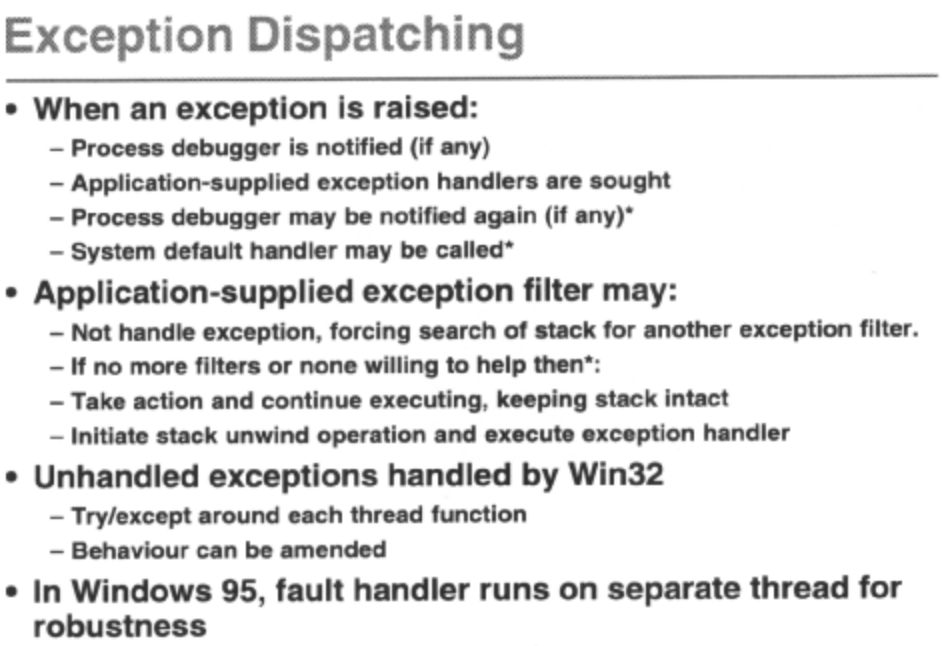
Because an exception handler can be associated with each *'call frame'* in a program, the Win32 API defines a standard calling convention for the construction of a call frame and its associated exception handler. All language compilers for Win32 must follow this standard.

The operating system and Win32 provide default kernel-mode and user-mode exception handlers respectively, which in most cases simply call `ExitProcess()` to terminate the process that caused the exception. Language compilers may also provide exception handlers as part of their run-time support.

Sometimes, it is useful for an application to attempt to recover from exceptions, or at least to terminate gracefully. For example, it is perfectly feasible to provide an exception handler that automatically commits uncommitted pages in a sparse object.



A termination handler and an exception handler cannot be associated with the same block of code. 10 achieve this you must use nested blocks.



When an exception occurs, the operating system saves the state of the current thread in a *'context record'*. It then determines why the exception occurred and constructs an *'exception record'* that describes it. The executive then *'dispatches'* the exception; the result of this depends on the processor mode.

If the processor was in kernel mode, the kernel stack call frames are searched, looking for a handler. If no handler is found or none handle the exception, then this is considered fatal and the system is shut down. The operating system provides exception handlers for all kernel-mode exceptions, so this should never happen in practice.

If the exception occurred when the processor was in user mode, then:

- An attempt is made to notify the debugger of the process in which the exception occurred. The debugger may handle the exception (e.g., breakpoint or single step) and modify the thread state as appropriate. If the process is not being debugged, or if the associated debugger does not handle the exception, then:
- The current threads call frames are searched, looking for an exception handler. If no handler can be found, or none of them deal with the exception, then:
- The debugger is given another chance. If the exception remains unhandled, the system provides default handling based on the exception type.

Win32 puts a try/except block around each thread function in a process. The associated filter picks up the fact that the whole thread stack has been traversed and the exception not dealt with. By default, it passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an Application Error message box and causes an exception handler to be executed, which exits the process.

This behavior can be overridden. See a later slide.

Handling Exceptions: C/C++ Syntax

- **Try/except statement is language specific**

```
_try
{
   try_body
}
_except ( exception_filter )
{
   exception_handler
}
```

- *exception_filter* **evaluated on exception in** try_body
  - **evaluation determines action**
  - EXCEPTION_CONTINUE_SEARCH
    EXCEPTION_CONTINUE_EXECUTION
    EXCEPTION_EXECUTE_HANDLER **passes control to** exception_handler

Applications can provide an exception filter and exception handler for each call frame (block of code). The try { clause defines a call frame around a *'guarded statement'* or *'try_body'*. If an exception occurs during execution of the guarded statement, the *'exception _filter'* in the _except() clause is evaluated.

The exception filter may be an in-line expression or a separate function, and may be as complex as desired. The Win32 documentation says that the exception filter is evaluated in the context of the try body, thus local variables may be accessed. This does not work with the Visual C++ compiler!

The evaluation of the exception filter determines what happens next. The exception filter could decide not to handle the exception by evaluating EXCEPTION_CONTINUE_SEARCH, which causes the operating system to keep searching through the stack call frames, looking for an exception handler.

The exception filter could handle the exception, but continue executing. An example of taking action might be to commit a reserved page on a memory-access violation. If execution can be continued, the filter may change the machine state by modifying the context record (for example, advance the continuation instruction address) and evaluate to EXCEPTION_CONTINUE_EXECUTION, telling the operating system to abandon its search for a handler and to continue from the machine state in the context record. In this case the stack is unaffected.

If execution cannot be continued from the point at which the exception occurred, then the exception filter could evaluate to EXCEPTION_EXECUTE_HANDLER, which causes the operating system to execute the handler associated with this try_body. To execute the exception handler the stack is unwound by traversing back through the stack frames to point on the stack frame that contains the exception handler. Unwinding the stack in this way will cause any termination handlers associated with a stack frame to be called first. After the exception handler is called, execution continues sequentially in the stack frame in which the exception handler was found.

Even though the exception filter is executed in the context of the try body, the stack is not unwound to get there. If it was, then it would be impossible to continue execution if any filter evaluated to EXCEPTION_CONTINUE_EXECUTION.

Handling Termination - C/C++ Syntax

- **Try/finally statement is language specific**

```
_try
{
   try_body
}
_finally
{
   termination_handler
}
```

- `termination_handler` **always entered**
  - **Will be executed even on**
    - `return` **from the function in** `try_body`
    - `goto` **out of the** `try_body`

You may not have both `_except()` and `_finally{}` clauses on the same block. The `_finally { . . . }` clause defines a *'termination handler'* that will be called whenever the try body is exited. There are three possibilities for the exit from a try-body:

1. Normally exit (i.e. sequential execution past the final curly bracket, return, break, continue or *goto* statements).

2. Via an exception handler (the termination handler is called before the exception handler).

3. Via an unwind (the termination handler is called before unwinding).
4.

The termination handler itself can exit in a variety of ways:

- A jump (return, break, etc.). Any exception handling/unwinding that led to the calling of the termination handler is completed.

- Normally. Control continues according to how it reached the handler. This could mean going back to an exception handler (case 2 above), continuing to unwind (case 3) or going back to the instruction to which the try-body passes control (case 1).

The guarded statement, exception filter, exception handler and termination handler all share the same set of automatic (stack) variables, because they execute within the same call frame.

Compilers generate errors if you attempt to explicitly enter an exception or termination handler, e.g. with a go t o statement.

Exception Handling API

- `GetExceptionCode()`
  - **Returns code describing exception type**
  - **Callable from exception filter or exception handler**
- `GetExceptionInformation()`
  - **Returns information describing exception**
  - **Callable only from exception filter**
- `AbnormalTermination()`
  - **Returns whether guarded code raised an exception**
  - **Callable from termination handler**

These three intrinsic functions are frequently used in exception handling code. Note that `GetExceptionInformation()` returns a pointer to an EXCEPTION_POINTERS structure that is only valid during execution of the exception filter. The exception filter must make a copy of any data that is required by the exception handler.

A full list of exception codes can be found in ***winnt.h.***

Termination Handling Example

- **Ensuring resources are freed**

```
char *CreateCopy ( char *psz )
{
  char * p = NULL;

  _try
  {
    EnterCriticalSection( &g_cs );
    p = malloc ( strlen ( psz ));
    strcpy( p, psz );
  }
  _finally
  {
     if ( AbnormalTermination() && NULL != p )
     {
          free ( p );
          p = NULL;
     }

     LeaveCriticalSection( &g_cs );
  }

  return p;
}
```

`strcpy()` will fail if `p` is `NULL`.

`strcpy()` has no exception handler.

The stack is searched to find one.

If an exception handler is found which handles the exception, the unwind causes the termination handler to be executed.

The termination handler will be called whether an exception is raised or not, so this code fragment is guaranteed to leave the critical section.

On exit from the termination handler in this example, control passes to the next statement if no exception is raised, and unwinds to an exception handler in an enclosing call frame if an exception occurred.

The `AbnormalTerrmination()` function is supplied so that a finally block can check whether it has been entered after the try body has raised an exception, or not (TRUE for an abnormal termination).

Exception-Handling Example

- **A safe floating-point division**

```
float SafeDivision( float dividend, float divisor)
{
   _try
   {
     return dividend/divisor;
   }
   _except
   (GetExceptionCode()==
    EXCEPTION_FLT_DIVIDE_BY_ZERO ?
    EXCEPTION_EXECUTE_HANDLER :
    EXCEPTION_CONTINUE_SEARCH)
   {
     return 0;
   }
}
```

This code fragment returns 0 if the divisor is zero. Any exception other than a *'floating point divide by zero'* unwinds through the stack call frames.

Exception Information

- GetExceptionInformation() **returns a pointer to an** EXCEPTION_POINTERS **structure containing:**
  - **A chain of** EXCEPTION_RECORD **structures describing machine-independent information:**
    - **Exception code (a number)**
    - **Exception flags (attributes)**
    - **Pointer to next** EXCEPTION_RECORD **(if any)**
    - **Address where exception occurred**
    - **Additional information (array of 32-bit values)**
  - **A hardware specific** CONTEXT **structure describing the CPU state at time of exception**

An exception filter can retrieve information describing the exception with the GetExceptionlnformation() API. This consists of a chain of EXCEPTION_RECORD structures that identify the exception and provide processor-independent information, together with a CONTEXT structure that provides processor-specific information about the thread of execution at the time of the exception.

The EXCEPTION_RECORD looks like this:

```
typedef struct _EXCEPTION_RECORD
{
      DWORD ExceptionCode;

      DWORD ExceptionFlags;

      struct _EXCEPTION_RECORD *ExceptionRecord;

      PVOID ExceptionAddress;

      DWORD NumberParameters;

      DWORD ExceptionInformation [EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *LPEXCEPTION_RECORD;
```

The contents of `ExceptionInformation` and `ExceptionFlags` members are dependent on the `ExceptionCode` member.

Handling Memory-Access Violations

- **Exception handlers can be used to commit pages in a sparse memory region automatically**
- **When system raises an access violation exception, the additional information in the `EXCEPTION_RECORD` structure contains the virtual address causing the exception**
- **This virtual address can be used in `VirtualAlloc()` to commit the appropriate page**
- **Note: no easy way of automating memory de-commitment!**

Win32 applications cannot change the size of a memory region once it has been reserved; instead, they must allocate a *'sparse'* region of the 'worst case maximum size. To minimize the demands on physical memory and disk swap space, only a minimum number of pages should be committed. By using an exception handler, it is possible to commit reserved pages automatically, as they are accessed.

When the system calls an exception handler with an access-violation exception, it places the virtual address that caused the exception in the ExceptionInformation array of EXCEPTION_RECORD structures.

If the ExceptionCode is EXCEPTION_ACCESS_VIOLATION, the ExceptionInformation contains two elements. The first is a flag signaling whether the thread did a read or a write to the memory, and the second is the virtual address that caused the exception.

Note that before calling `VirtualAlloc()` to commit a page, the handler should call `VirtualQuery()` to confirm that the violation was caused by attempted access to a reserved but uncommitted page.

System functions will probably not generate an exception on access to invalid memory. Instead, they test your parameters and fail the call without raising the exception. You must rely on return values from system calls to indicate the outcome of the function.

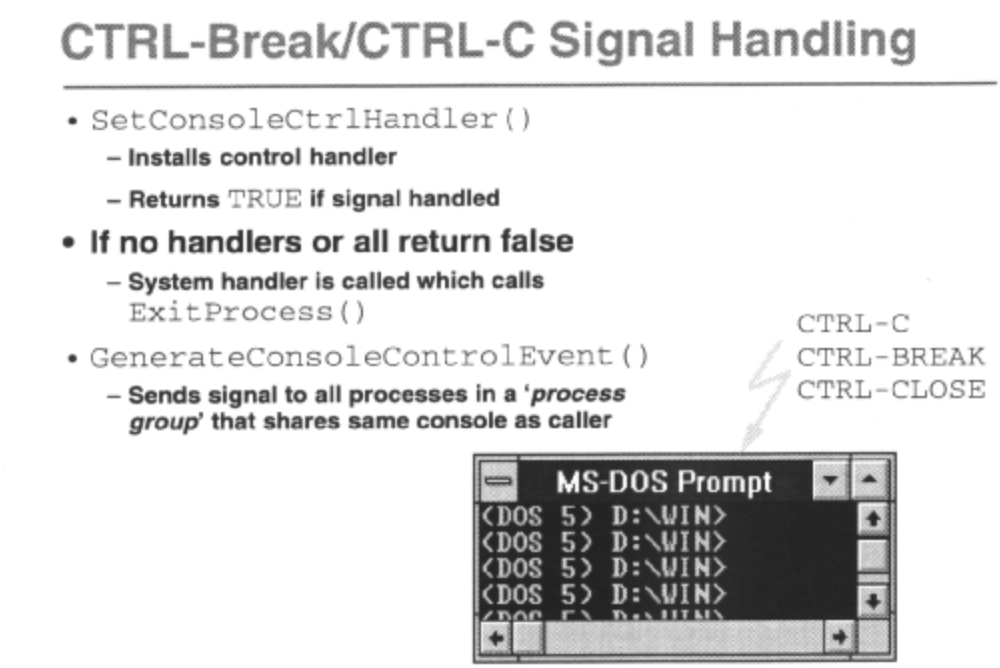
Addresses could be touched first before passing them to the system call when using an exception handler to automatically manage your memory. This way, your exception handler is able to manage your memory before it is passed to system calls.

Note that there is no easy way of automating the decommitment of memory. A common technique is to use a low-priority garbage-collection thread that knows when a page of memory is finished with. This assumes that you have written your own kind of heap-handling code!

Miscellaneous Exception-Handling API

- `RaiseException()`
  - Generates an exception in the current thread
- **No ability to raise exceptions in other threads**
  - i.e. no general purpose signals
- **This is useful for extending the single exception-handling mechanism to encompass program-defined errors**

`RaiseException()` can be used to raise exceptions that are not raised by the system. For example, if an application-defined function gets an out of range parameter passed to it, this could be handled by calling `RaiseException()` It effectively invokes the thread's own exception handler. Win32 provides no mechanism for raising an exception in another thread.



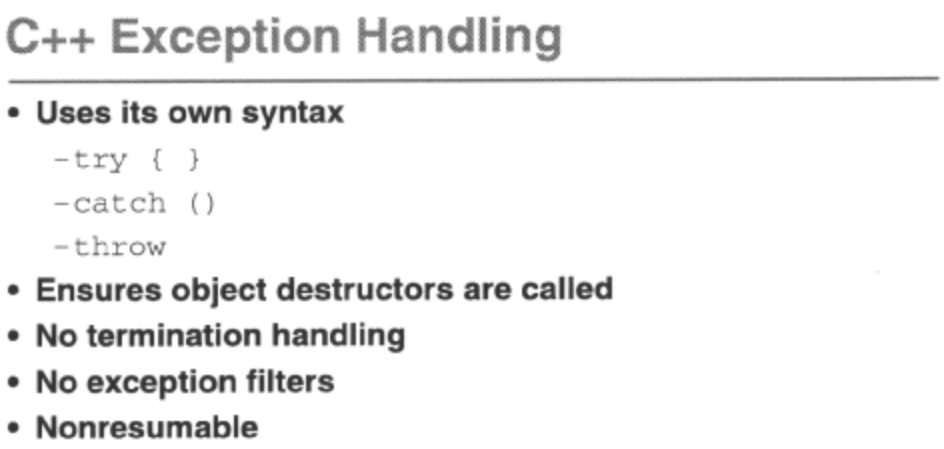
Console processes can install a *'Control Handler'* to deal with CTRL-BREAK, CTRL-C and CTRLCLOSE signals. Each console process has its own list of handler functions that handle these signals. Initially, this list contains only a default handler function that calls `ExitProcess()`. A console process adds or removes additional handler functions by calling `SetConsoleCtrlHandler()` When a console process receives any of the control signals, its handler functions are called on a last registered, first called basis until one of the handlers returns TRUE. If none of the handlers returns TRUE, the default handler is called.

If a process is created with the CREATE_NEW_PROCESS_GROUP flag, that process and any descendants of that process are said to belong to the same process group. All processes attached to the same console as the root process receive any signals.

It is possible for a process within this group to generate a CTRL-BREAK or CTRL-C signal by calling `GenerateConsoleControlEvent()`

The CTRL-CLOSE signal is generated when the user closes a console. This gives a process an opportunity to clean up before termination. When a process receives this signal, the handler function can do one of the following, after performing any cleanup operations:

- Call `ExitProcess()` to terminate the process.
- Return FALSE. If none of the registered handler functions returns TRUE, the default handler terminates the process.
- Return TRUE. In this case, no other handler functions are called, and a pop-up dialog asks the user whether to terminate the process.



Visual C++ provides support for C++ exceptions, as defined in the ANSI draft standard. Three keywords are provided; try, catch and throw.

try is used to begin a guarded block similar to _cry in Win32 exceptions. An exception of any type can be 'thrown' by using the throw keyword. A thrown exception is then 'caught' by a catch compound statement.

The main advantage of C++ exception handling over Win32 exception handling is that C++ object destructors are called if an exception occurs. This just won't happen with Win32 SEH. Indeed, if local stack-based C++ objects are declared in a function, the Visual C++ compiler will give the following error if *Win32 SEH* is used:

"Cannot use try in functions that require object unwinding"

This error will always be generated if you have declared any objects with destructors inside a function with try. .. `_finally` or `try... _except` clauses.

There is no equivalent to `try... finally` in C++ exception handling. Object destructors do get called on a stack unwind, so some cleanup can be done there. If you need try. finally semantics you may need to create a class on entry to the try block, and put the cleanup code in its destructor.

```
try {
      CMyCleanUpClass cinup;
//    Do some work
//    CMyCleanUpClass destructor called however the try block is left

catch ( ...
//    Some exception handler
```

There is no exception filter in C++ exception handling, and also no opportunity to continue program execution. Once control has left the try block, there is no way of resuming execution inside the try block.

Unhandled Exceptions

- **Win32 provides default "unhandled exception" handling for each thread in a process**
  - **Normally this allows process to be debugged or**
  - **Causes process to exit**
- **Can use Win32** `SetUnhandledExceptionFilter()` **to override unhandled exception handling**

- **C++ calls run-time** `terminate()` **function**
- **Can redefine** `terminate()` **using** `set_terminate()`

Win32 puts a *try/except* block around each thread function in a process and calls the Win32 API entry point `UnhandledExceptionFilter()` from the exception filter. This "unhandled exception filter function", installed on a per-process basis for each thread, picks up the fact that the whole thread stack has been traversed and the exception not dealt with. Of course, filter functions can dictate whether an exception handler is called or not and can provide limited exception recovery.

By default, `UnhandledExceptionFilter()` passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an Application Error message box and causes the Win32 exception handler to be executed, which exits the process.

An application can amend the Win32 top-level exception handling by calling `SetUnhandledExceptionFilter()` and supplying the address of a new filter function. Subsequently, if `UnhandledExceptionFilter()` is reached and the application is not being debugged then the application-defined unhandled exception filter is called instead.

UnhandledExceptionFilter() can be called from within (and only from within) the filter expression of any try/except block.

In the C++ world, if a catch statement cannot be found to handle a thrown exception, then the default behavior is to call the `terminate()` run-time routine, which in its standard incarnation calls `abort()`
You can supply your own termination routine by passing its address to the `set_terminate()` function.
Your termination routine must not return, i.e. it must call `exit()`, or end the thread or process with some other function.
If it does return to its caller then `abort()` is called.

Here is an example:

```
void MyTerminate(){
      cout<<"MyTerminate called by runtime”<< endl;
      exit( -l );
}

int main() {
try {
      set_terminate ( MyTerminate );      // Set termination function
      throw “Oh no, an exception”;        // char *exception...
}
catch (int i) { // But we only have a catch for integers...
      cout << "Integer exception caught: “ << i << endl;
}
return 0;
}
```

Summary

- An exception is an unexpected error that threatens continued execution of a thread
- An application may provide exception and termination handlers to attempt to recover from synchronous exceptions
- Win32 provides limited support for raising and handling asynchronous alerts
- C++ provides exception handling and object cleanup