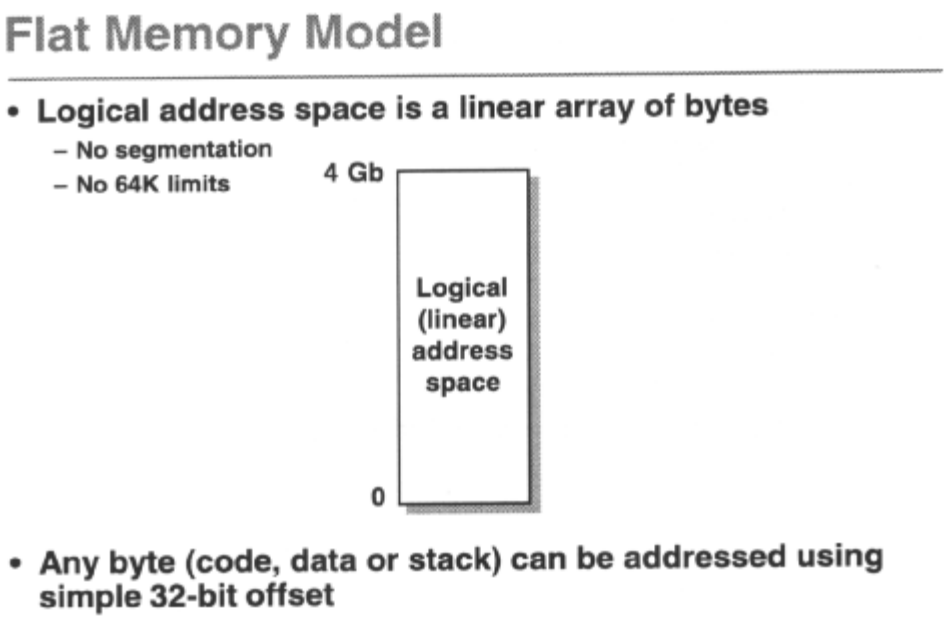
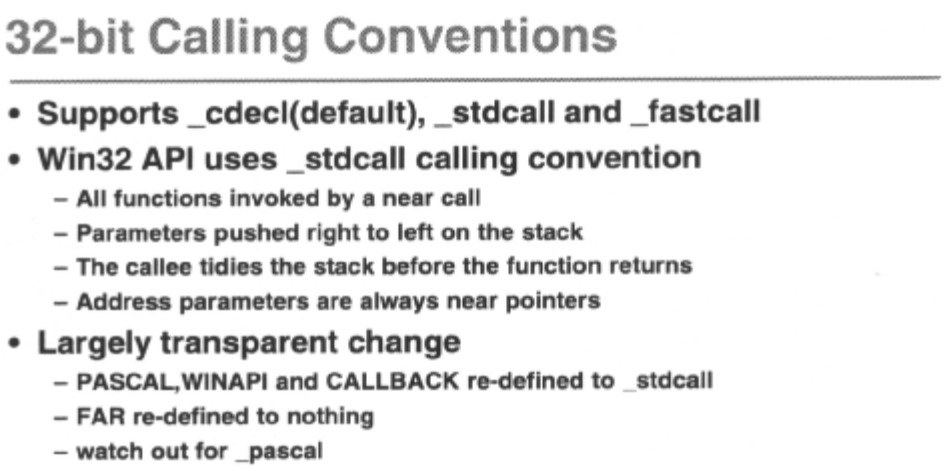


The 'flat *memory model'* used by Windows NT provides a 4Gb logical address space which may be conceived as a linear array of bytes. There is no segmentation, and data objects are not limited to 64Kb as they are under Windows 3.1, without huge address compiler support.

There are no far calls; any bytes in the logical address space, whether code, data, or stack, can be addressed using a simple 32-bit offset.



This discussion is currently limited to the Intel-based compiler. All arguments are widened to 32-bits when passed to functions. Return values are 32-bits and are returned in EAX, except 8 byte structures returned in EAX: EDX. The 32-bit C compiler provides several ways to call functions, but the pascal _fortran and _syscall conventions are obsolete. Supported calling conventions for functions are defined by the keywords:

_cdecl Default C calling convention. Caller cleans up stack, so vararg functions are supported. Arguments pushed on stack right to left. _cdecl 1 functions prefixed by
when decorated.

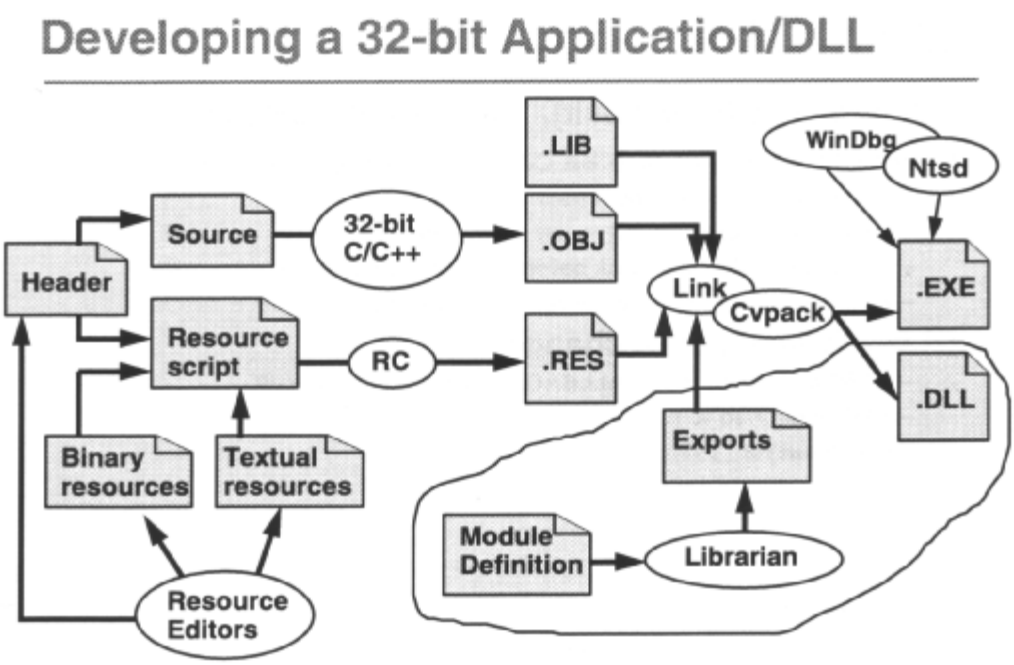
_stdcall Stacked cleaned up by called function, so compiler makes vararg functions cdec 1, and a function prototype is required. Arguments pushed on stack right to left and function is prefixed with _and postfixed with @x when decorated, where x is the number of bytes to be pushed on the stack. Faster calling and smaller code than for _cdecl, if number of function calls > number of functions.

_fastcall Similar to _stdcall but speeds up call by storing at most two arguments in EAX
and EDX. All other arguments pushed as for _stdcall. Functions are prefixed with @ ***and*** postfixed with @x when decorated, where x is the number of bytes to be pushed on the stack..

These are also controlled for entire source files by the /Gd, /Gz, and /Gr options respectively.

The strict calling convention for Windows functions generally is WINAPI, and CALLBACK for user-defined callbacks. Both were defined to be FAR (far) PASCAL (pascal) in the Windows 3.x API, passing parameters in left to right order, and tidying the stack by removing their arguments before passing control back to the calling environment. They are invoked by a far call, and pointers passed as arguments are far pointers.

In contrast, Win32 API functions and user-defined callbacks use the _stdcall calling convention as PASCAL,WINAPI, CALLBACK etc are defined to be _stdcall. Functions are invoked by a near call, and addresses passed as arguments are always near pointers. FAR has been defined to nothing. The /Zf option on the compiler ignores the _far keyword, which aids I 6->32-bit portability. The compiler still allows the .._pascal keyword for pascal calling convention, so watch out if you've defined any callbacks using pascal explicitly. Pascal calling convention is callee cleans up stack, arguments pushed left to right.

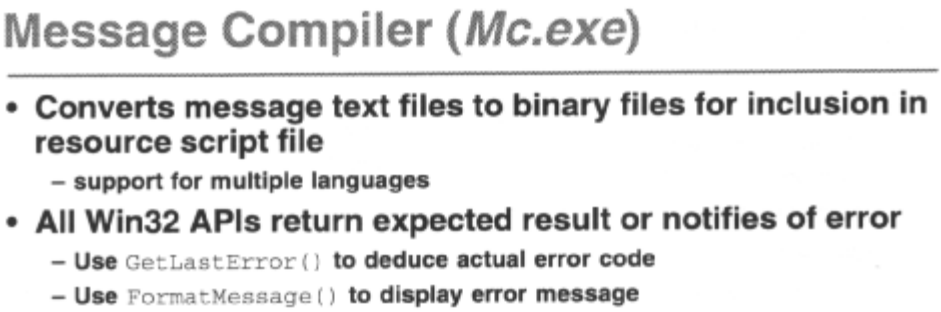


The development cycle for 32-bit graphical or console applications follows an established pattern; source code written in assembler or a high level language is compiled or assembled to give object code, which is linked with libraries and run-time support modules to produce a final executable file. 32-bit Graphical Windows applications may also need to compile resource scripts into binary form and link them into the executable.

Visual C++ 2.0 provides a complete Windows NT based 32-bit development environment; source and resource editors, compiler, linker, librarian, debugger and resource compiler.

In order to make use of Win32' s advanced features, however, the source code must invoke the 32-bit Windows API functions, and should #include the *windows.h* file, which will in turn #include the appropriate 32-bit header files. The source code can also use standard C library functions.

For those familiar with the process of building applications and DLLs for Windows 3.x, there are some differences. Resources are linked along with object modules and libraries. You do not run the resource compiler to add resources to the executable file. Use CVTRES to convert a resource file to a linkable object file.



The Message Compiler, *mc.exe,* converts ASCII message text (.mc) source tiles into binary tiles containing a message table resource for inclusion in a resource script (.rc) file. The source-format supports multiple versions of the same message text, one for each national language supported, thus allowing support for multiple languages within the same image file.. *Mc.exe* automatically assigns numbers to each message, and generates a C/C++ include file for use by the application to access a message using a symbolic constant. For each Language statement, it generates a binary file containing a message table resource. It also generates a single resource script file that contains the appropriate Resource Compiler statements to include each binary output file as a resource with the appropriate symbolic name and language type.

Applications that perform event logging typically use an independent resource-only DLL that contains the messages, rather than carry the messages in the application image. This DLL is then specified as the source of message text for the events that it logs, when using the event logging APIs or FormatMessage() to retrieve and use the message text.

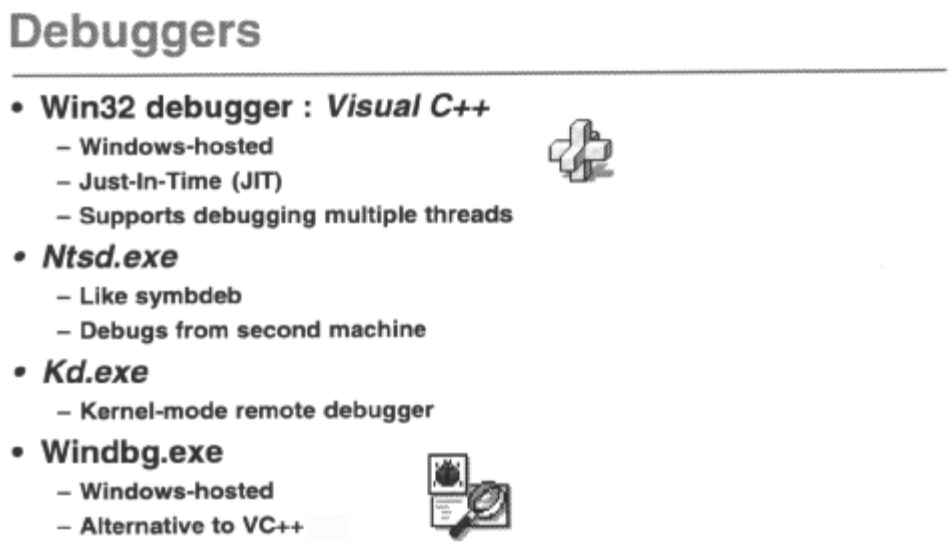
The Win32 API is complex and it is essential that programmers routinely check the return value for each function invoked, and check error codes. Win32 functions return an expected result or tell you if an error has occurred; they don't always indicate what the error was, e.g. CreateWindow() returns an HWND or NULL. However, the Win32 APIs do record errors on a per-thread basis, which can be obtained with `GetLastError(),FormatMessage()` allows the conversion of error codes, including those returned by `GetLastError()` into error strings. E.g.

```
LPVOID lpMessageBuffer;

FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
NULL, GetLastError(), LANG_USER_DEFAULT, (LPSTR)&lpMessageBuffer, 0, NULL);

/* display this string */

LocalFree( (HLOCAL) lpMessageBuffer )
```



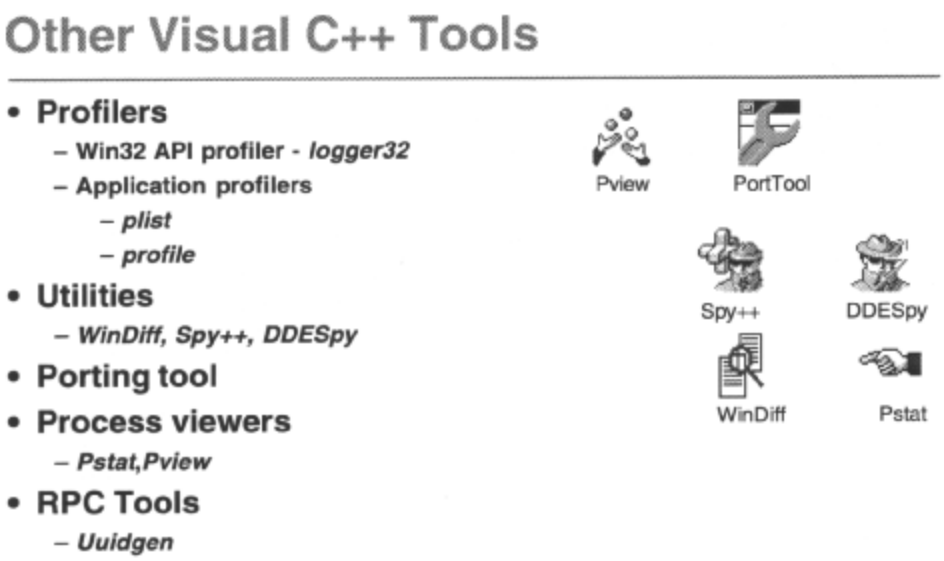
Visual C++ 6.0 has a built in debugger which is very good! It supports Just In Time debugging which allows you to build your app for debug, run it and when it crashes VC++ will catch the error and jump to the source line that caused the problem.

Ntsd is a good deal less user friendly and allows debugging from a second machine. It is *symdeb-like* and accepts text-based commands to step through the debugging process. It supports multi-threaded debugging and also multi-process debugging. Both Ntsd and WinDbg are user mode programs, and allow source-level and machine-level debugging. At a later date, a remote debugging capability will be added to WinDbg, obviating the need for Ntsd.

Kd is a kernel mode remote debugger that runs on a separate NT hosted computer. It provides assembly language debugging of kernel mode code and is intended for system developers and device driver developers. It also supports multi-processor debugging. Ntsd and Kd share much of the same syntax. However Kd is not well suited to debugging user mode code; it does not provide support for threads, cannot set breakpoints in user mode code and cannot be used to examine paged-out memory.

The /DEBUGTYPE COFF flag must be used in the linker options to create debugging information for the *Ntsd.exe* and *Kd.exe* debuggers.

Windbg is another windows-hosted debugger similar to Visual C++.



An automated porting tool, *Porttool.exe,* comes with Visual C++ and assists in porting 16-bit applications to 32-bit applications. It parses source files and uses a table driven search facility to find and highlight non-portable program elements, describe the problem, and where appropriate, offers alternatives.

The *Pview.exe* process viewer allows the examination and modification of processes and threads running on the system. It indicates how much memory is being used, and how much of it is paged. It reports which processes and threads are using most CPU time. It allows the examination of how threads run at different priorities, and reports on thread status. It also reports thread API usage. Because Pview allows the modification of thread and process state, it has the potential for causing serious disruption to the system!

Spy++.exe is similar to that for Windows 3.x, and allows the monitoring of messages, and associated parameters, being transmitted around the system. It also allows monitoring of processes and threads..
Ddespy.exe, is also similar to that for Windows 3.x, and allows the examination of client and server DDE activity.