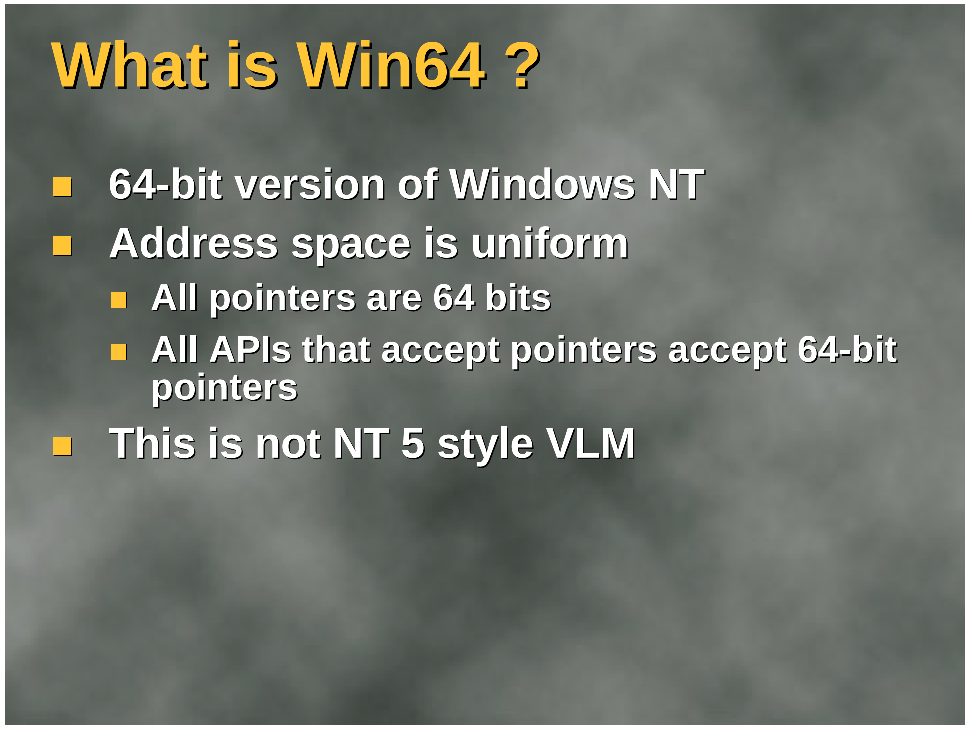
Win64™ Architectural Overview

Oscar Newkerk
Developer Relations Group
Microsoft Corporation

©1999 Microsoft Corporation. All rights reserved.

Introduction

- What is Win64
- Why are we building it
- Brief History Lesson
- Data Models
- API Strategy
- Code Migration
- Interoperability/Legacy Support



What is Win64 ?

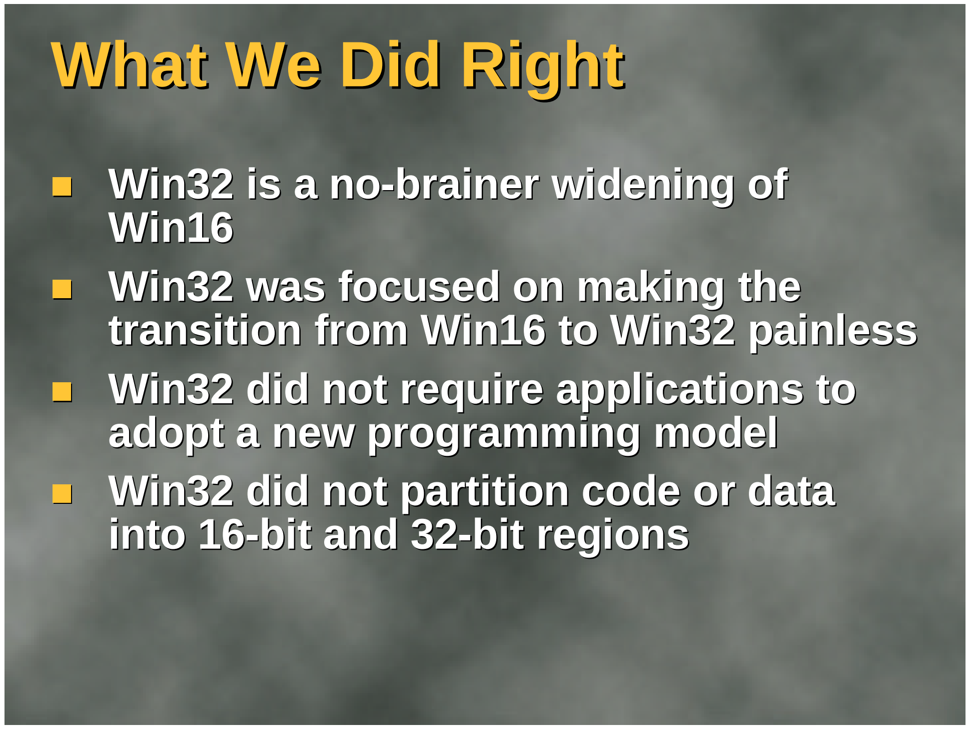
- 64-bit version of Windows NT
- Address space is uniform
  - All pointers are 64 bits
  - All APIs that accept pointers accept 64-bit pointers
- This is not NT 5 style VLM

Purpose and Scope

- Provide ISVs with a uniform, very large (4TB user, 4TB kernel), flat address space
- ISVs choose how to use this large address space, not Microsoft
- Deliver Win64 on ALL 64-bit capable processors that support Windows NT
- Drivers and applications are all Native 64-bit code

History

- Win32 addressed several deficiencies of Win16
  - address space separation
  - flat 32-bit address space
- Design goal was to make porting from Win16 to Win32 as easy as possible
  - API names, parameter meanings, semantics stayed the same between Win16 and Win32
  - No new programming model



What We Did Right

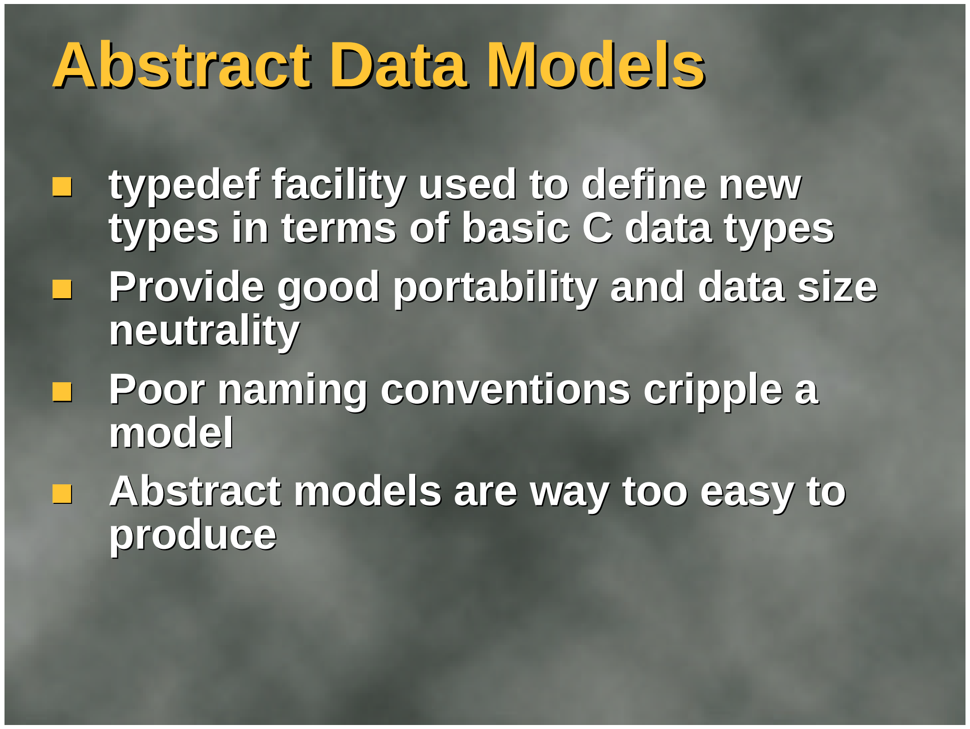
- Win32 is a no-brainer widening of Win16
- Win32 was focused on making the transition from Win16 to Win32 painless
- Win32 did not require applications to adopt a new programming model
- Win32 did not partition code or data into 16-bit and 32-bit regions

Win64 Success Metrics

- **Porting from win32 to win64 should be simple**
- **Supporting win64 and win32 with a single source code base is our goal**
- **No new programming models**
- Require minimal change to existing win32 code data models
- **No-brainer widening of win32 to win64**

Data Models

- Mapping of the basic C types to a specific precision (Win32 is ILP32)
- C has some problems
  - formal precision relationships between basic data types are absent
  - pointers are an arbitrary size and there is no formal relationship between the precision of a pointer and any of the built in types
  - The model becomes the foundation for the abstract data model used to describe the system interfaces

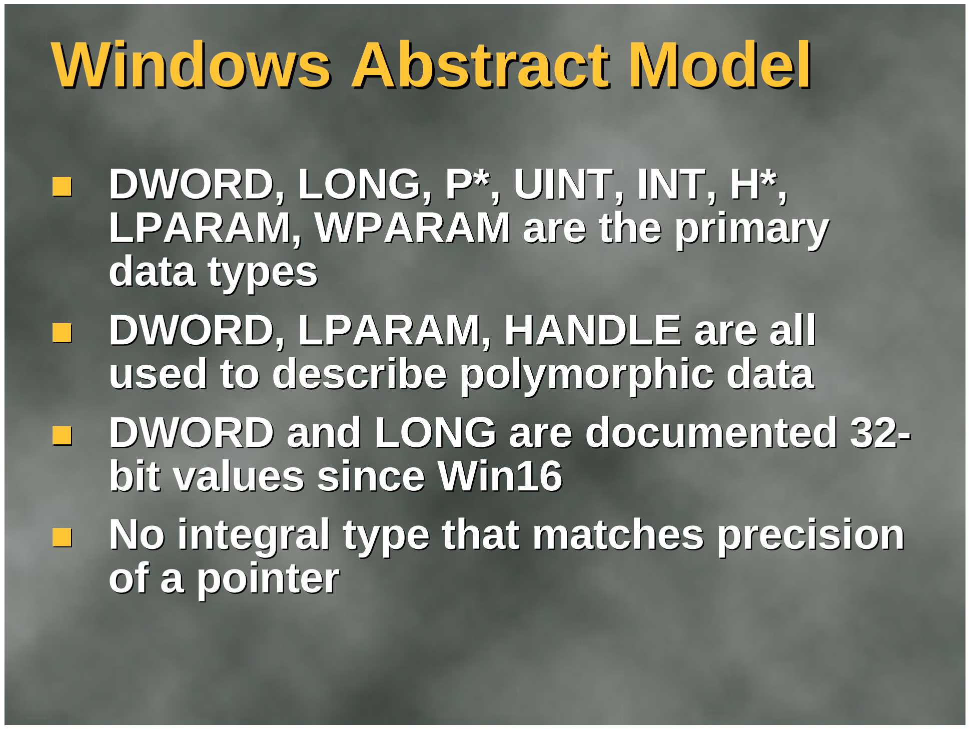


Abstract Data Models

- typedef facility used to define new types in terms of basic C data types
- Provide good portability and data size neutrality
- Poor naming conventions cripple a model
- Abstract models are way too easy to produce

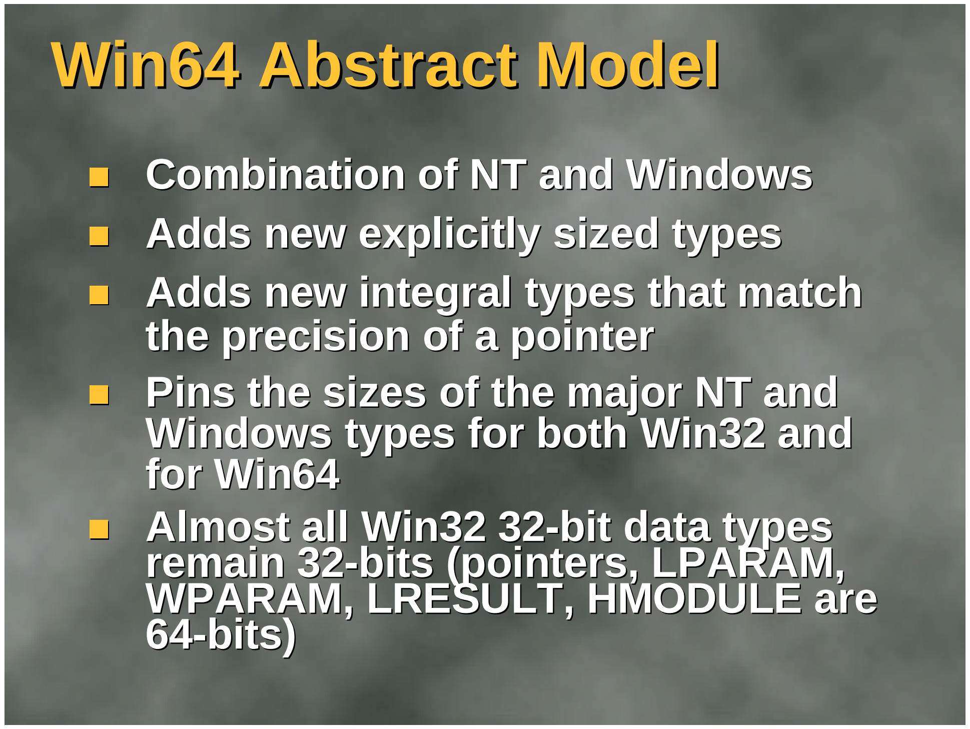
NT Abstract Model

- LONG, ULONG, P*, HANDLE, NTSTATUS are primary data types
- The naming of LONG/ULONG implies the basic C type *long*
- No formal size relationships, so code assumes LONG is 32-bits, SHORT is 16-bits, HANDLE is 32-bits
- No integral type that matches precision of a pointer



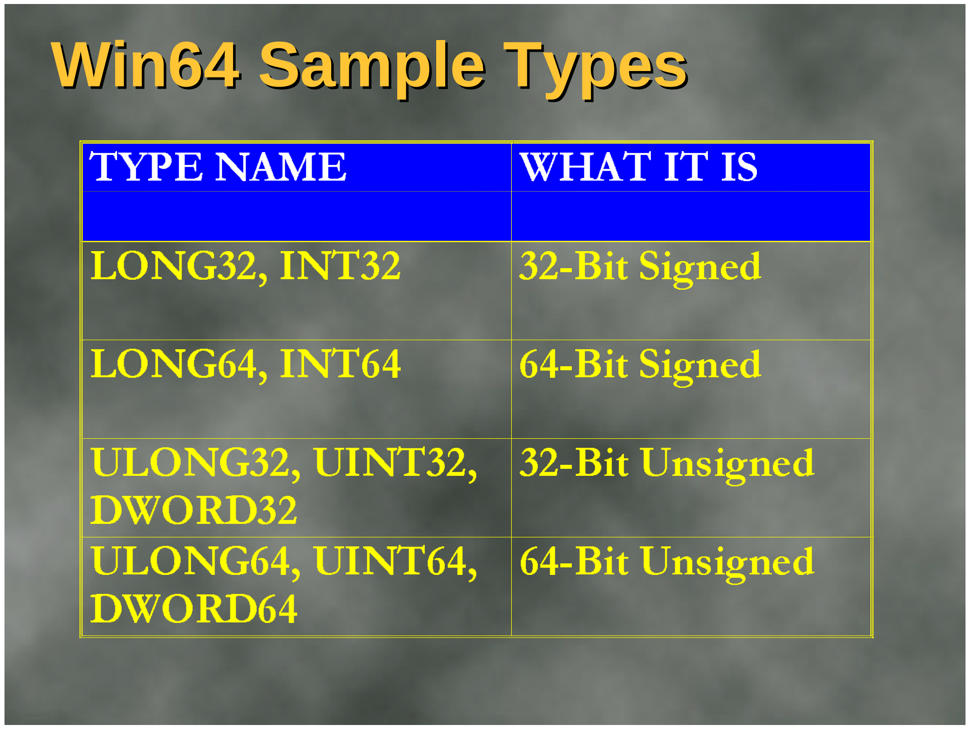
Windows Abstract Model

- DWORD, LONG, P*, UINT, INT, H*, LPARAM, WPARAM are the primary data types
- DWORD, LPARAM, HANDLE are all used to describe polymorphic data
- DWORD and LONG are documented 32-bit values since Win16
- No integral type that matches precision of a pointer



Win64 Abstract Model

- Combination of NT and Windows
- Adds new explicitly sized types
- Adds new integral types that match the precision of a pointer
- Pins the sizes of the major NT and Windows types for both Win32 and for Win64
- Almost all Win32 32-bit data types remain 32-bits (pointers, LPARAM, WPARAM, LRESULT, HMODULE are 64-bits)



Win64 Sample Types

TYPE NAME	WHAT IT IS
LONG32, INT32	32-Bit Signed
LONG64, INT64	64-Bit Signed
ULONG32, UINT32, DWORD32	32-Bit Unsigned
ULONG64, UINT64, DWORD64	64-Bit Unsigned

More Win64 Sample Types

TYPE NAME	WHAT IT IS
INT_PTR, LONG_PTR	Signed Int, Pointer precision
UINT_PTR, ULONG_PTR, DWORD_PTR	Unsigned Int, Pointer precision
SIZE_T	Unsigned count, Pointer precision
SSIZE_T	Signed count, Pointer precision

Win64 Data Model Rules

- If you need an integral pointer type, use UINT_PTR, INT_PTR, ULONG_PTR, or DWORD_PTR. Do not assume that DWORD, LONG or ULONG can hold a pointer
- Use SIZE_T to specify byte counts that span the range of a pointer
- Make no assumptions about the length of a pointer or xxxx_PTR or xSIZE_T. Just assume these are all compatible precision.

LLP64 Issues

- Relationship between *int* and *long* is preserved
- *Long* remains a 32-bit type
- Only data structures that contain pointers change in size
- ISV abstract models remain correct
- *__Int64* must be used for integral 64-bit types
- Win64 is built assuming LLP64

Win64 API Set

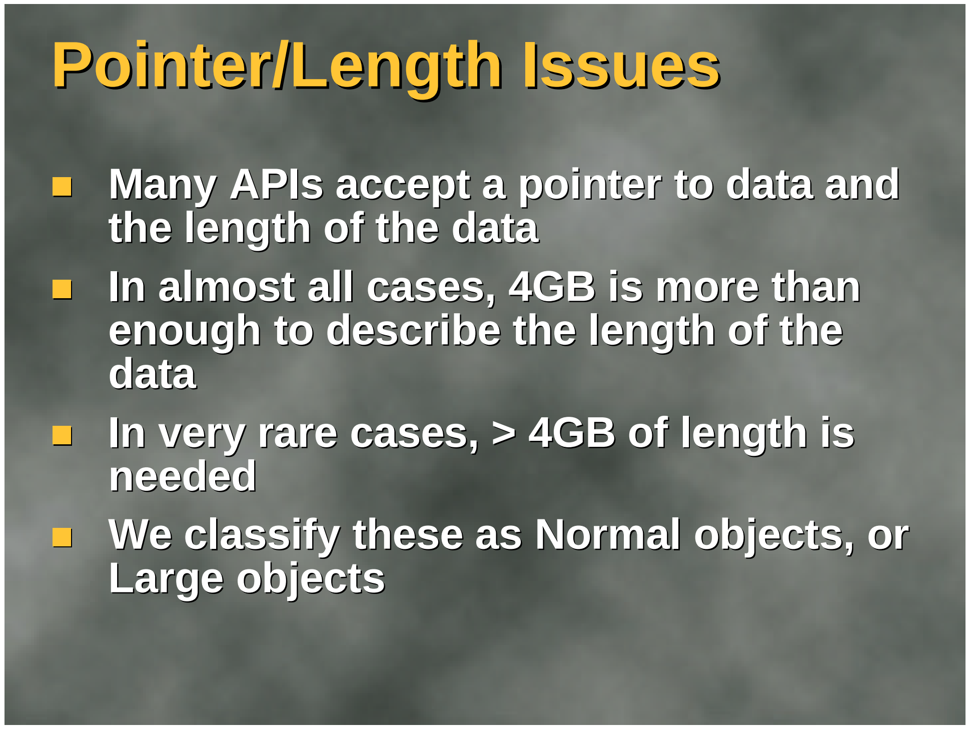
- Simple pointer stretch port of Win32 (and NT Native) API set
- Win64 data type definitions define most of the port
- Porting Issues are Polymorphic Data usage, pointer/length combinations, and miscellaneous cleanup

Polymorphic Data Issues

- Pointing to data with a PVOID and enum is fine
- Passing via DWORD is wrong
  - RaiseException DWORD *lpArguments changes to ULONG_PTR *lpArguments
- DWORD structure members is wrong
  - ULONG ExceptionInformation[] in _EXCEPTION_RECORD changes to ULONG_PTR ExceptionInformation

Polymorphic Data Issues (cont.)

- LPARAM/WPARAM is OK since in Win64, these are LONG_PTR and UINT_PTR.
  - Don't assume LPARAM/WPARAM are LONG or DWORD based
- Window and Class Data
  - New APIs Get/SetWindowLongPtr and Get/SetClassLongPtr are added to extract pointer sized data from your window and class structures. You MUST use FIELD_OFFSET to compute your offsets



Pointer/Length Issues

- Many APIs accept a pointer to data and the length of the data
- In almost all cases, 4GB is more than enough to describe the length of the data
- In very rare cases, > 4GB of length is needed
- We classify these as Normal objects, or Large objects

Migrating 32-bit Code

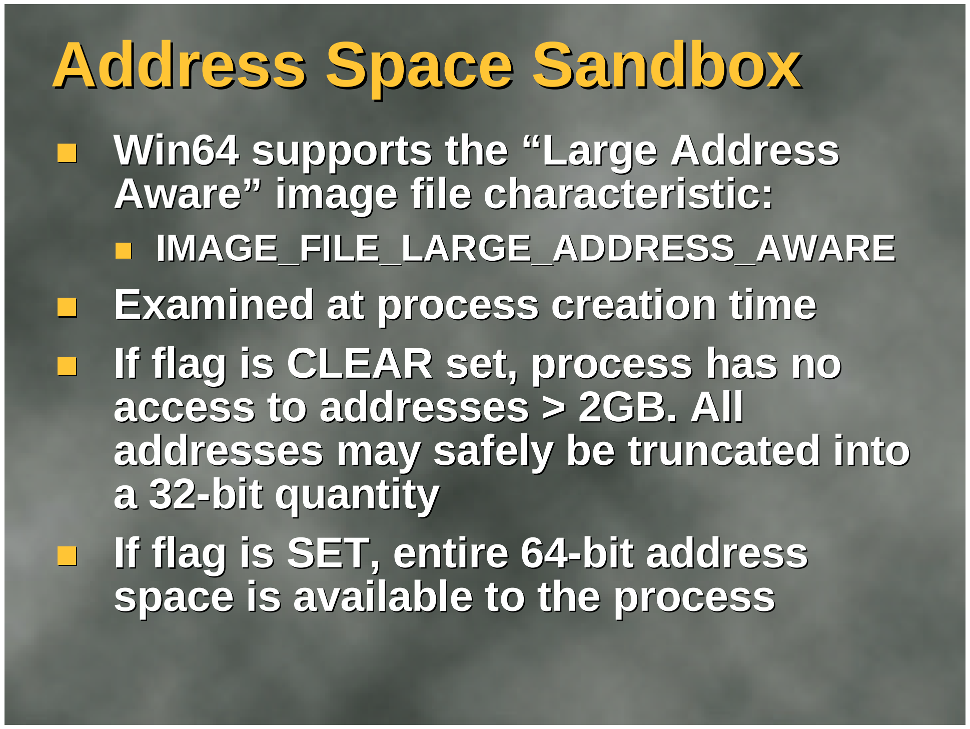
- Recoding pointer arithmetic
  - Change DWORD/ULONG casts to DWORD_PTR or ULONG_PTR
  - This is our biggest work item in NT
- Storing pointers on-disk, on-wire, or in shared memory
- Mixing pointers and offsets in the same storage
  - MakeSelfRelativeSD
  - MakeAbsoluteSD

Migrating 32-bit Code (cont.)

- Adapt to Win64 API Changes
  - Use LPARAM and WPARAM properly
  - Use GetWindowLongPtr and SetWindowLongPtr where appropriate
  - Match our API definitions without using typecasts
- Identify polymorphism in your internal interfaces
- Pay attention to compiler warnings
- Do not blindly cast warnings away

Rapid Migration - 2GB

- You want to be on Win64
- 2GB is plenty of address space
- Pointer truncation warnings are everywhere
- Pointers and int/long are freely mixed
- Polymorphism via 32-bit types is used heavily
- Use our “Address Space Sandbox”



Address Space Sandbox

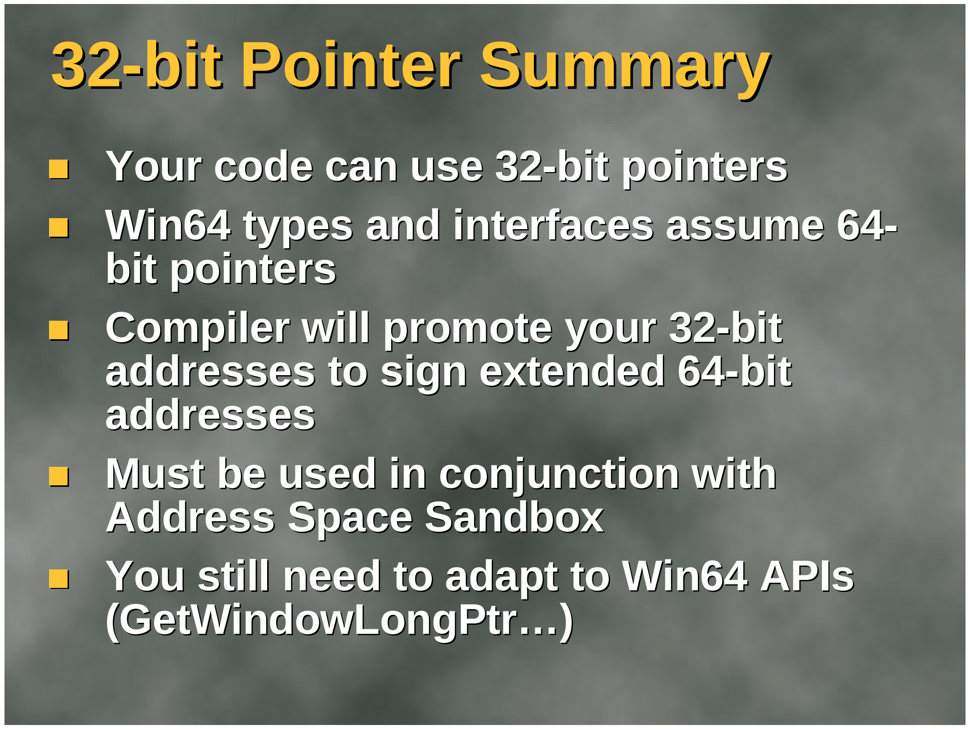
- Win64 supports the “Large Address Aware” image file characteristic:
  - IMAGE_FILE_LARGE_ADDRESS_AWARE
- Examined at process creation time
- If flag is CLEAR set, process has no access to addresses > 2GB. All addresses may safely be truncated into a 32-bit quantity
- If flag is SET, entire 64-bit address space is available to the process

Address Space Sandbox (cont.)

- Pointers are still 64-bits
- The upper 33-bits are 0
- The following code sequence is valid when “Large Address Aware” is cleared:

```
DWORD dw;
PVOID dest, src = malloc(IO_BUFFER);

dw = (DWORD)src;
dest = (PVOID)dw;
ASSERT(((DWORD_PTR)src & 0xffffffff80000000) == 0);
ASSERT(src == dest);
```



32-bit Pointer Summary

- Your code can use 32-bit pointers
- Win64 types and interfaces assume 64-bit pointers
- Compiler will promote your 32-bit addresses to sign extended 64-bit addresses
- Must be used in conjunction with Address Space Sandbox
- You still need to adapt to Win64 APIs (GetWindowLongPtr…)

Address Space SandBox Summary

- You can ignore pointer truncation warnings
- This is by far the biggest work item in doing your port
- Your port to Win64 becomes a recompile and minor API adaptations
- You are limited to 2Gb of address space
- Not for DLL suppliers

Rapid Migration Summary

- If you don't need > 2Gb
  - Use Address Space SandBox
  - Ignore pointer truncation warnings
  - Fast, easy, rapid port
- If you are concerned about memory bloat associated with 64-bit pointer variables
  - Use 32-bit pointer support, but be very very careful… all types in our header files assume 64-bit pointers

Rapid Migration Summary (cont.)

- Win64 OS support amounts to:
  - Flat 64-bit API with Address Space Sandbox
- Compiler/Tool support amounts to:
  - Pragma to control pointer size
  - /Ap*nn* command line switch
  - __ptr64, __ptr32 qualifiers
- Be careful with rapid migration aids. These may cause you legacy problems when you try to move to full 64-bit addressing in the future

IA-64 HAL Features

- One HAL
- ACPI only
- One TLB domain
- No bus lock support
- PNP drivers only
- Alignment fix-ups off by default
- No ISA slots

Driver Issues

- Physical addresses > four gigabytes
  - Use Mm64BitPhysicalAddresses to determine if 64 bit addressing is needed
  - Use Dma64BitAddresses in DEVICE_DESCRIPTION to indicate that 64 bit addressing is supported
- The information field in the IoStatus block is a ULONG_PTR
- The parameters in the IRP stack locations are ULONG_PTR
- No sandbox addressing

RPC and COM

- Supports RPC between IA-32 processes and 64-bit native Win64 processes (same machine or cross machine)
- Supports LocalServer style (out of proc) COM between IA-32 processes and 64-bit native Win64 processes
- Of course IA-32 to IA-32 and native to native RPC and COM is supported

Mixing IA-32 and Win64

- Win64 does not explicitly support loading an IA-32 DLL into the address space of a native Win64 64-bit process
- Win64 does not explicitly support loading a native Win64 64-bit DLL into the address space of an IA-32 process

Call To Action

- Prepare for Win64 now
- Install the NT5 SDK and read readme64.txt
- Remove pointer truncations
- Correct your polymorphism
- Compile warning free
- msdn.microsoft.com/developer/news/feature/Win64/64bitwin.htm

Where do you want to go today?

Microsoft