Introduction

Overview

Migrating your application from 32-bit to 64-bit will impact your data size in a number of ways.

- Most fundamentally every pointer in your application will require an additional 4 bytes of data.
- In addition, when you compile an application with the Intel C++ Compiler for Itanium®-based Applications, structures in your applications will be padded with extra bytes such that structure members are aligned on 64-bit boundaries. This means that the total size of a structure can be larger than the sum of the sizes of the structure members.
- Lastly, when you migrate your application for the Itanium architecture you may choose to change the type of some existing variables from a 32-bit type to an equivalent 64-bit type to prevent future problems. This too will impact application data size.

This lab will allow you to practice the concepts you learn. The migration lab includes using an Itanium-based platform running 64-bit Windows Server 2003* to migrate a 32-bit application.
Pointer Cleaning

Redesigning data structures

Pointers in the Intel® Itanium® architecture are twice the size of pointers in IA-32, which may effectively double the size of any data structure composed largely of pointers. If this results in degraded application performance you can redesign your data structures.

You can replace 64-bit pointers with 32-bit offsets from a base pointer within a contiguous block of memory. Modifying a structure definition to use 32-bit offsets must be accompanied by changes to all code which uses the structure. Instead of a simple C de-reference "->", your code must calculate an address.

Example

Consider the following structure.

```c
struct _NODE
{
    struct DATA data;
    struct NODE * pParent;
};
```

```c
struct NODE *pMyNode;
// Allocate and initialize pMyNode
struct NODE *pMyParent = pMyNode->pParent;
```

The pParent pointer can be replaced with an offset to some base address.

```c
PVOID myBase; // Ptr to contiguously allocated block of memory
struct _NODE
{
    struct DATA data;
    DWORD       dwOffsetParent;
};
```

```c
struct NODE *pMyNode;
// Allocate and initialize pMyNode
struct NODE *pMyParent = myBase + pMyNode->dwOffsetParent;
```

Careful use of macros or C++ overloaded operators can help to preserve the readability of your source code. Applying the __based and __ptr32 attributes to pointers has the same effect, but allows code to use normal pointer syntax.

Alternative - Convert pointers into smaller-sized array indexes

Alternately you can convert pointers into smaller-sized array indexes. Using array indexes will require that your application be changed to use array subscripting "[]". Using alternates to pointers can result in more complex and error-prone code.
Structure Padding

**Problem**

By default, structures in code compiled for the Itanium architecture are aligned on 64-bit boundaries. When members do not align, the structure is “padded” with extra bytes. A large number of padded records can waste a significant amount of memory and degrade performance.

**Solution**

You can fix this problem by reordering fields, placing the longest variable first.

---

**Alternative solution**

You can also use the C++ `#pragma pack` to set a non-default alignment. Architectures that are not Itanium®-based architectures can align on a different boundary. Therefore if your code is compiled to target multiple platforms, ensure that it does not hard-code any offsets or assume a structure size.
Conserving Resources

Use the appropriate data type

To conserve resources, carefully consider how large each integral variable in your application must be and be sure to use the appropriate data type. For integral types you must decide when to use 32-bit types such as INT32 or UINT32 versus 64-bit types such as INT64 or UINT64.

Integral type variable size

Consider the largest value the integral type variable will need to contain.

If the value will always be 0 or positive and will be less than $2^{32}$ use the UINT32 type.

If the value will be in the range of $-2^{15}$-1 and $2^{15}$ use the INT32 type.

If an unsigned value could be as large as $2^{64}$ or a signed value could in the range of $-2^{63}$-1 and $2^{63}$, use a UINT64 or INT64 respectively.

Use data store design

When application data is persisted to a data store you can be guided by type decisions made in the data store design. For example, if you are designing an application that works with Employee records, you can use the same size data type to represent an EmployeeID as is used in the underlying database.

Guidelines when a data store does not exist

When a data store does not exist, use the following guidelines when determining a type to represent a unique identifier.

1. Account for current maximum value.
2. Consider growth in the data values.
3. Factor in the system lifetime.

The following formula can be useful:

$$FV = X(1 + r)^n$$

where FV is the future maximum value, X is the current maximum value, r is the rate of growth per some timespan, and n the lifetime of the system given in the same timespan.

Example of the growth formula

Consider an application in which a unique identifier is generated for each employee. Currently there are 15000 employees and the number of employees grows by 6% each year. Using the growth forumula, you can calculate the maximum EmployeeID needed after 10 years to be 15000 $(1 + .06)^{10} = 26863$. This value is still $< 2^{32}$ so an INT32 can safely be used to represent an EmployeeID.
Lab Instructions

Description
In this lab you will migrate an existing 32-bit Windows* application to 64-bit. This exercise includes a file BankingTypes.h which defines account and transaction structures that will be used in a banking application. The banking application will be a 64-bit application that loads tens of thousands of accounts and transactions in memory so you need to ensure that these types are optimized for 64-bit execution. This lab includes a simple application that tests the structures defined in BankingTypes.h.

Lab Setup

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Copy the Exercise 3 zip file to the C:\Documents and Settings\Exercise3\ folder on the Itanium-based system.</td>
</tr>
<tr>
<td>2</td>
<td>Use notepad to open and examine the file Exercise3.c in the \Start folder.</td>
</tr>
</tbody>
</table>
### Migrating the Application

To migrate the application:

<table>
<thead>
<tr>
<th>Item</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On the Itanium system, navigate to the C:\Documents and Settings\Exercise3\Start folder.</td>
</tr>
<tr>
<td>2</td>
<td>Use notepad to open and edit the file BankingTypes.h.</td>
</tr>
<tr>
<td>3</td>
<td>Use the following checklist to optimize data structures for 64-bit execution.</td>
</tr>
<tr>
<td></td>
<td><strong>a.</strong> Evaluate the size of the acctID member of the _ACCT structure. acctID represents a unique account identifier. In the existing IA-32 application identifiers are in the range 1 – 1,000,000. However, the migrated code will be used for the next 10 years, and the number of accounts grows by 10% annually. Change the size of the acctID if you anticipate the current data size will not be adequate.</td>
</tr>
<tr>
<td></td>
<td><strong>b.</strong> Evaluate the size of the transactionID member of the _TRANSACTION structure. transactionID represents a unique transaction identifier. The migrated code will be used for the next 10 years. On average, a single account generates 500 transactions per year. Change the size of transactionID if you anticipate the current data size will not be adequate.</td>
</tr>
<tr>
<td></td>
<td><strong>c.</strong> Rearrange the members of the _ACCT structure and the _TRANSACTION structure to minimize padding, if needed.</td>
</tr>
<tr>
<td></td>
<td><strong>d.</strong> Modify the _TRANSACTION_LINK structure. Replace the pNext pointer to contain a DWORD offset.</td>
</tr>
</tbody>
</table>
4. Edit the file exercise3.c file. Replace the two references to the pNext member with the offset member of the _TRANSACTION_LINK that you added. Set this member to the offset from the pBase pointer. To represent the NULL value, use (DWORD) -1.

To test the application:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>At the command prompt run nmake.exe to build the 64-bit version of the application with the provided makefile. Verify that there are no compiler errors.</td>
</tr>
<tr>
<td>2</td>
<td>Run the exercise3.exe executable in the SRV2003_DEBUG folder.</td>
</tr>
<tr>
<td>3</td>
<td>Verify your changes against the solution provided in the C:\Documents and Settings\Exercise3\Solution folder.</td>
</tr>
</tbody>
</table>