Getting Your Software Ready for the IA-64 Architecture

James D. Howard
Senior Staff Software Engineer
ISV Performance Labs
Intel Corporation

originally presented at:
Intel Developer's Forum
August 31,1999 - September 2, 1999

Intel Labs

Agenda

- **Benefits of 64-bit Computing – and Issues**
- **Prerequisites and Resources**
- **Identifying and Resolving Issues**
- **Tools and Assistants: (Semi-)Automation**
- **Humans are *Still* Needed!**

Presentation plus a *ready reference*

intel®

Intel Labs

What *won't* be discussed…

- General programming for Windows or UNIX®
  - dealing with big/little endian (byte order) issues
- Ancient history: Win16, DOS, 16-bit `ints`, …
- How to use compilers and linkers
- IA-64 Instruction Set or Architecture
  - See Tue. IA-64 Track Presentation:
    Understanding the IA-64 Instruction Set Architecture
  - Instruction Set (excellent on-line tutorial):
    http://developer.intel.com/vtune/cbts/ia64tuts/index.htm
  - Architecture Guide (.PDF):
    http://developer.intel.com/design/ia64/downloads/adag.htm

Lots of IA-64 information is available *now* elsewhere; take advantage of it!

What does IA-64 do for you?

- **Removes performance bottlenecks**
  - Large register files and huge physical memory
  - Parallelism, branch prediction, memory latency hiding
- **64-bit Apps solve bigger customer problems**
  - Larger IC chips, better solids modeling, bigger databases, …
- **Fast, IEEE-accurate floating point**
  - Scalar or parallel fp; world-class SPECfp ratings
- **iA32 *binary* compatibility**
  - 32-bit x86 applications can run as they are!

IA-64 improves performance & headroom while retaining compatibility

What do 64-bit OSes enable?

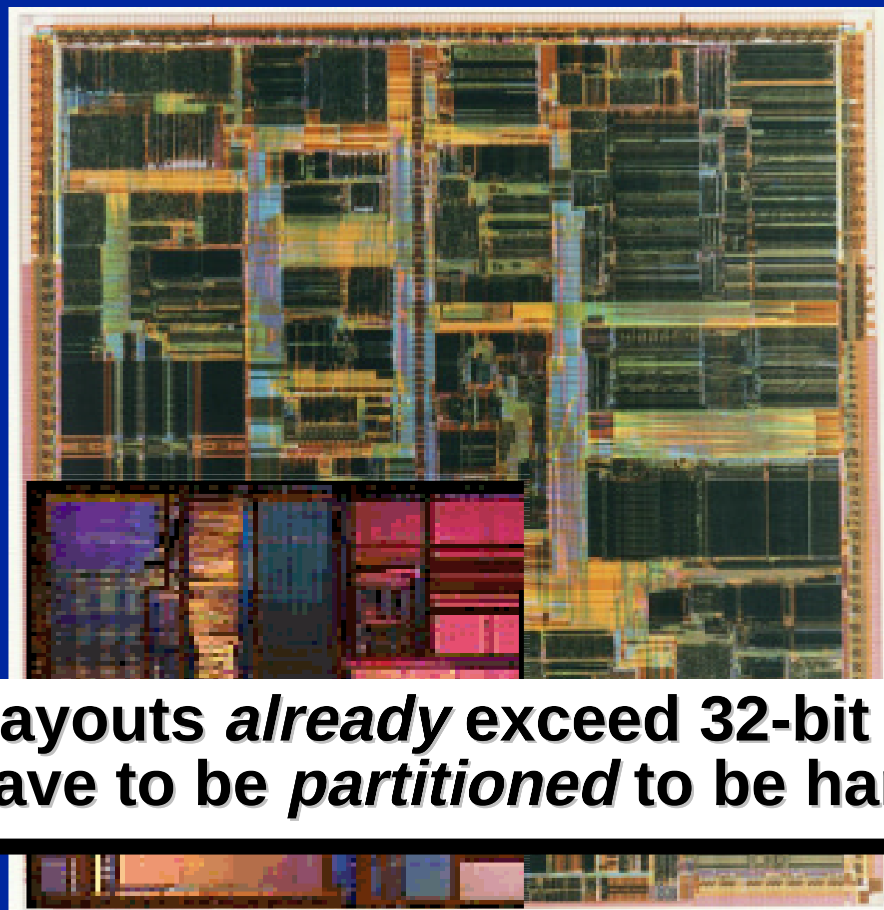
- **Big in-memory data structures**
- **Large file systems and data files**
- **Efficient large integer calculations**

- **Continued ability to run 32-bit applications**
  - recompiled into IA-64 native instructions, *or*
  - as-is IA-32 (“x86”) binaries

64-bit OSes (with their compilers) enable use of the CPU’s 64-bit features

Do I *really need* 64-bit apps?

- **Isn't 32-bits (== 2^{32} == $4.3 \times 10^9 \approx$ 4GBy) enough?**

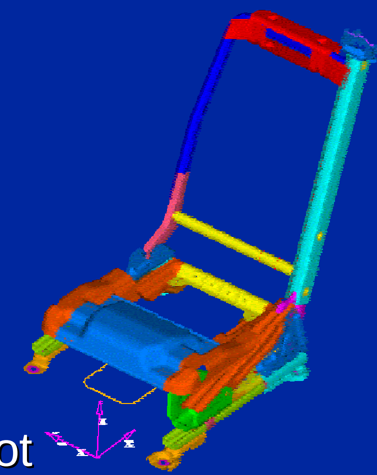


today's **13 GBy of polygon data**

4 GBy of polygon data

Today's IC layouts *already* exceed 32-bit data capacity and have to be *partitioned* to be handled…

More Apps Needing "64-bits"…



- **Mechanical CAD:**
  - Part Assemblies: >2 GBy just for a *simple* assembled part
- **Data Warehousing:**
  - DB: 100 GBy file sizes
- **ECAD:**
  - Simulation: 6×10^{10} cycles to simulate a full OS boot
- **Imaging:**
  - 2^{32} bytes is only about 20 sec. of editable, true-color HDTV!
- **Security:**
  - *fast*, wide (64-bit) multiply and add

***Real* problems *already past* 32-bit computing limits;**
Data space needs growing at ≈1 bit every 18 months!

Important Terminology

How do we talk about the issues of source code which targets multiple architectures?

- **Code Clean**
  - Revising source code to be compilable in *both* 64-bit *and* 32-bit environments
- **Polymorphic**
  - One data item having a different type to different users/viewers
  - A key source of code-clean problems
- **“Data Bloat”**
  - Increase in size of data (n.b. `structs`) in 64-bit applications
- **Cardinality**
  - The range of numbers a data item can count

Topics that could *previously* be ignored in the 32-bit world…

intel®

Intel Labs

More Terms

- **Reserved Word**
  - A source code identifier with a special meaning to the OS, compiler, source code language, header files or function call libraries
  - New ones have appeared; you *must not* redefine them
- **Base Type**
  - A language-defined, built-in data type like `long` or `short`

… as opposed to:

- **Derived Type**
  - A type defined by an OS or function library to more clearly indicate a type's purpose or size – like `HANDLE` or `time_t`
  - defined via `typedef` or `#define` in terms of a base type

Special kinds of identifiers in your source code

“Data Model”

- Size relationship between built-in (base) data types and pointers for a particular OS and hardware combination
- Set of derived types used with OS and API library functions
- Notion of “natural” or “most efficient” data type
  - `int` / `DWORD`
- Notion of “biggest natively usable type”
  - `long` / `__int64` / `long long`
- Notion of a scalable type
  - 2^{32} capacity on 32-bit machine
  - 2^{64} capacity on 64-bit machine

A Data Model defines the world the programmer “lives” in

Windows & UNIX have diverged

- **The 32-bit world: one, happy "ILP32" family**
  - **`int`**, **`long`**, **`void *`** (pointer): all 32 bits, UNIX *or* Windows
    - Same (base) types for UNIX and Windows
  - Both have named types derived from the base types
    - UNIX: **`pid_t`**, **`size_t`**, **`time_t`**, **`off_t`**, …
    - Win32: **`LONG`**, **`HANDLE`**, **`WPARAM`**, **`LPARAM`**, …
- **The 64-bit world has differences**

OS	Data Model	int	long	pointer
UNIX/64	I32,LP64	32	64	64
Windows (Win64)	IL32,P64	32	32	64

UNIX & Windows *both* tried hard to minimize changes needed in existing source code; different derived type models resulted in `long` being different

Intel Labs

UNIX/64

- **64-bit sized types:**
  - **`int64_t`**, **`uintptr_t`**, …
- **Revise some externally-visible OS fields/structures**
  - socket structure data fields changed from **`long`** to **`int`**
  - preserve size of on-disk fields like **`mode_t`**
- **carefully defining ranges for derived types like `useconds_t` – [1..1,000,000]**
- **explicit-size data types `<inttypes.h>`:**
  - **`int32_t`**, **`uint16_t`**, …
- **scalable "biggest architectural integer" type**
  - **`long` (bit-size polymorphic)**

But, Win64 approaches *scalability* differently…

Win64

- **64-bit sized types:**
  - **`__int64`**, **`intptr_t`**, **`INTPTR`**
- **Replacement data types (upgrades)**
  - New types like **`DWORD_PTR`**, **`GWLP_WNDPROC`**, …
  - now to be used in Win32 sources also
- **explicit-sized data types:**
  - **`LONG32`**, **`INT16`**, **`DWORD64`**, …
- **scalable "biggest architectural integer" type:**
  - **`__int3264`** (bit-size polymorphic)

Available and meaningful *now* for *both* Win32 and Win64 compilations (since mid-1998 MSFT platform SDK)

intel®

Intel Labs

Windows Uniform Data Model

- **Allows single source code base**
  - need to use up-to-date MSFT Platform SDK
- **New integral data types**
  - fixed bit-size integers
  - immediately recognizable pointer-as-integer type
- **Benefits:**
  - portability across Win32 and Win64
  - precise type use improves code design, understandability, maintainability

Update! Your code then works on all Windows platforms

Fixed-precision integers

- not just INT, LONG, DWORD any more

`DWORD32`	32-bit *un*signed integer
`DWORD64`	64-bit *un*signed integer
`INT32`	32-bit signed integer
`INT64`	64-bit signed integer
`LONG32`	32-bit signed integer
`LONG64`	64-bit signed integer
`UINT32`	32-bit *un*signed `INT32`
`UINT64`	64-bit *un*signed `INT64`
`ULONG32`	*un*signed `LONG32`
`ULONG64`	*un*signed `LONG64`

Explicit types can clarify code purpose

Pointer-precision integers

- These are *polymorphic* types
  - they scale with architecture

DWORD_PTR	*un*signed long type having pointer precision
INT_PTR	signed integral type having pointer precision
LONG_PTR	signed long type having pointer precision
UINT_PTR	*un*signed **INT_PTR**
ULONG_PTR	*un*signed **LONG_PTR**
SIZE_T	the maximum number of bytes to which a pointer can refer – or can span; *un*signed
SSIZE_T	signed **SIZE_T**

Many Win32 APIs use *now* these new types; your source *must* adapt to them

Intel Labs

Special pointer types

- **sized pointers**
  - `__ptr64` – 64-bit pointer
  - `__ptr32` – 32-bit pointer
- **based pointers**
  - base- plus- displacement addressing expressible in C source!
  - displacements can be 32-bits (saving space)
- **half pointers**
  - used in `struct`s that have a pointer and 2 small fields
  - can use to hold *offsets* from some 64-bit base pointer
  - `HALF_PTR`, `UHALF_PTR`
- **see Microsoft's "New Data Types" web page:** **http://msdn.microsoft.com/library/sdkdoc/buildapp/64bitwin_7zg3.htm**

Useful for careful control of `struct` sizes

New OS APIs

- **OS designers didn't *always* think ahead**
  - "size" parameters couldn't count big enough (cardinality)
  - Derived types didn't describe new capabilities
  - External (on-wire and on-disk) data structures used type names that became size-polymorphic
- **New, upward-compatible APIs and types**

OSes have added or changed APIs for 64-bits, but tried to maintain backward compatibility

UNIX

- **Expanded sizes of basic types, "consistent-ized" type usage**
- **64-bit sizes, offsets and pointers:**
  - `int64_t`, `uint32_t`, `uintptr_t`, …
- **Very few new functions**
  - 64-bit offset file operations
- **Some changed functions**
  - use "correct" size-polymorphic type

Agreed-upon by all UNIX vendors at "Aspen" conference in Feb. 1996

New Windows Functions

- **GetClassLongPtr(), SetClassLongPtr()**
- **GetWindowLongPtr(), SetWindowLongPtr()**

old:

```
LONG prev;
prev= SetWindowLong(hWnd, 12/*field 3*/, (long)pMyData);
```

new:

```
LONG_PTR prev;
prev= SetWindowLongPtr(hWnd, (int)(2*sizeof(LONG_PTR)),
    (LONG_PTR)pMyData); // C-style field count 0, 1, 2, …
```

In *both* Win32 and Win64 platform MSFT SDKs (as of mid 1998 and NT 4 SP 4)

Revised Windows functions

- **Most due to polymorphism – scale by platform**
  - **big list – 440 at last count**
    - IA-64 Track Techn. Collateral: **`CodeCln-WinAPIchg.htm`**
- **some examples:**
  - **`InterlockedIncrement()`**, **`VirtualAllocEx()`**, **`CallWindowProc()`**, **`WinHelp()`**, **`accept()`**, **`CoInstall()`**, …

old:

```
LPVOID VirtualAllocEx(HANDLE hPrcss, LPVOID addr,
   DWORD size, DWORD allocTyp, DWORD protct);
```

new:

```
LPVOID VirtualAllocEx(HANDLE hPrcss, LPVOID addr,
   SIZE_T size, DWORD allocTyp, DWORD protct);
```

Carefully changed by MSFT so that almost *everything* still works in Win32…

IA-64 *still* Runs 32-bit Code!

- **IA-64 can be used in 32-bit "mode"**
  - 32-bit source re-compiled to fast IA-64 instructions
    - best performance
  - IA-32 *binary* compatibility
    - least work
- **OSes offer *both* 32-bit *and* 64-bit runtimes**
  - 64-bit and 32-bit processes run *concurrently*
  - visit the OSV breakout sessions Thu. 2-Sep-99 for more details

OSes help IA-64 carry your 32-bit code forward: compatibility *with* native performance

Sometimes 32-bits is Enough!

- **Minimize work by leaving some apps 32-bits:**
  - Text editors
  - Line-Drawing “Drafting” applications
  - Many compiler-like apps

Useful Rule-of-Thumb: 1.5 “bits” of capacity per year

- **Measure your largest data space’ (file size, in-memory data, net transmission, etc.) capacity**
- **Multiply by 1.5^{10} ($\approx$58)**

If the result is *still* $\leq 2^{32}$, the app can stay 32 bits

  - native, fast IA-64 instructions, *or*
  - as competitive, already-built x86 binaries

Do you *really* need a 64-bit “vi” or notepad?

intel®

Intel Labs

Summary: Benefits & Issues

- **64-bit hardware, OSes and applications are *here***
  - APIs and Data Models already defined, available & usable
  - Some source code changes *are* necessary
- **Application tasks to be solved get bigger over time**
  - Need for 64-bit applications *will* increase
- **Intel's IA-64 adds other benefits to basic 64-bit capability**
  - Fast fp, µ-parallelized code, cache control, back compatibility
- **Some apps can stay 32-bit for foreseeable future**

64-bit CPUs & OSes let you reach new markets, *but* you have to prepare for it

Agenda Checkpoint:

Prerequisites and Resources

- ✓ Benefits of 64-bit Computing – and Issues
- ➢ **Prerequisites and Resources**
- Identifying and Resolving Issues
- Tools and Assistants: (Semi-)Automation
- Humans are *Still* Needed!

Find and Prepare to Fix Compatibility Issues

intel®

Intel Labs

Assumptions

- **Your source *already* compiles, links and runs in *<u>some</u>* environment**
  - Most analyses report machine-independent data that can be used to your advantage during code clean
- **You have adequate test cases**
  - Necessary: a test set that hits most any situation a user might
  - Even better: test set known to mimic user work

Make use of what already works to get more information

Characterize Your Source Code

- **Does source code use ANSI C/C++ features?**
  - Records knowledge of data types and their safe use
  - Are function prototypes used?
- **Have you used the newer data types?**
  - e.g. `intptr_t`, `size_t`, `__int64`
- **What data structures are seen *outside* your app?**
  - files, socket communication (or RPC), via shared memory, etc.
- **Are any "size" values hard-coded?**
  - e.g. use of "**4**" instead of `sizeof()` or `offsetof()`

Know what your source code *is* and *does*

intel®

Intel Labs

Source code must be up-to-date

- **Must use ANSI C function prototypes**
  - Tools like MSFT “cl /Zg” can help do this (semi-)automatically
- **Must use header files**
  - Assures implementer and user agree on interfaces
  - Don’t use undeclared functions (assumed to return `int`)
    - Find where they’re described and `#include` that header
- **Don’t use `<varargs.h>`; use `<stdarg.h>`**

Source code must have enough information for programmers or tools to perform updates

CPU/OS variant naming

- **Need a way to *name* machine differences**
- **Usable within:**
  - Source code (i.e. macros)
  - Directory structures
  - Makefiles and other build rules
  - Distribution media (i.e. file suffixes)
- **Suggestion: CPU/OS macros**
  - Examples: **`IA64-Win64`**, **`IA32-Win32`**, **`IA64-Linux`**, …
  - capable of being generated by other macros (string-pasting, …)

Standardize names of any machine variations that are *necessary*

intel®

Intel Labs

#ifdef code guards

- **examples:**

`_M_IA64`	**compilation is generating code for IA-64 execution**
`_WIN64`	**generate code for Win64 ABI**
`_WIN32`	**code uses the Win32 API (this *includes* Win64)**
`_WIN32_WINNT`	**code runs *only* on the Windows NT variant of Win32 (not Win/95 nor Win/98)**

- **see more complete list on IDF CD-ROM**
  - there's *lots*!
  - IA-64 Track Techn. Collateral: **`CodeCln-CPreDefsWin32.htm`**

Use *standard*, up-to-date conditional compilation macros

What are the *Important* Parts?

- **Data structures that are used a lot**
  - Confirm size needed for each data item
  - Can pointers be eliminated?
    - Array indexes or base-plus-displacement addressing
  - Can some fields be moved to another (or new) structure
- **Routines that are called a lot**
  - Do you *really* need all those parameters?
  - Are any parameters "too big"?
    - Are you returning TRUE/FALSE in a `long`?
    - Will a 32-bit offset be enough – instead of a pointer pass?

Optimize where you'll get the best results

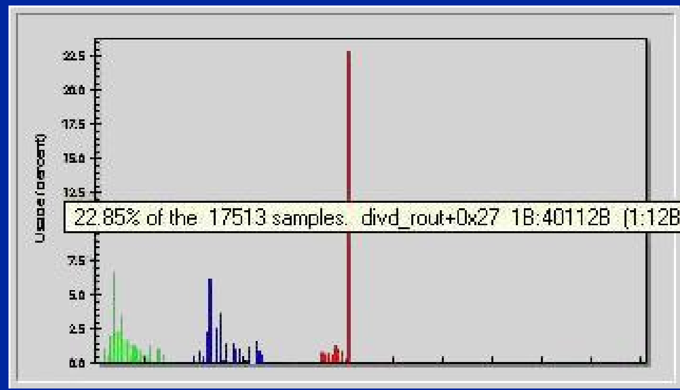
Static Analysis

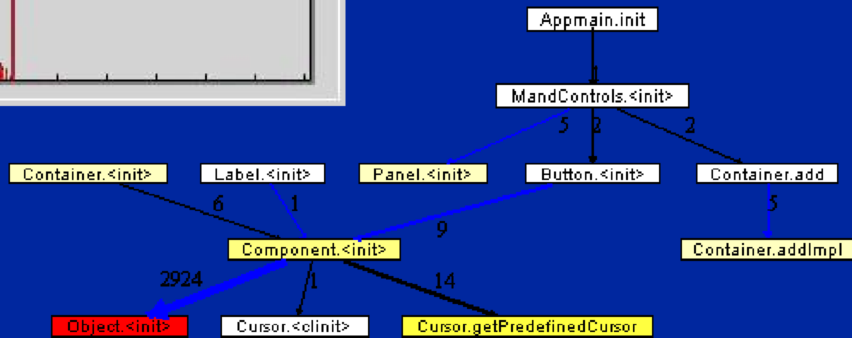
- **Can give a good initial guess**
  - What type mismatches are immediately identifiable?
  - How many times are structures referenced?
  - How many times are functions called?
- **Tools are available to do this**
  - Compiler
    - header nesting list “**`icl /QH`**”, “**`/QM`**”, similar gcc opts.
    - (filtered) warnings lists
  - Source code browsers
    - trace “definers” and “users” of variables and structures

Identify where to apply the most effort

Dynamic Analysis

- **Runtime analysis gives an accurate picture**
  - application “hotspots”, call graph
- **Tools:**
  - VTune
  - Quantify
  - APIMON
  - …





Tools can help you pinpoint *best-benefit* spots

Intel Labs

Use *multi*-Standard Data Types

- **Use types available in *both* UNIX and Windows**
  - **`short`**, **`int`** mean the same thing everywhere
    - *very* likely to stay that way
  - **`signed`** and **`unsigned`** qualifiers available everywhere
  - **`char`** is always an 8-bit type
    - use **`*sign*`** qualifier if really needed (rarely)
- **Newer types available in *both* worlds**
  - **`intptr_t`** - integer big enough to hold a pointer
    - **`uintptr_t`** (unsigned variant)
- **Not perfect – does not help reader/maintainer as much as it could…**

Avoid conditional compilation by using types common to *both* UNIX and Windows

Some Types Aren’t Portable

- Avoid “one world” types whenever possible

Windows *only*	ANSI C/C++ or *common, both worlds*	UNIX
`INT_PTR`	`intptr_t`	`intptr_t`
`SIZE_T`	`size_t`	`size_t`
`int32_t`	`int`	`__int32`

Avoid conditional compilation by using types common to *both* UNIX and Windows

intel®

Intel Labs

Still Some Data Type Problems

- **Need "largest efficiently handled integral" type**
  - **`long`** in UNIX/32, UNIX/64 and Win32, but…
  - **`__int64`** in Win64 (and **`long`** not right)
  - **`__int3264`** in Win32 and Win64 in **`<basetsd.h>`**
    - But, not recommended by Microsoft?
- **No good, consistent solution**
  - Users want a type that *visually* means **`int`** (helps reader)
- **Possible solutions**
  - Give up on visual semantics; use **`intptr_t`** or **`uintptr_t`**
  - Make a **`typedef __int3264`** in UNIX code compilations

Sometimes there is no built-in solution common to *both* UNIX and Windows; use a macro or `typedef`

intel®

Intel Labs

Explicitly Sized Data Types?

- **Sized data types: good aids to reader/maintainer**

type size	UNIX `<inttypes.h>`	ANSI	Windows
8 bits	`int8_t` `uint8_t`	`char`	`__int8` `unsigned __int8`
16 bits	`int16_t` `uint16_t`	`short` `unsigned short`	`__int16` `unsigned __int16`

32 bits	`int32_t`
	`uint32_t`
		`int`
		`unsigned int`
			`__int32 INT32`
			`unsigned __int32`
			`UINT32`

64 bits	`int64_t`
	`uint64_t`
		(no uniform std.)	`__int64 INT64`
			`unsigned __int64`
			`UINT64`

avoid confusion: pick *one, others* are `typedefs`…

Use a compatibility header file

- **`<compatible.h>` #include file**
  - used by *all* your source files
  - write your source using these standard-inspired types
  - example in IA-64 Track Technical Collateral: **`CodeCln-compatible.h`**

```
/* compatible.h */
#if defined(_WIN32)
… // stuff related to Win32
#if !defined(_WIN64)
… // Win32 without Win64 (regular Win32)
#else /* is _WIN64 also */
… // Win64 variant of Win32
#endif /* _WIN64 */
#endif /* _WIN32 */
#elif defined(__unix) || …
… // various UNIXes
#else /* some other OS */
#error Unhandled OS;
#error   update <compatible.h>!
#endif
```

Enhance portability and readability of source using a common compatibility #include file

Intel Labs

Summary: Prereq. & Resources

- **Know your source**
  - structure
  - dependencies
  - data types
- **Find “hot spots”**
  - most bang-for-the-buck for your work
- **Decide on data model for *you***
  - compilation macros unify code readability & maintainability
  - create a **`<compatible.h>`** **`#include`** file
    - see sample in IA-64 Track Technical Collateral: **`CodeCln-compatible.h`**

Knowledge of source and data models gets you ready for Code Clean

Agenda Checkpoint:

Identifying & Resolving Issues

- ✓ Benefits of 64-bit Computing – and Issues
- ✓ Prerequisites and Resources
- ➢ **Identifying and Resolving Issues**
- ● Tools and Assistants: (Semi-)Automation
- ● Humans are *Still* Needed!

Find and fix these issues in your code

Catalog of Tips & Techniques

- **Code clean involves many little details**
  - each easily recognizable
- **Useful list of situations**
  - most with easy fixes
  - some require careful human thinking
- **Some repairs can be automated**

Ready reference for Issues & Fixes

Pointer casts to integer type

```
char *buf;
…
int i;
…
i= (int)buf;


uintptr_t ip;
…
ip= (uintptr_t)buf;
```

Problem(s):

- Pointers are bigger than `int`s in some architectures
- Using `long` won't help in Win64
- Pointers *logically* unsigned

Remedy:

- Use `uintptr_t`; works on both UNIX and Windows

eliminate *all* cases of `(int)pointer` casts

intel®

Intel Labs

Setting Bits in Pointers

```
char *handle;
…
return (PVOID)((UINT)handle|1);
```

Problem(s):

- (cast) prior to logical "OR"
- (UINT) is *also* smaller than a pointer

```
…
return
  (void *)((uintptr_t)handle|1);
```

Remedy:

- Use `uintptr_t` and `void *`; works on both UNIX and Windows

`(UINT)` is just as bad as `(int)`

intel®

Intel Labs

Field Indexing

```
struct S {
  void *pn;
  int ln;
};
S *Ps= new(S);
int i;
i= *(int *)((uintptr_t)Ps + 4);

i= *(int *)((uintptr_t)Ps +
  offsetof(S,ln));
```

Problem(s):

- field offsets can vary across compilers
- any constant added to a pointer should be suspect
- natural alignment differs

Remedy:

- use ANSI C `offsetof()` macro – *not `sizeof()`*

Let the compiler calculate field offsets

#define for constants: don't!

```
#define mask 0x37FFC;
```

```
const int mask= 0x37FFC;
```

Problem(s):

- using a #define that the compiler can't type check

Remedy:

- use ANSI C's "const"
- use a specific data type
- you'll get warned if any misuse is attempted

Let the compiler check declarations for you

Integer Constant Type Suffixes

```
// from UNIX/64-safe code
const long ll= 3015L;
```

compatibility.h

```
#ifdef _WIN64
#define CONST3264(a) (a##i64)
#else
#define CONST3264(a) (a##L)
#endif
…
```

```
const long ll= CONST3264(3015);
```

Problem(s):

- "L" (or "l" – long) suffix only OK for UNIX/32, UNIX/64 and Win32
- Win64 needs "i64" suffix

Remedy:

- #ifdef test for _WIN64
- Use token pasting in a macro you define elsewhere

Test: What's *still* wrong?

Centrally-defined macro makes source generic

"Big" integer declarations

```
long bigCount;
```

compatibility.h

```
#ifdef _WIN64
…
typedef __int64 int3264_t;
#else
…
#endif
```

```
int3264_t bigCount;
  // an int as big as architecture permits
```

Problem(s):

- `long` used to declare "architecture's biggest integral type" variable
- want "`__int64`" on Win64

Remedy:

- `#ifdef` test for platform
- use a typedef where needed
- reader/maintainer can see variable is morphic

Centrally-defined macro makes source generic

Issues with Hex constants

```
0xFFFFFFFF
   // 32-bits: -1, 64-bits: 4,294,967,295
0x100000000
   // 32-bits: 0, 64-bits: 4,294,967,296

const int all1s= 0xFFFFFFFF;
```

Problem(s):

- generating "all 1s" in hex
- using a #define that the compiler *can't* type check
- "-1" in 32-bit system, 4,294967,295 in a 64-bit system

Remedy:

- use ANSI C's "const"
- use a specific data type
  - signed/unsigned
- use type suffixes – "L", "UL"

Count the digits!

Truncation of long via doubles

```
long l;     // if UNIX
__int64 l; // if Windows

double d;
extern double f(double);
…
d= l; // looses significant bits!
l= d; // this can be a problem, too
…
d= f(l);
```

Problem(s):

- double: 52 significant bits
- 32-bit world: double can hold all significant bits
- 64-bit world: precision of double smaller than 64-bit integer; bits can be lost

Remedy:

- have programmer verify:
  - always OK?
  - loss of significant digits OK?
- use long double (or REAL*10) if possible

longs and doubles don't mix any more

Reading legacy data

OnDisk.h

```
struct OnWire {
  long count;
  // more fields
  };
```

OnDisk.h

```
#include <compatibility.h>
struct OnWire {
  int32_t count; // data's own size
  // more fields
  };
```

Problem(s):

- existing files or protocols have 32-bit data described as “long”

Remedy:

- use specific-sized type of your choice (from your compatibility header)

headers describe external data in *its* terms

Structure Padding

```
struct dim_t {
  int height;
  long width;   // UNIX original
  int weight;
  }; // size: 32-bit: 12, 64-bit: 24
```

save 33%!

```
struct dim_t {
  __int3264 width;
    // now scales portably
  int height;
  int weight;
  }; // size: 32-bit: 12,  64-bit: 16
```

Problem(s):

- **64-bit: field order yields padding**
  - data structure bloat
  - *very* bad if many of these records
- **is record internal-only?**

Remedy:

- **Re-order fields, longest first**
- **check cardinality: does `width` *really* have to be a `long`?**

field alignment changes in 64-bit world

intel®

Intel Labs

unions (please don't)

```
union {
  long l;
  char bytes[4];
  };
…
for (i= 0; i<4; i++) …
```

Problem(s):

- union for alternate access method incorrect for 64-bits

Remedy:

- 1st: avoid union if at all possible

```
union {
  __int3264 l; // chg w/ architecture
  char bytes[sizeof(__int3264)];
  };

for (i= 0; i<sizeof(bytes); i++) …
```

else:

- fix primary data type size
- use C `sizeof()` builtin for array

unions look ugly and cause lots of problems; don't use them unless necessary

Intel Labs

printf() format strings

```
long *Pl; // Win32 source
printf("%08lX->%ld\n",Pl,*Pl);
```

compatibility.h

```
#ifdef _WIN64
#define FMTSZ3264 "I64"
#else /* Win32 or UNIX */
#define FMTSZ3264 "l"
#endif
```

```
__int3264 Pl;
printf("%p->%" FMTSZ3264 "d\n",Pl,*Pl);
```

Problem(s):

- "l" argument size specifier used with platform-scaled type
- don't use "%X" for pointers
- "I64" does not scale – it is *not* polymorphic

Remedy:

- use "%p" to print a pointer
- use a macro and adjacent string catenation

Fix printing of pointers and "big" integers

intel®

Intel Labs

setjmp/longjmp

```
DWORD jbuf[16];
…
longjmp(jbuf,1);
…
setjmp(jbuf);
```

```
#include <setjmp.h>
…
jmp_buf jbuf;
```

Problem(s):

- not using ANSI `jmp_buf`

Remedy:

- use ANSI C header
- declare using `jmp_buf`
- don't assume you can know *anytyhing* about the "insides" of `jmp_buf`

Don't use hard-coded constants for system-defined features

intel®

Intel Labs

Pointer truncation - *carefully*

```
mystruct *p;

unsigned int lowBits=
  (unsigned int)p;
  // truncation warning in Win64

unsigned int lowBits=
  PtrToInt(p);
  // truncation warning silenced

p= (mystruct *)lowBits;
```

- **only do this if you *really* have to…**

Problem(s):

- **pointer truncations dirty your compilation listings**

Remedy:

- **Windows' `PtrToInt()` silences warnings**
- **see `<basetsd.h>`**

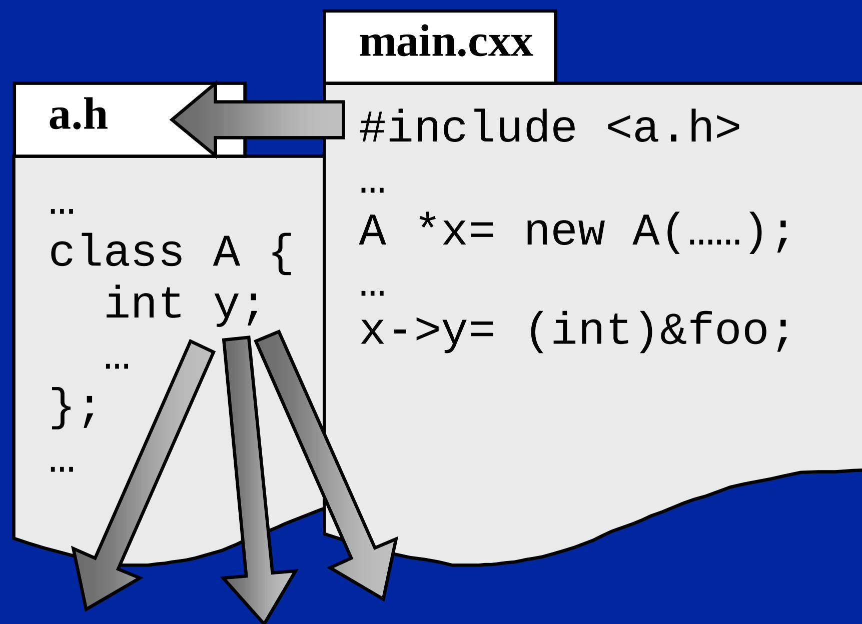
Caution:

- **Never, ever use data as pointer again; significant bits are *gone***

Be careful if you forcefully silence warnings

Source change consequences

- Fix at use, implies
- fix in header, implies
- lots of other places to check
  - busy work; ideal opportunity for a tool
- More about this shortly…



Changes can ripple widely

Intel Labs

Summary: Identify/Resolve Issues

- **Quite a list of issues and gotcha*!*s**
- **Most have simple solutions**
  - aid: compatibility declarations in an **`#include`** file
- **Many situations can be machine-recognized**

Fixing issues will improve code reliability and reduce support costs on *all* platforms!

Agenda Checkpoint:

Tools, Assistants & Automation

- ✓ Benefits of 64-bit Computing – and Issues
- ✓ Prerequisites and Resources
- ✓ Identifying and Resolving Issues
- ➢ **Tools and Assistants: (Semi-)Automation**
- ● Humans are *Still* Needed!

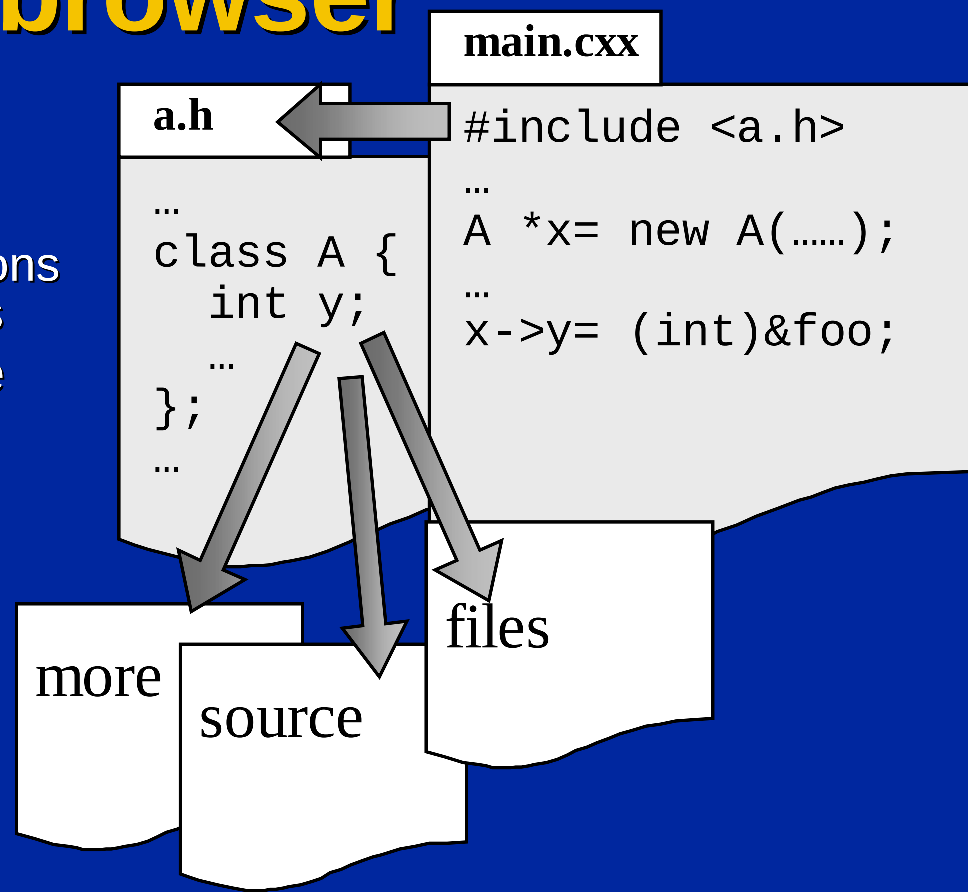
All is not lost; tools can help do lots of the busy work…

intel®

Intel Labs

Source code browser

- **use browsers to identify definitions and uses**
  - identify all uses of definitions made in common headers
- **`int`** cast ⇒ 64-bit truncate
- **`(int)`** ⇒ **`(intptr_t)`**
- “**`=`**“ now mismatch!
- find **`typeof(*x)`**
- fix field **`y`**
- *now* have to check other uses of **`A.y`** throughout sources
- **browser finds them for you**



Browsers find definitions and uses for you

C/C++ Compiler

- **Microsoft "cl" for IA-64**
  - in Windows 2000 Platform SDK; free download: get it!
    - http://msdn.microsoft.com/developer/sdk/platform.asp
  - Compiler front-end (only)
    - does syntax, semantic and type checking
  - Use it to find "bad" spots – "/Wp64"
  - Fix them
  - Compile with IA-32 (VC++) "cl" to verify portability
- **Intel will offer an IA-64 compiler product also**
- **visit OSV breakouts Thu. 2-Sep-99 for information on other OSes**

IA-64 type-test compilers available *now*

Conversion Scripts

- **Usually for a specific from/to conversion**
  - works when you *know* your source initial characteristics

Example script on IDF CD-ROM: `CodeCln-U64toW64.zip`

- **Edit UNIX/64-safe code to be also Win64-safe**
  - applicable to command-line / batch tools

  actions:
  - adds **`#include`** defining transparency macros
  - change uses of **`long`** to **`int3264_t`**
  - fix **`printf()`**/**`scanf()`** format strings via string catenation
  - fix type-suffixed constants
  - converts *all* **`.c`** and **`.h`** files in a source tree
- identify points for further human investigation

scripts do *only part* of the conversion

Win32⇒Win64 conversion

Another example script on IDF CD-ROM:
`CodeCln-W32toW64.zip`

- **Revise Win32 code to be Win64 safe**
  - updates out-dated API calls
  - fixes some polymorphic mixing of integers and pointers
  - converts *all* `.c` and `.h` files in a source tree

Script needed to catch all Win32/Win64 changes

Summary: Tools & Assistants

- **Source-knowledgeable tools identify problems**
- **Tools can automatically edit your source for portability**
- **These tools can be:**
  - as simple as scripts
  - as capable as tools built on compiler front-ends

Much conversion drudgery can be avoided

Humans are *Still* Needed!

- ✓ Benefits of 64-bit Computing – and Issues
- ✓ Prerequisites and Resources
- ✓ Identifying and Resolving Issues
- ✓ Tools and Assistants: (Semi-)Automation
- ➢ **Humans are *Still* Needed!**

Is automated code clean enough?

Automation is Non-optimal

- **Automation tools make safest choice**
  - conditional compilation (rather than multi-platform fix)
  - conflicts are resolved by "growing" to larger data types
  - automation implies "data bloat"
- **Automation tools don't know code logic**
  - no knowledge of "how big" counts or offsets become
  - don't know when a (cast) would be OK
  - Can't check algorithms for 64-bit appropriateness
- **Source and build areas become multi-platform**

Architect must guide/correct automation

Cardinality (data type size)

- **You have a `long` variable; what *should* it be?**

  (Assuming it is *not* pointer polymorphic)

Apply a Rule-of-Thumb:

- **Consider your largest data range – say “[0..1,000,000]”**
- **Multiply by 1.5^{10} (≈58; $1^1/_2$ bits a year for 10 years)**

If the result Is *still* <<2^{32}

- **code the variable as `int`**

Avoid data bloat by knowing *how big* a variable must *count*

More data size issues (review)

- **`long` (I32,LP64) and `__int64` (IL32,P64) vs. `double`**
  - Mantissa/significand of 64-bit double is only 52 bits
  - Can't be converted to/from 64-bit integer without loss of precision
- **`structure` fields may need to be re-ordered:**
  - To remain naturally aligned
  - To minimize padding inserted by compilers
- **Packing `#pragmas` may be needed**
  - To force back-compatible fields in on-disk or on-wire structures

Architect must guide/correct automation

Data Structure Expansion Fixes

```
struct f {
  f* Pnextf;    // 32-bits: 4 bytes,
  g* Psibling; // 64-bits: 8 bytes
  h* Pparent;
  …
  };
```

```
struct f {
  HALF_PTR off_nextf;
  int      idx_sibling;
  h __based(heap)*__ptr32 parent;
    // 32-bit displ from 64-bit base
  …
  };
```

Problem(s):

- **in-memory data structures with lots of pointers**
- **every pointer 2× bigger**

Remedy:

- **Try to use base+displacement addressing**
- **Can the structures be arrays?**
- **try __based pointers (MSFT specific)**

Algorithm re-work needed to make structures efficient

intel®

Intel Labs

Check hash algorithms

- **hashing depends on “scrambling” data values**
  - bit shifts and logical operations
  - distribution good *only when* lots of *significant* data to scramble
- **pointers & seek keys frequently used as hash keys**
  - 32-bit pointers or object identifiers
    - about 30 significant, varying address bits
  - 64-bit pointers or object identifiers
    - about 35 significant, varying address bits
    - lots of 0s (zeros) and 1s that are largely unchanging
- **check shifting and logical operations in hash**

operate on *significant* bits of pointers

Build Rules

- `makefiles` **and build rules need to be parametric**
- **Multiple source and result directories**
- **Different file suffixes on different platforms:**
  - **`.o`**/**`.OBJ`**; **`.EXE`**/(no suffix); **`.a`**/**`.LIB`**; **`.so`**/**`.DLL`**

Parameterize build rules by platform

Software directory tree

- **source mostly same**
  - some `#ifdefs`
  - occasional alternate files
- **built (derived) files**
  - data files mostly platform independent
  - binaries vary by platform
    - **`.OBJ`**, **`.LIB`**, **`.DLL`**, **`.EXE`**
- **Explicitly model CPU/OS variants**

organize projects by machine in/dependence

Humans are *Still* Needed: Software Build Issues: Directory Tree

Example

```
SoftwareTree\
|   master.mak
+---Project1
|   |   Project1.mak
|   +---src
|   |   |   BTree.cxx, PtrHash.cxx, main.cxx, msgstrings-Ena.txt, ...
|   |   +---iA64-Win64
|   |   :       Win64Ifc.cxx
|   +---include
|   |       BTree.h, PtrHash.h, ...
|   +---data
|   |       msgstrings.Ena
|   +---IA64-Win64
|   |   +---lib
|   |   |       FuncFwk.lib
|   |   +---exe
|   |   |       FuncFwk.dll, DrawMap.exe
|   |   `---data
|   |           MachDepChars.dat
|   +---IA64-Linux
|   |   +---lib
|   |   |       FuncFwk.a
:   :   :
```

Most source platform-*in*dependent or safe via localized `#ifdefs`

Machine-varying source *should be* rare, at tree bottom

Most built objects are platform *de*pendent; use CPU/OS id

Best code clean has human help

- **Automation tools only make good guesses**
- **Code architect knows**
  - what data fields hold
  - which are used most
- **Human is best weapon against data bloat**

Your boss can't replace you **(yet)**

intel®

Intel Labs

Summary: Humans *Still* Needed

- **Automated scripts and tools can only go so far**
  - their “safe” decisions are not always most optimal
  - they *can* alert the programmer to situation to look at
- **Only architects can decide**
  - safe optimizations
  - algorithm tweaks
- **Changes to source edit and build environments**
  - allow for machine-dependent sources
  - per-machine generated files (`.EXE`s, `.DLL`s, resources, …)

Only humans can make code logic changes

Agenda Checkpoint: We're Finished!

Getting Readying for iA64

- ✓64-bit CPUs & OSes let you reach new markets, *but* you have to prepare for it
- ✓Knowledge of source and data models gets you ready for Code Clean
- ✓Fixing issues will improve code reliability and reduce support costs on *all* platforms!
- ✓Much conversion drudgery can be avoided
- ✓Only humans can make code logic changes

Now you know enough to do it!

intel®

Intel Labs

Call to Action

Pick some favorite small tool and migrate it to IA-64 as soon as you get out of here!

Contact your favorite OS Vendors about IA-64 development kits

- **Get the Win64 Platform SDK**
- **Learn about your source code – it *will* surprise you**
- **Use the migration scripts and make them your own**
- **Sell more by being *early*, but *timely*, to market**

Collateral

Some sources of added/updated information

- **Intel IA-64 information**
  - "IA-64 Application Developer's Guide," Intel Literature order # 245188-001
  - "IA-64 Software Conventions and Runtime Architecture Guide", Intel Literature order # 245256-001
  - http://developer.intel.com/design/ia64
- **UNIX I32,LP64 rationale:**
  - **http://www.opengroup.org/public/tech/aspen/lp64_wp.htm**
- **Windows "win64" Programming information**
  - **http://msdn.microsoft.com/library/sdkdoc/buildapp/64bitwin_410z.htm**

Much information now available

intel®

Intel Labs

Contacts

- **Your Intel “SRM” or “AM” (Field Application Engineering Facilitators)**
- **Intel’s Developer Web Site: http://developer.intel.com**
  - hardware *and* software information
- **Author**
  - `james.d.howard@intel.com`

intel®

Intel Labs