Subject: SSE2(/AVX) and alignment issues
Posted by mirek on Fri, 28 Jan 2011 09:55:45 GMT
View Forum Message <> Reply to Message

I have spent some time thinking about some more direct support of SSE2 in U++ and there is a stupid problem:

U++ allocator guarantees 8 byte alignment only, while SSE2 requests 16 bytes alignment.

I am now undecided which approach would be good:

- I could make allocator 16-byte aligned, at the price of about 4% less space effectivity. This is fine, but when AVX comes, we are then at 32bytes alignment, then with another AVX version at 64 bytes etc... and that would have much worse results.

- I could make _containers_ (Vector etc) alignment aware. This is much superior approach as it does not require any sacrifices in space efficiency and is future proof w.r.t. changing alinment requirements as AVX grows in width. However, one big problem is that you would not be able to use 'new' to allocate SSE2 objects.

Of course, there is not much use for 'new' in U++ anyway, but still...

(Actually, life would be much easier in C++ if there would not be any 'new' and 'delete'

Mirek

Subject: Re: SSE2(/AVX) and alignment issues
Posted by tojocky on Fri, 28 Jan 2011 12:48:01 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 28 January 2011 11:55I have spent some time thinking about some more direct support of SSE2 in U++ and there is a stupid problem:

U++ allocator guarantees 8 byte alignment only, while SSE2 requests 16 bytes alignment.

I am now undecided which approach would be good:

- I could make allocator 16-byte aligned, at the price of about 4% less space effectivity. This is fine, but when AVX comes, we are then at 32bytes alignment, then with another AVX version at 64 bytes etc... and that would have much worse results.

- I could make _containers_ (Vector etc) alignment aware. This is much superior approach as it does not require any sacrifices in space efficiency and is future proof w.r.t. changing alinment requirements as AVX grows in width. However, one big problem is that you would not be able to use 'new' to allocate SSE2 objects.

Of course, there is not much use for 'new' in U++ anyway, but still...

(Actually, life would be much easier in C++ if there would not be any 'new' and 'delete'

Mirek

I vote for first realization. It is impossible to live without new and delete.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by cbpporter on Fri, 28 Jan 2011 12:55:51 GMT
View Forum Message <> Reply to Message

How about an optional flag for option number 1. Compile with alignment if memory is not an issue for you.

for option number two, can't we have VectorSSE2 variant living in parallel with normal one?

Subject: Re: SSE2(/AVX) and alignment issues
Posted by Didier on Fri, 28 Jan 2011 17:43:53 GMT
View Forum Message <> Reply to Message

Hi,

new and delete operators can be overloaded to do what you want.
I already had to do this in my SW for exactly the same reason.

Another solution is to use a <template> replacement new().

I explain myself:

There is the 'placement new()' operator which calls the constructor of an object on a previously allocated memory.
The following code takes advantage of this to create generic allocators/constructors classes. (NB these classes do not need to be instanciated, all methods are static ==> 'typedef' can be used to create custom allocators) !

```
class DefaultAllocator
{
 private:
  DefaultAllocator ( void ) {}
  DefaultAllocator ( const DefaultAllocator& ) {}
  ~DefaultAllocator ( void ) {}
```

```
  DefaultAllocator& operator= ( DefaultAllocator& ) { return *this; }

 public:
  static char* alloc ( size_t size )
  {
   return new char[size];  // -------- use the allocator you want -----------
  }

  static void free ( char* buf )
  {
   delete [] buf;         // --------- use the allocator you want -----------
  }
};


template < class ALLOCATOR = DefaultAllocator >
class AllocatorConstructor
{
 public:
  // allocation helper methods
  static void* alloc(size_t size)
  {
   return ( (void*) ALLOCATOR::alloc(size) );
  }

  template<class BUFFER>
  static BUFFER* alloc(void)
  {
   return ( (BUFFER*) ALLOCATOR::alloc(sizeof(BUFFER) ) );
  }

  template<class BUFFER>
  static void free(BUFFER* buf)
  {
   ALLOCATOR::free((char*) buf);
  }

  // allocation / construction  methods
  template<class OBJ>
  static OBJ* allocConstruct ()
  {
   OBJ* res = alloc<OBJ>(); // Allocate the memory
   new ( res ) OBJ (); // Call the constructor on allocated memory
   return res;
  }


  // allocation / construction  methods
```

```
  template<class OBJ, class PARAM1 >
  static OBJ* allocConstruct ( PARAM1 p1 )
  {
   OBJ* res = alloc<OBJ>(); // Allocate the memory
   new ( res ) OBJ ( p1 ); // Call the constructor on allocated memory
   return res;
  }

  template < class OBJ, class PARAM1, class PARAM2 >
  static OBJ* allocConstruct ( PARAM1 p1, PARAM1 p2 )
  {
   OBJ* res = alloc<OBJ>();    // Allocate the memory
   new ( res ) OBJ ( p1, p2 ); // Call the constructor on allocated memory
   return res;
  }

  template<class OBJ>
  static void freeDestruct ( OBJ* buf )
  {
   buf->~OBJ();      // explicit call to destructor
   free<OBJ>( buf ); // free the memory
  }
};


// ============= USAGE  EXAMPLE ==================
class MallocAllocator
{
 private:
  MallocAllocator ( void ) {}
  MallocAllocator ( const MallocAllocator& ) {}
  ~MallocAllocator ( void ) {}
  MallocAllocator& operator= ( MallocAllocator& ) { return *this; }

 public:
  static char* alloc ( size_t size )
  {
   return (char*) malloc(size);  // -------- use the allocator you want -----------
  }

  static void free ( char* buf )
  {
   free(buf);          // --------- use the allocator you want -----------
  }
};
```

```
class Myclass
{
 public:
  int a;
  Myclass() { a = 5; }
  Myclass(int p) { a = p; }
};

// USING 'typedef' you can define system wide definition of you're allocator
typedef AllocatorConstructor<DefaultAllocator> DefaultAllocConst;
typedef AllocatorConstructor<MallocAllocator> MyAllocConst;

void testFct()
{
 Myclass* ptr = DefaultAllocConst::allocConstruct<Myclass>();
 Myclass* ptr2 = MyAllocConst::allocConstruct<Myclass>(10);

 // do you're thing !

 DefaultAllocConst::freeDestruct( ptr );
 MyAllocConst::freeDestruct( ptr2 );
}
```

This sample class can be easily extended to manage alignement as well (using template parameter ==> 8/16/32... would therefore be set using the typedef.

This is surely can be fit into Upp.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by Novo on Fri, 28 Jan 2011 20:53:48 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 28 January 2011 04:55However, one big problem is that you would not be able to use 'new' to allocate SSE2 objects.

Of course, there is not much use for 'new' in U++ anyway, but still...

What is SSE2 object?

You can allocate sizeof(object) + alignment - 1 of memory and use placement new/delete to call constructor/destructor.

```
// A must be power of two.
template <unsigned A>
void* AlignMem(void* addr)
{
   unsigned a = A - 1;
   return (void*)((unsigned)addr + a) & ~a;
}

void* ptr = UPP::MemoryAlloc(sizeof(object) + alignment - 1)
object* o = new(AlignMem<alignment>(ptr)) object(aarg1, arg2, e.t.c.);
...
o->~object();
UPP::MemoryFree(ptr);
```

You need an extra pointer to allocated memory.

As far as I understand these SSE2 commands are going to be used with complicated data structures like matrices, vectors, strings, e.t.c. So, this pointer and allocation/destruction logic can be hidden inside of overloaded new/delete.

I personally would prefer to have an allocator, which allocates memory aligned by 16/32/64 bytes. AVX seems to be supported by Intel only at this time. So, you do not want to use AVX for quite long time. By the time we will need 64-byte aligned memory it might cost 1$ for a Gbyte.

mirek wrote on Fri, 28 January 2011 04:55
(Actually, life would be much easier in C++ if there would not be any 'new' and 'delete'

Mirek

Garbage collector in C++ is a real pain in the neck also. Your object can be garbage collected when you are still in a constructor.

I personally would prefer new/delete.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Fri, 28 Jan 2011 22:59:16 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Fri, 28 January 2011 07:55
for option number two, can't we have VectorSSE2 variant living in parallel with normal one?

No need to. I can make Vector "alignment aware" without compromising performance...

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Fri, 28 Jan 2011 23:03:28 GMT
View Forum Message <> Reply to Message

Novo wrote on Fri, 28 January 2011 15:53mirek wrote on Fri, 28 January 2011 04:55However, one big problem is that you would not be able to use 'new' to allocate SSE2 objects.

Of course, there is not much use for 'new' in U++ anyway, but still...

What is SSE2 object?

You can allocate sizeof(object) + alignment - 1 of memory and use placement new/delete to call constructor/destructor.

```
// A must be power of two.
template <unsigned A>
void* AlignMem(void* addr)
{
   unsigned a = A - 1;
   return (void*)((unsigned)addr + a) & ~a;
}

void* ptr = UPP::MemoryAlloc(sizeof(object) + alignment - 1)
object* o = new(AlignMem<alignment>(ptr)) object(aarg1, arg2, e.t.c.);
...
o->~object();
UPP::MemoryFree(ptr);
```

You need an extra pointer to allocated memory.

As far as I understand these SSE2 commands are going to be used with complicated data structures like matrices, vectors, strings, e.t.c. So, this pointer and allocation/destruction logic can be hidden inside of overloaded new/delete.

This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Fri, 28 Jan 2011 23:05:54 GMT
View Forum Message <> Reply to Message

Didier wrote on Fri, 28 January 2011 12:43
This sample class can be easily extended to manage alignement as well (using template parameter ==> 8/16/32... would therefore be set using the typedef.

This is surely can be fit into Upp.

Sure. But all of that is not using 'regular' new. It is "new is deprecated" or "do not use regular new for SSE2 stuff" way...

Ergo, option 2.

Mirek

Subject: Re: SSE2(/AVX) and alignment issues
Posted by Novo on Sat, 29 Jan 2011 01:07:11 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 28 January 2011 18:03
This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

I'm voting for 16-byte aligned new. This will make life simple. And U++ is all about making life simple.

In an ideal world I'd like to see an allocator, which can allocate memory with a given alignment of power of two, compile time changing of alignment of 'new', and data structures which can adapt to different memory alignment.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by tojocky on Sat, 29 Jan 2011 08:23:33 GMT
View Forum Message <> Reply to Message

mirek wrote on Sat, 29 January 2011 01:03

This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

Mirek,

Can you give us an example of "new" realization and "allocator" realization? Why you are not agree with new realization (option 1)?
Maybe we can manage by directives and implement the both options?

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Sat, 29 Jan 2011 19:29:29 GMT
View Forum Message <> Reply to Message

tojocky wrote on Sat, 29 January 2011 03:23mirek wrote on Sat, 29 January 2011 01:03

This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

Mirek,

Can you give us an example of "new" realization and "allocator" realization?

```
struct AvxSomething {
  _m256 x;
};

Array<AvxSomething> foo;

foo.Add(new AvxSomthing); // not supported in option2. Actually, not even supported by any compiler today
```

foo.Add<AvxSomething>(); // supported in both options

Option2 could also support e.g.:

foo = New<AvxSomething>();
foo = new (Aligned<AvxSomething>) AvxSomething;
foo = NEW(AvxSomething);

Delete(foo);

(The crucial problem is that we need to know the type in new/delete).

Quote:
Why you are not agree with new realization (option 1)?

Because it wastes memory. It means that every single block allocated by 'new' must be a multiple 16 bytes allocated for SSE2.

Then 32 bytes for AVX and AVX is supposed to grow in width, so it can be easily 64 bytes etc.. So even if you request 24 bytes from new, you would waste 32 bytes (if we are 32 bytes aligned).

Moreover, U++ allocator today greatly benefits from the fact that alignment requirement is only 8 bytes. It would be possible to overcome this, but only at the price of quite a lot of wasted memory (or speed).

Still undecided. But if I consider that the issue does not stop at 32 bytes...

Mirek

Subject: Re: SSE2(/AVX) and alignment issues
Posted by tojocky on Sun, 30 Jan 2011 10:34:25 GMT
View Forum Message <> Reply to Message

mirek wrote on Sat, 29 January 2011 21:29tojocky wrote on Sat, 29 January 2011 03:23mirek wrote on Sat, 29 January 2011 01:03

This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

Mirek,

Can you give us an example of "new" realization and "allocator" realization?

```
struct AvxSomething {
  _m256 x;
};

Array<AvxSomething> foo;

foo.Add(new AvxSomthing); // not supported in option2. Actually, not even supported by any compiler today

foo.Add<AvxSomething>(); // supported in both options
```

Option2 could also support e.g.:

```
foo = New<AvxSomething>();
foo = new (Aligned<AvxSomething>) AvxSomething;
foo = NEW(AvxSomething);

Delete(foo);
```

(The crucial problem is that we need to know the type in new/delete).

Quote:
 Why you are not agree with new realization (option 1)?

Because it wastes memory. It means that every single block allocated by 'new' must be a multiple 16 bytes allocated for SSE2.

Then 32 bytes for AVX and AVX is supposed to grow in width, so it can be easily 64 bytes etc.. So even if you request 24 bytes from new, you would waste 32 bytes (if we are 32 bytes aligned).

Moreover, U++ allocator today greatly benefits from the fact that alignment requirement is only 8 bytes. It would be possible to overcome this, but only at the price of quite a lot of wasted memory (or speed).

Still undecided. But if I consider that the issue does not stop at 32 bytes...

Mirek

What about to implement something like?
```
void* operator new( size_t size, size_t alignment ){
    return __aligned_malloc( size, alignment );
}
```
and in code:
```
AlignedData* pData = new( 16 ) AlignedData;
```
or
```
AlignedData* pData = new( 32 ) AlignedData;or
AlignedData* pData = new( 64 ) AlignedData;
```
or
```
AlignedData* pData = new( 128 ) AlignedData;
```
?

User need to know when he uses sse2/3/4 data

First option I saw in the source code by this link. I do now agree with this because it is waste of space and speed.

How do you want to implement the second method?

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Sun, 30 Jan 2011 10:51:30 GMT
View Forum Message <> Reply to Message

tojocky wrote on Sun, 30 January 2011 05:34mirek wrote on Sat, 29 January 2011 21:29tojocky wrote on Sat, 29 January 2011 03:23mirek wrote on Sat, 29 January 2011 01:03

This is not a question. The question is whether _regular_ 'new' should return 16-byte aligned values or not. (And later, with AVX, 32, then maybe in 4 more years 64 etc...)

As long as we agree that allocating SSE2 stuff with 'new' is not a regular thing, we are at option 2..

Mirek

Mirek,

Can you give us an example of "new" realization and "allocator" realization?

```
struct AvxSomething {
  _m256 x;
};

Array<AvxSomething> foo;

foo.Add(new AvxSomthing); // not supported in option2. Actually, not even supported by any compiler today

foo.Add<AvxSomething>(); // supported in both options
```

Option2 could also support e.g.:

```
foo = New<AvxSomething>();
foo = new (Aligned<AvxSomething>) AvxSomething;
foo = NEW(AvxSomething);

Delete(foo);
```

(The crucial problem is that we need to know the type in new/delete).

Quote:
 Why you are not agree with new realization (option 1)?

Because it wastes memory. It means that every single block allocated by 'new' must be a multiple 16 bytes allocated for SSE2.

Then 32 bytes for AVX and AVX is supposed to grow in width, so it can be easily 64 bytes etc.. So even if you request 24 bytes from new, you would waste 32 bytes (if we are 32 bytes aligned).

Moreover, U++ allocator today greatly benefits from the fact that alignment requirement is only 8 bytes. It would be possible to overcome this, but only at the price of quite a lot of wasted memory (or speed).

Still undecided. But if I consider that the issue does not stop at 32 bytes...

Mirek

What about to implement something like?

```
void* operator new( size_t size, size_t alignment ){
    return __aligned_malloc( size, alignment );
}
```

and in code:
AlignedData* pData = new(16) AlignedData;
or
AlignedData* pData = new(32) AlignedData;or
AlignedData* pData = new(64) AlignedData;
or
AlignedData* pData = new(128) AlignedData;
?

User need to know when he uses sse2/3/4 data

This is still "option2".

I guess that the key difference is that

a) you cannot use regular 'new' for objects with special alignment
b) you cannot use regular 'delete' for objects with special alignment....

If you would want a) and b) work, you would definitely need ALL allocations to be aligned to highest possible alignment value.

That said, I guess that my suggestion of

New<FooClass>()

is somewhat superior, as it detects the alignment automatically...

Quote:
How do you want to implement the second method?

Well, that part is actually pretty simple:) Just if alignment >8, allocate more memory (add alignment + sizeof(void *)) and align, but void * before the aligned block to point back to allocated block.

Mirek

Subject: Re: SSE2(/AVX) and alignment issues
Posted by tojocky on Sun, 30 Jan 2011 15:52:29 GMT
View Forum Message <> Reply to Message

Looking in boost code:
class X{
public:

```
    explicit X(int n): n_(n){
    }

    void * operator new(std::size_t){
        return std::allocator<X>().allocate(1, static_cast<X*>(0));
    }

    void operator delete(void * p){
        std::allocator<X>().deallocate(static_cast<X*>(p), 1);
    }

private:

    X(X const &);
    X & operator=(X const &);

    int n_;
};

or
class Y{
public:

    explicit Y(int n): n_(n){
    }

    void * operator new(std::size_t n){
        return boost::detail::quick_allocator<Y>::alloc(n);
    }

    void operator delete(void * p, std::size_t n){
        boost::detail::quick_allocator<Y>::dealloc(p, n);
    }

private:

    Y(Y const &);
    Y & operator=(Y const &);

    int n_;
};
```

where something in the memory alocator we can get alignmet of type by:
for Codegear: alignof(T)
for GCC: __alignof__(T)
for MSC: __alignof(T)

according by IBM link and boost source code.

According by boost source code in file intrinsics.hpp MSC __alignof(T) fails when used with /Zp property. We need to be care of.

In this case we can easily use sse2/3/4/...:

Y y_v = new Y(1);

Is possible to implement a tool that can be integrated in the "operator new" of the classes with sse2/3/4 types properties?

Sorry, if it is a stupid question. I have not experience with sse, but i'me very interested to speed up the program by using sse/2/3/4.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Sun, 30 Jan 2011 17:24:35 GMT
View Forum Message <> Reply to Message

tojocky wrote on Sun, 30 January 2011 10:52Looking in boost code:

```
class X{
public:

   explicit X(int n): n_(n){
   }

   void * operator new(std::size_t){
      return std::allocator<X>().allocate(1, static_cast<X*>(0));
   }

   void operator delete(void * p){
      std::allocator<X>().deallocate(static_cast<X*>(p), 1);
   }

private:

   X(X const &);
   X & operator=(X const &);

   int n_;
};
```

or

```
class Y{
public:
```

```
    explicit Y(int n): n_(n){
    }

    void * operator new(std::size_t n){
        return boost::detail::quick_allocator<Y>::alloc(n);
    }

    void operator delete(void * p, std::size_t n){
        boost::detail::quick_allocator<Y>::dealloc(p, n);
    }

private:

    Y(Y const &);
    Y & operator=(Y const &);

    int n_;
};
```

where something in the memory alocator we can get alignmet of type by:
for Codegear: alignof(T)
for GCC: __alignof__(T)
for MSC: __alignof(T)

according by IBM link and boost source code.

According by boost source code in file intrinsics.hpp MSC __alignof(T) fails when used with /Zp property. We need to be care of.

In this case we can easily use sse2/3/4/...:

```
Y y_v = new Y(1);
```

Is possible to implement a tool that can be integrated in the "operator new" of the classes with sse2/3/4 types properties?

Sorry, if it is a stupid question. I have not experience with sse, but i'me very interested to speed up the program by using sse/2/3/4.

```
struct Foo {
   int bar;
   Y y;
};
```

and we are back where we were...

Anyway, deeper research has revealed that all this is somewhat obsolete. Where I am heading now is larger vectors of values that are fully encapsulated in some object (which can keep proper alignment) and using the most advanced ISA available...

Subject: Re: SSE2(/AVX) and alignment issues
Posted by Novo on Sun, 30 Jan 2011 18:35:52 GMT
View Forum Message <> Reply to Message

mirek wrote on Sun, 30 January 2011 12:24
Anyway, deeper research has revealed that all this is somewhat obsolete. Where I am heading now is larger vectors of values that are fully encapsulated in some object (which can keep proper alignment) and using the most advanced ISA available...

Could you please explain this in more details?
As far as I understand you are going to use CPU dispatching.

TIA

Subject: Re: SSE2(/AVX) and alignment issues
Posted by mirek on Sun, 30 Jan 2011 19:08:45 GMT
View Forum Message <> Reply to Message

Novo wrote on Sun, 30 January 2011 13:35mirek wrote on Sun, 30 January 2011 12:24
Anyway, deeper research has revealed that all this is somewhat obsolete. Where I am heading now is larger vectors of values that are fully encapsulated in some object (which can keep proper alignment) and using the most advanced ISA available...

Could you please explain this in more details?
As far as I understand you are going to use CPU dispatching.

TIA

Well, first of all, all of this is so far purely theoretical.

Anyway, I think the right idea is to emulate "vector processor", define float/double vector classes and operations on them.

I mean something like

DoubleVector x(200), y(200);
double a;

x = a * x + y;

and then, in implementation, use SSE2 or AVX or whatever to speed things up...

At this point, allocation is internal bussines of DoubleVector and alignment does not cause any problems anymore.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by Novo on Sun, 30 Jan 2011 19:50:48 GMT
View Forum Message <> Reply to Message

I agree with that. This is basically an answer to my question "what is SSE2 object?"

UPP just needs a vectorized library for vectors, matrices, and strings, which are vectors. And also generic vectorized algorithms.

Subject: Re: SSE2(/AVX) and alignment issues
Posted by dolik.rce on Sun, 30 Jan 2011 20:48:55 GMT
View Forum Message <> Reply to Message

mirek wrote on Sun, 30 January 2011 20:08I mean something like

DoubleVector x(200), y(200);
double a;

x = a * x + y;

and then, in implementation, use SSE2 or AVX or whatever to speed things up...

At this point, allocation is internal bussines of DoubleVector and alignment does not cause any problems anymore.
Great, that looks cool and makes a lot of sense. I actually use classes like this (no optimzation) just to make some computations code readable. If it would be SSE/AVX optimized, it would definitely give great performance boost to some of my apps.

So I definitelly vote for this option

Honza

Subject: Re: SSE2(/AVX) and alignment issues
Posted by tojocky on Sun, 30 Jan 2011 21:03:35 GMT
View Forum Message <> Reply to Message

mirek wrote on Sun, 30 January 2011 19:24

```
struct Foo {
  int bar;
  Y y;
};
```

and we are back where we were...

Anyway, deeper research has revealed that all this is somewhat obsolete. Where I am heading now is larger vectors of values that are fully encapsulated in some object (which can keep proper alignment) and using the most advanced ISA available...

You are right. But ObjectData *p = new ObjectData; and delete (ObjectData*)p; remain. Or I'm wrong?