

Re: A taxonomy of types

Source: <http://newsgroups.derkeiler.com/Archive/Comp/comp.lang.misc/2009-02/msg00035.html>

- *From:* "cr88192" <cr88192@xxxxxxxxxxxx>
 - *Date:* Fri, 6 Feb 2009 13:08:48 +1000
-

[original got lost...]

"Rod Pemberton" <do_not_have@xxxxxxxxxxxx> wrote in message
[news:gmejmi\\$5cm\\$1@xxxxxxxxxxxx](news:gmejmi$5cm$1@xxxxxxxxxxxx)

"cr88192" <cr88192@xxxxxxxxxxxx> wrote in message
[news:gmea2f\\$5np\\$1@xxxxxxxxxxxxxxxxxxxxxxxx](news:gmea2f$5np$1@xxxxxxxxxxxxxxxxxxxxxxxx)

(Correction: C doesn't have arrays. C has the offset operator.
C has
array
declarations, which are designed to work with the offset
operator...)

theoretically, and in terms of the functioning...

however, to actually compile the code correctly (AKA: following correct
array semantics, ...)

How can there be "correct array semantics" when no arrays exist in C? Did
you mean "following correct offset operator semantics"... ?

go ask the people in comp.std.c...
I took issue with this at first, but it is an issue...

the offset operator works against pointers or arrays, but pointers and
arrays are different.
more so, it is possible to create pointers-to-arrays which have different
semantics than both a plain pointer and a plain array.

```
int foo[32][16];
```

```
i=(&foo)[0][9];
```

Re: A taxonomy of types

which is expected to be equivalent to:
`i=foo[0][9];`

apparently, these semantics differ, where C99 and C95 have these semantics, but C90 and earlier did not.

one actually needs to have explicit array types, which are considered distinct from the pointer-based types.

IMO, both parts of that would be in error. Explicit array types don't exist in C. What does exist is based exclusively on pointer type. That pointer is hidden from the user an implementation specific according to the spec. That is for computing platforms that have problems using a generic pointer to effective an implement array type. For most systems, it's just a pointer.

I think your understanding of the standard is in error.
go check with the comp.std.c people...

now, if one were to say "as far as the representation goes, an array is no different than a pointer, nor is a pointer to an array distinct from the array", this would be correct, however, the typesystem thinks differently...

this is too simplistic...

consider a user types: 'foo->x'.

`foo->x` is equivalent to `(*foo).x`

dereference `foo`, get `x` using `x`'s offset from start of struct pointed to by `foo`

I'd think it'd be something like this where 'foo' and 'x' are 32-bit offsets

in assembly and 'foo' and 'x' is an 32-bit values in C, using `eax` as a temp:

```
mov eax, foo ; pointer or address foo to eax
mov eax, [eax] ; *foo (dereference) to eax
lea eax,[eax+x] ; *foo plus offset of x into struct to eax
mov eax, [eax] ; x to eax
```

Re: A taxonomy of types

can't be done, however, if knowledge of the type is not retained...
for example, you know that foo is a pointer, or a pointer to a struct, but
without a specific identification of the structure used for 'foo'.

or 'foo->bar(3, 4);'

foo->bar(3,4) is equivalent to (*foo).bar(3,4)

Same thing except you need to call a function...

these are lexical adjustments, but don't don't resolve the typesystem
issue...

if the representation can't deal with more than base-types, the above
expressions can't be compiled...

I don't understand your claim. Why? Why can't they be compiled? I mean,
I
just provided one possible implementation of the first situation in
assembly
right off the top of my head... As I see it, they just need to be broken
down into equivalent smaller sequences that reduce to pointers or
addresses
and offsets.

they can't be compiled if the compiler does not know how to compile them,
which will happen if one only has base-types, since past the first few steps
there will be a lapse in the type information, and it is no longer possible
to compile the expression (unless one falls back to being untyped, but this
is not really sufficient either for structures or function calls, since a
struct needs to know its fields, and a function call needs to know its
argument types for correct argument type conversions...).

s/t:
signed/unsigned
16-bit

Re: A taxonomy of types

short;
i/j:
signed/unsigned
32-bit int;
l/m:
signed/unsigned
64-bit long;
n/o:
signed/unsigned
128-bit int;

Hmm, I guess it's time for me to go back to the C spec., since I thought for sure that "int" had to be a "char", "short", "long", or "long long", not

a

distinct type by itself as you've done... (Am I wrong?)

int and long may have the same size (at least on x86, or in MSVC on

x64,

but

not in Linux on x86-64, PPC64, ...),

Well, technically, I think you comply with the wording of ISO C99... I don't have a copy of ANSI C available here to check. But, I think that doesn't comply with K&R C:

In K&R C in A.4.2 Basic Types:

"Besides the char types, up to three sizes of integer, declared short

Re: A taxonomy of types

int,

int, and long int, are available. Plain int objects have the natural

size

suggested by the host machine architecture; the other sizes are provided to meet special needs. Longer integers provide at least as much storage as shorter ones, but the implementation may make plain integers equivalent

to

either short integers, or long integers."

The critical point being an "int" is either "short" or "long", not

between

the two...

Also, K&R C in 2.2 Data Types and Sizes :

"int an integer, typically reflecting the natural size of integers on

the

host machine"

...

"The intent is that short and long should provide different lengths of integers where practical; int will normally be the natural size for a particular machine. short is often 16 bits long, and int either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the the restriction that shorts and ints are

at

Re: A taxonomy of types

least 16 bits, longs are at least 32 bits, and short is no longer than
int,
which is no longer than long."

yes.

With what are you agreeing? All of it? Then, you understand the last sentence quoted of 4.2 gives you a choice between implementing int as a short or long... If not, see my reply to Verlinde a post above...

by your definitions, many modern compilers, including GCC and LLVM, would be incorrect...

I agree with the text, as it is stated literally... but the text does not uphold your interpretation, as I see it.

not exactly...

"the same size as" does not mean "the same as"...

I don't see that wording...

try restructuring thinking in terms of tables (x86-case):

```
table Type(*Name, Bits, ...)  
char 8  
short 16  
int 32  
long 32  
long-long 64
```

```
table Bits(Size)  
8  
16  
32  
64
```

"short is often 16 bits long, and int either 16 or 32 bits"

Re: A taxonomy of types

```
'select from Type where name=short' 16  
'select from Type where name=int' 32
```

(now, switching to a lighter notation as representing my thoughts as queries would be too long and horrid...).

```
'<name>.<field>' -> 'select <field> from Type where name=<name>'
```

or:

```
short.bits==16  
(int.bits==16) || (int.bits==32)
```

"subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long"

```
short.bits>=16  
int.bits>=16  
long.bits>=32  
int.bits>=short.bits  
long.bits>=int.bits
```

now, your claim:

```
(int.bits==short.bits) || (int.bits==long.bits)  
can't be inferred from the text in the question.
```

```
and, for x86-64/PPC64  
table Type(*Name, Bits, ...)  
char 8  
short 16  
int 32  
long 64  
long-long 64
```

now, your assertion breaks, but the prior constraints hold...

in fact, the only way in which it can be asserted to be held from the text, is if we assume that the type table links types:

```
table Type(*Name, NameB, ...)  
char char  
short short  
int long  
long long  
long-long long-long
```

which is, as I see it, incorrect...

Re: A taxonomy of types

Rod Pemberton