

Re: opinions on logical OR variation

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2004-09/2889.html

From: Dave (*recneps.w.divad_at_elcaro.moc*)

Date: 09/28/04

Date: Tue, 28 Sep 2004 16:01:52 +0100

j0mbolar wrote:

- > *I would like to see what the majority of people prefer*
- > *when it comes to elegance, clarity, etc.*
- >
- > *I'll present the problem first based on*
- > *having a product, and one product only, at a*
- > *given time and then for each product,*
- > *when one is chosen, we call a generic*
- > *product function.*
- >
- > *So in the case that we have no product,*
- > *we don't want to do anything but when we do*
- > *we want to rely upon a single function that*
- > *is applied to all products.*
- >
- > *#define APPLE (1<<0)*
- > *#define ORANGE (1<<1)*
- > *#define PEAR (1<<2)*
- > *#define PLUMB (1<<3)*

I'd start by trying to figure out what you are really trying to do.

This kind of approach is generally used when something can have multiple characteristics; e.g. a MessageBox in Windows can have an OK button or a CANCEL button, so you can do MB_OK | MB_CANCEL and there are other varieties; MB_OK and MB_CANCEL are powers of 2 and they can be OR-ed together.

However, presumably a product cannot be both an apple and an orange, unless this software is about bizarre genetic experimentation, so why do you need the types of product to be combinable in this way?

```
#define APPLE 1
#define ORANGE 2
#define PEAR 3
#define PLUM 4
```

would be clearer; an enum would be better still.

```
>
> int main(void)
> {
> int product = APPLE; /* start with an apple */
>
> do_something(product);
>
> return 0;
> }
>
> variations follow:
> [1]
> void do_something(int product)
> {
> int items = (APPLE | ORANGE | PEAR | PLUMB);
>
> if(product & items)
> generic_func();
>
> }
```

Well, this could work, but it implies that a product can be both an apple and an orange. If this were the case I would expect the program to throw an error, not continue working. The structure is fine if it is appropriate.

```
>
> [2]
> void do_something(int product)
> {
> if((product == APPLE) || (product == ORANGE)
> || (product == PEAR) || (product == PLUMB))
> generic_func();
>
> }
```

I don't see what's wrong with this. Presumably there are possible values for product that you don't want do_something to act on, so either you do this or you say "if (product != SHOEBOX)"... If do_something acts on apples, oranges, pears and plums, and ignores anything else, then there's nothing wrong with this. If the attributes are combinable, then this will not work for product=APPLE|ORANGE.

[2] is not the same as [1]; it is functionally different. [2] works for a product that is exactly equal to only one of the attributes. [1] works if the product has several attributes; (APPLE | PLUM) & (APPLE | ORANGE | PEAR | PLUM) is TRUE so generic_func would be called for an apple/plum hybrid by [1] but not by [2].

```
>
> [3]
```

```
> void do_something(int product)
> {
> switch(product) {
> case APPLE:
> case ORANGE:
> case PEAR:
> case PLUMB: generic_func();
> break;
> }
> }
>
> or perhaps you would go about it in a completely
> different way? and if so, which way?
```

Again this won't work for product=APPLE|ORANGE. Whether you use this approach or [2] is down to personal preference IMO. Again it differs from [1] so if [1] is what you want to do then [3] is not an appropriate substitute. However if [2]/[3] is what you want to do I'd say they're both clearer than [1] (i.e. the intent is clearer. One isn't left wondering about hybrid fruits and why generic_func would be called in such a situation).

```
>
>
> personally, I can't stand [2] and think it is vile.
```

If it's appropriate to the situation then you just have to get over thinking that particular code constructs are vile. Personally I have a much bigger problem with the fact that you're using powers of 2 for what appear to be mutually exclusive attributes and your misspelling of "plum." [2] does benefit from being extremely readable – we call generic_func if product is apple, orange, pear or plum; the code is effectively self-documenting.

Dave.