

Stepping Back

by Stephen C. Dewhurst

Most aspects of pointers and containers in C++ are straightforward. Pointers to constants and pointers to derived classes behave in a reasonable and intuitive manner with respect to conversion and behavior. Most uses of arrays and user-defined containers also behave in simple, reasonable ways. However, by taking a step backward and dealing with pointers to pointers, arrays of class objects, or polymorphic container element hierarchies, we encounter some rather unintuitive and attractively dangerous language features. This month's installment of Common Knowledge describes some of the more common problems and makes recommendations as to what can be done about them.

Pointers to Const, Pointers to Derived Classes

First, let's get some terminology straight. A "const pointer" is a pointer that is constant. There is no implication that what the pointer refers to is constant. True, the C++ standard library has a concept of `const_iterator`, which is a non-constant iterator into a sequence of constant elements, but that is a symptom of design-by-committee or some similar disease.

```
const char *pci; // pointer to const
char * const cpi = 0; // const pointer
char const *pci2; // pointer to const, a la Dan Saks1
const char * const cpci = 0; // const pointer to const
char *ip; // pointer
```

The standard permits conversions that "increase constness." For example, we can copy a pointer to non-const to a pointer to const. This allows us to, among other things, pass a pointer to non-constant character to `strcmp` or `strlen` even though those functions are declared to accept pointer to constant character. Intuitively, we understand that, in allowing a pointer to const to refer to non-constant data, we are not violating any constraints implied by the data's declaration; we also understand the reverse conversion is invalid because it would grant greater permission than the declaration of the data specifies.

```
int i = strlen( cpi ); // OK...
pci = ip; // OK...
ip = pci; // error!
```

Note that the language rules take the conservative point of view: it may actually be "OK," in the sense of not dumping core immediately, to modify the data referred to by a pointer to const if that data actually is not const, or if it is const but the particular platform does not assign const data to a read-only area of memory. However, the use of const is typically a statement of design intent as well as physical property. The language can be seen to enforce the intent of the designer.

We have similar behavior for pointers to derived classes. These conversions can be seen to be almost entirely in support of the designer's intent, because they enforce the notion of an is-a relationship without any attendant requirement of protecting a physical property. Therefore, we are allowed to convert a pointer to a derived class to a pointer to a public, but not to a protected or private, base class.²

```
class B { /*...*/ }, *bp;
class D1 : public B { /*...*/ }, *pd1;
class D2 : public B { /*...*/ }, *pd2;
bp = pd1; // OK, D1 is-a B
pd1 = bp; // error!
bp = pd2; // OK, D2 is-a B
pd1 = pd2; // error!
```

Pointers to Pointers

This happily simple state of conversions does not hold in the case of pointers to pointers. Consider an attempt to convert a pointer to a pointer to a char to a pointer to a pointer to a const char (that is, `char ** to const char **`).

```
char **ppc;
const char **ppcc = ppc; // error!
```

It looks harmless, but like many harmless-looking conversions, it opens the door to a subversion of the type system. (I would like the reader to note that I did not, at this point, attempt a sociological analogy.)

```
const T t = init;
T *pt;
const T **ppt = &pt; // error, fortunately
*ppt = &t; // put a const T address in a T *!
*pt = value; // trash t!
```

This compelling subject is treated in section 4.4 of the standard, under “Qualification Conversions” (technically, `const` and `volatile` are known as type-qualifiers, and the standard tends to refer to them as cv-qualifiers). There we find the following simple rules for determining convertibility:

A conversion can add cv-qualifiers at levels other than the first in multi-level pointers, subject to the following rules:

Two pointer types T1 and T2 are *similar* if there exists a type *T* and integer $n > 0$ such that:

T1 is $c_{v1,0}$ pointer to $c_{v1,1}$ pointer to \dots $c_{v1,n-1}$ pointer to $c_{v1,n} T$

and

T2 is $c_{v2,0}$ pointer to $c_{v2,1}$ pointer to \dots $c_{v2,n-1}$ pointer to $c_{v2,n} T$ where each $c_{vi,j}$ is `const`, `volatile`, `const volatile`, or nothing.

In other words, two pointers are similar if they have the same base type and have the same number of *'s. So, for example, the types `char * const **` and `const char ***const` are similar, but `int * const *` and `int ***` are not.

The n -tuple of *cv*-qualifiers after the first in a pointer type, e.g., $cv1,1, cv1,2, \dots, cv1,n$ in the pointer type T1, is called the *cv-qualification signature* of the pointer type. An expression of type T1 can be converted to type T2 if and only if the following conditions are satisfied:

— the pointer types are similar.

— for every $j > 0$, if `const` is in $cv1,j$ then `const` is in $cv2,j$, and similarly for `volatile`.

— if the $cv1,j$ and $cv2,j$ are different, then `const` is in every $cv2,k$ for $0 < k < j$.

Armed with these rules--and a little patience--we can determine the legality of pointer conversions such as the following:

```
int * * * const cnnn = 0;
    // n==3, signature == none, none, none
int * * const * ncnn = 0;
    // n==3, signature == const, none, none
int * const * * nncn = 0;
    // signature == none, const, none
int * const * const * nccn = 0;
    // signature == const, const, none
const int * * * nncn = 0;
    // signature == none, none, const

// examples of application of rules
ncnn = cnnn; // OK
nncn = cnnn; // error!
nccn = cnnn; // OK
ncnn = cnnn; // OK
nncn = cnnn; // error!
```

Very useful at cocktail parties when the conversation lags, but is familiarity with these rules of any practical use in production coding? Well, yes. Consider the following rather common situation.

```
extern char *namesOfPeople[];
for( const char **currentName = namesOfPeople; // error!
     *currentName; currentName++ ) //...
```

In my experience, the typical developer response to this error is to file a bug report with the compiler vendor, cast away the error, and dump core later on. As usual, the compiler is right and the developer is not.

We face a similar situation with pointers to pointers to derived classes.

```
D1 d1;
D1 *d1p = &d1; // OK
B **ppb1 = &d1p; // error, fortunately
D2 *d2p;
```

```
B **ppb2 = &d2p; // error, fortunately
*ppb2 = *ppb1; // now d2p points to a D1!
```

Look familiar? Just as the const-conversion property does not hold if one introduces another level of indirection, the same is the case of the is-a property. While a pointer to a derived class is a pointer to a public base, a pointer to a pointer to a derived class is not a pointer to a pointer to a public base. As with the analogous const example, the situation that results in an error initially looks rather contrived. However, it's easy to construct a situation where an error is produced as a cooperative effort between a bad interface design and an incorrect use of the interface.

```
void doBs( B *bs[], B *pb ) {
    for( int i = 0; bees[i]; ++i )
        if( somecondition( bees[i], pb ) )
            bees[i] = pb; // oops!
}
//...
extern D1 *array[];
D2 *aD2 = getMeAD2();
doBs( (B **)array, aD2 ); // another casted death wish...
```

Once again, the developer assumes the compiler is in error and circumvents the type system with a cast. Naughty.³ In this case, though, the designer of the function interface has a lot to answer for as well. A safer design would have employed a container that did not permit spoofing by cast, as an array does. We'll discuss this shortly.

Arrays of Classes

While we're on the subject of arrays, prudence suggests that we should also sound the standard warning about arrays of class types, especially of base class types.

```
void
apply( B array[], int length, void (*f)( B & ) ) {
    for( int i = 0; i < length; ++i )
        f( array[i] );
}
main() {
    D1 *dp = new D1[42];
    apply( dp, 42, somefunc ); // disaster!
}
```

The trouble is that the type of the formal argument to `apply` is “pointer to B,” not “array of B.” As far as the compiler is concerned, we're initializing a `B *` with a `D1 *`. This is legal, since a `D1` is-a `B`. However, an array of `D1` is not an array of `B`, and the code will fail badly when we attempt pointer arithmetic using `B` offsets on an array of `D1` objects.

Incremental attempts to make the array behave sensibly fail. If the base class `B` were declared to be abstract (a good idea in general), that would prevent any arrays of `B` from being created, but the `apply` function would still be legal (if incorrect), since it deals with pointers to `B` rather than `B` objects.

Declaring the formal argument to be a reference to an array (as in `B (&array) [42]`) is effective, but is not practical as we must then fix the size of the array to a given bound, and can not pass a pointer (to an allocated array, for instance) as an actual argument. Arrays of base classes are just plain inadvisable, and arrays of classes in general have to be watched closely.

Real Containers

I can tell by all the smug looks out there that many of the readers of this article know the usual mechanism for avoiding these problems. An array is a thinly disguised pointer, and a pointer to a pointer is usually a thinly disguised array of pointers. Why not get rid of the thin disguises and use “real” containers instead of arrays?

The STL containers are the default mechanism of choice for C++ programmers. In the examples above, a simple substitution of a `vector<const char *>` or `vector<B *>` would have circumvented, or at least made obvious, any conversion problems without noticeable effect on efficiency. However, STL containers do not answer all needs, in part because their strengths also imply some limitations. One of the nice things about the STL containers is that, because they are implemented with templates, most of the decisions about their structure and behavior are made at compile time. This results in small and efficient implementations that are precisely tuned to the static context of their use.

However, all relevant information may not be present at compile time. For example, consider a simplified framework-oriented structure that supports the “open-closed principle” in that it may be modified and extended without recompilation of the framework.

```
template <class T>
class Container {
public:
    virtual ~Container();
    virtual Iter<T> *genIter() const = 0; // factory method
    virtual void insert( const T & ) = 0;
    //...
};

template <class T>
class Iter {
public:
    virtual ~Iter();
    virtual void reset() = 0;
    virtual void next() = 0;
    virtual bool done() const = 0;
    virtual T &get() const = 0;
};
```

```

template <class T>
void print( Container<T> &c ) {
    auto_ptr< Iter<T> > i( c.genIter() );
    for( i->reset(); !i->done(); i->next() )
        cout << i->get() << endl;
}

```

This snippet illustrates the use of runtime polymorphism in support of framework extension. The customizer of the framework derives substitutable container and iterator types and connects them with a Factory Method.⁴ For example, we could derive `List`, `Set`, or `Vector` containers and iterators and manipulate them with the framework code.

Non-Substitutable Polymorphism

It is fairly easy to design substitutable derived `Container` types. A `Set<T>` would then be substitutable for a `Container<T>`, and the usual conversions from `Set<T> *` to `Container<T> *` would hold.

However, there is an unfortunate and common tendency to assume that substitutability of container elements implies substitutability of the containers of these elements. We've already seen that this relationship does not hold for arrays of substitutable classes, or arrays of pointers to substitutable classes. The same warnings apply to user-defined containers of substitutable elements. Consider the following simple container hierarchy in support of a financial instrument pricing framework.

```

class Object
{ public: virtual ~Object(); };
class Instrument : public Object5
{ public: virtual double pv() const = 0; };
class Bond : public Instrument
{ public: double pv() const; };
class ObjectList {
public:
    void insert( Object * );
    Object *get();
    //...
};
class BondList : public ObjectList { // bad idea!!!
public:
    void insert( Bond *b )
        { ObjectList::insert( b ); }
    Bond *get()
        { return static_cast<Bond *>(ObjectList::get()); }
}

```

```
    //...
};
double
bondPortfolioPV( BondList &bonds ) {
    double sumpv = 0.0;
    for( each bond in list ) {
        Bond *b = current bond;
        sumpv += b->pv();
    }
    return sumpv;
}
```

Now, there is nothing wrong with implementing a list of `Bond` pointers with a list of `Object` pointers (although a better design would have employed a list of `void *`, and drop-kicked the entire notion of an `Object` class into the bit bucket). The error is in using public inheritance to force an is-a relationship on types that are not substitutable. In essence, we “stepped back” when we wrapped access to our substitutable pointers in a container, rendering them unsubstitutable. However, unlike the case in which we have a pointer to a pointer (or an array of pointers) the compiler can no longer warn us of our folly.

```
class UnderpaidMinion : public Object {
public:
    virtual double pay()
        { /* deposit $1M in minion's account */ }
};

void sneaky( ObjectList &list )
    { list.insert( new UnderpaidMinion ); }

void victimize() {
    BondList &blist = getBondList();
    sneaky( blist );
    bondPortfolioPV( blist ); //done!
}
```

Here we have managed to substitute one sibling class object for another; we’ve plugged in an `UnderpaidMinion` where the pricing framework is expecting a `Bond`. Under most environments, the result will be an invocation of `UnderpaidMinion::pay` rather than `Bond::pv`; an undetectable runtime type error. Just as an array of substitutable derived objects is not substitutable for an array of base objects or pointers, a user-defined container of substitutable derived objects or pointers is not substitutable for a user-defined container of base objects or pointers. Container substitutability, if present at all, should focus on the structure of the container, and not of the contained elements.

You Shouldn't, Because They Will

I am reminded at this point of that puerile SUV ad campaign that is current at the time this article is being written--“Not that you would, but you could”--in which various dangerous and inadvisable uses of the vehicle are showcased. In my experience on the highway in my undersized LEV, many SUV drivers not only can, but do. In most of the real-world examples above, the damage was accomplished by a cooperative effort of the designer and user of the feature. If one wanted to stretch a point(er), one could say the producer of a dangerous facility should share in the blame for its misuse. As designers, it is our responsibility to produce safe and intuitive interfaces, and advertise their correct use. As we have seen above, that implies paying close attention to pointer to pointer conversions and container substitutability. Because, as any experienced designer knows, users of our designs not only can, they will.

end of article

¹ This is a reference to an ongoing discussion I've been having with (CUJ Editorial Board Member) Dan Saks over declaration-specifier ordering. I claim that `const T *` is the least confusing way to write “pointer to constant T,” whereas Dan claims the preferred form is `T const *`. On his side of the argument, Dan has performed field studies in production environments, taken statistics, and generally proceeded in a scientific manner. On my side, I'm right.

² Note that there is a very heavy responsibility on the part of the designer in this case to ensure that the derived class is substitutable for each of its public base classes.

³ For more judgmental observations about casts and the people who employ them, see S.C. Dewhurst, “A Question of Respect,” *C/C++ Users Journal*, 19 (4), April 2001.

⁴ Gamma, et al., *Design Patterns*, Addison Wesley, 1995, p. 107.

⁵ This is a “cosmic hierarchy” design. For an explanation as to why this is evil, see S.C. Dewhurst, “Don't Ask, Don't Tell,” *C++ Report*, 12 (5), May 2000.