# 34. Index Tables

**Difficulty: 5**

*Index tables are a genuinely useful idiom and a technique that's worth being aware of. But how can we implement the technique effectively… nay, even better than that, exceptionally?*

### JG Question

1. Who benefits from clear, understandable code?

### Guru Question

2. The following code presents an interesting and genuinely useful idiom for creating index tables into existing containers. For a more detailed explanation, see the original article [Hicks00].

   Critique this code and identify:

   a) Mechanical errors, such as invalid syntax or nonportable conventions.

   b) Stylistic improvements that would improve code clarity, reusability, and maintainability.

```
// program sort_idxtbl(...) to make a permuted array of indices
#include <vector>
#include <algorith>

template <class RAIter>
struct sort_idxtbl_pair
{
  RAIter it;
  int i;

  bool operator<( const sort_idxtbl_pair& s )
    { return (*it) < (*(s.it)); }

  void set( const RAIter& _it, int _i ) { it=_it; i=_i; }

  sort_idxtbl_pair() {}
};

template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
  int iDst = last-first;
  typedef std::vector< sort_idxtbl_pair<RAIter> > V;
  V v( iDst );
```

```
  int i=0;
  RAIter it = first;
  V::iterator vit = v.begin();
  for( i=0; it<last; it++, vit++, i++ )
    (*vit).set(it,i);

  std::sort(v.begin(), v.end());

  int *pi = pidxtbl;
  vit = v.begin();
  for( ; vit<v.end(); pi++, vit++ )
    *pi = (*vit).i;
}

main()
{
  int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

  cout << "##################" << endl;
  std::vector<int> vecai(ai, ai+10);
  int aidxtbl[10];
  sort_idxtbl(vecai.begin(), vecai.end(), aidxtbl);

  for (int i=0; i<10; i++)
  cout << "i=" << i
       << ", aidxtbl[i]=" << aidxtbl[i]
       << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
       << endl;
  cout << "##################" << endl;
}
```

# Solution

## Clarity: A Short Sermon

***1. Who benefits from clear, understandable code?***

In short, just about everyone benefits.

First, clear code is easier to follow while debugging and, for that matter, is less likely to have as many bugs in the first place, so writing clean code makes your own life easier even in the very short term. (For a case in point, see the discussion surrounding Example 27-2 in Item 27.) Further, when you return to the code a month or a year later—as you surely will if the code is any good and is actually being used—it's

much easier to pick it up again and understand what's going on. Most programmers find keeping full details of code in their heads difficult for even a few weeks, especially after having moved on to other work; after a few months or even a few years, it's too easy to go back to your own code and imagine it was written by a stranger—albeit a stranger who curiously happened to follow your personal coding style.

But enough about selfishness. Let's turn to altruism: Those who have to maintain your code also benefit from clarity and readability. After all, to maintain code well one must first *grok* the code. "To grok," as coined by Robert Heinlein, means to comprehend deeply and fully; in this case, that includes understanding the internal workings of the code itself, as well as its side effects and interactions with other subsystems. It is altogether too easy to introduce new errors when changing code one does not fully understand. Code that is clear and understandable is easier to grok, and therefore, fixes to such code become less fragile, less risky, less likely to have unintended side effects.

Most important, however, your end users benefit from clear and understandable code for all these reasons: Such code is likely to have had fewer initial bugs in the first place, and it's likely to have been maintained more correctly without as many new bugs being introduced.

➢ **Guideline:** **By default, prefer to write for clarity and correctness first.**

### Dissecting Index Tables

2. ***The following code presents an interesting and genuinely useful idiom for creating index tables into existing containers. For a more detailed explanation, see the original article [Hicks00].***

   ***Critique this code and identify:***

   ***a) Mechanical errors, such as invalid syntax or nonportable conventions.***

   ***b) Stylistic improvements that would improve code clarity, reusability, and maintainability.***

Again, let me repeat that which bears repeating: This code presents an interesting and genuinely useful idiom. I've frequently found it necessary to access the same container in different ways, such as using different sort orders. For this reason it can be useful indeed to have one principal container that holds the data (for example, a **vector<Employee>**) and secondary containers *of iterators* into the main container that support variant access methods (for example, a **set<vector<Employee>::iterator, Funct>** where **Funct** is a functor that compares **Employee** objects indirectly,

yielding a different ordering than the order in which the objects are physically stored in the **vector**).

Having said that, style matters too. The original author has kindly allowed me to use his code as a case in point, and I'm not trying to pick on him here; I'm just adopting the technique, pioneered long ago by such luminaries as P. J. Plauger, of expounding coding style guidelines via the dissection and critique of published code. I've critiqued other published material before and have had other people critique my own, and I'm positive that further dissections will no doubt follow.

Having said *all* that, let's see what we might be able to improve in this particular piece of code.

## Correcting Mechanical Errors

***a) Mechanical errors, such as invalid syntax or nonportable conventions.***

The first area for constructive criticism is mechanical errors in the code, which on most platforms won't compile as shown.

```
#include <algorith>
```

*1. Spell standard headers correctly.* Here the header **<algorithm>** is misspelled as **<algorith>**. My first guess was that this is probably an artifact of an 8-character file system used to test the original code, but even my old version of VC++ on an old version of Windows (based on the 8.3 filename system) rejected this code. Anyway, it's not standard, and even on hobbled file systems the compiler itself is required to support any standard long header names, even if it silently maps it onto a shorter filename (or onto no file at all).

Next, consider:

```
main()
```

*2. Define **main** correctly.* This unadorned signature for **main** has never been standard C++ [C++98], although it is a conforming extension as long as the compiler warns about it. It used to be valid in pre-1999 C, which had an implicit **int** rule, but it's nonstandard in both C++ (which never had implicit **int**) and C99 [C99] (which as far as I can tell didn't merely deprecate implicit **int**, but actually removed it outright). In the C++ standard, see:

- §3.6.1/2: portable code must define **main** as either **int main()** or **int main( int, char*[] )**
- §7/7 footnote 78, and §7.1.5/2 footnote 80: implicit **int** banned
- Annex C (Compatibility), comment on 7.1.5/4: explicitly notes that bare **main()** is invalid C++, and must be written **int main()**

➢ **Guideline:** **Don't rely on implicit int; it's not standard-conforming portable C++. In particular "void main()" or just "main()" has never been standard C++, although many compilers still support them as conforming extensions.**

```
cout << "##################" << endl;
```

*3. Always* **#include** *the headers for the types whose definitions you need.* The program uses **cout** and **endl** but fails to **#include <iostream>**. Why did this probably work on the original developer's system? Because C++ standard headers can **#include** each other, but unlike C, C++ does not specify which standard headers **#include** which other standard headers.

In this case, the program does **#include <vector>** and **<algorithm>**, and on the original system it probably just so happened that one of those headers also happened to indirectly **#include <iostream>** too. That might work on the library implementation used to develop the original code, and it happens to work on mine too, but it's not portable and not good style.

*4. Follow the guidelines in Item 36 in* More Exceptional C++ *[Sutter02] about using namespaces.* Speaking of **cout** and **endl**, the program must also qualify them with **std::** or write **using std::cout; using std::endl;**. Unfortunately it's still common for authors to forget namespace scope qualifiers—I hasten to point out that this author did correctly scope **vector** and **sort**, which is good.

### Improving Style

***b) Stylistic improvements that would improve code clarity, reusability, and maintainability.***

Beyond the mechanical errors, there were several things I personally would have done differently in the code example. First, a couple of comments about the helper struct:

```
template <class RAIter>
struct sort_idxtbl_pair
{
  RAIter it;
  int i;

  bool operator<( const sort_idxtbl_pair& s )
    { return (*it) < (*(s.it)); }

  void set( const RAIter& _it, int _i ) { it=_it; i=_i; }

  sort_idxtbl_pair() {}
};
```

*5. Be const correct.* In particular, **sort_idxtbl_pair::operator<** doesn't modify ***this**, so it ought to be declared as a **const** member function.

➢ **Guideline:** **Practice const correctness.**

*6. Remove redundant code.* The program explicitly writes class **sort_idxtbl_pair**'s default constructor even though it's no different from the implicitly generated version. There doesn't seem to be much point to this. Also, as long as **sort_idxbl_pair** is a **struct** with public data, having a distinct **set** operation adds a little syntactic sugar but because it's called in only one place the minor extra complexity doesn't gain much.

➢ **Guideline:** **Avoid code duplication and redundancy.**

Next, we come to the core function, **sort_idxtbl**:

```
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
 int iDst = last-first;
 typedef std::vector< sort_idxtbl_pair<RAIter> > V;
 V v( iDst );

 int i=0;
 RAIter it = first;
 V::iterator vit = v.begin();
 for( i=0; it<last; it++, vit++, i++ )
  (*vit).set(it,i);

 std::sort(v.begin(), v.end());

 int *pi = pidxtbl;
 vit = v.begin();
 for( ; vit<v.end(); pi++, vit++ )
  *pi = (*vit).i;
}
```

*7. Choose meaningful and appropriate names.* In this case, **sort_idxtbl** is a misleading name because the function doesn't sort an index table… it creates one! On the other hand, the code gets good marks for using the template parameter name **RAIter** to indicate a random-access iterator; that's what's required in this version of the code, so naming the parameter to indicate this is a good reminder.

➢ **Guideline:** **Choose clear and meaningful names.**

*8. Be consistent.* In **sort_idxtbl**, sometimes variables are initialized (or set) in **for** loop initialization statements, and sometimes they aren't. This just makes things harder to read, at least for me. Your mileage may vary on this one.

*9. Remove gratuitous complexity.* This function adores gratuitous local variables! It contains three examples. First, the variable **iDst** is initialized to **last-first** and then used only once; why not just write **last-first** where it's used and get rid of clutter? Second, the **vector** iterator **vit** is created where a subscript was already available and could have been used just as well, and the code would have been clearer. Third, the local variable **it** gets initialized to the value of a function parameter, after which the function parameter is never used; my personal preference in that case is just to use the function parameter (even if you change its value—that's okay!) instead of introducing another name.

*10. Reuse Part 1: Reuse more of the standard library.* Now, the original program gets good marks for reusing **std::sort**—that's good. But why handcraft that final loop to perform a copy when **std::copy** does the same thing? Why reinvent a special-purpose **sort_idxtbl_pair** class when the only thing it has that **std::pair** doesn't is a comparison function? Besides being easier, reuse also makes our own code more readable. Humble thyself and reuse!

---

➢ **Guideline:** **Know about and (re)use the standard library's facilities wherever appropriate instead of hand-rolling your own.**

---

*11. Reuse Part 2: Kill two birds with one stone by making the implementation itself more reusable.* Of the original implementation, nothing is directly reusable other than the function itself. The helper **sort_idxtbl_pair** class is hardwired for its purpose and is not independently reusable.

*12. Reuse Part 3: Improve the signature.* The original signature

```
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
```

takes a bald **int*** pointer to the output area, which I generally try to avoid in favor of managed storage (say, a **vector**). But at the end of the day the user ought to be able to call **sort_idxtbl** and put the output into a plain array or a **vector** or anything else. Well, the requirement "be able to put the output into any container" simply cries out for an iterator, doesn't it? (See also Items 5 and 6.)

```
template< class RAIn, class Out >
void sort_idxtbl( RAIn first, RAIn last, Out result )
```

➢ **Guideline:** **Widen the reusability of your generic components by not hardcoding types that don't need to be hardcoded.**

*13. Reuse Part 4, or Prefer comparing iterators using `!=`:* When comparing iterators, always use `!=` (which works for all kinds of iterators) instead of `<` (which works only for random-access iterators), unless of course you really need to use `<` and only intend to support random-access iterators. The original program uses `<` to compare the iterators it's given to work on, which is fine for random-access iterators, which was the program's initial intent: to create indexes into **`vector`**s and arrays, both of which support random-access iteration. But there's no reason we might not want to do exactly the same thing for other kinds of containers, like **`list`**s and **`set`**s, that don't support random-access iteration, and the only reason the original code won't work for such containers is that it uses `<` instead of `!=` to compare iterators.

As Scott Meyers puts it eloquently in Item 32 of [Meyers96], "Program in the future tense." He elaborates:

> *Good software adapts well to change. It accommodates new features, it ports to new platforms, it adjusts to new demands, it handles new inputs. Software this flexible, this robust, and this reliable does not come about by accident. It is designed and implemented by programmers who conform to the constraints of today while keeping in mind the probable needs of tomorrow. This kind of software—software that accepts change gracefully—is written by people who program in the future tense.*

➢ **Guideline:** **Prefer to compare iterators using !=, not <.**

*14. Prefer preincrement unless you really need the old value.* Here, for the iterators, writing preincrement (`++i`) should habitually be preferred over writing postincrement (`i++`); see [Sutter00, Item 39]. True, that might not make a material difference in the original code because **`vector<T>::iterator`** might be a cheap-to-copy **`T*`** (although it might not be, particularly on checked STL implementations), but if we implement point 13 we may no longer be limited to just **`vector<T>::iterator`**s, and most other iterators are of class type—perhaps often still not too expensive to copy, but why introduce this possible inefficiency needlessly?

➢ **Guideline:** **Prefer to write preincrement rather than postincrement, unless you really need to use the previous value.**

That covers most of the important issues. There are other things we could critique but that I didn't consider important enough to call attention to here; for example,

production code should have comments that document each class's and function's purpose and semantics, but that doesn't apply to code accompanying magazine articles where the explanation is typically written in better English and in greater detail than code comments have any right to expect.

I'm deliberately not critiquing the mainline for style (as opposed to the mechanical errors already noted that would cause the mainline to fail to compile), because, after all, this particular mainline is only meant to be a demonstration harness to help readers of the magazine article see how the index table apparatus is meant to work, and it's the index table apparatus that's the intended focus.

### Summary

Let's preserve the original code's basic interface choice instead of straying far afield and proposing alternate design choices.[40] Limiting our critique just to correcting the code for mechanical errors and basic style, then, consider the three alternative improved versions below. Each has its own benefits, drawbacks, and style preferences as explained in the accompanying comments. What all three versions have in common is that they are clearer, more understandable, and more portable code—and that ought to count for something, in your company and in mine.

```
// An improved version of the code originally published in [Hicks00].
//
#include <vector>
#include <map>
#include <algorithm>

// Solution 1 does some basic cleanup but still preserves the general structure
// of the original's approach. We're down to 17 lines (even if you count "public:"
// and "private:" as lines), where the original had 23.
//
namespace Solution1 {
  template<class Iter>
  class sort_idxtbl_pair {
  public:
    void set( const Iter& it, int i ) { it_ = it; i_ = i; }

    bool operator<( const sort_idxtbl_pair& other ) const
      { return *it_ < *other.it_; }

    operator int() const { return i_; }
```

[40] The original author also reports separate feedback from another reader demonstrating another elegant, but substantially different, approach: He creates a containerlike object that wraps the original container, including its iterators, and allows iteration using the alternative ordering.

```
  private:
    Iter it_;
    int i_;
  };

  // This function is where most of the clarity savings came from; it has 5 lines,
  // where the original had 13. After each code line, I'll show the corresponding
  // original code for comparison. Prefer to write code that is clear and concise,
  // not unnecessarily complex or obscure!
  //
  template<class IterIn, class IterOut>
  void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
    std::vector<sort_idxtbl_pair<IterIn> > v(last-first);
      // int iDst = last-first;
      // typedef std::vector< sort_idxtbl_pair<RAIter> > V;
      // V v(iDst);

    for( int i=0; i < last-first; ++i )
      v[i].set( first+i, i );
      // int i=0;
      // RAIter it = first;
      // V::iterator vit = v.begin();
      // for (i=0; it<last; it++, vit++, i++)
      // (*vit).set(it,i);

    std::sort( v.begin(), v.end() );
      // std::sort(v.begin(), v.end());

    std::copy( v.begin(), v.end(), out );
      // int *pi = pidxtbl;
      // vit = v.begin();
      // for (; vit<v.end(); pi++, vit++)
      // *pi = (*vit).i;
  }
}

// Solution 2 uses a pair instead of reinventing a pair-like helper class. Now we're
// down to 13 lines, from the original 23. Of the 14 lines, 9 are purpose-specific,
// and 5 are directly reusable in other contexts.
//
namespace Solution2 {
  template<class T, class U>
  struct ComparePair1stDeref {
    bool operator()( const std::pair<T,U>& a, const std::pair<T,U>& b ) const
      { return *a.first < *b.first; }
  };
```

```
  template<class IterIn, class IterOut>
  void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
    std::vector< std::pair<IterIn,int> > s( last-first );
    for( int i=0; i < s.size(); ++i )
      s[i] = std::make_pair( first+i, i );

    std::sort( s.begin(), s.end(), ComparePair1stDeref<IterIn,int>() );

    for( int i=0; i < s.size(); ++i, ++out )
    *out = s[i].second;
  }
}

// Solution 3 just shows a couple of alternative details—it uses a map to avoid a
// separate sorting step, and it uses std::transform() instead of a handcrafted loop.
// Here we still have 15 lines, but more are reusable. This version uses more space
// overhead and probably more time overhead too, so I prefer Solution 2, but this
// is an example of finding alternative approaches to a problem.
//
namespace Solution3 {
  template<class T>
  struct CompareDeref {
    bool operator()( const T& a, const T& b ) const
      { return *a < *b; }
  };

  template<class T, class U>
  struct Pair2nd {
    const U& operator()( const std::pair<T,U>& a ) const { return a.second; }
  };

  template<class IterIn, class IterOut>
  void sort_idxtbl( IterIn first, IterIn last, IterOut out ) {
    std::multimap<IterIn, int, CompareDeref<IterIn> > v;
    for( int i=0; first != last; ++i, ++first )
      v.insert( std::make_pair( first, i ) );

    std::transform( v.begin(), v.end(), out, Pair2nd<IterIn const,int>() );
  }
}

// I left the test harness essentially unchanged, except to demonstrate putting
// the output in an output iterator (instead of necessarily an int*) and using the
// source array directly as a container.
//
#include <iostream>

int main() {
```

```
  int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

  std::cout << "##################" << std::endl;
  std::vector<int> aidxtbl( 10 );

  // use another namespace name to test a different solution
  Solution3::sort_idxtbl( ai, ai+10, aidxtbl.begin() );

  for( int i=0; i<10; ++i )
  std::cout << "i=" << i
            << ", aidxtbl[i]=" << aidxtbl[i]
            << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
            << std::endl;
  std::cout << "##################" << std::endl;
}
```

# 35. Generic Callbacks

**Difficulty: 5**

*Part of the allure of generic code is its usability and reusability in as many kinds of situations as reasonably possible. How can the simple facility presented in the cited article be stylistically improved, and how can it be made more useful than it is and really qualify as generic and widely usable code?*

### JG Question

1. What qualities are desirable in designing and writing generic facilities? Explain.

### Guru Question

2. The following code presents an interesting and genuinely useful idiom for wrapping callback functions. For a more detailed explanation, see the original article. [Kalev01]

   Critique this code and identify:

   a) Stylistic choices that could be improved to make the design better for more idiomatic C++ usage.

   b) Mechanical limitations that restrict the usefulness of the facility.

```
template < class T, void (T::*F)() >
class callback
{
public:
  callback(T& t) : object(t) {}          // assign actual object to T
  void execute() {(object.*F)();}        // launch callback function

private:
  T& object;
};
```

# Solution

## Generic Quality

***1. What qualities are desirable in designing and writing generic facilities? Explain.***

Generic code should above all be usable. That doesn't mean it has to include all options up to and including the kitchen sink. What it does mean is that generic code ought to make a reasonable and balanced effort to avoid at least three things:

*1. Avoid undue type restrictions.* For example, are you writing a generic container? Then it's perfectly reasonable to require that the contained type have, say, a copy constructor and a nonthrowing destructor. But what about a default constructor or an assignment operator? Many useful types that users might want to put into our container don't have a default constructor, and if our container uses it, then we've eliminated such a type from being used with our container. That's not very generic. (For a complete example, see Item 11 of *Exceptional C++* [Sutter00].)

*2. Avoid undue functional restrictions.* If you're writing a facility that does X and Y, what if some user wants to do Z, and Z isn't so much different from Y? Sometimes you'll want to make your facility flexible enough to support Z; sometimes you won't. Part of good generic design is choosing the ways and means by which your facility can be customized or extended. That this is important in generic design should hardly be a surprise, though, because the same principle applies to object-oriented class design.

Policy-based design is one of several important techniques that allow "pluggable" behavior with generic code. For examples of policy-based design, see any of several chapters in [Alexandrescu01]; the **SmartPtr** and Singleton chapters are a good place to start.

This leads to a related issue:

*3. Avoid unduly monolithic designs.* This important issue doesn't arise as directly in our style example under consideration below, and it deserves some dedicated consideration in its own right, hence it gets not only its own Item, but its own concluding miniseries of Items: see Items 37 through 40.

In these three points, you'll note the recurring word "undue." That means just what it says: Good judgment is needed when deciding where to draw the line between failing to be sufficiently generic (the "I'm sure nobody would want to use it with anything but **char**" syndrome) on the one hand and overengineering (the "what if someday someone wants to use this toaster-oven LED display routine to control the booster cutoff on an interplanetary spacecraft?" misguided fantasy) on the other.

## Dissecting Generic Callbacks

***2. The following code presents an interesting and genuinely useful idiom for wrapping callback functions. For a more detailed explanation, see the original article. [Kalev01]***

Here again is the code:

```
template < class T, void (T::*F)() >
class callback
{
public:
  callback(T& t) : object(t) {}          // assign actual object to T
  void execute() { (object.*F)(); }      // launch callback function

private:
  T& object;
};
```

Now, really, how many ways are there to go wrong in a simple class with just two one-liner member functions? Well, as it turns out, its extreme simplicity is part of the problem. This class template doesn't need to be heavyweight, not at all, but it could stand to be a little less lightweight.

## Improving Style

***Critique this code and identify:***

***a) Stylistic choices that could be improved to make the design better for more idiomatic C++ usage.***

How many did you spot? Here's what I came up with:

*4. The constructor should be* ***explicit****.* The author probably didn't mean to provide an implicit conversion from `T` to `callback<T>`. Well-behaved classes avoid creating the potential for such problems for their users. So what we really want is more like this:

```
explicit callback(T& t) : object(t) {}  // assign actual object to T
```

While we're already looking at this particular line, there's another stylistic issue that's not about the design *per se* but about the description:

> ➢ **Guideline:** **Prefer making constructors explicit unless you really intend to enable type conversions.**

*(Nit) The comment is wrong.* The word "assign" in the comment is incorrect and so somewhat misleading. More correctly, in the constructor, we're "binding" a `T` object

to the reference and by extension to the callback object. Also, after many rereadings I'm still not sure what the "to **T**" part means. So better still would be "bind actual object."

```
explicit callback(T& t) : object(t) {}   // bind actual object
```

But then, all that comment is saying is what the code already says, which is faintly ridiculous and a stellar example of a useless comment, so best of all would be:

```
explicit callback(T& t) : object(t) {}
```

*5. The **execute** function should be **const**.* The **execute** function isn't doing anything to the **callback<T>** object's state, after all! This is a "back to basics" issue: Const correctness might be an oldie, but it's a goodie. The value of const correctness has been known in C and C++ since at least the early 1980s, and that value didn't just evaporate when we clicked over to the new millennium and started writing lots of templates.

```
void execute() const { (object.*F)(); }          // launch callback function
```

> ➢ **Guideline:** **Be const correct.**

While we're already beating on the poor **execute** function, there's an arguably more serious idiomatic problem:

*6. (Idiom) And the **execute** function should be spelled **operator()**.* In C++, it's idiomatic to use the function-call operator for executing a function-style operation. Indeed, then the comment, already somewhat redundant, becomes completely so and can be removed without harm because now our code is already idiomatically commenting itself. To wit:

```
void operator()() const { (object.*F)(); }       // launch callback function
```

"But," you might be wondering, "if we provide the function-call operator, isn't this some kind of function object?" That's an excellent point, which leads us to observe that, as a function object, maybe callback instances ought to be adaptable too.

> ➢ **Guideline:** **Provide operator() for idiomatic function objects rather than providing a named execute function.**

*Pitfall: (Idiom) Should this callback be derived from **std::unary_function**?* See Item 36 in [Meyers01] for a more detailed discussion about adaptability and why it's a Good Thing in general. Alas, here, there are two excellent reasons why callback should not be derived from **std::unary_function**, at least not yet:

- It's not a unary function. It takes no parameter, and unary functions take a parameter. (No, **void** doesn't count.)
- Deriving from **std::unary_function** isn't going to be extensible anyway. Later on, we're going to see that callback perhaps ought to work with other kinds of function signatures too, and depending on the number of parameters involved, there might well be no standard base class to derive from. For example, if we supported callback functions with three parameters, we have no **std::ternary_function** to derive from.

Deriving from **std::unary_function** or **std::binary_function** is a convenient way to give callback a handful of important **typedef**s that binders and similar facilities often rely upon, but it matters only if you're going to use the function objects with those facilities. Because of the nature of these callbacks and how they're intended to be used, it's unlikely that this will be needed. (If in the future it turns out that they ought to be usable this way for the common one- and two-parameter cases, then the one- and two-parameter versions we'll mention later can be derived from **std::unary_function** and **std::binary_function**, respectively.)

## Correcting Mechanical Errors and Limitations

***b) Mechanical limitations that restrict the usefulness of the facility.***

*7. Consider making the callback function a normal parameter, not a template parameter.* Non-type template parameters are rare in part because there's rarely much benefit in so strictly fixing a type at compile time. That is, we could instead have:

```
template < class T >
class callback {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(t), f(func) {}        // bind actual object
  void operator()() const { (object.*f)(); }               // launch callback function

private:
  T& object;
  Func f;
};
```

Now the function to be used can vary at run-time, and it would be simple to add a member function that allowed the user to change the function that an existing callback object was bound to, something not possible in previous versions of the code.

> ➢ **Guideline:** **It's usually a good idea to prefer making non-type parameters into normal function parameters, unless they really need to be template parameters.**

*8. Enable containerization.* If a program wants to keep one callback object for later use, it's likely to want to keep more of them. What if it wants to put the callback objects into a container, such as a **vector** or a **list**? Currently that's not possible, because callback objects aren't assignable—they don't support **operator=**. Why not? Because they contain a reference, and once that reference is bound during construction, it can never be rebound to something else.

Pointers, however, have no such compunction and are quite happy to point at whatever you'd ask them to. In this case it's perfectly safe for callback instead to store a pointer, not a reference, to the object it's to be called on and then to use the default compiler-generated copy constructor and copy assignment operator:

```
template < class T >
class callback {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(&t), f(func) {}      // bind actual object
  void operator()() const { (object->*f)(); }             // launch callback function

private:
  T* object;
  Func f;
};
```

Now it's possible to have, for example, a **list< callback< Widget, &Widget::SomeFunc > >**.

> ➢ **Guideline:** **Prefer to make your objects compatible with containers. In particular, to be put into a standard container, an object must be assignable.**

"But wait," you might wonder at this point, "if I could have that kind of a **list**, why couldn't I have a list of arbitrary kinds of callbacks of various types, so that I can remember them all and go execute them all when I want to?" Indeed, you can, if you add a base class:

*9. Enable polymorphism: Provide a common base class for callback types.* If we want to let users have a **list<callbackbase*>** (or, better, a **list< shared_ptr<callbackbase> >**) we

can do it by providing just such a base class, which by default happens to do nothing in its **operator()**:

```
class callbackbase {
public:
  virtual void operator()() const { };
  virtual ~callbackbase() = 0;
};

callbackbase::~callbackbase() { }

template < class T >
class callback : public callbackbase {
public:
  typedef void (T::*Func)();

  callback(T& t, Func func) : object(&t), f(func) {}      // bind actual object
  void operator()() const { (object->*f)(); }             // launch callback function

private:
  T* object;
  Func f;
};
```

Now anyone who wants to can keep a **list<callbackbase*>** and polymorphically invoke **operator()** on its elements. Of course, a **list< shared_ptr<callback> >** would be even better; see [Sutter02b].

Note that adding a base class is a tradeoff, but only a small one: We've added the overhead of a second indirection, namely a virtual function call, when the callback is triggered through the base interface. But that overhead actually manifests only when you use the base interface. Code that doesn't need the base interface doesn't pay for it.

---

➢ **Guideline:** **Consider enabling polymorphism so that different instantiations of your class template can be used interchangeably, if that makes sense for your class template. If it does, do it by providing a common base class shared by all instantiations of the class template.**

---

*10. (Idiom, Tradeoff) There could be a helper **make_callback** function to aid in type deduction.* After a while, users may get tired of explicitly specifying template parameters for temporary objects:

```
list< callback< Widget > > l;
l.push_back( callback<Widget>( w, &Widget::SomeFunc ) );
```

Why write **Widget** twice? Doesn't the compiler know? Well, no, it doesn't, but we can help it to know in contexts where only a temporary object like this is needed. Instead, we could provide a helper so that they need only type:

```
list< callback< Widget > > l;
l.push_back( make_callback( w, &Widget::SomeFunc ) );
```

This **make_callback** works just like the standard **std::make_pair**. The missing **make_callback** helper should be a function template, because that's the only kind of template for which a compiler can deduce types. Here's what the helper looks like:

```
template<typename T >
callback<T> make_callback( T& t, void (T::*f) () ) {
  return callback<T>( t, f );
}
```

*11. (Tradeoff) Add support for other callback signatures.* I've left the biggest job for last. As the Bard might have put it, "There are more function signatures in heaven and earth, Horatio, than are dreamt of in your **void (T::*F) ()**!"

> ➢ **Guideline:** **Avoid limiting your template; avoid hardcoding for specific types or for less general types.**

If enforcing that signature for callback functions is sufficient, then by all means stop right there. There's no sense in complicating a design if we don't need to—for complicate it we will, if we want to allow for more function signatures!

I won't write out all the code, because it's significantly tedious. (If you really want to see code this repetitive, or you're having trouble with insomnia, see books and articles like [Alexandrescu01] for similar examples.) What I will do is briefly sketch the main things you'd have to support and how you'd have to support them:

First, what about const member functions? The easiest way to deal with this one is to provide a parallel callback that uses the **const** signature type, and in that version, remember to take and hold the **T** by reference or pointer to **const**.

Second, what about non-**void** return types? The simplest way to allow the return type to vary is by adding another template parameter.

Third, what about callback functions that take parameters? Again, add template parameters, remember to add parallel function parameters to **operator()**, and stir well.

Remember to add a new template to handle each potential number of callback arguments.

Alas, the code explodes, and you have to do things like set artificial limits on the number of function parameters that callback supports. Perhaps in a future C++0x language we'll have features like template "varargs" that will help to deal with this, but not today.

### Summary

Putting it all together, and making some purely stylistic adjustments like using **typename** consistently and naming conventions and whitespace conventions that I happen to like better, here's what we get:

```
class CallbackBase {
public:
  virtual void operator()() const { };
  virtual ~CallbackBase() = 0;
};

CallbackBase::~CallbackBase() { }

template<typename T>
class Callback : public CallbackBase {
public:
  typedef void (T::*F)();

  Callback( T& t, F f )       : t_(&t), f_(f) { }
  void operator()() const  { (t_->*f_)(); }

private:
  T* t_;
  F  f_;
};

template<typename T>
Callback<T> make_callback( T& t, void (T::*f) () ) {
  return Callback<T>( t, f );
}
```

# 36. Construction Unions

**Difficulty: 4**

*No, this Item isn't about organizing carpenters and bricklayers. Rather, it's about deciding between what's cool and what's uncool, good motivations gone astray, and the consequences of subversive activities carried on under the covers. It's about getting around the C++ rule of using constructed objects as members of unions.*

### JG Questions

1. What are unions, and what purpose do they serve?
2. What kinds of types cannot be used as members of unions? Why do these limitations exist? Explain.

### Guru Questions

3. The article [Manley02] cites the motivating case of writing a scripting language: Say that you want your language to support a single type for variables that at various times can hold an integer, a **string**, or a **list**. Creating a **union { int i; list<int> l; string s; };** doesn't work for the reasons covered in Questions 1 and 2. The following code presents a workaround that attempts to support allowing any type to participate in a **union**. (For a more detailed explanation, see the original article.)

   Critique this code and identify:

   a) Mechanical errors, such as invalid syntax or nonportable conventions.

   b) Stylistic improvements that would improve code clarity, reusability, and maintainability.

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
  MYUNION() : currtype( NONE ) {}
  ~MYUNION() {cleanup();}

  enum uniontype {NONE,_INT,_LIST,_STRING};
  uniontype currtype;
```

```
  inline int& getint();
  inline LIST& getlist();
  inline STRING& getstring();

protected:
  union {
    int i;
    unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;

  void cleanup();
};

inline int& MYUNION::getint()
{
  if( currtype==_INT ) {
    return U.i;
  } else {
    cleanup();
    currtype=_INT;
    return U.i;
  } // else
}

inline LIST& MYUNION::getlist()
{
  if( currtype==_LIST ) {
    return *(reinterpret_cast<LIST*>(U.buff));
  } else {
    cleanup();
    LIST* ptype = new(U.buff) LIST();
    currtype=_LIST;
    return *ptype;
  } // else
}

inline STRING& MYUNION::getstring()
{
  if( currtype==_STRING) {
    return *(reinterpret_cast<STRING*>(U.buff));
  } else {
    cleanup();
    STRING* ptype = new(U.buff) STRING();
    currtype=_STRING;
    return *ptype;
  } // else
}
```

```
void MYUNION::cleanup()
{
  switch( currtype ) {
    case _LIST: {
      LIST& ptype = getlist();
      ptype.~LIST();
      break;
    } // case
    case _STRING: {
      STRING& ptype = getstring();
      ptype.~STRING();
      break;
    } // case
    default: break;
  } // switch
  currtype=NONE;
}
```

4. Show a better way to achieve a generalized variant type, and comment on any tradeoffs you encounter.

# Solution

### Unions Redux

***1. What are unions, and what purpose do they serve?***

Unions allow more than one object, of either class or built-in type, to occupy the same space in memory. For example:

```
// Example 36-1
//
union U {
  int i;
  float f;
};

U u;

u.i = 42;                              // ok, now i is active
std::cout << u.i << std::endl;

u.f = 3.14f;                           // ok, now f is active
std::cout << 2 * u.f << std::endl;
```

But only one of the types can be "active" at a time—after all, the storage can hold only one value at a time. Also, unions support only some kinds of types, which leads us into the next question:

2. ***What kinds of types cannot be used as members of unions? Why do these limitations exist? Explain.***

From the C++ standard:

> *An object of a class with a non-trivial constructor, a non-trivial copy constructor, a non-trivial destructor, or a non-trivial copy assignment operator cannot be a member of a union, nor can an array of such objects.*

In brief, for a class type to be usable in a union, it must meet all the following criteria:

- The only constructors, destructors, and copy assignment operators are the compiler-generated ones.
- There are no virtual functions or virtual base classes.
- Ditto for all of its base classes and nonstatic members (or arrays thereof).

That's all, but that sure eliminates a lot of types.

Unions were inherited from C. The C language has a strong tradition of efficiency and support for low-level close-to-the-metal programming, which has been compatibly preserved in C++; that's why C++ also has unions. On the other hand, the C language does not have any tradition of language support for an object model supporting class types with constructors and destructors and user-defined copying, which C++ definitely does; that's why C++ also has to define what, if any, uses of such newfangled types make sense with the "oldfangled" unions and do not violate the C++ object model including its object lifetime guarantees.

If C++'s restrictions on unions did not exist, Bad Things could happen. For example, consider what could happen if the following code were allowed:

```
// Example 36-2: Not standard C++ code, but what if it were allowed?
//
void f() {
  union IllegalImmoralAndFattening {
    std::string s;
    std::auto_ptr<int> p;
  };

  IllegalImmoralAndFattening iiaf;

  iiaf.s = "Hello, world";        // has s's constructor run?
  iiaf.p = new int(4);            // has p's constructor run?
}    // will s get destroyed? should it be? will p get destroyed? should it be?
```

As the comments indicate, serious problems would exist if this were allowed. To avoid further complicating the language by trying to craft rules that at best might only partly patch up a few of the problems, the problematic operations were simply banished.

But don't think that unions are only a holdover from earlier times. Unions are perhaps most useful for saving space by allowing data to overlap, and this is still desirable in C++ and in today's modern world. For example, some of the most advanced C++ standard library implementations in the world now use just this technique for implementing the "small **string** optimization," a great optimization alternative that reuses the storage inside a **string** object itself. Here's the idea: For large strings, space inside the **string** object stores the usual pointer to the dynamically allocated buffer and housekeeping information like the size of the buffer; for small strings, however, the same space is instead reused to store the string contents directly and completely avoid any dynamic memory allocation. For more about the small **string** optimization (and other string optimizations and pessimizations in considerable depth), see Items 13 through 16 in *More Exceptional C++* [Sutter02]; see also the discussion of current commercial **std::string** implementations in [Meyers01].

## Dissecting Construction Unions

***3. The article [Manley02] cites the motivating case of writing a scripting language: Say that you want your language to support a single type for variables that at various times can hold an integer, a string, or a list. Creating a union { int i; list<int> l; string s; }; doesn't work for the reasons covered in Questions 1 and 2. The following code presents a workaround that attempts to support allowing any type to participate in a union. (For a more detailed explanation, see the original article.)***

On the plus side, the cited article addresses a real problem, and clearly much effort has been put into coming up with a good solution. Unfortunately, from well-intentioned beginnings, more than one programmer has gone badly astray.

The problems with the design and the code fall into three major categories: legality, safety, and morality.

***Critique this code and identify:***

***a) Mechanical errors, such as invalid syntax or nonportable conventions.***

***b) Stylistic improvements that would improve code clarity, reusability, and maintainability.***

The first overall comment that needs to be made is that the fundamental idea behind this code is not legal in standard C++. The original article summarizes the key idea:

> *The idea is that instead of declaring object members, you instead declare a raw buffer [non-dynamically, as a char array member inside the object pretending to act like a union] and instantiate the needed objects on the fly [by in-place construction].*—[Manley02]

The idea is common, but unfortunately it is also unsound.

Allocating a buffer of one type and then using casts to poke objects of another type in and out, is nonconforming and nonportable because buffers that are not dynamically allocated (i.e., that are not allocated via **malloc** or **new**) are not guaranteed to be correctly aligned for any other type than the one with which they were originally declared. Even if this technique happens to accidentally work for some types on someone's current compiler, there's no guarantee it will continue to work for other types or for the same types in the next version of the same compiler. For more details and some directly related discussion, see Item 30 in *Exceptional C++* [Sutter00], notably the sidebar titled "Reckless Fixes and Optimizations, and Why They're Evil." See also the alignment discussion in [Alexandrescu02].

For C++0x, the standards committee is considering adding alignment aids to the language specifically to enable techniques that rely on alignment like this, but that's all still in the future. For now, to make this work reasonably reliably even some of the time, you'd have to do one of the following:

- Rely on the **max_align** hack (see [Manley02] which footnotes the **max_align** hack, or do a Google search for **max_align**).
- Rely on nonstandard extensions like Gnu's **__alignof__** to make this work reliably on a particular compiler that supports such an extension. (Even though Gnu provides an **ALIGNOF** macro intended to work more reliably on other compilers, it too is admitted hackery that relies on the compiler's laying out objects in certain ways and making guesses based on **offsetof** inquiries, which might often be a good guess but is not guaranteed by the standard.)

You could work around this by dynamically allocating the array using **malloc** or **new**, which would guarantee that the **char** buffer is suitably aligned for an object of any type, but that would still be a bad idea (it's still not type-safe) and it would defeat the potential efficiency gains that the original article was aiming for as part of its original motivation. An alternative and correct solution would be to use **boost::any** (see below), which incurs a similar allocation/indirection overhead but is at least both safe and correct; more about that later on.

Attempts to work against the language, or to make the language work the way we want it to work instead of the way it actually does work, are often questionable and should be a big red flag. In the *Exceptional C++* [Sutter00] sidebar cited earlier, while

in an ornery mood, I also accused a similar technique of "just plain wrongheadedness" followed by some pretty strong language. There can still be cases where it could be reasonable to use constructs that are known to be nonportable but okay in a particular environment (in this case, perhaps using the **max_align** hack), but even then I would argue that that fact should be noted explicitly and, further, that it still has no place in a general piece of code recommended for wide use.

### Into the Code

Let's now consider the code:

```
#include <list>
#include <string>
#include <iostream>
using namespace std;
```

Always include necessary headers. Because **new** is going to be used below, we need to also **#include <new>**. (Note: The **<iostream>** header is okay; later in the original code, not shown here, was a test harness that emitted output using iostreams.)

```
#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
  MYUNION() : currtype( NONE ) {}
  ~MYUNION() {cleanup();}
```

The first classic mechanical error here is that **MYUNION** is unsafe to copy because the programmer forgot to provide a suitable copy constructor and copy assignment operator.

**MYUNION** is choosing to play games that require special work to be done in the constructor and destructor, so these functions are provided explicitly as shown; that's fine as far as it goes. But it doesn't go far enough, because the same games require special work in the copy constructor and copy assignment operator, which are *not* provided explicitly. That's bad because the default compiler-generated copying operations do the wrong thing; namely, they copy the contents bitwise as a **char** array, which is likely to have most unsatisfactory results, in most cases leading straight to memory corruption. Consider the following code:

```
// Example 36-3: MYUNION is unsafe for copying
//
{
  MYUNION u1, u2;
  u1.getstring() = "Hello, world";
  u2 = u1;                              // copies the bits of u1 to u2
} // oops, double delete of the string (assuming the bitwise copy even made sense)
```

> ➢ **Guideline:** **Observe the Law of the Big Three [Cline99]: If a class needs a custom copy constructor, copy assignment operator, or destructor, it probably needs all three.**

Passing on from the classic mechanical error, we next encounter a duo of classic stylistic errors:

```
enum uniontype {NONE,_INT,_LIST,_STRING};
uniontype currtype;

inline int& getint();
inline LIST& getlist();
inline STRING& getstring();
```

There are two stylistic errors here. First, this **struct** is not reusable because it is hard-coded for specific types. Indeed, the original article recommended hand-coding such a struct every time it was needed. Second, even given its limited intended usefulness, it is not very extensible or maintainable. We'll return to this frailty again later, once we've covered more of the context.

> ➢ **Guideline:** **Avoid hard-wiring information that needlessly makes code more brittle and limits flexibility.**

There are also two mechanical problems. The first is that **currtype** is public for no good reason; this violates good encapsulation and means any user can freely mess with the type flag, even by accident. The second mechanical problem concerns the names used in the enumeration; I'll cover that in its own section, "Underhanded Names," later on.

```
protected:
```

Here we encounter another mechanical error: The internals ought to be private, not protected. The only reason to make them protected would be to make the internals available to derived classes, but there had better not be any derived classes because **MYUNION** is unsafe to derive from for several reasons—not least because of the murky and abstruse games it plays with its internals and because it lacks a virtual destructor.

> ➢ **Guideline:** **Always make all data members private. The only exception is the case of a C-style struct which isn't intended to encapsulate anything and where all members are public.**

```
  union {
    int i;
    unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;

  void cleanup();
};
```

That's it for the main class definition. Moving on, consider the three parallel accessor functions:

```
inline int& MYUNION::getint()
{
  if( currtype==_INT ) {
    return U.i;
  } else {
    cleanup();
    currtype=_INT;
    return U.i;
  } // else
}

inline LIST& MYUNION::getlist()
{
  if( currtype==_LIST ) {
    return *(reinterpret_cast<LIST*>(U.buff));
  } else {
    cleanup();
    LIST* ptype = new(U.buff) LIST();
    currtype=_LIST;
    return *ptype;
  } // else
}

inline STRING& MYUNION::getstring()
{
  if( currtype==_STRING) {
    return *(reinterpret_cast<STRING*>(U.buff));
  } else {
    cleanup();
    STRING* ptype = new(U.buff) STRING();
    currtype=_STRING;
    return *ptype;
  } // else
}
```

A minor nit: The **// else** comments add nothing. It's unfortunate that the only comments in the code are useless ones.

> ➢ **Guideline:** **Write (only) useful comments. Never write comments that repeat the code; instead, write comments that explain the code and the reasons why you wrote it that way.**

More seriously, there are three major problems here. The first is that the functions are not written symmetrically, and whereas the first use of a **list** or a **string** yields a default-constructed object, the first use of **int** yields an uninitialized object. If that is intended, in order to mirror the ordinary semantics of uninitialized **int** variables, then that should be documented; because it is not, the **int** ought to be initialized. For example, if the caller accesses **getint** and tries to make a copy of the (uninitialized) value, the result is undefined behavior—not all platforms support copying arbitrary invalid **int** values, and some will reject the instruction at run-time.

The second major problem is that this code hinders const-correct use. If the code is really going to be written this way, then at least it would be useful to also provide **const** overloads for each of these functions; each would naturally return the same thing as its non-**const** counterpart, but by a reference to **const**.

> ➢ **Guideline:** **Practice const-correctness.**

The third major problem is that this approach is fragile and brittle in the face of change. It relies on type switching, and it's easy to accidentally fail to keep all the functions in sync when you add or remove new types.

Stop reading here and consider: What do you have to do in the published code if you want to add a new type? Make as complete a list as you can.

• • • • • •

Are you back? All right, here's the list I came up with. To add a new type, you have to remember to:

- Add a new **enum** value;
- Add a new accessor member;
- Update the **cleanup** function to safely destroy the new type; and
- Add that type to the **max** calculation to ensure **buff** is sufficiently large to hold the new type too.

If you missed one or more of those, well, that just illustrates how difficult this code really is to maintain and extend.

Pressing onward, we come to the final function:

```
void MYUNION::cleanup()
{
 switch( currtype ) {
  case _LIST: {
   LIST& ptype = getlist();
   ptype.~LIST();
   break;
  } // case
  case _STRING: {
   STRING& ptype = getstring();
   ptype.~STRING();
   break;
  } // case
  default: break;
 } // switch
 currtype=NONE;
}
```

Let's reprise that small commenting nit: The **// case** and **// switch** comments add nothing; it's unfortunate that the only comments in the code are useless ones. It is better to have no comments at all than to have comments that are just distractions.

But there's a larger issue here: Rather than having simply **default: break;**, it would be good to make an exhaustive list (including the **int** type) and signal a logic error if the type is unknown—perhaps via an **assert** or a **throw std::logic_error(...);** .

Again, type switching is purely evil. A Google search for *switch C++ Dewhurst* will yield all sorts of interesting references on this topic, including [Dewhurst02]. See those references for more details if you need more ammo to convince colleagues to avoid the type-switching beast.

➢ **Guideline:** **Avoid type switching; prefer type safety.**

### Underhanded Names

There's one mechanical problem I haven't yet covered. This problem first rears its ugly, unshaven, and unshampooed head in the following line:

```
enum uniontype {NONE,_INT,_LIST,_STRING};
```

Never, ever, ever create names that begin with an underscore or contain a double underscore; they're reserved for your compiler and standard library vendor's exclusive use so that they have names they can use without tromping on your code. Tromp on their names, and their names might just tromp back on you![41]

*Don't stop! Keep reading!* You might have read that advice before. You might even have read it from me. You might even be tired of it, and yawning, and ready to ignore the rest of this section. If so, this one's for you, because this advice is not at all theoretical, and it bites and bites hard in this code.

The enum definition line happens to compile on most of the compilers I tried: Borland 5.5, Comeau 4.3.0.1, gcc 2.95.3 / 3.1.1 / 3.4, Intel 7.0, and Microsoft Visual C++ 6.0 through 8.0 (2005) beta. But under two of them—Metrowerks CodeWarrior 8.2 and EDG 3.0.1 used with the Dinkumware 4.0 standard library—the code breaks horribly.

Under Metrowerks CodeWarrior 8, this one line breaks noisily with the first of 52 errors (that's not a typo). The 225 lines of error messages (again, that's not a typo) begin with the following diagnostics, which point straight at one of the commas:

```
### mwcc Compiler:
#    File: 36.cpp
# --------------
#      17:     enum uniontype {NONE,_INT,_LIST,_STRING};
#   Error:                                    ^
#   identifier expected
### mwcc Compiler:
#      18:     uniontype currtype;
#   Error:    ^^^^^^^^^
#   declaration syntax error
```

followed by 52 further error messages and 215 more lines. What's pretty obvious from the second and later errors is that we should ignore them for now because they're just cascades from the first error—because **uniontype** was never successfully defined, the rest of the code which uses **uniontype** extensively will of course break too.

But what's up with the definition of **uniontype**? The indicated comma sure looks like it's in a reasonable place, doesn't it? There's an identifier happily sitting in front of it, isn't there? All becomes clear when we ask the Metrowerks compiler to spit out

[41] The more specific rule is that any name with a double underscore anywhere in it **like__this** or that starts with an underscore and a capital letter **_LikeThis** is reserved. You can remember that rule if you like, but it's a bit easier to just avoid both leading underscores and double underscores entirely.

the preprocessed output… omitting many many lines, here's what the compiler finally sees:

```
enum uniontype {NONE,_INT, , };
```

Aha! That's not valid C++, and the compiler rightly complains about the third comma because there's no identifier in front of it.

But what happened to **_LIST** and **_STRING**? You guessed it—tromped on and eaten by the ravenously hungry Preprocessor Beast. It just so happens that Metrowerks' implementation has macros that happily strip away the names **_LIST** and **_STRING**, which is perfectly legal and legitimate because it (the implementation) is allowed to own those **_Names** (as well as **other__names**).

So Metrowerks' implementation happens to eat both **_LIST** and **_STRING**. That solves that part of the mystery. But what about EDG's/Dinkumware's implementations? Judge for yourself:

```
"1.cpp", line 17: error: trailing comma is nonstandard
    enum uniontype {NONE,_INT,_LIST,_STRING};
                                            ^

"1.cpp", line 58: error: expected an expression
    if( currtype==_STRING) {
                           ^

"1.cpp", line 63: error: expected an expression
        currtype=_STRING;
                         ^

"1.cpp", line 76: error: expected an expression
        case _STRING: {
                      ^

4 errors detected in the compilation of "36.cpp".
```

This time, even without generating and inspecting a preprocessed version of the file, we can see what's going on: The compiler is behaving as though the word **_STRING** just wasn't there. That's because it was—you guessed it—tromped on, not to mention thoroughly chewed up and spat out, by the still-peckish Preprocessor Beast.

I hope that this will convince you that when some of us boring writers natter on about not using **_Names like__these**, the problem is far from theoretical, far more than mere academic tedium. It's a practical problem indeed, because the naming restriction directly affects your relationship with your compiler and standard library writer. Trespass on their turf, and you might get lucky and remain unscathed; on the other hand, you might not.

The C++ landscape is wide open and clear and lets you write all sorts of wonderful and flexible code and wander in pretty much whatever direction your development heart desires, including that it lets you choose pretty much whatever names you like outside of namespace **std**. But when it comes to names, C++ also has one big fenced-off grove, surrounded by gleaming barbed wire and signs that say things like "**Employees_Only**—Must Have Valid **_Badge** To Enter Here" and "Violators Might be Tromped and Eaten." This is a stellar example of the tromping one gets for disregarding the **_Warnings**.

➢ **Guideline:** **Never use "underhanded names"—ones that begin with an underscore or that contain a double underscore. They are reserved for your compiler and standard library implementation.**

### Toward a Better Way: boost::any

***4. Show a better way to achieve a generalized variant type, and comment on any tradeoffs you encounter.***

The original article says:

> *[Y]ou might want to implement a scripting language with a single variable type that can either be an integer, a string, or a list.*—[Manley02]

This is true, and there's no disagreement so far. But the article then continues:

> *A union is the perfect candidate for implementing such a composite type.* —[Manley02]

Rather, the article has served to show in some considerable detail just why a union is not suitable at all.

But if not a union, then what? One very good candidate for implementing such a variant type is [Boost]'s **any** facility, along with its **many** and **any_cast**.[42] Interestingly, the complete implementation for the fully general **any** (covering any number/combination of types and even some platform-specific **#ifdef**s) is about the same amount of code as the sample **MYUNION** solution hardwired for just the special case of the three types **int**, **list<int>**, and **string**—and **any** is fully general, extensible, and type-safe to boot, and part of a healthy low-cholesterol diet.

There is still a tradeoff, however, and it is this: dynamic allocation. The **boost::any** facility does not attempt to achieve the potential efficiency gain of avoiding a dynamic memory allocation, which was part of the motivation in the original article.

[42] For further discussion of this facility, see [Hyslop01].

Note too that the **boost::any** dynamic allocation overhead is more than if the original article's code was just modified to use (and reuse) a single dynamically allocated buffer that's acquired once for the lifetime of **MYUNION**, because **boost::any** also performs a dynamic allocation every time the contained type is changed.

Here's how the article's demo harness would look if it instead used **boost::any**. The old code that uses the original article's version of **MYUNION** is shown in comments for comparison:

```
any u;                                  // instead of: MYUNION u;
```

Instead of a handwritten struct, which has to be written again for each use, just use **any** directly. Note that **any** is a plain class, not a template.

```
// access union as integer
u = 12345;                              // instead of: u.getint() = 12345;
```

The assignment shows **any**'s more natural syntax.

```
cout << "int=" << any_cast<int>(u) << endl;          // or just int(u)
                                // instead of: cout << "int=" << u.getint() << endl;
```

I like **any**'s cast form better because it's more general (including that it is a non-member) and more natural to C++ style; you could also use the less verbose **int(u)** without an **any_cast** if you know the type already. On the other hand, **MYUNION**'s **get***[type]* is more fragile, harder to write and maintain, and so forth.

```
// access union as std::list
u = list<int>();
list<int>& l = *any_cast<list<int> >(&u);   // instead of: LIST& list = u.getlist();
l.push_back(5);                             // same: list.push_back(5);
l.push_back(10);                            // same: list.push_back(10);
l.push_back(15);                            // same: list.push_back(15);
```

I think **any_cast** could be improved to make it easier to get references, but this isn't too bad. (Aside: I'd discourage using **list** as a variable name when it's also the name of a template in scope; too much room for expression ambiguity.)

So far we've achieved some typability and readability savings. The remaining differences are more minor:

```
list<int>::iterator it = l.begin();         // instead of: LIST::iterator it = list.begin();

while( it != l.end() ) {
  cout << "list item=" << *(it) << endl;
  it++;
} // while
```

Pretty much unchanged. I've let the original comment stand because it's not germane to the side-by-side style comparison with **any**.

```
// access union as std::string
u = string("Hello world!");              // instead of:  STRING& str = u.getstring();
                                         //              str = "Hello world!";
```

Again, about a wash; I'd say the **any** version is slightly simpler than the original, but only slightly.

```
cout << "string='" << any_cast<string>(u) << "'" << endl;     // or just "string(u)"
                        // instead of: cout << "string='" << str.c_str() << "'" << endl;
```

As before.

### Alexandrescu's Discriminated Unions

Is it possible to fully achieve both of the original goals—safety and avoiding dynamic memory—in a conforming standard C++ implementation? That sounds like a problem that someone like Andrei Alexandrescu would love to sink his teeth into, especially if it could somehow involve complicated templates. As evidenced in [Alexandrescu02], where he describes his discriminated unions (aka **Variant**) approach, it turns out that:

- it is (something he would love to tackle), and
- it can (involve weird templates, and just one quote from [Alexandrescu02] says it all: "Did you know that unions can be templates?"), so
- he does.

In short, by performing heroic efforts to push the boundaries of the language as far as possible, Alexandrescu's **Variant** comes very close to being a truly portable solution. It falls only slightly short and is probably portable enough in practice even though it too goes beyond the pale of what the Standard guarantees. Its main problem is that, even ignoring alignment-related issues, the **Variant** code is so complex and advanced that it actually works on very few compilers—in my testing, I only managed to get it to work with one.

A key part of Alexandrescu's **Variant** approach is an attempt to generalize the **max_align** idea to make it a reusable library facility that can itself still be written in conforming standard C++. The reason for wanting this is specifically to deal with the alignment problems in the code we've been analyzing so that a non-dynamic **char** buffer can continue to be used in relative safety. Alexandrescu makes heroic efforts to use template metaprogramming to calculate a safe alignment. Will it work portably? His discussion of this question follows:

> *Even with the best* ***Align****, the implementation above is still not 100-percent portable for all types. In theory, someone could implement a compiler that respects the Standard but still does not work properly with discriminated unions. This is because the Standard does not guarantee that all user-defined types ultimately have the alignment of some POD type. Such a compiler, however, would be more of a figment of a wicked language lawyer's imagination, rather than a realistic language implementation.*
>
> *[…] Computing alignment portably is hard, but feasible. It never is 100-percent portable.* — [Alexandrescu02]

There are other key features in Alexandrescu's approach, notably a **union** template that takes a **typelist** template of the types to be contained, visitation support for extensibility, and an implementation technique that will "fake a vtable" for efficiency to avoid an extra indirection when accessing a contained type. These parts are more heavyweight than **boost::any** but are portable in theory. That "portable in theory" part is important—as with Andrei's great work in *Modern C++ Design* [Alexandrescu01], the implementation is so heavy on templates that the code itself contains comments like, "Guaranteed to issue an internal compiler error on: [various popular compilers, Metrowerks, Microsoft, Gnu gcc]," and the mainline test harness contains a commented-out test helpfully labeled "The construct below didn't work on any compiler."

That is **Variant**'s major weakness: Most real-world compilers don't even come close to being able to handle this implementation, and the code should be viewed as important but still experimental. I attempted to build Alexandrescu's **Variant** code using a variety of compilers: Borland 5.5; Comeau 4.3.0.1; EDG 3.0.1; gcc 2.95, 3.1.1, and 3.2; Intel 7.0; Metrowerks 8.2; and Microsoft VC++ 6.0, 7.0 (2002), and 7.1 (2003). As some readers will know, some of the products in that list are very strong and standards-conforming compilers. None of these compilers could successfully compile Alexandrescu's template-heavy source as it was provided.

I tried to massage the code by hand to get it through any of the compilers but was successful only with Microsoft VC++ 7.1 (2003). Most of the compilers didn't stand a chance, because they did not have nearly strong enough template support to deal with Alexandrescu's code. (Some emitted a truly prodigious quantity of warnings and errors—Intel 7.0's response to compiling **main.cpp** was to spew back an impressive 430K worth—really, nearly half a megabyte—of diagnostic messages.)

I had to make three changes to get the code to compile without errors (although still with some narrowing-conversion warnings at the highest warning level) under Microsoft VC++ 7.1 (2003):

- Added a missing **typename** in class **AlignedPOD**.
- Added a missing **this->** to make a name dependent in **ConverterTo<>::-Unit<>::DoVisit()**.
- Added a final newline character at the end of several headers, as required by the C++ standard (some conforming compilers aren't strict about this and allow the absence of a final newline as a conforming extension; VC++ is stricter and requires the newline).[43]

As the author of [Manley02] commented further about tradeoffs in Alexandrescu's design:

> *It doesn't use dynamic memory, and it avoids alignment issues and type switching. Unfortunately I don't have access to a compiler that can compile the code, so I can't evaluate its performance vs.* ***myunion*** *and* ***any****. Alexandrescu's approach requires 9 supporting header files totaling ~80KB, which introduces its own set of maintenance problems.* — K. Manley, private communication

Those points are all valid.

I won't try to summarize Andrei's three articles further here, but I encourage readers who are interested in this problem to look them up. They're available online as indicated in the bibliography.

---

➢ **Guideline:** **If you want to represent variant types, for now prefer to use boost::any (or something equally simple).**

---

Once the compiler you are using catches up (in template support) and the Standard catches up (in true alignment support) and **Variant** libraries catch up (in mature implementations), it will be time to consider using **Variant**-like library tools as type-safe replacements for unions.

### Summary

Even if the design and implementation of **MYUNION** are lacking, the motivating problem is both real and worth considering. I'd like to thank Mr. Manley for taking the time to write his article and raise awareness of the need for variant type support and Kevlin Henney and Andrei Alexandrescu for contributing their own solutions to this area. It is a hard enough problem that Manley's and Alexandrescu's approaches are not strictly portable, standards-conforming C++, although Alexandrescu's **Vari-**

---

[43] Thanks to colleague Jeff Peil for pointing out this requirement in [C++03] § 2.1/1, which states: "If a source file that is not empty does not end in a new-line character, or ends in a new-line character immediately preceded by a backslash character, the behavior is undefined."

**ant** makes heroic efforts to get there—Alexandrescu's design is very close to portable in theory, although the implementation is still far from portable in practice because very few compilers can handle the advanced template code it uses.

For now, an approach like **boost::any** is the preferred way to go. If in certain places your measurements tell you that you really need the efficiency or extra features provided by something like Alexandrescu's **Variant**, and you have time on your hands and some template know-how, you might experiment with writing your own scaled-back version of the full-blown **Variant** by applying only the ideas in [Alexandrescu02] that are applicable to your situation.