

# Using Generic Programming Techniques In C++ With The Mac OS Toolbox

=====

by Joshua Juran

## Abstract

-----

Whereas inheritance, polymorphism, and other object-oriented techniques are familiar to Mac developers using C++, generic programming techniques (judging by current common practice) typically are not. Generic programming (specifically, the use of templates) allows the various classes and other types of an application that were developed without knowledge of each other to be combined in useful ways, while maintaining type safety (without even requiring the use of `dynamic_cast`). Classes may implement a required interface without deriving from an abstract base (e.g. iterators).

Examples of generic programming applied include reference-counting pointer objects, also known as 'smart pointers'. Not only are smart pointers useful for deallocating memory, when combined with system resource objects (e.g. Mac resource handles, file descriptors, sockets, windows, etc.) they are indispensable for ensuring that these resources get released, even (or especially) when exceptions are thrown.

An in-depth look at the Pedestal framework demonstrates these techniques in action.

(Source will be provided under the GNU GPL on the MacHack CD.)

## Outline

-----

Introduction to Generic Programming

What It Isn't

What It Is

How C++ Provides Generic Programming

Templates

Overloaded Operators and Functions

Template Libraries  
Standard Template Library (STL)  
Boost

Combining Templates and Classes  
Stack-based Objects  
Resource Leaks  
An example of a resource leak  
A Limitation of Stack-based Objects  
Smart Pointers  
A Smart Pointer Class Template

Other Examples of Generic Programming  
Windowing

Bibliography

-----

Using Generic Programming Techniques In C++ With The Mac OS Toolbox

=====

=

by Joshua Juran

Introduction to Generic Programming

-----

One of the major advantages of programming in C++ is the language's capacity for the application of /generic programming/. Generic programming, like object-oriented programming, is a style of programming that includes practices to increase the correctness and reuseability (and therefore, usefulness) of source code. But whereas the intended outcomes of these disciplines may match, their respective practices are quite different.

What It Is Not

Generic programming is not object-oriented programming. The two are by no

means incompatible; in fact they're quite useful when combined, although unrelated -- much the way a flame, a skillet, a spatula, and some eggs are, if you happen to like scrambled eggs. In a nutshell, object-oriented programming involves the concept of the class -- essentially, a struct-like type with its own namespace and bindings to functions. Key object-oriented techniques are encapsulation, inheritance, and polymorphism. Generic programming techniques include factoring and combination.

Generic programming also does not mean cross-platform or platform-independent programming. The latter of those two simply means restricting one's code to features guaranteed by the relevant standard (be it a language or API), while the other involves selective use of complementary extensions on different platforms. For example, code written to use the TK windowing toolkit is platform-independent, whereas TK itself is cross-platform. While these disciplines allow code to be reused across different platforms, generic programming promotes reuse within the logic of the program itself.

### What It Is

If the basic premise of writing classes is that the code that directly manipulates an object's representation (its methods) should be written only once (and not duplicated among calling functions), then the idea behind generic programming is that algorithms should be written only once, and not duplicated across different types. Generic programming requires a way to write a single, type-safe definition of an algorithm that works across any type that supports the operations the algorithm requires. For example, a generic factorial algorithm should accept a parameter of any type that supports equality-with-integer, subtraction-of-integer, multiplication-by-same-type and assignment-from-same-type operations. All built-in numerical types (i.e. integral and floating-point types) are candidates, but so are user-defined types (such as classes) that meet the criteria. (By way of example, a 'bignum' class would be well-suited to this application, a complex number class would compile but possibly loop forever if the imaginary part was nonzero and the factorial implementation didn't check for this, and a matrix class would fail to compile because a matrix can't be compared for equality with an integer.) Herein lies one of the key strengths of generic programming -- an algorithm and a user-defined type, each developed without knowledge of the other, can nevertheless be combined (given matched demand and support for operations) without requiring any modification whatsoever.

## How C++ Provides Generic Programming

-----

### Templates

The mechanism whereby C++ provides the ability to use generic programming is templates. A general sense of what templates are and why they're needed is illustrated by examples of how to cope without them, attempting to implement `min(a, b)` or a simple stack:

#### Non-solution #1: C preprocessor macros

```
#define min(a, b) (a < b) ? a : b
```

The first problem with macros is that they look like functions but they're not. With this definition, `(min(1, 2) + 3)` will return 1, not 4. Trying again with the requisite parentheses:

```
#define min(a, b) (((a) < (b)) ? (a) : (b))
```

The second problem with macros is that they're often confusing to read, and therefore less maintainable than alternatives. But the final nail in macros' collective coffin is the problem of side effects: for example, `min(a, b++)` increments `b` twice. Assuming, of course, the algorithm was simple enough to be implemented by a macro in the first place -- most of the interesting algorithms, such as factorial, are too complex.

#### Non-solution #2: Function overloading

```
inline int min(int a, int b) { return (a < b) ? a : b; }
inline double min(double a, double b) { return (a < b) ? a : b; }
```

This non-solution works perfectly, except that it fails to meet the 'single definition' requirement. Any changes to the algorithm logic must be propagated to each definition. This example shows only two of the potentially infinite number of types implementing the less-than operator.

#### Non-solution #3: Abstract base class

```
class Stack {
public:
```

```
class Item {}; // abstract base class
```

```
void Push(Item* it);  
Item* Pop();  
bool Empty();  
};
```

```
class Thing : public Stack::Item {};
```

Not only is this non-solution invasive, permitting storage only objects of types derived from the abstract base class (requiring advance knowledge during development), upon retrieval the type information is lost. Also, storing pointers instead of objects requires additional memory management. It's possible (and in C, necessary) to use void pointers to the same effect.

The solution: Templates

Templates allow a programmer to achieve the same effect as function overloading (#2, above), but writing the function (or class) once as a /template/, /instantiating/ the template across various parameters (types or constants) as necessary, and leaving the repetitive task of 'filling in' the template to the compiler.

```
template <class Num>  
inline Num min(Num a, Num b) { return (a < b) ? a : b; }  
  
int i = min(1, 2);  
double d1 = min(double(22/7), 3.14159); // oops -- integer division  
double d2 = min(22/7.0, 3.14159); // okay  
string str1 = min("this", "that"); // oops -- pointer comparison  
string s1 = "this";  
string s2 = "that";  
string str2 = min(s1, s2); // okay, uses string::operator<()
```

```
template <class Item>  
class Stack {  
public:  
void Push(const Item& it);  
Item Pop();  
bool Empty();  
};
```

```
class Thing {}; // developed independently of Stack
```

```
Stack<Thing> stack;  
Thing thing;  
stack.Push(thing);
```

In this implementation, `min()` is a function template and `Stack` is a class template. It's important to note that `Stack` is not a class, nor even a type. `Stack<Thing>` is not a derived class of `Stack`, `Stack<>`, `Stack<Item>`, or any such thing. `Stack<Thing>` is the instantiation of class template `Stack` across the type parameter `Thing`, resulting in a new class. Likewise, instantiating the function template `min()` across the type parameter `int` yields a new function, `min(int, int)`. These instantiations happen at compile time -- there is no run-time overhead.

## Overloaded Operators and Functions

Generic programming of algorithms requires the ability to operate on built-in types (such as `int`) and user-defined types (such as classes) through the same interface. Typically, values of built-in types are manipulated with operators and class objects are accessed through methods. How does one bridge the gap? Java takes an object-based approach -- everything's an object, so you might wind up with constructions like an `Addable` interface with a `Plus()` method. In C++, however, the reverse is true -- a class can overload the operators and behave like a built-in type. For example, see the `min()` template above.

## Template Libraries

C++ templates are a powerful tool for creating reusable code; even better is that others have been doing exactly that -- developing libraries of reusable code that are themselves powerful tools for writing software. Most notable is the Standard Template Library (STL), now part of the C++ Standard Library. The STL provides containers (including strings), iterators, and algorithms. There is rarely good cause to look elsewhere for a feature that exists in the STL.

Another generic programming library is Boost <<http://www.boost.com/>>. Boost picks up where STL left off -- in fact, some of the Boost modules are being considered for inclusion in the C++ Standard Library. Boost includes libraries for threading and 'smart pointers' (explained later) among others.

## Combining Templates and Classes

-----

### Stack-based Objects

One feature of C++ that may be initially overlooked by programmers familiar with C and Object Pascal is the availability of stack-based class objects. Object Pascal programmers know about objects, of course, and C programmers know that auto variables are automatically released. But in C++, the auto (or stack-based) variable can be an object with a destructor, which the compiler automatically calls for you -- even if an exception is thrown. In this author's opinion, this is the single most important feature of C++ (and one of several reasons why Java will never compare). Class destructors (along with constructors) allow the programmer to implement the 'resource acquisition is initialization' pattern described by Bjarne Stroustrup. In other words, when an object is constructed, some resource (like a block of memory, or an open file access path, or even a window) is acquired (or an exception is thrown if the acquisition failed), and upon destruction, the resource is released. This is how the STL classes work -- just declare a `map<string, string>` object and you have an instant symbol table (for example, if you were implementing environment variables), and at the end of the scope in which the variable was declared, the entire map, strings and all, gets destroyed.

### Resource Leaks

The importance of robust resource management is illustrated by the consequences of not implementing it. The example that follows, while disappointingly trivial in impact, nonetheless merits attention, because it occurs in a shipping piece of code[1].

/\*

This routine is the low level routine used by the `SendODOCToSelf` routine. It gets passed the list of files (in an `AEDescList`) to be sent as the data for the 'odoc', builds up the event and sends off the event.

It is broken out from `SendODOCToSelf` so that a `SendODOCListToSelf` could easily be written and it could then call this routine - but that is left as an exercise to the reader.

```

Read the comments in the code for the order and details
*/
void _SendDocsToSelf (AEDescList *aliasList)
{
    OSErrerr;
    AEAddressDesctheTarget;
    AppleEventopenDocAE, replyAE;

    /*
    First we create the target for the event.  We call another
    utility routine for creating the target.
    */
    err = GetTargetFromSelf(&theTarget);
    if (err == noErr) {
        /* Next we create the Apple event that will later get sent. */
        err = AECreatAppleEvent(kCoreEventClass, kAEOpenDocuments, &theTarget,
        kAutoGenerateReturnID, kAnyTransactionID, &openDocAE);

        if (err == noErr) {
            /* Now add the aliasDescList to the openDocAE */
            err = AEPutParamDesc(&openDocAE, keyDirectObject, aliasList);

            if (err == noErr) {
                /*
                and finally send the event
                Since we are sending to ourselves, no need for reply.
                */
                err = AESend(&openDocAE, &replyAE, kAENoReply + kAECanInteract,
                kAENormalPriority, 3600, NULL, NULL);

                /*
                NOTE: Since we are not requesting a reply, we do not need to
                need to dispose of the replyAE.  It is there simply as a
                placeholder.
                */
            }

            /*
            Dispose of the aliasList descriptor
            We do this instead of the caller since it needs to be done
            before disposing the AEVT
            */

```

```
err = AEDisposeDesc(aliasList);
}
```

```
/*and of course dispose of the openDoc AEVT itself*/
err = AEDisposeDesc(&openDocAE);
}
}
```

Notice how the multilevel if-structure makes the code harder to read. (The best alternative in C is to use a 'goto failed' construction.) Also notice that theTarget is never disposed -- it's allocated in GetTargetFromSelf() (but that's not obvious without viewing its source). Every time the user chooses a file using the menu interface, an AEResourceDesc carrying a PSN (i.e. an 8-byte Handle) is leaked. Finally, notice that in the (admittedly unlikely) event that GetTargetFromSelf() or AECreatAppleEvent() returns an error, aliasList doesn't get disposed either (which the comments indicate it should be). The code is corrected (and the comments removed for readability) as follows:

```
void _SendDocsToSelf (AEDescList *aliasList)
{
    OSErrerr;
    AEResourceDesctheTarget;
    AppleEventopenDocAE, replyAE;

    err = GetTargetFromSelf(&theTarget);
    if (err == noErr) {
        err = AECreatAppleEvent(kCoreEventClass, kAEOpenDocuments, &theTarget,
            kAutoGenerateReturnID, kAnyTransactionID, &openDocAE);

        if (err == noErr) {
            err = AEPutParamDesc(&openDocAE, keyDirectObject, aliasList);

            if (err == noErr) {
                err = AEResourceDesc(&openDocAE, &replyAE, kAENoReply + kAECanInteract,
                    kAENormalPriority, 3600, NULL, NULL);
            }
            err = AEDisposeDesc(aliasList);
        }
        err = AEDisposeDesc(&openDocAE);
    }
    err = AEDisposeDesc(&theTarget);
}
```

```
err = AEDisposeDesc(aliasList);  
}
```

Stack-based objects are guaranteed to be destroyed (provided the program doesn't terminate abruptly due to `abort()`, `ExitToShell()`, or a crash, for example) and thus are suited to further guarantee the destruction (or other release) of dynamically-existing (i.e. non-stack-based) objects, like `AEDescs`. (The `AEDesc` itself is an automatic struct, but the contents of the `dataHandle` member are dynamically allocated.) Here's a way that an `AEDesc` might be wrapped by a class for stack-based use:

```
static AEDesc gNullDesc = {typeNULL, NULL};
```

```
class StAEDesc {  
public:  
StAEDesc() : myAEDesc(gNullDesc) {}  
StAEDesc(const AEDesc& inAEDesc) : myAEDesc(inAEDesc) {}  
~StAEDesc() { AEDisposeDesc(&myAEDesc); }
```

```
AEDesc myAEDesc; // public member for simplicity of demonstration only  
};
```

```
typedef StAEDesc StAEDescList, StAEAddressDesc, StAppleEvent;
```

Here's how we might use it:

```
void _SendDocsToSelf (AEDescList *aliasList)  
{  
StAEDescList stAliasList(*aliasList);  
StAEAddressDesc theTarget;
```

```
OSErr err = GetTargetFromSelf(&theTarget.myAEDesc);  
if (err == noErr) {  
StAppleEvent openDocAE;
```

```
err = AECreatAppleEvent(kCoreEventClass, kAEOpenDocuments,  
&theTarget.myAEDesc,  
kAutoGenerateReturnID, kAnyTransactionID, &openDocAE.myAEDesc);
```

```
if (err == noErr) {  
err = AEPutParamDesc(&openDocAE.myAEDesc, keyDirectObject, aliasList);
```

```

if (err == noErr) {
    AppleEvent replyAE;
    err = AESend(&openDocAE.myAEDesc, &replyAE, kAENoReply + kAECanInteract,
    kAENormalPriority, 3600, NULL, NULL);
}
}
}
}
}

```

Now that the code is exception-safe, we can dispense with the if-fortress:

```

void _SendDocsToSelf (AEDescList *aliasList)
{
    StAEDescList stAliasList(*aliasList);
    StAEAddressDesctheTarget;

    OSerr err = GetTargetFromSelf(&theTarget.myAEDesc);
    // Normally we'd ThrowIfOSerr_(err), but this function doesn't recover
    // from errors and the caller doesn't catch exceptions.
    if (err) return;

    StAppleEvent openDocAE;
    err = AECREATEAppleEvent(kCoreEventClass, kAEOpenDocuments,
    &theTarget.myAEDesc,
    kAutoGenerateReturnID, kAnyTransactionID, &openDocAE.myAEDesc);
    if (err) return;

    err = AEPutParamDesc(&openDocAE.myAEDesc, keyDirectObject, aliasList);
    if (err) return;

    AppleEvent replyAE;
    err = AESend(&openDocAE.myAEDesc, &replyAE, kAENoReply + kAECanInteract,
    kAENormalPriority, 3600, NULL, NULL);
}

```

And actually using exceptions:

```

StAEDesc& StAEDesc::operator=(const AEDesc& inAEDesc)
{
    AEDisposeDesc(myAEDesc);
    myAEDesc = inAEDesc;
    return *this;
}

```

```

}

AEDesc GetSelfTarget()
{
AEDesc desc;
OSErr err = GetTargetFromSelf(&desc);
ThrowIfOSErr_(err);
return desc;
}

AEDesc CreateAppleEvent(AEEventClass inClass, AEEventID inID,
const AEAddressDesc* inTarget,
AEReturnID inReturnID = kAutoGenerateReturnID,
AETransactionID inTransactionID = kAnyTransactionID)
{
AEDesc desc;
OSErr err = AECreatAppleEvent(inClass, inID, &inTarget,
inReturnID, inTransactionID, &desc);
ThrowIfOSErr_(err);
return desc;
}

StAEDesc::StAEDesc(AEEventClass inClass, AEEventID inID,
const StAEAddressDesc& inTarget,
AEReturnID inReturnID = kAutoGenerateReturnID,
AETransactionID inTransactionID = kAnyTransactionID)
: myAEDesc(CreateAppleEvent(inClass, inID, &inTarget.myAEDesc, inReturnID,
inTransactionID))
{
}

// StAEDesc::PutParamDesc() and SendWithNoReply() are left as an exercise.

void _SendDocsToSelf (AEDescList *aliasList)
{
StAEDescList stAliasList(*aliasList);

try {
StAEAddressDesc theTarget = GetSelfTarget();

StAppleEvent openDocAE(kCoreEventClass, kAEOpenDocuments, theTarget);

```

```

openDocAE.PutParamDesc(keyDirectObject, aliasList);

openDocAE.SendWithNoReply(kAECanInteract, kAENormalPriority, 3600, NULL,
NULL);
} catch (...) {
}
}
}

```

## A Limitation of Stack-based Objects

The function's code could be reduced by yet another line if we made `GetSelfTarget()` return `StAEAddressDesc` and eliminated the temporary variable `theTarget`:

```
StAppleEvent openDocAE(kCoreEventClass, kAEOpenDocuments, GetSelfTarget());
```

But there is one very big problem with this: When a function returns a result, the value gets copied. The compiler-generated copy constructor performs a byte-wise copy of the `AEDesc` member, and then the original is destructed, disposing the `AEDesc`'s `dataHandle` and setting it to `{typeNull, NULL}`. But the copy retains a dangling `Handle`, while otherwise valid (and non-null). Using this damaged `AEDesc` is bad enough; the destruction guaranteed to follow will be worse, if the program hasn't crashed already. While it's possible that a compiler might optimize away the copy operation in this example, we still run the risk of the programmer assigning the result to a variable elsewhere.

## Smart Pointers

This snarl could be resolved by defining the copy constructor to call `AEDuplicateDesc()`, but at great expense performance-wise. Another approach is to simply make the class noncopyable by defining a private copy constructor. This prevents the mishap described above, but how does one get a noncopyable object outside of the scope it was created in? The answer: You can't. The only way to keep an object beyond its lexical scope without copying it is to create the object with dynamically-scoped duration (using the `'new'` operator). An object with lexically-scoped duration perishes when the variable containing it goes out of scope. But doesn't using dynamic allocation (with its required explicit deallocation) defeat our aim of guaranteed destruction? Not if the dynamically allocated object is wrapped in a stack-based object. That, in turn, might seem to reintroduce the lexical scoping limitation, except that this new class is

copyable. So, how can we design a wrapper class that can be copied without copying the thing it wraps, but making sure to dispose it exactly once? A pointer is copied without copying the thing it points to. A 'smart pointer' would also take care of the resource management. By overloading some operators, we can create a class with a pointer member which resembles a pointer itself.

```
// StAEDesc ought to be renamed since we don't use it from the stack
class AEDescPtr {
public:
AEDescPtr() : myPtr(NULL) {} // Null initialization isn't required, but
why not?
AEDescPtr(StAEDesc* inPtr) : myPtr(inPtr) {}

AEDescPtr& operator=(StAEDesc* inPtr) { myPtr = inPtr; return *this; }
operator StAEDesc*() const { return myPtr; }
StAEDesc* operator->() const { return myPtr; }
StAEDesc& operator*() const { return *myPtr; }
private:
StAEDesc* myPtr;
};
```

Now for the tricky part. First, we must choose copying semantics, either shared-ownership or passed-ownership. With shared-ownership semantics the item pointed to is shared (i.e. exists as a single copy) until all pointer objects are destroyed, at which point the shared object is destroyed too. When using passed-ownership semantics, copying a pointer object erases it -- the copy becomes the sole owner of the owned object, which is destroyed along with its owner. Keeping track of shared ownership costs in overhead, so passed ownership is more efficient -- but it's not as versatile. While it suffices for our example above (returning an object from a function) it doesn't work if the data flow is not strictly linear (with no branching). To illustrate: If AEDescPtr uses passed-ownership semantics and a AEDescPtr object (pointing to a StAEDesc object) is passed by value to a function, or even if passed by reference and the function makes a copy, when the function returns, the calling code will have no way to access the StAEDesc object, which will have been destroyed anyway.

The implementation details of a smart pointer class are beyond the scope of this paper. The reader is invited to consult an appropriate reference or peruse available source code. [One implementation of shared-ownership is provided on the MacHack 17 CD, in Joshua Juran's Genesis library. The C++

Standard Library provides passed-ownership with `auto_ptr<>.`]

Proceeding with our intention to eliminate the `Target`:

```
AEDescPtr GetSelfTarget()
```

```
{  
    AEDesc desc;  
    OSerr err = GetTargetFromSelf(&desc);  
    ThrowIfOSerr_(err);  
    return new StAEDesc(desc);  
}
```

```
StAEDesc::StAEDesc(AEEventClass inClass, AEEventID inID,  
    const AEDescPtr& inTarget,  
    AEReturnID inReturnID = kAutoGenerateReturnID,  
    AETransactionID inTransactionID = kAnyTransactionID)  
: myAEDesc(CreateAppleEvent(inClass, inID, &inTarget->myAEDesc,  
    inReturnID, inTransactionID))
```

```
{  
}
```

```
void _SendDocsToSelf (AEDescList *aliasList)
```

```
{  
    StAEDescList stAliasList(*aliasList);
```

```
    try {
```

```
        StAppleEvent openDocAE(kCoreEventClass, kAEOpenDocuments, GetSelfTarget());
```

```
        openDocAE.PutParamDesc(keyDirectObject, aliasList);
```

```
        openDocAE.SendWithNoReply(kAECanInteract, kAENormalPriority, 3600, NULL,  
        NULL);
```

```
    } catch (...) {
```

```
    }  
}
```

## A Smart Pointer Class Template

The astute reader may observe that the smart pointer class presented here has nothing to do with generic programming. There is one more step remaining -- to transform the class into a class template:

```

template <class Thing>
class SmartPtr {
public:
SmartPtr() : myPtr(NULL) {} // Null initialization isn't required, but
why not?
SmartPtr(Thing* inPtr) : myPtr(inPtr) {}
SmartPtr(const SmartPtr<Thing>& other); // copy constructor
~SmartPtr(); // destructor

SmartPtr<Thing>& operator=(const SmartPtr<Thing>& other); // copy assignment
SmartPtr<Thing>& operator=(Thing* inPtr) { myPtr = inPtr; return *this; }
operator Thing*() const { return myPtr; }
Thing* operator->() const { return myPtr; }
Thing& operator*() const { return *myPtr; }
private:
Thing* myPtr;
};

typedef SmartPtr<StAEDesc> AEDescPtr;

```

The template SmartPtr can be used with any type that requires no explicit destruction (i.e the call of its own destructor (if any) is sufficient).

## Other Examples of Generic Programming

-----

### Windowing

One of the problems faced in the development of Pedestal (a Mac application framework in C++) is the management of windows. Designing a generic solution requires examining the problem space for generic structures and algorithms. The developer's analysis follows.

In an application (and therefore in an application framework), there may be different kinds of windows, and different policies pertaining to them. For example, document windows (of which there can be many) have associated document state, and certain policy requirements (closure of the window may be cancelled or involve saving the document first), whereas an About window (of which there can be only one) has no state of its own beyond the window itself (e.g. location and visibility).

Therefore: Window is a class implementing behavior which is common to all windows. A window Client class must be defined by the application for each kind of content that will appear in a window. Client objects are responsible for populating the window with the necessary controls and other visual elements, storing references to them, and manipulating them as appropriate. A Director class template implements windowing policy (whether there can be multiple windows) and maintains a set of windows of a particular type. The application needs a distinct Director class for each type of window, even if they have the same windowing policy, but can instantiate them from the same template. An Agent acts as a link between a Client and its Director. The Agent class template implements closure requests. Possible closure policies are always-close (e.g. About), save-first (CodeWarrior projects), and ask-first (most documents). The same Agent template can be used for different window types with different directors.

## Bibliography

-----

[1] DropShell 2.0 (DSUtils.c), by Leonard Rosenthol, Stephan Somogyi, and Marshall Clow