New Technical Notes
Macintosh

Developer Support

My Life as a PascalObject
Platforms & Tools **M.PT.PascalObjects**

Written by: Kent Sandvik & Mark Bjerke April 1991

This Technical Note discusses the `PascalObject` base class, used, for instance, with MacApp programming. The Technical Note describes how to write PascalObject derived classes that work with both Object Pascal and C++ code linking. It also describes the current restrictions and bugs with writing C++ code using `PascalObject` as the base class. This Technical Note is based on MacApp 2.0(.1), MPW 3.2 and MPW C++ 3.1.

Introduction

PascalObjects are useful. If you use `PascalObject` as the base class you are able to use the object libraries with both Object Pascal code and C++ code. MacApp 3.0 is written in C++ using `PascalObject` as the base class. Code written in Object Pascal is link compatible with C++ as long as any C++ objects which are accessed are derived from a PascalObject base class.

There are C++ semantics that will not work with Object Pascal, and there are semantic limitations with Object Pascal that the C++ programmer should be aware of if the code is to be used with Object Pascal.

An important issue is the class interfaces, C++ and Object Pascal. Any C++ language constructs that don't work in Object Pascal should not be present in the Object C++ header files. For instance there is no notion of `const` in Object Pascal, so a C++ declaration using `const` would be misleading because the value could be changed (from Object Pascal). Note that code inside the class methods [11] (that are not accessible for the class user) does not need to conform to Object Pascal limitations, with some exceptions which we try to cover in this Technical Note. Be careful when defining interface definitions for both Object Pascal and C++ use. Avoid any C++ syntax or semantics which can't be mapped to Object Pascal, if the intention is to produce libraries that will work with both C++ and Object Pascal.

PascalObjects behave like classes defined in Object Pascal, with the same kind of relocatable, handle-based objects and the same kind of method lookup tables. In the case of MacApp the TObject class is inherited from the PascalObject base class. Do not confuse PascalObjects with the HandleObject base class, even if both use handles they differ at the base class level.

---

[11] We use the Smalltalk terminology in this Technical Note, where *method* loosely corresponds to the C++ definition *member function*.

How to Write C++ PascalObject Classes

Constructors/Destructors in the PascalObject Domain

In the wonderful world of C++ programming the constructor takes arguments for values that are needed for the creation of the instance of the class. The C++ programmer is also able to pass arguments up to the parent classes if needed. In C++ the construction of a class starts with the base class, and each constructor down the inheritance chain is run.

This will not work with Object Pascal code. The construction of a PascalObject is usually an assignment of memory, and any possible initialization is done in a special method called `IClassName` (where `ClassName` is substituted to the class name, for example `IMyApplication`).

So if the C++ programmer assumes that everyone in the galaxy will use C++ notation for signalling information to the construction of the class, she/he is wrong. The policy is to create initialization methods for each class, and inside each method call the parent initialization method. This way all the initialization methods all the way to the top level are called. Note also that the order of calling base class constructors is implementation dependent, whereas in C++ the base constructors are called first, and the child constructors later. It usually makes sense to define this as well with PascalObject hierarchies, so the class library user could rely on the order of class initializations.

This is also true of destructors: instead of calling the destructor you need to define a method call `Free` (when using PascalObjects). You also need to call the method `Free` yourself, instead of relying as in the C++ world that the destructor will automatically work when the object goes out of scope.

Here's a simple MacApp example:

```
class TFooApplication: public TApplication
{
  public:
      virtual pascal IFooApplication(void);     // this is our "constructor"
      virtual pascal Free(void);                // this is our "destructor"
// other methods and fields...
};

void TFooApplication::IFooApplication(void)
{
      this->IApplication(kFileType);      // call the base class constructor
      // do own stuff...
}

void TFooApplication::Free(void)
{
      inherited::Free();                  // call base class functions above
      // do own stuff...
}
```

pascal Keyword, virtual/OVERRIDE

Always define every method with Pascal calling conventions, as in the following example:

```
class TFoo: public TObject
{
  public:
     virtual pascal void Reset(void);
// ....
}
```

This means that you are able to call the function from Object Pascal. If you define the method virtual, and it's PascalObject based, then you are able to override the method from Object Pascal using the `OVERRIDE` keyword.

Function overloading as such does not work from Object Pascal, because Object Pascal does not have the notion of function name mangling.

Stack Objects (Objects on the Stack)

Object Pascal is not capable of defining objects on the stack . PascalObjects (as handles) are heap based. The compiler also complains if you try to define stack-based PascalObjects with C++.

A C++ example of a object declared on the stack is shown below:

```
void foo(void)
{
TDaffyDuck myDaffyDuck;                    // declared on the stack

      myDaffyDuck.ShootMeNow(kDuckSeason);
//    continue with the function…
}
```

private/protected/public in C++ and Object Pascal

C++ has access control of methods and fields inside classes, using `private`, `protected` and `public` as keywords. Field checks are done during compilation time, not linktime or runtime. Because the C++ modules are compiled separately from the Object Pascal modules, access control is not active from the Pascal code. It does not hurt to specify access control for C++ classes - quite the contrary - but Object Pascal code is able to access any method or function inside the C++ class.

So beware that any dependencies of the C++ class access control definitions will be broken under Object Pascal if you use PascalObject as the base class.

Str255 - Pascal Strings

Pascal strings are the common method for passing strings between multilanguage modules. (There are exceptions in the MacOS Toolbox.) If a method or function sends or receives a string, it should be declared as a Pascal String (`Str255`, `Str63`, `Str31`…). The MPW libraries have functions for changing, copying and comparing both Pascal strings and C strings (null terminated strings).

Breakpoint Information (%_BP and %_EP)

If you want to generate breakpoint information in the object files from C++, specify the `'-trace on'` flag to the C++ compiler, or use the new `#pragma trace on` and `#pragma trace off` switches inside the code. This only works with MPW 3.2 C and future releases. This is equivalent to the `$D++` and `$D-` statements in Object Pascal.

Default Arguments

Object Pascal does not have the syntax for defining default argument values inside function prototypes. This means that you can't define default argument values in the `FUNCTION` and `PROCEDURE` methods in the Unit header files for Object Pascal (as you can do with C++ methods). If the user wants to change these default values there's no way to achieve this with Object Pascal.

Public Base Class

Classes that inherit from PascalObject should be defined with a `public` interface because the operator `new` is overloaded. An example of this looks like:

```
class TFoo: public PascalObject
{
// class contents
};
```

Inlining

Inlining of C++ methods works with the C++ header files, even if the semantics is not supported with Object Pascal header files. We assume that the inlining is used for C++ PascalObject class construction, where the classes are implemented in C++.

General C/Pascal Issues

Try to write code that works and functions well under both Object Pascal and C++. All the rules concerning Pascal and C code reusability are true for writing C++ and Object Pascal object libraries for as well. For instance avoid function and variable names with changes in capital case only, for example Foo and foo are identical function names under Pascal.

Also try to use the new "call by reference" notation of C++ (`&` - resembles VAR in Pascal) when passing references to variables to functions, instead of using the normal pointer notation. This way developers can write similar looking code for calling functions with values.

You also need to create Unit header files for Object Pascal use with the class definitions in Object Pascal. Remember to define any interface constants in Object Pascal that are defined as enums in the C++ class.

Bugs and Limitations with PascalObjects

General

Please consult the latest MPW and C++ Release Notes for the latest information about known bugs and limitations.

Pure Virtual Functions

Pure virtual functions are allowed in C++ PascalObject hierarchies, as long as these functions are defined. The compiler will complain if the programmer tries to create an instance of an abstract class (the class that contains the pure virtual function) with new. However the linker does not like that the pure virtual function is not defined, so when the programmer links object modules with classes containing pure virtual functions which are not defined the linker will complain.

The following code example shows the problem and the workaround:

```
class TAbstract : public PascalObject {   // pure abstract class
      virtual void Method() =0;           // define pure virtual
};

class TDerived  : public TAbstract    {   // not abstract because
     virtual void Method() {};            // of this definition (non-pure)
};

void TAbstract::Method()                  // you need to define
{                                         // this function for the linker
// dummy, this is abstract anyway
}

TAbstract* noWay;
TDerived*  okClass;

main()
{
      noWay = new TAbstract;              // compiler will complain!

    okClass = new TDerived;               // OK, not an abstract class

    return 0;
}
```

Pointers to PascalObject Members

You may not have pointers to PascalObject member functions if you are using the MPW 3.1 Link tool. You must use the new MPW 3.2 linker and a new MPW 3.2 C++ compiler for using this feature. The old way of doing method dispatch was broken, but it is fixed in the new optimized dispatch code. (See `UObject.a` in MacApp 2.0.1.)

Multiple Inheritance

Because PascalObjects method lookup is based on Object Pascal method lookup tables (instead of normal C++ vtables), Multiple Inheritance does not work with PascalObjects.

Problems with including PascalObject Runtime Support

This bug has to do with not including the call to `_PGM` which brings in the segment containing the `%_SelProcs` and the method tables used in PascalObject method dispatch. This is flagged (for including this runtime support) when CFront sees a use of a member function belonging to a (type derived from) PascalObject. If it compiles `main()` before it sees a use of the member function then the call to `_PGM` will not be included. Note that even a call to operator `new` inside of `main()` does not do the trick for PascalObjects with constructors because the constructor calls operator `new`, not `main()`.

The way to get around this is to invoke operator `new` on a dummy object with no constructor (much like anti-dead stripping code). Remember that this is only necessary in cases where there is no code before `main()` referencing a PascalObject method. Below is code which reproduces this problem. Note that the call to operator `'new'` in `main` normally would be enough except that class `foo` has a constructor.

```
class foo : PascalObject {
      public:
            foo(void);
            virtual void meth1(void);
};

void main()
{
      foo* afoo = new foo;
}

foo::foo(void) { ; }
void foo::meth1(void) { ; }

void non_member_func(foo* theClass)
{
      theClass->meth1();
}
```

`pascal` Keyword

The `pascal` keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a Pascal-style function. The C compiler currently misinterprets the C function as a Pascal-style function and the function result is lost.

Problems with returning Structs/Objects

Methods may not return structs/objects or anything that requires the C compiler to push and address for the called routine to copy return values. This will break the method dispatch which is expecting a handle (`this->` pointer) to the object as the last thing pushed.

Alignment Problems with Arrays

There is an alignment bug involving the size the C compiler calculates for certain PascalObjects and the actual size CFront allocates for such objects using operator `new`. Basically if an object has a multidimensional array of a byte sized quantity (char, Boolean, etc) whose total size in bytes is odd, then pad bytes are added by CFront and the C compiler and everything is fine. Now if you have two such arrays declared (see example) back to back, then CFront makes the mistake of not adding the pad bytes.This results in the C compiler accessing memory that is off the end of the object in question (since the `new` was done with the size parameter too small). For example:

```
class foo  {          // CFront generates size as 20 - C compiler
                     // uses 22
 char bytes1[3][3];
 char bytes2[3][3];
 short x;             // Access of this field falls off end of object
};
```

set_new_handler()

To give more control over memory allocation, you could define an extern pointer from `set_new_handler` (`_new_handler`) to be called if operator `new` fails. This is not supported in the operator `new` used for `PascalObject` because the code for operator `new` fails to make the call to the user handler through the function pointer set with `set_new_handler()`.

Call of the Wrong Member Function

There is a bug that involves calling the wrong member function in the case of PascalObjects whose names differ only in case (for example `class Foo` and `class foo`).

This example shows the problem:

```
class foo : public PascalObject            // note the all lowercase name
{
      public:
      virtual pascal void foobar(void);
};

class Foo : public PascalObject            // n.b Foo
{
      public:
      virtual pascal void foobar(void);
};
```

```
pascal void foo::foobar(void) {}
pascal void Foo::foobar(void) {}


main()
{
foo *afoo = new foo;
Foo *aFoo = new Foo;

afoo->foobar();
aFoo->foobar();               // Calls the wrong 'foobar()'
}
```

Information

For more inside information about PascalObjects, check the MacApp files `UObject.a`, which describes how method lookup is handled, and `UObject.Global.p`, which shows how `NEW` is implemented under Object Pascal.

Conclusion

Using PascalObjects as the base class for your class libraries will get you many Object Pascal programmers as new friends. If you use TObject (from MacApp) as your base class library (subclass of PascalObject), you get a lot of meta-information and meta-methods for free. And PascalObject classes are handle based, so you get less memory allocation problems on small memory configuration Macintosh computers.

Further Reference:

- M.PT.PascalToCProcParams
- M.PT.HandleObjects
- MPW C++ 3.1 Reference
- MPW C++ 3.1 Release Notes
- *C++ Programming with MacApp* , Wilson, Rosenstein, Shafer, Addison&Wesley
- *The Annotated C++ Reference Manual,* Ellis&Stroustrup, Addison&Wesley