quantum LeaPs
innovating embedded systems

"C+" 3.0
Programmer's Manual

Document Revision C

May 2007

Miro Samek, Ph.D.

quantum LeaPs™, LLC

www.quantum-leaps.com

Copyright © 2002-2007 quantum LeaPs, LLC

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with this copyright notice being preserved. A copy of the license is available from the Free Software Foundation at: www.gnu.org/copyleft/fdl.html.

Table of Contents

1 Introduction 1
1.1 Licensing 1
2 Getting Started with "C+" 2
2.1 Installation 2
2.2 Borland Turbo C++ 1.01 3
2.3 GNU Make 3
2.4 Compiling the "C+" Library 3
2.5 Building the Test Application 4
2.6 Running the Test 4
3 "C+" Overview 5
3.1 Encapsulation 5
3.2 Inheritance 6
3.3 Polymorphism 8
3.4 Costs and Overhead 13
4 An Annotated Example 14
4.1 Subclassing Shape 17
4.2 Executing the Test 18
5 Summary 19
6 References 20
7 Contact Information 20

● Copyright © 2002-2007 Quantum Leaps, LLC ●

C makes it easy to shoot yourself in the foot; C++ makes it harder,
but when you do, it blows away your whole leg.
—Bjarne Stroustrup

1 Introduction

Object-oriented programming (OOP) is *not* the use of a particular language or a tool. It is rather a way of design based on the three fundamental design meta-patterns:

- **Encapsulation**—the ability to package data and functions into classes
- **Inheritance**—the ability to define new classes based on existing classes in order to obtain code reuse and code organization
- **Polymorphism**—the ability to substitute objects of matching interfaces for one another at runtime

Although these meta-patterns are traditionally associated with object-oriented languages, such as Smalltalk, C++, or Java, you can implement them in almost any programming language including C[1] and even assembly[2]. Indeed, as Frederick Brooks [Brooks 95] observes:

> *... any of these disciplines [object-oriented meta-patterns] can be had without taking the whole Smalltalk or C++ package—many of them predated object-oriented technology.*

In fact, virtually any larger software system, regardless of implementation language, uses the meta-patterns of "Abstraction", "Inheritance", or "Polymorphism" in some form or another. Easy to identify examples include OSF/Motif (the popular, object-oriented graphical user interface), and Java Native Interface, both of which are implemented in C. You don't need to look far to find many more such examples.

OOP in an object-oriented language is straightforward, because such a language natively supports the three fundamental meta-patterns. However, you can also implement these patterns in other languages, such as C, as sets of conventions and idioms. I call my set of such conventions and idioms "**C+**" [Samek 02]. The main objective of "C+" is to achieve performance and maintainability equivalent to the C++ object model. In fact, "C+" is, to a large degree, an explicit re-implementation of the C++ object model, as described, for example in [Lippman 96].

1.1 Licensing

This software may be distributed and modified under the terms of the BSD open source license, which is provided below.

> *Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:*
>
> *Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.*

[1] The original cfront C++ compiler translated C++ into C, which is perhaps the most convincing argument that all C++ constructs can be in fact implemented in plain C.

[2] For example, Borland Turbo Assembler v4.0 [Borland 93] directly supports abstraction, inheritance, and polymorphism and therefore can be considered an object-oriented language.

Neither the name of the Quantum Leaps nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL QUANTUM LEAPS OR OTHER CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NOTE: Not all products of Quantum Leaps are offered under the BSD open source license. In fact, most software available from Quantum Leaps is licensed under the terms of the GNU General Public License (GPL) as well as commercially under the dual-licensing business model. Please refer to Quantum Leaps licensing information available online.

4

2 Getting Started with "C+"

2.1 Installation

"C+" is distributed in a ZIP archive, that can be installed (unzipped) into any folder. After the installation, you should end up with the following files and directories (not all files are shown):

```
<cplus/>                - directory for "C+"
  |
  +-cplus_manual.pdf    - this file
  |
  +-cplus/              - "C+" directory
  | +-include/          - "C+" public header files
  | +-source/           - "C+" implementation
  | |
  | +-dosb1/            - "C+" port to DOS with Borland Turbo C++ 1.01
  |   +-Makefile        - GNU make file to build "C+" library
  |   +-cplus.mak       - Borland Turbo C++ 1.01 make file to build "C+" library
  |   +-release/        - contains cplus.lib (release version)
  |
  |
  +-cplustst/           - Shapes sample application to test "C+"
  | +-include/          - platform-independent application header files
  | +-source/           - platform-independent application implementation
  | +-dosb1/            - port of the application to DOS with Borland Turbo C++ 1.01
  |   +-Makefile        - GNU make file to build cplustst executable
  |   +-cplus.mak       - Borland Turbo C++ 1.01 make file to build cplustst executable
  |   +-link.rsp        - Turbo Linker response file
  |   +-release/        - contains cplustst.exe (DOS version)
```

Listing 1 "C+" distribution directory and file structure

2.2 Borland Turbo C++ 1.01

Borland recently donated its legacy Borland Turbo C++ 1.01 compiler for free downloads from the Borland "Museum" (http://bdn.borland.com/article/0,1410,21751,00.html). In addition, Borland provides a scanned image of the original "Turbo C++ User's Guide" in the PDF format (http://www.borland.com/cbuilder/tsuite/). The User's Guide is for Borland C++ v3.0, but still largely applies to version 1.01.

The free Borland Turbo C++ 1.01 suite is a "good enough" tool for evaluating, learning, and experimenting with "C+ code and this "C+ Programmer's Manual" provides support for Turbo C++ 1.01.

To install Borland Turbo C++ 1.01, unzip the TCPP101.ZIP archive from the Borland "Museum" onto your hard drive. Run the INSTALL.EXE program and follow the installation instructions to install the software. For compatibility with the provided make files, you should install the compiler into the directory C:\tools\tcpp101. If you choose a different directory, you'll need to modify the make files and the linker response files provided in the "C+" distribution.

2.3 GNU Make

This software is distributed with makefiles conforming to the GNU make standard, which in turn conforms to Section 6.2 of IEEE Standard 1003.2-1992 (POSIX.2). The GNU make utility is freely available for Windows and UNIX™ platforms, most notably Linux™.

If you work on Windows, you basically have two choices for using GNU make. The first one is with Cygwin from Red Hat, which is a rather heavyweight UNIX emulation for Win32. The other, lightweight version of GNU make is available from the MinGW project (Minimal GNU for Windows), which is a native implementation of POSIX API for Win32. The basic MinGW runtime has been placed in the public domain (see MinGW licensing terms at http://www.mingw.org/licensing.shtml), while utilities such as GNU make are licensed under the terms of the GPL.

You can download the MinGW GNU make either directly from the MinGW download site (http://www.mingw.org/download.shtml), or from Quantum Leaps mirror at http://www.quantum-leaps.com/downloads/tools.htm#GNUmake). The code described here has been actually tested only with the MinGW GNU make and might not work correctly with the Cygwin GNU make.

> **NOTE:** The mingw32-make.exe make invoking DOS tools, such as the Borland Turbo C++ 1.01 compiler, has been tested on Windows XP. However, on earlier versions of Windows (such as 98/ME), the mingw32-make.exe make utility didn't work correctly with the 16-bit DOS tools. Therefore, the QEP/C distribution includes both GNU Makefiles and *.mak files for the make utility shipping with Borland Turbo C++ 1.01.

2.4 Compiling the "C+" Library

"C+"+ is deployed as a static library that you link with your application. To build the "C+" library with Borland Turbo C++ 1.01, you open a console window on a Windows PC, change directory to <cplus>\cplus\dosb1, and type at the command prompt the following command:

```
mingw32-make.exe

or

C:\tools\tcpp101\bin\make –fcplus.mak
```

This command line and the make file Makefile (cplus.mak) assume that Borland Turbo C++ 1.01 has been installed in the directory C:\tools\tcpp101. You need to adjust both if you've installed Turbo C++ 1.01 into a different directory.

The make process should produce the "C+" library in the location: <cplus>\cplus\dosb1\release\cplus.lib.

2.5 Building the Test Application

This "C+" distribution comes with the classical "Shapes" example. The Turbo C++ 1.01 make file for the test application is located in <cplus>\cplustst\dosb1\cplustst.mak. You invoke the make file from the <cplus>\cplustst\dosb1 directory through the following command:

```
mingw32-make.exe

or

C:\tools\tcpp101\bin\make –fcplustst.mak
```

This command line, the make file Makefile (cplus.mak), and the Turbo Linker response file link.rsp assume that Borland Turbo C++ 1.01 has been installed in the directory C:\tools\tcpp101. You need to adjust all these elements if you've installed Turbo C++ 1.01 into a different directory.

2.6 Running the Test

The latter make process should produce the test application in the location: <cplus>\cplustst\dosb1\release\cplustst.exe.

NOTE: the executable is provided as part of the build, so even if you don't install Borland Turbo C++ 1.01 and don't build the application from source files, you still can run the executable.

You can run this executable from any Windows PC. The application outputs the current status of each dining philosopher to the console. You can run this executable from any command prompt by typing:

```
release\cplustst.exe
```

Here is the expected output generated by the test application:

```
release\cplustst.exe
name="Circle", area()=3.14, scale(2), name="Circle", area()=12.57,
name="Rectangle-0", area()=0.00, scale(2), name="Rectangle-0", area()=0.00,
name="Rectangle-1", area()=1.00, scale(2), name="Rectangle-1", area()=4.00,
name="Rectangle-2", area()=2.00, scale(2), name="Rectangle-2", area()=8.00,
name="Rectangle-3", area()=3.00, scale(2), name="Rectangle-3", area()=12.00,
```

The internals of the test application will be discussed later in this manual.

3 "C+" Overview

3.1 Encapsulation

As a C programmer, you must have already used *abstract data types* (ADTs). Take for example the family of functions fopen(), fclose(), fread(), fwrite(), fseek(), ftell(), and so on, from the standard C run-time library. All these functions operate on objects of type FILE. The FILE ADT is **encapsulated** so that the clients have no need to access the FILE's internal attributes (have you ever looked at what's inside the FILE structure?). The only interface to FILE is through functions (methods), such as fopen(), fclose(), fread(), fwrite(), fseek(), ftell(), and so on. All these functions take a pointer to a FILE object as one of the arguments. You can think of the FILE structure and the associated functions that operate on it as the FILE **class**.

Let's quickly summarize the way in which the C run-time library implements the FILE class:

- Attributes of the class are defined with a C struct (the FILE struct).
- Methods of the class are defined as C functions. Each function takes a pointer to the attribute structure (FILE*) as an argument. Class methods typically follow a common naming convention (e.g., all FILE class methods start with f).
- Special methods are used for initializing and cleaning up the attribute structure (fopen() and fclose(), respectively). These methods play the roles of class constructor and destructor.

```
1: typedef struct StringTag String;
2: struct StringTag {
3:     char *buf__;                              /* private character buffer */
4: };
5:
6: void String_ctor1(String *me, char const *str);            /* public Ctor1 */
7: void String_ctor2(String *me, String *other);              /* public Ctor2 */
8: void String_xtor(String *me);                               /* public Xtor */
9: char const *String_toChar(String *me);                /* to-char conversion */
```

Listing 2 Declaration of String class.

Listing 2 declares String class and demonstrates how a coding convention can strengthen the association between the attributes and methods. Each class method starts with the common class prefix (String) and takes the pointer to the attribute structure (String *) as the first argument. In C+, this argument is consistently called me. In C++, me corresponds to the implicit this pointer. In C, the pointer must be explicit. I could have named that argument this in the analogy to C++ (which, in fact, was my first guess), but such a choice precludes using C classes in C++ because this is reserved in C++. The need for mixing C with C++ can easily arise when you want to share common code between C and C++ projects. Besides, me is shorter than this, and you will find yourself using many me->... constructs.

The next aspect that Listing 2 addresses with a coding convention is access control. In C, you cannot restrict the level of access permitted to a particular attribute or a method. All you can do is convey the intended level of protection in the name of an attribute or a method (which is typically better than merely indicating the intended level of access in a comment at the declaration point). This way, any unintentional access to class members is easy to detect during a code review. Most OO designs distinguish the following three levels of protection:

- **Private**—accessible only from within the class.
- **Protected**—accessible only by the class and its subclasses.

- **Public**—accessible to anyone (the default in C).

C+ convention is that the double-underscore suffix (foo__) indicates private attributes, and a single-underscore suffix (foo_, Foo_doSomething_()) indicates protected members. Public members do not require underscores (foo, Foo_doSomething()). You typically don't need to include private methods in the class interface (in the .H file), because you can hide such methods completely in the class implementation file (define them as static in the .C file).

Optionally, a class might provide one or more constructors and a destructor for initialization and cleanup, respectively. To distinguish these special methods, C+ uses base names ctor (Foo_ctor, Foo_ctor1) for constructors and xtor (Foo_xtor) for destructors. A C+ constructor takes the me argument to allow initialization of externally pre-allocated memory. The destructor takes only the me argument and returns void.

As in C++, you can allocate objects statically, dynamically (on the heap), or automatically (on the stack). However, because of C syntax limitations, generally you can't initialize objects at the definition point. For static objects, you can't invoke a constructor at all, because function calls aren't permitted in a static initializer. Automatic objects (objects allocated on the stack) must all be defined at the beginning of a block (just after the opening '{' brace). At this point, you generally do not have enough initialization information to call the appropriate constructor. Therefore, you often have to separate object allocation from initialization. Some objects might require destruction, and explicitly calling destructors for all objects when they become obsolete or go out of scope is a good programming practice. As described in Section 3.3 later in this manual, destructors can be polymorphic.

Exercise 1 Define three preprocessor macros—CLASS(class_), METHODS, and END_CLASS — so that the declaration of class String from Listing 2 can be rewritten as

```
CLASS(String)
    char *buf__;                                  /* private character buffer */
METHODS
    void String_ctor1(String *me, char const *str);         /* public Ctor1 */
    void String_ctor2(String *me, String *other);           /* public Ctor2 */
    void String_xtor(String *me);                            /* public Xtor */
    char const *String_toChar(String *me);            /* to-char conversion */
END_CLASS
```

3.2 Inheritance

Inheritance is a mechanism that defines new and more specialized classes in terms of existing classes. When a child class (**subclass**) derives from a parent class (**superclass**), the subclass then includes the definitions of all the attributes and methods that the superclass defines. Usually, the subclass extends the superclass by adding new attributes and methods. Objects that are instances of the subclass contain all data defined by both the subclass and its ancestor classes, and can perform all operations defined by both the subclass and its ancestor classes.

Example of Inheritance in C

Seasoned C programmers often intuitively arrive at designs that use inheritance. For example, in the original µC/OS Real-Time Kernel, Jean Labrosse defines a type OS_EVENT [Labrosse 92]. This abstraction captures a notion of an operating system event, such as a semaphore, a mailbox, or a message queue. The µC/OS clients never deal with OS_EVENT directly, because it is an abstract concept. Such an **abstract class** captures the commonality among inter-task synchronization mechanisms and enables uniform treatment of all operating system events.

The subsequent versions of μC/OS provide an interesting case study in the evolution of the OS_EVENT concept. In the original version (v1.10), no OS_EVENT methods exist, but rather the author replicates identical code for semaphores, mailboxes, and message queues. In MicroC/OS-II [Labrosse 99], OS_EVENT is fully factored out as a separate entity (class) with a constructor (OSEventWaitListInit()) and methods (OSEventTaskRdy(), OSEventTaskWait(), OSEventTaskTO()). The methods are subsequently reused in all specializations of OS_EVENT, such as semaphores, mailboxes, and message queues. This reuse significantly simplifies the code and makes it easier to port to different microprocessor architectures.

You can implement inheritance in C in a number of ways. The objective is to embed the parent attributes in the child so that you can invoke the parent's methods for the child instances as well. One of the techniques is to use the C preprocessor to define class attributes as a macro [Van Sickle 97]. Subclasses invoke the parent class attribute macro when defining their own attributes. C+ implements single inheritance by literally embedding the parent-class attribute structure as the first member of the child-class structure. As shown in Figure 1(c), this arrangement lets you treat any pointer to the Child class as a pointer to the Parent class. In particular, you can always safely cast (upcast) a Child pointer to the Parent and pass such a pointer to any C function that is expecting a pointer to the Parent class. Consequently, all methods designed for the Parent class are automatically available to any Child class—they are inherited.

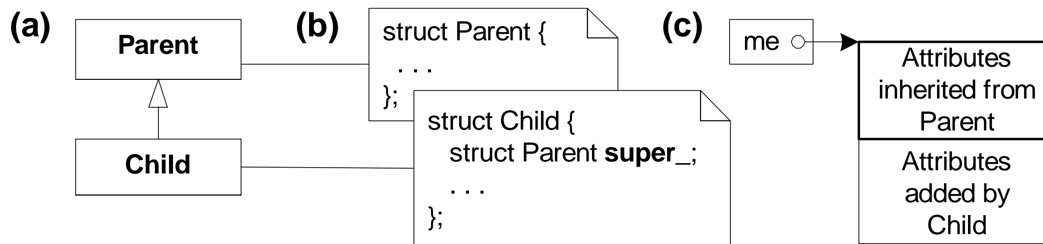


Figure 1 UML class diagram showing the inheritance relationship between Child and Parent classes (a); Declaration of Child structure with embedded Parent as the first member super (b); Memory alignment of a Child object (c).

This simple approach works only for single inheritance (one parent class) because a class with many parent classes cannot align attributes with multiple parents.

The C+ convention is to name the inherited protected attribute super_ (a loan from Java) to make the inheritance relationship between classes more explicit. The super member provides a handle to access the superclass' attributes. For example, a grandchild class can access its grandparent's attribute either as foo as: me->super_.super_.foo, or by direct upcasting: ((Grandparent *)me)->foo.

Inheritance adds responsibilities to class constructors and destructors. Because each child object contains an embedded parent object, the child constructor must initialize the portion controlled by the parent through an explicit call to the parent's constructor. To avoid potential dependencies, the superclass constructor should be called before initializing the attributes. Exactly the opposite holds true for the destructor. The inherited portion should be destroyed as the last step.

Exercise 2 Define preprocessor macro SUBCLASS(class_, super_), so that a class Circle derived from class Shape can be defined as follows:

```
1:  #include "shape.h"
2:
3:  SUBCLASS(Circle, Shape)                        /* Class Circle extends Shape */
4:      double r__;                                          /* private radius */
5:  METHODS
6:      void Circle_ctor(Circle *me, char *name, double r);
7:      double Circle_area(Circle *me);
8:      void Circle_scale(Circle *me, double mag);
9:  END_CLASS
```

Exercise 3 Provide definition of the Circle class constructor Circle_ctor(). Hint: don't forget to explicitly construct the superclass Shape.

3.3 Polymorphism

Subclasses often have a need to redefine and refine methods inherited from their ancestor classes. More specifically, a subclass often needs to override behavior defined by its ancestor class by providing a different implementation of one or more inherited methods. For this process to work, the association between an object and its methods cannot be established at compile time[1]. Instead, the binding must happen at run time and is therefore called **dynamic binding**. Dynamic binding lets you substitute objects with identical interfaces (objects derived from a common superclass) for each other at run time. This substitutability is called **polymorphism**.

Perhaps the best way to appreciate dynamic binding and polymorphism is to look at some real-life examples. You can find polymorphism in many systems (not necessarily object-oriented) often disguised and called hooks or callbacks.

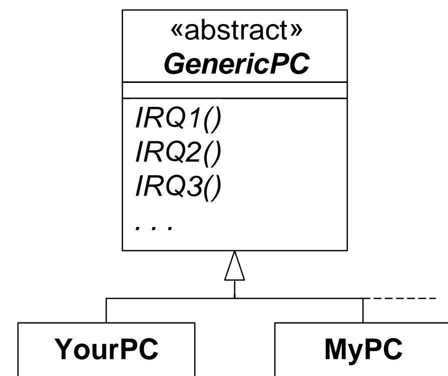


Figure 2 YourPC and MyPC as subclasses of GenericPC class

As the first example, let's examine dynamic binding implemented in hardware. Consider interrupt vectoring of a typical microprocessor system, for example, an x86-based PC. In the PC, the programmable interrupt controller provides for the run-time association between the interrupt request (IRQ) and the interrupt service routine (ISR). Interrupt handling is polymorphic, because all IRQs are handled uniformly in hardware. Concrete PCs (subclasses of the GenericPC class), such as YourPC and MyPC (see Figure 2), can react quite differently to the same IRQ. For example, IRQ4 can cause YourPC to fetch a byte from COM1 and MyPC to output a byte to LPT2.

As another example of a system using polymorphism, consider the MS-DOS device driver design shown in Figure . MS-DOS specifies two abstract types of devices: character and block. A character device performs input and output by a single character at a time. Specific character devices include the keyboard, screen, serial port, and parallel port. A block device performs in-

[1] Some subclasses might not even exist yet at the time the superclass is compiled.

put and output in structured pieces, or blocks. Specific block devices include disk drives and other mass storage devices.

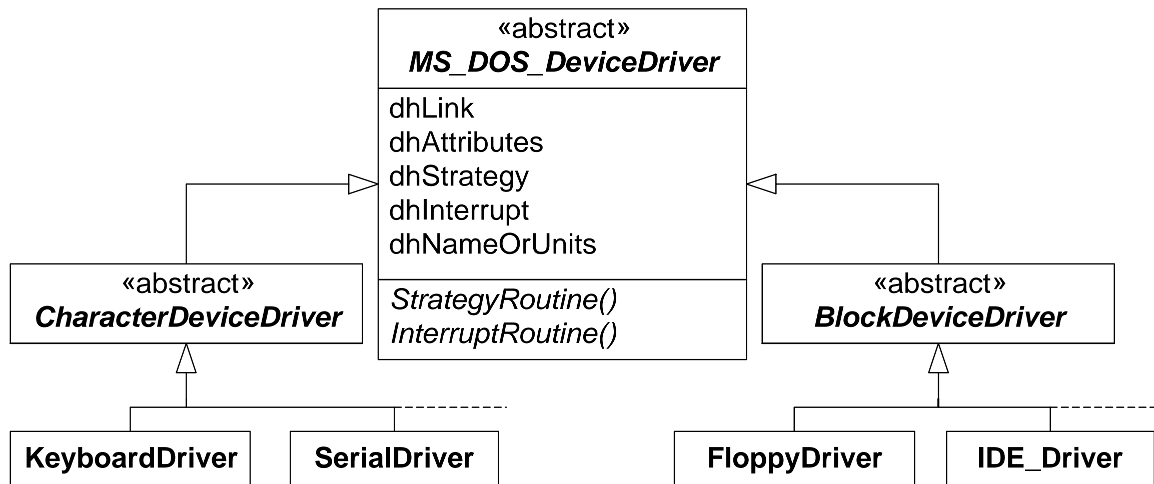


Figure 3 MS-DOS device-driver taxonomy.

The abstract classes MS-DOS_Device_Driver, CharacterDeviceDriver, and BlockDeviceDriver from Figure 3 are specified only in the MS-DOS documentation, rather than in any programming language. Still, MS-DOS drivers clearly use the polymorphism design pattern. As long as device drivers comply with the specification (which is to extend one of the two abstract device driver classes), they are substitutable for one another and are treated uniformly by the operating system.

MS-DOS itself can be viewed as a abstract superclass for specific implementations, such as MS-DOS 5.0 or MS-DOS 6.22 (see Figure 4). The Interrupt-21H functions provide the dynamic-binding mechanism to invoke operating system services from applications. Among others, the dynamic binding allows you to change the implementation of MS-DOS (for example, upgrading from MS-DOS 5.0 to MS-DOS 6.22) without affecting the MS-DOS applications (even without reinstalling them).

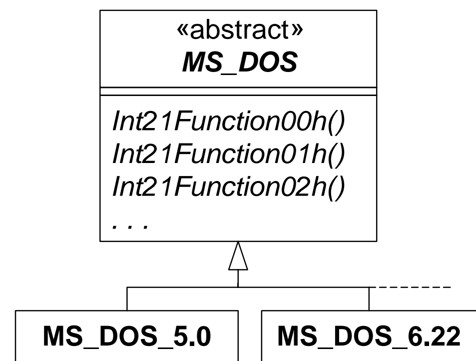


Figure 4 Dynamic binding in MS-DOS implemented with the Interrupt-21H functions.

As you probably noticed in the previous examples, dynamic binding always involves a level of indirection in method invocation. In C, this indirection can be provided by function pointers grouped into **virtual tables** (VTABLEs), see Figure 5. The function pointer stored in the VTABLE represents a method (virtual function in C++), which a subclass can override. All instances of a given class have a pointer to that class' VTABLE (exactly one VTABLE per class exists). This pointer is called the **virtual pointer** (VPTR). Dynamic binding is a two-step process of (1) de-referencing the VPTR to get to the VTABLE, and (2) de-referencing the desired function pointer to invoke the specific implementation.

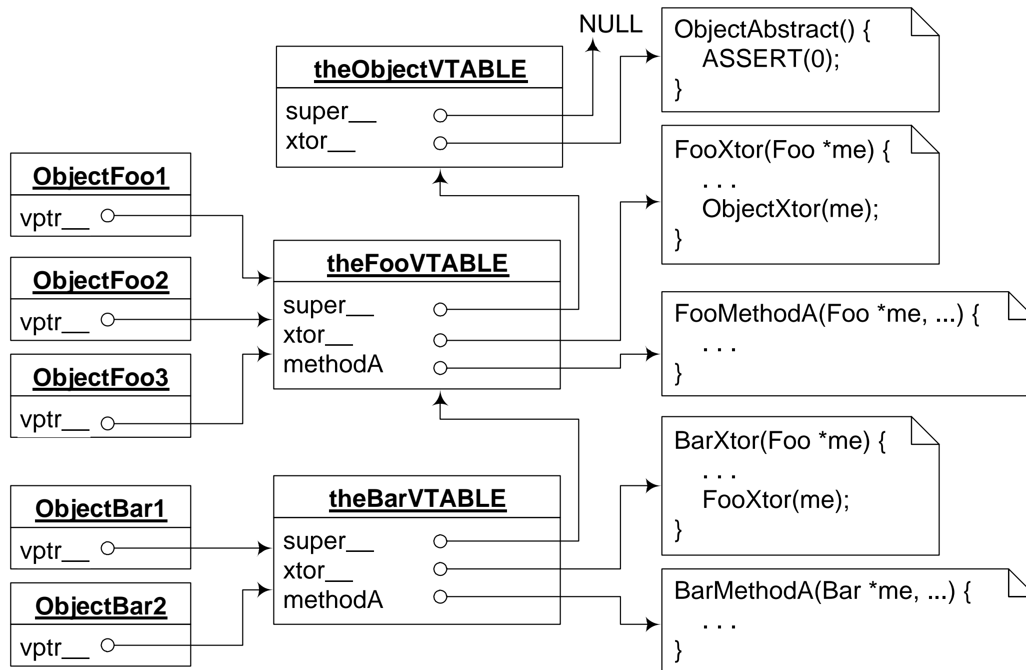


Figure 5 Run-time relations between objects, VTABLEs, and method implementations.

Each object involved in dynamic binding must store the VPTR to its class' VTABLE. One way to enforce the availability of the VPTR is to require that all classes using polymorphism must directly or indirectly derive from a common abstract base class Object (again a loaner from Java). The VTABLEs themselves require a separate and parallel class hierarchy, because the virtual methods need to be inherited, as well as the attributes. The root abstract base class for the VTABLE hierarchy is the ObjectVTABLE class. Listing 2 provides the C+ declaration of these two base classes

```
 1: CLASS(Object)
 2:     struct ObjectVTABLETag *vptr__;              /* private virtual pointer */
 3: METHODS
 4:                                           /* protected constructor 'inline'... */
 5:     #define Object_ctor_(me_) ((me_)->vptr__ = &CP_ObjectVTABLE, (me_))
 6:
 7:                                            /* protected destructor 'inline'... */
 8:     #define Object_xtor_(me_) ((void)0)
 9:
10:                                    /* dummy implementation for abstract methods */
11:     void Object_abstract(void);
12:
13:                                          /* Run Time Type Identification (RTTI) */
14:     #define Object_IS_KIND_OF(me_, class_) \
15:         Object_isKindOf__((Object *)(me_), &CP_##class_##VTABLE)
16:
17:     int Object_isKindOf__(Object const *me, void const *vtable);
18: END_CLASS
19:
20: CLASS(ObjectVTABLE)
21:     ObjectVTABLE *super__;                        /* pointer to superclass' VTBL */
22:     void (*xtor)(Object *);                         /* public virtual destructor */
```

```
23:  METHODS
24:  END_CLASS
25:
26:  extern ObjectVTABLE CP_ObjectVTABLE;             /* Object class descriptor */
```

Listing 2 Declaration of `Object` and `ObjectVTABLE` abstract base classes.

The Object class is declared in Listing 2, lines 1–18. The Object's only attribute is the private virtual pointer vptr__ (line 2). The Object class is abstract, which means that it is not intended to be instantiated (but only inherited from), and therefore protects its constructor Object_ctor_() and destructor Object_xtor_(). Other facilities supplied by the Object class include a dummy implementation Object_abstract() (line 11) to be used for pure virtual methods, and a simple run-time type identification (RTTI), defined as a macro Object_IS_KIND_OF() (lines 14–15).

The purpose of class ObjectVTABLE (Listing 2, lines 20–26) is to provide an abstract base class for the derivation of VTABLEs. The first private attribute super__ (line 21) is a pointer to the superclass' VTABLE. You can identify this pointer with the arrow pointing from the subclass to the superclass in the UML class diagram[1]. The second attribute (line 22) is the virtual destructor, which is inherited subsequently by all subclasses of ObjectVTABLE. Consistently with the C+ convention, the destructor is defined as a pointer to a function that takes only the me pointer and returns void. VTABLE is a Singleton[2], which means that there should be exactly one instance of the VTABLE for any given class. This sole instance for any given class <Class> is called the<Class>VTABLE. Listing 2 declares the VTABLE instance for the Object class (CP_ObjectVTABLE) in line 26.

The hierarchies of the attribute classes (rooted in the Object class) and VTABLEs (rooted in the ObjectVTABLE class) must be exactly parallel. The following macro SUBCLASS() encapsulates the construction of a subclass (see Exercise 4)

```
#define SUBCLASS(class_, superclass_) \
    CLASS(class_) \
        superclass_ super_;
```

Similarly, constructing the VTABLE hierarchy and declaring the VTABLE singletons can be encapsulated in the macro VTABLE()

```
#define VTABLE(class_, superclass_) }; \
    typedef struct class_##VTABLETag class_##VTABLE; \
    extern class_##VTABLE CP_##class_##VTABLE; \
    struct class_##VTABLETag { \
        superclass_##VTABLE super_;
```

The VTABLE Singletons, as all other objects, need to be initialized through their own constructors. Preprocessor macros can automate the generation of these constructors. The body of the VTABLE constructor can be broken into two parts: (1) copying the inherited VTABLE and (2) initializing or overriding the chosen function pointers. The first step is generated automatically by the macro BEGIN_VTABLE()

```
 1:  #define BEGIN_VTABLE(class_, superclass_) \
 2:      class_##VTABLE CP_##class_##VTABLE; \
 3:      static ObjectVTABLE *class_##VTABLECtor(class_ *t) { \
```

[1] That is why the arrow denoting inheritance points from the subclass to the superclass.

[2] I use here the name Singleton as the Singleton design pattern [Gamma+ 95] just to denote a class with the single instance, not necessarily to strictly apply the pattern.

```
4:          class_##VTABLE *me = &CP_##class_##VTABLE; \
5:          *(superclass_##VTABLE *)me = \
6:              *(superclass_##VTABLE *)((Object *)t)->vptr__;
```

Listing 3 BEGIN_VTABLE() macro.

This macro first defines the object, which is the<Class>VTABLE instance (Listing 3, line 2), and then starts defining the static VTABLE constructor (line 3). The first part of the constructor makes a copy (copy-by-value) of the inherited VTABLE (lines 5-6), which guarantees that adding new virtual functions to the superclass won't break subclasses. Consequently, no manual changes to the subclasses are required after adding new attributes or methods to the superclass (you only have to recompile the subclass code). Unless a given class explicitly chooses to override the superclass behavior, the inherited or copied virtual functions are adequate. Of course, if a class adds its own virtual functions, the corresponding function pointers are not be initialized during this step.

The second step of binding virtual functions to their implementation is facilitated by the macro VMETHOD()

```
#define VMETHOD(class_, meth_) ((class_##VTABLE *)me)->meth_
```

This macro is an l-value, and its intended use is to assign to it the appropriate function pointer, for example

```
VMETHOD(Object, xtor) = (void (*)(Object *))&Shape_xtor;
```

Generally, in order to avoid compiler warnings, you must explicitly upcast the function pointer to take the superclass me pointer (Object* in this case) rather than subclass pointer (Shape* in this case).

You should initialize all function pointers in the VTABLE, even if you intended some methods to be abstract (pure virtual in C++) and don't want to provide the implementation. The Object base class offers the Object_abstract() dummy implementation specifically for initializing the abstract methods. An attempt to execute Object_abstract()aborts the execution (through a failing assertion), which helps detect unimplemented abstract methods at run time.

The attribute and virtual-method class hierarchies can grow independently. However, they are coupled together by the VPTR attribute, which needs to be initialized to point to the appropriate VTABLE Singleton, as shown in Figure 5. The appropriate place to set up this pointer is, of course, the constructor. The VPTR initialization must be done *after* the superclass constructor call because the superclass constructor sets the VPTR to point to the superclass' VTABLE. If the VTABLE for the object under construction is not yet initialized, the VTABLE constructor should be called. The following macro VHOOK()accomplishes these two steps

```
1: #define VHOOK(class_) \
2:     if (((ObjectVTABLE *)&CP_##class_##VTABLE)->super__ == 0) \
3:         ((Object *)me)->vptr__ = class_##VTABLECtor(me); \
4:     else \
5:        (((Object *)me)->vptr__ = (ObjectVTABLE *)&CP_##class_##VTABLE)
```

Listing 4 VHOOK() macro.

To determine whether the VTABLE has been initialized, the macro VHOOK() checks the super__ attribute (Listing 4, line 2). If the attribute is NULL (value implicitly set up by the guaranteed static pointer initialization), then the VTABLE constructor must be invoked (line 3) before setting up the VPTR; otherwise, just the VPTR must be set up (lines 5-6). Note that because

VHOOK() is invoked after the superclass constructor, the superclass' VTABLE is already initialized by the same mechanism applied recursively, so the whole class hierarchy is initialized properly.

Finally, after all the setup work is done, you are ready to use dynamic binding. For the virtual destructor (defined in the class Object), the polymorphic call takes the form

```
(*obj->vptr__->xtor)(obj);
```

where, obj is assumed to be of Object* type. Note that the obj pointer is used in this example twice: once for resolving the method and once as the me argument.

In a general case, you deal with Object subclasses rather than Objects directly. Therefore you have to upcast the object pointer (on type Object*) and downcast the virtual pointer vptr__ (on the specific VTABLE type) to find the function pointer. These operations, as well as double-object pointer referencing, are encapsulated in the macros VPTR(), VCALL(), and END_CALL

```
#define VPTR(class_, obj_) \
    ((class_##VTABLE *)(((Object *)(obj_))->vptr__))

#define VCALL(class_, meth_, obj_) \
    (*VPTR(class_, obj_)->meth_)((class_*)(obj_)
#define END_CALL )
```

For example, the virtual destructor call on behalf of object foo of any subclass of class Object takes the form

```
VCALL(Object, xtor, foo)
END_CALL;
```

If a virtual function takes arguments other than me, these arguments should be sandwiched between macro VCALL() and END_CALL. The virtual function can also return a result. For example

```
result = VCALL(Foo, computeSomething, obj), 2, 3,
         END_CALL;
```

where, obj points to a Foo class or any subclass of Foo, and the virtual function computeSomething() is defined in FooVTABLE. Note the use of the comma after VCALL().

3.4 Costs and Overhead

Any OO programmer can benefit from understanding costs associated with using the OO layer.

Abstraction typically incurs no overhead and actually often brings some performance boost. If an ADT truly abstracts some useful concept, the OO style of programming typically results in fewer arguments passed to the methods because all attributes are passed as only one me argument.

Inheritance is also mostly free. The invocation of an inherited method on behalf of a distant successor object is exactly as expensive as invocation on behalf of the parent object. The only overhead caused by inheritance comes from constructor invocation, which must initialize all parts inherited from ancestors. If the hierarchy is deep, nested, superclass, constructor calls can consume significant stack space.

In contrast, polymorphism always incurs some memory and run-time costs. As far as memory is concerned, each class requires space for its VTABLE. The space required is typically several bytes for function pointers. In addition to this one-time memory cost, each object must contain the VPTR, which is inherited directly or indirectly from the Object class. If many instances of a class exist, the VPTRs in each object can easily add up to something significant.

The run-time cost of dynamic binding in C+ is similar to C++. In fact, most compilers generate identical code for C+ and C++ virtual calls. The following code fragment highlights this overhead for a typical CISC processor (x86 running in protected 32-bit mode)

```
; static binding: Shape_xtor(c)
    push   ebx                   ; push "me" (in ebx) onto the stack
    call   _ShapeXtor            ; static call
    add    esp, 4                ; pop the stack

; dynamic binding: VCALL(Object, xtor, c)END_CALL
    mov    eax, DWORD PTR [ebx+0] ; get VPTR into eax
    push   ebx                   ; push "me" (in ebx) onto the stack
    call   DWORD PTR [eax+4]     ; dynamic call
    add    esp, 4                ; pop the stack
```

As you can see, dynamic binding requires only one more assembly instruction than does static binding. Additional work that to do involves de-referencing VPTR (the me pointer is already in the ebx register) and placing the address of VTABLE into the eax register. The actual call requires also one more memory access to fetch the address of the appropriate function from the VTABLE.

To complete the picture, consider now the virtual call overhead on a RISC architecture, using an ARM instruction set

```
; static binding: Shape_xtor(c)
    mov    a1,v1               ; move "me" (in v1) into a1 (argument1)
    bl     _ShapeXtor          ; static call (branch with link)

; dynamic binding: VCALL(Object, xtor, c)END_CALL
    mov    a1,v1               ; move "me" (in v1) into a1 (argument1)
    ldr    a2,[v1,#0]          ; get VPTR into a2
    mov    lr,pc               ; save return address
    ldr    pc,[a2,#4]          ; dynamically call xtor
```

In this case, the static call is extremely fast with only two instructions and does not involve any data accesses (thanks to the ARM branch-and-link instruction bl). Unfortunately, the dynamic call cannot take advantage of the bl instruction because the address cannot be statically embedded in the bl opcode, and therefore, an additional instruction for saving the return address into the link register lr is necessary. Otherwise, dynamic binding overhead is very similar to that of the CISC processor and involves two additional data accesses (the two highlighted ldr instructions) to de-reference the VPTR and to de-reference the function pointer.

4 An Annotated Example

The example application implements in "C+" the classic polymorphic example of geometric shapes. Concrete shapes, such as Rectangle and Circle, derive from the common abstract base class Shape. The Shape class provides the abstract method area() that returns the area of a given shape. Concrete shapes implement this method differently (e.g., Rectangle computes its area as h×w, while Circle computes its area as 3.14×r*r). Similarly, method scale() must be implemented differently for each subclass of Shape. For the Rectangle, scale() multiplies bothe the width w and the height h by the scale factor. For the Circle, scale() multiples only the radius r by the scale factor. Additionally, class Shape illustrates aggregation (by composition) of other objects (an object of class String in this case). The following Figure shows a UML class diagram of this design.

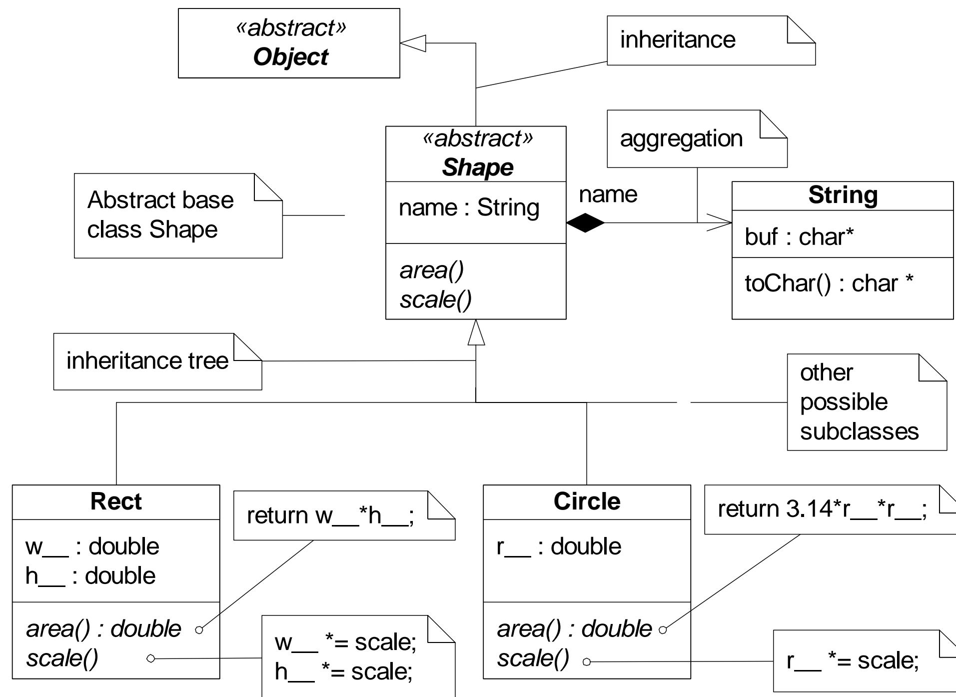


Figure 6 Simple inheritance tree implemented in "C+"

The following header file declares classes String and Shape:

```
 1: #include "cplus.h"
 2:
 3: CLASS(String)
 4:     char *buf__;                              /* private character buffer */
 5: METHODS
 6:     void String_ctor1(String *me, char const *str);          /* public Ctor1 */
 7:     void String_ctor2(String *me, String *other);             /* public Ctor2 */
 8:     void String_xtor(String *me);                              /* public Xtor */
 9:     char const *String_toChar(String *me);               /* to-char conversion */
10: END_CLASS
11:
12: SUBCLASS(Shape, Object)
13:     String name;                                       /* public shape's name */
14: VTABLE(Shape, Object)
15:     double (*area)(Shape *me);                              /* pure virtual! */
16:     void (*scale)(Shape *me, double mag);                   /* pure virtual! */
17: METHODS
18:     void Shape_ctor_(Shape *me, char *name);              /* protected Ctor */
19:     void Shape_xtor_(Shape *me);                          /* protected Xtor */
20: END_CLASS
21:
22: /* test for the abstract Shape class */
23: void test_area(Shape *s);
24: void test_scale(Shape *s);
```

Listing 5 Declaration of the String and Shape classes

Please note that class String does not derive from any class (notice macro CLASS() in Lising 5. line 3), whereas class Shape inherits from Object (notice macro SUBCLASS() in line 12), which makes it polymorhphism-ready.

```
1: /* String class -----------------------------------------------------------*/
 2: void String_ctor1(String *me, char const *str) {
 3:     me->buf__ = (char *)malloc(strlen(str) + 1);
 4:     assert(me->buf__ != (char *)0);
 5:     strcpy(me->buf__, str);
 6: }
 7: /*..........................................................................*/
 8: void String_ctor2(String *me, String *other) {
 9:     String_ctor1(me, String_toChar(other));
10: }
11: /*..........................................................................*/
12: char const *String_toChar(String *me) {
13:     return me->buf__;
14: }
15: /*..........................................................................*/
16: void String_xtor(String *me) {
17:     free(me->buf__);                                    /* release buffer */
18: }
19:
20: /* Shape class ------------------------------------------------------------*/
21: BEGIN_VTABLE(Shape, Object)
22:     VMETHOD(Object, xtor) = (void (*)(Object *))&Shape_xtor_;
23:     VMETHOD(Shape, area) = (double (*)(Shape *))&Object_abstract;
24:     VMETHOD(Shape, scale) = (void (*)(Shape *, double))&Object_abstract;
25: END_VTABLE
26:
27: /*..........................................................................*/
28: void Shape_ctor_(Shape *me, char *name) {
29:     Object_ctor_(&me->super_);                     /* construct superclass */
30:     VHOOK(Shape);                                      /* hook Shape class */
31:     String_ctor1(&me->name, name);                     /* construct member */
32: }
33: /*..........................................................................*/
34: void Shape_xtor_(Shape *me) {
35:     String_xtor(&me->name);                              /* destroy member */
36:     Object_xtor_(&me->super_);                       /* destroy superclass */
37: }
38:
39: /* tests for Shape ========================================================*/
40: void test_area(Shape *s) {
41:     assert(Object_IS_KIND_OF(s, Shape));
42:     printf("name=\"%s\", area()=%.2f, ",
43:            String_toChar(&s->name),                      /* static binding */
44:            VCALL(Shape, area, s)END_CALL);              /* dynamic binding */
45: }
46: /*..........................................................................*/
47: void test_scale(Shape *s) {
48:     double mag = 2.0;
49:     assert(Object_IS_KIND_OF(s, Shape));
50:     printf("scale(%.0f), ", mag);
51:     VCALL(Shape, scale, s), mag END_CALL;               /* dynamic binding */
52: }
```

Listing 6 Definition the String and Shape classes

Listing 6 shows the definition (implementation) of classes String and Shape. The two constructors of String (Listing 6, lines 2-10) are simple because they deal only with initialization of the local String attribute. However, the Shape constructor (lines 28-32) is more involved because it

is responsible for initialization of the super_ attribute inherited from Object (line 29), as well as for hooking the virtual pointer in line 30. Moreover, the Shape constructor is also responsible for initialization of all aggregated objects, such as the name attribute of type String.

Perhaps the most important part of the implementation is definition of the Shape's virtual table, that's accomplished via macros: BEGIN_VTABLE()/END_VTABLE and macros VMETHOD() used in between (lines 21-25). The Shape class declares both its methods area() and scale() as abstract (purely virtual), which means that they must be defined in the subclasses of Shape.

Finally, Listing 6 contains also the unit test for the class Shape, which exercises the two virtual methods area() and scale(), respectively. Please note that the test is written only in terms of the Shape class interface, without any knowledge of concrete subclasses of Shape. These subclasses are completely independent and can be added at a later time because of the late binding that occurs in test methods (lines 44 and 51, respectively).

4.1 Subclassing Shape

Here is the declaration of the Circle sublclass of Shape:

```
 1: #include "shape.h"
 2:
 3: SUBCLASS(Circle, Shape)                              /* Class Circle extends Shape */
 4:     double r__;                                                 /* private radius */
 5: VTABLE(Circle, Shape)                      /* make sure Circle has a virtual table */
 6: METHODS
 7:     void Circle_ctor(Circle *me, char *name, double r);
 8:     double Circle_area(Circle *me);
 9:     void Circle_scale(Circle *me, double mag);
10: END_CLASS
```

Listing 7 Declaration of class Circle

And here is the definition (implementation) of Circle

```
 1: /*..........................................................................*/
 2: BEGIN_VTABLE(Circle, Shape)
 3:     VMETHOD(Shape, area) = (double (*)(Shape *))&Circle_area;
 4:     VMETHOD(Shape, scale) = (void (*)(Shape *, double))&Circle_scale;
 5: END_VTABLE
 6: /*..........................................................................*/
 7: void Circle_ctor(Circle *me, char *name, double r) {
 8:     Shape_ctor_(&me->super_, name);                       /* construct superclass */
 9:     VHOOK(Circle);                                            /* hook Circle class */
10:     me->r__ = r;                                          /* initialise member(s) */
11: }
12: /*..........................................................................*/
13: double Circle_area(Circle *me) {
14:     return 3.141592535 * me->r__ * me->r__;                     /* pi * r-squared */
15: }
16: /*..........................................................................*/
17: void Circle_scale(Circle *me, double mag) {
18:     me->r__ *= mag;
19: }
```

Listing 8 Definition the Circle class

For comparison, there are the declaration and definition of class Rect:

```
#include "shape.h"
```

```
SUBCLASS(Rect, Shape)                               /* Class Rect extends Shape */
    double w__;                                               /* private width  */
    double h__;                                               /* private height */
VTABLE(Rect, Shape)
METHODS
    void Rect_ctor(Rect *me, char *name, double w, double h);
    double Rect_area(Rect *me);
    void Rect_scale(Rect *me, double mag);
END_CLASS
```

Listing 9 Declaration of class Rect

```
/*..........................................................................*/
BEGIN_VTABLE(Rect, Shape)
    VMETHOD(Shape, area) = (double (*)(Shape *))&Rect_area;
    VMETHOD(Shape, scale) = (void (*)(Shape *, double))&Rect_scale;
END_VTABLE

/*..........................................................................*/
void Rect_ctor(Rect *me, char *name, double w, double h) {
    Shape_ctor_(&me->super_, name);                     /* construct superclass */
    VHOOK(Rect);                                             /* hook Rect class */
    me->h__ = h;                                        /* initialise member(s) */
    me->w__ = w;
}
/*..........................................................................*/
double Rect_area(Rect *me) {
    return me->w__  * me->h__;
}
/*..........................................................................*/
void Rect_scale(Rect *me, double mag) {
    me->w__ *= mag;
    me->h__ *= mag;
}
```

Listing 10 Definition the Rect class

4.2 Executing the Test

```
#include "shape.h"
#include "circle.h"
#include "rect.h"

enum { NRECT = 4 };

int main() {
    Circle circle;                        /* Circle instance on the stack frame */
    Circle *c;
    Rect r[NRECT];
    int i;

    /* construct objects... */
    Circle_ctor(&circle, "Circle", 1.0);
    c = &circle;
    for (i = 0; i < NRECT; i++) {
        char name[20];
        sprintf(name, "Rectangle-%d", i);                   /* prepare the name */
        Rect_ctor(&r[i], name, (double)i, 1.0);               /* construct Rect */
    }
                                                         /* test the Circle ... */
```

```
    test_area((Shape *)c);
    test_scale(&c->super_);
    test_area((Shape *)c);
    printf("\n");

                                                  /* test the Rectangles ... */
    for (i = 0; i < NRECT; i++) {
        test_area((Shape *)&r[i]);
        test_scale(&r[i].super_);
        test_area((Shape *)&r[i]);
        printf("\n");
    }

                                                     /* detstroy objects ... */
    VCALL(Object, xtor, c)END_CALL;  /* destroy the Circle, dynamic binding */
    for (i = 0; i < NRECT; i++) {
        VCALL(Object, xtor, &r[i])END_CALL;    /* destroy the Rectangles ...*/
    }
    return 0;
}
```

Listing 11 Test harness for the sample application

5 Summary

OOP is a design method rather than use of particular language or a tool. Indeed, as David Parnas writes:

> *Instead of teaching people that OO is a type of design, and giving them design principles, people have taught that OO is the use of a particular tool. We can write good or bad programs with any tool. Unless we teach people haw to design, the languages matter very little.*

OO languages support OO design directly, but you can also successfully implement OO design in other languages, such as C. Abstraction, inheritance, and polymorphism are nothing but design meta-patterns at the C level. Many C programmers, very likely you also, have been using these fundamental patterns in some form or another for years, often without clearly realizing this fact. As with all design patterns, the three patterns combined allow you to work at a higher OO level of abstraction by introducing their specific naming conventions and idioms.

This manual describes "C+", which is one specific set of such C conventions and idioms. C+ achieves performance and maintainability of code comparable to C++. A particularly important feature of C+ is that adding new attributes and methods, including virtual functions, to a superclass does not require manual changes to subclasses. As in C++, after extending the superclass, you only need to recompile the subclass implementation files.

I have been successfully using C+ in many projects for number of years, and I challenge you to find a more efficient, scaleable, portable, and maintainable implementation. However, perhaps the weakest aspect inherent in any attempt to implement OOP in C (not only C+) is the compromised type safety. The fundamental problem is that a C compiler does not know that some types are generalizations of the others and treats related types as completely different. This issue requires a lot of type casting (upcasting), which is awkward and defeats much of the type safety of the language.

Therefore, if you have access to a decent C++ compiler for your platform, I recommend that you consider using C++ instead of C+. Contrary to some misunderstandings, especially among embedded systems programmers, C++ is not inherently bulky and slow. By sticking only to the essential OO features of C++ and omitting pretty much everything else (a policy embodied in the Embedded C++ standard, see [EC++ 01]), you can achieve very good performance, elegance, convenience, and full compiler support for OOP.

6 References

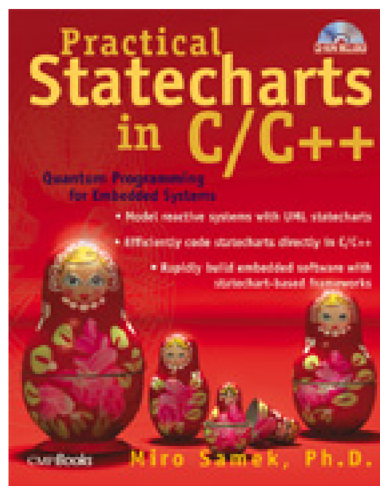
[Brooks 95]	Brooks, Frederick. 1995. The mythical man-month. Anniversary ed. Addison Wesley.
[Gamma 95]	Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns, elements of reusable object-oriented software. Addison Wesley, 1995.
[Labrosse 99]	Labrosse, Jean J., MicroC/OS-II, The Real-Time Kernel. R&D Publications, 1999, ISBN 0-87930-543-6
[Meyer 97]	Meyer, Bertrand, Object-Oriented Software Construction 2nd Edition, Prentice Hall, 1997. ISBN: 0-136-29155-4
[Samek 97]	Samek, Miro, "Portable Inheritance and Polymorphism in C", Embedded Systems Programming, December 1997.
[Samek 02]	Samek, Miro, Practical Statecharts in C/C++: Quantum Programming for Embedded Systems, CMP Books 2002, ISBN 1-57820-110-1
[QL 05a]	Quantum Leaps, Quantum Leaps C/C++ Coding Standard, January 2005, http://www.quantum-leaps.com/resources/goodies.htm#CodingStandard
[QL 05]	Quantum Leaps website, http://www.quantum-leaps.com
[Van Sickle 97]	Van Sickle, Ted. Reusable software components, object-oriented embedded systems programming in C. Prentice Hall, 1997.

7 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 866-450-LEAP (toll free)
+1 919-869-2998 (fax)

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com



"Practical Statecharts in C/C++",
by Miro Samek, Ph.D.
CMP Books 2002,

ISBN 1578201101