PROGRAMMER'S GUIDE

Data Structures and Algorithms with Object-Oriented Design Patterns in C++

Bruno R. Preiss
B.A.Sc., M.A.Sc., Ph.D., P.Eng.

Associate Professor
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Canada

MMI

1 Introduction

This programmer's guide accompanies the book "Data Structures and Algorithms with Object-Oriented Design Patterns in C++." The purpose of this guide is to advise instructors and students who intend to use code taken from the book in their own programs.

The code in the book is written in conformance with the proposed draft standard for the C++ programming language as defined in [1]. At the time of writing, very few compilers comply with the proposed draft standard in its entirety. However, there are many compilers that come close and, with time, I expect that all vendors will comply with the standard.

The programs in the book take advantage of certain features of the proposed draft standard which have been introduced into the language only recently. Table 1 below lists the features used in the book that are not yet be supported by all C++ compilers.

Table 1: New C++ Features Used in the Book

feature	description
A	declarations in `for` statement have block scope
B	nested classes
C	nested class as template argument
D	string class
E	new style include files
F	`std` namespace
G	new-style casts
H	standard exceptions
I	overriding functions can narrow return type
J	run-time type information (`typeid` operator and downcast past a virtual-base class)
K	`numeric_limits` template

The remainder of this document is organized as follows: Section 2 gives the results of compiler compatibility tests conducted by the author to determine to what extent commonly available compilers conform the the proposed standard. This section also identifies the programs in the book which may be affected by non-conforming compilers. Section 3 gives workarounds for each of the features shown in Table 1. These workarounds show how to use the programs in the book with a non-conforming compiler.

2 Compiler Compatibility

This section presents the results of compiler compatibility tests conducted by the author. The objective of these tests is to determine to what extent various commonly available C++ compilers conform to the proposed draft standard and to identify which programs in the book are affected by non-standard compiler behaviour.

Altogether five compilers have been tried:

- Borland C++ 5.0
- Microsoft Visual C++ 5.0
- Sun SPARCompiler C++ 4.1
- Sun SPARCompiler C++ 4.2
- GNU C++ 2.7.2.1

The table below indicates for each compiler whether that compiler supports the new C++ features given in Table 1. The table below also shows for each of the 316 programs in the book, which features (if any) are used by that program.

With the information below you can quickly determine whether the programs you intend to use will work without modification using a particular compiler. Section 3 describes workarounds for each of the features given in Table 1. Thus, if your compiler does not support a required feature, you will have to modify the program as explained in the corresponding workaround.

	language feature supported										
compiler	A	B	C	D	E	F	G	H	I	J	K
Borland C++ 5.0	●	●	●	●	●	●	●	●	●	●	
Microsoft Visual C++ 5.0		●		●	●	●	●	●		●	●
Sun SPARCompiler C++ 4.1		●	●								
Sun SPARCompiler C++ 4.2		●	●				●			●	
GNU C++ 2.7.2.1	●	●	●	●	●		●	●	●	●	

	language feature used										
program	A	B	C	D	E	F	G	H	I	J	K
Program 2.5					●	●					
Program 3.5	●										
Program 4.7								●			
Program 4.14								●			
Program 4.18								●			
Program 4.19							●	●			
Program 4.20		●									
Program 4.21								●			
Program 4.23								●			
Program 5.1										●	
Program 5.3					●						
Program 5.7							●				
Program 5.8				●		●					
Program 5.12					●						
Program 5.20							●				
Program 6.2		●									
Program 6.4								●			
Program 6.7		●									

	language feature supported										
compiler	A	B	C	D	E	F	G	H	I	J	K
Borland C++ 5.0	●	●	●	●	●	●	●	●	●	●	
Microsoft Visual C++ 5.0		●		●	●	●	●	●		●	●
Sun SPARCompiler C++ 4.1		●	●								
Sun SPARCompiler C++ 4.2		●	●				●			●	
GNU C++ 2.7.2.1	●	●	●	●	●		●	●	●	●	

	language feature used										
program	A	B	C	D	E	F	G	H	I	J	K
Program 6.9								●			
Program 6.12						●	●				
Program 6.16								●			
Program 6.19								●			
Program 6.20							●			●	
Program 6.25								●			
Program 6.28								●			
Program 7.2		●									
Program 7.3								●			
Program 7.4	●										
Program 7.5								●			
Program 7.7							●	●			
Program 7.8							●	●			
Program 7.9							●	●			
Program 7.10		●									
Program 7.11								●			
Program 7.12	●										
Program 7.13								●			
Program 7.15							●	●			
Program 7.16							●	●			
Program 7.17							●	●			
Program 7.18							●				
Program 7.19							●				
Program 7.22								●			
Program 7.25								●			
Program 7.28								●			
Program 7.30							●				
Program 8.2						●					
Program 8.3				●		●					
Program 8.13		●	●								
Program 8.15								●			
Program 8.16								●			
Program 8.17		●	●								
Program 8.19								●			
Program 8.21								●			
Program 8.22								●			
Program 8.23				●		●	●				
Program 9.1		●								●	
Program 9.4							●			●	
Program 9.7							●				

	language feature supported										
compiler	A	B	C	D	E	F	G	H	I	J	K
Borland C++ 5.0	●	●	●	●	●	●	●	●	●	●	
Microsoft Visual C++ 5.0		●		●	●	●	●	●		●	●
Sun SPARCompiler C++ 4.1		●	●								
Sun SPARCompiler C++ 4.2		●	●				●			●	
GNU C++ 2.7.2.1	●	●	●	●	●		●	●	●	●	

	language feature used										
program	A	B	C	D	E	F	G	H	I	J	K
Program 9.8							●			●	
Program 9.9									●	●	
Program 9.11								●	●		
Program 9.12									●	●	
Program 9.14	●							●			
Program 9.15								●	●		
Program 9.16										●	
Program 9.20							●			●	
Program 9.21						●	●			●	
Program 10.2									●		
Program 10.3							●		●		
Program 10.4									●		
Program 10.5								●	●		
Program 10.6								●	●		
Program 10.7									●		
Program 10.9								●	●		
Program 10.10								●	●		
Program 10.11									●		
Program 10.12								●			
Program 10.13									●	●	
Program 10.14	●										
Program 10.16								●			
Program 10.17								●			
Program 10.18								●			
Program 10.20								●			
Program 10.22				●		●	●				
Program 11.4								●			
Program 11.5								●			
Program 11.6								●			
Program 11.7									●	●	
Program 11.8							●		●	●	
Program 11.10								●			
Program 11.11								●	●		
Program 11.12									●		
Program 11.13								●			
Program 11.16								●			
Program 11.17							●				
Program 11.20								●	●		
Program 11.21							●				
Program 11.22							●				

	language feature supported										
compiler	A	B	C	D	E	F	G	H	I	J	K
Borland C++ 5.0	●	●	●	●	●	●	●	●	●	●	
Microsoft Visual C++ 5.0		●		●	●	●	●	●		●	●
Sun SPARCompiler C++ 4.1		●	●								
Sun SPARCompiler C++ 4.2		●	●				●			●	
GNU C++ 2.7.2.1	●	●	●	●	●		●	●	●	●	
	language feature used										
program	A	B	C	D	E	F	G	H	I	J	K
Program 12.1										●	
Program 12.3							●				
Program 12.4	●							●			
Program 12.5	●							●			
Program 12.7							●				
Program 12.8	●							●			
Program 12.11							●				
Program 12.12	●							●			
Program 12.14								●			
Program 12.15								●			
Program 12.16									●		
Program 12.18	●										
Program 12.19							●		●		
Program 12.20							●	●			
Program 12.21							●		●		
Program 12.22							●				
Program 12.23							●				
Program 12.24										●	
Program 13.2						●	●				
Program 13.5								●			
Program 13.6							●	●			
Program 13.9							●	●			
Program 13.10								●			
Program 13.13								●			
Program 13.14							●	●			
Program 13.15							●				
Program 13.16							●				
Program 14.3								●			●
Program 14.5							●				
Program 14.7							●				
Program 14.8							●				
Program 14.9								●			
Program 14.12	●										
Program 14.14	●										●
Program 14.16								●			
Program 14.18						●					
Program 15.20	●										
Program 15.22	●										
Program 16.7							●				
Program 16.8							●				

	language feature supported										
compiler	A	B	C	D	E	F	G	H	I	J	K
Borland C++ 5.0	●	●	●	●	●	●	●	●	●	●	
Microsoft Visual C++ 5.0		●		●	●	●	●	●		●	●
Sun SPARCompiler C++ 4.1		●	●								
Sun SPARCompiler C++ 4.2		●	●				●			●	
GNU C++ 2.7.2.1	●	●	●	●	●		●	●	●	●	

	language feature used										
program	A	B	C	D	E	F	G	H	I	J	K
Program 16.9	●						●				
Program 16.13											●
Program 16.14	●						●				
Program 16.15	●						●				●
Program 16.16	●						●				
Program 16.17							●			●	
Program 16.18							●				
Program 16.19							●				
Program A.5						●					
Program A.7				●		●		●			

3 Workarounds for Unsupported Features

3.1 Feature A. For Statement Declaration Scope

In C++ it is often convenient to declare a loop index right in a `for` statement like this:

```
for (int i = 0; i < 100; ++i)
    // ...
```

It used to be the case that the scope of a variable is the block in which the `for` statement appears. However, in the proposed draft standard C++ standard, the scope of such a declaration is now the body of the `for`. Thus, we may now write:

```
for (int i = 0; i < 100; ++i)
    {}
for (int i = 0; i < 100; ++i)
    {}
```

If your compiler balks at this, you can rewrite the code given above like this:

```
for (int i = 0; i < 100; ++i)
    {}
for (i = 0; i < 100; ++i)
    {}
```

I.e., remove the second declaration of the variable `i` and replace it with a simple assignment.

3.2 Feature B. Nested Classes

When writing C++ code it is often desirable to nest the declaration of one class inside another. (Nested class definitions are allowed in the proposed draft C++ standard). Most C++ compilers now allow nested class definitions.

```
class A
{
public:
    class B
    {
    public:
        void F ();
    };
};

void A::B::F ()
    {}
```

If your compiler will not compile a class that contains a nested class, you will have to move the nested class out like this:

```
class A
{
};

class A_B
{
    void F ();
};

void A_B::F ()
    {}
```

In this case, two separate classes are defined at file scope: `A` and `A_B`.

3.3 Feature C. Nested Class as Template Argument

While most C++ compilers now allow nested class definitions, at least one of the compilers tested does not compile a program if a nested class is used as a template argument:

```
template <class T>
class Array
{
    T* data;
public:
    Array (int length) : data (new T [length])
        {}
    ~Array ()
        { delete [] data; }
};

class A
{
    class B
    {
    };
    Array<B> array;
public:
    A (int length) : array (length)
        {}
};
```

Here the nested class `B` is used as a template argument in the declaration of the member variable `array` of class `A`.

If your compiler will not compile the use of a nested class as a template argument, you will have to move the nested class out like this:

```
class A_B
{
};

class A
{
    Array<A_B> array;
public:
    A (int length) : array (length)
        {}
};
```

In this case, two separate classes are defined at file scope: `A` and `A_B`.

3.4 Feature D. Standard `string` Class

The proposed draft C++ standard defines a standard `string` class. (The standard string class is declared in the new standard header file `<string>`). The standard string class provides myriad member functions for manipulating strings.

If your compiler does not provide the standard string class, you will have to write it yourself. However, it is not necessary to write the ultimate string class to support the programs in the text. Only a subset of the functionality is used. In fact, only the functions declared in the following following code are needed:

```
class string
{
    // ...
public:
    string ();
    string (char);
    string (char const*);
    string (string const&);
    ˜string ();
    char operator [] (unsigned int) const;
    string& operator = (char);
    string& operator = (char const*);
    string& operator = (string const&);
    string& append (string const&);
    int compare (char const*) const;
    int compare (string const&) const;
};
ostream& operator << (ostream&, string const&);
istream& operator >> (istream&, string&);
```

3.5 Feature E. New-Style Include Files

The proposed draft C++ standard language introduces a new model for specifying `include` files. The new model is motivated by the introduction of namespaces and, in particular, by the new `std` namespace.

The old C++ way of doing things was to name the header files with a `.h` extension. The proposed standard drops the `.h` from the standard header files. If your compiler supports only the old-style header files you simply need to put back the `.h` in the `include` directive.

3.6 Feature F. The `std` Namespace

The proposed draft C++ standard introduces the notion of namespaces and, in particular, the `std` namespace. Virtually all the functions and classes required by the standard are declared within the `std` namespace. In particular, standard functions such as `sqrt`, `isalpha`, `isdigit`, `malloc`, and `free`, have been moved into the `std` namespace. I.e., they are now to be known as `std::sqrt`, `std::isalpha`, etc. If your compiler does not yet support the `std` namespace, you may continue to refer to them without using the `std::` scope specifier.

3.7 Feature G. New-style Casts

The following new cast operators have been introduced in the new C++ language definition: `static_cast<T>()`, `dynamic_cast<T>()`, `const_cast<T>()`, and `reinterpret_cast<T>()`. These cast operators are used as follows:

```
class B {};
class D : public B {};

D d;
B const& b;
D const& d1 = static_cast<D const&> (b);
D const& d2 = dynamic_cast<D const&> (b);
B& b2 = const_cast<B&> (b);
```

An expression of the form `static_cast<T>(e)` is equivalent to `(T)e`. The effect is to convert the expression `e` to type `T`. No run-time check is done to ensure that the conversion is type safe.

A `dynamic_cast` can be used to convert a pointer or a reference to an object from one type to another. A run-time check is make in order to ensure that the conversion is valid. It is only possible to use this conversion operator on a polymorphic object (i.e., an object that has at least one virtual member function).

The `const_cast` operator is used to cast away `const`ness. It cannot be used to change the type of the expression.

If your compiler does not support the new-style cast operators, then you will have to replace all of the new-style casts with old-style casts:

```
D const& d1 = (D const&) b;
```

```
D const& d2 = (D const&) b;
B& b2 = (B&) b;
```

However, it should be noted that the old-style cast does not do any run-time type checking. Also, it is not possible to downcast past a virtual base class using an old style cast. (See Section 3.10).

3.8 Feature H. Standard Exceptions

The new C++ language definition introduces a number of standard exceptions. For example, the `<new>` header file defines the exception `bad_alloc` which is thrown by `operator new` when it is unable to allocate storage.

Similarly, the header `<typeinfo>` header file defines the exception `bad_cast` which is thrown by the `dynamic_cast` operator when when an invalid cast operation is attempted.

Similarly, the proposed draft standard for C++ defines a new header file `<stdexcept>` that declares a number of exceptions classes used in by various library functions. These include `logic_error`, `domain_error`, `invalid_argument`, `range_error`, and `out_of_range`.

If the compiler you are using does not implement these exceptions, you can provide your own implementations. For example, a minimal implementation of the `logic_error` class looks like this:

```
class logic_error : public exception
{
    string what;
public:
    logic_error (string const& _what) : what (_what) {}
};
```

If you are using an older compiler, you may find that it does not provide the `exception` class. Older versions of C++ called this class `xmsg`.

Finally, if you are using a really old C++ compiler, it may not support exceptions at all. In that case, the workaround is to replace every `throw` statement with a call to the `abort()` function.

3.9 Feature I. Overriding Functions Can Narrow Return Type

According to the proposed draft standard for the C++ language, it is now possible for a function in a derived class which overrides an inherited function to narrow the return type. Consider the following code:

```
class B
{
public:
    virtual B& Func ()
        { return *this; }
};
```

```
class D : public B
{
public:
    virtual D& Func ()
        { return *this; }
};
```

The member function `Func` defined in the base class `B` returns a reference to `B`. In the derived class `D`, the member function `Func` is overridden. Note that `B::Func` and `D::Func` have the same signature. However, they differ in the return type. Nevertheless, the declaration of `D::Func` is valid and type-safe because the return type is a reference to `D` and `D` is derived from `B`.

Older compilers may not allow an overriding function to have a different return type. It is possible to work around the problem like this:

```
class B
{
public:
    virtual B& Func ()
        { return *this; }
};

class D : public B
{
public:
    virtual D& DFunc ()
        { return *this; }
    virtual B& Func ()
        { return DFunc (); }
};
```

First, introduce the member function `DFunc` in the derived class which returns a reference to `D`. Second, override the inherited `Func` function. The `Func` function simply calls `DFunc`.

3.10 Feature J. Run-Time Type Information

The proposed draft C++ standard introduces a facility which provides *run-time type information.* E.g., the `typeid` operator can be used to obtain information about an object at run-time. By using the `typeid` operator, one can determine whether objects have the same type, one can order objects by their type, and one can obtain a `string` that contains the name of the type. The following code illustrates these features:

```
class B
{
    virtual ~B() {}
};
class D1 : public B {};
class D2 : public B {};

D1 d1;
D2 d2;
B& b1 = d1;
B& b2 = d2;
if (typeid(b1) == typeid(b2))
    cout << "same" << endl;
if (typeid(b1).before(typeid(b2)))
    cout << "before" << endl;
cout << typeid(b1).name() << endl;
cout << typeid(b2).name() << endl;
```

The `typeid` operator can only used with polymorphic classes. (I.e., only classes that have at least one virtual function provide run-time type information).

If your compiler does provide support run-time type information, you can write the code like this:

```
class B
{
    virtual string TypeId () const = 0;
    virtual string name () const = 0;
};
class D1 : public B
{
    int TypeId () { return 1; }
    int name () { return "D1"; }
};
class D2 : public B
{
    int TypeId () { return 2; }
    int name () { return "D2"; }
};

D1 d1;
D2 d2;
B& b1 = d1;
B& b2 = d2;
if (b1.TypeId() == b2.TypeId())
    cout << "same" << endl;
if (b1.TypeId() < b2.TypeId())
    cout << "before" << endl;
cout << b1.name() << endl;
cout << b2.name() << endl;
```

Another capability supported by run-time type information is the ability to downcast past a virtual base class like this:

```
class B
{
    virtual ~B() {};
};
class D : public virtual B {};

D d;
B& b = d;
D& d2 = dynamic_cast<D&> (b);
```

Without run-time type information, the conversion from `B&` to `D&` cannot be done. Note that this particular operation requires both the new-style cast operator *and* run-time type information (see Section 3.7).

If your compiler does not support run-time type information and/or downcasting past a virtual base class, you can write the code like this:

```
class D;
class B
{
    virtual D& DownCastToD ()
        { throw bad_cast(); }
};
class D : public virtual B
{
    D& DownCastToD ()
        { return *this; }
};
D d;
B& b = d;
D& d2 = b.DownCastToD();
```

Note that this approach requires a separate member function in the base class `B` for every possible derived class to which a downcast operation is required.

3.11 Feature K. The `numeric_limits` Template

The proposed draft C++ standard introduces the header file `<limits>` which defines the template `numeric_limits`. Specializations of this class are to be provided for all the built-in types, including `bool`, `int`, `double`, etc. The following is an excerpt from the template declaration:

```
template<class T> class numeric_limits
{
public:
    static T min();
    static T max();
    // ...
};
```

Note that each specialization provides the static functions `max` and `min` which return the maximum and minimum values of type `T`. For example `numeric_limits<int>::max()` returns the maximum value of of an `int`.

If your compiler does not provide the `numeric_limits` templates, you can always resort using the old preprocessor constants defined in the header file `<climits>`: `INT_MAX`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`, ...

References

[1] ANSI Accredited Standards Committee X3, Information Processing Systems. *Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*, December 1996. Document Number X3J16/96-0225 WG21/N1043.