asdasd

A

ANSI/ISO Resolutions

Prefer the standard to the offbeat
– Strunk & White

This chapter summarizes the major resolutions by the joint ANSI X3J16 and ISO WG21 committees for their draft working paper for a C++ standard as of March 1995. Minor clarifications are incorporated into their respective chapters rather than presented here.

a.1 Standardization

In December, 1989, the American National Standards Institute's committee for the standardization of C++ had its organizational meeting in Washington D.C., USA. In June, 1991, in Lund, Sweden, this committee formally joined with C++ standards efforts in other countries such as France, Germany, Denmark, Japan, Sweden, and the UK under the auspices of the International Standards Institute. A first draft of a standard is expected to be presented for public review in the spring of 1995. For more details of the work of the committee and its decisions see Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, 1994.

Please remember that there are necessary delays between the time the committees pass a resolution and the time implementations become available.

§r.2.4: As mentioned in §r.16.2, many of the characters traditionally used to represent C++ programs are available only in the American national character set ASCII and in extended character sets such as Unicode. To allow C++ use with national 7 bit character sets, the following keywords are provided:

```
and  and_eq  bitand  bitor  compl  not  or  or_eq  xor  xor_eq
```

These keywords are equivalent to the following tokens, respectively:

```
&&   &=      &       |      ~      !    ||  |=     ^    ^=
```

The lists of tokens in §r.2.4 are augmented with the following tokens:

```
<%   %>   <:   :>   %:   %:%:   not_eq
```

These tokens are equivalent to the following tokens, respectively:

```
{    }    [    ]    #    ##     !=
```

The trigraph support in §r.16.2 is still available and necessary for putting missing characters such as `{` into strings and character constants. For example `"??<"` is equivalent to `"{"`, and `"<%"` is a two character string as ever. Having provided backwards compatibility to 7 bit national character sets, the committee is working on allowing the use of extended character sets such as Unicode.

§r.2.5.5 Boolean Literals: There are two literals of type `bool`: `true` and `false`.

§r.3.1: A declared object need not be defined if it is unused or is used only as the operand of `sizeof`.

§r.3.3.1 Namespaces: A mechanism, called namespaces, was introduced for controlling the scope of otherwise global names.

The grammar for the namespace constructs is:

namespace-definition:
  `namespace` *identifier*$_{opt}$ `{` *declaration-list*$_{opt}$ `}`

namespace-alias-definition:
  `namespace` *identifier* `=` *namespace-name* `;`

using-declaration:
  `using` *namespace-name* `::` *identifier* `;`

using-directive:
  `using namespace` *namespace-name* `;`

namespace-name:
  `::`$_{opt}$ *identifier*
  `::`$_{opt}$ *namespace-name* `::` *identifier*

An *identifier* in a *namespace-name* must have been specified to refer to a namespace by being the *identifier* in a *namespace-definition* or a *namespace-alias-definition*.

§r.3.3.1.2 Namespaces and Namespace Members: The *identifier* in a *namespace-alias-definition* defines a synonym for the *namespace-name* mentioned. A namespace alias may be defined to refer the same namespace more than once; otherwise, it may not not be defined more than once in a single scope. The *identifier* in a *namespace-definition* must either be undefined in its scope or be a *namespace-name* defined in a *namespace-definition* in the same scope. All namespaces with the same *namespace-name* in the same scope are considered part of the same namespace. All global namespaces with the same *namespace-name* are considered part of the same namespace. An *identifier* used as a global *namespace-name* cannot be used as the name of any other global namespace, template, type, function, object, or value in the program.

A *namespace-definition* or a *namespace-alias-definition* is a *declaration*. The *identifier*s mentioned after `::` in a *using-declaration* must have been declared in the namespace named by the *namespace-name*.

A *namespace-definition* can occur only at the global scope or within another namespace. A *using-declaration* can be used as a *declaration-statement*, a *member-declaration*, at the global scope, or within another namespace. A *using-directive* can be used as a *declaration-statement*, at the global scope, or within another namespace (that is not a class).

All unnamed namespaces within a compilation unit are treated as one namespace, ‘‘the unnamed namespace,’’ and treated as if it had a name that is unique in a program. A name from an unnamed namespace can be accessed without qualification after its point of declaration.

A type, object, etc., declared within a namespace is a member of that namespace and does not clash with names defined outside that namespace. For example:

```
namespace A {
        class String { /* ... */ };
        void f(String);
}

namespace B {
        class String { /* ... */ };
        void f(String);
}
```

Members of a namespace can be defined within that namespace. Members of a named namespace can also be defined outside their namespace using explicit qualification. In that case, the definition from the point of the qualification until the end of the declaration is considered in the scope of the namespace. For example:

```
void A::f(String s)          // A::String
{
        String ss = "asdf";  // A::String
        // ...
}
```

An `extern` function, `friend` function, or type name first declared within a namespace or in a non-namespace member of a namespace is a member of that namespace.

An object or function declared `extern "C"` can link to a C object or function with the same unqualified name. For example:

```
namespace X {
        extern "C" void f(); // will link to a C
                             // function named f()
}
```

§r.3.3.1.2 Explicit Qualification: A name defined in a namespace can be accessed qualified by a *namespace-name* for its namespace using the `::` operator. For example:

```
void g()
{
        A::String s;
        A::f(s);
}
```

§r.3.3.1.2 Using-declarations and Using-directives: The member names specified in a *using-declaration* are considered declared in the scope in which the *using-declaration* appears. The names thus declared are aliases for their original declarations so that the *using-declaration* does not affect the type, linkage, etc., of the members referred to. For example:

```
void h()
{
        using A::String;
        using A::f;

        String s; // A::String
        f(s);     // A::f
}
```

A *using-declaration* is not a definition. Thus, redundant *using-declaration*s of a name are allowed.

A *using-directive* specifies that the names in the namespace can be used in the scope in which the *using-directive* appears exactly as if the names from the namespace had been declared outside a namespace at the point where their namespace was declared. For example:

```
void k()
{
        using namespace A;

        String s; // A::String
        f(s);     // A::f
}
```

A *using-directive* does not declare any names in the scope in which it occurs.

Neither *using-declaration* nor a *using-directive* affects names being declared.

§r.3.3.1.3 Classes: For purposes of name lookup, the scope of a class is a namespace. The scope of a derived class is nested in the scope of each of its base classes. Qualification and `using` apply to classes exactly as to other namespaces. For example:

```
class B {
public:
        virtual void f(int);
        virtual void f(double);
        //...
};

class C : public B {
public:
        using B::f;   // make B::f(int) and B::f(double)
                      // available in C
        virtual void f(int);  // override B::f(int)
        virtual void f(char); // add new f
        //...
};
```

Note that this allows the programmer to request overloading of base and derived class member functions. For purposes of overload resolution, a function from a base class made available by a *using-declaration* is considered a member of the derived class. For example:

```
void f(C* p)
{
        p->f(2.0); // call B::f(double);
}
```

A *using-directive* may not be a member declaration. A *using-declaration* used as a member declaration must refer to accessible base classes and/or accessible members of base classes. A namespace that is not a class cannot be defined within a class. A namespace that is a class cannot have names added by further namespace declarations.

§r.3.3.1.4 Operator lookup: Operators are looked up in a context consisting of both the context of their use and the context of their operand. Ordinary overload resolution is used to select from the set of all operators found. The context of an operand consists of the namespace in which its type was declared and the namespaces of any bases of that type. For example:

```
namespace My {
        class string { /* ... */ };
        std::ostream& operator<<(std::ostream&,string);
        // ...
};
```

```
void f(My::string& s)
{
        std::cout << s; // invoke My's <<
}
```

§r.3.3.1.5 Deprecated features: Unnamed namespaces make global `static` (§r.7.1.1) redundant. For example:

```
namespace {
        void f(); // accessible from this translation
                  // unit; inaccessible from other units
}
```

*Using-directive*s make *access-declaration*s (§r.11.3) redundant. For example:

```
class X {
public:
        void f();
};

class Y : private X {
        // by default, X::f() is inaccessible through Y
public:
        using X::f; // make X::f accessible through Y
}
```

Global `static` and *access-declaration*s are deprecated. That is, these features are slated to be removed in a future revision of the C++ standard.

§r.3.4: If control reaches the end of `main()` the effect is that of executing `return 0;` that is, indicating successful completion.

§r.3.6.1: `bool` is a unique signed integral type. A `bool` value may be converted to `int` by promotion: `true` converts to `1`, `false` converts to `0`. A numeric or pointer value may be implicitly converted to `bool`. A zero value becomes `false`; a non-zero value becomes `true`. The relational operators yield `bool` values.

§r.5.2: Operators for inquiring about type at runtime have been added:

postfix-expression:
    ... old stuff
    `static_cast <` *type-name* `> (` *expression* `)`
    `reinterpret_cast <` *type-name* `> (` *expression* `)`
    `const_cast <` *type-name* `> (` *expression* `)`
    `dynamic_cast <` *type-name* `> (` *expression* `)`
    `typeid (` *expression* `)`
    `typeid (` *type-name* `)`

§r.5.2.3: If a type `T` does not have an explicitly defined default constructor then `T()` is allowed if and only if a static variable of type `T` can be defined without an initializer, and the value of `T()` is the default value of such a static variable.

§r.5.2.4: In addition to the `(T)e` cast notation, three cast operators `static_cast`, `reinterpret_cast`, and `const_cast` are provided to express specific cast operations directly. Between them, these casts can express any operation `(T)e` can, except casting to a private base class. The purpose of these new casts is to make the intent of a cast explicit.

`const_cast<T>(e)` casts `e` to a `T` exactly as `(T)e` would have done provided `T` and the type of `e` differ only in `const` and `volatile` modifiers. Otherwise, `const_cast<T>(e)` is an error. Thus, `const_cast` manipulates ‘‘constness’’ only and not other aspects of type. To complement, both `static_cast` and `reinterpret_cast` respect ‘‘constness’’ in the sense that `const` and `volatile` modifiers may not be changed for pointers or references in a way that is not allowed in an implicit conversion.

`static_cast<T>(e)` casts `e` to a `T` exactly as `(T)e` would have done provided there is an implicit conversion from `T` to the type of `e` or an implicit conversion from the type of `e` to `T`. Otherwise, `static_cast<T>(e)` is an error. Note that `static_cast` is relatively well behaved and yields values that are usable without further casting.

`reinterpret_cast<T>(e)` casts `e` to a `T` provided `(T)e` is allowed. Otherwise, `reinterpret_cast<T>(e)` is an error. `Reinterpret_cast` treats pointer and reference types as if they were incomplete types (that is, base/derived relationships do not affect the meaning of `reinterpret_cast`); otherwise the meaning of `reinterpret_cast<T>(e)` is the same as `(T)e`. Note that `reinterpret_cast` yields values that are typically not guaranteed to be usable without casting them back to their original types; pointers to functions is an example of this: it is *not* safe to call a function through a pointer to function with argument types that differ from those declared in the function definition. For example:

```
void f(char* p) { *p = 'x'; }
typedef void (*FP)(const char*);
FP p = static_cast<FP>(&f);
```

A call of `f` through `p` is not guaranteed to work (in this, C++ agrees with ISO C).

§r.5.2.6: The result of `dynamic_cast<T>(v)` is of type `T`. The type `T` must be a pointer or a reference to a defined class or `void*`.

If `T` is a pointer type and `v` is a pointer of a type for which `T` is an accessible base class, the result is a pointer to the unique sub-object of that class. If `T` is a reference and `v` is an object of a type for which `T` is an accessible base class, the result is a reference to the unique sub-object of that class. Otherwise, `v` must be a pointer or reference to a polymorphic type. That is, the type pointed or referred to must have at least one virtual function.

If `T` is `void*` then `v` must be a pointer, and the result value is a pointer to the complete object (§r.12.6.2) pointed to by `v`.

Otherwise, a run-time check is applied to see if the value `v` can be converted to a `T`. If not, the cast is said to have failed. The result value of a failed cast to pointer type is `0`. A cast to reference type throws `bad_cast` if it fails.

The run-time check logically executes like this: If the object pointed (referred) to by `v` is an object representing a base class of an object of type pointed (referred) to by `T` the result is a pointer (reference) to that object. Otherwise, find the complete object (§r.12.6.2) pointed to by `v`. If that object has a unique public base class of type `T`, the result is a pointer (reference) to the object representing that base class. Otherwise, the run-time check fails. For example:

```
class B { /* at least one virtual function */ };
class D : public B {  /* ... */ };

B* pb1 = new B;
B* pb2 = new D;

D* pd1 = dynamic_cast<D*>(pb1);
D* pd2 = dynamic_cast<D*>(pb2);
```

will leave `pd1` with the value `0` and `pd2` pointing to the `D` object.

§r.5.2.7: The result of a `typeid` expression is of type `const type_info&`. The value is a reference to a `type_info` that represents the *type-name* or the type of the *expression* respectively.

Where the *expression* is a reference to a polymorphic type, the `type_info` for the complete object referred to is the result. Where the *expression* is a pointer to a polymorphic type dereferenced using `*` or `[`*expression*`]`, the `type_info` for the complete object pointed to is the result provided the pointer is not `0`; otherwise `bad_typeid` is thrown. Otherwise, the result is the `type_info` representing the (static) type of the *expression*. For example:

```
class X { /* ... */ };

void g(X* p)
{
        typeid(p);  // type_info(X*)
        typeid(*p); // type_info for X or
                    // class derived from X
}
```

Class `type_info` is declared in `<typeinfo>` like this:

```
class type_info {
public:
        virtual ~type_info();  // type_info is polymorphic

        bool operator==(const type_info&) const;
        bool operator!=(const type_info&) const;
        bool before(const type_info&) const;

        const char* name() const;
private:
        type_info(const type_info&);
        type_info& operator=(const type_info&);

        // implementation-dependent representation
};
```

The ordering defined by `before()` is complete and valid only for the duration of the execution of a program. There is no guaranteed relation between the relationships defined by `before()` and inheritance relationships. The purpose of `before()` is to allow sorting of `type_info` objects .

The naming conventions for library classes such as `type_info` and library headers such as `typeinfo` are still being debated. I have used the names considered the most likely outcomes of that discussion.

§r.5.3.2: A type may not be defined in a `sizeof` expression. For example:

```
sizeof(struct S { int a,b; }) // error: struct defined
                              // in sizeof
```

§r.5.3.3, 5.3.4: The `new` operator will call the function `operator new[]()` to obtain storage for an array (§r.12.5). To free the storage used for an array, operator `delete[]` will call the function `operator delete[]()` (§r.12.5).

One can `delete` a pointer to constant.

An `operator delete` function can be declared with additional arguments after the required `void*` argument. If an exception occur during the construction of an object allocated using the placement syntax (§r.5.3.3), the `operator delete()` function with argument types matching the `operator new()` used, if declared, will be invoked to delete the memory. The arguments of an `operator delete()` and a `operator new()` match if their second and subsequent argument types are identical. For example:

```
class My_heap;

class X {
        X() { throw "Boom!"; }
        void* operator new(size_t,void*);
        void* operator new(size_t,My_heap&);
        void operator delete (void*,My_heap&);
};

void f(My_heap& h, void* p)
{
        X* p1 = new(My_heap) X();
                // memory allocated by
                // X::operator new(size_t,My_heap&)
                // and freed by
                // X::operator delete(void*,My_heap&
        X* p2 = new(p) X();
                // memory allocated
                // by X::operator new(size_t,void*)
                // and not freed
}
```

This allows a programmer to make programs using placement new exception safe.

§r.5.4: A value of integral type may be explicitly converted to an enumeration type. The result of such a conversion is undefined unless the value is within the range of the enumeration (§r.7.2). For example:

```
enum flag { a=1, b=2, c=4, d=8 };

flag f1 = 3; // type error: 3 is not of type flag

flag f2 = flag(d|c); // OK: flag(12) is of type flag
                     // and within the range of flag
flag f3 = flag(99);  // undefined: 99 is not within
                     // the range of flag
```

§r.5.9, 5.10, 5.16: Pointer conversions are performed on pointer operands of the relational, equality, and conditional operators to bring them to the same type, which must be the qualified or unqualified version of the type of one of the arguments.

§r.5.16, 5.17, 17.2: The syntax for the *conditional-expression* and the *assignment-expression* have been changed to the following:

conditional-expression:
    logical-or-expression
    logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator assignment-expression
  `throw` *assignment-expression*$_{opt}$

For example:

```
throw a, b;      // means (throw a),b
i ? x=y : x=z;   // means i?(x=y):(x=z)
p->*m = 0;       // means (p->*pm) = 0
```

Note that the sub-expression `b` will not be executed.

§r.6: Declarations may appear in conditions:

selection-statement:
  `if (` *condition* `)` *statement*
  `if (` *condition* `)` *statement* `else` *statement*
  `switch (` *condition* `)` *statement*

iteration-statement:
  `while (` *condition* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  `for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

condition:
  expression
  type-specifier declarator = *expression*

The *declarator* may not specify a function or an array. The *type-specifier* may not define a class or an enumeration.

A name introduced by a declaration in a *condition* or a *for-init-statement* is in scope from its point of declaration until the end of the statements controlled by the condition. The value of a *condition* that is an initialized declaration is the value of the initialized variable. For example:

```
for (int i = f(j,k); int c = g(i); i++) {
        // use i and c here
}
// i and c are not in scope here
```

A variable, constant, etc. in the outermost block of a statement controlled by a condition may not have the same name as a variable, constant, etc. declared in the condition.

If a *condition* can be syntactically resolved as either an expression or a declaration of a local name, it is interpreted as a declaration.

§r.7.1.1: The `extern` specifier cannot be applied to an anonymous union.

`mutable` is a storage class specifier that can be applied to a non-`const` non-

`static` data member of a class. A member specified `mutable` behaves like others members except that it is never `const` – not even when it is part of a `const` object.

§r.7.1.6: The rule that if a *type-specifier* is missing in a declaration then `int` is assumed is removed. That is, every declaration must specify a type. For example:

```
f() { /* ... */ } // error: return type missing
```

Except that any class member declared `mutable` may be modified, any attempt to modify a `const` object after it has been initialized and before it is destroyed results in undefined behavior. A `const` object with a constructor is initialized after completion of the constructor. A `const` object with a destructor is considered destroyed upon entry into the destructor.

§r.7.2: Enumerators used to have to be `int`s, and assignment of a value that was not an enumerator to an enum variable used to be undefined. The rules for enumerations were amended to allow `long` initializers, and the common usage of assigning suitably cast non-enumerator values to enum variables was accepted:

An enumerator can be initialized by a *constant-expression* of *integral type*. The range of an enumeration holds all the enumeration's enumerator values rounded up to the nearest larger binary power minus `1` and down to `0` if the smallest enumerator is non-negative and to the nearest smaller binary power plus `1` if the smallest enumerator is negative. This defines the smallest bit-field capable of holding the enumerator values. For example:

```
enum e1 { dark, light }  // range 0:1
enum e2 { a=3, b=9 }     // range 0:15
enum e3 { a= -10, b = 1000000 }  // range -15:1048575
```

§r.8.2.6: Default argument expressions for non-member functions have their names bound and their types checked at the point of declaration, and are evaluated at each point of call. Default argument expressions for member functions have their names bound and their types checked at the end of the class declaration; this treatment is like that of names in the bodies of inline member functions. (§r.9.3.2).

§r.9.1: A declaration of a class or function declared as a friend will be entered into the nearest enclosing non-class scope unless it has already been declared in a scope enclosing the friend declaration. For an example:

```
class X {
        friend void f();
        // ...
};

void g() { f(); } // call X's friend f()

void f() { /* ... */ } // this f() is the friend of X
```

Similarly, a declaration of a class name used in an *elaborated-type-specifier* that declares a function or an object will be entered into the nearest enclosing non-class scope if it has not already been declared in a scope enclosing the declaration introducing the class name. For example:

```
void f(class Y* p1) { /* ... */ }

Y* p2; // same Y as in f()
```

§r.9.3.2: The aim of the ‘‘rewrite rule’’ for inline functions (§r.9.3.2) and the ‘‘no redefinition after use’’ rule for type names (§r.9.9) was to limit order dependence and dependence on names outside the scope of the class. This was only partially successful and a set of more precise, more comprehensive, and easier to understand rules has been adopted:

[1] The scope of a name declared in a class consists not only of the text following the name's declarator but also of all function bodies, default arguments, and constructor initializers in that class (including such things in nested classes). It excludes the name's own declarator.

[2] A name used in a class `S` must refer to the same declaration when re-evaluated in its context and in the completed scope of `S`. The completed scope of `S` consists of the class `S`, `S`'s base classes, and all classes enclosing `S`.

[3] If reordering member declarations in a class yields an alternate valid program under [1] and [2], the program's meaning is undefined.

Note that only very few programs are affected by this change of rules. The new rules are primarily a clearer statement of the original intent. At first glance these rules seem to require a multi-pass algorithm in a C++ implementation. However, they can be implemented by a single pass followed by one or more passes over information gathered during the first pass, and do not cause a performance bottleneck.

§r.9.4: The initializer for a static data member can access anything that a static member function of the class can. The initializer of a static member of `Z` is in the scope of `Z`; thus

```
class Z {
        static int m1, m2;
        // ...
};

int Z::m1 = m2; // m2 is Z::m2
```

A static `const` member of integral type may be initialized by a constant expression in its member declaration within the class declaration. If a member is initialized in-class it cannot also be initialized in its definition. A member initialized in-class must still be defined elsewhere. For example:

```
int i = 7;

class X {
        static const int a1 = 1;
        const int a2 = 2;          // error: non-static
                                   // initialized.
        static int a3 = 3;         // error: non-const
                                   // initialized.
        static const float a4 = 4; // error: non-integral
                                   // initialized.
        static const int a5 = i;   // error: non-const
                                   // initializer.
        // ...
};

const int X::a1; // definition
```

§r.9.7: Like a member function, a nested class may be declared within the enclosing class and defined elsewhere in the same or containing scope, using the qualified name for the class definition:

class-head:
  class-key identifier$_{opt}$ *base-spec*$_{opt}$
  class-key complete-class-name base-spec$_{opt}$

For example:

```
class enclose {
        class inner;
        // ...
};


class enclose::inner {
        // ...
};
```

§r.10.2: To override a virtual function, a function in a derived class was required to have the same type as the base class virtual function. This rule has been relaxed to allow an overriding function's return type to differ: If the derived class function's return type is `D*` or `D&`, and the base class function's return type is `B*` or `B&`, respectively, and `B` is an accessible base class of `D`, then overriding takes place. For example,

```
class B {
        // ...
        virtual B* f();
};

class D : public B {
        // ...
        D* f(); // formerly an error,
                // now D::f overrides B::f
        void g();
};

void f(D* pd,B* pb)
{
        pd->f()->g();   // now allowed
        pb->f()->g();   // error: B::f() returns a B*
}
```

Note that there is no relaxation of the requirement for exact match for argument types. Such a relaxation would require implicit run-time type checking.

§r.10.3: The base classes of an abstract class need not be abstract. Further, a pure virtual function may override a virtual function that is not pure.

§r.11.2: A base class may be `protected`. Access to a protected base is the same as to a protected member.

§r.12.2: The life time of a temporary object extend to the end of the full expression in which it is created, except when used as part of an initializer for a reference. In the latter case, it lives as long as the reference. A full expression is an expression that is not a sub-expression of any other expression.

§r.12.3.1: A single-argument constructor declared with the prefix `explicit` will be used for explicit construction only and not for implicit conversions. For example:

```
class String {
public:
        explicit String(char*);
        // ...
};
```

```
void f(const String&);

void g(char * p)
{
        String s1(p);  // ok: explicit construction.
        String s2 = p; // error: implicit
                       //        conversion required.
        String s3 = String(p); // ok: explicit
                               //     construction.
        f(p);          // error: implicit
                       //        conversion required.
        f(String(p)); // ok: explicit construction.
}
```

This mechanism can be used to prevent ‘‘surprising’’ conversions and ambiguities. It is expected that `explicit` will be allowed for conversions specified as operator functions also.

§r.12.4: A destructor may not be invoked more than once for an object.

§r.12.5: Two operators are introduced specifically to handle allocation and deallocation of arrays:

```
void* operator new[](size_t);
void operator delete[](void*);
void operator delete[](void*,size_t);
```

The use of the two forms of `delete[]` corresponds to their `delete` equivalents.

§r.13.4: An operator may be overloaded based on an enumeration. For example:

```
enum Season {  winter, spring, summer, fall };

Season operator++(Season s)
{
        int i = int(s)+1;
        return (i>fall) ? winter : Season(i);
}
```

§r.14: Templates as described in §r.14 were accepted by the committee.

A function template has external linkage. A static member of a class template has external linkage. If a function template appears in more than one compilation unit, it must have the same definition.

A template can be a non-virtual member of a class or a class template.

A class template can be a template argument. For example:

```
template<class T, template<class U> class S> class X {
        S<T> s1;
        // ...
};

template<class T> class Vector { /* ... */ };

X<int,Vector> xiv;
```

§r.14.4: A function template can have non-type template arguments. For example:

```
template<int i> void f(bits<i>& v) { /* ... */ }
```

As for type arguments, the value of a non-type argument must be deducible at compile time for each call of the template function.

The usual conversions are applied to arguments not involved in the deduction of a template argument.

The following conversions are allowed for arguments used to deduce a template argument:

[1] For a parameter of the form `B<params>`, where `params` is a template argument list containing one or more deduced parameters, and argument of a class derived from `B<params>` may be converted to a `B<params>`. Additionally, for a parameter of the form `B<params>*`, an argument of type pointer to a class derived from `B<params>` may be converted to a `B<params>*`. Similarly for references.

[2] A pointer (reference) may be converted to a more qualified pointer (reference) type according to the rules in §r.4.6 (§r.4.7).

[3] ‘‘Array of `T`’’ to ‘‘pointer to `T`’’.

[4] ‘‘Function ...’’ to ‘‘pointer to function ...’’.

For example:

```
template<class T> struct B { };
template<class T> struct D : B<T> { };

template<class T> void f(const B<T>*);


void g(B<int>* pb, D<int>* pd)
{
        f(pb);
        f(pd); // ok: f( (B<int>*)pd )
}
```

If more than one template function could be chosen for a call, the most specific template function is chosen. For example:

```
template<class T> void f(T);           // #1
template<class T> void f(T*);          // #2
template<class T> void f(vector<T>*);  // #3

void g(int i, char* pc, vector<int>* pv)
{
        f(i);   // use #1
        f(pc);  // use #2
        f(pv);  // use #3
}
```

It is possible to overload template functions so that specialization of two different template functions have the same type. For example:

```
// file1.c:
template<class T> void f(T);  // #1
void g(int* p)
{
        f(p);  // call #1
}

// file2.c:
template<class T> void f(T*);  // #2
void g(int* p)
{
        f(p);  // call #2
}
```

Such specializations are distinct functions and do not violate the one-definition rule.

§r.14.5: Clarifications: A template function is generated (instantiated) at the point where it is first called or its address is first taken. A template class is generated (instantiated) at the point where its name is first mentioned. More precisely, the point of instantiation of a template is (as ever) immediately before the external declaration that first names it. This would seem to imply that a global name used from within a template could be bound to different objects or functions in different compilation units or even at different points within a compilation unit. However, should that happen, the resulting template function or class is rendered illegal by the ‘‘one-definition’’ rule (§r.7.1.2).

A non-function name in a template definition is bound to a declaration at the template’s point of definition. If no such declaration is found the program has an error. A function name used explicitly or implicitly in a template definition is bound to a declarations in scope at the template’s point of definition, provided there is such a declaration and provided the types of all arguments are known and allow a call to the declared function. Other functions calls are bound at the point of instantiation and these calls must depend on a template argument. For example:

```
void g(int);

template<class T> void f(T t)
{
        g(1);      // ok: call g(int).

        g(t);      // cannot be bound yet
                   // later: ok, t depends
                   // on template argument.

        g("asdf"); // cannot be bound yet
                   // later: error, the call
                   // does not depend on
                   // template argument.
}

void g(char*);
void g(double);

void h(double d)
{
        f(d); // cause f<double>() to call g(double)
}
```

Avoid writing code where it is not clear what names are bound where.

In a template, a name is assumed to be the name of a non-type unless it has been explicitly specified to refer to a type. The keyword `typename` can be used to specify that an immediately following qualified name is a type name. For example:

```
template<class T> class B {
        T* p1;    // ok
        T::x* p2; // error: ``T::x'' is not a type name
        typename T::x* p3; // ok
        typedef typename T::x Tx;
        Tx* p4;   // ok
};
```

A specialization must be declared before it is used. For example:

```
template<class T> void sort(vector<T>& v)
        { /* ... */ }

void sort(vector<char*>& v) { /* ... */ }

void f(vector<char*>& v1,vector<String>& v2)
{
        sort(v1); // use specialization
                  // sort(vector<char*>&)
```

```
        sort(v2); // use general template
                  // sort(vector<T>&), T is String
}

void sort(vector<String>& v) // error:
                             // specialize after use
        { /* ... */ }
```

If a specialization is declared and used, its definition must be provided by the programmer. If a specialization is declared, a declaration of it must be in scope for every use of the template with that set of template arguments. An ordinary function declaration is not a specialization and does not suppress the generation of definitions from a function template with the same name and matching argument types.

A specialization of a template can itself be a template (a ‘‘partial specialization’’). A primary class template declaration is one where the class template name is an *identifier*. A class template declaration where the class template name is a *template-id* is a partial specialization of the template named in the *template-id*. The *template-parameters* for the primary class template determine which *template-arguments* a user must supply in a use of the template. The most specific version of a template is chosen for each use of the template. For example:

```
template<class T> class List { /* ... */ };  // primary

template<class T>
class List<T*> { /* ... */ };  // partial specialization
                               // for pointers
class List<void*> { /* ... */ };  // specialization
                                  // for void*

List<int> li;     // use primary
List<int*> lip;   // use partial specialization
List<void*> lvp;  // use void* specialization
```

§r.14.8: Initializers can be specified for static members of template classes. Given:

```
template<class T> class X {
        static T s;
        // ...
};
```

The static member `s` can be defined for a template class `X<int>` like this:

```
int X<int>::s = 997;
```

Further, one can specify a default value for `X<T>::s` in general like this:

```
template<class T> T X<T>::s = 2;
```

That is, if no initializer is given for a template class `X<T>`, its initial value will default to 2.

§r.14.9 Explicit instantiation: A user can explicitly indicate that a particular version of a template should be instantiated. For example:

```
template void sort(vector<int>&); // instantiate function
template class X<int>; // instantiate class X, including
                       // all member functions and
                       // static members.
```

Duplicate explicit instantiation of a specific template (such as `X<int>`) is technically an error, but an implementation is not obliged to cause the linking of a program to fail because of it.

§r.14.9 Explicit qualification: Zero or more template arguments may be explicitly specified in a template function call. For example:

```
template<class T, class V> T convert(V v) { return v; }

void f(int i, const char* p)
{
        convert<double>(i); // ok
        convert(i);         // error: T unspecified
        convert<double>(p); // error: cannot convert
                            // char* to double
}
```

If function template arguments are specified in a call, they are specified in declaration order. Trailing template arguments may be left out. Any template argument not explicitly specified must be deduced according to the ordinary rules for template function arguments. Overload resolution is handled by first selecting templates based on the explicit template arguments and then using the usual rules for further resolution.

A < after a template name is considered the start of a *template-arg-list*.

The keyword `template` can be used after `.` `->` and `::` to indicate that a name is supposed to be the name of a member template. This usage is needed where the syntax analyzer cannot recognize the name as a template name without help from the programmer. For example:

```
class X {
public:
        template<size_t> X* alloc();
};
```

```
void f(X* p)
{
        X* q = p->alloc<200>();  // error:
                                 // < means less than
        q = p->template alloc<200>() // ok: < starts
                                     // explicit
                                     // qualification
}
```

§r.15: Exceptions as described in §r.15 were accepted by the committee.

§r.15.4: The following rules strengthens both the static and the dynamic checking of *exception-specification*s compared to the original design:

A pointer to function can have an *exception-specification*. For example:

```
void f() throw(int);          // ok
void (*fp)() throw(int);      // ok
void(void af() throw(int));   // ok
```

If a virtual function has an *exception-specification*, all declarations of that function shall have an *exception-specification* with the same set of exceptions. If a virtual function has an *exception-specification*, an overriding function must have an *exception-specification* at least as restrictive as that in the base class. Similarly, any pointer to function assigned to, or initializing, a pointer to function shall have an *exception-specification* at least as restrictive as that of the pointer to function assigned to. For example:

```
void (*pf1)();      // no exception specification
void (*pf2)() throw(A);

void g()
{
        pf1 = pf2;  // ok: pf1 is less restrictive
        pf2 = pf1;  // error: pf2 is more restrictive
}
```

In such assignments and initializations, exception specifications on return types and parameters shall match exactly.

In other assignments or initializations, *exception-specification*s shall match exactly.

Calling a function through a declaration whose *exception-specification* is less restrictive is an error. No diagnostic is required.

An `unexpected()` function may throw an exception that satisfy the *exception-specification* that triggered the call of `unexpected`, and in this case the search for a handler will continue. If an `unexpected()` function throws an exception not in that *exception-specification* then the exception `Xunexpected` will be thrown provided `Xunexpected` was specified in that *exception-specification*; otherwise `terminate()` is called.

Thus, an *exception-specification* guarantees that only the listed exceptions will be thrown.

The body of a function can be a *try-block*. If the body of a constructor is a *try-block* the `try` goes before the base and member initializers. For example:

```
int main()
try { /* ... */ } catch(...) { /* ... */ }

X::X(int a)
try :B(1), mem(a) { /* ... */ ) catch(Berr) { /* ... */ }
```

This allows exceptions thrown by base or member initializer to be caught.

ffasfdfaffdd