

Doc No: SC22/WG21/N????

J16/06-????

Date: July ??, 2006

Project: JTC1.22.32

Reply to: Bjarne Stroustrup

Computer Science Dept.

Texas A&M University, TAMU 3112

College Station TX USA 77843-3112

Fax: +1-979-458-0718

Email: bs@cs.tamu.edu

Control of class defaults

Bjarne Stroustrup
Texas A&M University

Abstract

This paper presents a solution to the problem of how to control default behavior of classes and objects of classes. The solution is based on a general “default behavior specification” mechanism plus suggestions for a set of default behaviors that can be controlled by users. In addition to the usual defaults, control of derivation, generation of `==` and `!=`, and specification of **virtual** and **explicit** members are proposed. Many further alternatives are discussed (but rejected).

Caveat: This is still a draft, writing in haste to allow discussion at the UK meeting. As ever, constructive comments are actively sought.

The discussion is based on the earlier papers (notably Glassborow’s N1582, Glassborow & Goldthwaith’s N1702 and Stroustrup & Dos Reis’ N1890) and on discussions in the evolutions group.

1 What’s the problem?

From C, C++ inherited the idea that objects of a **struct** by default have some operations defined. In particular, we can

- copy a **struct** value (in initialization and assignment)

- define a **struct** object with a default value
- take the address of a **struct** object
- use the comma operator for a **struct** object

For the **struct** itself, we have the operations

- Define an object on the stack
- Define an object statically

C++ added to the set of default operations on objects. For example, we can

- Destroy an object (explicitly or implicitly)
- Implicitly apply single-argument constructor (suppressed by **explicit**)

In addition, C++ added default operations for the **class** itself

- Derive from a class
- Allocate an object of a class on the free store

There are more, but they don't appear to be important so I'll just list them much later to avoid distracting from the main argument.

Remembering all the defaults can be hard. Quick: If I define only a copy constructor do I still get the default constructor? This complexity is a problem for novices (especially for novices who have not been trained to expect the defaults by coming from a C background) and can be a maintenance problem (though we have only anecdotal evidence). Many of these "default behaviors" can be changed by defining the right functions (e.g. default constructors and copy constructors) and some of the default behaviors can be suppressed (and often are) by declaring their corresponding operations private. In conclusion:

- sometimes, a default is not suitable, so we want to suppress its use
- sometimes, we want to specify our own version of one of these operations (writing the appropriate function)
- sometimes, we want to be explicit about using the default.

The last reason is reinforced by myths about the suitability of the default copy operations: some people think that they can write them better than the compiler can generate them, some people doubt that compilers generate them correctly, and some people note that occasionally the generated copy operations are unfortunate. Sometimes, they are indeed unsuitable (typically when a class has a destructor), but when they are suitable, a user cannot define them better than the compiler.

This raises the design questions:

- "How can you suppress defaults when they are not needed or not suitable?" ("give me the usual except the following: ...")
- "How can you explicitly say that you want some or all of the defaults?" ("give me nothing except the following: ...")

There have been suggestions for additional "default semantics" or facilities, such as

- make all functions **virtual**
- implicitly define == and != based on member operations
- implicitly define < based on member operations
- implicitly define += based on + and =

- allow **x.f()** to be called as **f(x)**

Some of these questions, such as why can we use = but not == has been asked by essentially everybody since the beginning of C. The suggestions for adding to the set of defaults raise the design question:

- “How can you request additional defaults?” (“Given me the usual plus the following: ...”)

Consequently, I propose a mechanism for stating a list of defaults, for subtracting from the set of defaults, and for adding to the list of defaults.

2 Naming the defaults

First, we need a way of referring to each of the default operations. We could use function signatures, but that’s verbose and indirect. Because it is verbose and indirect, we open the possibility of error. Consider this example using the notation from N1702:

```
explicit class X {    // “explicit” means
                    // “no defaults constructors, assignment, or destructor”
    X(int);
    X(X&) { default }
    X& operator=(X&) { default }
    // ...
};

const X x1(2);
X x2(x1);    // error: can’t bind a const to a plain reference
X x3(x2);    // ok
```

This mistake is basically harmless; it simply leads to otherwise acceptable uses being rejected by the compiler. The programmer can fix the code, as ever. However, it would be nice not to open such opportunities.

Second, we would like to address all the problems with defaults or at least all the problems that arise frequently. We have no objective measures for “arise frequently” and for whom (novices and experienced developers are likely to have different patterns), but I will propose this

- Frequent: prevent copy operations, default constructor, static allocation
- Infrequent: prevent comma, address-of, stack allocation

In addition, requests for forcing heap allocation, preventing derivation, and generating “the obvious” ==, < operations are frequent.

Most importantly, if we choose to base our solution on specifying operations, we would need a separate mechanism to dealing with class operations or sets of operations. The keyword **final** comes to mind for disabling derivation. Alternatively, we could follow N1702 and decide not to address any problems beyond the copy operations, default

constructor, and destructor. I would prefer a more general solution if we can find a sufficiently elegant one.

If we don't use function declarations (possibly augmented with keywords, such as `final`), we need a set of shorthand names for the default behaviors. Here is a suggestion for "names" for default behaviors:

:	Derivation (so its absence is what is referred to as <code>final</code> in Java)
=	Copy (by constructor and by assignment)
&	Address of (only explicit use)
,	Comma (not the comma in argument lists)
()	Default construction
~	Default destruction
new	Free store allocation
delete	Free store deallocation
static	Static allocation
virtual	All member functions virtual
=0	All member functions pure virtual
explicit	All constructors and conversion operators explicit
==	Equality and inequality
default	All the usual defaults

I will get to the potential new defaults (e.g. `==`) later.

This naming scheme do not provide separate names for copy construction and copy assignment. Consequently, we cannot restore or suppress the default meaning of the copy constructor without also affecting the copy assignment operator (and vice versa). This could be resolved by some more "magic" syntax, but I suspect that having a single name for "copy" is actually a good thing because cases where we want the one copy operation default and not the other are hard to imagine.

3 Controlling defaults

The idea is to express defaults by listing them either as a complete list:

```
default { = () } // default copy and default construction only
```

Or by additions or subtractions to the set of defaults:

```
default - { : } // subtract (disallow): inheritance
default + { explicit } // add: make all single-argument constructors explicit
```

The trickiest design problem here is to make it simple to state the two edge cases: give me the usual defaults and give me no defaults. This is the suggestion:

```
default { } // give no defaults
```

```
default { default } // give all the usual defaults
```

This gives us a mechanism for stating in absolute terms a desired set of defaults and for building up a set of defaults by additions and subtractions. Both features have been frequently asked for, though of course not in such general terms. Note that the design addresses and combines two schools of thoughts:

- Full control: Let me specify everything (but save me from implementation mistakes)
- Modification: I'm happy with the defaults except please suppress (add) this one.

Where should we put the default behavior specifiers? There are two obvious alternatives:

- As part of the class header:
 - to make changes from the default highly visible
 - because we are making changes to several operations (e.g. both copy operations)
- As part of the class body:
 - because we are declaring/defining operations
 - because default behavior specifiers are large enough to make class headers messy
 - because we could apply access control
- either place or both
 - to give people a choice.

Consider a few examples chosen as likely in real code. First the suppression of the default copy operations:

```
class YY { // in body  
  default -{ = } // no copy  
  // ...  
};
```

or

```
default -{ = } class YY { // as prefix in head  
  // ...  
};
```

or

```
class YY default -{ = } { // as suffix in head  
  // ...  
};
```

Next an explicit specification of all desired defaults:

```
class YY { // in body
```

```

    default { : & new () } // inheritance, address of,
                        // free store allocation, default construction
    // ...
};

```

or

```

default { : & new () } class YY { // as prefix in head
    // ...
};

```

or

```

class YY default { : & new () } { // as suffix in head

    // ...
};

```

A N1702-style explicit class:

```

class YY { // in body
    default { default }-{() = ~} // the usual defaults except
                                // default construction, copy, and destructor
    // ...
};

```

or

```

default { default }-{() = ~} class YY { // as prefix in head
    // ...
};

```

I think “in body” is the clear winner on aesthetic grounds.

1.1 *Syntax alternatives*

It could be imagined that a default behavior specification could be “lost in the details” deep in a class body. This would be similar to what happens to functions changing the meaning of defaults today. I don’t think that will be a problem with default specifications because they are syntactically distinct.

If needed, we could require the **explicit** keyword in the class header to announce the presence of a default behavior specification. Alternatively, we could require a default behavior specification to precede all other declarations. I don’t think there is a need.

We could have a rule that kept default behavior specification at the top of the class: simply decree that they must precede ordinary declarations. I don't think that's necessary – it would be a style rule enforced by the language.

Obviously I chose **default** to avoid introducing a new keyword. It would be tempting to leave out the { } after the default, but then we'd need a terminator. It is tempting to use a **default:** (note the colon) but then we'd need a terminator and a different notation for derivation.

Space is unusual as a separator in C++, but we can't use comma as it is one of the default operators. Using space as the separator preserves the usual tokenization and using the usual tokens preserves a degree of familiarity. Lois Goldthwaite has with reasonable accuracy referred to the scheme as “hieroglyphics”. However, given the size of the set of defaults, I think it is important that the specification of each is short. The alternatives that avoid the “hieroglyphics” require about a line per operation. Such verbosity is in itself a problem. Also, I predict that the syntax will seem “normal” after a short period to gain familiarity. Often familiarity is confused with simplicity. In this case simplicity (the proposed syntax is minimal and corresponds directly to the desired semantics) should help it become familiar.

I have allowed both multiple lists in a single default behavior specification and multiple default behavior specification in a class. For example:

```
class YY { // in body
    default { default }- { = }+ { virtual } // the usual defaults except copy,
                                           // make all member functions virtual
    // ...
};
```

And

```
class YY { // in body
    default { default } // the usual defaults
    default - { = } // except copy,
    default + { virtual } // make all member functions virtual
    // ...
};
```

We could enforce one style or the other, but I don't see the need.

1.2 Access

Can we make the default behavior specifications obey access control rules; that, it make the operations they define **protected** or **private**? Yes, but I don't propose to use this notation for that: if you want, say, a private default constructor, just declare one. However, I conjecture that such definitions would be rare – the most common use of private constructors is to inhibit copying, and that we can express directly with default

behavior specifications. If this conjecture is wrong; that is, there are important use cases where, say a protected default behavior specification is a significant advantage, the decision should be reversed, but we'd need such use cases.

If we wanted to use default behavior specifications together with access control (e.g. to be able to specify that the default copy operations should be **protected**) we would need to allow multiple default behavior specifications in a class. Without that it is only a convenience.

For stylistic reasons and maintainability, I prefer default behavior specifications to be in one place (whether as a single declaration or as a consecutive set of declarations) near the top of a class declaration. Having them subject to access control would lead to them being spread out in the class declaration.

4 Defaults and declarations

What happens if a programmer specifies a default behavior and also define one of the functions controlling that behavior? For example, we might try to say that we want the default copy behavior but also define **operator=()**. We have three design alternatives:

- Ban combinations
- A declaration overrides a default behavior specification
- A default behavior specification overrides a declaration
- It depends on the individual behavior

I think that the first answer is the simplest and least error prone: You can either assert control through the usual declarations or ask for the default, but not both. Obviously, you can control the default for one behavior, such as copying, with declarations while controlling unrelated behavior, such as default construction, with a default behavior specification. For example:

```
class X {
    default ~{ = }
    X& operator=(const X&); // ok, we said we didn't want that default
    // ...
};

class Y {
    default { default }
    Y& operator=(const Y&); // error, we said we wanted that default
    // ...
};

class Z {
    Z& operator=(const Z&); // ok, we said we didn't say
                           // anything about that default
    // ...
};
```

Note that there is a difference between getting the defaults by saying nothing and explicitly requesting the defaults. I think that's right. If I see an explicit request for default behavior (such as, **default { default }**) it would be quite a surprise for me to find that the default was in fact overridden by a declaration hidden deep in the class body.

5 Inheritance of defaults

Can we inherit defaults? Yes, by default we do; that is, a derived class gets the union of the defaults of its bases. However, we can override base class defaults in the derived class. This form of “overriding” is needed to deal with multiple inheritance and pure virtual functions. For example:

```
class X {
    default+{ == }
    // ...
};

class XX : public X {
    // ...
};
```

Now **XX** also have default equality.

```
class Y {
    default+{ =0 }
    // ...
};

Y y; // error: Y is abstract

class YY : public Y {
    default -{ =0 } // like Y but not pure virtual
    // ...
};

YY yy; // ok, maybe
```

The **-{ =0 }** means that new functions declared in **YY** are not pure virtual (as they would have been had there been no change of the default for **=0** in **YY**) and the functions **YY** inherits from **Y** and overrides are no longer pure. That is all is back to the C++98 default behavior.

Merging the default behavior specifiers from two base classes is a simple union.

A default behavior specifier in a derived class does not affect its bases.

6 Should we change some defaults?

Since 1984 or so, people have suggested that copying should be prohibited under certain circumstances (e.g. for classes with pointers or classes with destructors). Similarly, people have suggested that a class with a virtual function should automatically have a virtual destructor. I am sympathetic, but note that these are separate issues.

The right thing to do would be for the presence of a destructor to imply that default copy is suppressed (without any other effects). This would undoubtedly break some existing code, forcing people who (typically ill advised) to add copy operations explicitly. This is a separate issue – not part of this proposal – but I’d like to see it pursued.

Suppressing copy if a pointer is present would cause problems with some PODs. So, we can’t proceed with this.

Making a destructor implicitly virtual if a class has a virtual function is attractive to avoid a common, but well-known error. However, it implies “magic” addition of functionality and overhead and is rumored to break a lot of COM code. If the rumor is true, we can’t proceed with this.

7 Should we add to the set of defaults?

Given the mechanism for removing all defaults and then selectively adding the ones we want, we can consider what else we might want to add. People have asked for a feature like Java or C# interfaces. We could provide something like:

```
class Z {  
    default +{ virtual } // make every function virtual  
    // ...  
};
```

Any proposal in this direction would be independent of the main proposal for a mechanism for specifying control of default behaviors. However, the proposed default behavior specifications provides a general mechanism for adding class-wide semantic mechanisms. We should consider using it to address common requests:

- Final (for functions and classes)
- Abstract (function and class)
- Generate comparison operators
- Generate arithmetic operators
- Ban global variables or all objects on the stack
- Make all constructors explicit

These are the common requests relating to default behavior that immediately springs to mind. What others might there be?

The summary of my suggestions is to provide

:	Derivation (so its absence is what is referred to as final in Java)
=	Copy (by constructor and by assignment)
()	Default construction
new	Free store allocation
static	Static allocation
virtual	All member functions virtual
=0	All member functions pure virtual
explicit	All constructors and conversion operators explicit
==	Equality and inequality
default	All the usual defaults

1.3 *Final*

This request frequently occurs in two forms (corresponding to two of the uses in Java): “prevent further derivation from this class” and “prevent overriding of this function”. The default behavior specifications easily address the first, using **:** to indicate the current default behavior: “derivation from this class is allowed”. I propose that. For example:

```
class X {
    default -{ : } // usual defaults except derivation
    // ...
};

class Y : public X { // error: derivation from X is prohibited
    // ...
};
```

The default behavior specifications cannot address the second (“prevent overriding of this function”) because it is a concern of an individual function rather than a class. I think Java overloaded a keyword here.

1.4 *Abstract*

People ask for several related things, using words like “abstract”, “interface”, and “pure”:

- Prevent creation of objects from this class
- Make all member functions virtual
- Make all member functions pure virtual
- Make a function virtual
- Make a function pure virtual
- Restrict the members of a class to virtual functions and constants

The default behavior specifications cannot address the issues related to individual functions. Apart from that, it’s a question of what we want. Currently the general notion of an “abstract class” is controlled through the low-level mechanism of pure virtual

functions or alternatively through access controls on constructors. The question is whether we want to provide direct support for a kind of class which we could call “abstract” or “interface”.

I suggest we provide the two behaviors that are simple to specify, simple to implement, and are high on the list of requests:

virtual	all member functions virtual
=0	all member functions pure virtual

In this context, constructors are not considered member functions. I do not propose a mechanism for specifying a non-**virtual** function in a class where **virtual** has been defined to be the default (e.g., **!virtual**). I don’t see a compelling need.

The other meanings of “abstract” and “interface” seem too varied, intrusive, and tricky (at least to be handled as part of this proposal).

Note that I propose to have { **virtual** } make all (non-constructor) member functions virtual. That’s the simplest rule. We could make exceptions, such as copying and ==. However, for the kind of class where all operations are virtual, == is usually a horrid problem and so are the default copy constructors, so I don’t feel like complicating rules to make the semantic mess less obvious.

1.5 *Generate comparison operators*

The most frequent request is to generate == with the meaning “all members are equal”. The second most frequent (just by my reckoning, of course) is != with the meaning “not == if == is defined otherwise any member not equal”. I propose that == should provide a consistent pair of == and !=. For example:

```
class X {
    default+{ == }
    M1 m1;
    M2 m2;
};
```

This would generate

```
X operator==(const X& a, const X& b)
    { return a.m1==b.m1 && a.m2==b.m2; }

X operator!=(const X& a, const X& b)
    { return !(a==b); }
```

Would these functions be inline? I’d say yes, but this is a place where compiler writers could use the discession given to them by inline.

What if the user defined ==, but not !=? Should we be able to request that != be generated? That would be easy to support, but it is also trivial to define **operator!=()** so I don't propose that.

The next most frequent request is to generate <, <=, >, and >= from their member versions. For example:

```
class X {
    default+{ < }
    M1 m1;
    M2 m2;
};
```

Could mean “generate an **operator<** for **X**”. Unfortunately, we have more than one way to do that. We also face a problem with combinations of operators. If we have < and ==, should we generate <=? What if we generate < but there is a declaration for <=? Now < and <= may be inconsistent. My inclination is to treat <, <=, >, and >= as a group when it comes to defaults (as we treat == and != as a group and also the copy operations), but I don't want to propose anything without further work.

An alternative way of providing this functionality is through **concept_maps**, but that works only for templates.

1.6 *Generate arithmetic operators*

Given += and =, we can generate +:

```
X operator+(const X& a, const X& b) { X r = a; return r+=b; }
```

Given + and =, we can generate +=:

```
X operator+=( const X& b) { return *this=*this+b; }
```

Should we support that? I don't know, but if we did, it would look like this:

```
class X {
    default +{ += }
    X operator+(const X& a, const X& b);
    // ...
};
```

And

```
class X {
    default +{ + }
    X operator+=( const X& b);
    // ...
```

```
};
```

How would we decide if this extension was worth while? Until I have a better answer to that question, I'm not proposing this.

1.7 *Ban global variables or all objects on the stack*

We can allocate an object

- On the free store (with **new**)
- On the stack (automatically, **auto**)
- In static storage (**static**)

In addition, one could consider member definitions a separate category. What would we like to control here? My impression is that the most common wishes are for:

- Put all objects on the free store
- Ban global variables

This could be approximated by

```
class X {
    default-{auto static} // must use new
    // ...
};

class Y {
    default-{static}      // no static allocation
    // ...
};
```

In both cases, the intent isn't expressed directly:

- We wanted to say “use **new**” but said “don't use **auto** or **static**” (so we can still use **X** as a member).
- We wanted to say “no globals” but said “no static allocation” (so we cannot use **Y** in a namespace).

The problem with **new** is that C++'s defaults are permissive, so we have to subtract from them or disallow “everything” and then specify what we really want:

```
class X {
    default { new } // must use new
                // (because stack and static allocation are not allowed)
    // ...
};
```

But here, we “lost” all the other defaults. That may not be as bad as it sounds. For example, the desire for free store allocation often comes from a desire to control memory for a strongly object-oriented program. In that case, the default copy is probably wrong anyway. To guarantee that a **X*** points to free store, we would also have to ban **Xs** from being members. A more realistic example might be

```

class X {
    default { new : virtual () explicit } // must use new
    // ...
};

```

This would be what people who prefer to be explicit would write anyway. Doing it this way wouldn't require us to use **auto** for stack allocation, just as we are teaching people to use it to mean "take the type from the initializer". So I don't propose an **auto** default specifier just as I don't propose a "member" default specifier.

It seems reasonable to allow control of static allocation rather than explicitly deal with global variables. Often, it is static allocation people want to avoid (often because of potential initialization problems) and many forget about namespaces.

1.8 *Make constructors explicit*

We could define a default to make all constructors and conversion operators explicit:

```
default+{ explicit }
```

That would be a minor benefit because the alternative would be to sprinkle **explicit**s lightly over those functions. However, doing it this way would help consistency and prevent a few silly mistakes.

8 Other operations

There are more "operations" on class objects and classes that have default behavior:

&	Address of (only explicit use)
,	Comma (not the comma in argument lists)
~	Default destruction
->~	explicit destruction (for a pointer)
delete	Free store deallocation
auto	stack allocation
member	declaring a member
[]	declare arrays
cast	casting (there are six alternative forms of casts)
.	
->	for a pointer
.*	
->*	for a pointer
::	
?:	
sizeof	
typeid	

I don't propose to add these to the list of behaviors we can control with default behavior specifications. The arguments for including them are "completeness" and "someone might want to". The arguments against are "simplicity", "minimalism", and "show me a use case".

I thought that controlling address-of, comma, destruction, **delete**, and/or **auto** might be good ideas. I changed my mind because of lack of really convincing use cases. A good use case could push me back to my original position.

There is a real benefit in keeping the list of controlled defaults short. That eases the work of people who (for a variety of reasons) prefer to provide a complete list of defaults rather than working relative to the default defaults.

Banning explicit use of **&** (address of) seems a good idea until you think of pass-by-reference where a free store allocated objects can appear as a reference, so that e need **&** to gets a pointer.

9 Global defaults

Should it be possible set the defaults for all classes, for all classes in a namespace, for all classes in a source file, etc.?

I think not, and I don't propose that. The reason I don't think so is that I don't see how something sufficiently expressive could be provided without a major increase in complexity.

10 Detailed description of defaults

???

11 Grammar

???

12 Acknowledgements

Obviously, much of this design came from earlier papers and discussions, especially in the evolution working group. The main papers are listed in Section 1. Special thanks to Francis Glassborow, Lois Goldthwaite, Michel Michaud, Jack Reeves, Gabriel Dos Reis, and Michael Wong.