Doc No: SC22/WG21/N1466
J16/03-0049
Date: 01-Apr-2003
Project: JTC1.22.32
Reply to: Daniel Gutson
danielgutson@hotmail.com

EXPLICITING DEFAULT PARAMETERS

1. THE PROBLEM

C++ has a very simple mechanism for defaulting parameters. However, they are required to be provided at the end of the signature declaration. Additionally, there is no way to use their value in the function call when the next parameter value is specified.
There is also a problem of ambiguity when overloading functions when their signture differs only in the "defaulted" parameters.
For graphic purposes, the following notational examples illustrate such sub problems:

Subproblem # 1:
declaration: function (<non default>, <default>, <non default>) **not possible**.

Subproblem # 2:
declaration: function'(<default>, <default>);

invocation: function'(<default>, <value>); **not possible**

Subproblem # 3
declaration: f (<non-default>, <default1>);
declaration: f(<non-default>, <default2>);

invocation: f (<non-default>); **error:ambiguity.**

- Why is the problem important?

- Default values are part of the interface, allowing the user to be abstracted from the value itself. The situations (subproblems) mentioned above force the user to -somehow- know (and mess) with the value, breaking such "abstraction" (in fact, subproblem #1 forbids the interface to use default values in that way).

- the simplicity (and usability) of default parameters is discontinued in th situations mentioned above, requiring the use of additional code, maintenance overhead, and/or user involvement. (see the impact analysis of workarounds later in this document).

- Default parameters also contribute to the self documentation of the interface. This aspect is also

discontinued in the mentioned situations.

- Changing the value of a default parameter is straight forward, visible (for the user) and punctually (locally), not affecting the rest (user's) code when the user "agrees" to use the functions providers' default value.
On the other hand, without using the prototype for default parameters, the change has to be propagated.

- Common workarounds:

A) Forwarding functions
B) "Named" parameters using structures
C) Constants

The following analysis will be performed for each workaround:

- Implementation cost:
    -from the library interface: Code overhead, additional required elements - structures, functions,constants-, documentation

    -from the user: additional code and knowledge, loose of abstraction

- Change cost (maintenance):
    -of a default value
    -of signature

In order to explain the different workarounds, let's name"f" a function that has "m" (non def.) parameters and "n" default parameters. Do note thar the numeric analysys is based in ONE function. The results should be considered when –as often- an API contains more than one function.

A) Forwarding functions:
A technique based on making "f" with all its parameters non default (n+m non default parameters) and adding "helper" functions overloading F, combining different signatures (with non default parameters). In their body, the original "f" is called, passing the received parameters plus constants for those not received. Therefore, the "defaulted" values are in the implementation of the forwarding functions.
Forwarding functions cover all the possible calling combinations of "f" as far as the type of the parameters are different.
If n' is the number of elements of a subset of the n parameters, such as all the parameters of this subset are of different type ($n' \leq n$), then

$2^{n'}$ forwarding functions are required if $n'<n$, or

$2^n - 1$ forwarding functions, if $n' = n$ (that is , all parameters are of different types).

Example: n= 3, m=0
F: f(type1, type2, type3)

The $2^3 - 1 = 7$ forwarding functions prototypes are:

f();
f(type 1); f(type 2) ; f(type 3);
f(type1, type2); f(type1, type3); f(type2, type3).

Assuming Allman's style, 4 lines per functions are required:
The header +the opening brace+the (original) "f" call+ the closing brace,
which results in

additional_code= $4 * (2^n - 1)$ LOC, without blank lines.

Assuming "K&R" style, 3 lines per forwarding function are required (as far as the opening brace is placed in the same line of the header), resulting in $3*(2^n - 1)$ additional LOC, without blank lines.

Readability evaluation:

- differentiation relies on type and arity.
- The default values are "hidden" in the fwd. function' s bodies, and repeated for those "defaulting" the same parameter.
  Each value is repeated 2^{n-1}-1 times.
  In order to avoid the impact of updating such number of times when the value changes, it is also common to centralize these value instances in constants, arriving to the situation of work around C). In this case, one constant for each def. parameter is provided, resulting in n additional lines; therefore

  $4 * (2^n - 1) + n$ additional lines for Allman's style,
  and
  $3 * (2^n - 1) + n$ for K&R style

  Considering that the initial intention was a function capable to receive "default" parameters, the header file would be overheaded with $2^n - 1+n$ LOC per function, or $3 * (2^n - 1) + n$ / $4 * (2^n - 1) + n$ LOC if the fwd function are inlined (which is the most common situation). This additional code is an impact to the readability of the API.

Maintenance Analysis:
Modification of the signature of "f"

- If a type of one of the n parameters is modified, $2^n - 1$ updates are required in the fwd. functions, plus the constant declaration, resulting in
2^{n-1} updates (API only).

- If a parameter is added or removed, all the existing fwd. functions have to be updated (as far as they call "f" in their body).

If a parameter is added (let's keep n with the old value)

$2^n - 1$ fwd. functions are updated,

and

2^n fwd. functions are added ($(2^{n+1} - 1)$ - $(2^n - 1)$),

resulting in:

$2^n - 1$ LOCs modified (their bodies)

plus

$4*(2^n)+1$ LOCs added (Allman's, inlined, plus the new constant)

If the added parameter is non-default (increases m), the existent fwd. functions are modified:

$2^n - 1$ LOCs modified (their bodies).

Finally, this technique does not solve sub problem # 3.

B) "Named Parameters" through structures:

This technique requires a structure per signature.

The idea is to pack all the parameters in the structure, assign their values using projectors (that return to self reference), and pass the reference of the (temporary) structure to the function.
This technique implies:
- one structure per desired (logical) signature
- the real signature (or the types that the function receives) are inside the structure.
- one projector + one attribute for each logical parameter
- the default values are placed (hidden) in the implementation of the constructor.
- no static checking that all the required parameters were provided. (another version of this technique is to keep all required parameters as physical parameters in the signature, and pack only the 'defaulted' parameters in the structure).

This technique is commonly used for "large" number of logical parameter. Let's call 'n' this number.

The following analysis is a simplified version that does snot perform the run time checking in the signature, and pack only the "defaultable" parameters in the structure.

For Allman's style, and inlining the methods, the length of the structure is

```
1 x header
1 x {
1 x         constructor:
n x                 default values initialization
1 x         { }

          ⎧ 1 x   parameter projector header
          ⎪ 1 x   {
5 x n     ⎨ 1 x     assignment
          ⎪ 1 x     return * this
          ⎩
```

1 x }

n x attribute for storing parameters' value

1 x };

_____

Total= 5 + 7n LOC (without blank lines)

For K&R style

1 x Header {
1 x constructor:
n x def. Values initialization

1 x { }

4x n {
1x parameter project header {
1x assignment
1x return *this
1x }

n x attributes
1 x };

____

Total = 4 + 6n LOCs

Additionally, this technique impacts in readability of the API (as for as the signature and default values dwell inside the structure), and might decrease the coding productivity as far as the parameter's names have to be explicitly keytyped (coded), specially for programmers familiarized with the signatures (mnemonic).
Example:

f(structure().parameter1(value1).parameter2.(value2));

If the function receives 3 parameters, and the last was intended to preserve its default value, the previous line would be written

f(value1, value 2);

which is clearly faster to type, and less messy to read (specially for the programmer familiarized with the system).

Finally, this technique represents additional complexity for function overloading (which must be accomplished with another structure for packing parameters) which also represents an issue for sub problem # 3.

Changes in the logical signature impact in the structure:

- in the constructor
- in the projectors
- in the attributes

C) Using constants:

This technique requires constant definition for each parameter to be "defaulted" of each function.
It is recommended to have a naming convention (extra documentation) for the constants, combining the name of the function each constant is intended to value-default, and the parameter name.
This makes the coding more error prone due to typo errors of misunderstandings, as far as the user is responsible for explicitly provide the proper constant (representing the default value). (Specially when many constants are of the same type)

At this point, this proposal exposed some of the workarounds with their implicancies , their impact to the interfaces and users from a user-API point of view, as well as some maintenance costs, coding and readability, and how they deal with the 3 exposed subproblems.
This proposal **does not intend to invalidate the preceding techniques**, but proposes an altenative preserving the simplicity and 'spirit' of the current default-parameters mechanism.

2. THE PROPOSAL

- Allow default parameters before non-default ones in signatures
- Explicit the default value usage through the 'default' keyword as parameter in function invocation statement- optionally followed by <type> for avoiding ambiguities on overloaded functions-.

  default type_specifier$_{opt}$
  type_specifier: <type>

2.1. Basic cases

```
void  f(int x = 1, int y=2);

void use()
{
        f (default, 3);
        f( );
}
```

2.2. Advanced cases

- Specifying the type
- Non POD default parameters

- templates

```
template <class T> T* allocAndClone (
            AllocatorType1 <T> alloc = MyAllocator <T>::instance(),
            T& prototype = T() , size_t size);

template <class T> T* allocAndClone (
            AllocatorType2 <T> alloc = MyAllocator <T>::instance(),
            T& prototype = T() , size_t size);

void process(char t = '\n');
void process(const char* = "end of line");

void use(void)
{
      MyClonableClass* p;

      p = allocAndClone(
                  default<AllocatorType1<MyClonableClass> >,
                  default,
                  10);

      process(default<char>);
      process(default);             // error: ambiguity
      process(default<const char*>);
      process(default<float>);      //error
}
```

3. INTERACTIONS AND IMPLEMENTABILITY

3.1. Interactions

- The type of specification is optional when the defaulted parameters type is unique in the contex of the overloaded functions;
- When present, the specified type can be also a template with its proper template parameter (as shown in advanced cases)

3.2. Implementability

- The proposed syntax does not violate compatibility due to its current incorrectness
- An implementation consideration should be taken in the case of an invocation in the context of a template, which can specify the template parameter as type specification, as far as information on default parameter values should be also available at template instantiation phase.

Example:

```
template <class T> void doCall()
{
      f(default<T>);
}
```