Doc No: SC22/WG21/N1465
J16/03-0048
Date: 01-Apr-2003
Project: JTC1.22.32
Reply to: Daniel Gutson
danielgutson@hotmail.com

CONSTANT INHERITANCE

1. The problem

Differentiating the interface of a class in "const" and "non-const" methods, there is no way of inheriting only from the "const" part, that is, an "is a [read only]…" relationship. However, it is possible in other relationships such as containment and aggregation (as a "const member" and "const ref/ptr to" respectively).

- Why is the problem important?

Inheriting publicly from a base class "appends" (publish) all its attributes and methods, making them accessible to third-users (according to visibility attributes), including const and non-const methods.
If a class "wants" (from design) to provide the read-only part of another, and treat its attributes as read-only too, it has to contain it and provide projectors and wrappers for each attribute and methods (respectively) controlling and adding const modifiers. If, -as often- the base class changes and evolves, the derived class has to be updated representing a maintenance overhead.

Suppose class Base has "m" const methods, and "a" attributes. If class Der needs to mimic the "is a read-only Base" relationship through const containment, Der will have to have "m" forwarding methods and "a" getters. Additionally, upcasting is not automatic unless Der implements a "const Base&" casting operator, and optionally, a & operator returning a "const Base*" pointer.

This feature fits in the following subset of categories:

* improve support for systems programming: eliminates coding and maintenance overheads, and allows inheritance in a situation not currently available.
* improve support for library building: this feature is specially useful for access control of third-party users (the base class, the subclass, and a user class), as well as self-documentation of design-intentions, which could be eclipsed by workarounds.
* remove embarrassments: there's no need of additional code and special mechanisms but inheritance and upcasting.

2. The proposal

Enable the "const" keyword in the base class list.

2.1. Basic Example

```
class DbAccess
{
public:
      //modifying/editing section
      void removeClient(size_t ID);
      virtual void  removeAll();
      size_t addClient(const char* name);
      void updateClient(size_t ID, const char* name);

      // Querying section:
      virtual size_t getMemoryUsage(void) const;
      size_t getClientsCount( ) const;
      const char* getClientName(size_t ID) const;
      const_ClientIterator  getClientsIterator () const;
};

class SuperQuerier: public const DbAccess
{
public:
      size_t  countClientsThruWildcard (
                        const char* wildcard) const;

      const_ClientsIterator  getClientsThruWildcard(
                        const char* wildcard) const;

      size_t registerSelectQuery(SelectQuery * qry);
      const_ClientsIterator runSelectQuery(size_t qryID);
      void deRegisterAllQueries(void);
      void deRegisterSelectQuery(size_t qryID);
      virtual void removeAll(void);
      virtual size_t getMemoryUsage(void) const;
private:
      size_t _myUsedMemory;
};
```

2.2 Advanced Cases

default operator = becomes unavailable when inheriting a const class:

```
SuperQuerier  s1,s2;
s1.registerSelectQuery(q1);
s1.registerSelectQuery(q1);

// copy all registered queries to S2
s2 = s1;  //error: operator = not available (same as
containing const attributes)
f(DbAccess&);
f(s1);      //error: receives a non-const, violating
upcasting

g(const DbAccess* p)
{
```

```
        SuperQuerier* s = dynamic_cast< SuperQuerier* > (p);
}

void SuperQuerier: deRegisterAllQueries (void)
{
        removeAll ( ); /*error: not accessible  since
                            DbAccess::removeAll is non-const*/
}

void SuperQuerier::removeAll(void)
{
        DbAccess::removeAll ( ); //error: same as above
        DeRegisterAllQueries(); //ok (if above is corrected)
}

size_t SuperQuerier: getMemoryUsage() const
{
        return DbAccess::getMemoryUsage() + _myUsedMemory;
                                                    //ok
}
```

3. Interactions and Implementability

3.1. Interactions

- the derived class becomes “const” when upcasted to a const-inherited base class
- methods of the derived class as well as non-member methods can only invoke const methods of the (const-inherited) base class.
- attributes of the (const-inh.) base class are constant (read-only) for the derived class, and other derived class’ users (external or third-level sub classes).
- virtual non-const methods of the derived class can override virtual non-const of the (const-inh) base class, but cannot invoke them (supermessage)
- The above mentioned interactions also apply to pointers to members
- dynamic_cast from a const upcasted derived class pointer to the derived class would act also as a const_cast (returning non-const pointer to Der)

3.2. Implementability

- backguard compatibility is not impacted as far as the proposed syntax is currently invalid
- this feature enhances static checking of the compilation phase, basically in the following situations:
    - upcasting checkings
    - dynamic casting
    - attributes access checkings
    - const methods invoking checkings
    - assignments

4. Acknowledgements

Thanks to Philippe Mori for his vital feedback.