# Continuity Methods

a proposal for the C++ programming language by T. P. K. Healy

**This proposal is to augment the functionality of classes in C++ to allow a new kind of member function (or '*method*') that automatically invokes all matching methods belonging to the base classes.**

**Minimal code sample to demonstrate the problem:**

```
#include <iostream>
using std::cout;
using std::endl;

class Laser {
public:
    virtual void Trigger(void)
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) override
    {
        this->Laser::Trigger();
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};

class Laser_PicoSecond : virtual public Laser {
public:
    void Trigger(void) override
    {
        this->Laser::Trigger();
        cout << "Send trigger command to the PicoSecond laser over RS232" << endl;
    }
};

class Laser_NitrogenPicoSecond : public Laser_Nitrogen, public Laser_PicoSecond {
public:
    void Trigger(void) override
    {
        this->Laser_Nitrogen::Trigger();
        this->Laser_PicoSecond::Trigger();
    }
};

int main(void)
{
    Laser_NitrogenPicoSecond object;

    object.Trigger();
}
```

**This program prints the following output:**

```
Set pin high to switch transistor to allow laser current
Set TTL pin high to activate nitrogen laser
Set pin high to switch transistor to allow laser current
Send trigger command to the PicoSecond laser over RS232
```

**The problem here is that the method ‘`void Laser::Trigger(void)`’ is invoked twice on the same ‘`class Laser`’ object. This code could be improvised by complicating one of the derived classes, for example ‘`class Laser_PicoSecond`’ could be modified as follows:**

```
class Laser_PicoSecond : virtual public Laser {
protected:
    bool invoke_laser_trigger;

public:

    Laser_PicoSecond(bool const arg = true) : invoke_laser_trigger(arg) {}

    void Trigger(void) override
    {
        if ( invoke_laser_trigger ) this->Laser::Trigger();

        cout << "Send trigger command to the PicoSecond laser over RS232" << endl;
    }
};

class Laser_NitrogenPicoSecond : public Laser_Nitrogen, public Laser_PicoSecond {
public:
    Laser_NitrogenPicoSecond(void) : Laser_PicoSecond(false) {}

    void Trigger(void) override
    {
        this->Laser_Nitrogen::Trigger();
        this->Laser_PicoSecond::Trigger();
    }
};
```

**Ater making this code change, the program now successfully prints:**

```
Set pin high to switch transistor to allow laser current
Set TTL pin high to activate nitrogen laser
Send trigger command to the PicoSecond laser over RS232
```

**This code change would be fine in a small project with only one or two programmers, but it would become bewildering if the project were to expand to have more base classes, more derived classes, and more engineers.**

**My proposed addition to the C++ language is to mark the method in the derived class with the keyword '`continue`' as follows:**

```
class Laser {
public:
    void Trigger(void)
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) continue
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};

class Laser_PicoSecond : virtual public Laser {
public:
    void Trigger(void) continue
    {
        cout << "Send trigger command to the PicoSecond laser over RS232" << endl;
    }
};

class Laser_NitrogenPicoSecond : public Laser_Nitrogen, public Laser_PicoSecond {
public:
    void Trigger(void) continue
    {
        /* nothing to do here */
    }
};
```

**When an object of type '`class Laser_NitrogenPicoSecond`' is created and the '`Trigger(void)`' method is invoked on it, the compiler knows to invoke the following methods in the correct order, without making the mistake of invoking '`Laser::Trigger(void)`' twice on the same object:**

```
    Laser::Trigger              (Set pin high to switch transistor to allow laser current)
    Laser_Nitrogen::Trigger     (Set TTL pin high to activate nitrogen laser)
    Laser_PicoSecond::Trigger   (Send trigger command to the PicoSecond laser over RS232)
```

**Furthermore I propose that the base class can also be marked in order to indicate that the base class's implementation of the method contains essential processing -- essential processing that any derived class must invoke. The method in the base class can be marked with '`requires continue`' as follows:**

```
class Laser {
public:
    void Trigger(void) requires continue
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};
```

**If the method in the base class is marked with '`requires continue`', then the method in the derived class must be marked with '`continue`', otherwise the compiler shall terminate compilation and issue an error. The following code is erroneous:**

```
class Laser {
public:
    void Trigger(void) requires continue
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void)      ← This won't compile
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};
```

**The compiler shall issue the following error:**

```
main.cpp:3:20: error: method in base class 'Laser' marked 'requires continue', but
                      method in derived class 'Laser_Nitrogen' not marked 'continue'
    3 |     void Trigger(void) requires continue
      |                        ^~~~~~~~~~~~~~~
```

**In order to avoid this compiler error and to allow the derived class to ignore the base class's requirement, the method in the derived class can be marked with '`break`' as follows. The following code snippet is well-formed and its behaviour is well-defined:**

```
class Laser {
public:
    void Trigger(void) requires continue
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) break
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};
```

**However if the method in the derived class is marked with '`break`', and the method in the base class is not marked with '`requires continue`", then the compiler shall terminate compilation and issue an error:**

```
main.cpp:11:20: error: method in derived class 'Laser_Nitrogen' marked 'break', but
                       method in base class 'Laser' not marked 'requires continue'
   11 |     void Trigger(void) break
      |                        ^~~~~
```

**For added assurance, the programmer can mark the method in the derived class with '`continue(explicit)`' to indicate that the method in the base class must be marked with '`requires continue`', as follows:**

```
class Laser {
public:
    void Trigger(void)
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) continue(explicit)      ← This won't compile
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};
```

**The compiler shall issue the following error:**

```
main.cpp:3:20: error: method in derived class 'Laser_Nitrogen' marked 'continue(explicit)',
                      but method in base class 'Laser' not marked 'requires continue'
   11 |     void Trigger(void) continue(explicit)
      |                        ^~~~~~~~~~~~~~~~
```

**Furthermore the programmer can mark the method in the base class with "`requires break`" in order to prevent the method in the derived class from being marked as continue, as follows:**

```
class Laser {
public:
    void Trigger(void) requires break
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) continue        ← This won't compile
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};
```

**The compiler shall issue the following error:**

```
main.cpp:3:20: error: method in base class 'Laser' marked 'requires break', but
                      method in derived class 'Laser_Nitrogen' marked 'continue'
   11 |     void Trigger(void) continue
      |                        ^~~~~~~~
```

**This compiler error can be prevented by declaring the method in the derived class as follows:**

```
void Trigger(void) continue(!break)
```

**The code sample given so far in this draft proposal shows a scenario in which the method in the base class must be invoked *before* the remaining code of the method in the derived class is executed. The programmer can control the order of the invocation of methods by using the statement '`goto continue`' as follows:**

```
class Laser {
public:
    void Trigger(void) requires continue
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
publi
    void Trigger(void) continue
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;

        goto continue;
    }
};

int main(void)
{
    Laser_Nitrogen object;
    object.Trigger();
}
```

**The above program shall print:**

```
Set TTL pin high to activate nitrogen laser
Set pin high to switch transistor to allow laser current
```

**Furthermore, the arguments passed to the method in the base class can be customised by using parentheses to invoke "`goto continue`" with function call syntax:**

```
class Laser_Nitrogen : virtual public Laser {
publi
    void Set_Intensity(int const arg) continue
    {
        cout << "Nitrogen Laser Intensity = " << arg << endl;

        goto continue(arg + 3);
    }
};
```

**So the following code snippet:**

```
int main(void)
{
    Laser_Nitrogen object;
    object.Set_Intensity(5);
}
```

**shall print out:**

```
Nitrogen Laser Intensity = 5
Laser Intensity = 8
```

**The chain of continuity methods can be interchangeably `virtual`, non-`virtual`, `override`, non-`override`, `final`, non-`final`. The following program is well-formed and its behaviour is well-defined:**

```
class Base1 {
public:
    virtual void Trigger(void) requires continue
    {
        cout << "Base1" << endl;
    }
};

class Base2 {
public:
    void Trigger(void)
    {
        cout << "Base2" << endl;
    }
};

class Derived : public Base1, virtual public Base2 {
public:
    void Trigger(void) override continue
    {
        cout << "Derived" << endl;
    }
};

class FurtherDerived : public Derived, virtual public Base2 {
public:
    void Trigger(void) continue
    {
        cout << "FurtherDerived" << endl;
        goto continue;
    }
};

int main(void)
{
    FurtherDerived object;
    object.Trigger();
}
```

**The output of this program is:**

```
FurtherDerived
Base1
Base2
Derived
```

**Note here that the method ‘`Base2::Trigger(void)`’ is only invoked once.**

**If the derived class has several base classes, for example:**

```
class Derived : public Base1, public Base2, public Base3, public Base4 {
public:

    void Func(void) continue
    {
        /* nothing to do in here */
    }
};
```

**The method in the derived class shall call the methods in the base classes in the correct order, as though it were written:**

```
class Derived : public Base1, public Base2, public Base3, public Base4 {
public:

    void Func(void)
    {
        this->Base1::Func();
        this->Base2::Func();
        this->Base3::Func();
        this->Base4::Func();
    }
};
```

**However specific base classes can be explicitly excluded by name. For example the following code:**

```
class Derived : public Base1, public Base2, public Base3, public Base4 {
public:

    void Func(void) continue(!Base2,!Base4)
    {
        /* nothing to do */
    }
};
```

**behaves as though it were written:**

```
class Derived : public Base1, public Base2, public Base3, public Base4 {
public:

    void Func(void)
    {
        this->Base1::Func();
        //this->Base2::Func();   ⟵ This one isn't invoked
        this->Base3::Func();
        //this->Base4::Func();   ⟵ This one isn't invoked
    }
};
```

**If the method in the base class cannot be called because it is '`private`' then it is just simply not called -- this is not a compiler error (nor is it a warning -- there is no need to issue a diagnostic). Only '`public`' or '`protected`' methods shall be invoked.**

**An abstract base class can provide an implementation of a pure virtual method if it is marked as '`requires continue`', for example:**

```
class Laser {
public:
    virtual void Trigger(void) requires continue = 0        ← Pure virtual
    {
        cout << "Set pin high to switch transistor to allow laser current" << endl;
    }
};

class Laser_Nitrogen : virtual public Laser {
public:
    void Trigger(void) override continue
    {
        cout << "Set TTL pin high to activate nitrogen laser" << endl;
    }
};
```

**The type '`class Laser`' still remains abstract and you cannot create an object of it:**

```
int main(void)
{
    Laser object;      ← This won't compile because 'Trigger(void)' is pure virtual
}
```

**For compilers that implement virtual methods by means of a V-table, the entry for '`Trigger(void)`' in the base class's V-table shall not be a `nullptr`. The implementation of '`Laser_Nitrogen::Trigger(void)`' can invoke '`Laser::Trigger(void)`' by getting its address from the base class's V-table.**

**To give a thorough and extreme example:**

```
struct Base1A {
	virtual void Set_Int(int const arg) requires continue
	{
		cout << "Base1A value = " << arg << endl;
	}
};

struct Base1B {
	void Set_Int(int const arg)
	{
		cout << "Base1B value = " << arg << endl;
	}
};

struct Derived1A : virtual Base1A {
	void Set_Int(int const arg) override break
	{
		cout << "Derived1A value = " << arg << endl;
	}
};

struct Derived1B : virtual Base1A, Base1B {
	void Set_Int(int const arg) continue(!Base1B)
	{
		cout << "Derived1B value = " << arg << endl;

		goto continue(arg / 3);
	}
};

struct Base2 {
	void Set_Int(int const arg)
	{
		cout << "Base2 value = " << arg << endl;
	}
};

struct Derived2 : virtual Derived1A, virtual Derived1B, virtual Base2 {
	void Set_Int(int const arg) override continue
	{
		cout << "Derived2 value = " << arg << endl;
	}
};

struct FurtherDerived : virtual Base2, Derived2  {
	void Set_Int(int arg) continue
	{
		cout << "FurtherDerived starts, value = " << arg << endl;

		for ( unsigned i = 0; i < (arg / 4); ++i )
			goto continue(++arg);

		cout << "FurtherDerived ends" << endl;
	}
};

int main(void)
{
	FurtherDerived object;
	object.Set_Int(7);
}
```

**This program shall print:**

```
FurtherDerived starts, value = 7
Base2 value = 8
Derived1A value = 8
Derived1B value = 8
Base1A value = 2
Derived2 value = 8
Base2 value = 9
Derived1A value = 9
Derived1B value = 9
Base1A value = 3
Derived2 value = 9
FurtherDerived ends
```

---

## Change History

**Draft No. 2: Total length of document extended from 9 pages to 11 pages**

1) **New feature:** `continue(explicit)`
2) **New feature:** `requires break`
3) **New feature:** `continue (!break)`
4) **New feature: implementation of pure** `virtual` **methods marked '**`requires continue`**'**

---

**The idea for this proposal was first shared to the mailing list for proposals for the C++ Standard:**

**std-proposals@lists.isocpp.org**

**on 6 April 2022 under the title "*Derived class's function invokes base class's function*". The original archived post can be viewed at:**

**https://lists.isocpp.org/std-proposals/2022/04/3765.php**

**Please share any feedback by subscribing to the mailing list:**

**https://lists.isocpp.org/mailman/listinfo.cgi/std-proposals**