C++0X: IMPACT ON THE PROGRAMMING PRACTICE

Miroslav Virius
České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská
miroslav.virius@fjfi.cvut.cz

ABSTRACT:

Selected features introduced by the expected new C++ programming language standard are presented in this article and their impact on the common programming practice is briefly discussed. Short examples of selected features are given.

KEYWORDS:

C++0x, object oriented programming, declaration, list initialization, lambda expression, variadic template, constructor,

1. INTRODUCTION

This article discusses some of the expected changes and their impact on the common programming practice. We shall refer to the C++ language conforming to the valid standard [1] as to the C++03; the expected new standard is referred to as C++0x. This article is based on the latest draft [2].

First, we briefly recall the history of the C++ programming language here. In the following sections, we present the new features concerning the object oriented programming in C++ 0x, the new form of initialization and function definition. The next section contains very terse summary of the other news in the C++ programming language.

2. BRIEF HISTORY OF THE C++ PROGRAMMING LANGUAGE

Since 1983, when the C++ programming language was introduced by B. Stroustrup, it developed significantly.

The first versions of this language were compiled to the C code. Thus, the corresponding C++ specifications were denoted as “C front”. The Cfront 1.0 was introduced by B. Stroustrup in 1985 on behalf of the Bell Laboratories of the AT&T Company. It was the C programming language with the classes supporting single inheritance only without virtual functions. The first commercial version, released in 1985, added the virtual functions, the references and the operator overloading, among other features.

Next major version, the Cfront 2.0 released in1989, introduced the multiple inheritance and the multiple inheritance, abstract classes, static member functions and protected members.

In 1990, *The Annotated C++ Reference Manual* [4] was published. This book acted for long tima as an unofficial C++ standard and became the basis for the future standard. The new features introduced were the templates, the namespaces, the run-time type identification and the exceptions.
The first version of the international standard ISO/IEC 14882 was published in 1998. The main feature added was the Standard Template Library. It was well known that this library was incomplete, mainly due to the standard template library design and related changes it the language syntax (viz. the templates and template metaprogramming features use in the STL). The revision of this standard, published in 2003 [1], fixed elementary problems, but din dot bring any substantial change to the core language, (i.e. to the C++ syntax as well as to the

C++ standard library). The new version of the standard, informally known as C++0x, shall appear in the end of the year 2011 and will bring some changes in the core language.

2. OBJECT ORIENTED FEATURES IN THE C++0x

The main intent of the changes is to submit the programmer some useful language features that appeared in some newer programming languages, such as Java or C#.

2.1 Class Declaration

The class declaration may contain additional specifications describing properties of the class.

2.1.1 Final Class

It was not possible to prohibit the descendant class derivation in C++03, if you did not use syntactic tricks like private constructors – and this "solution" was sometimes unusable. In C++0x, the **`final`** specifier in the class heading declares the final class, i.e. the class that cannot serve as a base class. This specifier, if used, should follow the class name, prior to the base class's specification.

2.1.2 Explicit Override of the Virtual Methods

If you declare a virtual method in the base class in C++03, you need not repeat the **`virtual`** specifier in the descendant class. Moreover, there is no syntactical difference between the declaration of a new virtual method and the override of an existing method. This may introduce some tricky errors:

- You could accidentally override a virtual method, even though you intended to introduce a new one.
- You could introduce a new virtual method, even though you intended to override existing one.

The C++0x brings a solution. If you declare the derived class with the **`explicit`** specifier, you will have to add the **`override`** specifier to any method declaration that overrides an inherited virtual method. This specifier is inserted between the function header and the function body; it follows the **`const`** method specifier, if used. The compiler will report an error in the case you forget to add this specifier to an overriding method, as well as in the case you add it to a method that does not override any virtual method in the base class.

Note that if you omit the **`explicit`** specifier, the C++0x compiler will treat the derived class in the same way as the C++03 compiler.

2.1.3 Hiding Inherited Members

If you declare a new member in the derived class that has the same name as a member of the base class, you hide it. It is still accessible using some syntactic tricks, but if you did it accidentally, you may get some tricky errors.

In C++0x, if you declare the derived class with the **`explicit`** specifier, you will have to add the **`new`** specifier to any member declaration that hides the same name in the base class. If you add the **`new`** specifier to a member that does not hide a name in the base class, or if you forget to add it, you will get a compile-time error, if the declared member hides a name.

2.1.4 Examples

A brief example follows. Given the base class,

```
class Base
{
  public:
   virtual void function();
```

```
  typedef int T;
};
```

we can derive a new class,

```
class Derived explicit: public Base
{
   public:
      // ...
};
```

If we declare a method in the **Derived** class

```
virtual void funtion() override;
```

we get an error message because of the misspelled name, because the new method does not override any method of the **Base** class. If we declare a new method

```
void T();
```

we will get an error, because this declaration hides the name **T** from the base class. If it is our intent, we have to write

```
void T() new;
```

2.2 Constructors

Even the rules for the constructors have been subject of changes. The basic limitations concerning the call of another constructor of the same class or the constructor inheritance have been alleviated.

2.2.1 Call of another constructor of the same class

There is no way to call another constructor of the same class to perform a part of the initialization in C++03. If you have a common part in more than one constructor, you have to extract it into a private method and to call it from all the constructors that use it.

C++0x allows one constructor to call another one. To call it, you have to write the constructor call in the initializer list of the calling constructor, in a similar manner as the call of the base class constructor. A brief example follows:

```
class X
{
      int x, y;
   public:
      X(): x(0), y(0) {}
      X(char* text): X() {cout << text << endl;}
};
```

Both constructors initialize the data members by zeroes; moreover, the second one prints the text.

2.2.2 Constructor inheritance

Constructors are not inherited in C++03; instead, the derived class constructors call one of the base class constructors. In C++0x, you may demand that all the constructors of the base class are inherited. This is done by the **using** declaration:

```
class Y: public X
{
    public:
       using X::X;
};
```

There is no way to inherit only one constructor or selected ones. You have to inherit all or no one.

2.2.3 Explicitly defaulted constructors

If you do not declare any constructor in a class, the compiler declares, and if necessary even creates the default constructor. If you declare at least one constructor, the compiler will not create any one. This can be changed in C++0x. You may ask the compiler to generate the default constructor using the syntax

```
class S
{
   public:
      S() = default;
};
```

The same syntax applies for some other special member functions, viz. for the copy constructor, the destructor or the copy assignment operator.

2.2.4 Member initialization

The new version it the C++ language shall allow the member initialization to be written in the member declaration. Thus, the following will be correct:

```
class Y: public X
{
   int n = 6;
    // ...
};
```

This initialization will be performed by any constructor.

2.3 Deleted member functions

One the interesting feature of the C++0x is the possibility to delete a member function. The syntax is similar to the explicit declaration of the default member function, but it uses the **`delete`** keyword. The deleted function may not be called; any attempt to do it will cause a compile-time error. This feature may be useful to forbid some special member functions that would be generated by default. E.g., the non-copyable class may be declared in C++0x as follows:

```
class CopyMeNot
{
   public:
      CopyMeNot(const CopyMeNot&) = delete;
```

```
      CopyMeNot(const CopyMeNot&&) = delete;
      CopyMeNot& operator=(const CopyMeNot&) = delete;
      CopyMeNot& operator=(const CopyMeNot&&) = delete;
};
```

This feature may be used for some special effects, e.g. to prevent the function to be called with all argument types except one:

```
class IntOnly
{
   public:
      void f(int);
      template<typename T> void f(T) = delete;
};
```

This will prevent the compiler to do any parameter conversions, because the template instantiation would be used instead – and the template is deleted.

3. INITIALIZATION

The C++03 uses different syntax for the initialization of the so called POD-types (arrays and structs in the C-style) and different syntax for the class objects and built-in types. The C++0x brings uniform initializer syntax and the way to initialize the containers with the array-like syntax.

3.1 Uniform Initialization

The new syntax for the initialization is based on the list of inicializers, i.e. on the syntax of the array initialization. The initializer is enclosed in the brackets:

```
class Test
{
   int member;
 public:
   Test(int value) : member{value} { /* ... */ }
   // ...
```

The instance if the **`Test`** class may be initialized as follows:

```
Test test{6};
```

or

```
Test test = {6};
```

The same syntax may be used for the POD-types (arrays and C-style structs), too.

3.2 Initializer List

The new container class **`std::initializer_list<T>`**, declared in the **`<initializer_list>`** header, may be initialized by the initializer list of the form **`{value_1, ... value_n}`**.

```
auto x = {1,2};
```

will be of the type **std::initializer_list<int>** and will contain the given values. Any class, which has the constructor of the std::initializer_list<T**>** type, may be initialized by the initializer list. This is the case of all the STL containers (**vector<>**, **queue<>** etc.) The braced list of coma separated initializers may appear in the function argument list, in the return statement, on the right hand side of the assignment operator, as an initializer in the **new** expression and in some other contexts. Nevertheless, it is not considered to be an expression of the **std::initializer_list<T>** type, thus it may mot appear in more complex expressions.

4. FUNCTION DECLARATIONS AND LAMBDA EXPRESSIONS

Lambda expression is an old functional concept appearing in the most common contemporary programming languages recently. In fact, it is a brief form of the function declaration that may be used as a function argument, as a member initialization etc.

4.1 New function declaration syntax

To start the function declaration with the return type is not always suitable; especially template function return type may depend on the argument types. To resolve this, a new (alternative) function declaration syntax was introduced:

- The **auto** keyword stays instead the return type.
- Identifier and parenthesized parameter list follows in usual manner.
- The return type specification stays behind the **->** arrow.
- If the declaration is the definition, the normal function body follows.

An example follows:

```
template<typename T, typename U>
auto f(T x, U y) -> decltype(x+y){return r+y;};
```

Note the **decltype()** specifier in the return type declaration. This is new to C++0x. It deduces the type of the expression in the parentheses and may be used in declarations.

4.2 Lambda expressions

Lambda expression starts with the lambda introducer, which may be of the form **[]**. Next is the parameter list in usual form. This “lambda header” may be followed by the **mutable** keyword. Next is the **->** arrow followed by the return type specification; the last part is the lambda expression body (i.e. the function body). The lambda introducer may contain the so called capture list, i.e. the list of all local variables in the enclosing scope that are used in the lambda expression body. It specifies which variable is captured by value and which one is captured by reference.
An example follows:

```
auto f = [=]() -> int { cout << "hello" << endl; };
```

The = in the lambda introducer declares that all the local variables are captured by value. The **f** variable will be of an implementation defined class type that has an overloaded function call operator. Thus, this function may be called by writing simply **f();**.

Lambda expression can be implicitly converted to pointer to function with the same signature. The **`mutable`** keyword, if used, forbids the changes of the local copies of the local variables captured by value.

5. SHORT REVIEW OF SOME OTHER NEWC++0x FEATURES

We discussed selected news in detail in the previous sections. Here we give a very terse synopsis of some other changes expected in the C++ standard.

- The **`throw`** specification in function declarations will be deprecated (even thouch it will be included in the standard); new specification, **`noexcept`**, will be introduced for functions that do not throw exceptions.
- Range based **`for`** command (a loop iterating through a container) will be added.
- The **`explicit`** keyword may be used not only in the constructor declaration, but in the conversion function declaration in the C++0x. As you can expect, it will prevent the compiler to use of this conversion tool for implicit conversions.
- Attributes like **`[[carries_dependency]]`** or **`alignas()`** may be used in declarations. Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.
- The **`auto`** keyword will mean the storage class specifier no more. Instead, it is used either in the variable declaration where the compiler should infer the variable type from the initializer type, or in the new style function declaration to denote that the return type follows the parameter specification.
- R-value references (declared using the **`&&`** modifier) are introduced.
- The **`nullptr`** constant of the predefined type **`std::nullptr_t`** representing the null pointers should be used instead of the **`NULL`** macro.
- The **`export`** templates will be omitted. This keyword will be reserved “for further use”.
- The **`extern`** template specification is introduced. This specification will prevent the template instantiation.
- Template aliases, i.e. the declaration of the **`typedef`** templates, will be added.
- Variadic templates (templates with variable number of parameters) may be used.
- New container classes including **`unordered_set`** and **`unordered_multiset`**, **`unordered_map`** and **`unordered_multimap`** will be added to the STL. (Those are the hash table implementations.)
- Support for multiple random number generators producing uniform probability distribution and “engines” generating a set of predefined probability distributions will be added to the STL.
- Multiple auxiliary classes were added to STL.
- Data types **`long long`**, **`unsigned long long`**, new character types and headers to treat integer types (according to the C standard [3]) will added, as well as the tools to access the computational environment. Rules for the integral promotions will be based on the conversion ranks of the integral types.
- String literals supporting the UTF-16, UTF-32 and other Unicode formats will be added.
- Operators for the programmer-defined literal definition will be added. (It will be possible to define the suffix for the definition the short constant, e.g.)
- In the **`enum`** declaration, it will be possible to specify an integral type as the *underlying type.* The variables of the **`enum`** type will be stored with the layout of the underlying type.

- New forms of the **enum** declaration will be available. The **enum class** and **enum struct** declarations will introduce the *scoped* (strongly typed) *enum*. The enumerators of this type will have to be qualified by the type name using the **::** operator and no implicit conversion to integral types will be provided. Note that **enum class** and **enum struct** should be semantically equivalent.
- Multithreading support will be part of the core language. It will be based on the **std::thread** class, mutexes (represented by the **std:: mutex** and **std:: recursive_mutex** classes) and other synchronization facilities and the promises and futures representing the not yet finished computations.
- The **thread_local** storage class specifier will serve to declare thread-specific static data.
- New class **std::unique_ptr** for the smart pointers should be introduced.
- The **std::auto_ptr<>** template, the **std::unary_function**, **std::binary_function** and some other auxiliary classes should be deprecated.
- Alternative integer types (**int32_t** etc., described in **<inttypes.h>**) according to the current C standard [3] are introduced.

Some features, included in the previous standard proposals, will not be included:

- Concepts (probably not yet finished),
- decimal types,
- special mathematical functions in the like the Bessel functions, Legendre polynomials etc.

6. C++0x AND THE PROGRAMMING PRACTICE

The new features introduced by C++0x may be divided into several groups.

6.1 Completing the missing features

Some standard library classes, e.g. the hash tables, were intended, but not included in the previous C++ standards, because their proposal was not finished in time. The fact that they are eventually supplied will close the gap in the standard library that was filled in by several proprietary – and thus mutually incompatible – implementations.

6.2 Deprecated features of the previous standard

There were a few features of the C++03 that led to unclean code; some of them seemed to be very useful in the early 90's, when they were designed, but proved to be rather problematic. One the most apparent one was the freedom not to declare explicitly the override of the virtual function in the derived class. This could result in obscure errors in the program behavior that were difficult to find.

Other problematic feature is the **throw** clause in the function declaration, because it introduces strong code coupling. Changing the exceptions thrown in one function could result in sudden death of the whole program, because the specification of the thrown exceptions was not changed in some other function calling the first one.

These features must be kept for the backward compatibility, but should be prevented in the new code.

6.3 Features increasing the programmers comfort

Language features like the **auto** keyword denoting that the compiler should infer the type of the variable, the lambda-expressions etc. simply increase the programmers comfort. They enable terser, more concise expression of the programmer's intent. *If carefully used*, they may lead to cleaner, easier-to-understand and easier-to-maintain programs.

6.4 Backward compatibility

This is probably the most important aspect of the new standard. The standardization committee declares that the new C++ will be "almost" 100% compatible with the C++03. It is clear that introduction of the new features or the extensions of existing ones will not cause problems in programs adhering to the C++03 standard. The `auto` keyword was in fact never used, thus the change of its meaning would cause no troubles, too.
The export templates omission will be more problematic, because some compilers supported it recently. This will probably raise the necessity to rewrite some proprietary header files. The deprecation of the throw clause in the function declaration will – may be – change the programming practice, but the compilers will have to support it for some time, because it was in use for a long time and in some cases it was required by the previous standard.

7. CONCLUDING REMARK

This article is based on the draft of the standard, thus some features described here may change in the final release. Nevertheless, it is obvious that the new C++ version will be easier to use for experienced programmers. On the other hand, C++ is probably the most complicated programming language ever used; some of the changes introduce new complexity to it.
Up to now, only few compilers implement the new features – and of course no one implement them fully. It will take several years to the full impact of the changes.

Acknowledgement
The work on this article was supported by grants no. LA08015 a SGS 11/167 of the Ministry of Education, Youth and Sports of the Czech Republic.

REFERENCES

[1] ISO/IEC 14882:1998. Programming Languages – C++. Genève: ISO/IEC 2003.

[2] Working Draft. Standard for Programming Language C++. Doc. No. N3242. Genève: ISO/IEC 2011. Available at
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
[3] ISO/IEC 9899:1999. Programming Languages – C. Genève: ISO/IEC 1999.

[4] Stroustrup, B., Ellis, M.: In 1990, *The Annotated C++ Reference Manual*. Addison-Wesley 1991. ISBN 0-201-51459-1