# C++11, C++14, and C++17 for the Impatient: Opportunities in Computational Finance

**Daniel J. Duffy**
Datasim Finance, e-mail: dduffy@datasim.nl
**Avi R. Palley**
Quantitative Developer, Baruch/Quantnet C++ TA

**Abstract**
This is the first article in a mini series of two articles on applying C++11 to computational finance. Here we focus on the new syntax and features that improve and enhance the efficiency, reliability, and usability of C++ as a language for application development. We introduce the C++ *building blocks* in the form of data types, containers, and polymorphic function types that allow developers to design applications based on a combination of the object (-oriented), generic, and functional programming styles. In the second article we will introduce a language-independent *defined process* based on domain architectures (Duffy, 2004) to decompose a software system into loosely coupled subsystems, each of which has a single responsibility and having well-defined interfaces to and from other subsystems. Having created a *design blueprint* (similar to an architectural drawing), we then implement the subsystems using the multiparadigm programming features in C++. We then design and implement a Monte Carlo option pricing framework in C++11 by mapping a specific domain architecture to C++11. We also show how the same design blueprint can be implemented in C#.

**Keywords**
C++11, C++14, computational finance, functional programming, Monte Carlo, option pricing

## Background

What is C++? C++ is a general-purpose programming language that was originally designed as an extension to the C programming language. Its original name was "C with classes" and its object-oriented roots can be traced to the programming language Simula, which was one of the first object-oriented languages. C++ has been standardized by the International Organization for Standardization (ISO) in 1998 (called the C++03 standard) and C++14 is the standard at the moment of writing. This can be seen as a minor extension to C++11, which is a major update to the language.

C++ was designed primarily for applications in which performance, efficiency, and flexibility play a vital role. In this sense it is a *systems programming language* and early adopters in the 1990s were organizations in telecommunications, embedded systems, medical devices, and computer-aided design (CAD), as well as first-generation option pricing risk management systems. The rise in popularity continued well into the late 1990s as major vendors such as Microsoft, Sun, and IBM began to endorse object-oriented technology and C++. It was also in this period that the Java programming language appeared, which in time became a competitor to C++.

C++ remains one of the most important programming languages at the moment of writing. It is evolving to support new hardware such as multicore processors, GPUs (graphics processing units), and heterogeneous computing environments. It also has a number of mathematical libraries that are useful in computational finance.

## C++ as a multiparadigm programming language

We give an overview of the programming paradigms that C++ supports. In general, a *programming paradigm* is a way to classify programming languages according to the style of computer programming. Features of various programming languages determine which programming paradigms they belong to; as a result, some languages fall into only one paradigm, while others fall into multiple paradigms. C++ is in this sense a *multiparadigm programming* language because it supports the following styles.

- *Procedural*: it organizes code around functions, as typically seen in programs written in C, Fortran, and Cobol. The style originates from *structured programming*, in which a function or program is decomposed into simpler functions.
- *Object-oriented*: it organizes code around classes. A class is an abstract entity that encapsulates functions and data into a logical unit. We instantiate a class to produce objects. Furthermore, classes can be grouped into hierarchies. It is probably safe to say that this style is the most popular one in the C++ community.
- *Generic/template*: templates are a feature of C++ that allow functions and classes to operate with generic types. A function or class can then work on different data types without having to rewrite code for each one.

- *Functional*: it treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm; this means that programming is done with expressions and declarations instead of statements. The output value of a function depends only on its input arguments.

The generic programing style is becoming more important and pronounced in C++, possibly at the expense of the pure object-oriented model which is based on class hierarchies and subtype (dynamic) polymorphism. In this sense, template code tends to perform better at run-time while many errors are caught at compile-time in contrast to object-oriented code, where the errors tend to be caught by the linker or even at run-time.

The most recent style that C++ has (limited) support for is functional programming. This style predates both structured and object-oriented programming. Functional programming has its origins in *lambda calculus*, a formal system developed in the 1930s to investigate computability, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. C++ supports the notion of lambda functions. A *lambda function* in C++ is an unnamed function but it has all the characteristics of a normal function. Here is an example of defining a *stored lambda function* and then calling it as a normal function:

```
#include <iostream>
#include <string>

int main()
{
       // Captured variable
       std::string cVar("Hello");

       // Stored lambda function, with captured variable
       auto hello = [cVar](const std::string& s)
       { // Return type automatically deduced

          std::cout << cVar << " " << s << '\n';
       };

       // Call the stored lambda function
       hello(std::string("C"));
       hello(std::string("C++"));

       return 0;
}
```

In this case we see that the lambda function has a formal input string argument and it uses a so-called *captured variable* `cVar`. Lambda functions are simple but powerful and we shall show how they can be used in computational finance.

C++11 is a major improvement on C++03 and it has a number of features that facilitate the design of software systems based on a combination of structured analysis and object-oriented technology. In general, we have a defined process to decompose a system into loosely coupled subsystems (Duffy, 2004). We then implement each subsystem in C++11, as we shall see in later articles.

This article is organized as follows: in Section I we introduce smart memory management in C++11, which resolves many of the problems that developers experienced with native pointers in previous versions of the language. Their application in code promotes robustness and reliability. In Section II we discuss a number of useful low-level features that avoid some shortcomings in C++03. Section III is devoted to a bird's-eye view of functional programming and how C++ supports it using lambda functions and universal function wrappers. Tuples (generalizations of `std::pair`) and variant data types are discussed in Section IV. Finally, in Sections V and VI we discuss how to model stochastic differential equations (SDEs) using the multiparadigm features in C+11. We shall continue in this vein in the second article when we create a framework for option pricing using the Monte Carlo method.

A more extensive discussion of all these topics is given in Duffy (2017).

# I Smart memory management and move semantics

## Using smart pointers in code

We discuss three pointer classes in C++. In the next three subsections we focus on the syntax of each one and give some simple examples to show what they do.

### Class std::shared_ptr

This smart pointer class implements the concept of *shared ownership*. A *resource* or object (a piece of memory on the heap) is shared among a number of shared pointers. Only when the resource is no longer needed is it deleted. This is when the related *reference count* becomes zero.

We discuss shared pointers. First, we show how to create empty shared pointers and shared pointers that are coupled to heap resources. We also show how a shared pointer gives up ownership of one resource and how it becomes owner of another resource. We can always see how many shared pointers *own* a resource by using the member function `use_count()`:

```
#include <memory>

// Handy alias
template <typename T>
    using SP = std::shared_ptr<T>;
using value_type = double;

// Creating shared pointers with default deleters
SP<value_type> sp1;                              // empty shared ptr
SP<value_type> sp2(nullptr);                     // empty shared ptr for C++11 nullptr

SP<value_type> sp3(new value_type(148.413));     // ptr owning raw ptr
SP<value_type> sp4(sp3);                         // share ownership with sp3
SP<value_type> sp5(sp4);                         // share ownership with sp4 and sp3

// The number of shared owners
std::cout << "sp2 shared # " << sp2.use_count() << '\n';  // 0
std::cout << "sp3 shared # " << sp3.use_count() << '\n';  // 3
std::cout << "sp4 shared # " << sp4.use_count() << '\n';  // 3

sp3 = sp2;  // sp3 now shares ownership with sp2;
            // sp3 no longer has ownership of its previous resource
std::cout << "sp3 shared # " << sp3.use_count() << '\n';          // 0
std::cout << "sp4 shared # " << sp4.use_count() << '\n';          // 2
```

In the above cases the last owner of the resource is responsible for destroying the resource and by default this is achieved by a call of the operator `delete`.

### Class std::unique_ptr

Whereas `std::shared_ptr` allows a resource to be shared among several shared pointers in the case of `std::unique_ptr`, there is only one transferable owner of a resource. In this case we speak of *exclusive* or *strict ownership*. Its main added value is in avoiding *resource leaks* (for example, missing calls to `delete` when using raw pointers) and for this reason is can be called an *exception-safe* pointer. Its main member functions are as follows.

- Constructors (similar to those in `std::shared_ptr`).
- Assign a unique pointer.
- Release; return a pointer to the resource and release ownership.
- Reset; replace the resource.
- Operator overloading (==, !=, <, and so on).

The interface is similar to that of `std::shared_ptr`, which means that code will be easy to understand. Finally, `std::unique_ptr` succeeds `auto_ptr`, the latter being considered deprecated.

Our first example entails creating a unique pointer in a scope. Under normal circumstances, when the pointer goes out of scope, the corresponding resource is cleaned up but in this case we (artificially) throw an exception before the end of the scope is reached. What happens? When we run the code we see that the resulting exception is caught *and* the resource is automatically destroyed:

```
template <typename T>
    using UP = std::unique_ptr<T>;

try
{
    // Unique pointers

    // Stored lambda function as deleter
    auto deleter = [](value_type* p)
             { std::cout << "bye, bye unique pointer\n"; delete p; };
    UP<value_type> sp32(new value_type(148.413), deleter);

    throw - 1;
}
catch(int& n)
{
    std::cout << "error but memory is cleaned up\n";
}
```

This code also works when we use shared pointers instead of unique pointers. In C++14 we can create unique pointers using `std::make_unique()` for non-array types:

```
// Other examples with unique pointers
// More efficient ways to construct unique pointers
auto up = std::make_unique<int>(42);
(*up)++;
std::cout << "up: " << *up << '\n';        // 43

auto up2 = std::make_unique<Point2d>(-1.0, 2.0);
(*up2).print();             // (-1, 2)

auto up3 = std::make_unique<Point2d>();
(*up3).print();             // (0,0)
```

### std::weak_ptr

The third smart pointer class holds a non-owning (*weak*) reference to an object (resource) that is managed by a shared pointer. It must be converted to a shared pointer in order to access the resource. It is a helper class to `std::shared_ptr` and it is needed when the latter's behavior does not work as intended, namely:

- Resolving *cyclical dependencies* between shared pointers – dependencies that occur when two objects refer to each other using shared pointers. We cannot release the objects because each has a use count of 1. It is like a deadlock.
- Situations in which you wish to share an object but you do not own it. In this case we define a reference to a resource and that reference outlives the resource.

## II Low-level syntax improvements

We now give a discussion of a number of useful features in C++. They improve the readability and robustness of code. In other cases they represent structural solutions to workarounds that were used in previous versions of C++ to resolve problems. You may need to use these new features in your code.

### Type alias and alias templates

Developers are already familiar with the `typedef` specifier, which allows aliases to be defined for existing types. The specifier cannot be used to change the meaning of an existing type name and hence it does not represent the declaration of a new type. A `typedef` has effect in the scope in which it is visible. An example is

```
typedef std::map<std::string, double> Dictionary;
```

The use of an alias promotes code readability and maintainability. Unfortunately, `typedef` cannot be used for class templates and hence the following code will not compile:

```
/*
template <typename Key, typename Value>
    typename std::map<Key, Value> DictionaryII;

template <typename Value>
    typename std::map<std::string, Value> DictionaryIII;
*/
```

But there is hope! C++11 offers the possibility to define an alias for class templates and non-template classes alike. The term is sometimes called `alias template` or `template typedef`. The above example can now be formulated as follows:

```
using Dictionary = std::map<std::string, double>;

template <typename Key, typename Value>
    using DictionaryII = std::map<Key, Value>;

template <typename Value>
    using DictionaryIII = std::map<std::string, Value>;
```

We can also use this technique in combination with user-defined types with long names and with types having many template parameters in order to make it easier to work with them. Let us take the simple example

```
template <typename T1, typename T2>
    class C
{
private:
public:
    T1 t1;
    T2 t2;
public:
    C() : t1(T1()), t2(T2()) {}
    C(const T1& t1Val, const T2& t2Val) : t1(t1Val), t2(t2Val) {}

    void print() const
    {
       // T1 and T2 must support <<
       std::cout << t1 << ", " << t2 << '\n';
    }
    // other code here
};
```

We can imagine that clients might wish to use this class in different ways, and to this end we can create various aliases, for example

```
template <typename T1, typename T2>
    using C1 = C<T1, T2>;

// 'Nested' alias
template <typename T>
    using Vec = std::vector<T>;

template <typename T1, typename T2>
    using C2 = C<T1, Vec<T2>>;

template <typename T>
    using C3 = C<T, int>;

using C4 = C<double, int>;
// Using alias template
C4 c4; // 0,0
c4.print();

float f = 1.0F;
int j = 3;
C3<float> c3(f, j);
c3.print(); // 1,3

// Nested alias in class C2 for readability
Vec<double> v(100);
int n = 10;
C2<int, double> c2(n, v);
C<int, std::vector<double>> c2_1(n, v);

C1<int, int> c1;
C<int, int> c(1, 2);
```

We have given a number of examples to show the usefulness of this technique. It should not be confused with the well-known `using` declaration, which introduces a name that is defined elsewhere.

## Automatic type deduction and the auto specifier

This feature allows us to declare a variable or an object without using its specific type. In other words, the variable is automatically deduced from its initializer. Some examples are

```
// auto variable
double d1 = 1.0; float f1 = 2.0f; int p1 = 10;
auto d2 = d1 + f1;
auto d3 = d1 + p1;
auto d4 = f1 + p1;
```

We could have used explicit return types, but the compiler is clever and it knows what the correct type should be, as the following checks show based on the C++ type traits library:

```
// Example resulting type by type_traits (chapters 4 and 6)
static_assert(std::is_same<decltype(d1), double>::value, "ouch");
static_assert(std::is_same<decltype(d2), double>::value, "ouch");
static_assert(std::is_same<decltype(d3), double>::value, "ouch");
// FAIL static_assert(std::is_same<decltype(d4), double>::value, "ouch");
static_assert(std::is_same<decltype(d4), float>::value, "ouch");
```

We can also use the `auto` specifier to make life easier when working with user-defined types:

```
// auto variable with user-defined types

using CP = std::complex <double>;
CP c(1.0, 2.0);
C<CP, CP> cA(c, c);
auto cB = cA;
static_assert(std::is_same<decltype(cB), C<CP, CP>>::value, "ouch");
```

## Deleted and defaulted member functions

The specifiers in this subsection are primarily concerned with C++03's *special member functions*, namely:

- Default constructor.
- Copy constructor, which corresponds to member-wise construction of non-static data members. It is generated only if the class does not have a user-defined copy constructor.
- Copy assignment operator. Generation of this function in a class with a user-defined copy constructor is deprecated.

These functions are generated only if they are needed. For example, a default constructor is generated only if a class declares no constructors whatsoever. Generated special member functions are implicitly public and inline. In C++11 the club of special member functions has been extended to include the move constructor and the move assignment operator.

We use the keyword `default` to let the compiler generate the body of a constructor. At the other extreme, we may wish to suppress the use of special member functions. The standard approach in C++03 is to declare them as `private` and to not define them. C++11 uses the specifier `delete` to mark these functions as so-called *deleted* functions, which is an improvement on C++03 for a number of reasons:

- Deleted functions may not be used, even by friends.
- Deleted functions are `public`; C++ checks accessibility before deleted status.
- Errors are caught at compile-time rather than at link-time, in contrast to the case with C++03.
- More user-friendly error messages.
- Any function (and not just member functions) can be given the `delete` status. This is useful when we wish to avoid implicit conversions.

We take an initial example of a class with a mixture of deleted and defaulted functions:

```
// Explicitly defaulted and deleted functions
class F
{
public:
    // Support default operations OK
    F() = default;

    // Move ctor and move assignment are defined
    F(F&&) = default;
    F& operator = (F&&) = default;

    F(const F&) =  delete;
    F& operator = (const F&) = delete;

    virtual ~F() = default;
};
```

Some test code shows what does and does not compile:

```
// Explicitly defaulted and deleted functions
F f;

// Both give error message like 'F::F(const F &)': attempting to
// reference a deleted function
//F f2 = f;
//F f3(f);

F f4 = std::move(f);
```

Finally, we give an example of a deleted non-member function. In this case we define some overloaded print functions that are deleted when the input argument is a `double` or a `bool`:

```
// Overloaded non-member function
void print(float f)
{
    std::cout << f << '\n';
}

void print(long n)
{
    std::cout << n << '\n';
}
// Deleted for these types

void print(bool b) = delete;
void print(double d) = delete;
```

An example of use (commented lines do not compile):

```
// Deleted overloaded non-member function
float f5 = 4.0F;
double d = 5.0;
long p = 3;
print(f5);
print(p);

// void print(bool)': attempting to reference a deleted function
// print(true);
// print(d);
```

## Uniform initialization

In previous versions of C++ there was some confusion concerning how to initialize variables and objects. It was not always clear when to use braces, parentheses, or assignment operators. In C++11 we can avail a single common syntax called *uniform (braced) initialization* by placing the values between braces. Some simple examples are

```
// Initializer list (they are recursive)
print<int>({ 3, 4, 5, 6, 99 });

// Uniform initialization
double arr[] {1.0, 2.0, 3.0};
std::list<int> arr2{ 1, 3, 5, 7 };

std::tuple<double, double> tup(1.0, 2.0);
std::map<int, std::tuple<double, double>> database{ { 1, tup }, { 2, tup } };

for (auto it=std::begin(database);it != std::end(database); ++it)
{
   std::cout << it->first << ": "
          << std::get<0>(it->second) << ", " << std::get<1>(it->second) << std::endl;
}
```

We can use this technique to initialize the members of an aggregate object. Consider the classes

```
// Struct that holds data together
struct Person
{

    std::string name_;
    std::string address_;
    int age_;

};

struct Person2
{
    std::string name_;
    std::string address_;
    int age_;

    Person2(int age, const std::string name, const std::string address)
       : age_(age), name_(name), address_(address) {}
};
```

We use uniform initialization as follows:

```
// Aggregate initialization; initialize the data in a
// struct without needing a constructor
Person p1{ "Joe", "Denver", 23 };

// ERROR! Person p1_a( "Joe", "Denver", 23 );
// (No function talking three arguments!!)

// Using braced initialization and constructor to initialize data
Person2 p2{ 55, "Piet", "Zurich" };
Person2 p3( 55, "Piet", "Zurich" );
```

We see from class `Person` that we can initialize a class's member data using braced initialization, even though the class does not have the appropriate constructors.

# III Functions and functional programming

## An introduction to functional programming

*Functional programming* is a style of programming that models computation as the evaluation of expressions. In other words, we execute programs by evaluating expressions. Functional programming requires that functions be treated as *first-class objects* so that they can be used just like an ordinary variable in a programming language, for example:

- they can be passed as arguments to other functions;
- they can be returned as a result of a function;
- we can define and manipulate functions from within other functions.

In general, functional programming languages support these features while historically *imperative programming languages* (such as C++) did not. In recent times a number of extensions to C++ have been proposed that support some of the above features, in particular several Boost C++ libraries and C++11. An *imperative programming language* is one in which programs are composed of statements. These statements can change global state when executed, while in functional programming languages program state tends to be *immutable*.

Some features of functional programming languages are

- Support for *higher-order functions* (HOFs). A higher-order function can take other functions as arguments. An example of a higher-order function is a loop of the form (pseudocode)

```
foreach(record.begin(), record.end(), func);
```

where `record` is a data collection and `func` is a function that operates on each element of that collection. Higher-order functions are useful in code refactoring projects because their use reduces the amount of code repetition. We shall see how to simulate higher-order functions in C++ by the use of *function objects* (*functors*) and *lambda functions*.

- *Purity*. In general, some functional programming languages allow expressions to yield actions in addition to returning values. These actions are called *side effects*. A *pure language* is one that does not allow side effects.
- *Recursion*. This technique is pervasive in functional programming and it is sometimes the only way to iterate. Use is often made of *tail call optimization*, which avoids recursion using too much memory.
- *Strict* and *non-strict evaluation*. These terms refer to how function arguments are processed when an expression is being evaluated. In the former case, all arguments are instantiated and then the function returns a value. In the latter case, some arguments have not yet been instantiated (these are called *delayed*

*arguments*). It is then not possible to call the function in the traditional sense but what we get is a function with a given arity, which can later be called by instantiating all the remaining delayed arguments. This is a useful feature because we can create new functions from existing ones by *binding* one or more arguments to specific values.

Functional programming languages provide better support for structured programming than imperative programming languages. This is mainly due to the fact that we can model abstractions and decompose them into *software components*. It is possible to implement such components using higher-order functions. We shall see in Duffy (2017) how to define higher-order functions in C++ and use them to design and implement applications for computational finance.

## New functionality in C++: `std::function<>`

The class `std::function` (defined in `<functional>`) provides polymorphic wrappers that generalize the function pointer concept. Its origins can be traced to the *Boost C++ Function library* (as discussed in Demming and Duffy, 2010). The main advantage is that it can be used as a bridge to callable objects such as free functions, member functions, function objects, and lambda functions. To this end, the class `std::function<>` allows us to model these callable objects as first-class objects. As an example, we show how to use `std::function<>` in combination with free functions. Let us first define three functions:

```
double Func(double x)
{
	return x*x;
}

double Func2(double x)
{
	return std::sqrt(x);
}

double Func3(double x)
{
	return std::exp(x);
}
```

We see that these functions accept a `double` as input argument and they return a `double`. In order to subsume these functions in a polymorphic function wrapper as it were, we define

```
std::function<double (double sx)> f;
```

We can then assign an instance of `f` to a free function, for example

```
// Assign to a global function
f = Func;
std::cout << "First function: " << f(2.0) << std::endl;
```

In client code we can work with instances of `std::function<>` which have been instantiated somewhere else. This promotes code reusability and we can use `std::function<>` with lambda functions, function objects, and member functions.

The second example is to define an array of functions and then execute them in turn. We add each of the above free functions to the array of `std::function<>` instances and use a lambda function to execute each one:

```
// Create an array of functions
typedef std::function<double (double sx)> Task;
typedef std::vector<Task> Tasks;
Tasks tasks;
tasks.push_back(Func);
tasks.push_back(Func2);
tasks.push_back(Func3);

double val = 10.0;              // Captured variable
std::for_each(tasks.begin(), tasks.end(), [&val] (const Task& t)
                        { std::cout << t(val) << std::endl;});
```

This example gives an indication of what is possible with `std::function<>`. For example, we could use it to implement *callback functions* in the *Observer* pattern (Gamma *et al.*, 1995). More generally, we can use the *Boost C++ signals2* library as implementation of *Observer* (Demming and Duffy, 2010).

## New functionality in C++: Lambda functions and lambda expressions

Lambda functions are unnamed functions that can be declared within the scope of other functions. They are alternatives to using free functions and *function objects* (*functors*) in C++.

### Basic syntax

The general syntax for a lambda function is

```
[capture variables](input arguments) <mutable> -> return type
{
  function implementation
}
```

First, we see that the presence of the brackets [] announces a new lambda function having a given return type. Second, a lambda function has input arguments as well as so-called *captured variables* that belong to the lambda function's closure. These variables correspond to the lambda function's state. It is possible to capture variables by value or by reference, depending on whether we wish to copy by value or by reference into the body of the lambda function. We can specify the choice by using the following list as exemplar:

```
[a]       // Capture local variable 'a' by value.
[a, b]    // Capture vars 'a' and 'b' by value
[&a]      // Capture var 'a' by reference
[a, &b]   // Capture 'a' by value, 'b' by reference
[=]       // Capture all used variables by value
[&]       // Capture all used vars by reference
[=, &a]   // 'a' by ref; other vars by value
```

We also note the presence of the optional keyword `mutable` in the above specification. This is sometimes needed because lambda functions are `const` and hence their state cannot be changed. Thus, in order to make the lambda function non-`const`, we declare it as `mutable`. Finally, the body of the lambda function is standard C++ and it may use both its input and captured variables.

The simplest lambda function has the structure

```
[]
{
	function implementation
}
```

All lambda functions have a return type (which can be `void`). It is not necessary to specify explicitly what this return type is unless the body of the function contains embedded if–else logic. We say that the return type is *inferred* or *deduced* when it is absent.

### Lambda functions and classes: Capturing member data

It is possible to define lambda functions in a class and it is also possible to capture its member data. The rules are

| | |
|---|---|
| `[this]` | captures the *this* pointer only |
| `[=]` | captures the *this* pointer and local variables by value |
| `[&]` | captures the *this* pointer and local variables by reference. |

Inside the body of the lambda function we access the members in the usual way. We take an example of a class to calculate the trigonometric sine of a vector with input values. To this end, we transform the input vector using a lambda function. The objective is to use a lambda function to effect the transformation:

```
class CalculateSine
{
private:
    double m_amplitude;

public:
    CalculateSine(): m_amplitude(0.0) {}
    CalculateSine(double amplitude): m_amplitude(amplitude) {}

    // Calculate sine.
    std::vector<double> Calculate(vector<double>& input)
    {
       // Create result vector.
       std::vector<double> result(input.size());

       // Use STL algorithm
       std::transform(input.begin(), input.end(), result.begin(), [this](double v)-> double
                         {return m_amplitude*sin(6.28319*v/360.0); } );

       // Return the result.
       return result;
    }
};
```

# IV Data types: Tuples and variants

## Tuple fundamentals and simple examples

Tuples are generalizations of pairs in the sense that the number of elements in the latter type is greater than or equal to two.

We give an initial example. We present the code and it should be easy to understand, as it is similar to the corresponding code for `std::pair<>`:

```
using TupleType = std::tuple<std::string, int>;

// Creating tuples
TupleType myTuple(std::string("A"), 1);
TupleType myTuple2 = std::make_tuple("B", 0);
TupleType myTuple3(myTuple);
auto myTuple4 = myTuple2;

// Accessing the elements of a tuple
std::cout << std::get<0>(myTuple) << std::get<1>(myTuple) << '\n';       // A,1

std::get<0>(myTuple) = std::string("C");
std::get<1>(myTuple) = 3;
std::cout << std::get<0>(myTuple) << std::get<1>(myTuple) << '\n';       // C,3
```

Extending the code to tuples with more than two elements is straightforward.

## Tuple nesting

A useful feature in C++ is that we can define fixed-size recursive and composite structures. In other words, we can define tuples that contain other tuples as well as other types as elements. This feature can be applied to creating *configuration data* in an application. This data can be created by dedicated factory objects and the data is then processed in an application at a suitable *entry point* in the code.

Some typical applications spring to mind:

- Modeling data for rainbow and multi-asset options by a tuple with three elements. The first and second elements are themselves tuples containing the data (like strike, expiration) pertaining to the first and second assets, respectively, while the third element contains the value of the correlation between these assets.
- Mathematical properties of differential equations encapsulated in reusable tuples.
- Essential defining parameters pertaining to numerical processes, for example step size, tolerances, and maximum number of iterations.
- Any non-trivial application that needs to be configured from various sources. To this end, we use tuples that client code reads without having to know where the data comes from.

We give a simple example of how to create nested tuples to model people and their whereabouts (it can be seen as a simple database):

```
// Nested tuples
// First and last names
typedef std::tuple<std::string, std::string> Name;

// Home address and city and code
typedef std::tuple<std::string, std::string, long> HomeAddress;
typedef std::tuple<std::string, std::string> Location;

// Name + Address + Location
typedef std::tuple<Name, HomeAddress, Location> NHL;

Name pName;
std::get<0>(pName) = std::string("Frankie");
std::get<1>(pName) = std::string("Carbone");

HomeAddress ha;
std::get<0>(ha) = std::string("Sunnyside");
std::get<1>(ha) = 199;

Location loc;
std::get<0>(ha) = std::string("Queens");

std::string code("AK");
std::get<1>(ha) = code;

// Create the top-level (Whole)tuple
NHL nhl;
std::get<0>(nhl) = pName;
std::get<1>(nhl) = ha;
std::get<2>(nhl) = loc;
```

## Variadic tuples

In general, a *variadic template* is one that takes a variable (arbitrary) number of input arguments. In particular, tuples can be defined as having a variable number of elements:

```
template<class... Types>
    class tuple;
```

This declaration states that this class can take any number of typenames as its template parameters. Incidentally, the number of arguments can be zero. If we wish to forbid the definition of a template with zero arguments, then we can modify the definition as follows (in which case the tuple must have at least one element):

```
template<typename First, typename... Rest>
    class tuple;
```

Variadic templates also apply to functions, for example free functions:

```
template<class... Args>
    void print(const std::tuple<Args...>& t)
{

}
```

The advantages of variadic tuples are that:

- they can take an arbitrary number of arguments of any type;
- they reduce code explosion – you (*supplier*) can define code once that uses variadic templates and *clients* can customize the code by choosing the types and number of arguments that they wish to use;
- it is possible to design and implement highly generic code for some cases in which traditional object-oriented technology (using subtype polymorphism, for example) would lead to code that is difficult to understand or maintain;
- variadic templates are compile-time, which should lead to efficient and robust code.

We take an initial example of using variadic templates. In particular, we write a function to print a tuple with any number of elements. We peel the elements of a tuple from its end, print the last value, and then call the same function again on a tuple with one element less than the original one. The code is

```
// General case; recursive call from <N> to <N-1>
template<class Tuple, std::size_t N>
    struct TuplePrinter {
       static void print(const Tuple& t)
       {
          TuplePrinter<Tuple, N - 1>::print(t);
          std::cout << ", " << std::get<N - 1>(t);
       }
};
```

Since the function is recursive, we know that we need to define its very last action as it were (this is the equivalent of *tail recursion*). To this end, we employ template specialization:

```
// Tail recursion case when N = 1 (stopping criteria)
template<class Tuple>
    struct TuplePrinter<Tuple, 1> {
       static void print(const Tuple& t)
       {
          // Assume << is overloaded
          std::cout << std::get<0>(t);
       }
};
```

## std::variant (C++17) and boost::variant

The union data type is not very flexible. It seems that they will be superseded in C++17 by `std::variant`, which can then replace unions and union-like classes. Since we do not have a suitable C++17 compiler at the moment of writing, we resort to a discussion of how `boost::variant` works (Demming and Duffy, 2010). From the Boost online documentation:

> The variant class template is a safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as `std::vector` may be thought of as "multi-value, single type," variant is "multi-type, single value."

The main uses of variants are:

- typesafe storage and retrieval of user-specified sets of types;
- storing heterogeneous types in STL containers;
- compile-time checked visitation of variants (using the *Visitor* pattern);
- efficient, stack-based storage for variants.

We take an example of a variant consisting of three members:

```
#include <boost/variant.hpp> // Includes all other header files

using boost::variant;

// Some ways to construct a variant
variant<long, std::string, Point> myVariant;
// Give some values
myVariant = 24;

myVariant = std::string("It's amazing");
myVariant = Point(3.0, 4.0);
```

For completeness, we include the definition for class `Point`:

```
struct Point
{
    double x, y;
    Point() : x(0.0), y(0.0) {}
    Point(double xVal, double yVal) : x(xVal), y(yVal) {}

};

void print(const Point& pt)
{
    std::cout << "(" << pt.x << "," << pt.y << ") ";
}
```

We can now get the current value in a variant by calling the `get()` function:

```
// Try to get the value out of the variant
Point ptA;
try
{
     ptA = boost::get<Point>(myVariant);
}
catch (boost::bad_get& err)
{
    std::cout << "Error: " << err.what() << std::endl;
}

 std::cout << "Value got from Variant: "; print(ptA);
```

# V Example: Stochastic differential equations in C++11

## Application to SDEs

Most developers are familiar with subtype polymorphism and developing class hierarchies in application domains. In general, we can create a base class consisting of pure `virtual` functions, which are then overridden by derived classes. For example, this was the approach taken by Duffy and Kienitz (2009) to model classes for SDEs. In particular, we model drift and diffusion as functions that can be customized in derived classes. While this is an accepted technique, we do run the risk of creating redundant code by essentially promoting objects to the level of a class.

In this section we create a single class that models an SDE. The class is composed of two universal functions that implement the drift and diffusion function. It is up to clients to decide how to implement these functions when creating an instance of the SDE. This can be realized by the following:

- a free function (traditional function pointer);
- a stored lambda function or a lambda function;
- a function object;
- a binded member function or static member function of a class.

This approach reduces code bloat, especially at the server side, because only a single class is needed. We first describe some supporting syntax and shorthand notation to make the resulting code more readable:

```
// Functions of arity 2 (two input arguments)
template <typename T> using FunctionType
    = std::function<T (const T& arg1, const T& arg2)>;

// Interface to simulate any SDE with drift/diffusion
// Use a tuple to aggregate two functions into what is similar to an
// interface in C# or Java.
template <typename T>
    using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;
```

We are now in a position to post the class that models the SDE; we note that it makes use of functions and tuples:

```
template <typename T = double>
    class Sde
{
private:
    FunctionType<T> dr_;
    FunctionType<T> diff_;

    T ic; // Initial condition
public:
    Sde() = default;
    Sde(const Sde<T>& sde2, const T& initialcCondition)
       : dr_(sde2.dr_), diff_(sde2.diff_), ic(initialcCondition) {}
    Sde(const ISde<T>& functions, const T& initialcCondition)
       : dr_(std::get<0>(functions)),
       diff_(std::get<1>(functions)), ic(initialcCondition) {}

    T drift(const T& S, const T& t) { return dr_(S, t); }
    T diffusion(const T& S, const T& t) { return diff_(S, t); }
};
```

Some examples of how to interact with this class are when creating SDEs for geometric Brownian motion (GBM) and constant elasticity of variance (CEV) models:

```
// Second test case; GBM dS = a S dt + sig S dW
double r = 0.08; double sig = 0.3;
auto drift = [&r](double t, double S) { return r * S; };
auto gbmDiffusion = [&sig](double t, double S) { return sig * S; };

double beta = 0.5;
auto cevDiffusion = [&sig, &beta](double t, double S)
                    { return sig * std::pow(S, beta); };

auto iFace = std::make_tuple(drift, gbmDiffusion);
ISde<double> gbmFunctions = std::make_tuple(drift, gbmDiffusion);
ISde<double> cevFunctions = std::make_tuple(drift, cevDiffusion);

// Create Sdes
double ic = 60.0;
Sde<> sdeGbm(gbmFunctions, ic);
```

We see that we can create instances of SDE<> by defining a single lightweight class and in-place lambda functions.

## SDE factories

From the above code sample we see how to create instances of SDE<>. In larger applications this ad-hoc approach breaks down, in the sense that the code to initialize the data and functions that are used to create instances of SDE<> tends to become unmaintainable. For this reason we apply the separation of concerns approach, by creating code blocks with each block having its own (single) responsibility. A typical example is to create loosely coupled modules for the following actions that we describe in a top-down manner:

1. Create multiple instances of SDE<>; create a specific instance of SDE<> based on some user choice.
2. Create the functions that are the components of the SDE<> instances in step 1.
3. Create the data that the functions in step 2 need.

Each of these steps can be executed by dedicated factory objects that are in fact instances of *creational design patterns* as described by Gamma *et al.* (1995). We give an initial example of code in which steps 1, 2, and 3 are amalgamated into one code block. We can refine this process even more when we use the *Builder* design pattern (Gamma *et al.*, 1995) to configure a complete application. We take an example of a function that creates an arithmetic SDE (notice that the data is hard-coded but the code can be generalized). We use an alias for the factory class:

```
// Interface for a factor to create SDE<T> interfaces
template <typename T>
    using SdeFactory = std::function<std::shared_ptr<Sde<T>> () >;
```

The function to create an SDE instance and an example of use is

```
template <typename T>
    std::shared_ptr<Sde<T>> ArithmeticSde()
{ // dx = nu dt + sig dW, after transformation x = log(S)

    std::cout << "Arithmetic\n";

    // Initial condition for SDE
    double S0 = 60.0;

    // Necessary data
    T r = 0.08; T sig = 0.3;
    double nu = r - 0.5 * sig * sig;

    // Create the components of the SDE
    auto drift = [=](T t, T S) { return nu; };
    auto diffusion = [=](T t, T S) { return sig; };

    ISde<T> functions = std::make_tuple(drift, diffusion);

    // Create Sde
    return std::shared_ptr<Sde<T>>(new Sde<T>(functions, S0));
}

// Switch to a new factory and generate a sde
SdeFactory<float> arithmeticSdeFactory = ArithmeticSde<float>;
auto sde3 = arithmeticSdeFactory();
std::cout << "Drift, diffusion: " << sde3->drift(t, S) << ", "
          << sde3->diffusion(t, S) << '\n';
```

Instead of a new factory, we could have recycled as it were:

```
sdeFactory = ArithmeticSde<float>;
auto sde3 = sdeFactory();
std::cout << "Drift, diffusion: " << sde3->drift(t, S) << ", "
          << sde3->diffusion(t, S) << '\n';
```

Finally, we can simulate the *factory method* pattern by creating a function with an internal switch, allowing us to choose the type of SDE that we are interested in:

```
template <typename T>
    std::shared_ptr<Sde<T>> ChooseSde(int choice)
{ // Simple factory method

    // Initial condition for SDE
    double S0 = 60.0;

    if (1 == choice)
    {
       std::cout << "GBM\n";
       // Second test case; GBM dS = a S dt + sig S dW
       T r = 0.08; T sig = 0.3;
```

```
        auto drift = [=](T t, T S) { return r * S; };
        auto gbmDiffusion = [=](T t, T S) { return sig * S; };

        ISde<T> gbmFunctions = std::make_tuple(drift, gbmDiffusion);

        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(gbmFunctions, S0));

    else
    {
        std::cout << "CEV\n";
        double r = 0.08; double sig = 0.3;
        auto drift = [=](double t, double S) { return r * S; };
        auto gbmDiffusion = [=](double t, double S) { return sig * S; };

        double beta = 0.5;
        auto cevDiffusion = [=](double t, double S) { return sig * std::pow(S, beta); };

        ISde<double> cevFunctions = std::make_tuple(drift, cevDiffusion);

        // Create Sde
        return std::shared_ptr<Sde<T>>(new Sde<T>(cevFunctions, S0));
    }
}
```

An example of use is (notice the clever application of `std::bind`)

```
// Higher-level factory, can switch
SdeFactory<float> sdeFactory = std::bind(ChooseSde<float>, 1);
auto sde2 = sdeFactory();

std::cout << "Drift, diffusion: " << sde2->drift(t, S) << ", "
          << sde2->diffusion(t, S) << '\n';
```

## VI Emerging multiparadigm design patterns: Summary

In the previous section we applied modern C++ syntax to design a small framework to model SDEs as a typical example that we can generalize. We have avoided subtype polymorphism and instead took an approach based on creating a single class composed of universal function wrappers. This is similar to *interface programming* in languages that support interfaces (such as C# and Java, for example). In general, an *interface* is a *pure specification* or protocol and it describes a collection of abstract methods. An interface may not contain non-abstract methods nor may it contain member data. In this sense it is very different from an abstract class in C++ (even one without non-abstract member functions and member data). C++ does not support interfaces; however, we can emulate them by creating (variadic) tuples whose elements are universal function wrappers. In this case we define our interface emulator as

```
template <typename T>
    using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;
```

We create an instance of `ISde` as follows, for example:

```
auto drift = [=](T t, T S) { return nu; };
auto diffusion = [=](T t, T S) { return sig; };

ISde<T> functions = std::make_tuple(drift, diffusion);
```

We can then use functions in a constructor of `Sde<>`, as already discussed.

## Summary and conclusions

In this article we introduced some of the essential language features of C++11. With these fundamental building blocks we are able to create generic and compact data structures that we can use in future applications based on an innovative design.

**Daniel J. Duffy** is founder of Datasim Financial and has been involved with C++ and its applications since 1989. More recently he has been involved with the design of computational finance applications in C++11. He has a PhD in numerical analysis from Trinity College (Dublin University) and can be contacted at dduffy@datasim.nl.

**Avi R. Palley** works as a quantitative developer and is TA for the highly successful C++ Programming for Financial Engineering and the Advanced C++11/C++14 and Multidisciplinary Applications online certification courses (offered by Quantnet.com and originated by Daniel J. Duffy). He has a Masters in Financial Engineering from Baruch College, New York.

## References

Demming, R. and Duffy, D. 2010. *Introduction to the Boost C++ Libraries, Volume I.* Amsterdam: Datasim Press.
Demming, R. and Duffy, D. 2012. *Introduction to the Boost C++ Libraries, Volume II.* Amsterdam: Datasim Press.
Duffy, D. J. 2004. *Domain Architectures.* Chichester, UK: Wiley.
Duffy, D. 2017. *Financial Instrument Pricing Using C++*, 2nd edn. Chichester, UK: Wiley.
Duffy, D. and Kienitz, J. 2009. *Monte Carlo Frameworks.* Chichester, UK: Wiley.
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns.* Boston, MA: Addison-Wesley.