Immutables in C++: Language Foundation for Functional Programming

Zoltán Porkoláb(✉)

Faculty of Informatics, Department of Programming Languages and Compilers,
Eötvös Loránd University,
Pázmány Péter sétány 1/C, Budapest 1117, Hungary
gsd@elte.hu
http://gsd.web.elte.hu

Abstract. The C++ programming language is a multiparadigm language, with a rich set of procedural, object-oriented, generative and, since C++11, functional language elements. The language is also well-known for its capability to map certain semantic features into the language syntax; therefore, the compiler can reason about them at compile time. Supporting functional programming with immutables is one of such aspects: the programmer can mark immutable components and the compiler checks potential violation scenarios and also optimizes the code according to the constant expectations.

The paper targets the non-C++ programmer audience less familiar with the technical details of C++ immutables and functional elements, as well as those C++ programmers who are interested in the development of the newest standard. We will survey the functional programming features of modern C++. The various types of constants and immutable memory storage will be discussed as well as the rules of const correctness to enable the static type system to catch const violations. Const and static const members of classes represent support for immutables in object-oriented programming. Specific programming tools, like mutable and const_cast enable the programmer changing constness for exceptional cases. Constexpr and relaxed constexpr (since C++14) objects and functions as well as lambda expressions have recently been added to C++ to extend the language support for functional programming. We also discuss the fundamentals of C++ template metaprogramming, a pure functional paradigm operating at compile time working with immutable objects.

Understanding the immutable elements and the rich set of functional language features with their interactions can help programmers to implement safe, efficient and expressive C++ programs in functional style.

Keywords: C++ · Functional programming · Immutable · Const · Constexpr · Template metaprogramming

V. Zsók et al. (Eds.): CEFP 2015, LNCS 10094, pp. 75–110, 2019.
https://doi.org/10.1007/978-3-030-28346-9_3

1 Introduction

The C++ programming language is a strongly typed, compiled, multiparadigm programming language supporting procedural, object-oriented, generative and (partially) functional programming styles. Its root goes back to the procedural C programming language [17] extending it with essential object-oriented features, like classes, inheritance and runtime polymorphism inherited from Simula67 [38]. Soon generic programming, implemented via templates became fundamental part of the language [8]. The Standard Template Library (STL), part of the standard C++ library, is still the prime example for the generic paradigm [37].

Functional programming features [11] were barely represented in the earlier versions of the C++ language. One of the existing elements was the pointer to function originated in the C language. A pointer to a function allows the programmers to represent algorithms as data; store them in variables, pass them as parameters or returning them from functions. The applications using pointers to functions are restricted in both semantical power and syntactical ease.

Pointers to member functions also exist in C++. As member functions are bound to classes, the type of such a pointer includes information about the class itself. However, the implementation is tricky as the same pointer can point either to a non-virtual member function (represented as an "ordinary" function pointer) or to a virtual function (effectively an offset in the virtual table).

Besides pointers to functions C++ programmers can use *functors* as functional programming elements as predicates, comparisons or other kind of executable code to pass to standard algorithms as parameters. Functors are classes with function call (parenthesis) operator defined, and they behave like higher order functions: they can be instantiated (with constructor parameters if necessary), and can be called via the function call operator. Although functors are not necessarily *pure* as they may have state (separate for each instantiations), they are fundamental tools for implementing functional elements, like currying, binding, etc.

In many cases functors are syntactically interchangeable with pointers to functions, e.g. when passing them as template parameters to STL algorithms.

Despite all the restrictions, syntactical and semantical issues, early attempts were made to implement complex libraries to achieve functional programming in the C++ language [7,18]. Most notable, FC++ [20,21] has implemented lazy lists, Currying and function composition among other functional features. The following works on functional features in C++ also targeted template metaprogramming [32,33].

As functional programming becomes more and more popular – not only as powerful programming languages, but also as formalism to specify semantics [19,47] – further language elements supporting functional style have come into prominence [25]. The *lambda expressions* have been added to C++ first as a library for more direct support for functional programming [15,16]. Lambda expressions provide an easy-to-use definition of unnamed function objects – closures [14]. To eliminate the shortcomings of a library-based implementation, the C++11 standard introduced native lambda support. Lambda functions are directly translated to C++ functors, where the function call operator is defined

as const member function. C++14 further enhanced the usability of lambda captures with *generalized* lambdas, and with the possibility of using initialization expressions in the capture [13].

Despite all these achievements towards programming in functional style, C++ is not and never will be a (pure) functional programming language. As it is neither a pure object-oriented nor a procedural language. C++ is essentially a *multiparadigm* programming language: it does not prefer a single programming style over the others. One can write (a part of) a C++ program using the set of features from one or more paradigms according the problem to solve [5]. Moreover, these paradigms are necessary collaborating to each other. STL containers and algorithms with iterators form a generic library. In the same time, containers are implemented as (templated) classes with features of object-oriented programming, e.g. separation of interface and implementation, public methods and operators, etc. (Virtual functions, however, are mostly avoided for efficiency reasons except in some i/o related details.) On the lower end of the abstraction hierarchy, member functions are implemented in a procedural way.

Functional programming interweaves this hierarchy. STL algorithms are often parameterized with lambdas or functors representing predicates and comparators. Although no language rule forbids them having states, most experts suggest to implement predicates and comparators as pure functions since algorithms may copy them.

The STL itself suggests a functional approach: instead of writing loops and conditional statements, the programmer is encouraged to use algorithms, like `std::for_each` and `std::remove_if`. When the highly composable *ranges* will be incorporated to C++17 or later [28] they will add an additional support to this style. As explained in [26], the range comprehensions are in fact, monads.

The design goals of C++ according to Stroustrup include type safety, resource safety, performance, predictability, readability and the ease of learning the language [40]. These goals were achieved in different language versions to different extents.

Resource safety can be achieved only by the thoughtful process of the programmer using the appropriate language elements. C++ is not automatically garbage collected. (Although the standard allows using garbage collection, everyday C++ implementations usually avoid it). The programmer, however, can use smart pointers, like `std::unique_ptr` or `std::shared_ptr` (part of the standard since C++11) to control heap allocated memory. What is different in C++ from other usual object-oriented languages is that the notion of *resource* is far more general than just memory: every user defined feature can be handled as resource and controlled by the *Resource Acquisition Is Initialization* (RAII) method: resources can be allocated by constructors and will be (and usually shall be) disposed by a destructor. Special care should be paid to copying objects (either by copy over existing objects or initializing newly created ones) using assignment operators and copy contructors.

One of the major distinctive feature of C++ is the ability to map large variety of semantic concepts to compiler checked syntactical notations. Let us investigate

the code snippet written in ANSI C on Listing 1. The program crashes for obvious reasons: we opened `input.txt` for read-only, but later we try to write into it. In the C language, the concept whether a file has been opened for read-only, write-only or read-write is not mapped to the language syntax at all; therefore, the compiler is unable to check the correctness of the usage (and it does not even attempt to do it). If we fail to use the files properly, we won't get diagnostic messages (like warnings) about it, our program compiles and (likely) crashes at runtime.

```
#include <stdio.h>

int main()   // wrong C program
{
    FILE *fp = fopen( "input.txt", "r");
    // ...
    fprintf( fp, "%s\n", "Hello input!");
    // ...
    fclose(fp);
}

$ gcc -std=c99 -pedantic -Wall -W wrong.c
$ ./a.out
Segmentation violation
```

Listing 1: Erroneous usage of C style input/output

In the C++ standard library, however, there are separate (abstract) types for read-only and write-only streams. Read-write streams in fact are inherited from both bases. Real resources (e.g. files and *stringstreams*, i.e. streams working over in-memory character strings) are represented by objects belonging to derived classes inherited from either the input or the output base classes.

As input and output operations are defined in the respective base classes, improper usage of streams cause compile time errors, as seen on Listing 2. Although, the diagnostic message caused by the improper usage is a bit diffuse, the first lines point to the exact problem. Again, the essence of the solution was that the library mapped the concepts of opening a stream either for reading or writing into the appropriate C++ types and, thus, the compiler is able to detect the mismatch as type error.

The concept of *immutability* is handled in a very similar way. The type of a (mutable) variable of type `X` is different from the type of an immutable variable of the same type `X`. Although, such distinction is not unusual in other programming languages, the generality and completeness of the mapping are what makes C++ solution special. We are speaking about a language, where multiple aliases of the same memory area (in form of *pointers*, *references*) are usual, and where objects of user defined classes are expected to behave the very same way as built-in types do.

```
#include <fstream>

int main()
{
    std::ifstream f;
    // ...
    f << "Hello input!" << std::endl;
}

$ g++ -std=c++11 -pedantic -Wall -W w.cpp

w.cpp: In function ‘int main()’:
w.cpp:10:8: error: no match for ‘operator<<’ in ‘f << "Hello input!"’

w.cpp:10:8: note: candidates are:
/usr/include/c++/4.6/ostream:581:5: note: template<class _CharT,
  class _Traits, class _Tp> std::basic_ostream<_CharT, _Traits>&
  std::operator<<(std::basic_ostream<_CharT,_Traits>&&,const _Tp&)
/usr/include/c++/4.6/ostream:528:5: note: template<class _Traits>
  std::basic_ostream<char, _Traits>& std::operator<<(
    std::basic_ostream<char, _Traits>&, const unsigned char*)
...
```

Listing 2: C++ can detect erroneous input/output usage at compile time

This paper is organized as follows. In Sect. 2 we survey the various ways we can define immutable objects in C++. We analyse *const correctness*, the set of complex rules allowing C++ to catch constant violations at compile time in Sect. 3. Here we will also discuss how constness works in STL and how the programmer can make exceptions of constness via const_cast or mutables. The constexpr objects and functions became official part of C++ since version C++11, and were substantially extended in C++14. We overview their possibilities in Sect. 4. Lambdas, their usage and whether and how they are pure functions are explained in Sect. 5. Template metaprograms discussed in Sect. 6 are pure functional language elements forming a compile time Turing complete sublanguage of C++. All such elements are immutable of necessity. The paper concludes in Sect. 7.

2 Immutable Elements in C++

In this section we first survey the elementary immutable objects in C++. Then we will see how we can express immutability for more complex constructs.

2.1 Preprocessor Macros

The C++ preprocessor runs as the first step of the compiler. The preprocessor executes trigraph replacement, line splicing, tokenization, comment replacement,

and finally: macro expansion and directive handling. In this last step, identifiers defined as macros are replaced by their defined values. Naturally, many of such macros are defined as literals, and; therefore, they are immutable as we will see in the next subsection.

2.2 String Literals

String literals are sequences of characters surrounded by double quotes (and optionally prefixed since C++11). By the C++11 standard [12], a string literal is an *lvalue*, a notion which are usually connected to modifiable memory location. String literals however are strictly *read-only* character arrays with static lifetime. In many environments, the compiler may place such literals to read-only storage. Any attempt to modify such string literals results in *undefined behaviour*. Moreover, compilers are allowed to re-use the storage of string literals for other equal or overlapping literals. Most of the modern compilers do this, but remember: this is not mandatory, actual compilers may choose different implementations.

```
#include <iostream>

int main()
{
    char *hello1 = "Hello"; // hello1 points to 'H' and hello2
    char *hello2 = "Hello"; // likely points to the same place

    std::cout << static_cast<void*>(hello1) << "\t"
              << static_cast<void*>(hello2) << std::endl;

    *hello1[1] = 'a';   // could cause runtime error
}

$ g++ -std=c++14 -pedantic -Wall -W string1.cpp
In function ‘int main()’:
 warning: deprecated conversion from string constant to ‘char*’
 [-Wwrite-strings]
     char *hello1 = "Hello"; // hello1 points to 'H' and hello2
                  ^
 warning: deprecated conversion from string constant to ‘char*’
 [-Wwrite-strings]
     char *hello2 = "Hello"; // likely points to the same place
                  ^
$ ./a.out
0x4009f5    0x4009f5
Segmentation fault (core dumped)
```

Listing 3: String literals are immutable objects

Both string literals in the code example in Listing 3 have type `const char[6]` (one extra character is allocated for the terminating zero character of the string) which is converted to a *pointer to const* – `const char *`. We will see in Sect. 3.1 that normally pointers to const are not converted to pointers to non-consts. Here the exceptional rule is explained by the reverse compatibility requirement with millions of lines of legacy C code, where such assignments were legal and frequent [9]. At least the compiler warns us that this usage is deprecated in modern C++.

String literals must not be confused with named character arrays, which can be initialized by string literals and all their elements are mutable (unless they are declared as constant arrays). In Listing 4 `arr1` and `arr2` are two separate mutable character arrays of type `char[6]` placed strictly into different memory areas. The arrays are initialized by the same sequence of characters. The second notion is just a simplification to denote character array initialization list.

```
#include <iostream>

int main()
{
    char arr1[] = {'H','e','l','l','o','\0'};
    char arr2[] = "Hello";
    const char arr3[] = "Hello forever";
    arr1[1] = 'a';   // ok, arr1 is mutable
}
```

Listing 4: Character arrays are mutable by default

Naturally both `arr1` and `arr2` are mutable, so we can modify any of the array elements (although setting/removing the zero character may confuse some string-related standard library functions). The `arr3` is declared as `const`, therefore, that array is immutable.

2.3 Named Constants

A *const object* is an object of type `const T` or a non-mutable subobject of such an object. Such named const objects have some less known properties. They may appear, for example, as `case` labels in `switch` statements, and they may serve as the size of static arrays.

In Listing 5, `c1` and `c2` are initialized with *constant expressions* (constants whose values can be computed by the compiler at compile time). Such constant objects can serve as case labels or as the size in an array declaration. (Since C++11 *variable sized arrays* are allowed, but only for objects with automatic life time, like non-static local variables.) Object `c3`, however, must not be used, as it is initialized with a runtime value coming from a non-constexpr function `f` as a return value. In all other aspects `c1`, `c2`, and `c3` have the same behaviour. Similar situations with *constexpr* function will be discussed in Sect. 4.

```
int f(int i) { return i; }  // not constexpr

int main()
{
    const int c1 = 1;    // initialized at compile time, optimized out
    const int c2 = 2;    // initialized at compile time, but needs memory
    const int *p = &c2;  // ...since a pointer points to it
    const int c3 = f(3); // f() is initialized at runtime, needs memory

    static int t1[c1];
    static int t2[c2];

    int i;
    // read i
    switch(i)
    {
    case c1: std::cout << "c1"; break;
    case c2: std::cout << "c2"; break;
    // case label c3 does not reduce to an integer constant
    // case c3: std::cout << "c1"; break;
    }
}
```

Listing 5: Named constants

The compiler tries to optimize the const objects. Here the compiler may decide not to allocate memory for object `c1`, but should allocate memory for `c2` and `c3`. The reason, why `c3` requires memory is obvious: it is initialized at runtime, therefore the compiler cannot replace all of its occurrences with its value. The situation with object `c2` is a bit more interesting: its address is used to initialize a pointer (a pointer to const). Since pointers must point to legal memory locations by the C++ standard, the compiler must store `c2` in such a location.

2.4 Static Const Members

In object-oriented programming, we try to organize our code in classes. Objects are instantiations of such classes. The names of types, methods and data belonging to a certain class are expressed as members of that class. For immutables this is the same.

Immutable values, which are the same for all objects of a given class, are declared as `static const` in the class declaration. We may think of them as global constants nested into the namespace of the class. Static members of a class have a *static* lifetime, i.e. they are initialized before the start of the `main` function and destructed after the `main` function successfully finished. For elementary types this means that the initialization is done by the compiler. For those

types having non-trivial constructor functions (like most of the standard containers) the constructor is called before starting the `main` function.

Static members are not subobjects of their class. Similar to other static members, static constants should also be defined to tell the compiler which source is responsible to store the static member. In this definition, static consts should be initialized either implicitly, having a default constructor or explicitly with an initializer expression. For *integral* and *enumeration* types (like `bool`, `char`, or `long` the initialization can be placed inside class definition. According the *One Definition Rule* (ODR) initialization should happen in exactly one of the places as seen on Listing 6.

```
class X
{
    static const int  c1 = 7; // ok, but remember definition
    // static const int  c2 = f(2); // error: not const expression
    static const double c3; // not integral, don't initialize here
};
void f()
{
  const int X::c1;           // must not re-initialized
  const double X::c3 = 3.14; // not integral, must initialized here
}
```

Listing 6: Static const members allocated outside of objects

Static consts are immutables with the same value for all objects of the class. There are situations where a member should be immutable for the lifetime of the object but having a different value for different objects.

(Non-static) const members represent such subobjects. Unlike their static counterparts, such constant members are *subobjects*, i.e. they are part of the object. In Listing 7, each object of class Y has its own (possibly different) immutable value, initialized by the constructor.

```
class Y
{
    Y() : id1( gen_id1() ) { }
    const int id1;
    const int id2 = gen_id2();// since C++11 same as Y():id2(gen_id()) { }
    int gen_id1() { ... }
    int gen_id2() { ... }
};
```

Listing 7: Non-static const members are allocated inside every instance

Object-level constants are immutable, but their values can be different in different objects. Hence, they must be stored in every object. Since C++11 we can initialize non-static data members – the meaning is the same as the use of the initializer list in the constructor as seen in Listing 7.

3 Const Correctness

In the previous chapter we had a survey of the immutable C++ objects. However, in C++ objects may be accessed via aliases: pointers, references. To catch possible violations of constness at compile time C++ provides a complex set of *const-correctness* rules. In the following we enumerate these rules.

3.1 Non-class Types

While C++ objects are mutable by default (with the exceptions we discussed in the previous section), const qualified objects are immutable. Any attempt to write to them causes compile time error. The const qualifier is part of the object's type: hence on Listing 8 the type of `ci` is `const int`.

```
        int  i = 4;  // not const
             i = 5;  // i is mutable
  const int ci = 6;  // const
            ci = 7;  // error: ci is immutable
```

Listing 8: Mutable and immutable variables

In C++ we use the *address-of* (`&`) operator to create a pointer value pointing to an object. We can access and modify mutable objects via pointers. The problem is that in most cases it is impossible to determine at compile time where a pointer points to at runtime. If ordinary pointers to type `T` could point to objects of type `const T`, then it would introduce a Trojan horse to modify immutable objects as we see on Listing 9.

```
int   i = 4;   // non const
int *ip = &i;
    *ip = 5;   // ok
const int ci = 6;   // const

if ( runtime_value )
{
    ip = &ci; // ???
}
*ip = 7;   // where does ip point now?
```

Listing 9: Const-correctness

In order to avoid this issue, C++ forbids the assignment of pointers to const-qualified objects to ordinary pointers. We can say that the type of `const T*` is not convertible to `T*`, as we see on Listing 10.

```
int    i = 4;
int  *ip = i;
const int ci = 6;  // const

if ( runtime_value )
{
    // this would be compile error:
    // ip = &ci;
}
*ip = 7;   // ok, ip points to mutable object
```

Listing 10: Constness must not be lost in assignment

Since pointers are fundamental in various situations in C++ including many use cases of the standard library, the language provides a way to set pointers to immutable objects. An object with type of pointer to `const T` can store a pointer value to an immutable object of type `T`, it can be dereferenced, but the dereferenced value is immutable. To keep const correctness, pointers to const values cannot be assigned to pointers to mutable objects. We must not "lose" constness.

This rule is not symmetric: we can still assign addresses of mutable objects to pointers to const variables. In that case, the pointed (originally mutable) object is handled as immutable when accessed via the pointer to const, see Listing 11.

```
int  i = 4;         // mutable
const int ci = 6;  // const

int *ip = &i;           // ok
const int *cip = &ci;  // ok

 ip  = cip;         // compile error: T* <- const T*
 cip = ip;          // ok:      const T* <- T*
*cip = 7;           // compile error: *cip is immutable
```

Listing 11: Conversion rules between pointers

There are different conventions to define a pointer to const. Some programmers prefer to use the `const T *` order, while others emphasize the immutability of the pointed location to move the `const` keyword between the type name and the * declarator in form of `T const *`. Both versions are supported by the literature and have the same meaning as long as we use the `const` keyword on the left side of the * declarator as on Listing 12.

```
const int *cip1;      // pointer to immutable
int const *cip2;      // cip2 has the same type as cip1's
int *const ptr1 = &i; // ptr1 is immutable
const int *ptr2;      // ptr2 points to immutable
const int *const ptr3 = &i;
```

Listing 12: Immutable pointers and pointers to immutables

To declare a pointer itself as immutable (pointing either to a mutable or an immutable object) we use the `const` keyword on the right side of the * declarator. In this way, in the next example we declare `ptr1`, `ptr2` and `ptr3` as a const pointer to mutable object, a non-const pointer to immutable object and a const pointer to immutable object, respectively.

Note, that const (immutable) objects (like `ptr1` and `ptr3`) must be initialized when defined.

3.2 Constness of Class Types

The rules discussed in the previous Subsect. 3.1 create a set of compile time guidelines to ensure that objects defined as consts won't be modified during runtime. However, these rules are not yet complete when we consider class types.

Consider the `Date` class on Listing 13 which encapsulates three `int` data members and provides related access methods representing a (very simplified) data class.

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // ...
    int getYear();
    int getMonth();
    int getDay();

    void set(int y, int m, int d);
    // ...
private:
    int year;
    int month;
    int day;
};
```

Listing 13: The original Date class

The rules discussed in Subsect. 3.1 stand here too. Any attempt to modify a const object causes a compile error on Listing 14. The major difference between elementary types and classes is that elementary objects can be modified only by assignment (via the objects themselves, or via references or pointers denoting them), classes, however, can be also modified by member functions. How should the compiler handle the method calls?

```
void f()
{
  const Date my_birthday(1963,11,11);
        Date curr_date(2015,7,10);
  // my_birthday =  curr_date;   // compile error: my_birthday is const
  cout << myBirthday.getYear();  // read const
  myBirthday.set(2015,7,10);     // modify const?
  cout << currDate.getYear();    // read non-const
  currDate.set(2015,7,11);       // modify non-const
}
```

Listing 14: Access const and non-const objects via member functions

It is obvious, that `myBirthday.set(2015,7,10)` violates the seemingly complete set of const correctness rules. But how can the compiler make difference between methods allowed and forbidden for immutable objects?

The naïve approach to check the body of the methods fails for various reasons; the definition of the method can be in a different source file, the method can call other methods, etc. C++ has chosen a more syntax-driven approach: we should explicitly mark methods callable on immutable objects as `const` methods as part of their signature. Non-const methods cannot be applied to const objects regardless whether they attempt to modify their objects or not.

In the code snippet on Listing 15, the compiler allows the call of `getYear` on the immutable object `myBirthday`, since it is declared as a const method, but emits diagnostics for the call of `set`.

The `this` parameter, passed as the hidden first argument for all non-static methods is used to check the call. The `this` parameter of a const method is declared as pointer to const, whereas in a non-const method it is a pure pointer to the class. As the address of an immutable object is obviously a *pointer to const*, such an address can be passed to const methods. Otherwise, const methods can not be applied to non-const objects based on the required const to non-const conversion. On the other hand, since a non-const `this` argument can be converted to pointer to const, const methods are callable on non-immutable objects.

On the other hand, the compiler must not "trust" on whether the programmer denoted the constness in the correct way for the const methods. Here the situation is also covered by the rules we learned in Subsect. 3.1. Data members

```
class Date
{
public:
    Date( int year, int month = 1, int day = 1);
    // ...
    int getYear() const;
    int getMonth() const;
    int getDay() const;

    void set(int y, int m, int d);
    // ...
private:
    int year;
    int month;
    int day;
};
void f()
{
  const Date my_birthday(1963,11,11);  // immutable
        Date curr_date(2015,7,10);     // mutable
  // my_birthday =  curr_date;    // compile error: my_birthday is const
  cout << myBirthday.getYear();   // fine: const member on const
  // myBirthday.set(2015,7,10);   // error: non-const member on const
  cout << currDate.getYear();     // fine: const member on mutable
  currDate.set(2015,7,11);        // fine: non-const member on mutable
}
```

Listing 15: Const and non-const member functions

in member functions are accessed via the `this` pointer. As the `this` parameter for such methods is implicitly declared as pointer to const, all modifications via `this` are marked as errors by the compiler. Similarly, the calls of non-const methods from const methods yield errors.

The invisible `this` parameter can also be used for overloading. This pattern on Listing 16 is frequently used for access operators, which should be used both for reading const objects and for modifying mutables.

```
      T& operator[](size_t idx);
const T& operator[](size_t idx) const;
```

Listing 16: Overloading on constness

Only non-static member functions can be declared as const methods. Static member functions and namespace functions should be declared as `constexpr` to express their "pure" behaviour. We will discuss constexpr in Sect. 4.

3.3 Mutable

There are certain situations where we want to modify subobjects inside const objects. Consider the `Point` class on Listing 17 with objects accessed concurrently from multiple threads. To read consistent `x` and `y` pairs of coordinates, the method `getXY(int &x, int &y)` locks the object. For this purpose, we place a `mutex` object as a class member. Naturally, `getXY` is declared as a `const` method as it is just reading the object.

```
struct Point
{
    void getXY(int& x, int& y) const;

    double xCoord;
    double yCoord;
    std::mutex  m;
};
void Point::getXY(int& x, int& y) const   // does not compile
{
    std::lock_guard<std::mutex>  guard(m); // constructor locks m
    x = xCoord;
    y = yCoord;
}                                         // destructor unlocks m
```

Listing 17: Reading x and y coordinates is protected by a mutex

Unfortunately, the code above does not compile. Locking and unlocking obviously are changing the state of the mutex object, i.e. they are non-const methods. As we learned in Sect. 3.2, we normally cannot alter an object's (or its particular subobject's) state (like the mutex `m` in our example) using a const member function.

To make an exception from the rule, we can declare the mutex `m` as `mutable`. Mutable means that the (sub)object can be altered even when it is part of a const object or when accessed from a const member function.

```
struct Point
{
    void getXY(int& x, int& y) const;

    double xCoord;
    double yCoord;
    mutable std::mutex  m;
};
```

Listing 18: A class with a member declared as mutable

Mutables are exceptional objects, and we should use them only in exceptional cases. Such situations include among others managing internal states, like a cache or counters, and also managing mutexes, like in our example on Listing 18.

3.4 Constant Correctness in STL

The Standard Template Library (STL) is an essential part of the C++ language. The success of the STL is based on its flexibility and extensibility. Programmers can re-use standard containers and algorithms connecting them by iterators instead of writing specific code for each individual problem [4,27]. STL makes the programmer's work faster, safer and more predictable [29]. Naturally, the STL should support the const correct programming [22].

Consider the usual implementation of the `find` STL algorithm on Listing 19.

```
template <typename It, typename T>
It find( It begin, It end, const T& t)
{
    while (begin != end)
    {
        if ( *begin == t )
        {
            return begin;
        }
        ++begin;
    }
    return end;
}
```

Listing 19: Canonical implementation of the STL find algorithm

When we apply the algorithm to an immutable array, e.g. to a const array of integers on Listing 20, the iterator will be deduced to the same type as the first two parameters, i.e. to a pointer to const. Thus, const correctness stands.

For STL containers, the situation is a bit different as instead of pointers *iterators* and *const_iterators* are used to walk through container elements and to refer to them. Dereferencing (applying the star operator for) an `iterator` results in a left value reference to the `value_type` of the container. Dereferencing a `const_iterator` results in a non-writable *const reference*. The type `const_iterator` is not a `const iterator` which would mean an immutable iterator. However, a `const_iterator`, is a mutable object, e.g. one can modify it and can walk through the container, but the referred objects are handled as immutables as seen on Listing 21.

The `iterator` and `const_iterator` values are generated by the `begin` and `end` methods of the containers. Since C++11 the `std` namespace `begin` and `end` functions are also available. Namespace functions provide the iterators in a

```
const int t[] = { 1, 2, 3, 4, 5 };
auto len = sizeof(t)/sizeof(t[0]);
auto *p = std::find( t, t+len, 3)  // const int *p

if ( p != t+len )
{
    std::cout << *p; // ok to read
    // *p = 6;       // error to write
}
```

Listing 20: Const correctness in STL

```
void f(const std::vector<int> &v)
{
    auto i = std::find( v.begin(), v.end(), 3); // const_iterator
    if ( v.end() != i )
    {
        std::cout << *i; // ok to read
        // error: *i = 6;
    }
}
```

Listing 21: Const containers provide read only access to elements

unique and non-intrusive way for STL containers and classical C-style arrays. The trick here is the overloading on constness: const variations of the `begin` and `end` methods are the methods callable on constant containers and they return `const_iterator`.

C++11 provides more advanced type deduction with the `auto` keyword. Its motivation was mainly to shorten iterator and const_iterator declarations. However, it is a frequent situation when we want to apply a `const_iterator` to an originally non constant container. This will not work with `auto` and `begin` or `end` methods, as the auto declaration deduces the type from the initializer expression, which is the return value of the `begin` and `end` methods, i.e. an `iterator` in case of a non-const container [23].

```
void f(std::vector<int> &v, const std::vector<int> &cv)
{
    auto i = std::find(  v.begin() , v.end(), 3); //iterator
    auto i = std::find( cv.begin(), cv.end(), 4); //const_iterator
    auto i = std::find( v.cbegin(), v.cend(), 5); //const_iterator
}
```

Listing 22: Using `cbegin` and `cend` to return `const_iterator`

To enforce the return of `const_iterator` type even for non-const containers, new methods `cbegin` and `cend` were introduced in the C++11 standard, see Listing 22. Similar namespace methods exist as `cbegin`, `cend`, and `crbegin`, and `crend` for iterators and reverse iterators. For some mysterious reason, namespace functions returning `const_reverse_iterator` were missing in C++11, and the issue has been fixed only in C++14.

3.5 Casting Const Away

Based on the C/C++ philosophy that the ultimate control belongs to the programmer, there is an explicit cast to converting const objects to non-const ones. However, as all other cast operations, it should be used with extra care to avoid *undefined behaviour*. The basic rule is that we can cast away constness of pointers to objects and lvalues.

Even when the compiler allows us to cast constness away, the result may be surprising. In the example on Listing 23, we declare a variable const, then we modify it via `const_cast` and a non-const pointer.

```
#include <iostream>

int main()
{
  const int ci = 10;

  int *ip = const_cast<int *>(&ci);
  ++*ip;
  std::cout << ci << " " << *ip << std::endl;
}
$ g++ -std=c++11 -pedantic -Wall -W const.cpp
$ ./a.out
10 11
```

Listing 23: Const cast may lead to undefined behaviour

There are a few cases when we cannot avoid the use of `const_cast`. One example is when a member function modifying the object's state should be defined as a constant member function. Suppose, we have a tree data structure to store elements ordered by some key values. We might provide a member function to balance the tree. Calling such `balance` member function from `insert`, which is a non-const member function itself, is fine. However, when we want `balance` to be available as a standalone API method, we should decide about its constness. Strictly by C++ terms, `balance` is not a constant method as it is modifying the object's data. However, from the viewpoint of the user, the method behaves like a const: no new element has been inserted, the order of the elements remains the same. It is tempting to declare `balance` as const member function and simply cast the constness of the `this` pointer away to make class members modifiable.

Although, the scenario above is possible in technical terms, there are strong arguments against it. Using `const_cast` is extremely dangerous for objects that were originally declared as `const` (see Listing 23). Whenever it is possible, use `mutable` objects instead. Declaring a member function as `const` tells the user that this function can be used in a multithreaded environment in a safe way. That is not true in the example above (except that we use some very aggressive locking inside `balance`, which may ruin the performance).

4 Constexpr

Compile-time expressions, i.e. expressions that can be evaluated at translation time, always had a specific role in the C++ language. Such expressions can be used to define an array size, a `case` label or a non-type template argument. On the other hand, the compiler environment is enforced to compute these expressions; C++ template metaprogramming is largely based on this fact. The phrase *translation time* usually means compile time but may include link time activity as well [36].

Interesting enough, by the C++ standard, the value of an expression computed at translation time is not necessary equal to the value of the same expression computed at runtime [50]. Let us see the example on Listing 24 quoted from the standard. The size of the character array must be compiled at translation time, but the value of the integer variable *size* can be evaluated at runtime. In such cases their values may differ.

```
bool f()
{
    // Must be evaluated at translation time
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)];

    // May be evaluated at run-time
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1);

    return sizeof(array) == size; // unspecified: true or false
}
```

Listing 24: Compile-time expressions and run-time expressions

In classical C++03 constant expressions are restricted to the use of literals, built-in operators, and macros. They must not contain functions or operators of any kind, even when their return value could be trivially computed from the compile-time given arguments as we see on Listing 25.

C++11 makes constant expressions more manageable introducing `constexpr` functions and expressions. The idea is to make translation time constant computation more expressive and thus partially replacing unmanageable macro and template metaprogramming elements.

```
#define AVERAGE(X,Y)  (((x)+(y))/2)
double average(double x, double y) { return (x+y)/2; }

size_t s1 = sizeof(long);
size_t s2 = sizeof(short);

// constant expressions in C++03
const int a = (sizeof(long)+sizeof(short))/2;
const int b = AVERAGE(sizeof(long),sizeof(short));

// not constant expressions in C++03
const int c = AVERAGE(s1,s2);
const int d = average(sizeof(long),sizeof(short));
```

Listing 25: Constant and non-constant expressions in C++03

4.1 Constexpr Functions

C++11 introduced constexpr functions, functions that can be computed at translation time when all their parameters are known. As initially this feature was planned as a minor feature to replace hard to maintain macros and small template metaprograms, it had a minimalist design. The body of a constexpr function was restricted to a single *return* statement. Constexpr member functions also were implicitly constant member functions.

As the new feature was a success, constexpr rules have been relaxed. Constexpr functions since C++14 may contain declarations, sequences, and control statements similar to "normal" functions as demonstrated on Listing 26. The rules also have been changed in the way that non-static constexpr member functions are not const member functions any more.

```
// in C++11
constexpr int pow( int base, int exp) noexcept
{
    return  exp == 0 ? 1 : base * pow(base, exp-1) ;
}

// in C++14
constexpr int pow( int base, int exp) noexcept
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Listing 26: Constexpr in C++11 and in C++14

Even in C++14 there are serious restrictions on constexpr functions. They must not contain `asm` definitions, `goto` statements, `try` blocks, and must not declare static or thread local variables. Member constexpr functions must not be virtual. These rules ensure compile time computability. Other C++ rules, e.g overloading, work as usual.

Rules for constexpr functions are also designed to avoid side effects as we experience on Listing 27. The safe rule is to access only variables which have life time started inside the `constexpr` expression.

```
constexpr int f(int n)
{
    static int value = n;  // error, cause side effect
    int i = 1;
    int j = n;             // ok, j is not constexpr
    constexpr int x = n;   // error
    constexpr int y = i;   // ok, life of i starts in f
    return y;
}
```

Listing 27: Constexpr rules are to avoid side effects

Sometimes there is a thin line between what can be constexpr and what can not. On the Listing 28 a ternary operator defines the value of member `m`. When the constructor parameter is true, the value of `m` can be computed at translation time. Otherwise, the initialization depends on a run-time value, so the compiler flags an error.

```
int x;                          // not constant
struct A
{
    constexpr A(bool b) : m( b ? 42 : x) { }
    int m;
};
constexpr int v = A(true).m;   // OK: constructor initializes m with 42
constexpr int w = A(false).m;  // error: initializer of m is x
                               // which is not known at translation time
```

Listing 28: Expression computed at translation time or flags an error depending on the value of `b`

One of the advantages of constexpr functions over template metaprograms is that template metaprograms can only emulate floating point numbers, usually with a pair of integer, while constexpr functions can work with native floating point types.

4.2 Constexpr Objects

Constexpr objects are constant objects having values that are known (or computable) at translation time. We can apply the `constexpr` keyword both for variables and for variable templates as seen on Listing 29.

```
constexpr size_t sizeof_long  = sizeof(long);
constexpr size_t sizeof_short = sizeof(short);
template <typename T>
T PI()
{
  constexpr T Pi = T(3.1415926535897932385);
  return Pi;
}
```

Listing 29: Constexpr objects

Not only objects of built-in types can be specified as constexpr objects, but also objects from those class types which can be safely constructed at translation time. Such types are called *literal types*. Literal types must not include any component which would indicate runtime activity, e.g. a virtual base. Scalar types, reference types, the `void`, and arrays of literal types are considered as literal types. Also classes with non-static members of (non-volatile) literal types, `constexpr` member functions, `constexpr` constructor and with trivial destructor are literal types [49].

On the Listing 30 we created a literal type representing a circle with a given radius set by the constructor, inspired by [3]. Member functions are defined to compute the perimeter and the area of the object as well as a non-const member function `magnify` to change the radius by a ratio. The namespace function `create` returns a new circle created by the first argument applying the second argument as magnifying ratio.

In the example we created a literal type `Circle` with a single attribute `radius` initialized by the only constructor. The constexpr getter methods `perim` and `area` are declared `const` since in C++14 non-static constexpr member functions are no longer implicit const member functions.

The `magnify` function is a constexpr but non-const member, as it changes the object's attribute value. Obviously, such a member cannot be applied to a constexpr object, but can be used for non-const objects, like the local `c2` object inside the `create` namespace function. Declaring `magnify` as `constexpr` guarantees, that all the constexpr restrictions are hold, therefore, the function can be safely called from the constexpr function `create`.

The local variable `c2` of `Circle` type is not defined constexpr inside the `create` function – we will modify it in the next line. We are allowed to create non-const local variables in constexpr functions. The important aspect here is that the lifetime of `c2` starts inside the constexpr function.

```
constexpr double sqr(double d) { return d*d; }
constexpr double Pi = 3.1415926535897932385;

class Circle  // literal type
{
public:
    constexpr Circle( double r) noexcept : radius(r) { }
    constexpr double perim() const noexcept { return 2*radius*Pi; }
    constexpr double area() const noexcept { return sqr(radius)*Pi; }
    constexpr void magnify(double ratio) noexcept { radius *= ratio; }
private:
    double radius;
};
constexpr Circle create( const Circle &c, double ratio) noexcept
{
    Circle c2 = c;
    c2.magnify(ratio);
    return c2;
}
int main()
{
    constexpr Circle c(2.5);
    constexpr double p = c.perim();
    constexpr double a = c.area();
    constexpr Circle c2 = create(c,1.5);
}
```

Listing 30: A Circle class implemented as a literal type

In the `main` function all objects can be defined as constexpr. Such objects can be placed into ROM if the environment supports that. The constructors of these objects will run at translation time.

Constexpr functions and methods can be called with non-constexpr arguments. In such situations, they will be executed at run-time. Allowing to call constexpr functions at run-time avoid code duplication. However, the restrictions for these functions as described in this section still hold.

Constexpr functions are "running" at translation time, therefore, their inspection is extremely hard. The usual method is to inject a runtime argument, and debug the function at runtime. Proper constexpr debuggers are yet to be implemented.

5 Lambda Expressions

The Standard Template Library (STL) is a major component of the standard C++ library. In STL *containers* implement various data structures, and *algorithms* present numerous activities over them in form of namespace functions.

To provide a smooth, generic connection between these two components, algorithms access the containers via *iterators* [4,27].

STL supports functional style programming as it replaces the necessity of the iteration over containers with the use of predefined algorithms, like `remove_if` or `for_each`. Such algorithms are frequently parameterized by some *callable* objects. The predicate in `remove_if` and the repeated activity for `for_each` are provided as a callable parameter for these algorithms. In its most primitive form, such a callable object is a pointer to function.

However, these functions often require access to the local variables in the scope of the algorithms, e.g. the predicate for the `remove_if` may depend on the values of local variables in the call site. Ordinary C++ functions have no access to the context of the call site. To bridge the problem, a *functor* – a class with function call operator – can be defined and used instead of the function pointer. Function objects (instances of functor classes) can be created and the mentioned local variables are either copied into or referenced by its attributes. These objects are passed to the STL algorithms as parameters and can be called inside the algorithms. The procedure, however, requires a significant amount of boiler-plate code as we see on Listing 31.

```
struct BetweenFunctor
{
public:
    BetweenFunctor(int a, int b) : m_a(a), m_b(b) { }
    bool operator()(int n) const { return m_a < n && n < m_b; }
private:
    int m_a;
    int m_b;
};
void filter(vector<int>& v, int x, int y)
{
    v.erase( remove_if(v.begin(),v.end(),BetweenFunctor(x,y)),
             v.end());
}
```

Listing 31: Removing elements from a container using a functor

The idea to provide an easy-to-use definition of unnamed function objects – so called *closures* – which are capable of accessing (capturing) the variables in the call context led to the notion of *lambda expressions*. Lambdas have been introduced to C++ first as a user library in Boost.Lambda [16,48]. From C++11 they are part of the core language. Since then, lambdas are widely used as parameters in the Standard Template Library algorithms, functions executed by `std::thread`, and various other places.

Lambda expressions in C++ can be associated with equivalent functor classes and function objects. The runtime object created from the lambda expression

is called *closure* and is assignable and callable. Its type, the *closure class* is unnamed. Nevertheless, we can refer to it by using the C++11 `decltype` expression. On Listing 32 we see the equivalent program snippet to Listing 31. Here we are using lambda expression to remove elements from the vector. The expressive power of the lambda solution over the functor is well worth observing.

```
void filter(vector<int> &v, int x, int y)
{
    v.erase( remove_if(v.begin(),v.end(),
                       [x,y](int n) { return x < n && n < y; }),
             v.end());
}
```

Listing 32: Removing elements from a container using lambda expression

We can understand the lambda construction by comparing it to the equivalent functor on Listing 31. The lambda expression starts with the `[ ]` *lambda introducer*. The optional *captured variables* `x` and `y` represent the data members of the functor class initialized by the `x` and `y` variables of the calling context respectively. The parameter and the function body of the lambda expression form the function call operator of the functor defined as a constant member. It can contain multiple statements. The return type is automatically deduced by the corresponding C++ language rules. When that type is not suitable, the required return type can be denoted explicitly [15].

5.1 Capture

The major advantage of a lambda over a functor is that the lambda can access the calling context using captured variables. These variables can be captured either by value or by reference. Default capture is by value: that is, the captured variables are copied into the closure object when it is created. As a consequence, further changes of the original variables captured by value are invisible in the lambda expression. We can imagine it as the capture by value creates a "snapshot" of the calling context.

When variables are captured by reference, the closure initializes references to the original storage. The lambda expression thus always sees the actual value of the captured variables. Capturing variables by reference is denoted by the `&` symbol.

On Listing 33, the `filter` function removes all elements from vector `v` which have a value between `x` and `y`. The parameters `x` and `y` of the function are captured by value, while the local variable `cnt` is captured by reference. Therefore, this latter variable can be modified from the lambda expression, and at the end of the function `cnt` contains the number of the elements removed.

```
void filter(vector<int> &v, int x, int y)
{
    int cnt = 0;
    v.erase( remove_if(v.begin(),v.end(),
                       [x,y,&cnt](int n) { if ( x < n && n < y )
                                           { ++cnt; return true; }
                                           else
                                             return false;     }),
             v.end());
}
```

Listing 33: Variables captured by value and by reference

Lambda expressions are equivalent with *constant* function call operators on the closure type. Any attempt from the lambda expression to modify the captured `x` and `y` will result in an error. Interesting enough, the lambda expression is allowed to modify the variables captured by reference. This has the same behaviour as we can experience with traditional classes: constant member functions can make modifications via reference members.

We can allow the modification of the *copies* of the variables captured by value. We indicate non-constness of the lambda function with the `mutable` keyword before the body of the lambda expression. However, the modifications affect only the copies, the original variables remain unchanged.

We can capture multiple values without enlisting them individually. The `[=]` sign means capturing *all* variables by value and `[&]` means capturing all by reference. We can mix values and reference captures, like `[=,&cnt]`. Naturally, when using the `=` or `&` notion, only the variables actually used in the lambda expression will be stored/referred. Global variables or static members are not captured, but can be used as usual.

5.2 Capturing `this` Pointer

The `this` pointer is not captured by default, it should be captured explicitly by value or by using the `[=]` notation. Capturing `this` is a necessary and sufficient requirement to access members of a class. In Listing 34 the lambda inside the `print` member function should capture `this` to access the data member `s`.

```
struct X
{
    int s;
    vector<int> v;
    void print() const {
        for_each(v.begin(), v.end(), [=](int n) { cout << n*s << " "; });
    }
};
```

Listing 34: Capturing `this` pointer

Capturing pointers and particularly capturing `this` can be dangerous. If the pointed memory area is destroyed but the pointer to it still holds in the closure object, calling the lambda can be fatal. In the example on Listing 35 the closure captures the `this` pointer and then it is stored in a `std::function` object. Later it is activated twice: once when the pointed object is still alive, and the second time after the object is destroyed. That second call will likely cause runtime error.

```
std::function<void (int)> f;

struct X
{
    X(int i) : ii(i) {}
    int ii;
    void addLambda()    {
        f = [=](int n) { if (n == ii) cout << n; else cout << ii; };
    }
};
int main()
{
    {
        std::unique_ptr<X> up = std::make_unique<X>(4);
        up->addLambda();
        f(4);  // calls lambda: ok
    } // destroys the X object

    f(4);  // calls lambda: likely aborts!
}
```

Listing 35: Wrong usage of lambda with captured `this` pointer

In C++17, there will be possible to capture the enclosing object by value, instead of capturing the `this` pointer.

5.3 Constant Initialization by Lambda

One of the special use cases of the lambda expressions are the initializations of constant objects. Constants must be initialized and later they must not be assigned to. In some cases, the initialization value heavily depends on the calling context and should be computed by complex calculations. The usual way to do this is to execute the necessary computations in a separate function and to initialize the constant object by the return value of that function. However, this solution has a number of drawbacks. The code of the function will be separated from the object to be initialized. Using the actual environment of the initialization requires to pass a possible large number of parameters to the function.

It would be somehow useful to handle the variable as non-const for a while, and make it immutable only after we calculated its “final” value. Although, this is not possible literally, we can simulate it by lambda.

```
void f()
{
    bool some_variable_in_context = ...;
    const int ci = [&]{
    int ci;  // non-const shadow variable
    ci = some_default_value;
    if ( some_variable_in_context )   // using the context
    {
        // and do some operations and calculate the value of ci
        ci = some_calculated_value;
    }
    return ci;
    } (); // note: () invokes the lambda!
    // using the const ci
    // ...
}
```

Listing 36: Initialization of a constant using lambda

On Listing 36 we are going to initialize the constant `ci` variable. Instead of initializing it by a function, we define a lambda expression right in the place of initialization capturing the whole context. In this lambda first we define a non-const variable with the same name as the const to be initialized. This "shadow" variable will be used to calculate the initializer value. The body of the lambda expression looks like and acts like the continuation of the original function. Once we calculated the required value, we close the lambda expression and immediately call it, thus, initializing the constant by its return value.

This method is usually more readable and manageable than the alternatives.

5.4 Generic Lambdas

Lambda expressions in C++11 were not generic: i.e. we had to apply various tricks to handle lambdas in templated environment. In C++14, however, we can write *generic* lambda expression, which works in a *polymorphic* way, similarly to a template functor [43]. To express generality we use the `auto` keyword at parameter declaration. Advanced C++14 return type deduction is also applied on the example on the Listing 37.

```
// in C++11
for_each( begin(v), end(v),
             [](const decltype(*begin(v))& x) { cout << x; } );
// in C++14
for_each( begin(v), end(v), [](const auto& x) { cout << x; } );
```

Listing 37: Generic (polymorphic) lambda expressions in C++14

5.5 Generalized Lambda Capture

As we have seen earlier, lambdas can capture variables in the environment either by value or by reference. The first will copy them into the closure object, the second will initialize a reference inside the closure object to the captured variable outside. Capture by reference can be dangerous, especially when the closure object lives longer than the captured variable. In the same time, not all variables can be captured by value. Since C++11 *move-only* types exist: types that cannot be copied only just moved. Objects from such types like `std::unique_ptr`, `std::thread` and many `iostream`-related types cannot be copied, thus we cannot capture them by value as it would apply the copy semantics.

To handle these types in a safe way from lambda expressions we should *move* them into the closure object. For old-style functors, the solution would be trivial, we could *move* the objects into the closure using the initializer list of the constructor. (However, you must not forget to use the `std::move` right-value cast operator.) To provide the same functionality for lambda expressions, C++14 presents generalized lambda capture, or *init capture*. An init capture behaves as if it declares and explicitly captures a variable declared as `auto` and initialized by the initialization expression. However, no real new variable is constructed: e.g. no additional copy and destruction operations will be executed.

```
// since C++14
#include <iostream>
#include <memory>
int main()
{
    int x = 10;
    std::unique_ptr<int> up = make_unique<int>(42);
    [&x = x, up = std::move(up), n = 1] { x = *up+n; } ();
    std::cout << x << std::endl;
}
$ ./a.out
43
```

Listing 38: Init (generic) capture in C++14

In Listing 38 variable `x` is captured by reference, the `unique_ptr` `up` is *moved* into the data member of the closure object and an `int` type data member is created and initialized to 1. The program prints 43. It is also important to notice, that the heap area allocated by `make_unique` and initialized to 42 is already destroyed when we reach the output calls, as its *ownership was moved* from the `up` pointer to the closure's data member which has been destructed at the end of the execution of the lambda expression.

6 C++ Template Metaprogramming

In [30] we explored C++ template metaprogramming as functional programming in a great detail. Thus, in this section we just briefly recap the generic idea and discuss immutability.

Templates are key language elements of C++ enabling algorithms and data structures to be parametrized by types or constants without performance penalties at runtime [39]. This abstraction is essential when using general algorithms, such as finding an element in a data structure, sorting, or defining data structures like vector or set. The generic features of these templates (like the behaviour of the algorithms or the layout of the data structures) are the same, only the actual type parameter is different. The abstraction over the type parameter – often called parametric polymorphism [6] – emphasizes that this variability is supported by compile-type template parameters. Reusable components – containers and algorithms – are implemented in C++ mostly using templates. The Standard Template Library (STL), an essential part of the C++ standard, is the most notable example [22,27].

Templates are code skeletons with placeholders for one or more type parameters. In order to use a template it has to be instantiated. This can be initiated either implicitly, when a template is referred with actual type parameters or explicitly. During instantiation the template parameters are substituted with the actual arguments and new code is generated. Thus, a different code segment is generated when a template is instantiated with different type parameters.

There are certain cases when a template with a specific type parameter requires a special behaviour, that is different from the generic one. Such "exceptions" can be specified using template specializations. During the instantiation of a template the compiler uses the most specialized version of that template.

Templates can refer to other templates (even recursively) thus complex chains of template instantiations can be created. This mechanism enables us to write smart template codes affecting the compilation process. To demonstrate this capability of C++ templates Erwin Unruh wrote a sample program [42]. The program, when compiled, emitted a list of prime numbers as part of the error messages. This way Unruh demonstrated that with cleverly designed templates it is possible to execute a desired algorithm at compile time. This compile-time programming is called *C++ Template Metaprogramming* (TMP) [1].

The classical example on Listing 39 demonstrates how to compute the value of factorial at compile time. We can see that the implementation uses recursion at compile-time. The static constant value, `Factorial<5>::value` is referred to inside the `main` function, thus the compiler is enforced to compute it. The instantiation process of the class `Factorial<5>` begins. Inside the `Factorial` template, `Factorial<N-1>::value` is referred. The compiler now is forced to instantiate `Factorial<4>`, then to instantiate `Factorial<3>`, etc. The `Factorial` template class is instantiated several times recursively. The recursion stops when `Factorial<1>` is referred to, since there is a *specialization* for that argument. At the end, the compiler generates five classes and `Factorial<5>::value` is calculated at compile time.

```
template <int N>
struct Factorial
{
    static const int value = N * Factorial<N-1>::value;
}
template<>
struct Factorial<1>  // specialization
{
    static const int value = 1;
};
int main()
{
    int r = Factorial<5>::value;  // known compile time
    cout << r << endl;
}
```

Listing 39: Simple factorial C++ template metaprogram

Similarly, one can use control branches using template specialization. In the example on Listing 40 example we declare the variable `i` to be of type `int` or `long` depending on whether the size of the `long` type is greater then the size of `int`.

```
template <bool condition, class Then, class Else>
struct if_
{
    typedef Then type;
};
template <class Then, class Else>
struct if_<false, Then, Else>
{
    typedef Else type;
};
int main()
{
    if_< sizeof(int)<sizeof(long), long, int>::type i;
    cout << sizeof(i) << endl;
    return 0;
}
```

Listing 40: (Runtime) conditional choice in template metaprograms

As template metaprograms are "executed" by the compiler, they fundamentally differ from usual runtime programs. Compilers among other actions evaluate constant values, deduce types and declare variables – all of these are immutable actions. Once a constant value has been computed, a type has been

decided, a variable has been declared then they remain the same. There is no such thing as assignment in template metaprograms. In this way C++ template metaprograms are similar to the pure functional programming languages with referential transparency [30]. However, one can still write control-structures, using specializations. Loops are implemented using recursive templates, terminated by specializations. Control branches are based on partial or full specializations.

Having recursion and branching with pattern matching we have a complete programming language – executing programs at compile time. C++ templates have been proven to form a Turing complete sublanguage of C++ at compile time [44]. Template metaprograms are used intensively to implement active libraries [45], expression templates [46], DSL integrations [34], parser generation [41], target of translation of functional programming systems [35] or even for type safe hosting of SQL queries [10].

We can use data structures at compile time. For example the list structure used by most functional programming languages can be implemented by a class, `NullType`, representing the empty list and a template class, `Typelist`, representing the list constructor [2]. One can represent any list by using the constructor recursively. These classes can be implemented and used in Listing 41:

```
class NullType {};
template <class Head, class Tail>
struct Typelist {};

typedef Typelist< char,
                  Typelist<signed char,
                           Typelist<unsigned char, NullType>
                          >
                > Charlist;
```

Listing 41: Representing data in metaprograms

Preprocessor macros make the use of typelists more handy (on Listing 42):

```
#define TYPELIST_1(x) Typelist< x, NullType>
#define TYPELIST_2(x, y) Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z) Typelist< x, TYPELIST_2(y,z)>
#define TYPELIST_4(x, y, z, w) Typelist< x, TYPELIST_3(y,z,w)>
// ...
typedef TYPELIST_3(char, signed char, unsigned char) Charlist;
```

Listing 42: Representing data with typelist at template metaprogramming

The most commonly used data types are implemented by the Boost.MPL library in an efficient way and with an easy to use syntax, without having to use the preprocessor for creating lists. The above list can be created using `boost::mpl::list` as shown in Listing 43.

```
typedef boost::mpl::list<char, signed char, unsigned char> Charlist;
```

Listing 43: Using typelist in boost metaprogramming library

The similarities between template metaprogramming and the functional paradigm are obvious. Static constants have the same role in template metaprograms as ordinary values have in the runtime ones. Template metaprogramming uses symbolic names (typenames, typedefs) instead of variables. Specific classes are used to replace runtime functions.

To bring C++ metaprogramming from an ad-hoc approach to a more structured form, Czarnecki and Eisenecker defined the term template metafunction as a special template class [6]. The template metafunction is the unit to encapsulate compile time computations in a standard way. The arguments of the metafunction are the template parameters of the class, the value of the function is a nested type of the template. The name of this nested type has been standardised by Boost.MPL, and it is called `type`. To evaluate a metafunction we provide actual parameters for the arguments, and we refer to the nested type as the value.

The possibility of writing compile-time metaprograms in C++ was not intentionally designed. Therefore, C++ compilers are not focused on template metaprograms as primary targets. The syntax of the metaprograms is far from trivial, and in most cases it is hard to understand. Debugging and profiling template metaprograms, although now supported by various tools [24,31], are still challenging.

7 Summary

More than 35 years after it has been created, the C++ programming language has still among the most important and frequently used mainstream programming languages. One of the reasons of its vitality is that C++ has successfully addressed challenges that have emerged from time to time. The RAII idiom and its consequences, like smart pointers, handled resource safety issues, generative programming and the STL created a complex, fully comprehensive, still effective and easy to use standard library. Lately, the new memory model and the multithreading library addressed the emerging request for supporting concurrent programming.

Not independently from concurrent programming, we experience a growing enthusiasm for functional programming and its toolset. Historically C++ was not rich in language elements directly supporting the functional paradigm. In this paper we attempted the summarize those classical and new language elements

that provide support for one of the major characteristics of functional paradigm: immutable programming. Immutability or referential transparency requires that objects must not change their value during runtime.

Although there is no direct support for immutable data types in C++, various existing language features can be used to achieve immutability. Constants, and const-correctness rules have been used in C++ from the beginning. STL supports and always supported constant correctness. Lambda functions, introduced to C++11, have a pure behaviour by default. Constant expressions, especially their extended form since C++14, provide a feasible way to implement immutable objects and pure functions. Template metaprograms are referentially transparent by nature, and compile time data structures, like typelists, are immutable. In the upcoming C++17 version constexpr lambdas, folding expressions will enhance the functional toolset of C++.

With this rich set of available language features, one can safely implement in modern C++ immutable data structures, pure functions and all the other means of functional programming.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond, p. 400. Addison-Wesley, Boston (2004). ISBN 0321-22725-6
2. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Boston (2001)
3. Allain, A.: Constexpr - Generalized Constant Expressions in C++11. http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html
4. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, Boston (1998)
5. Coplien, J.O.: Multi-Paradigm Design for C++. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Boston (2000)
7. Dami, L.: More functional reusability in C/C++/ Objective-C with curried functions. Object Composition, pp. 85–98. Centre Universitaire d'Informatique, University of Geneva, June 1991
8. Ellis, M., Stroustrup, B.: The Annotated C++ Reference Manual. Addison-Wesley, Boston (1990)
9. Fejerčák, V., Szabó, Cs., Bollin, A.: A software reverse engineering experience with the AMEISE legacy system. In: Electrical Engineering and Informatics 6: Proceedings of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice, pp. 357–362. FEI TU, Košice (2015). ISBN 978-80-553-2178-3
10. Gil, Y., Lenz, K.: Simple and safe SQL queries with C++ templates. In: Consela, C., Lawall, J.L. (eds.) 6th International Conference on Generative Programming and Component Engineering, GPCE 2007, Salzburg, Austria, 1–3 October, pp. 13–24 (2007)
11. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. **21**(3), 359–411 (1989). https://doi.org/10.1145/72551.72554

12. The C++11 Standard: ISO International Standard, ISO/IEC 14882:2011(E) - Information technology - Programming languages - C++ (2011)
13. The C++14 Standard: ISO International Standard, ISO/IEC 14882:2014(E) - Programming Language C++ (2014)
14. Järvi, J., Powell, G., Lumsdaine, A.: The Lambda library: unnamed functions in C++. Softw. Pract. Exper. **33**(3), 259–291 (2003). https://doi.org/10.1002/spe.504
15. Järvi, J., Freeman, J.: C++ lambda expressions and closures. Sci. Comput. Program. **75**(9), 762–772 (2010)
16. Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Boston (2005)
17. Kernighan, B.W., Ritche, D.M.: The C Programming Language, vol. 2. Prentice-Hall, Englewood Cliffs (1988)
18. Kiselyov, O.: Functional style in C++: closures, late binding, and Lambda abstractions. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, (ICFP 1998), p. 337. ACM, New York (1998). https://doi.org/10.1145/289423.289464
19. Koopman, P., Plasmeijer, R., Achten, P.: An executable and testable semantics for iTasks. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 212–232. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24452-0_12
20. McNamara, B., Smaragdakis, Y.: Functional programming in C++. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pp. 118–129 (2000)
21. McNamara, B., Smaragdakis, Y.: Functional programming in C++ using the FC++ library. SIGPLAN Not. **36**(4), 25–30 (2001)
22. Meyers, S.: Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley, Boston (2001)
23. Meyers, S.: Effective Modern C++. O'Reilly Media, Sebastopol (2014). ISBN 978-1-4919-0399-5, ISBN 10 1-4919-0399-6
24. Mihalicza, J., Pataki, N., Porkoláb, Z.: Compiler support for profiling C++ template metaprograms. In: Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST 2011), pp. 32–43, October 2011
25. Milewski, B.: Functional Data Structures in C++. C++Now, Aspen (2015). https://www.youtube.com/watch?v=OsB09djvfl4
26. Milewski, B.: C++ Ranges are Pure Monadic Goodness. B. Milewski's blog. https://bartoszmilewski.com/2014/10/17/c-ranges-are-pure-monadic-goodness/
27. Musser, D.R., Stepanov, A.A.: Algorithm-oriented generic libraries. Softw.-Pract. Exper. **27**(7), 623–642 (1994)
28. Niebler, E.: Ranges for the Standard Library proposal, Rev. 1, N4128, 10 October 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html
29. Pataki, N., Szűgyi, Z., Dévai, G.: C++ standard template library in a safer way. In: Proceedings of Workshop on Generative Technologies (WGT 2010), pp. 46–55 (2010)
30. Porkoláb, Z.: Functional programming with C++ template metaprograms. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 306–353. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17685-2_9
31. Sinkovics, Á.: Interactive metaprogramming shell based on clang. In: Lecture at C++Now Conference. Aspen, Co., US (2015). https://www.youtube.com/watch?v=oCbeXpJKzlM

32. Sinkovics, Á., Porkoláb, Z.: Expressing C++ template metaprograms as lambda expressions. In: Horváth, Z., Zsók, V., Achten, P., Koopman, P. (eds.) Proceedings of Tenth Symposium on Trends in Functional Programming, Komárno, Slovakia, 2–4 June 2009, pp. 97–111 (2009)
33. Sinkovics, Á., Porkoláb, Z.: Implementing monads for C++ template metaprograms. In: Science of Computer Programming. https://doi.org/10.1016/j.scico.2013.01.002, http://www.sciencedirect.com/science/article/pii/S0167642313000051. ISSN 0167-6423. Accessed 23 Jan 2013
34. Sinkovics, Á, Porkoláb, Z.: Domain-specific language integration with C++ template metaprogramming. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, pp. 32–55. IGI Global (2013). https://doi.org/10.4018/978-1-4666-2092-6.ch002. Accessed 30 Apr 2014
35. Sipos, Á., Zsók, V.: EClean – an embedded functional language. Electron. Not. Theoret. Comput. Sci. **238**(2), 47–58 (2009)
36. Sommerlad, P.: C++14 Compile-time computation (ACCU 2015). http://wiki.hsr.ch/PeterSommerlad/files/ACCU2015VariadicVariableTemplates.pdf
37. Stepanov, A.: From Mathematics to Generic Programming, 1st edn. Addison-Wesley, Boston (2014). ISBN-10: 0321942043, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3649.html
38. Stroustrup, B.: A history of C++: 1979–1991. In: The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II), pp. 271–297. ACM, New York (1996). https://doi.org/10.1145/154766.155375
39. Stroustrup, B.: The C++ Programming Language, 4th edn. Addison-Wesley Professional, Boston (2013). ISBN-10 0321563840
40. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Boston (1994)
41. Szűgyi, Z., Sinkovics, Á., Pataki, N., Porkoláb, Z.: C++ metastring library and its applications. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 461–480. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_15
42. Unruh, E.: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462
43. Vali, F., Sutter, H., Abrahams, D.: N3649 Generic (Polymorphic) Lambda Expressions (Revision 3). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3649.html
44. Veldhuizen, T.: C++ Templates are Turing Complete. Technical report, Indiana University Computer Science (2003). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670
45. Veldhuizen, T., Gannon, D.: Active libraries: rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, OO 1998 (1998)
46. Veldhuizen, T.: Expression Templates. C++ Report, vol. 7, pp. 26–31 (1995)
47. Zsók, V., Koopman, P., Plasmeijer, R.: Generic executable semantics for d-clean. Electron. Not. Theoret. Comput. Sci. **279**(3), 85–95 (2011)
48. Järvi, J.: The Boost Lambda library. http://www.boost.org/doc/libs/1_60_0/doc/html/lambda.html
49. Literal types in Draft C++14 standard. Working Draft, Standard for Programming Language C++. ANSI C++ N4290, 19 November 2014. 3.9. [10]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf
50. Constant expressions in Draft C++14 standard. Working Draft, Standard for Programming Language C++. ANSI C++ N4290, 19 November 2014. 5.20. [4]