Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 63–71
doi: 10.14794/ICAI.10.2017.63

Programming Language History: Experiences based on the Evolution of C++

Tibor Brunner, Zoltán Porkoláb

Eötvös Loránd University
bruntib@caesar.elte.hu
gsd@caesar.elte.hu

Abstract

Programming languages evolve continuously. From the very first experimental versions users desire new features and techniques to make their life safer and easier. The assumption is that most of these requests are about to raise the level of abstractions on which the programmers can define their programs. However, this general belief has not been supported by scientific researches yet. In this paper we investigate the evolution of the C++ programming language from the point of the abstraction level of certain constructions. As the C++ language has a long history with well-defined steps of standardization introducing various new language features and occasionally deprecating deadends; it is a fortunate choice for such a research. We will present how the abstraction level solving typical problems has been raised version by version.

Keywords: C++ programming language, language evolution, language history

MSC: 68N15 Programming languages

1. Diversity of programming languages

The main tools of software development are programming languages. The programming principles, e.g. object orientation, polymorphism, etc. are provided by different solutions and techniques. For example JavaScript has object types for encapsulation, function closure for data hiding and prototype based inheritance for code reuse [5]. C++ or Java and many other languages provide classes for covering these three main elements of object orientation. Comparing C++ and Java way of parametric polymorphism we can experience instantiation and type erasure (universal polymorphism) respectively [6]. The languages can also be categorized by many other features. There are domain specific languages for solving the tasks of a restricted domain area [2] and general purpose languages for a broad range

of projects. There are interpreted languages for fast prototyping and compiled languages for efficient, but platform specific production development.

Naturally the languages evolve so they can satisfy the programmers' requirements to make coding more effective and less error prone. This leads to the introduction of new language features and occasionally the removal of deprecated structures proved to be dangerous. These modifications should happen by conserving the main principles of the given language, however we can observe a tendency according to which the languages tend to converge towards multi-paradigm style [3].

The language creators have to take into account that the previously written code bases already use many language features so deprecating them would be very costly for the users. Therefore, it would be important to design a language feature so precisely and consistently that minimizing the need to change them. The question arises that which features should stay and which should go? Can we change backward compatibility in order to keep the language clean? In the next section we will examine some language features to answer these questions.

2. Exception handling in C++

Every program may come across an unexpected situation: a file supposed to use is unavailable, a network connection is broken or some other conditions do not meet the expectations. The definitive way to handle these unexpected situations or certain kind of errors in modern programming languages is exception handling. An exception is an event which cannot be handled at the place of its occurrence either because of the lack of information of the calling context, or because it is not its responsibility to handle this case. When such an event happens, the caller has to be notified to manage the failure. It is the caller's choice to rerun the questionable block maybe after some modification of the environment, or skip the code block and continue by ignoring the error or just finish the program with a message indicating the error.

```
struct S {...};

void f() throw(int, const char*, S) {
  if (error1)
    throw 42;
  else if (error2)
    throw "Something bad happened.";
  else
    throw S();
}
```

Note that exception handling means an unconditional jump from the place of failure towards the callers on the path of stack trace. Along this path an object can be transported which carries information about the reason of the error. It is

said that the object is *thrown* and the callers can *catch* it. In C++ the thrown object may have any type [4].

For the first sight it seems that it would be useful to inform the caller side what type of errors can occur so it can prepare on these, otherwise a general branch has to be created to receive all unknown thrown object types.

```
int main() {
  try {
    f();
  } catch (int i) {
    do_something_with_int(i);
  } catch (const S& s) {
    do_something_with_S(s);
  } catch (...) {
    do_something_else();
  }
}
```

In the signature of function `f()` we can see the possible types which can be thrown. Not like in Java, the C++ functions are not enforced to enumerate all the types it may throw. Both solutions have advantages and disadvantages. The advantage of listing all thrown object types is that the caller can count on their arrival. The disadvantage is that a function may invoke another function which is provided by a third party library. In a new version of that library the exception specification may change and this would make the exception specification of our function invalid resulting the rewriting a possibly great amount of code which relies on this information. Since the primary principle of C++ language is runtime performance, it is not recommended to throw exceptions at all, as it is a very costly operation. Therefore according to C++ conventions it is not a good attitude to use exceptions if not necessary. This is also the reason why C++ chose not to oblige listing of exception types.

It is also an important question that what happens if the rule is violated, i.e. if an exception object is thrown of which the type is not enumerated in the specification list of the function. The following rules guide the control of this event.

- If the type of the thrown exception (or any of its base types) is listed in the specification list then it can be handled properly.
- When the thrown type not in the specification list, then `std::unexpected()` function is called. By default it calls `std::unexpected_handler()` or an own handler can be set instead.
- `std::unexpected_handler()` calls `std::terminate()`. By default it calls `std::terminate_handler()` or an own handler can be set instead. The `std::terminate()` calls `std::abort()`.

- If an own handler throws an exception from `std::unexpected()` and it is not a descendant of `std::bad_exception` type then it is re-thrown, otherwise `std::terminate()` is called.

It is clearly visible that exception handling rules are quite complicated in the case when an unexpected exception happens in a function call. No wonder this feature has been deprecated in the language and `noexcept` specifier has been introduced instead since C++11. This requires a boolean expression which indicates whether the function may throw any type of exception. Should this rule be violated, the `std::terminate()` function will be called.

3. Sum of areas

In the previous section we saw that exception specification turned out to be a disadvantageous tool on the palette of C++ features, thus has been lead out from the language. By contrast in this section we will follow through a set of tools which have been introduced in chronological order and made the language simpler and less error prone. We will present a series of solutions for the same task, namely for summarizing the areas of different shapes.

The traditional solution is to maintain a vector of pointers to shapes where a shape is represented by a class. This class should have a virtual `area()` function. Subtype polymorphism allows us to point to a circle or a square with a pointer of type `Shape*`. This way, as the definition of virtual function regulates, the `area()` function based on the pointer's dynamic type will be invoked.

Solution in C In C programming language there were no classes, nor methods (i.e. function defined inside a class) [9], thus virtual function calls had to be emulated by implementing this logic explicitly. A typical implementation of virtual function calls is to build a *virtual table*. This means that every class which has virtual methods, a virtual table is given which contains pointers to the specific override functions in child classes. This way we can find which function needs to be run based on the dynamic type of the object.

Let us create a `Shape` structure then which represent an ordinary shape by its center coordinates x and y.

```
struct ShapeVTable {
  double (*area)(const struct Shape* this);
};

struct Shape {
  struct ShapeVTable* vptr;
  double x, y;
};
```

Next let us create a `Circle` structure. In C programming language we do not have the opportunity to express subtype connection between the two structures, but these can be linked by a member which contains the "inherited" members from `Shape` structure. After all these we can create a `sum()` function which computes the sum of the area of different shapes in an array.

```
struct Circle {
  struct Shape base;
  double r;
};

double circleArea(const struct Circle* c)
{
  return c->r * c->r * 3.14;
}

void initCircle(struct Circle* c, double r) {
  c->base.vptr->area = circleArea;
  c->r = r;
}

double sum(Shape* shapes[], int size) {
  double s = 0;
  for (int i = 0; i < size; ++i)
    s += shapes[i]->vptr->area(shapes[i]);
  return s;
}
```

In this solution one can find many inconveniences. The "constructor"-like function, named `initCircle()` has to be called explicitly by the user of this module, since it is needed to set the elements of the virtual table. The second is the invocation of the `area()` function. It is ugly to indicate the circle `shapes[i]` twice: for acquiring the corresponding area function from the virtual table and for passing it as parameter to the area function, as required by its signature.

Virtual functions C++ language gives a solution on these drawbacks by classes and virtual functions [12]. The logic programmed manually above can be substituted by indicating that a function is *virtual*. This means that given a pointer to a base class (i.e. its static type is `Shape*`) which actually points to a child class (i.e. its dynamic type is `Circle*`) the `area()` function based on the dynamic type will be invoked.

Of course subtype relation is needed between the two types which is expressed by "public inheritance" [14]. This makes the `sum()` function simpler:

```
class Shape {
  double x, y;
public:
  virtual double area() const = 0;
};
class Circle : public Shape {
  double r;
public:
  virtual double area() const {
    return r * r * 3.14;
  }
};
double sum(Shape* shapes[], int size) {
  double s = 0;
  for (int i = 0; i < size; ++i)
    s += shapes[i]->area();
  return s;
}
```

Templates and iterators Let us consider some more simplification of this program. The function parameter restrict the container type on array. This could be more general by using parametric polymorphism, i.e. templates for parameterize the container type. The new signature of `sum()` function looks as follows:

```
template <typename Cont>
double sum(const Cont<Shape*>& cont);
```

The bridge between the container and the algorithms are iterators [1]. This is an appropriate tool for inspecting the single elements of a container. Iterators can also be considered the generalization of pointers, since the provided operations are similar. Depending on the type of an iterator they can be incremented, decremented, dereferenced, added or subtracted to integers, etc. [11].

```
for (typename Cont::const_iterator it = shapes.begin();
      it != shapes.end(); ++it)
  s += (*it)->area();
```

It is not too comfortable to use such long iterator types, not to mention that in case of changing the underlying container, the loop variable's type might also change. C++11 introduced the automatic type detection which means that the type of a variable can be deduced from the type of the initialization expression. Thus it is enough to use `auto` as the type of `it`.

Sometimes it is sufficient to process only a subrange of the whole container. However most of the times all elements must be considered. To support this use case C++11 also introduced the range based `for` loop [10]. Its semantics are equivalent as the code fragment in the previous example. This means that a range based `for` loop can only be used for containers which provide `begin()` and `end()` methods which define the range to iterate through.

```
for (Shape* shape : shapes)
  s += shape->area();
```

Standard Template Library For common algorithms the Standard Template Library gives a solution [13, 1, 11]. If not necessary then it is not worth writing a code twice. Summarizing a bunch of elements is a so common task that the built-in library already provides a solution which is even more general in the sense that besides addition it can apply any binary operation on the elements and it can start with any initial value (e.g. 1 for multiplication).

```
double op(double d, const Shape* shape) {
  return d + shape->area();
}
template <typename Cont>
double sum(const Cont<Shape*>& shapes) {
  return std::accumulate(shapes.begin(), shapes.end(), 0, op);
}
```

One can see the next step of simplification: it is unnecessary to define `op()` operation as a standalone function if it is used only at one place in the program. It would be more concise the use unnamed, in-line function for this purpose. In C++11 lambdas [8] are available:

```
return std::accumulate(shapes.begin(), shapes.end(), 0,
  [](double d, const Shape* shape){ return d + shape->area(); });
```

Shapes from different sources Now let us change the specification a bit and suppose that the shapes come from several different sources: files, network, standard input, etc. It may not worth to maintain a container for storing the arriving shapes first and then summarizing their areas in a second step. We create a function instead which may accept any number of parameters:

```
double s = sum(
  circleFromFile(),
  squareFromNetwork(),
  triangleFromStdIn());
```

In C++11 we can create so called *variadic templates* which means that the number of template arguments may vary based on the caller [7]. This way a set of `sum()` function can be instantiated in compile time of which the argument numbers are different. The recursion needs to stop for which a non-template function is used:

```
double sum() {
  return 0;
}
template <typename... Args>
double sum(const Shape* shape, Args... shapes) {
  return shape->area() + sum(shapes...);
}
```

The disadvantage of this solution is that the number of template instantiations depend on the number of arguments. This increases the size of the compiled binary. In C++17 *folding expression* solves this problem [15]. This expression applies a binary operator on all elements of the variadic template argument list:

```
template <typename... Args>
double sum(Args... shapes) {
  return (shapes->area() + ...);
}
```

4. Conclusion

Programming languages evolve continuously. New hardware solutions, changing programming paradigms put new requirements on the languages. As we have seen, C++ is not an exception; it has a long history of introducing new features and sometimes deprecated old elements. However, we can experience a general trend: new language features are aiming programming techniques which make the code shorter, clearer and more maintainable.

As the programming paradigms evolved from the structural/procedural programming represented by the C programming language towards object-orientation with inheritance and polymorphism in C++, the same solution was expressed in more compact and more understandable way. The abstraction level of the resulted program has been raised. This evolution accelerated with the application of the generic programming paradigm and the STL and summitted with the addition of such functional tools like the lambda function in C++11 and the folding expressions in C++17.

On the other hand, language elements which proved to be inherently complex and uncomfortable, like the exception specifications, should be removed from modern programming languages.

References

[1] Austern, M. H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, 1998.

[2] Czarnecki, K., Eisenecker, U. W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, 2000.

[3] Coplien, J. O.: Multi-Paradigm Design for C++. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1998.

[4] Ellis, M., Stroustrup, B.: The Annotated C++ Reference Manual. Addison-Wesley, 1990.

[5] Flanagan, D.: JavaScript: the definitive guide. O'Reilly Media, Inc., 2006.

[6] Ghosh, D.: Generics in Java and C++: a comparative model. ACM SIGPLAN Notices, 2004, 39.5: 40-47.

[7] Gregor, D., Järvi, J.: Variadic templates for C++. In: Proceedings of the 2007 ACM symposium on Applied computing. ACM, 2007. p. 1101-1108.

[8] Järvi, J., Freeman, J.: C++ lambda expressions and closures. Science of Computer Programming 75.9 (2010): 762-772.

[9] Kernighan, B. W., Ritchie, D. M. (2006). The C programming language.

[10] Meyers, S.: Effective modern C++ (O'Reilly Media, 2014) ISBN 978-1-4919-0399-5 | ISBN 10 1-4919-0399-6

[11] Musser, D. R., Stepanov, A. A.: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.

[12] Stroustrup, B.: A history of C++: 1979–1991. In The second ACM SIGPLAN conference on History of programming languages (HOPL-II). ACM, New York, NY, USA, 271-297. DOI=http://dx.doi.org/10.1145/154766.155375

[13] Stroustrup, Bjarne, *The C++ Programming Language, 4th ed.*, Addison-Wesley, ISBN 978-0321563842, 2013.

[14] Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)

[15] Sutton, A., Smith, R.: Folding expressions. N4191 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4191.html

[16] ISO International Standard, *ISO/IEC 14882:2011(E) – Programming Language C++*, 2011.

[17] ISO International Standard, *ISO/IEC 14882:2014(E) – Programming Language C++*, 2014.