Part IV

Broadening the View

Ideals and History

"When someone says,
'I want a programming language
in which I need only say what I wish done,'
give him a lollipop."

—Alan Perlis

This chapter is a very brief and very selective history of programming languages and the ideals they have been designed to serve. The ideals and the languages that express them are the basis for professionalism. Because C++ is the language we use in this book, we focus on C++ and languages that influenced C++. The aim is to give a background and a perspective to the ideas presented in this book. For each language, we present its designer or designers: a language is not just an abstract creation, but a concrete solution designed by individuals in response to problems they faced at the time.

22.1 History, ideals, and professionalism

"History is bunk," Henry Ford famously declared. The contrary opinion has been widely quoted since antiquity: "He who does not know history is condemned to repeat it." The problem is to choose which parts of history to know and which parts to discard: "95% of everything is bunk" is another relevant quote (with which we concur, though 95% is probably an underestimate). Our view of the relation of history to current practice is that there can be no professionalism without some understanding of history. If you know too little of the background of your field, you are gullible because the history of any field of work is littered with plausible ideas that didn't work. The "real meat" of history is ideas and ideals that have proved their worth in practical use.

We would have loved to talk about the origins of key ideas in many more languages and in all kinds of software, such as operating systems, databases, graphics, networking, the web, scripting, etc., but you'll have to find those important and useful areas of software and programming elsewhere. We have barely enough space to scratch the surface of the ideals and history of programming languages.

The ultimate aim of programming is always to produce useful systems. In the heat of discussions about programming techniques and programming languages, that's easily forgotten. Don't forget that! If you need a reminder, take another look at Chapter 1.

22.1.1 Programming language aims and philosophies

What is a programming language? What is a programming language supposed to do for us? Popular answers to "What is a programming language?" include

- A tool for instructing machines
- A notation for algorithms
- A means of communication among programmers
- A tool for experimentation
- A means of controlling computerized devices
- A way of expressing relationships among concepts
- A means of expressing high-level designs

Our answer is "All of the above – and more!" Clearly, we are thinking about general-purpose programming languages here, as we will throughout this chapter. In addition, there are special-purpose languages and domain-specific languages serving narrower and typically more precisely defined aims.

What properties of a programming language do we consider desirable?

- Portability
- Type safety
- Precisely defined
- High performance
- Ability to concisely express ideas
- Anything that eases debugging
- Anything that eases testing
- Access to all system resources
- Platform independence
- Runs on all platforms (e.g., Linux, Windows, smartphones, embedded systems)
- Stability over decades
- Prompt improvements in response to changes in application areas
- Ease of learning
- Small
- Support for popular programming styles (e.g., object-oriented programming and generic programming)
- Whatever helps analysis of programs

- Lots of facilities
- Supported by a large community
- Supportive of novices (students, learners)
- Comprehensive facilities for experts (e.g., infrastructure builders)
- Lots of software development tools available
- Lots of software components available (e.g., libraries)
- Supported by an open software community
- Supported by major platform vendors (Microsoft, IBM, etc.)

Unfortunately, we can't have all this at the same time. That's sad because every one of these "properties" is objectively a good thing: each provides benefits, and a language that doesn't provide them imposes added work and complications on its users. The reason we can't have it all is equally fundamental: several of the properties are mutually exclusive. For example, you cannot be 100% platform independent and also access all system resources; a program that accesses a resource that is not available on every platform cannot run everywhere. Similarly, we obviously want a language (and the tools and libraries we need to use it) that is small and easy to learn, but that can't be achieved while providing comprehensive support for programming on all kinds of systems and for all kinds of application areas.

This is where ideals become important. Ideals are what guide the technical choices and trade-offs that every language, library, tool, and program designer must make. Yes, when you write a program you are a designer and must make design choices.

22.1.2 Programming ideals

The preface of *The C++ Programming Language* starts, "C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer." Say what? Isn't programming all about delivering products? About correctness, quality, and maintainability? About time-to-market? About efficiency? About supporting software engineering? That, too, of course, but we shouldn't forget the programmer. Consider another example: Don Knuth said, "The best thing about the Alto is that it doesn't run faster at night." The Alto was a computer from the Xerox Palo Alto Research Center (PARC) that was one of the first "personal computers," as opposed to the shared computers for which there was a lot of competition for daytime access.

Our tools and techniques for programming exist to make a programmer, a human, work better and produce better results. Please don't forget that. So what guidelines can we articulate to help a programmer produce the best software with the least pain? We have made our ideals explicit throughout the book so this section is basically a summary.

The main reason we want our code to have a good structure is that the structure is what allows us to make changes without excessive effort. The better the structure, the easier it is to make a change, find and fix a bug, add a new feature, port it to a new architecture, make it run faster, etc. That's exactly what we mean by "good."

For the rest of this section, we will

- Revisit what we are trying to achieve, that is, what we want from our code
- Present two general approaches to software development and decide that a combination is better than either alternative by itself
- Consider key aspects of program structure as expressed in code:
  - Direct expression of ideas
  - Abstraction level
  - Modularity
  - Consistency and minimalism

Ideals are meant to be used. They are tools for thinking, not simply fancy phrases to trot out to please managers and examiners. Our programs are meant to approximate our ideals. When we get stuck in a program, we step back to see if our problems come from a departure from some ideal; sometimes that helps. When we evaluate a program (preferably before we ship it to users), we look for departures from the ideals that might cause problems in the future. Apply ideals as widely as possible, but remember that practical concerns (e.g., performance and simplicity) and weaknesses in a language (no language is perfect) will often prevent you from achieving more than a good approximation of the ideals.

Ideals can guide us when making specific technical decisions. For example, we can't just make every single decision about interfaces for a library individually and in isolation (§14.1). The result would be a mess. Instead we must go back to our first principles, decide what is important about this particular library, and then produce a consistent set of interfaces. Ideally, we would articulate our design principles and trade-offs for that particular design in the documentation and in comments in the code.

During the start of a project, review the ideals and see how they relate to the problems and the early ideas for their solution. This can be a good way to get ideas and to refine ideas. Later in the design and development process, when you are stuck, step back and see where your code has most departed from the ideals – this is where the bugs are most likely to lurk and the design problems are most likely to occur. This is an alternative to the default technique of repetitively looking in the same place and trying the same techniques to find the bug. "The bug is always where you are not looking – or you would have found it already."

22.1.2.1 What we want

Typically, we want

- *Correctness:* Yes, it can be difficult to define what we mean by "correct," but doing so is an important part of the complete job. Often, others define for us what is correct for a given project, but then we have to interpret what they say.
- *Maintainability:* Every successful program will be changed over time; it will be ported to new hardware and software platforms, it will be extended with new facilities, and new bugs will be found that must be fixed. The sections below about ideals for program structure address this ideal.
- *Performance:* Performance ("efficiency") is a relative term. Performance has to be adequate for the program's purpose. It is often claimed that efficient code is necessarily low-level and that concerns with a good, high-level structure of the code cause inefficiency. On the contrary, we find that acceptable performance is often achieved through adherence to the ideals and approaches we recommend. The STL is an example of code that is simultaneously abstract and very efficient. Poor performance can as easily arise from an obsession with low-level details as it can from disdain for such details.
- *On-time delivery:* Delivering the perfect program a year late is usually not good enough. Obviously, people expect the impossible, but we need to deliver quality software in a reasonable time. There is a myth that "completed on time" implies shoddiness. On the contrary, we find that emphasis on good structure (e.g., resource management, invariants, and interface design), design for testability, and use of appropriate libraries (often designed for a specific application or application area) is a good way to meet deadlines.

This leads to a concern for structure in our code:

- If there is a bug in a program (and every large program has bugs), it is easier to find in a program with a clear structure.
- If a program needs to be understood by a new person or needs to be modified in some way, a clear structure is comprehensible with far less effort than a mess of low-level details.
- If a program hits a performance problem, it is often easier to tune a high-level program (one that is a good approximation of the ideals and has a well-defined structure) than a low-level or messy one. For starters, the high-level one is more likely to be understandable. Second, the high-level one is often ready for testing and tuning long before the low-level one.

Note the point about a program being understandable. Anything that helps us understand a program and helps us reason about it is good. Fundamentally, regularity is better than irregularity – as long as the regularity is not achieved through oversimplification.

22.1.2.2 General approaches

There are two approaches to writing correct software:

- *Bottom-up:* Compose the system using only components proved to be correct.
- *Top-down:* Compose the system out of components assumed to contain errors and catch all errors.

Interestingly, the most reliable systems combine these two – apparently contrary – approaches. The reason for that is simple: for a large real-world system, neither approach will deliver the needed correctness, adaptability, and maintainability:

- We can't build and "prove" enough basic components to eliminate all sources of errors.
- We can't completely compensate for the flaws of buggy basic components (libraries, subsystems, class hierarchies, etc.) when combining them in the final system.

However, a combination of approximations to the two approaches can deliver more than either in isolation: we can produce (or borrow or buy) components that are sufficiently good, so that the problems that remain can be compensated for by error handling and systematic testing. Also, if we keep building better components, a larger part of a system can be constructed from them, reducing the amount of "messy ad hoc code" needed.

Testing is an essential part of software development. It is discussed in some detail in Chapter 26. Testing is the systematic search for errors. "Test early and often" is a popular slogan. We try to design our programs to simplify testing and to make it harder for errors to "hide" in messy code.

22.1.2.3 Direct expression of ideas

When we express something – be it high-level or low-level – the ideal is to express it directly in code, rather than through work-arounds. The fundamental ideal of representing our ideas directly in code has a few specific variants:

- *Represent ideas directly in code*. For example, it is better to represent an argument as a specific type (e.g., **Month** or **Color**) than as a more general one (e.g., **int**).
- *Represent independent ideas independently in code*. For example, with a few exceptions, the standard **sort()** can sort any standard container of any

element type; the concepts of sorting, sorting criteria, container, and element type are independent. Had we built a "**vector** of objects allocated on the free store where the elements are of a class derived from **Object** with a **before()** member function defined for use by **vector::sort()**" we would have a far less general **sort()** because we made assumptions about storage, class hierarchy, available member functions, ordering, etc.

- *Represent relationships among ideas directly in code*. The most common relationships that can be directly represented are inheritance (e.g., a **Circle** is a kind of **Shape**) and parameterization (e.g., a **vector<T>** represents what's common for all vectors independently of a particular element type).
- *Combine ideas expressed in code freely – where and only where combinations make sense*. For example, **sort()** allows us to use a variety of element types and a variety of containers, but the elements must support **<** (if they do not, we use the **sort()** with an extra argument specifying the comparison criteria), and the containers we sort must support random-access iterators.
- *Express simple ideas simply*. Following the ideals listed above can lead to overly general code. For example, we may end up with class hierarchies with a more complicated taxonomy (inheritance structure) than anyone needs or with seven parameters to every (apparently) simple class. To avoid every user having to face every possible complication, we try to provide simple versions that deal with the most common or most important cases. For example, we have a **sort(b,e)** that implicitly sorts using less than in addition to the general version **sort(b,e,op)** that sorts using **op**. We could also provide versions **sort(c)** for sorting a standard container using less than and **sort(c,op)** for sorting a standard container using **op**.

22.1.2.4 Abstraction level

We prefer to *work at the highest feasible level of abstraction*; that is, our ideal is to express our solutions in as general a way as possible.

For example, consider how to represent entries for a phone book (as we might keep it on a PDA or a cell phone). We could represent a set of (name,value) pairs as a **vector<pair<string,Value_type>>**. However, if we essentially always accessed that set using a name, **map<string,Value_type>** would be a higher level of abstraction, saving us the bother of writing (and debugging) access functions. On the other hand, **vector<pair<string,Value_type>>** is itself a higher level of abstraction than two arrays, **string[max]** and **Value_type[max]**, where the relationship between the string and its value is implicit. The lowest level of abstraction would be something like an **int** (number of elements) plus two **void***s (pointing to some form of representation, known to the programmer but not to the compiler). In our

example, every suggestion so far could be seen as too low-level because it focuses on the representation of the pair of values, rather than their function. We could move closer to the application by defining a class that directly reflects a use. For example, we could write our application code using a class **Phonebook** with an interface designed for convenient use. That **Phonebook** class could be implemented using any one of the representations suggested.

The reason for preferring the higher level of abstraction (when we have an appropriate abstraction mechanism and if our language supports it with acceptable efficiency) is that such formulations are closer to the way we think about our problems and solutions than solutions that have been expressed at the level of computer hardware.

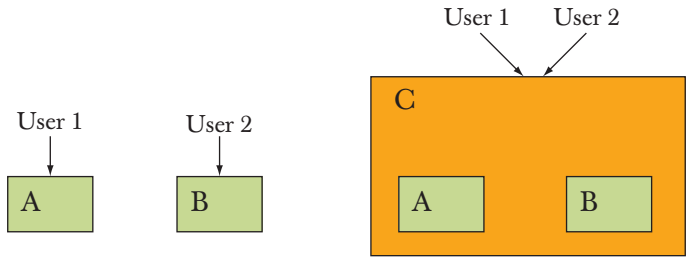
The reason given for dropping to a lower level of abstraction is typically "efficiency." This should be done only when really needed (§25.2.2). Using a lower-level (more primitive) language feature does not necessarily give better performance. Sometimes, it eliminates optimization opportunities. For example, using a **Phonebook** class, we have a choice of implementations, say, between **string[max]** plus **Value_type[max]** and **map<string,Value_type>**. For some applications the former is more efficient and for others the latter is. Naturally, performance would not be a major concern in an application involving only your personal directory. However, this kind of trade-off becomes interesting when we have to keep track of – and manipulate – millions of entries. More seriously, after a while, the use of low-level features soaks up the programmer's time so that opportunities for improvements (performance or otherwise) are missed because of lack of time.

22.1.2.5 Modularity

Modularity is an ideal. We want to compose our systems out of "components" (functions, classes, class hierarchies, libraries, etc.) that we can build, understand, and test in isolation. Ideally, we also want to design and implement such components so that they can be used in more than one program ("reused"). *Reuse* is the building of systems out of previously tested components that have been used elsewhere – and the design and use of such components. We have touched upon this in our discussions of classes, class hierarchies, interface design, and generic programming. Much of what we say about "programming styles" (in §22.1.3) relates to the design, implementation, and use of potentially "reusable" components. Please note that not every component can be used in more than one program; some code is simply too specialized and is not easily improved to be usable elsewhere.

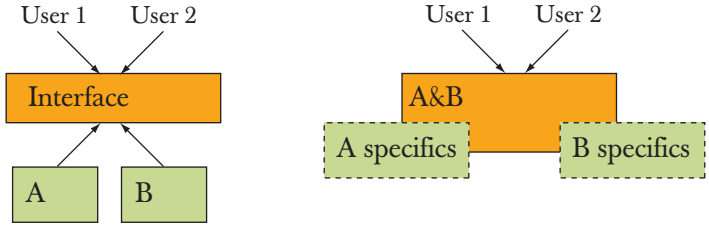
Modularity in code should reflect important logical distinctions in the application. We do not "increase reuse" simply by putting two completely separate

classes A and B into a "reusable component" called C. By providing the union of A's and B's interfaces, the introduction of C complicates our code:



Here, both User 1 and User 2 use C. Unless you look into C, you might think that User 1 and User 2 gained benefits from sharing a popular component. Benefits from sharing ("reuse") would (in this case, wrongly) be assumed to include better testing, less total code, larger user base, etc. Unfortunately, except for a bit of oversimplification, this is not a particularly rare phenomenon.

What would help? Maybe a common interface to A and B could be provided:



These diagrams are intended to suggest inheritance and parameterization, respectively. In both cases, the interface provided must be smaller than a simple union of A's and B's interfaces for the exercise to be worthwhile. In other words, A and B have to have a fundamental commonality for users to benefit from. Note how we again came back to interfaces (§9.7, §25.4.2) and by implication to invariants (§9.4.3).

22.1.2.6 Consistency and minimalism

Consistency and minimalism are primarily ideals for expressing ideas. So we might dismiss them as being about appearance. However, it is really hard to present a messy design elegantly, so demands of consistency and minimalism can be used as design criteria and affect even the most minute details of a program:

- Don't add a feature if you are in doubt about its utility.
- Do give similar facilities similar interfaces (and names), but only if the similarity is fundamental.
- Do give different facilities different names (and possibly different interface styles), but only if the differences are fundamental.

Consistent naming, interface style, and implementation style help maintenance. When code is consistent, a new programmer doesn't have to learn a new set of conventions for every part of a large system. The STL is an example (Chapters 20–21, §B.4–6). When such consistency is impossible (for example, for ancient code or code in another language), it can be an idea to supply an interface that matches the style of the rest of the program. The alternative is to let the foreign ("strange," "poor") style infect every part of a program that needs to access the offending code.

One way of preserving minimalism and consistency is to carefully (and consistently) document every interface. That way, inconsistencies and duplication are more likely to be noticed. Documenting pre-conditions, post-conditions, and invariants can be especially useful as can careful attention to resource management and error reporting. A consistent error-handling and resource management strategy is essential for simplicity (§19.5).

To some programmers, the key design principle is KISS ("Keep It Simple, Stupid"). We have even heard it claimed that KISS is the only worthwhile design principle. However, we prefer less evocative formulations, such as "Keep simple things simple" and "Keep it simple: as simple as possible, but no simpler." The latter is a quote from Albert Einstein, which reflects that there is a danger of simplifying beyond the point where it makes sense, thus damaging the design. The obvious question is, "Simple for whom and compared to what?"

22.1.3 Styles/paradigms

When we design and implement a program, we aim for a consistent style. C++ supports four major styles that can be considered fundamental:

- Procedural programming
- Data abstraction
- Object-oriented programming
- Generic programming

These are sometimes (somewhat pompously) called "programming paradigms." There are many more "paradigms," such as functional programming, logic programming, rule-based programming, constraints-based programming, and aspect-oriented programming. However, C++ doesn't support those directly, and we just can't cover everything in a single beginner's book, so we'll leave those to "future work" together with the mass of details that we must leave out about the paradigms/styles we do cover:

- *Procedural programming:* the idea of composing a program out of functions operating on arguments. Examples are libraries of mathematical functions, such as **sqrt()** and **cos()**. C++ supports this style of programming through the notion of functions (Chapter 8). The ability to choose to pass arguments by value, by reference, and by **const** reference can be

most valuable. Often, data is organized into data structures represented as **struct**s. Explicit abstraction mechanisms (such as private data members or member functions of a class) are not used. Note that this style of programming – and functions – is an integral part of every other style.

- *Data abstraction:* the idea of first providing a set of types suitable for an application area and then writing the program using those. Matrices provide a classic example (§24.3–6). Explicit data hiding (e.g., the use of private data members of a class) is heavily used. The standard **string** and **vector** are popular examples, which show the strong relationship between data abstraction and parameterization as used by generic programming. This is called "abstraction" because a type is used through an interface, rather than by directly accessing its implementation.
- *Object-oriented programming:* the idea of organizing types into hierarchies to express their relationships directly in code. The classic example is the **Shape** hierarchy from Chapter 14. This is obviously valuable when the types really have fundamental hierarchical relationships. However, there has been a strong tendency to overuse; that is, people built hierarchies of types that do not belong together for fundamental reasons. When people derive, ask why. What is being expressed? How does the base/derived distinction help me in this particular case?
- *Generic programming:* the idea of taking concrete algorithms and "lifting" them to a higher level of abstraction by adding parameters to express what can be varied without changing the essence of an algorithm. The **high()** example from Chapter 20 is a simple example of lifting. The **find()** and **sort()** algorithms from the STL are classic algorithms expressed in very general forms using generic programming. See Chapters 20–21 and the following example.

All together now! Often, people talk about programming styles ("paradigms") as if they were simple disjointed alternatives: either you use generic programming or you use object-oriented programming. If your aim is to express solutions to problems in the best possible way, you will use a combination of styles. By "best," we mean easy to read, easy to write, easy to maintain, and sufficiently efficient. Consider an example: the classic "**Shape** example" originated with Simula (§22.2.4) and is usually seen as an example of object-oriented programming. A first solution might look like this:

```
void draw_all(vector<Shape*>& v)
{
        for(int i = 0; i<v.size(); ++i) v[i]–>draw();
}
```

This does indeed look "rather object-oriented." It critically relies on a class hierarchy and on the virtual function call finding the right **draw()** function for every given **Shape**; that is, for a **Circle**, it calls **Circle::draw()** and for an **Open_polyline**, it calls **Open_polyline::draw()**. But the **vector<Shape*>** is basically a generic programming construct: it relies on a parameter (the element type) that is resolved at compile time. We could emphasize that by using a simple standard library algorithm to express the iteration over all elements:

```
void draw_all(vector<Shape*>& v)
{
        for_each(v.begin(),v.end(),mem_fun(&Shape::draw));
}
```

The third argument of **for_each()** is a function to be called for each element of the sequence specified by the first two arguments (§B.5.1). Now, that third function call is assumed to be an ordinary function (or a function object) called using the **f(x)** syntax, rather than a member function called by the **p–>f()** syntax. So, we use the standard library function **mem_fun()** (§B.6.2) to say that we really want to call a member function (the virtual function **Shape::draw()**). The point is that **for_each()** and **mem_fun()**, being templates, really aren't very "OO-like"; they clearly belong to what we usually consider generic programming. More interesting still, **mem_fun()** is a freestanding (template) function returning a class object. In other words, it can easily be classified as plain data abstraction (no inheritance) or even procedural programming (no data hiding). So, we could claim that this one line of code uses key aspects of all of the four fundamental styles supported by C++.

But why would we write the second version of the "draw all **Shape**s" example? It fundamentally does the same thing as the first version; it even takes a few more characters to write it in that way! We could argue that expressing the loop using **for_each()** is "more obvious and less error-prone" than writing out the **for**-loop, but for many that's not a terribly convincing argument. A better one is that "**for_each()** says what is to be done (iterate over a sequence) rather than how it is to be done." However, for most people the convincing argument is simply that "it's useful": it points the way to a generalization (in the best generic programming tradition) that allows us to solve more problems. Why are the shapes in a **vector**? Why not a **list**? Why not a general sequence? So we can write a third (and more general) version:

```
template<class Iter> void draw_all(Iter b, Iter e)
{
        for_each(b,e,mem_fun(&Shape::draw));
}
```

This will now work for all kinds of sequences of shapes. In particular, we can even call it for the elements of an array of **Shape**s:

```
Point p {0,100};
Point p2 {50,50};
Shape* a[] = { new Circle(p,50), new Triangle(p,p2,Point(25,25)) };
draw_all(a,a+2);
```

We could also provide a version that is simpler to use by restricting it to work on containers:

```
template<class Cont> void draw_all(Cont& c)
{
        for (auto& p : c) p->draw();
}
```

Or even, using C++14 concepts (§19.3.3):

```
void draw_all(Container& c)
{
        for (auto& p : c) p->draw();
}
```

The point is still that this code is clearly object-oriented, generic, and very like ordinary procedural code. It relies on data abstraction in its class hierarchy and the implementation of the individual containers. For lack of a better term, programming using the most appropriate mix of styles has been called *multi-paradigm programming*. However, I have come to think of this as simply *programming*: the "paradigms" primarily reflect a restricted view of how problems can be solved and weaknesses in the programming languages we use to express our solutions. I predict a bright future for programming with significant improvements in technique, programming languages, and support tools.

22.2 Programming language history overview

In the very beginning, programmers chiseled the zeros and ones into stones by hand! Well, almost. Here, we'll start (almost) from the beginning and quickly introduce some of the major developments in the history of programming languages as they relate to programming using C++.

There are a lot of programming languages. The rate of language invention is at least 2000 a decade, and the rate of "language death" is about the same. Here, we cover almost 60 years by briefly mentioning ten languages. For more information, see http://research.ihost.com/hopl/HOPL.html. There, you can find links to all the articles of the three ACM SIGPLAN HOPL (History of Programming Languages) conferences. These are extensively peer-reviewed papers – and

therefore far more trustworthy and complete than the average web source of information. The languages we discuss here were all represented at HOPL. Note that if you type the full title of a famous paper into a web search engine, there is a good chance that you'll find the paper. Also, most computer scientists mentioned here have home pages where you can find much information about their work.

Our presentation of a language in this chapter is necessarily very brief: each language mentioned – and hundreds not mentioned – deserves a whole book. We are also very selective in what we mention about a language. We hope you take this as a challenge to learn more rather than thinking, "So that's all there is to language X!" Remember, every language mentioned here was a major accomplishment and made an important contribution to our world. There is just no way we could do justice to these languages in this short space – but not mentioning any would be worse. We would have liked to supply a bit of code for each language, but sorry, this is not the place for such a project (see exercises 5 and 6).

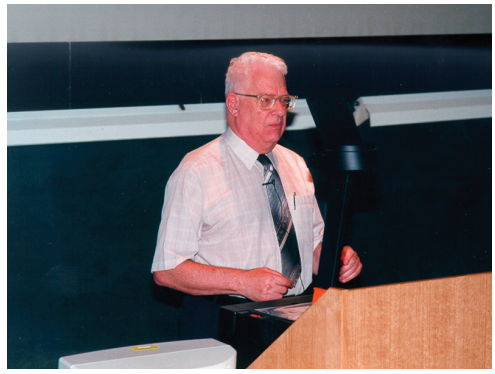
Far too often, an artifact (e.g., a programming language) is presented as simply what it is or as the product of some anonymous "development process." This misrepresents history: typically – especially in the early and formative years – a language is the result of the ideals, work, personal tastes, and external constraints on one or (typically) more individuals. Thus, we emphasize key people associated with the languages. IBM, Bell Labs, Cambridge University, etc. do not design languages; individuals from such organizations do – typically in collaboration with friends and colleagues.

Please note a curious phenomenon that often skews our view of history. Photographs of famous scientists and engineers are most often taken when they are famous and distinguished, members of national academies, Fellows of the Royal Society, Knights of St. John, recipients of the Turing Award, etc. – in other words, when they are decades older than when they did their most spectacular work. Almost all were/are among the most productive members of their profession until late in life. However, when you look back to the birth of your favorite language features and programming techniques, try to imagine a young man (there are still far too few women in science and engineering) trying to figure out if he has sufficient cash to invite a girlfriend out to a decent restaurant or a parent trying to decide if a crucial paper can be submitted to a conference at a time and place that can be combined with a vacation for a young family. The gray beards, balding heads, and dowdy clothes come much later.

22.2.1 The earliest languages

When – starting in 1949 – the first "modern" stored-program electronic computers appeared, each had its own language. There was a one-to-one correspondence between the expression of an algorithm (say, a calculation of a planetary orbit) and instructions for a specific machine. Obviously, the scientist (the users were most often scientists) had notes with mathematical formulas, but the program

was a list of machine instructions. The first primitive lists were decimal or octal numbers – exactly matching their representation in the computer's memory. Later, assemblers and "auto codes" appeared; that is, people developed languages where machine instructions and machine facilities (such as registers) had symbolic names. So, a programmer might write "LD R0 123" to load the contents of the memory with the address 123 into register 0. However, each machine had its own set of instructions and its own language.



David Wheeler from the University of Cambridge Computer Laboratory is the obvious candidate for representing programming language designers of that time. In 1949, he wrote the first real program ever to run on a stored-program computer (the "table of squares" program we saw in §4.4.2.1). He is one of about ten people who have a claim on having written the first compiler (for a machine-specific "auto code"). He invented the function call (yes, even something so apparently simple needs to have been invented at some point). He wrote a brilliant paper on how to design libraries in 1951; that paper was at least 20 years ahead of its time! He was co-author with Maurice Wilkes (look him up) and D. J. Gill of the first book about programming. He received the first Ph.D. in computer science (from Cambridge in 1951) and later made major contributions to hardware (cache architectures and early local-area networks) and algorithms (e.g., the TEA encryption algorithm [§25.5.6] and the "Burrows-Wheeler transform" [the compression algorithm used in bzip2]). David Wheeler happens to have been Bjarne Stroustrup's Ph.D. thesis adviser – computer science is a young discipline. David Wheeler did some of his most important work as a grad student. He worked on to become a professor at Cambridge and a Fellow of the Royal Society.

References

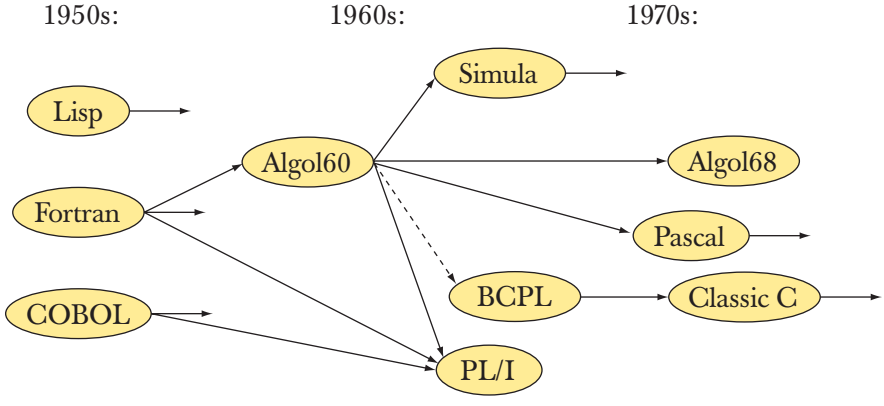
Burrows, M., and David Wheeler. "A Block Sorting Lossless Data Compression Algorithm." Technical Report 124, Digital Equipment Corporation, 1994.

Bzip2 link: www.bzip.org/.

Cambridge Ring website: http://koo.corpus.cam.ac.uk/projects/earlyatm/cr82.
Campbell-Kelly, Martin. "David John Wheeler." *Biographical Memoirs of Fellows of the Royal Society,* Vol. 52, 2006. (His technical biography.)
EDSAC: http://en.wikipedia.org/wiki/EDSAC.
Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley, 1968, and many revisions. Look for "David Wheeler" in the index of each volume.
TEA link: http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm.
Wheeler, D. J. "The Use of Sub-routines in Programmes." Proceedings of the 1952 ACM National Meeting. (That's the library design paper from 1951.)
Wilkes, M. V., D. Wheeler, and D. J. Gill. *Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley, 1951; 2nd edition, 1957. The first book on programming.

22.2.2 The roots of modern languages

Here is a chart of important early languages:



These languages are important partly because they were (and in some cases still are) widely used or because they became the ancestors to important modern languages – often direct descendants with the same name. In this section, we address the three early languages – Fortran, COBOL, and Lisp – to which most modern languages trace their ancestry.

22.2.2.1 Fortran

The introduction of Fortran in 1956 was arguably the most significant step in the development of programming languages. "Fortran" stands for "Formula Translation," and the fundamental idea was to generate efficient machine code from a notation designed for people rather than machines. The model for the Fortran notation was what scientists and engineers wrote when solving problems using

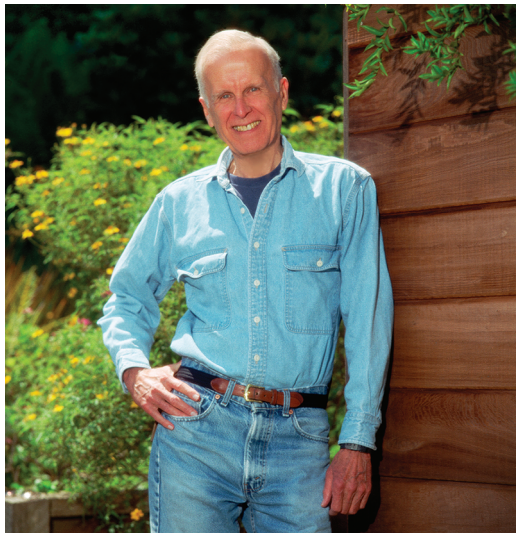
mathematics, rather than the machine instructions provided by the (then very new) electronic computers.

From a modern perspective, Fortran can be seen as the first attempt to directly represent an application domain in code. It allowed programmers to write linear algebra much as they found it in textbooks. Fortran provided arrays, loops, and standard mathematical functions (using the standard mathematical notation, such as **x+y** and **sin(x)**). There was a standard library of mathematical functions, mechanisms for I/O, and a user could define additional functions and libraries.

The notation was largely machine independent so that Fortran code could often be moved from computer to computer with only minor modification. This was a *huge* improvement over the state of the art. Therefore, Fortran is considered the first high-level programming language.

It was considered essential that the machine code generated from the Fortran source code was close to optimally efficient: machines were room size and enormously expensive (many times the yearly salary of a team of good programmers), they were (by modern standards) ridiculously slow (such as 100,000 instructions/second), and they had absurdly small memories (such as 8K bytes). However, people were fitting useful programs into those machines, and an improvement in notation (leading to better programmer productivity and portability) could not be allowed to get in the way of that.

Fortran was hugely successful in its target domain of scientific and engineering calculations and has been under continuous evolution ever since. The main versions of the Fortran language are II, IV, 77, 90, 95, 03. It is still debated whether Fortran77 or Fortran90 is more widely used today.



The first definition of and implementation of Fortran were done by a team at IBM led by John Backus: "We did not know what we wanted and how to do it. It just sort of grew." How could he have known? Nothing like that had been done before, but along the way they developed or discovered the basic structure of compilers: lexical analysis, syntax analysis, semantic analysis, and optimization. To this day, Fortran leads in the optimization of numerical computations. One thing that emerged (after the initial Fortran) was a notation for specifying grammars: the Backus-Naur Form (BNF). It was first used for Algol60 (§22.2.3.1) and is now used for most modern languages. We use a version of BNF for our grammars in Chapters 6 and 7.

Much later, John Backus pioneered a whole new branch of programming languages ("functional programming"), advocating a mathematical approach to programming as opposed to the machine view based on reading and writing memory locations. Note that pure math does not have the notion of assignment, or even actions. Instead you "simply" state what must be true given a set of conditions. Some of the roots of functional programming are in Lisp (§22.2.2.3), and some of the ideas from functional programming are reflected in the STL (Chapter 21).

References

Backus, John. "Can Programming Be Liberated from the von Neumann Style?" *Communications of the ACM,* 1977. (His Turing award lecture.)

Backus, John. "The History of FORTRAN I, II, and III." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 0521692695.

ISO/IEC 1539. *Programming Languages – Fortran*. (The "Fortran95" standard.)

Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0521390222.

22.2.2.2 COBOL

COBOL ("The Common Business-Oriented Language") was (and sometimes still is) for business programmers what Fortran was (and sometimes still is) for scientific programmers. The emphasis was on data manipulation:

- Copying
- Storing and retrieving (record keeping)
- Printing (reports)

Calculation/computation was (often correctly in COBOL's core application domains) seen as a minor matter. It was hoped/claimed that COBOL was so close to "business English" that managers could program and programmers would soon

become redundant. That is a hope we have heard frequently repeated over the years by managers keen on cutting the cost of programming. It has never been even remotely true.

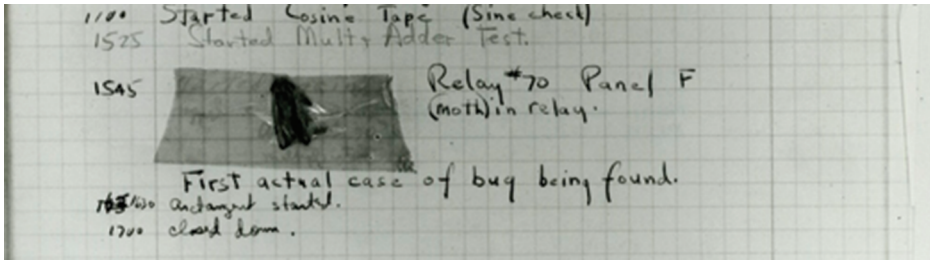
COBOL was initially designed by a committee (CODASYL) in 1959–60 at the initiative of the U.S. Department of Defense and a group of major computer manufacturers to address the needs of business-related computing. The design built directly on the FLOW-MATIC language invented by Grace Hopper. One of her contributions was the use of a close-to-English syntax (as opposed to the mathematical notation pioneered by Fortran and still dominant today). Like Fortran – and like all successful languages – COBOL underwent continuous evolution. The major revisions were 60, 61, 65, 68, 70, 80, 90, and 04.

Grace Murray Hopper had a Ph.D. in mathematics from Yale University. She worked for the U.S. Navy on the very first computers during World War II. She returned to the navy after a few years in the early computer industry:

"Rear Admiral Dr. Grace Murray Hopper (U.S. Navy) was a remarkable woman who grandly rose to the challenges of programming the first computers. During her lifetime as a leader in the field of software development concepts, she contributed to the transition from primitive programming techniques to the use of sophisticated compilers. She believed that 'we've always done it that way' was not necessarily a good reason to continue to do so."

–Anita Borg, at the "Grace Hopper Celebration of Women in Computing" conference, 1994

Grace Murray Hopper is often credited with being the first person to call an error in a computer a "bug." She certainly was among the early users of the term and documented a use:



As can be seen, that bug was real (a moth), and it affected the hardware directly. Most modern bugs appear to be in the software and have less graphical appeal.

References

A biography of G. M. Hopper: http://tergestesoft.com/~eddysworld/hopper.htm.

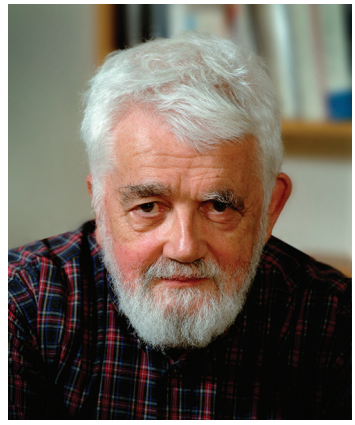
ISO/IEC 1989:2002. *Information Technology – Programming Languages – COBOL.*

Sammet, Jean E. "The Early History of COBOL." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

22.2.2.3 Lisp

Lisp was originally designed in 1958 by John McCarthy at MIT for linked-list and symbolic processing (hence its name: "LISt Processing"). Initially Lisp was (and is often still) interpreted, as opposed to compiled. There are dozens (most likely hundreds) of Lisp dialects. In fact, it is often claimed that "Lisp has an implied plural." The current most popular dialects are Common Lisp and Scheme. This family of languages has been (and is) the mainstay of artificial intelligence (AI) research (though delivered products have often been in C or C++). One of the main sources of inspiration for Lisp was the (mathematical notion of) lambda calculus.

Fortran and COBOL were specifically designed to help deliver solutions to real-world problems in their respective application areas. The Lisp community was much more concerned with programming itself and the elegance of programs. Often these efforts were successful. Lisp was the first language to separate its definition from the hardware and base its semantics on a form of math. If Lisp had a specific application domain, it is far harder to define precisely: "AI" and "symbolic computation" don't map as clearly into common everyday tasks as "business processing" and "scientific programming." Ideas from Lisp (and from the Lisp community) can be found in many more modern languages, notably the functional languages.



John McCarthy's B.S. was in mathematics from the California Institute of Technology and his Ph.D. was in mathematics from Princeton University. You may notice that there are a lot of math majors among the programming language designers. After his memorable work at MIT, McCarthy moved to Stanford in 1962 to help found the Stanford AI lab. He is widely credited for inventing the term *artificial intelligence* and made many contributions to that field.

References

Abelson, Harold, and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996. ISBN 0262011530.

ANSI INCITS 226-1994 (formerly ANSI X3.226:1994). *American National Standard for Programming Language – Common LISP*.

McCarthy, John. "History of LISP." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Steele, Guy L., Jr. *Common Lisp: The Language*. Digital Press, 1990. ISBN 1555580416.

Steele, Guy L., Jr., and Richard Gabriel. "The Evolution of Lisp." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices,* Vol. 28 No. 3, 1993.

22.2.3 The Algol family

In the late 1950s, many felt that programming was getting too complicated, too ad hoc, and too unscientific. They felt that the variety of programming languages was unnecessarily great and that those languages were put together with insufficient concern for generality and sound fundamental principles. This is a sentiment that has surfaced many times since then, but a group of people came together under the auspices of IFIP (the International Federation of Information Processing), and in just a couple of years they created a new language that revolutionized the way we think about languages and their definition. Most modern languages – including C++ – owe much to this effort.

22.2.3.1 Algol60

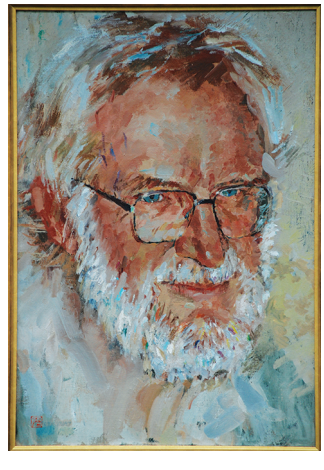
The "ALGOrithmic Language," Algol, which resulted from the efforts of the IFIP 2.1 group, was a breakthrough of modern programming language concepts:

- Lexical scope
- Use of grammar to define the language
- Clear separation of syntactic and semantic rules
- Clear separation of language definition and implementation
- Systematic use of (static, i.e., compile-time) types
- Direct support for structured programming

The very notion of a "general-purpose programming language" came with Algol. Before that, languages were scientific (e.g., Fortran), business (e.g., COBOL), list manipulation (e.g., Lisp), simulation, etc. Of these languages, Algol60 is most closely related to Fortran.

Unfortunately, Algol60 never reached major nonacademic use. It was seen as "too weird" by many in the industry, "too slow" by Fortran programmers, "not supportive of business processing" by COBOL programmers, "not flexible enough" by Lisp programmers, "too academic" by most people in the industry (including the managers who controlled investment in tools), and "too European" by many Americans. Most of the criticisms were correct. For example, the Algol60 report didn't define any I/O mechanism! However, similar criticisms could have been leveled at just about any contemporary language – and Algol set the new standard for many areas.

One problem with Algol60 was that no one knew how to implement it. That problem was solved by a team of programmers led by Peter Naur (the editor of the Algol60 report) and Edsger Dijkstra:

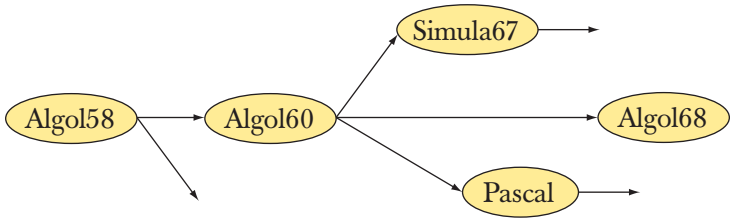


Peter Naur was educated (as an astronomer) at the University of Copenhagen and worked at the Technical University of Copenhagen (DTH) and for

the Danish computer manufacturer Regnecentralen. He learned programming early (1950–51) in the Computer Laboratory in Cambridge, England (Denmark didn't have computers that early), and later had a distinguished career spanning the academia/industry gulf. He was co-inventor of BNF (the Backus-Naur Form) used to describe grammars and a very early proponent of formal reasoning about programs (Bjarne Stroustrup first – in 1971 or so – learned the use of invariants from Peter Naur's technical articles). Naur consistently maintained a thoughtful perspective on computing, always considering the human aspects of programming. In fact, his later work could reasonably be considered part of philosophy (except that he considers conventional academic philosophy utter nonsense). He was the first professor of Datalogi at the University of Copenhagen (the Danish term *datalogi* is best translated as "informatics"; Peter Naur hates the term *computer science* as a misnomer – computing is not primarily about computers).

Edsger Dijkstra was another of computer science's all-time greats. He studied physics in Leyden but did his early work in computing in Mathematisch Centrum in Amsterdam. He later worked in quite a few places, including Eindhoven University of Technology, Burroughs Corporation, and the University of Texas (Austin). In addition to his seminal work on Algol, he was a pioneer and strong proponent of the use of mathematical logic in programming, algorithms, and one of the designers and implementers of THE operating system – one of the first operating systems to systematically deal with concurrency. THE stands for "Technische Hogeschool Eindhoven" – the university where Edsger Dijkstra worked at the time. Arguably, his most famous paper was "Go-To Statement Considered Harmful," which convincingly demonstrated the problems with unstructured control flows.

The Algol family tree is impressive:



Note Simula67 and Pascal. These languages are the ancestors to many (probably most) modern languages.

References

Dijkstra, Edsger W. "Algol 60 Translation: An Algol 60 Translator for the x1 and Making a Translator for Algol 60." Report MR 35/61. Mathematisch Centrum (Amsterdam), 1961.

Dijkstra, Edsger. "Go-To Statement Considered Harmful." *Communications of the ACM*, Vol. 11 No. 3, 1968.

Lindsey, C. H. "The History of Algol68." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

Naur, Peter, ed. "Revised Report on the Algorithmic Language Algol 60." A/S Regnecentralen (Copenhagen), 1964.

Naur, Peter. "Proof of Algorithms by General Snapshots." *BIT,* Vol. 6, 1966, pp. 310–16. (Probably the first paper on how to prove programs correct.)

Naur, Peter. "The European Side of the Last Phase of the Development of ALGOL 60." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Perlis, Alan J. "The American Side of the Development of Algol." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

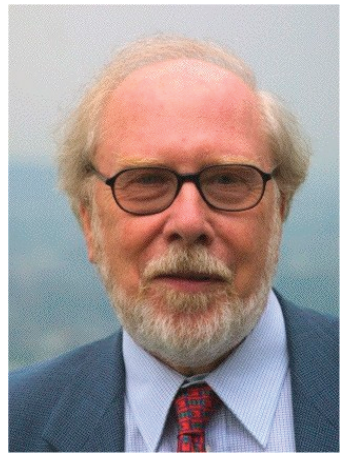
van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, eds. *Revised Report on the Algorithmic Language Algol 68* (Sept. 1973). Springer-Verlag, 1976.

22.2.3.2 Pascal

The Algol68 language mentioned in the Algol family tree was a large and ambitious project. Like Algol60, it was the work of "the Algol committee" (IFIP working group 2.1), but it took "forever" to complete and many were impatient and doubtful that something useful would ever come from that project. One member

of the Algol committee, Niklaus Wirth, decided simply to design and implement his own successor to Algol. In contrast to Algol68, that language, called Pascal, was a simplification of Algol60.

Pascal was completed in 1970 and was indeed simple and somewhat inflexible as a result. It was often claimed to be intended just for teaching, but early papers describe it as an alternative to Fortran on the supercomputers of the day. Pascal was indeed easy to learn, and after a very portable implementation became available it became very popular as a teaching language, but it proved to be no threat to Fortran.



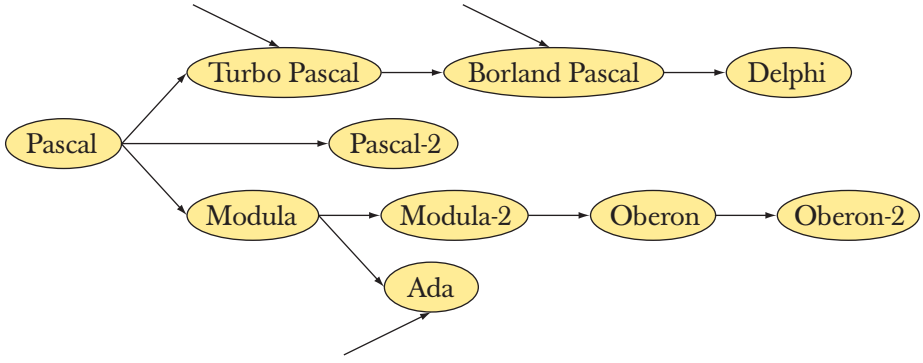
Pascal was the work of Professor Niklaus Wirth (photos from 1969 and 2004) of the Technical University of Switzerland in Zurich (ETH). His Ph.D. (in electrical engineering and computer science) is from the University of California at Berkeley, and he maintains a lifelong connection with California. Professor Wirth is the closest thing the world has had to a professional language designer. Over a period of 25 years, he designed and implemented

- Algol W
- PL/360
- Euler
- Pascal
- Modula
- Modula-2
- Oberon
- Oberon-2
- Lola (a hardware description language)

Niklaus Wirth describes this as his unending quest for simplicity. His work has been most influential. Studying that series of languages is a most interesting exercise. Professor Wirth is the only person ever to present two languages at HOPL.

In the end, pure Pascal proved to be too simple and rigid for industrial success. In the 1980s, it was saved from extinction primarily through the work of Anders Hejlsberg. Anders Hejlsberg was one of the three founders of Borland. He first designed and implemented Turbo Pascal (providing, among other things, more flexible argument-passing facilities) and later added a C++-like object model (but with just single inheritance and a nice module mechanism). He was educated at the Technical University in Copenhagen, where Peter Naur occasionally lectured – it's sometimes a very small world. Anders Hejlsberg later designed Delphi for Borland and C# for Microsoft.

The (necessarily simplified) Pascal family tree looks like this:



References

Borland/Turbo Pascal. http://en.wikipedia.org/wiki/Turbo_Pascal.

Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. *The C# Programming Language, Second Edition*. Microsoft .NET Development Series. ISBN 0321334434.

Wirth, Niklaus. "The Programming Language Pascal." *Acta Informatics,* Vol. 1 Fasc 1, 1971.

Wirth, Niklaus. "Design and Implementation of Modula." *Software–Practice and Experience,* Vol. 7 No. 1, 1977.

Wirth, Niklaus. "Recollections about the Development of Pascal." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices,* Vol. 28 No. 3, 1993.

Wirth, Niklaus. *Modula-2 and Oberon*. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. http://portal.acm.org/toc.cfm?id=1238844.

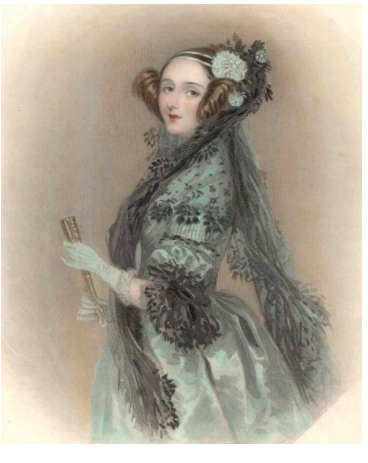
22.2.3.3 Ada

The Ada programming language was designed to be a language for all the programming needs of the U.S. Department of Defense. In particular, it was to be a language in which to deliver reliable and maintainable code for embedded systems programming. Its most obvious ancestors are Pascal and Simula (see §22.2.3.2 and §22.2.4). The leader of the group that designed Ada was Jean Ichbiah – a past chairman of the Simula Users' Group. The Ada design emphasized

- Data abstraction (but no inheritance until 1995)
- Strong static type checking
- Direct language support concurrency

The design of Ada aimed to be the embodiment of software engineering in programming languages. Consequently, the U.S. DoD did not design the language; it designed an elaborate process for designing the language. A huge number of people and organizations contributed to the design process, which progressed through a series of competitions, to produce the best specification and next to produce the best language embodying the ideas of the winning specification. This immense 20-year project (1975–98) was from 1980 managed by a department called AJPO (Ada Joint Program Office).

In 1979, the resulting language was named after Lady Augusta Ada Lovelace (a daughter of Lord Byron, the poet). Lady Lovelace could be claimed to have been the first programmer of modern times (for some definition of "modern") because she had worked with Charles Babbage (the Lucasian Professor of Mathematics in Cambridge – that's Newton's chair!) on a revolutionary mechanical computer in the 1840s. Unfortunately, Babbage's machine was unsuccessful as a practical tool.



Thanks to the elaborate process, Ada has been considered the ultimate design-by-committee language. The lead designer of the winning design team, Jean Ichbiah from the French company Honeywell Bull, emphatically denied that. However, I suspect (based on discussion with him) that he could have designed a better language, had he not been so constrained by the process.

Ada's use was mandated for military applications by the DoD for many years, leading to the saying "Ada, it's not just a good idea, it's the law!" Initially, the use of Ada was just "mandated," but when many projects received "waivers" to use other languages (typically C++), the U.S. Congress passed a law requiring the use of Ada in most military applications. That law was later rescinded in the face of commercial and technical realities. Bjarne Stroustrup is one of the very few people to have had his work banned by the U.S. Congress.

That said, we insist that Ada is a much better language than its reputation would indicate. We suspect that if the U.S. DoD had been less heavy-handed about its use and the exact way in which it was to be used (standards for application development processes, software development tools, documentation, etc.), it could have become noticeably more successful. To this day, Ada is important in aerospace applications and similar advanced embedded systems application areas.

Ada became a military standard in 1980, an ANSI standard in 1983 (the first implementation was done in 1983 – three years *after* the first standard!), and an ISO standard in 1987. The ISO standard was extensively (but of course compatibly) revised for a 1995 ISO standard. Notable improvements included more flexibility in the concurrency mechanisms and support for inheritance.

References

Barnes, John. *Programming in Ada 2005*. Addison-Wesley, 2006. ISBN 0321340787.

Consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995). *Information Technology – Programming Languages – Ada*, as updated by changes from *Technical Corrigendum 1* (ISO/IEC 8652:1995:TC1:2000).

Ada information page: www.adaic.org/.

Whitaker, William A. *ADA – The Project: The DoD High Order Language Working Group*. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices,* Vol. 28 No. 3, 1993.

22.2.4 Simula

Simula was developed in the early to mid-1960s by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Center and Oslo University. Simula is indisputably a member of the Algol family of languages. In fact, Simula is almost completely a superset of Algol60. However, we choose to single out Simula for special attention because it is the source of most of the fundamental ideas that

today are referred to as "object-oriented programming." It was the first language to provide inheritance and virtual functions. The words *class* for "user-defined type" and *virtual* for a function that can be overridden and called through the interface provided by a base class come from Simula.

Simula's contribution is not limited to language features. It came with an articulated notion of object-oriented design based on the idea of modeling real-world phenomena in code:

- Represent ideas as classes and class objects.
- Represent hierarchical relations as class hierarchies (inheritance).

Thus, a program becomes a set of interacting objects rather than a monolith.

Kristen Nygaard – the co-inventor (with Ole-Johan Dahl, to the left, wearing glasses) of Simula67 – was a giant by most measures (including height), with an intensity and generosity to match. He conceived of the fundamental ideas of object-oriented programming and design, notably inheritance, and pursued their implications over decades. He was never satisfied with simple, short-term, and shortsighted answers. He had a constant social involvement that lasted over decades. He can be given a fair bit of credit for Norway staying out of the European Union, which he saw as a potential centralized and bureaucratic nightmare that would be insensitive to the needs of a small country at the far edge of the Union – Norway. In the mid-1970s Kristen Nygaard spent significant time in the computer science department of the University of Aarhus, Denmark (where, at the time, Bjarne Stroustrup was studying for his master's degree).

Kristen Nygaard's master's degree is in mathematics from the University of Oslo. He died in 2002, just a month before he was (together with his lifelong friend Ole-Johan Dahl) to receive the ACM's Turing Award, arguably the highest professional honor for a computer scientist.

Ole-Johan Dahl was a more conventional academic. He was very interested in specification languages and formal methods. In 1968, he became the first full professor of informatics (computer science) at Oslo University.

In August 2000 Dahl and Nygaard were made Commanders of the Order of Saint Olav by the King of Norway. Even true geeks can gain recognition in their hometown!

References

Birtwistle, G., O-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Student-litteratur (Lund. Sweden), 1979. ISBN 9144062125.

Holmevik, J. R. "Compiling SIMULA: A Historical Study of Technological Genesis." *IEEE Annals of the History of Computing,* Vol. 16 No. 4, 1994, pp. 25–37.

Krogdahl, S. "The Birth of Simula." Proceedings of the HiNC 1 Conference in Trondheim, June 2003 (IFIP WG 9.7, in cooperation with IFIP TC 3).

Nygaard, Kristen, and Ole-Johan Dahl. "The Development of the SIMULA Languages." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

SIMULA Standard. *DATA Processing – Programming Languages – SIMULA*. Swedish Standard, Stockholm, Sweden (1987). ISBN 9171622349.

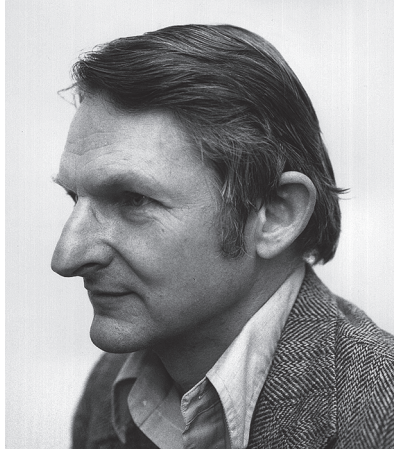
22.2.5 C

In 1970, it was "well known" that serious systems programming – in particular the implementation of an operating system – had to be done in assembly code and could not be done portably. That was much as the situation had been for scientific programming before Fortran. Several individuals and groups set out to challenge that orthodoxy. In the long run, the C programming language (Chapter 27) was by far the most successful of those efforts.

Dennis Ritchie designed and implemented the C programming language in Bell Telephone Laboratories' Computer Science Research Center in Murray Hill, New Jersey. The beauty of C is that it is a deliberately simple programming language sticking very close to the fundamental aspects of hardware. Most of the current complexities (most of which reappear in C++ for compatibility reasons) were added after his original design and in several cases over Dennis Ritchie's objections. Part of C's success was its early wide availability, but its real strength was its direct mapping of language features to hardware facilities (see §25.4–5). Dennis Ritchie succinctly described C as "a strongly typed, but weakly checked language"; that is, C has a static (compile-time) type system, and a program that uses an object in a way that differs from its definition is not legal. However, a C compiler can't check that. That made sense when the C compiler had to run in 48K bytes of memory. Soon after C came into use, people devised a program, called lint, that separately from the compiler verified conformance to the type system.

Together with Ken Thompson, Dennis Ritchie is the co-inventor of Unix, easily the most influential operating system of all times. C was – and is – associated with the Unix operating system and through that with Linux and the open-source movement.

For 40 years, Dennis Ritchie worked in Bell Laboratories' Computer Science Research Center. He was a graduate of Harvard University (physics); his Ph.D. in applied mathematics from Harvard University was never granted because he either forgot to or refused to pay a small ($60) registration fee.



In the early years, 1974–79, many people in Bell Labs influenced the design of C and its adoption. Doug McIlroy was everybody's favorite critic, discussion partner, and ideas man. He influenced C, C++, Unix, and much more.

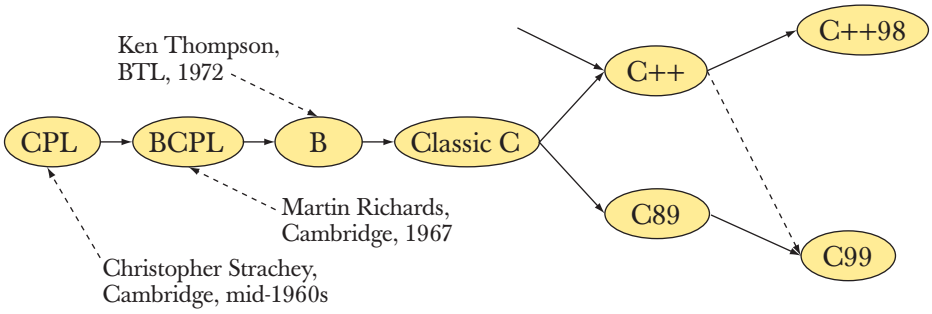
Brian Kernighan is a programmer and writer extraordinaire. Both his code and his prose are models of clarity. The style of this book is in part derived from the tutorial sections of his masterpiece, *The C Programming Language* (known as "K&R" after its co-authors, Brian Kernighan and Dennis Ritchie).

It is not enough to have good ideas; to be useful on a large scale, those ideas have to be reduced to their simplest form and articulated clearly in a way that is accessible to large numbers of people in their target audience. Verbosity is among the worst enemies of such presentation of ideas; so is obfuscation and over-abstraction. Purists often scoff at the results of such popularization and prefer "original results" presented in a way accessible only to experts. We don't: getting a nontrivial, but valuable, idea into the head of a novice is difficult, essential to the growth of professionalism, and valuable to society at large.

Over the years, Brian Kernighan has been involved with many influential programming and publishing projects. Two examples are AWK – an early scripting language named by the initials of its authors (Aho, Weinberger, and Kernighan) – and AMPL, "A Mathematical Programming Language."

Brian Kernighan is currently a professor at Princeton University; he is of course an excellent teacher, specializing in making otherwise complex topics clear. For more than 30 years he worked in Bell Laboratories' Computer Science Research Center. Bell Labs later became AT&T Bell Labs and later still split into AT&T Labs and Lucent Bell Labs. He is a graduate of the University of Toronto (physics); his Ph.D. is in electrical engineering from Princeton University.

The C language family tree looks like this:



The origins of C lay in the never-completed CPL project in England, the BCPL (Basic CPL) language that Martin Richards did while visiting MIT on leave from Cambridge University, and an interpreted language, called B, done by Ken Thompson. Later, C was standardized by ANSI and the ISO, and there were a lot of influences from C++ (e.g., function argument checking and **const**s).