Do general-purpose programming languages have a future?

Bjarne Stroustrup
Texas A&M University
(and AT&T Research)
http://www.research.att.com/~bs
Abstract

As the computing world matures, the roles of computer professionals are becoming more specialized. In particular, a programmer can spend a whole career doing work in embedded systems or data analysis without a need to gain expertise in other fields. Would such programmers be best served by completely different special-purpose languages? What are the fundamental and commercial factors that drive language evolution? What are the roles of program development environments, libraries, and tools? I think that general-purpose languages will have a key role in the programming world, but that the role will evolve and differ from what most people think of today. To make the discussion a bit concrete, I'll base some of my observations on examples from current C++ and its possible future developments.
Intellectual tradition

• My background (think Cambridge and Bell labs)
  – Pragmatic/empirical
    • Build the system the best you can, try it out, measure it, analyze it, fix it, then write about it
    • Primarily considers systems to be used by others
  – Idealistic
    • The best system should win
      – Even if it isn’t mine
    • There are usually many different criteria for “best”
      – Individual needs, taste, and opinions matter
    • There are unacceptable ways of winning
      – Lies (incl. gross exaggeration), money, many forms of marketing
My perspective

- Researcher
  - (Ideas and systems can be fascinating by all by themselves)
- Research manager
  - (yes, I can budget; I count costs and estimate economic benefits)
- Consultant
  - (usually unpaid – so I don’t have to tell people what they want to hear)
- Teacher
  - (mostly to professional programmers and managers)
- Academic
  - (that’s a recent development)
- My bias:
  - Applications with a high systems programming component
  - Industrial applications
  - Software is a very serious business
    - lives depend on software
    - key aspects of our civilization runs on software
Caveat

• My world view is heavily influenced by C++
  – And C++ reflects my world view in many ways

• I don’t consider C++ an ideal language
  – It’s a most useful language (http://www.research.att.com/~bs.applications)
  – I suspect I know its weaknesses as better than most people
    • E.g. irregular syntax and imperfect type system
  – No language is perfect

• I like programming languages
  – I don’t think there could be or should be just one language
  – or just one kind of language

• What we want/need is good software
  – A language is (just) a tool
  – There is no perfect language, and there never will be
Overview

• What is a general-purpose language?
  – And why would anyone care?
• What are the advantages of
  – Special-purpose languages?
  – General-purpose languages?
• Key examples
  – Object model
  – Container models
• Ideals for a general-purpose language
• How might this apply to C++?
Do general-purpose programming languages have a future?"

• For me, this is not just an academic question
  – Should I continue to work on C++?
  – Should I aim for generality?
  – Should I try to guide C++ into a (safe) niche?

(yes, yes, no)

• In general our answer has implications on
  – how we structure systems
  – what we teach
  – where we spend resources
    • Research
    • Tools
Programming languages

- For every one problem/purpose, the ideal language is a special-purpose one.
  - Examples:
    - Modeling mechanical systems (e.g. car engine, transmission)
    - Stereoscopic display of molecules
    - Engineering math (e.g. symbolic, numeric, visualization)
    - Video game engine (e.g. DOOM)
    - Graphical (e.g. GC)
    - Text manipulation (e.g. layout, analysis, transformation)
    - 2D and 3D Layout (e.g., architectural, chip design, graphics)
    - Graph computation (e.g. routing)
    - Expert systems (e.g. training simulators)

- We can’t always afford our ideals
  - So how can we best approximate them?
What can a language do for a programmer?

• No single language feature is essential
  – Lots of good programs have been written in languages deemed bad
    • C, Cobol, Fortran, …
  – Lots of projects have failed in languages proclaimed great
    • Most failing projects use a fashionable/popular language

• A language cannot
  – Prevent ill-conceived design strategies
  – Prevent ill-conceived implementation strategies

• A language can help a programmer to
  – express concepts directly
  – express independent concepts separately
  – in general
  – affordably
Why do we specialize languages?

• To radically simplify expression of ideas
• To provide stronger guarantees
• To make programming easier for
  – People who are not professional programmers
    • But understand an application domain far better than programmers and computer scientists
  – Novices (students)
    • This can be dangerous
  – The less smart and less highly educated
    • to be able to use more and cheaper programmers (this also can be dangerous)

• When done well, this necessarily limits the area of application
Problems with special-purpose languages

• By definition, an S-P language has an “edge” beyond which a problem cannot be expressed
  – So how do we reason about problems beyond the edge
    • You can’t reason without concepts, without a language
  – How do we extend the S-P language?
    • Modify compiler
    • Add new primitive
    • Link to program fragment written in another language
      – Another S-P language?
      – A low-level language (e.g. C or assembler)
      – A general-purpose language with a suitable library

• Some problems are messy
  – We don’t (yet) have a formal model that could be supported by a special-purpose (domain specific) language
What is a “general-purpose programming language”?

• Originally
  – Without specific restrictions of expressiveness or performance
    • “At least as expressive as Algol 60”
  – Without special facilities and restrictions for commercial or scientific programming
    • Not (just) COBOL
    • Not (just) Fortran
  – PL\1 was the original attempt to unify the programming world
    • Of course it simply added one more faction
      – Once in significant use, a language doesn’t die
    • And it wasn’t better than Fortran and COBOL in their core areas
      – So the special-purpose languages won round #1
Consider application areas
(We’re come a long way since the days of Algol 60)

• Fuel injectors
• Cell phones & systems
• PDAs
• Switching systems
• Games
• Individual business applications
• Database-based transaction systems
• Airspace control systems
• Expert systems
• Symbol manipulation
• Enterprise systems
• Data mining
• Scientific/numeric applications
• Parallel computing
• Missile guidance
• Robotics,
• Telemetry
• Speech recognition/analysis

• Compilers
• Natural language analyzer
• Image processing
• Image analysis
• Medical instrument control
• Payroll systems
• Billing systems
• Airline reservation systems
• Email systems
• Web browsers
• VLSI layout
• Chemical engineering process control
• Device drivers
• Electronic trading
• Engine control
• Graphics
• Geometric modeling,
• Operating systems
What is a “general-purpose programming language”?

- Do we have a general-purpose language?
  - Can a language be considered general-purpose if we can’t use it to write
    - a device driver?
    - an operating system?
    - a text analysis application?
    - a record processing database-intensive application?
    - an expert-system
    - symbolic manipulation application?
    - a web commerce application?
    - an engineering/numeric application?
- We have G-P languages
  - in the sense that we can use them for all such purposes
- We don’t have a G-P language
  - in the sense that a language is a close-to ideal for all such purposes
  - A G-P language is at best the second choice for any one application
What’s right about a G-P language?

• You can do everything in it
  – You can do any two tasks in it
    • And that’s by definition rarely the case for a special-purpose language
  – You can with a high probability collaborate with someone in a different field
    • Share source or link

• But
  – Doing anything without proper libraries is painful
    • Getting libraries from different producers to work together in non-trivial

• A general-purpose language rely on abstraction where special-purpose languages rely on built-in specialized features
  – To improve a general-purpose language, we must strengthen its abstraction mechanisms
There will always be many languages

• Significant systems rely on code written in many languages
• Not just legacy code
  – There are hundreds of millions of lines of code “out there”
  – “legacy code” approximately means “code that’s being used”
• Programmers are often more important than code
  – And programmers differ in their preferences of languages, tools, and programming styles

• A general-purpose language must enable (and preferably encourage and ease) interoperability
Which G-P languages do we currently have?

- Candidates
  - Ada, C, C++, C# (?), Java (?), ML (?), Pascal(?)
  - ...
- Not candidates
  - PERL, Visual Basic, Python
  - COBOL, Fortran
  - ...
- There are N*1000 languages
  - Domain specific
  - Dead
  - Unsupported
  - Academic
  - Platform specific
  - Proprietary
  - ...
Do G-P programming languages have a future?

• Of course, but should they have?
  – yes

• Just for “legacy code”?
  – no

• Would the world be better without them?
  – No, we can’t manage with just special-purpose languages
    • Explorations of new/immature fields
    • Implementation of special-purpose languages
    • As “glue” for special-purpose languages

• Should we try to improve them?
  – Yes, none is anywhere perfect

• What is it about G-P languages that we might improve?
  – Abstraction facilities
  – Interoperability
  – Performance
Can you restrict programming style?

- Type safety is good
  - Not a restriction except when dealing with hardware
  - Complete type safety implies garbage collection
    - for some degree of generality
- Forcing “object orientation” has been a failure
  - “methods” that can’t be overridden
  - “methods” that doesn’t operator on an object
  - Classes have been successful as modules, though
- Strongly condemned features are making a comeback
  - Overloading
  - Generic programming
  - Nested classes / events
  - Multiple inheritance
  - Static type checking
  - Value types
Programming Styles (paradigms)

• A G-P language **will** be used for different paradigms
  – “C style” (“Pascal style”)
    • Procedures, structures, pointers
  – Data abstraction
  – Object-Oriented Programming
  – Generic Programming
  – Constraints
  – Logical
  – Rule-based
  – Aspect-oriented
  – ...

• A general-purpose language needs broad support for paradigms
  – Multi-paradigm programming
Languages “stretch”

• “C++ is a stretch language”
  - Peter Deutch (it was not meant to be a compliment)

• All languages “stretch” to serve a larger user community
  - By serving A and B you server both A and B better than just
    serving A or B
  - Need to serve related uses
  - Need to help users meet new challenges
  - Applying lessons of experience
  - Pressure from other languages

• Languages never shrink
  - Older language have many features
    - Some mainly for historical reasons (compatibility)
      - warts
    - Typically offer several ways of doing something
  - All languages are older when they become mainstream
Languages “stretch”

- Classes, inheritance, exceptions, generics, abstract classes, overloading, value types, properties, reflection, type safety, garbage collection, modules, etc.
- **Simplified** chart: C, C++, and Java have evolved significantly
- Critical design points:
  - how to handle hardware
  - How to handle performance needs
Is C a G-P Language?

• No
  – It’s a low-level language
  – It offers hardly any type safety
  – It offers no advanced features
  – It offers no specific abstraction mechanisms

• Yes
  – It is used for a wider range of applications than any other language
    • Except C++
  – It offers practical portability
  – It runs on essentially every platform
  – It offers performance that allows programmers to compensate for lack of advanced features
  – It interoperates with essentially all languages
Is Java a G-P language?

• No
  – It’s an Object-oriented language 😊
  – It can’t handle low-level systems programming
  – It can’t handle high-performance computing
  – It’s strengths comes partly from restriction
  – It’s a platform
    • Give up portability and you can handle a wider range of applications

• Yes
  – It can do more than Algol60 😊
  – It can handle an huge range of applications “well enough”

• It is becoming a stretch language
  – Several “editions” to increase its range of underlying systems
    • At the cost of portability
  – Many new language features over the years
Is it fair to consider performance?

• Yes, performance often matters
  – Commerce: amazon, google, Amadeus, …
  – Images: medical, movies, games, …
  – Gadgets: cell phones, fuel injectors, …
  – Scientific computation: protein folding, heat transfer, weather forecasting, …
  – Data management: mining, data capture, real time analysis (e.g. fraud detection, monitoring), DBMS, …

• Naturally, performance isn’t always important
  – Often, it is not
  – But I can see echo delays in some modern single-user text processing systems running on a GHz machine
    • (I consider that a disgrace)
A general-purpose language must be efficient:

- In time
- In space
- Where needed
- Predictably
- Portably

(this is a very tough challenge)
Object models

- Primitive object
- Composite object
- Object on heap
- Polymorphic object

• Consider
  - Fortran, C, C++, Java, C#
  - Interoperability
  - Hardware access

Type info
Object model – C “container”

```c
struct Cmplx { double re, im; };
struct Cmplx a[MAX] ;
struct Cmplx *p = a;
Struct Cmplx *q = malloc(sizeof(struct Cmplx)*MAX);
```

- Can address specific hardware locations directly
  - bytes, half-words, words, double words, etc.
- Can match externally imposed layout exactly (bit fields)
- Explicit management of heap (2 words per array overhead)
Object model – C++ container

class complex { double re, im; public: /* operations and operators */ }; vector<complex> v;

- Like C plus abstraction
  - Enables, but doesn’t require run-time range checking
- User-defined types are fundamentally similar to built-in types
- 4 words per vector overhead
Object model – Java/C# container

class Complex { double re, im; /* operations and operators */ };  
Complex[] v = new Complex[2];  
v[0] = new Complex(1,2);

- References plus data on (garbage collected) heap
- Built-in types differ from user-defined types
- 2 words per vector plus 2 words per element overhead
Key example: Container access

• Most basic: C array
  – (support for contiguous sequence of element of built-in types;
    that’s what traditional hardware supports)

```c
int a[4];       // holds objects directly
a[2] = 7;
int x = a[2];

struct My_type p = 0;
void* aa[4];    // indirection needed for polymorphism
aa[2] = p;
p = (struct My_type*) aa[2];    // explicit, efficient, and
                                 // unchecked conversion (unsafe)
```
Key example: Container access

• Container of general ("universal") object
  – Java, C#, (and C++ if you really want to)
    ```java
    int[] a = new int[10]; // special case for Array and small built-in types
    a[5] = 5;
    int x = a[5];
    ```
    ```java
    ArrayList aa = new ArrayList(10); // container of references to objects
    aa[5] = new My_class(3);
    My_class v = (My_class)aa[5]; // explicit run-time check
    – That cast is ugly, expensive, and often logically unnecessary
    ```
Key example: Container access

- Typed container
  - C++ (and soon Java and C#)
    // no special case for small built-in types (in C++ at least)
    Vector<My_class> vmc[10]; // state the element type explicitly
    vmc[5] = My_class(3);
    My_class v = vmc[5];
  - C++
    - no run-time test
    - elements are access directly (store pointer if you want indirection)
  - C#, Java:
    - implicit (expensive) run-time test
    - elements are still stored indirectly
Roles for a general-purpose language

- Language for writing libraries
- Language for writing messy application parts
- Language for writing performance critical application parts
- Target for code generation
- Low-level glue language (e.g. C, unsafe, fast)
  - As opposed to scripting languages
- Higher-level glue language (e.g. Java, safe, slow)
- Language for writing complete applications (?)
  - Only through libraries, increasingly through libraries
- Teaching language (?)
  - It is much easier to teach a simplified language
  - Where, when, and how do you learn about real-world problems and constraints?
Can a general-purpose language be completely type safe?

- Depends on your definition
  - Strictly-speaking: No
  - But complete type safety is an advantage for a **huge** range of uses
  - Probably unfair to deem a language that has a large stretch “not general-purpose”

- We need to improve interoperability between type safe and (typically unsafe) low-level languages
  - Verification/proof techniques
  - Clear (and non-proprietary) interfaces
  - Clearly declared unsafe program areas (like Modula-3)
  - ...
Ideals for a G-P language

- Simplicity
  - Incl. teachability
- Precise specification
- Easy to analyze
- Run-time performance
  - Uncompromising
- Ability to run everywhere
  - And take advantage of local facilities
- Type safety
  - And a facility to do type-unsafe operations
- Extensibility
  - Good abstraction facilities
- Ability to interoperate
  - With code from different implementations
  - With code from different languages
Can any of this be used to improve C++?

- C++0x is being prepared by the ISO C++ committee
  - Plus national representatives, of course
  - Design by committee is a horror
    - Committees don’t have an overall aim/”vision”
    - (some) Individuals do (and they don’t agree)
    - Compromises are needed
      - “a language good enough for everyone and ideal for none”
  - Only a committee can deal with an established mainstream language
    - “the ISO committee process is the worst, except for all the alternatives” (with apologies to W. Churchill)
Overall Goals

• Make C++ a better language for systems programming and library building
  – Rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows application development)
  – Maintain the zero-overhead principle

• Make C++ easier to teach and learn
  – Through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)
  – Through better libraries
So, do general-purpose programming languages have a future?

• Yes
  – And we still have a long way to go to meet obvious ideals
    • Type safety
    • Elegant and general abstraction
    • Performance
    • Interoperability
    • Teachability
    • Regular syntax and semantics
  – Look to C, C++, C#, Java
    • That’s where the major use that shapes demands will be
    • Ideas can come from experimental languages