

C

Designed in 1972. Simple language, that creates efficient code.

Two levels of variables, **global** and **local**. (Avoid using global)

Emphasis on use of pointers: Arrays are pointers, pointers are used instead of reference passing.

Introduces { } instead of **begin,end** and =,== instead of :=,=.

Small language, big library approach.

C

The *C* language was developed between 1969 and 1973 by Dennis Ritchie. The Unix operating system was written in it.

It was made popular by the book 'The C Programming Language' by Brian Kernighan and Dennis Ritchie. I believe the reasons of its success were the following:

- The book was very well written.
- The idea of having a small core language with big libraries was good.
- *C* was complete, and standardized. It had separate compilation, file handling. A preprocessor (for switching code on an off, having different implementations on different machines.) Competing languages (Pascal, Algol) had incomplete definition.
- The execution model was attractive for those who understood

how the machine works. (Much more attractive than Pascal and Algol.) It has only two levels of variables, global and local. It supports recursion in the local variables.

Simula

Introduced **objects**, **classes** and **inheritance**.

Developed in 1960s for simulation. Ability to define objects (that represent objects in the real world).

History of C++

- Bjarne Stroustrup started thinking about 'C with classes' in 1979, after having completed this PhD thesis. In PhD project he had to simulate distributed software.

He wrote a simulator for this in Simula, which he liked, which was however too slow to be practical. He rewrote the program in BCPL (a predecessor of *C*.) It was fast enough but ugly.

In the *C* with classes project he tried to combine the speed of C with the niceness of Simula.

- 1983. The language was called *C++*.
- 1985. First public release. There are 500 users.
- 1990? Start of standardization committee.
- 1991. There are 400,000 users.

Users of C^{++}

It is very hard to estimate how many people use a language, and how many people do this by choice.

Bjarne Stroustrup claims that the number of users probably decreased between 2002-2004, and increased between 2005-2007.

Also, nobody seems to have an idea how much code is being written in C^{++} .

isocpp.org/ claims that there 'are more than 3 million' C^{++} programmers in the world, and that the number is increasing.

Since 2010, C^{++} becomes more important because energy use is becoming a factor (battery life in mobile applications, and power use in data centers.)

Important Features of C^{++} .

- C^{++} is in the public domain. It is not protected by copyright, or patents. It compiles into native machine code. No special environment is needed to run a C^{++} program after it has been compiled. It is not connected to a particular operating system.
- Extensions (there are only extensions) are democratically decided by the ISO standard committee. The committee is big and varied enough to be immune to one-sided commercial interests. Possible changes are discussed very long and thorough.

C^{++} is Fairly Efficient

Comparing speed of programming languages is harder than it seems. Comparing benchmarks does not answer this question.

In order to compare language A to language B , one must compare two realistically sized programs with the same functionality, the first one well-designed and well-written by good programmers of language A with realistic effort, and the second one well-designed and well-written by good programmers of language B with realistic effort.

Most benchmarks are ugly in one or both of the languages they try to compare, and too small. Usually, too much effort has been put in writing them. Because of this, benchmarking is useless.

Choice of language tends to affect functionality, even if one tries to avoid it.

Still, I never heard complaints about the speed of C^{++} programs.

C^{++} is Level Increasing

It is often said that C^{++} is low level. (Close to the machine, you have to worry about cleaning up resources, about pointers, etc.)

This opinion is based on the fact that C^{++} derives from C , which is a portable assembly language.

What makes a language high-level? \Rightarrow The fact that complicated objects (vectors, hash maps, lists, big numbers, trees) are built-in to the language, and that you can use them without having to know how they work internally.

In C^{++} you can define such objects, and after that use them, as if they are part of the language. There is no distinction between a thing being part of the language, or defined in a library.

People who write big, low level programs in C^{++} are doing something wrong.

Any language with good extension features will always win in the long term from a language which has some built-in features that you like.

Difference in Culture

Some languages (Java and $C^\#$) have strong marketing teams behind them, which are able to write things like this: (Google: Ten Reasons Java Has Supplanted C++.)

C^{++} doesn't have a strong, powerful organization behind it. As a consequence, it is not strongly marketed, and standardization is slower than in other languages.

Graphics, sound processing, big numbers, downloading web pages, are available, but not standardized.

Opponents of C^{++} use this to say ' C^{++} doesn't have ...'.

Main Difference between C^{++} and Java (Python) : Value Semantics

C^{++} is based on **value semantics**. This means:

- If there are two variables in a part of the program, these variables are independent.
- If you have two different elements at different positions in an array, then these elements are independent.
- Different fields of a struct/class are independent.

Two things are independent if changing one of them does not change the other.

Alternatively, one can say: In C^{++} , every value uniquely belongs to a live variable.

This makes it meaningful to say things like 'object X belongs to container Y', and to speak about ownership.

Value Semantics (2)

Value semantics is a good thing:

1. It corresponds to physical reality, where things are at one place, and changing things at one place does not affect things at other places. Food that is in your refrigerator is not in the shop anymore. If you cook it at home, it won't be cooked in the shop.
2. It can be mathematically modelled. Side effects are very difficult to model.

Mathematical Modelling

Let p be an array of X . Call:

```
function( p[i] );
```

Let p' be the new version of p . With value semantics, the following formula is true:

$$\forall j \ (0 \leq j < \text{sizeof}(p)) : \ i \neq j \rightarrow p'[j] = p[j].$$

Without value semantics (in Java) all bets are off.

Similar formulas can be written for fields of structs, and for different variables in a program.

This means that a well-written C^{++} is closer to functional (side effect free) style than a Java program.

Java

Java (and Python) does not have value semantics. Instead it has **reference semantics**. Consider the following code, which builds a menu:

```
public class Selection
    implements java.awt.event.ActionListener
{
    String s;

    public void actionPerformed(
        java.awt.event.ActionEvent event )
    {
        s = event. getActionCommand( );
    }
}
```

```
java.awt.Frame fr =  
    new java.awt.Frame( "Game of Life" );  
fr. add(f);  
    // Some Layout Options omitted.  
  
Selection sel = new Selection( );  
java.awt.MenuBar mb = new java.awt.MenuBar( );  
fr. setMenuBar(mb);  
  
java.awt.Menu m1 =  
    new java.awt.PopupMenu( "Pattern Select" );  
mb. add(m1);
```



```
java.awt.MenuItem it1 =  
    new java.awt.MenuItem( "Glider Gun" );  
m1. add( it1 );  
it1. setActionCommand( "glidergun" );  
it1. addActionListener( sel );  
  
java.awt.MenuItem it2 =  
    new java.awt.MenuItem( "Four Glider Guns" );  
m1. add( it2 );  
it2. setActionCommand( "fourgliderguns" );  
it2. addActionListener( sel );  
  
java.awt.MenuItem it3 = new java.awt.MenuItem( "Relay" );  
m1. add( it3 );  
it3. setActionCommand( "relay" );  
it3. addActionListener( sel );
```

The MenuItems `it1`, `it2`, `it3` are modified after they have been added to `m1`. This has effect on `m1`, and also on `mb`.

Any input from the user, will magically appear in the local variable `sel`. We see that in Java, the abandoning of value semantics is not just a concession to efficiency, but rather an essential part of the way information is being transferred in Java.

Containers

```
public static void main(String[] args)
{
    int x[] = new int[ 10 ];

    for( int i = 0; i < x. length; ++ i )
        x[i] = i;

    int [] y = x;
    y[5] = 12;

    for( int i = 0; i < x. length; ++ i )
        System. out. println( "" + i + " : " +
                               x[i] + " " + y[i] );
}
}
```

Same Example in C++

```
#include <iostream>
#include <vector>
int main( int argc, char *argv[] ) {
    std::vector<int> x(10);

    for( int i = 0; i < x.size( ); ++ i )
        x[i] = i;

    std::vector<int> y = x;
    y[5] = 12;

    for( int i = 0; i < x.size( ); ++ i )
        std::cout << i << " : " << x[i] << " " << y[i] << "\n";
    return 0;
}
```

Different Default Behaviour

- In Java, the default behaviour of assignment and parameter passing is lazy. Copying can be obtained by the `.clone()` method.

Lazy assignment results in shared ownership. Shared ownership requires garbage collection.

- In C^{++} , the default behaviour of assignment and parameter passing is to copy. Copying can be costly for large objects. In order to avoid this, a special variable type, called **reference** was introduced. References are short-lived, sharing variables. They don't introduce need for garbage collection.

Some General Advice on Programming

It is very very difficult to measure productivity: Every measure based on size of output will encourage repeated code.

One should always look for repeating patterns in code, and try to make these patterns into separate classes or functions.

Don't worry about efficiency of abstraction. That is the task of the compiler builder. It is not your job.

Such extracted components have a higher probability of being reusable. Try to make them nice.

Code must be used immediately after it is written.

Not only tested, but also used.

Most code has unpleasant interface in the first version.

If you write code, and keep it for later (or make a nice library), you will probably have to modify the interface when you use it for the first time.

Don't write code that you may possibly need later. Write only things that you (or somebody else) needs now.

Don't give up generality for efficiency. You run the risk of writing code that very efficiently does something that you or the user does not want.

Testing

It turns out that not every student knows how to test code. All code must be tested immediately when it is written.

Use print statements to check if variables have the values that you expect them to have.

Make sure that every part of your code has been executed, **if** statements must have gone both ways, **while** loops must have been repeated 0,1,2 times.

Small components are easier to test than big blocks of code. (Because the interfaces are cleaner, and you can call them separately.)

Don't worry about the efficiency of introducing many small components. That is the task of the compiler builder.

Summary

C^{++} is based on value semantics. Java aims at reference (sharing) semantics. It requires a different style of programming.

C has a problem with objects of unpredictable size. You see this immediately with strings and arrays.