ALGOL

ALGOrithmic **L**anguage dates back to 1950s.

It features structured programming, i.e. using **while** instead of **goto**, and using **begin end** blocks instead of **goto**s that skip parts of code.

Introduced the Backus-Naur form for describing syntax.

Introduced Local Variables, which enable recursive programming.

C

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

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

Use of pointer instead of reference passing.

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

Small language, big library.

File management and memory management were part of the standard.

Simula

Introduced objects, classes and **inheritance**.

Developed in 1960s for simulation.

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 practial. 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 voluntarily.

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^{++}.

It is not clear if the number of C^{++} users is increasing or decreasing at this moment.

I have no idea. In all cases, it is one of the main programming languages.

Role of University

Universities fail at teaching C^{++}. This has impact on industry when a programming language for some project is chosen.

My experience when talking with colleagues from university is, that they seriously do not like C^{++}.

Many people who work at universities have strong, pseudo-religious views about programming (it has to be functional, based on a mathematical formalism, a programming language must have a precise formal semantics, must be fully machine independent.) Yet, these people don't program by themselves.

If you want to make a university career, you must publish about theory of programming, instead of writing programs.

It is always nicer (and easier) to preach revolution than to do something concrete.

There is a a self-confirming cycle:

C^{++} is being taught badly $\Rightarrow$ students write bad C^{++} code (share this code, put it on the web) $\Rightarrow$ professors see this code, don't like it $\Rightarrow$ professors who could understand good-style C^{++} are reinforced in their belief that it is a terrible language, and leave teaching it to others.

General 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 enviromnent 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 immume to one-sided commercial interests. Possible changes are discussed very long and thorougly.

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 the culture of language A with realistic effort, and the second one well-designed and well-written by the culture of language B with realistic effort.

Most benchmarks are ugly in one or both of the languages they try to compare, and much 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 anyone complain 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? 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 10 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, unicode characters.

All of this is available, but it is not standardized. Opponents of C^{++} use this to say 'C^{++} doesn't have ...'.

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) were incomplete.
- 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.

C-Execution Model

C was designed to be a portable assembly language. This means that mapping to machine instructions should be easy.

A computer consists of the following parts:

Memory stores words (usually bytes) at certain addresses. The CPU can ask the memory for the value at a certain address (read), and it can ask the memory to store a value at a certain address (write). Such action usually takes 7.5 nanoseconds.

Registers store data inside the CPU. The CPU has not many registers, (16?), and they have designated functions: There may be int registers (4 bytes), float registers (6 bytes), double registers (8 bytes), and address registers (4-8 bytes).

Reading and writing into CPU registers is very fast (350 picoseconds) and the CPU can access multiple registers at the same time, while performing an instruction.

Distribution of the Memory

During execution, three areas of memory are assigned to the program.

1. Space for program itself. The CPU has one register, the program counter, PC that indicates the current point (address) where the program is. After excuting the instruction, the PC is increased to point to the next instruction. (Unless it was a goto or a branch. In that case, the new address is loaded into the PC.)
2. A stack of local variables. This stack usually grows downward. Local variables in a function or procedure are allocated on this stack. Whenever a function or procedure is called, the current PC is stored on this stack as well.
3. A designated area for global variables.

In addition to the three areas above, the program can ask the OS for blocks of memory. It is the responsibility of the program (or its author or the compiler) to return all memory that is not needed anymore, back to the system.

Forgetting to do this results in a situation where a program blocks much more memory than it is actually using. If a program is not well-written, it is quite possible that it uses up all memory in the computer, without really using it.

Such situation is called memory leak. It happens quite often. (Also with disk space.) We will hear a lot more about this delicate problem later in the course.

Lots of professionally written software has memory leaks.

Types of Operands (From Operands)

I am not trying to describe particular processor, only giving a general impression of the form of the type instructions that a typical CPU has.

A from operand has one of four possible forms:

- Immediate: The value is given as a sequence of bytes.
- Direct: The value is the contents of a register R_i.
- Indirect: The contents of a register R_i is interpreted as address, and the value is read from this address. This is written as $[R_i]$.
- Indexed indirect: Some constant c is added to the contents of a register R_i. The result is interpreted as address. The value is read from this address. This is written as $c[R_i]$.

Types of Operands: To Operands

A to operand has one of three possible forms:

- Direct: The result is written into a register R_i.
- Indirect: A register R_i is interpreted as address, and the value is written into this address. This is written as $[R_i]$.
- Indexed indirect: Some constant c is added to the contents of a register R_i. The result is interpreted as address. The value is written into this address. Indexed indirect operands are written as $c[R_i]$.

Real CPUs have more addressing modes. Sometimes, not all registers can be used in all addressing modes.

Types of Operands (Labels)

A label is a reference to a point in the program.

It has has one of two possible forms:

- A relative label gives the position relative to the current position in the program.
- A absolute address gives the position as address.

Relative labels have the advantage that the code is relocatable. A relative label remains valid when the point where it is used it moved, together with the point where it refers to. Relative labels are usually used in inside procedures.

Absolute labels are used for procedure calls.

Instructions

- Let OP = ADD,SUB,MULT,DIV,MOVE,CMP. Let S = INT/DOUBLE/BYTE. Then OPS (from), (to) is an instruction.
- Let OP = IFZERO, IFNONZERO, IFGEQZERO, IFLESSZERO. Then OP (label) is an instruction. The condition always refers to the result of the previous statement.
- Let S,T = INT/DOUBLE/BYTE. Then CONVST (from), (to) is an instruction.
- CALL (label) is an instruction. It writes the current value of PC into the stack and continues at (label).
- RETURN is an instruction.

The stack pointer (SP) is just one of the registers.

```
double fact( int i )
{
   double res = 1.0;
   while( i != 0 )
   {
      res = res * i;
      i = i - 1;
   }
   return res;
}
```

Before we can translate anything into machine code, we need a calling convention: Parameters to functions are passed in register `R0`. Results are returned in register `R1`.

```
fact: // We assume that i is in R0.
   SUBINT 8, SP;   // Make place for a double on stack.
   MOVEDOUBLE 1.0, [SP]; // Write 1.0 into it.
loop:
   CMPDOUBLE 0,R0;  // Compare i to 0.
   JUMPIFZERO end;   // Applies to previous operation.
   CONVINTDOUBLE R0, R1;
   MULTDOUBLE R1, [SP];
   SUBINT 1, R0;
   GOTO loop;   // Relative.
end:
   MOVEDOUBLE [SP], R1
   ADDINT 8, SP;  // Variable res goes out of scope.
   RETURN;   // Result is in R1.
```

```
{
   int i = 4;
   int j = 5;
   int k = i + j;
   ...
}

SUBINT 4, SP;     MOVEINT 4, [SP];
SUBINT 4, SP;     MOVEINT 5, [SP];
SUBINT 4, SP;
MOVEINT 8[SP],R0;
ADDINT 4[SP],R0;  MOVEINT R0, [SP];
...
ADDINT 12, SP;   // When the block is exited.
```

What does the C-compiler do for us:

- In case of $-, *, =, != $, it figures out which type of instruction to use (DOUBLE or INT). In the first example, it inserted the conversion CONVINTDOUBLE.
- It choose registers for the intermediate results.
- In ensures that we don't mess up the stack. When there is more than one variable, it keeps track of the relative positions of the variables in memory. It knows how much space should be allocated for a variable.

Observations

So now you understand why you have to declare variables in *C*. The compiler needs this information.

Local variables of variable length are problematic. (Strings, Inheritance, Arrays of Variable Length.)

```
#define MAXNAME 100
char name [ MAXNAME ];
print( "Hi, what’s your name?" );
scanf( "%s", name );
   // Lot of space being wasted, and possibly still not enough
```

Main Difference between C^{++} and Java: 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 indepent if assignment to one of them does not change the other.

Alternatively, one can say: In C^{++}, objects stand at a unique place.

This makes it meaningful to say things 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.
2. It can be mathematically modelled.

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:

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

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 );

jawa.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 apear 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 veriable type, called reference has been introduced. References are short-lived, second class variables, so there is no 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 invite 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. That is the task of the compiler builder.

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.

If you write code, and keep it for later (or makes a nice library), in most cases the interface is wrong, and you notice this only when you want to use the code.

Don’t write code for 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.

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.)