SFU CMPT-212 2008-1 Topic: Introduction

Ján Maňuch

E-mail: jmanuch@sfu.ca

Thursday 10th January, 2008

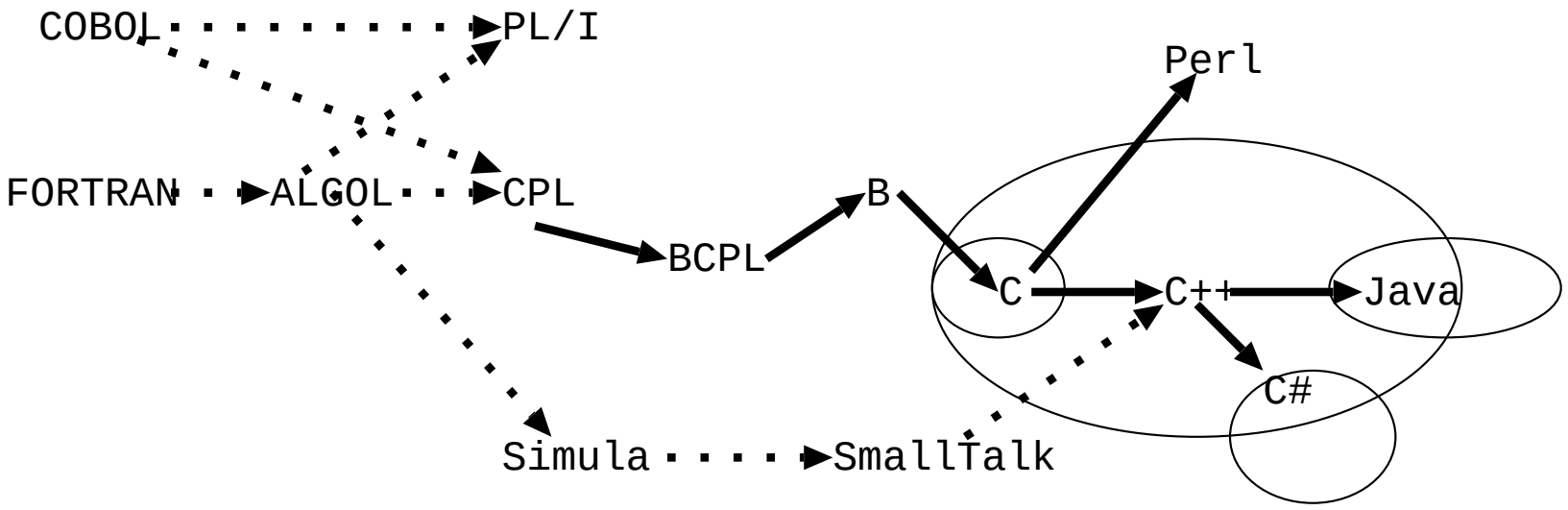
Programming languages

- Assembler - language very close to machine code; not portable at all
- **non-structured language** (FORTRAN, BASIC): use branching statements:

  ```
  IF (A<B) GOTO LAB1 ELSE GOTO LAB2
  ```

  — very difficult to understand what programs does, and almost impossible to modify it
- **structured language** (ALGOL, . . . , C, . . .): limits branching to a small set of well-behaved constructions (`if else` statements, loop statements); basic building block: functions

C++ History


COBOL
PL/I
Perl
FORTRAN
ALGOL
CPL
B
BCPL
C
C++
Java
C#
Simula
SmallTalk

Why C was created?

- a “system programming language” – designed for implementing UNIX systems on various machines
- requirements:
  - simple: the memory was very limited, needed a simple and compact compiler
  - close to machine instructions (capabilities) – very efficient
  - portable (simple – complilers could be easily and quickly rewritten for a new machine)
  - at the same time powerful – for more complex tasks (I/0, string handling, etc) there are libraries which can be loaded when needed
- it turned out that C was a general-purpose programming language useful for various tasks (numerical, text-processing, database programs, etc), hence it became widely used

C: main characteristics

- "Structured" language.
- Power and Simplicity
- Platform independence (at the level of source code) $\Longrightarrow$ portability.
- Close to the operations of most computers $\Longrightarrow$ leading to efficient implementations, allowing high performance code.
- Small size of the language (small core, with extensible libraries, "standard libraries").
- Open standard
  - first standard: "classic C" or "K&R" C described in the book [Kernighan, Ritchie (1978)]
  - ANSI/ISO committee (est. 1983), first standard in 1989, latest 1999

C++ History

- C++ is an extension of C, including more modern programming concepts, especially object-oriented programming and generic programming.
- Foundations:
  - SIMULA (1967), a direct extension of ALGOL, for writing simulations; the first language with object-oriented features `http://www.acm.org/announcements/turing_2001.html`
  - Smalltalk (1980) – real OOP language, but: interpreted language (executes slowly)
- C with Classes (Bjarne Stroustrup, 1980) – direct extension of C
- C++ (1983-84) – added virtual functions and operator overloading, documented in 1985 [Stroustrup, 1997 (3rd edition)]

So what is OOP?

- Procedural Programming: Decide which procedures you want; use the best algorithms you can find.
  Top-down design: breaking the main problem into smaller tasks.
- Object-Oriented Programming:
  - design of programs is shifted from the design of procedures towards the organization/representation of data
  - an *object* is a model of real or imaginary thing with which the program deals
  - the computer program is designed to manipulate these objects
  - an object has 2 kinds of properties: *attributes* (representing the state of the object) and *operations* (the methods how to change the state of the object)

- Object-Oriented Programming (continued):
  - a *class* is a description of a set of similar objects
  - class can be viewed as a complex type and objects as variables of that type
  - *Bottom-up* design: first design simple classes, and then use them to design more complex classes, etc.
  - another powerful concept of OOP is *inheritance* – you can derive new classes from old ones (and reuse the existing code)

C++: main characteristics

- Backwards compatibility with C, inherits all benefits:
  - "Structured" language.
  - Power and Simplicity
  - Platform independence (at the level of source code) $\Longrightarrow$ portability.
  - Close to the operations of most computers $\Longrightarrow$ leading to efficient implementations, allowing high performance code.
  - Small size of the language (small core, with extensible libraries, "standard libraries").
  - Open standard
- "Object-oriented" language.
- ANSI/ISO committee (est. 1990), standard in 1998 extending language with exceptions, RTTI, templates and STL

Sample questions

- What are the main characteristics of C++ language?
- How C language achieves at same time power and simplicity?
- What is the *main* difference between procedural and object-oriented programming?

Supplementary materials

- The Development of the C Language,
  `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`
- CPL,
  `http://www.wikipedia.org/wiki/Combined_Programming_Lan`
- Open standards,
  `http://perens.com/OpenStandards/Definition.html`
- Incompatibilities Between ISO C and ISO C++,
  `http://david.tribble.com/text/cdiffs.htm`
- Java vs. C++:
  `http://en.wikipedia.org/wiki/Comparison_of_Java_and_C%`

Phases of C++ compilation

- create a **source code**: write a program in C++ using a text editor or IDE (Integrated Development Environment)
- **compile** the source code: run a *compiler* to translate a program to "**object file**" (in machine language of the computer)
- **link** the object file(s): run a *linker* to link object file(s) and libraries together, creating the **executable code**

Compilers – UNIX, Linux

using the GNU C++ compiler from the Free Software Foundation; available at CSIL

`http://www.cs.sfu.ca/CourseCentral/SW/C/gcc/`

write at the prompt:

```
g++ program.cpp
```

generates `program.o` first, then links it to `a.out` and deletes `program.o`

Examples:
```
g++ program.cpp library.o -o prog.exe
g++ program.cpp -lm -c
```

Command-line compilers for DOS

- GNU compiler `gpp` available at
  `http://www.delorie.com/djgpp/`
- Borlands command-line compiler `bcc32` available at
  `http://www.borland.com/bcppbuilder/freecompiler/`

Windows Compilers

– offer IDE, debugger

- Microsoft Visual Studio C++ 2005
  – available in CSIL labs
  – students can order it for free from `https://services.cs.sfu.ca/msdnaa` starting January 22, 2008
  – this compiler should be used for all assignments
- Metrowerks compiler
- Borland C++, available in ACS Assignment Labs: `http://www.cs.sfu.ca/CourseCentral/SW/C/Borland/`
- Dev-C++ (non-commercial, my favourite) `http://www.bloodshed.net/devcpp.html` (not completely stable)

myfirst.cpp

```
1  // myfirst.cpp--displays a message
2
3  #include <iostream>      // a PREPROCESSOR directive
4  using namespace std;     // make definitions visible
5  int main()               // function heading
6  {                        // start of function body
7      cout << "Come up and C++ me some time.";
8                           // message
9      cout << endl;        // start a new line
10     return 0;            // terminate main()
11 }                        // end of function body
```

This is an example code from [Prata]; they can be downloaded from C++ Primer Plus

Analyzing myfirst.cpp

- `//` — C++ comments, `/*` `*/` — C-style comments
- `main()` function
  - a special function – interface between the program and the operating system
  - *function form*:
    type functionname(argumentlist) ← *function heading*
    {
    statements ← *function body*
    }
  - startup code calls `main` and the value returned by `main` is the exit code of the program (0 means "no error")
  - if `return` statement is omitted, compiler automatically adds
    `return 0`
    at the end of `main` function body

Analyzing myfirst.cpp

- preprocessor directive `#include`
- *header files* (`*.h` in C and old C++, no ending in C++98)
- namespaces
  - `using std::cout`
  - `using namespace std`
- old style of `iostream`
- `cout` — object of class `ostream` provided by `iostream` header file
  - output stream – sends data out (of the program)
  - operator `<<` – chosen so that it's clear which way the data flows
  - a smart object, recognizes type and behaves accordingly

Function prototype:

- function heading without function body

```
int append(int a,int b);
```

- typical program looks like this:

```
1  int func_a(int x);
2  void func_b(int x,int y);
3  char func_c();
4
5  int main()
6  { statements using calls to func_a, func_b and func_c }
7
8  int func_a(int x)
9  { statements }
10
11 void func_b(int x,int y)
12 { statements }
13
14 char func_c()
15 { statements }
```

- the same program without use of function prototypes:

```
1  int func_a(int x)
2  { statements }
3
4  void func_b(int x,int y)
5  { statements }
6
7  char func_c()
8  { statements }
9
10 int main()
11 { statements using calls to func_a, func_b and func_c }
```

- *Q:* can you imagine a situation when you need to use function prototypes?

getinfo.cpp [prata]

```
1  // getinfo.cpp -- input and output
2  #include <iostream>
3
4  int main()
5  {
6    using namespace std;
7    int carrots;
8    cout << "How many carrots do you have?" << endl;
9    cin >> carrots;    // C++ input
10   cout << "Here are two more. ";
11   carrots = carrots + 2;
12 // the next line concatenates output
13   cout <<"Now you have "<<carrots<<" carrots."<<endl;
14   return 0;
15 }
```

Analyzing getinfo.cpp

- *declaration statement:* `type variablename;`
  example: `int carrots;`
  - each variable has to be declared before use
  - it's a good practise to immediately assign value to declared variable
    `int carrots=0;`
- `cin` — object of class `istream` provided by `iostream` header file
  - input stream – takes data (flowing into the program)
  - "operator `>>`" — chosen so that it's clear which way the data flows
  - a smart object, recognizes type and behaves accordingly

Sample questions

- Which function is the interface between the program and the operating system and what is its prototype?
- What is the difference between function declaration and definition?
- Why are prototypes useful?
- Why is it recommended to initialize a variable when declaring it?

Coding standards

- **indentation:** statements inside braces (block) are indented

```
1 int main()
2 {
3   cout <<"Hello."<<endl;
4 }
```

- every nested statement/block is indented with respect to its parent

```
1 if (a>0) {
2   if (b>0)
3     cout <<"b";
4   cout <<"a";
5 }
```

- **comments:** after function heading (explaining inputs, outputs, what the function does)
- explain what a variable represents
- inside definition of function: explain non-trivial computations
- **variable names:** meaningful
- constant variables use all caps and underscores instead of spaces: `PI`, `KM_PER_LITER`
- regular variables use lowercase; if several words, capitalize the first letter of all words but first: `radius`, `xCoordinate`, `maxWidth`
- names of classes and complex types — the same but start with capital letter: `Point`, `RationalNumber`