Programming

1. The need for programming languages
   (a) CPU executes machine code
      i. Commands CPU can understand and execute
      ii. Numeric (binary) format: binary storage!
      iii. Writing program was not an easy task
   (b) Quest for something more “humane”

2. Solution: higher level programming languages
   (a) Level of abstraction above machine code
   (b) Allows humans to specify program in something humans can read and understand
   (c) Must of course be converted to machine code for computer
   (d) Conversion is performed by compiler or interpreter software
Programming Languages

Source Code -> Language Design

Compiler/Interpreter

CPU -> Machine Code

Page 2
Compiler vs. Interpreter

Compiler:

1. Transform entire program to one piece of CPU specific machine code
2. Error? Change program and compile again.
3. Difference computer: compile again
4. Languages: C++, Pascal, Java, Fortran

Interpreter:

1. Transform program to machine code one instruction at a time
2. Error? Stop program and change.
3. Languages: Basic, Perl
Owner

1) Take lawnmower in garage
2) Mow front side of house
3) Mow Sides
4) Put lawnmower in garage
5) Take Money on table

English Instructions

Dutch Lawnmower Man
English Instructions

1) Take lawnmower in garage
2) Mow front side of house
3) Mow sides
4) Put lawnmower in garage
5) Take Money on table

Dictionary

Owner

Neighbor Volunteers

Dutch Lawnmower Man

1) Neem grasmaaier in garage
2) Maal voorzant van huis
3) Maal zijdes
4) Zet grasmaaier in garage
5) Neem geld op tafel
Where do compilers come from?

1. First step: design language x
2. Write Compiler or Interpreter to transform language x programs to machine language
3. Use Compiler to transform all programs written in language x to machine language

1. If compiler = program, it can be written in any language, as long as compiler is available...
2. This means you bootstrap the development of new languages and compilers from existing ones, starting with machine code
3. Low-Level and High-Level languages: closer or farther from machine code or hardware
Low-level vs. High Level Programming Languages

1. Low Level: close to actual hardware/CPU
   (a) Instructions match what machine and CPU can do
   (b) Extremely tedious
   (c) Machine code: binary, or hexadecimal
   (d) Assembly Language

2. High Level: far remove from actual hardware/CPU
   (a) Language reflects philosophy of data storage and manipulation
   (b) Instructions do not match capabilities of specific CPU, but rather general high-level data processing features
   (c) C, C++, Java, Perl, Python, Delphi, etc.
Programming Languages
Some examples

Assembly Language: C++:

0    LD R0, 0A
2    LD R1, 0C
4    ADD R0, R0, R1
6    STO R0, 0E
8    HALT
0A   03
0C   05
0E   08

int main (void){
    int a, b, c;
    a = 3;
    b = 5;
    c = a+b;
    cout << c;
    return 0;
}

January 18, 2004

C and C++ Programming

1. C’s history:
   (a) Developed at Bell Labs, 1972
   (b) Based on existing BCPL and B
   (c) Idea: hardware independent programs
       i. Exchange of hardware independent source code
       ii. Compiled to fit local hardware
       iii. Requires local compiler
   (d) Standardized in 1989: ANSI C

2. C++: expansion of existing C standard
   (a) Object-oriented approach
   (b) Additional support for data abstraction
C++ Compilation

Source code to executable:

1. Source code is compiled to Object code
2. Object code: machine code
   (a) Incomplete: needs to be linked to libraries
   (b) Libraries: sets of predefined constant variables and functions
   (c) Are inserted (loaded) or linked into code
3. Final executable

Linking and Loading:

1. Linking and Loading are performed automatically by g++ compiler
2. Rationale: re-use and maintenance of existing, pre-defined modules
C++ Compilation: in practical terms

1. Write source
   (a) C++ syntax
   (b) Use any editor
   (c) save to source code file

2. Compile source code
   (a) Use compiler: g++
   (b) Errors: back to 1
   (c) No errors: executable has been generated

3. Run executable
   (a) Do not confuse with source code!
   (b) Machine-specific!
   (c) No compilation errors != correct program
Murphy’s law
Believe me, it applies to software development!

Compile Errors:
1. Error in source code:
2. Syntax: error in actual spelling
3. Structure: error in program structure
4. Picked out by compiler

Run-Time Errors:
1. Errors in actual execution of program
2. Example:
   (a) Values out of range
   (b) Memory allocation
   (c) Validity: is output really what it is supposed to be?
How to write a program?
It’s all about PROBLEM SOLVING

1. State the problem clearly
2. Describe the input and desired output information
3. Work the problem by hand, check for exceptions
4. Develop a solution and convert it to a computer program
5. Test the solution with a variety of data

The purpose is to decompose a problem into little steps and commands a computer can understand, i.e. rephrase the problem in a sequence of instructions and transformations that correspond to the syntax and semantics of the language you are programming in.
An example

1. Problem: compute straight line distance between two points in a plane

2. Input - Output description: blackbox model of program
Hand Example

Point 1: \( p_1 = (1, 3) \)

Point 2: \( p_2 = (4, 4) \)

Distance: hypothenusa of right triangle

\[
d(p_1, p_2) = \sqrt{\Delta x^2 + \Delta y^2}
\]

\[
d(p_1, p_2) = \sqrt{3^2 + 1^2} = \sqrt{9 + 1}
\]

\[
d(p_1, p_2) = 3.16
\]
Algorithm Development

Decompose problems in small steps that correspond to computer language

1. Ask for x and y coordinates for point 1, \( x_1 \) and \( y_1 \)
2. Ask for x and y coordinates for point 2, \( x_2 \) and \( y_2 \)
3. Compute \( \Delta_x = x_2 - x_1 \) and \( \Delta_y = y_2 - y_1 \)
4. Compute distance, \( d = \sqrt{\Delta_x^2 + \Delta_y^2} \)
5. Print \( d \)
Write and Compile C++ Program:

demo in prog
Compilation Process

1. Source code: your C++ program
   (a) Text file
   (b) Any editor (Let’s use pico)
   (c) Adheres to C++ language specifications

2. Compilation:
   (a) Production of Executable
   (b) Transformation of source code to machine code (executable)
   (c) Executable runs only on specific CPU/machine

3. UNIX commands:
   (a) We use g++
   (b) Assume your source code file: test.cpp
   (c) You want executable in file: test
   (d) Type: g++ -o test test.cpp
C++ Programming

1. Your C++ program must conform to specific syntax:
   (a) Every line terminates with semi-colon ‘;’ (except include!)
   (b) Blocks of commands are grouped with { and }
   (c) Comments are preceded by //, preprocessor directive by #

2. General Structure:
   (a) Preprocessor directives:
      i. Specifies libraries (include statement)
      ii. Definition of constants
   (b) main function: mother of all command groups and functions
      i. Variable declaration
      ii. Commands
Syntax and structure of C++ programs

```cpp
#include<library>
#define PI 3.1415

int main (void){

    Variable Declaration:
    double a,b=3;

    Commands:
    c=b*PI;
    cout << c;

}
```
Syntax and structure of C++ programs

Elements in example:

1. Comments, preceded by //</

2. Preprocessor Directives: (include)

3. Main function block: main {···}
   (a) Object/Variable declaration
   (b) Commands
// radius.cpp: Calculates radius from circumference
#include <iostream.h>
#define PI 3.1415

int main (void){
    // variable declaration
    double d, r;
    // input circumference
    cout << "Circumference? ":
    cin >> d;
    // calculate radius
    r = d / (2*PI);
    // output radius
    cout << "Radius = " "r "endl;
    return 0;
}
C++ Program structure

1. General Structure:
   
   (a) Preprocessor directives:
       i. Specifies libraries (\texttt{include} statement)
       ii. Definition of constants
   
   (b) \texttt{main} function: mother of all command groups and functions
       i. Variable declaration
       ii. Commands
Preprocessor directives

1. **include:**
   (a) Directs compiler to use specified library
   (b) Common Libraries:
      i. iostream: input and output (cin, cout)
      ii. stdlib: memory, random numbers, sort, etc
      iii. math: mathematical functions
   (c) Example: `#include <math>`

2. **define:**
   (a) Define constant for compiler
   (b) Way of conveniently defining and labeling constants
   (c) Example: `#define PI 3.1415`
On to **main**: elements of a C++ program.

1. Comments: mark “ignore” for compiler

2. Two aspects of programming languages:
   (a) Variables or Data Objects: store data
      i. Variable types
      ii. Constants
      iii. Declarations
   (b) Commands, functions: manipulate data
      i. Input, output
      ii. Operators (assignment, addition, subtraction, enz)
Comments

1. Programs will be read by you and other programmers
   (a) Always comment your code
   (b) Provide concise description of purpose and method
   (c) Adhere to conventions

2. Comments:
   (a) Single-line: preceded by //
   (b) Multi-line: in between /* … */
   (c) No spaces between //
   (d) Don’t forget to terminate */ with */ otherwise everything that follows is comment for compiler

3. Some conventions:
   (a) Start code with detailed description (author, date, purpose, etc)
   (b) Intersperse code with comments that apply to line below
#include <iostream>

int main(void) {
    /* output follows */

    /* we don’t want this next line!
    // cout << ”Goodbye World!” << endl;

    // this is better:*/
    cout << ”Hello World!!” << endl;
    return 0;
}

January 18, 2004