Advanced practical **Programming** for Scientists

Thorsten Koch

Zuse Institute Berlin

TU Berlin

WS2014/15

Administrative stuff

All emails related to this lecture should start with **APPFS** in the subject.

Everybody participating in this lecture, please send an email to <thorsten.koch@tu-berlin.de> with your Name and Matrikel-Nr.

We will setup a mailing list for announcements and discussion.

Everything is here http://www.zib.de/koch/lectures/ws2014_appfs.php

If you need the certificate, regular attendance, completion of homework assignments, and in particular participation in our small programming project is expected. Grades will be based on the outcome and a few questions about it ☺ .

No groups.

Planned Topics

Overview: Imperative, OOP, Functional, side effects, thread safe, Design by contract

Design: Information hiding, Dependencies, Coding style, Input checking, Error Handling

Tools: git, gdb, undodb, gnat

Design: Overall program design, Data structures, Memory allocation

Examples: BIP Enumerator, Shortest path/STP, SCIP call, Gas computation

Languages and correctness: Design errors/problems C/C++, C89 / C99 / C11, Compiler switches, assert, flexelint, FP, How to write correct programs

Testing: black-box, white-box, Unit tests, regression tests, error tests, speed tests

Tools: gcov, jenkins, ctest, doxygen, make, gprof, valgrind, coverity

Software metrics: Why, Examples, Is it useful?

Other Languages: Ada 2012, Introduction, Comparison to C/C++, SPARK

Parallel programming: OpenMP, MPI, Others (pthreads, OpenCL), Ada

How to design large programs

All information is subject to change.

Attitudes

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and easily usable implementation.

— Bader, Moret, Sanders

Real Programmers don't comment their code. If it was hard to write, it should be hard to understand and harder to modify.

— Fortune (6)

Beware of bugs in the above program. I have only proved it correct, not tried it.

— D.E.Knuth

*The single most important rule of testing is to **do** it.*

— Kernighan, Pike

Imperative Programming

In computer science terminology, **imperative programming** is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.

Imperative programming, http://en.wikipedia.org/w/index.php?title=Imperative_programming&oldid=624302389 (last visited Sept. 21, 2014)

.

Imperative Programmierung ist ein Programmierparadigma. Danach werden Programme so entwickelt, dass „ein Programm aus einer Folge von Anweisungen besteht, die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll “.

Die imperative Programmierung ist das am längsten bekannte Programmierparadigma. Diese Vorgehensweise war, bedingt durch den Sprachumfang früherer Programmiersprachen, ehemals die klassische Art des Programmierens. Sie liegt dem Entwurf von vielen Programmiersprachen, zum Beispiel ALGOL, Fortran, Pascal, Ada, PL/I, Cobol, C und allen Assemblersprachen zugrunde.

Seite „Imperative Programmierung“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 20. September 2014, 14:27 UTC. URL: http://de.wikipedia.org/w/index.php?title=Imperative_Programmierung&oldid=134202110 (Abgerufen: 21. September 2014, 20:41 UTC)

Imperative Programming

I am in command!

input->putput->output

Data separated from instructions (more or less ☺ as instructions are data)

Von-Neumann Architecture/Stored-Program-Computer

Access Memory, do computations incl. conditional, PC program counter

-> allows: goto (jump), if (conditional), while (loop)

1. Do it!
2. Anyway!

Structured programming vs. goto

Blocks, subroutines, scopes.

Object-Oriented Programming

Building your own world of objects

Object-oriented programming attempts to provide a model for programming based on objects. OO programming integrates code and data using the concept of an “object”. An object is an abstract data type with the addition of polymorphism and inheritance.

An object has both state (data) and behavior (code).

-> Information hiding

-> polymorphism comes naturally

-> single vs. multiple inheritance.

-> templates and generics

Functional Programming

The way mathematicians think.

Functional programming is a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) both times.

Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

-> side effects / mutable state -> rand(), getchar(), putchar()

-> call by value, call by reference

-> thread safeness -> errno

Programming Paradigms

- **Imperative programming** – defines computation as statements that change a program state (Assembler)
- Procedural programming, structured programming – specifies the steps the program must take to reach the desired state (C, Pascal, Fortran 77)
- **Functional programming** – treats computation as the evaluation of mathematical functions and avoids state and mutable data
  (Lisp, ML, Haskell, Erlang, Ocaml)
- **Object-oriented programming** (OOP) – organizes programs as objects: data structures consisting of datafields and methods together with their interactions
  (Smalltalk, C++, Java, Eiffel)
- **Declarative programming** – defines computation logic without defining its control flow (Prolog)
- **Event-driven programming** – the flow of the program is determined by events, such as sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads (JavaScript)

Have you ever ...

- wasted a lot of time coding the wrong algorithm?
- used a data structure that was much too complicated?
- tested a program but missed an obvious problem?
- spent a day looking for a bug you should have found in five minutes?
- needed to make a program run three times faster and use less memory?
- struggled to move a program from one architecture to another?
- tried to make a modest change in someone else's program?
- rewritten a program because you couldn't understand it?

Was it fun?

From: Kernighan,Pike „The practise of programming“

Design goals

These include

- **simplicity**, which keeps programs short and manageable;
- **clarity**, which makes sure they are easy to understand, for people as well as machines;
- **generality**, which means they work well in a broad range of situations and adapt well as new situations arise; and
- **automation**, which lets the machine do the work for us, freeing us from mundane tasks.

From: Kernighan,Pike „The practise of programming“

Winner OSC1990 Best small programm

```
v,i,j,k,l,s,a[99];
main()
{
for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,
j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j),
" #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&
v+i-j))&&!(l%=s),
v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

What might it possibly do?

How to achieve this

- Be able to follow control flow (-imperative, +structured, -OO, +functional)
- Structuring programs into units (-imperative, +structured, +OO)
- Minimize dependencies (between components)
  - Minimize scope
  - Minimize side effects (+functional)
  - Data hiding (+OO)
  - How about things happening automatic?
    (member functions in C++, Garbage collection)
  - Being clever?: while(*s++ = *t++);
  - DbC

Something to think about

```
int data[10000]; // all 0..255
long long fun = 0;
unsigned i;
[…]
int t = (data[i] - 128) >> 31;
fun += ~t & data[i];
```

This is not the way

```
/* The Computer Language Benchmarks Game http://benchmarksgame.alioth.debian.org/
Contributed by Dmitry Vyukov
*/
#define _GNU_SOURCE
#include <stdlib.h>
[…]
#define CL_SIZE 64

void* cache_aligned_malloc(size_t sz)
{
    char*                       mem;
    char*                       res;
    void**                      pos;

    mem = (char*)malloc(sz + 2 * CL_SIZE);
    if (mem == 0)
        exit(1);
    res = (char*)((uintptr_t)(mem + CL_SIZE) & ~(CL_SIZE - 1));
    pos = (void**)(res - sizeof(void*));
    pos[0] = mem;
    return res;
}
```

Design by Contract

(DbC), is an approach for designing software.

It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, post-conditions and invariants.

These specifications are referred to as “contracts”, in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Pre-conditions

Post-conditions

Invariants

Exercise 1: Setup

Please check out the data for this exercise located here:

https://github.com/mattmilten/appfs

You will find a programm named **`ex1_gen`** used to generate the input data.

Run this program as follows:

```
./ex1_gen 500000000 >ndata.dat
```

The file **`ndata.dat`** should then contain 500.000.001 numbers, binary stored as signed little endian 32 bit integers.

You can check by

```
ls -l ndata.dat
```

And it should say something like 2GB.

The first number is 123456789.

Exercise 1: What to do

Write a program named **`ex1`** in C or your favorite language, which

1. Reads in the numbers from **`ndata.dat`**

2. Prints the numbers,
   - starting from 0
   - in increasing order,
   - each number at most once,
   - ASCII representation,
   - one number per line,
   - no leading zeros or spaces,
   - lines ended by a single newline.

Exercise 1: How to report

1. use **`time ex1 ndata.dat`**
   to get the runtimes of your program

2. run **`./ex1 ndata.dat | wc`**

3. run ./**`ex1 ndata.dat | md5sum`**

Send the output of `time`, `wc`, and `md5sum` together with the source code to <thorsten.koch@tu-berlin.de> with a

subject of APPFS ex1 vorname nachname

Deadline: 23.10. 16 Uhr (earlier would be better)

Fun with FP Arithmetic

```
1  double s[2048];
2  double e = 1;
3  int    n = 0;
4
5  do { n = n + 1;   e = e / 2;   s[n] = 1 + e; }
6
7  while(    e > 0);       /* Alternative 1 */
8  while(1 + e > 1);       /* Alternative 2 */
9  while(s[n]  > 1);       /* Alternative 3 */
```

Does the loop terminate?

Will the program crash?

If it terminates will *n* have the save value in all alternatives? (Lines 7,8,9)

```
if ( (country == SING) || (country == BRNI) ||
   (country == POL) || (country == ITALY) )
{
/*
 * If the country is Singapore, Brunei or Poland
 * then the current time is the answer time
 * rather than the off hook time.
 * Reset answer time and set day of week.
 * /
```

…

```
for (theElementIndex = 0;
     theElementIndex < number0fElements;
     theElementIndex++)
    elementArray[theElementIndex] = theElementIndex;


for (i = 0; i < nelems; i++)
   elem[i] = i;
```

```
enum { DANGER, CAUTION, CLEAR} the_signal;

If (CLEAR == the_signal)
{
  open_gates();
  start_train();
}
; =   = 4
```

Chapter 1 K&P

- Use descriptive names for globals, short names for locals
- Be consistent
- Use active names for functions
- Be accurate
- Indent to show structure
- Use the natural form for expressions
- Parenthesize to resolve ambiguity
- Break up complex expressions
- Be clear
- Be careful with side effects
- Use a consistent indentation and brace style
- Use idoms for consistency
- Give names to magic numbers

Avoid function macros

```
#define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

parameter c occurs twice in the body of the macro.

If isupper is called in a context like this,

```
while (isupper(c = getchar()))
```

const and **

```
int main()
{
   int const   fixed = 20;
   int*        var;
   int const** constptr;

   constptr  = &var;
   *constptr = &fixed;
   *var      = 30;

   printf("x=%d, y=%d\n", fixed, *var);
}
```

Fun with FP Arithmetic

```
1  double s[2048];
2  double e = 1;
3  int    n = 0;
4
5  do { n = n + 1;   e = e / 2;   s[n] = 1 + e; }
6
7  while(    e > 0);      /* Alternative 1 */
8  while(1 + e > 1);      /* Alternative 2 */
9  while(s[n]  > 1);      /* Alternative 3 */
```

Does the loop terminate?

Will the program crash?

If it terminates will *n* have the save value in all alternatives? (Lines 7,8,9)

Fun with FP Arithmetic

```
1  double s[2048];
2  double e = 1;
3  int    n = 0;
4
5  do { n = n + 1;   e = e / 2;   s[n] = 1 + e; }
6
7  while(     e > 0);          /* n = 1075                  */
8  while(1 + e > 1);           /* n = 64     (Intel-P4)     */
9  while(s[n]  > 1);           /* n = 53                    */
```

The loop will terminate, the program will not crash and *n* is different in most cases, depending on the architecture, the compiler, and the switches.

Introduction of C++ comments

```
a //*
//*/ b

In old C: a / b
In C++  : a
```