Effective Analysis Programming

Hartmut Stadie, Christoph-Erdmann Pfeiler

GridKa school
September, 6th 2011

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

SPONSORED BY THE
Federal Ministry of Education and Research

Outline

- Introduction
  - Literature
  - Example: Path finding
  - Getting started
- Organization and policy
- Design Style
- Coding Style
- Functions

Literature

Literature:

- Stroustrup: “The C++ Programming Language”, 3rd edition
- Sutter, Alexandrescu: “C++ Coding Standards”
- Press et al.: “Numerical Recipes 3rd edition”
- Meyers: “Effective C++” etc.
- ...

Read a book on programming!
You spend a lot of time writing code and should know how to do this!

Coding Guidelines

There are many ways to write C++ code. Use the right one!

Disclaimer

The following is heavily influenced by the book “C++ Coding Standards”.

Main idea:

Minimize the chance of bugs appearing in your code and find them quickly:

- use the compiler to find bugs
- write simple code, use clear designs
- always assume that the code will last long and be used by someone else

Example: Path Finding

Use cases:

e.g. computer games (RTS, RPG, and shooter) :)

Example:

```
XXXXXXXXXX
X     X  X
X s   X zX
X     X  X
X     X  X
X        X
XXXXXXXXXX
```

Algorithms

Algorithms:

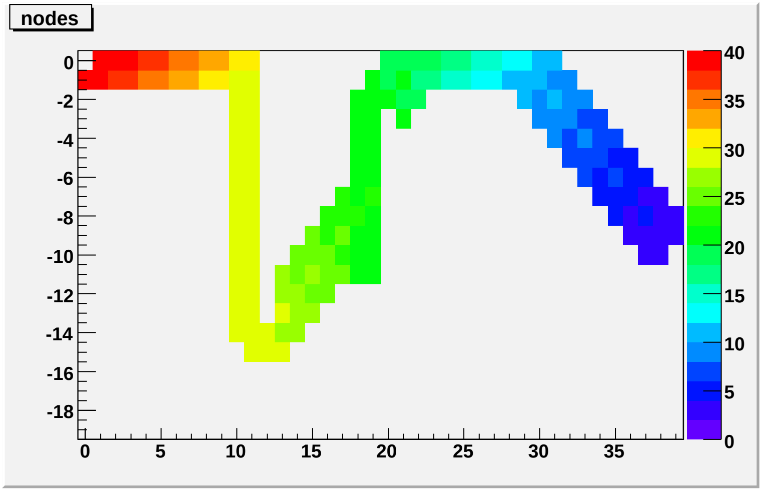
- naively
- Best-First-Search (distance to destination)
- Dijkstra-Algorithm (distance from start)
- A^* (combination)

Example map

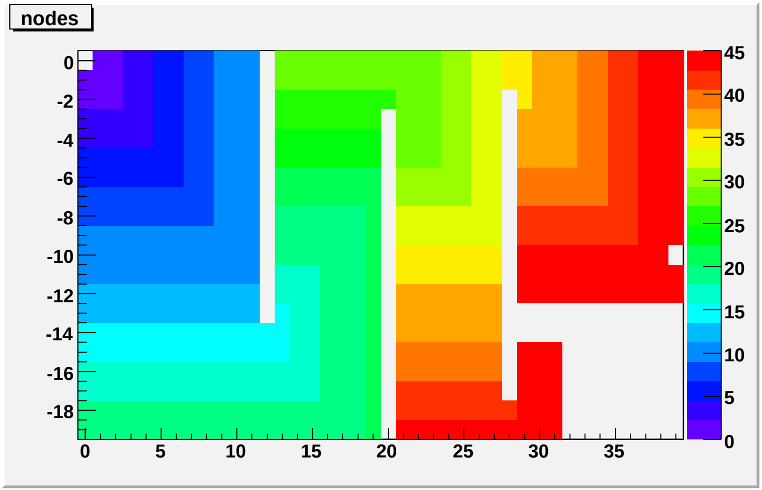
Map:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Xs           X                           X
X            X                           X
X            X               X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X           X
X            X       X       X          zX
X            X       X       X           X
X                    X       X           X
X                    X       X           X
X                    X       X           X
X                    X       X           X
X                    X                   X
X                    X                   X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

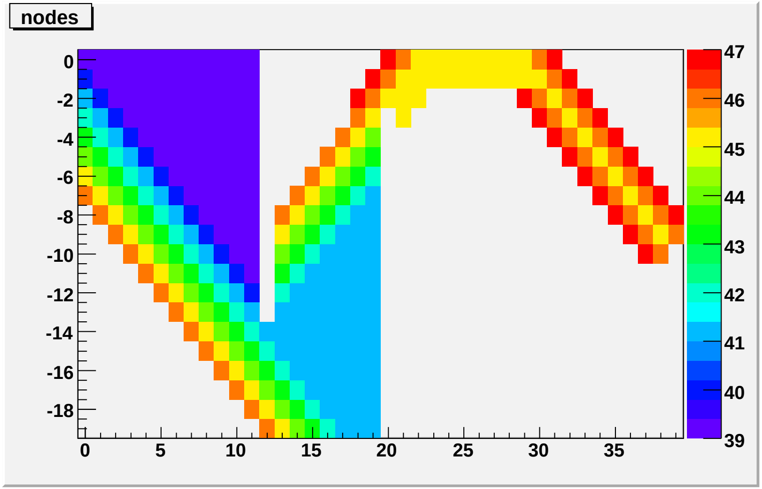
Best-First-Search



Dijkstra


nodes
0
-2
-4
-6
-8
-10
-12
-14
-16
-18
0
5
10
15
20
25
30
35
45
40
35
30
25
20
15
10
5
0

A^*


nodes
0
-2
-4
-6
-8
-10
-12
-14
-16
-18
0
5
10
15
20
25
30
35
47
46
45
44
43
42
41
40
39

Getting started

Worker nodes:

```
gks-1-133.fzk.de
gks-1-134.fzk.de
gks-1-135.fzk.de
gks-1-136.fzk.de
```

Examples:

Agenda at: https://indico.desy.de/conferenceDisplay.py?confId=4799

```
cp /tmp/stadie/astar.tgz .
```

- Introduction
- Organization and policy
  - Use a version control system
  - Use an automated build system
  - Compile cleanly and without warnings
  - Know and follow the coding style of your experiment
  - Review your code
- Design Style
- Coding Style
- Functions

Use a Version Control System

Version Control System:

Use the version control system that is provided by your experiment or institution!

Here: usage of CVS shown as an example

Version Control System: CVS

Create a repository:

```
mkdir cvsroot
cvs -d $PWD/cvsroot init
export CVSROOT=<full path to cvsroot>
```

Import project

```
cd astar
cvs import -m "start" AStar INITIAL start
```

Checkout project

```
cd ..
rm -rf astar
cvs co -d astar  AStar
```

Version Control System: CVS

How to commit code:

- find differences:

```
cvs diff
cvs status
```

- checkin:
  commit files by name and specify precisely what has changed

```
cvs commit -m"precise description" <files>
```

- check for missed files:

```
cvs diff --brief
```

- test in a second release:

```
cd ..
cvs co -d astar2 AStar
cd astar2; make
```

Tags

- use "cvs tag <tagname>" to create named snapshots of your project.
- Note: you can also check out the version of a certain date.
- "sticky tags": use "cvs up -A" to remove them.

Use an Automated Build System

Automated Build System:

Use the build system that is provided by your experiment or institution!

Here: usage of simple Makefile shown as an example

Makefile

```
#O2 for optimization, g for debugging
CFLAGS=-Wall -O2 -g -I. $(shell root-config --cflags)
LFLAGS=$(shell root-config --libs)
CC=g++
LD=g++

#all source files
SRCS=path.cxx Astar.cxx Map.cxx

OBJS = $(SRCS:.cxx=.o)

.PHONY: clean all

all: path

clean:
@rm -f *~ *.o *# *.d path

path: $(OBJS)
$(LD) $(LFLAGS) -o path $^

#rules
%.o : %.cxx
$(CC) $(CFLAGS) -MMD -c -o $@ $<
@sed -e 's/#.*//' -e 's/^[^:]*: *//' -e 's/ *\\$$//' \
             -e '/^$$/ d' -e 's/$$/ :/' < $*.d >> $*.d

-include $(SRCS:%.cxx=%.d)
```

Compile Cleanly and without Warnings

Warnings:

- enable all checks for warnings during compilation
- fix all warnings the compilers get better and better and some of them even give the same advices as the mentioned books

Know and Follow the Coding Style (of your Experiment)

Coding style:

- write useful comments
  - write code instead of comments where possible
  - do not write comments that repeat the code

```
//get node with lowest priority
miniter = min_element(m_open.begin(),m_open.end(),comparePriority);
```

  - write comments that explain the approach and rationale
- use a consistent naming convention
  - Classes, functions, Enums
  - MACROs
  - variables
  - private member variables_

Read and follow the guide lines of your experiment!

Review your Code

Discuss each others code in your group!

- Introduction
- Organization and policy
- Design Style
  - Give one entity one cohesive responsibility
  - Correctness, simplizity, and clarity come first
  - Know when and how to code for scalability
  - Do not optimize prematurely
  - Do not pessimize prematurely
  - Minimize global and shared data
  - Hide information
- Coding Style
- Functions

Give one entity one cohesive responsibility

Each variable, function, class should have one responsiblity that can be described in one sentance (or even better its name).

Correctness, simplizity, and clarity come first

KISS: Keep it simple software

"Programs must be written or people, and only incidentally for machines to execute" (H. Abelson and G.J. Sussmann)

- correct is better than fast.
- simple is better than complex.
- clear is better than cute.
- Safe is better than insecure.

Know when and how to code for scalability

- Use flexible, dynamically-allocated data instead of fixed-size arrays
- Know your algorithm’s actual complexity
- Prefer to use linear algorithms or faster whenever possible
- Try to avoid worse-than-linear algorithms whenever possible
- Never use an exponential algorithm

Do not optimize prematurely

"Premature optimization is the root of all evil." (D. Knuth)|
It is far, far easier to make a correct program fast than a fast program correct.

Example:

Do not **inline** by default.
Use a profiler to see what should be inlined.

Do not pessimize prematurely

When you have the choice between two similar constructs, do not choose the possibly slower one

- pass-by-refernce instead of pass-by-value
- prefix **++**, instead of postfix **++**

```
T& operator++() //prefix
T operator++(int) //postfix
```

- use intializer list instead of assignment in constructor
- use standard algorithms instead of own *loops*.

Minimize global and shared data

Avoid data with external linkage at namespace scope or as static class members.

Hide information

For example:

- do not make data members **public**
- return pointers or handles to them

Benefits:

- it localizes changes
- it strengthens invariants

Prefer compile- and link-time errors to run-time errors

That's actually the idea behind many guidelines listed here....

Examples: Use type checking

Type conversions:

- exact or trival
  e.g. T to const T
- promotions
  (integer promotions or float $\rightarrow$ double)
  e.g. bool $\rightarrow$ int, char $\rightarrow$ int, short $\rightarrow$ int,(+ unsigned)
  float $\rightarrow$ double
- standard conversions
  e.g. int $\rightarrow$ double, double $\rightarrow$ int, double $\rightarrow$ long double, int $\rightarrow$ unsigned int
- ....

use **enum** or full classes for symbolic constants:

Use **const** proactively

const

"**const** is your friend!"
Avoid **const** only when really needed and as pass-by-value parameters in function declaration.

Some subleties:

- **const** and pointers:

  ```
  const T* t = s; //pointer to constant
  T *const t t = s; //constant pointer
  T const* t = s; //pointer to constant
  ```
- you can use **mutable** (for cached data) in classes

Avoid Macros

Avoid magic numbers

Declare variables as locally as possible

Limit the scope of variables!
Only declare them where you need them!

Examples:

- declare variable in **for**:

```
int i = 0; for(; i < 10 ; ++i);//bad
for(int i = 0 ; i < 10 ; ++i);//better
```

- you can even do this in **if**:

```
if(TFile *f = TFile::Open("bla.roo")) ...
```

Always initialize variables

Example:

```
//bad
int switch;
if(bla) switch = 1;
else switch = 0;
//better
int switch = 0;
if(bla) switch = 1;
//or
int switch = bla ? 1 : 0;
//or
int switch = checkSwitch();
```

Avoid long functions, avoid deep nesting

Short is better, flat is better than deep

- Prefer cohesion give one function one responsibility
- do not repeat yourself do not cut-and-paste, use functions
- prefer **&&** avoid nested consecutive **if**s

```
if( A && B) ...//B is only evaluated(called) when A i
```

- prefer algorithms flatter than loops and easier to read
- do not **switch** on type tags use polymorphic functions

Minimize definitional dependencies

Use forward declarations, instead of includes

```
//bad
#include "T.hh"

class B {
  T* member_;
}

//better
class T;

class B {
  T* member_;
}
```

Make header files self-sufficient

Ensure that each header is compilable standalone

Do not reply on other headers that get included to include the headers you need.

Always write internal **#include** guards

Always add to headers:

```
#ifndef FOO_HH
#define FOO_HH
...
contents of file
...
#endif FOO_HH
```

- Introduction
- Organization and policy
- Design Style
- Coding Style
- Functions
  - Take parameters appropriately by value, (smart) pointer, or refernce
  - Miscellanea

Take parameters appropriately by value, (smart) pointer, or reference

Distinguish between input and output parameters and between value and reference parameters

for input parameters:

- always **const**-qualify pointer or references to input-only parameters
- prefer primitive(**int**,**double**) or cheap types by value
- prefer taking of inputs of other types as reference to **const**
- consider pass-by-value instead of reference if you need a copy anyways

for output:

- prefer passing by (smart) pointer if parameter is optional or the function takes/manipulates ownership
- prefer passing by reference if the parameter is needed and the function does not take/manipulate ownership

Miscellanea

- preserve natural semantics for overloaded operators
- prefer the canonical forms of arithmetic and assignment operators
- prefer the canonical form of ++ and –
- consider overloading to avoid implicit type conversions
- avoid overloading &$, ||, or (comma)