Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

The STCE Scripting Language

Uwe Naumann

Informatik 12 (STCE), RWTH Aachen University

Contents

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

- The STCE scripting language is a (very small) subset of C++ augmented with runtime support libraries (e.g. gnuplot[1] for visualization).
- The focus is on illustration of fundamental algorithmic aspects taught as part of STCE courses.
- The focus is not on efficiency nor on quality of software engineering.
- Learning the STCE scripting language will be straightforward if you are not new to (imperative) programming.
- The ability to use the STCE scripting language is a prerequisite for a number of exam formats for courses taught at STCE.

[1] www.gnuplot.info

Outline

Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

Requirements

- Shell, e.g.
  - RWTH High Performance Computing (Linux)
  - VirtualBox + STCE Linux image
  - Minimalist GNU for Windows
- Essential Linux

See tutorial.

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    cout << "Leave me alone, "
6         << "world!" << endl;
7    return 0;
8  }
```

yields output

```
1  Leave me alone, world!
```

- Use a text editor to generate source, e.g. `:-) vi main.cpp`
- Include relevant chapters of the C++ standard library (line 1).
- Avoid specification of std:: namespace globally (line 2).
- The function **int** main() is required for executable program (line 3).
- An integer value is returned to the caller, e.g. Linux shell (line 7).

- A string is written to the standard output stream cout (screen) (line 5).
- A special end of line marker endl is used for formatting (line 6).
- An STCE script is a source file with extension .cpp containing at least the function main.

Building

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

The sample code comes with a GNU makefile.[2]

Type `:-) make` to build; `:-) make clean` to clean up.

```
1  EXE=$(addsuffix .exe, $(basename $(wildcard *.cpp)))
2
3  COMPILER=g++
4  COMPILER_FLAGS=-Wall -Wextra -pedantic -O3
5
6  all : $(EXE)
7
8  %.exe : %.cpp
9    $(COMPILER) $(COMPILER_FLAGS) $< -o$@
10
11 clean:
12   rm -f $(EXE)
13
14 PHONY: all clean
```

[2] www.gnu.org/software/make/

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

Types

- void
- int
- bool
- string
- float
- double

```
#include<iostream>
using namespace std;

int main() {
  int i=1;
  cout << i << endl;
  cin >> i;
  cout << i << endl;
  return 0;
}
```

yields output

```
1
0815 // user input
815
```

- Variables have types, e.g. **int** for integers.
- Variables should be initialized at the time of their declaration, e.g. with a constant (line 5).
- Users can supply values of variables via the standard input stream cin (keyboard) (line 7).
- Integer constants are sequences of digits.
- The range of integers is limited to $[-2147483648, 2147483647]$.
- See `types_int.cpp`.

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    bool b=true;
6    cout << b << endl;
7    cin >> b;
8    cout << b << endl;
9    return 0;
10 }
```

yields output

```
1  1
2  2 // user input
3  1
```

- Truth values (constants **true** and **false**) are stored as Boolean variables of type **bool** (line 5).
- In output they are represented as `1` (**true**) and `0` (**false**).
- Implicit conversion transforms `0` into **false** and any nonzero value into **true**.
- See `types_bool.cpp`.

Types: string

```
#include<string>
#include<iostream>
using namespace std;

int main() {
  string s1="Hi ",s2;
  cin >> s2;
  cout << s1 << s2 << "!" << endl;
  return 0;
}
```

```
folks // user input
Hi folks!
```

- Strings are stored as variables of type string defined in <string> (line 1).
- String constants are enclosed in quotes (lines 6,8).
- The C++ standard library offers a wide range of functionalities for strings.
- See www.cppreference.com for further details on <string>.
- See types_string.cpp.

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    float f=1e-1;
6    cout << f << endl;
7    cin >> f;
8    cout << f << endl;
9    return 0;
10 }
```

yields output

```
1  0.1
2  3.14159265358979 // user input
3  3.14159
```

- Non-integer numerical values are stored in floating-point format.
- Real values are rounded to the grid of floating-point numbers yielding various unpleasant numerical effects.
- The range of **float** is limited to $\pm[1.17549 \cdot 10^{-38}, 3.40282 \cdot 10^{38}]$.
- **float** constants can be written in fixed-point (e.g. `3.1415`) or floating-point (e.g. `314.15e-2`$=314.15 \cdot 10^{-2}$) notation (line 5)
- The precision of **float** yields 6 significant digits in fixed-point notation.
- Implicit conversion of float-point values to **int** or **bool** uses rounding.
- See `types_float.cpp`.

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    double d=31.415e−1;
6    cout.precision(15);
7    cout << d << endl;
8    cin >> d;
9    cout << d << endl;
10   return 0;
11 }
```

yields output

```
1  3.141
2  3.14159265358979 // user input
3  3.14159265358979
```

- The range of **double** is limited to $\pm[2.22507 \cdot 10^{-308}, 1.79769 \cdot 10^{308}]$.
- **double** constants can be written in fixed-point or floating-point notation (line 5).
- The precision of **double** yields 15 significant digits in fixed-point notation.
- The precision of the cout needs to be changed from its default (6 digits) to inspect full precision of **double** (line 6).
- See `types_double.cpp`.

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

```
#include<iostream>
#include<cmath>
using namespace std;

int main() {
  float x=1e-0,y=1e1;
  y=exp(x+sin(x)/y);
  cout << y << endl;
  return 0;
}
```

yields output

```
2.95692
```

- Following resolution of flow of control imperative programs amount to sequences of assignments of results of numerical expressions to variables (line 7).
- Arithmetic operators include +,−,∗,/,%.
- Arithmetic functions include sin,cos,exp,log,fabs,fmax.
- See `www.cppreference.com` for further details on arithmetic operators and <cmath>.
- See `cmath.cpp`.

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

Flow of Control

- **if**
- **if—else [if—else ...]**
- **while**
- **do—while**
- **for**

```
#include<cmath>
#include<iostream>
using namespace std;

int main() {
  float x,y;
  cin >> x;
  if (x<0) {
    y=x+sin(x);
  }
  y=x-cos(x);
  cout << y << endl;
  return 0;
}
```

yields output

```
-1.5403
```

for input x=-1

- The code in curly brackets (e.g. y=x+sin(x)) is executed if the condition (e.g. x<0) evaluates to **true** (lines 8-10).
- Potentially very complex conditions can be formulated using relational (<,>,==,!=,<=,>=) and logical (!,&&,|| corresponding to negation, AND, inclusive OR) operators.
- The curly brackets can be omitted if they enclose a single statement (as it is the case here).
- See `if_1.cpp`.

```
#include<cmath>
#include<iostream>
using namespace std;

int main() {
  float x,y;
  cin >> x;
  if (x<0) {
    y=x+sin(x);
  } else {
    y=x-cos(x);
  }
  cout << y << endl;
  return 0;
}
```

See `if_2.cpp` and `if_3.cpp`.

```
#include<cmath>
#include<iostream>
using namespace std;

template<typename T>
void f(const T& x, T& y) {
}

int main() {
  float x,y;
  cin >> x;
  if (x<0) {
    y=x+sin(x);
  } else if (x>0) {
    y=x-cos(x);
  } else {
    y=exp(x);
  }
  cout << y << endl;
  return 0;
}
```

```
1  #include<cmath>
2  #include<iostream>
3  using namespace std;
4
5  int main() {
6    float x=1,y;
7    y=x-cos(x);
8    while (y<1) {
9      y=y+sin(x);
10   }
11   cout << y << endl;
12   return 0;
13 }
```

- The code in curly brackets (loop body; e.g. y=x+sin(x)) is executed while the condition (e.g. x<0) evaluates to **true** (lines 8-10).
- Zero or more loop iterations are performed; termination may not be guaranteed.
- The curly brackets can be omitted if they enclose a single statement (as it is the case here).
- See `while.cpp`.

```
 1  #include<cmath>
 2  #include<iostream>
 3  using namespace std;
 4
 5  int main() {
 6    float x=1,y;
 7    y=x-cos(x);
 8    do {
 9      y=y+sin(x);
10    } while (y<1);
11    cout << y << endl;
12    return 0;
13  }
```

- The loop body (e.g. y=x+sin(x)) is re-executed while the condition (e.g. x<0) evaluates to **true** (lines 8-10)
- one or more loop iterations are possible; termination may not be guaranteed
- the curly brackets can be omitted if they enclose a single statement (as it is the case here)
- See `do_while.cpp`.

```
1  #include<cmath>
2  #include<iostream>
3  using namespace std;
4
5  int main() {
6    float x=1,y;
7    y=x-cos(x);
8    int j=10;
9    for (int i=0; i<j; i=i+1) {
10     y=y+sin(x);
11   }
12   cout << y << endl;
13   return 0;
14 }
```

- The loop body (e.g. `y=x+sin(x)`) is executed while the condition (e.g. `i<j`) evaluates to **true** (lines 9-11).
- Variables local to the loop statement can be declared and initialized (e.g. `int i=0`) (line 9).
- Values of variables can be updated after each loop iteration (e.g. `i=i+1`) (line 9).
- Zero or more loop iterations are performed; termination may not be guaranteed.

- The curly brackets can be omitted if they enclose a single statement (as it is the case here).
- Typical use cases include the iteration through vectors or matrices.
- See `for.cpp`.

Outline

Software and Tools for Computational Engineering
RWTH AACHEN UNIVERSITY

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

- definition
- type generics
- arguments passed by value
- arguments passed by reference

Software and Tools for Computational Engineering · RWTH AACHEN UNIVERSITY

```
 1  #include<cmath>
 2  #include<iostream>
 3  using namespace std;
 4
 5  float g(float x, int i) {
 6    x=x+i;
 7    float y=x+sin(x);
 8    return y;
 9  }
10
11  int main() {
12    float f=sin(1e1);
13    cout << g(f,1) << endl;
14    return 0;
15  }
```

yields output

```
1  0.89632
```

- Subprograms have a name (e.g. g), a (return) type (e.g. **float**; **void** for missing return value), and a list of arguments (e.g. **float** x, **int** i; here passed by value; see below) (line 5).
- They can declare local variables (e.g. **float** y) (line 7).
- The return value (e.g. y) is passed back to the caller (e.g. main) (line 8).
- See `subprograms_1.cpp`.

```
1  #include<cmath>
2  #include<iostream>
3  using namespace std;
4
5  template<typename T1, typename T2>
6  T1 g(T1 x, T2 i) {
7    x=x+i;
8    T1 y=x+sin(x);
9    return y;
10 }
11
12 int main() {
13   float f=sin(1e1);
14   int i=1;
15   cout << g(f,i) << endl;
16   return 0;
17 }
```

yields output

```
1  0.89632
```

- Types of arguments and results can be made variable (e.g. `T1` and `T2`) (line 5).
- Type-generic subprograms become templates for the compiler to instantiate automatically for the given scenario; e.g. the compiler realizes that in line 15 `g` is called with arguments of types `T1`=**float** and `T2`=**int**; the corresponding instance of `g` is generated automatically by the compiler (lines 13-15).
- Replace **float** → **int** and **int** → **double** to see a different result (1).
- See `subprograms_2.cpp`.

Arguments Passed by Value


```
1  #include<cmath>
2  #include<iostream>
3  using namespace std;
4
5  template<typename T>
6  T g(T x) { // pass by value
7    x=x+1;
8    T y=x+sin(x);
9    return y;
10 }
11
12 int main() {
13   float f=sin(1);
14   cout.precision(15);
15   cout << g(f) << " " << f << endl;
16   double d=sin(1);
17   cout << g(d) << " " << d << endl;
18   return 0;
19 }
```

- By default all arguments are passed by value (line 6).
- Arguments passed by value result in a local copy used inside of the function.
- Potential modifications of this copy have no effect on the value of the original argument passed by the caller; hence, the code on the left yields the output

```
1 2.80506181716919 0.841470956802368
2 2.80506170934973 0.841470984807897
```

- See `subprograms_3.cpp`.

Arguments Passed by Reference

```
#include<cmath>
#include<iostream>
using namespace std;

template<typename T>
T g(T &x) { // pass by reference
  x=x+1;
  T y=x+sin(x);
  return y;
}

int main() {
  float f=sin(1);
  cout.precision(15);
  cout << g(f) << " " << f << endl;
  double d=sin(1);
  cout << g(d) << " " << d << endl;
  return 0;
}
```

- Arguments to be passed by reference need to be preceded by & (line 6).
- Arguments passed by reference are used directly inside of the function.
- Potential modifications inside of the function also apply to the value of the argument passed by the caller; hence, the code on the left yields the output

```
2.80506181716919 1.84147095680237
2.80506170934973 1.8414709848079
```

- See `subprograms_4.cpp`.

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

Assertions

```
#include<cmath>
#include<cassert>
#include<iostream>
using namespace std;

template<typename T>
T f(T x) {
  assert(x>=0);
  return sqrt(x);
}

int main() {
  float x;
  cin >> x;
  cout << f(x) << endl;
  return 0;
}
```

- Assertions are conditions to be fulfilled while executing the script; they are defined in <cassert> (line 2).
- E.g. the argument of the square root function must be non-negative (line 8).
- Assertions should be inserted into the script whenever such a condition can be formulated sensibly; understanding / debugging is simplified significantly.
- See `assert.cpp`.

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization

```
1  #include<fstream>
2  using namespace std;
3
4  template<typename T>
5  void in(T& x, string filename) {
6    ifstream ifs(filename);
7    ifs >> x;
8    ifs.close();
9  }
10
11 template<typename T>
12 void out(T x, string filename) {
13   ofstream ofs(filename);
14   ofs << x;
15   ofs.close();
16 }
17
18 int main() {
19   float x; in(x,"x.in"); out(x,"x.out");
20   return 0;
21 }
```

- Data can be read from / written to text files via file streams defined in <fstream> (line 1).
- Declaration of an input file stream of type ifstream requires specification of the name (of type string) of the file to be read from (line 6).
- Output file streams of type ofstream are declared analogously (line 13)
- Usage is then similar to cin and cout (lines 7,14).
- See `fstream.cpp`.

Outline

Environment

Types

Arithmetic

Flow of Control

Subprograms

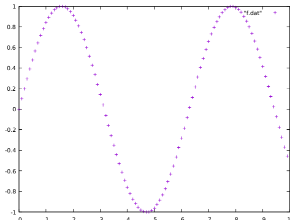
Assertions

File I/O

Visualization

```
1  #include<cmath>
2  #include<fstream>
3  using namespace std;
4
5  template<typename T>
6  T f(T x) { return sin(x); }
7
8  int main() {
9    float xmin=0, xmax=10;
10   int nsamples=100;
11   ofstream ofs("f.dat");
12   for (float x=xmin;
13     x<xmax;
14     x=x+(xmax−xmin)/nsamples) {
15     ofs << x << " "
16         << f(x) << endl;
17   }
18   ofs.close();
19   return 0;
20 }
```

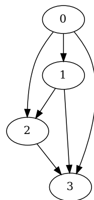
- Input files for gnuplot can be generated, e.g. by sampling a given function (lines 12-17)
- Run :-) `gnuplot` to open the gnuplot shell.
- Type `plot "f.dat"` to get



- See `www.gnuplot.info` for detailed information on gnuplot.
- See `vizualisation_1.cpp`.

```
1  #include<fstream>
2  using namespace std;
3
4  int main() {
5    ofstream ofs("g.dot");
6    ofs << "digraph {" << endl;
7    int n=4;
8    for (int i=0;i<n;i=i+1)
9      for (int j=i+1;j<n;j=j+1)
10       ofs << i << " -> "
11           << j << endl;
12   ofs << "}" << endl;
13   ofs.close();
14   return 0;
15 }
```

- Input files for graphviz can be generated, e.g. a directed acyclic complete graph with four vertices (line 6-12).
- :-) `dot -Tpdf g.dot -o g.pdf` generates the following `g.pdf`



- See `www.graphviz.org` for detailed information on graphviz and dot.
- See `vizualisation_2.cpp`.

Summary

Environment

Types

Arithmetic

Flow of Control

Subprograms

Assertions

File I/O

Visualization