Introduction to ROOT

Prof. Silvia Masciocchi
dr. Federica Sozzi

Heidelberg University

Day 1 – Programming basics

Based on the slides created by dr. Jens Wiechula, Frankfurt University

Layout of the course

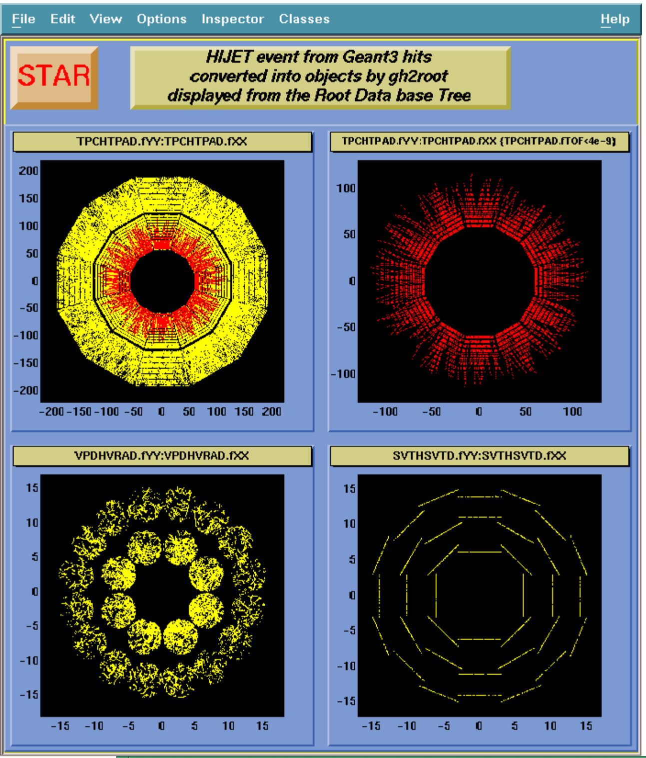
Schedule

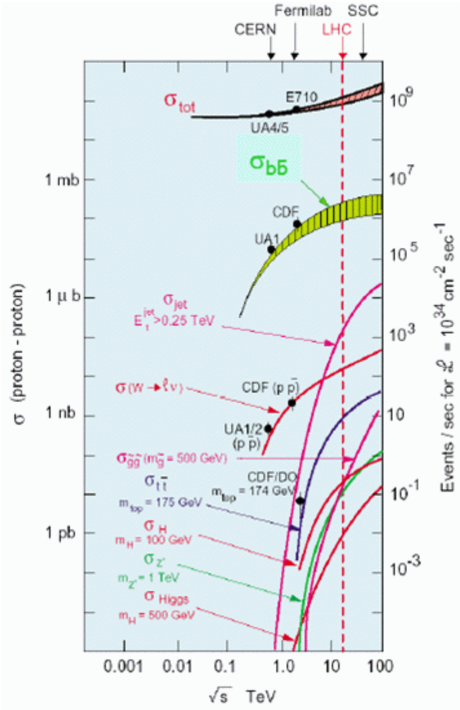
- Basics (1 day)
  - Linux primer
  - C/C++ primer
- The ROOT framework (4 days)
  - First steps with ROOT: histograms, graphs, functions, macros
  - I/O, directories, collections, string
  - Using trees
  - Projects with ROOT

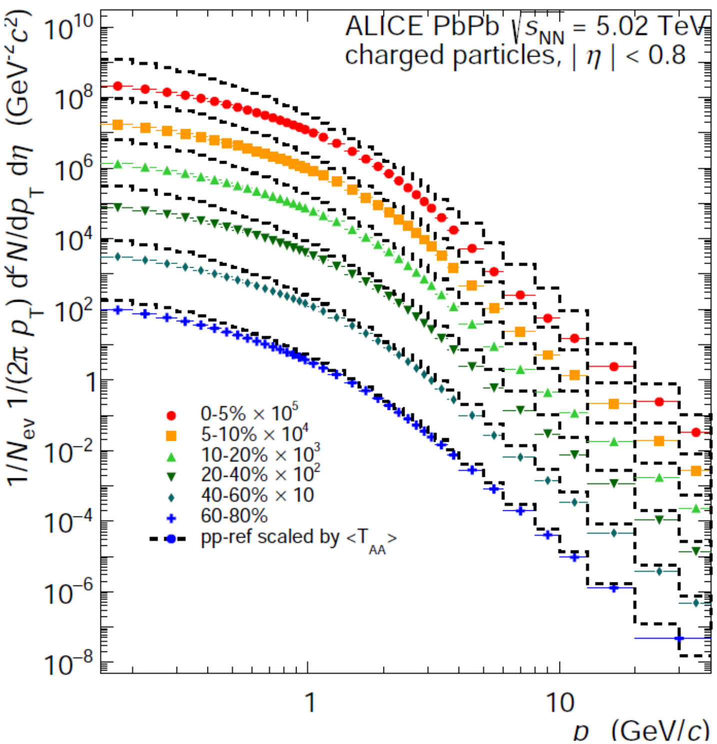
Material: http://web-docs.gsi.de/~fsozzi/

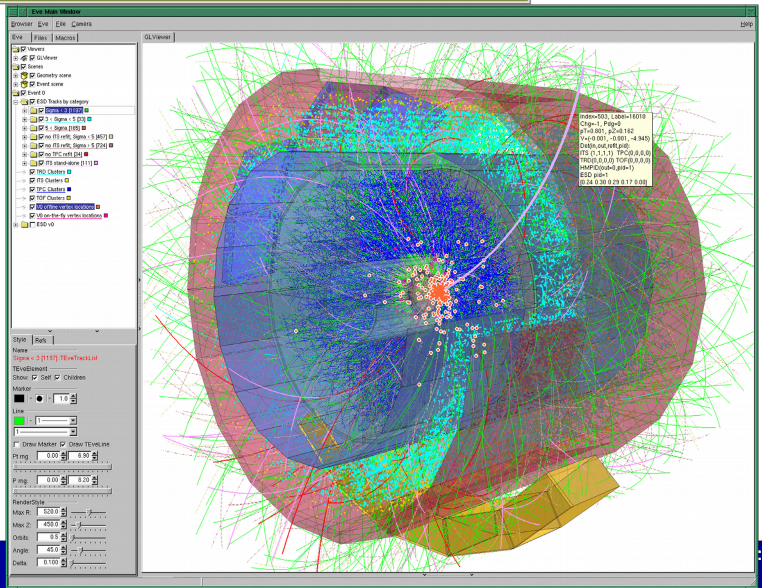
Introduction

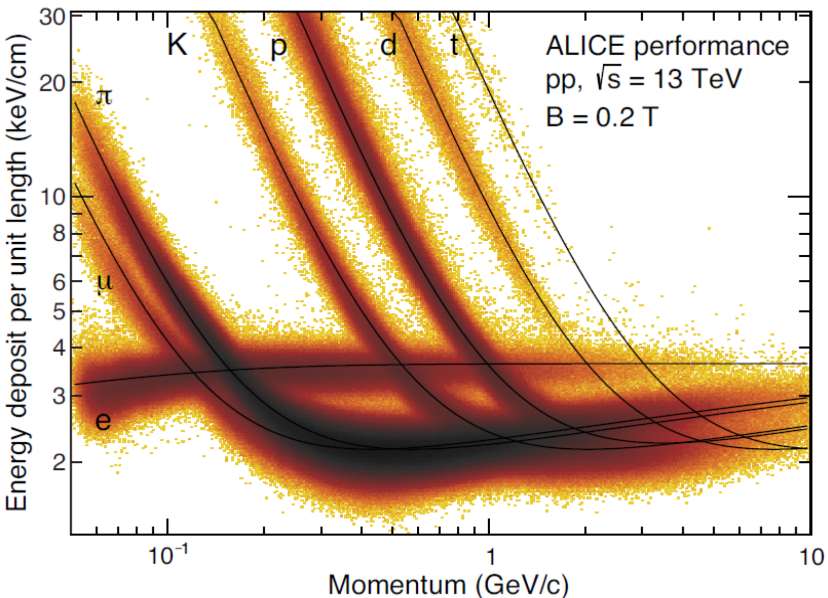
- Programming is one of the main tools for physicists working on data analysis
- ROOT is a set of object oriented frameworks based on C++ which allow storage and analysis of data
- ROOT is used by most high energy physics experiments (e.g. at CERN, GSI, BNL, Fermilab, neutrino experiments, …)
- Framework providing data storage, advance statistical analysis tools, visualization …


File Edit View Options Inspector Classes Help
STAR
HIJET event from Geant3 hits
converted into objects by gh2root
displayed from the Root Data base Tree
TPCHTPAD.fYY:TPCHTPAD.fXX
VPDHVRAD.fYY:VPDHVRAD.fXX
SVTHSVTD.fYY:SVTHSVTD.fXX


CERN
Fermilab
LHC
SSC
E710
UA4/5
CDF
UA1
1 mb
1 μ b
1 nb
1 pb
σ (proton - proton)
Events / sec for $\mathcal{L} = 10^{34}$ cm^{-2} sec^{-1}
$\sqrt{s}$ TeV


ALICE PbPb $\sqrt{s_{NN}}$ = 5.02 TeV
charged particles, | η | < 0.8
$1/N_{ev}\ 1/(2\pi p_T)\ d^2N/dp_T\ d\eta$ (GeV$^{-2}c^2$)
0-5% × 10^5
5-10% × 10^4
10-20% × 10^3
20-40% × 10^2
40-60% × 10
60-80%
pp-ref scaled by <T_{AA}>
p_T (GeV/c)




ALICE performance
pp, $\sqrt{s}$ = 13 TeV
B = 0.2 T
π
K
p
d
t
μ
e
Energy deposit per unit length (keV/cm)
Momentum (GeV/c)

Literature

Linux:
http://www.tldp.org/guides.html
http://www.linux.com/

C++:
http://www.cplusplus.com (http://www.cplusplus.com/doc/tutorial/)
The books of Bjarne Stroustrup (the inventor of C++), see Literature section in http://en.wikipedia.org/wiki/Bjarne_Stroustrup

Attention: the C++11 standard was implemented in 2011. If you intend to buy a book it might be interesting to check if it already discusses the new feature of C++11

Linux primer

- The Linux shell
- Basic commands
- More useful commands
- Redirecting/piping output
- Editing text
- Environment variables

The Linux shell

- Linux / Unix systems implement 'terminal emulators' also called 'shell' or 'console'
- Those provide command line interfaces to the system
- In a terminal, command language interpreters run, e.g. **bash**, csh, ksh, zsh, …

Basic commands I

ls (LiSt): show directory content
often used parameters: ls -l, ls -1, ls -lrt (-l:long, -t: sort by modification time, -r: --reverse)

pwd (Print Working Directory): show the current directory name

cd (Change Directory): to change the directory
cd (w/o parameter): go to users home directory
cd .. : go one directory up
cd /go/to/this/specific/directory : go to specific directory
cd - : go back to previous directory

cp SOURCE DEST (CoPy): copy source to destination

cp -r SOURCE DEST : copy recursively (including all sub-directories)

Basic commands II

mv SOURCE DEST (MoVe): move (rename) SOURCE to DEST
rm OBJECT (ReMove): remove an object (file)
rm -r OBJECT : remove objects recursively (incl. all sub-directories)
=> CAREFUL, NOT UNDOABLE!

mkdir NAME (MaKe Directory): create a new directory
rmdir NAME (ReMove Directory): remove (empty) directory

Getting help for commands:
Most commands implement a '-h' or '--help' option

On standard Linux system so called *man* pages (manuals) are installed.
They are called with 'man <command>'

Basic commands

Examples

```
fsozzi@kp1nbg055:~$ mkdir testCommands
fsozzi@kp1nbg055:~$ cd testCommands/
fsozzi@kp1nbg055:~/testCommands$ touch file1 file2 file3
fsozzi@kp1nbg055:~/testCommands$ ls
file1  file2  file3
fsozzi@kp1nbg055:~/testCommands$ mkdir newSubdir
fsozzi@kp1nbg055:~/testCommands$ cp file1 newSubdir/
fsozzi@kp1nbg055:~/testCommands$ mv file2 newSubdir/
fsozzi@kp1nbg055:~/testCommands$ ls
file1  file3  newSubdir
fsozzi@kp1nbg055:~/testCommands$ ls newSubdir/
file1  file2
fsozzi@kp1nbg055:~/testCommands$ rm -r newSubdir/
fsozzi@kp1nbg055:~/testCommands$ ls
file1  file3
fsozzi@kp1nbg055:~/testCommands$ rm *
fsozzi@kp1nbg055:~/testCommands$ ls
fsozzi@kp1nbg055:~/testCommands$
```

More useful commands / hints

less FILE	browse a text file
cat FILE	dump contents of a file to the output
head/tail [-n <lines>]	show first [n lines / default 10] of a file
awk/sed	very powerful commands to split/replace text
grep	search patterns in a file
sort	sort the output (alphabetic/numerically)
uniq	remove duplicate entries (need to be consecutive)
wc	count number of letters/words/lines
touch FILE	update the timestamp of FILE. If it does not exist create it

Tab completion

Pressing the <tab> key auto completes commands. If ambiguities exist, pressing <tab> twice shows the options

Redirecting/piping output

- Linux commands have two output streams: stdout and stderr
- One can redirect those streams using the operators |,>,>>,&>
- **|** 'pipe' the stdout from one command to the input of another (e.g. count number of entries in the current directory: ls | wc -l)
- **> file** redirect the stdout of a command to the file (overwriting it if it exists, e.g. store contents of the current directory in a file: ls > /tmp/outputOfLs)
- **>> file** same as above, but append to the file if it exists
- **&>** file redirect stdout AND stderr to the file

Redirecting/piping output

Examples

Terminal - fsozzi@kp1nbg055: ~/testCommands

File Edit View Terminal Tabs Help

```
fsozzi@kp1nbg055:~/testCommands$ ls
file1  file2  file3
fsozzi@kp1nbg055:~/testCommands$ ls | wc -l
3
fsozzi@kp1nbg055:~/testCommands$ ls > /tmp/out
fsozzi@kp1nbg055:~/testCommands$ cat /tmp/out
file1
file2
file3
fsozzi@kp1nbg055:~/testCommands$ rm file1
fsozzi@kp1nbg055:~/testCommands$ ls >> /tmp/out
fsozzi@kp1nbg055:~/testCommands$ cat /tmp/out
file1
file2
file3
file2
file3
fsozzi@kp1nbg055:~/testCommands$ cat /tmp/out | wc -l
5
fsozzi@kp1nbg055:~/testCommands$ ls testfile
ls: cannot access 'testfile': No such file or directory
fsozzi@kp1nbg055:~/testCommands$ ls testfile &> /tmp/outerr
fsozzi@kp1nbg055:~/testCommands$ less /tmp/outerr
fsozzi@kp1nbg055:~/testCommands$ cat /tmp/outerr
ls: cannot access 'testfile': No such file or directory
fsozzi@kp1nbg055:~/testCommands$
```

Environment variables

- Many things in the linux shell (commands) are handled by environment variables
- e.g. in which directories to look for executables ($PATH) or where the linker searches for shared object libraries ($LD_LIBRARY_PATH) (→ both discussed later)
- To see all environment variable use the command 'env'
- Environment variables are usually all upper case letters and are set using 'export MY_ENV_VAR=my_value'

Editing text

- Standard Linux distributions come with a wide variety of command line based and graphical editors
- Command line editors e.g.: vim, emacs, joe, pico
- Graphical editors e.g.: kate, xemacs, kedit, gedit, gvim

C++ Primer

- Introduction
- Hello world
- Compiler / Linker / Interpreter
- Variables
- Control structures
- Loops
- Functions
- Classes
- Do's and Don'ts

Introduction

What is C/C++

- C is a programming language developed in the 70s by Dennis Ritchie at AT&T Bell Labs
  - Close to hardware
  - Good performance
  - Used for programming of many OS kernels and programs (e.g. Unix/Linux + GNU)
- C++ is an object oriented programming (OOP) language extending C, developed by Bjarne Stroustrup from 1979 on at Bell Labs
  - The idea of OOP is to strengthen the encapsulation of data and operations on these from the user
  - C is nearly completely included in C++

The hello world example

Code

```cpp
/*

compile the code with

g++ hello_world.cpp -o hello_world -Wall
.
*/

// include a header file, needed for the cout command
#include <iostream>

// the cout command resides in a so-called namespace
// in order to use it globally we have to let the compiler know
using namespace std;

int main()
{
  cout << "hello world" << endl;
  return 0;
}

```

hello_world.cpp

The hello world example

Compilation / output

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ g++ hello_world.cpp -o hello_world -Wall
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./hello_world
hello world
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

The hello world example

Looking in detail I

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ g++ hello_world.cpp -o hello_world -Wall
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./hello_world
hello world
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

The compiler command

Option to specify the name
of the output (executable)
'a.out' by default

Name of the code file

Option to switch on compiler
warnings (more below)

The hello world example

Looking in detail II

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ g++ hello_world.cpp -o hello_world -Wall
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./hello_world
hello world
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

Code in the current directory is executed by pre-pending a './'
Or './' can be added to the PATH environment variable

Compiler / Linker / Interpreter

- Compiler
  - Translates C++ code into machine readable output
  - Creates the executable
- Linker
  - Code can be stored in so-called libraries, so that it can be used by different programs
  - The linker (often integrated in the compiler) combines the code with the required libraries into the executable
- Interpreter
  - A piece of software that reads the code line by line and executes it
  - Typically interpreted code is much less efficient than compiled code
  - Useful for small exercises / quick developments – we will use it to learn ROOT

Compiler warnings

- Use -Wall (perhaps -Weffc++ and -Wextra)
  - Read more in 'man g++'
- This turns on compiler warnings → potential problems inside the code are shown
- Take the compiler warnings seriously and solve them all!
- To compile C++11 use option -std=c++11

Comments

- A 'comment' is a part of the code which is skipped by the compiler
- Two types of comments can be used:
  - '//' : everything in this line appearing after a '//' is treated as a comment
  - '/* */' everything in between '/*' and '*/' is treated as a comment. Can span several lines
- Use comments to document your code
- Rule of thumb: A good code has at least as many comments as program lines

Variables

General remarks

- Values used inside the code are handled in 'variables'
- They consist of a type and a name
- The type defines what kind of data the variable can hold
- Names identify the variable in the code and can consist of letters (case sensitive), numbers and the underscore. First character needs to be a letter or an underscore
- Variables should be initialised (compiling with -Wall gives a warning if this is not respected)
- Always use meaningful names for variables, not var1, var2, var3 etc.

Variables

Example

```
18  // type name
19  // not initialised -> compiling with -Wall will give a warning
20  int myvar;
21
22  // variables should always be initialised:
23  int myvar2=2;
24
25  // another possibility for initialisation is the so-called 'constructor initialisation'
26  // both are equivalent and a matter of taste imo
27  int myvar3(3);
28
29  cout << "myvar:  " << myvar  << endl;
30  cout << "myvar2: " << myvar2 << endl;
31  cout << "myvar3: " << myvar3 << endl;
```

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ g++ variables.cpp -o variables -Wall
variables.cpp: In function 'int main()':
variables.cpp:29:25: warning: 'myvar' is used uninitialized in this function [-Wuninitialized]
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./variables
myvar:  0
myvar2: 2
myvar3: 3
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

variables.cpp

Variables

Validity range (scope)

- Variables are valid within the code block they are defined in
- A code block is the code enclosed in {}

Variables

Example - scope

```
16 int main()
17 {
18   // type name
19   // not initialised -> compiling with -Wall will give a warning
20   int myvar;
21
22   {
23     //myvar2 is only valid inside this code block enclosed by {}
24     int myvar2=2;
25
26     // myvar is valid here, since this code block is enclosed in the code block
27     // of the main function
28     cout << "myvar:  " << myvar  << endl;
29     cout << "myvar2: " << myvar2 << endl;
30   }
31
32   cout << "myvar:  " << myvar  << endl;
33
34   // this line will give a compiler error, since myvar2 is not valid here
35   cout << "myvar2: " << myvar2 << endl;
36   return 0;
37 }
38
```

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ g++ variables_valid.cpp -o variables_valid -Wal
l
variables_valid.cpp: In function 'int main()':
variables_valid.cpp:35:25: error: 'myvar2' was not declared in this scope
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

variables_valid.cpp

Variables

types

Character types: They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.

Numerical integer types: They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.

Floating-point types: They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

Boolean type: The boolean type, known in C++ as bool, can only represent one of two states, true or false.

Taken from: http://www.cplusplus.com/doc/tutorial/variables/

Basic types

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than `char`. At least 16 bits.
	char32_t	Not smaller than `char16_t`. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as `char`. At least 8 bits.
	signed **short** *int*	Not smaller than `char`. At least 16 bits.
	signed **int**	Not smaller than `short`. At least 16 bits.
	signed **long** *int*	Not smaller than `int`. At least 32 bits.
	signed **long long** *int*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than `float`
	long double	Precision not less than `double`
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

Taken from: http://www.cplusplus.com/doc/tutorial/variables/

char, int, and double are typically selected to represent characters, integers, and floating-point values, respectively.
The other types in their respective groups are only used in very particular cases

Basic types

Type sizes are expressed in bits;
the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory

level	width	range at full precision	precision*
single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	approx. 7 decimal digits
double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	approx. 16 decimal digits

* Precision: The number of decimal digits precision is calculated via number_of_mantissa_bits * $\mathrm{Log}_{10}(2)$.

Taken from: http://www.cplusplus.com/doc/tutorial/variables/

Basic types

Examples

```cpp
// inteter types with different capacity
// type exists in a singned and unsigned version
// if nothing is specified 'signed' is assumed by default
// NOTE: the only exception to this the the char type.
//       here signed has to be specified explicitly

  signed char   achar    = -1 ;
unsigned char   auchar   =  10;

           short ashort   = -5000;
 unsigned short aushort =    5000;

           int   anint    = -128000;
unsigned int     auint    =  256000;

           long  along    = -1000000000;
unsigned long    aulong   = 1000000;
```

```cpp
// floating point types with different capacity
float  afloat = 3.1415926;

double adouble = 5.2e-30;
```

data_types.cpp

Arrays

Syntax

- C/C++ allows to create arrays of variables

```
int myarray[5];  // integer array of size 5 (0..4)
myarray[0]=1;
myarray[1]=4;
...
myarray[4]=3;

int myarray2[3]={0,0,0}; // direct initialisation
```

Arrays
Syntax

- C/C++ allows to create arrays of variables

```
int myarray[5];  // integer array of size 5 (0..4)
myarray[0]=1;
myarray[1]=4;
...
myarray[4]=3;

Int myarray2[3]={0,0,0}; // direct initialisation
```

Important: Indexing starts at '0'

Arrays

Remarks

- Be careful with such c-arrays, since the compiler does not check if you access memory which is out of the validity range of the arrays
  - this can lead to code crashes
- Often it is better to use special classes for this purpose (e.g. vector (stl), TVectorF (ROOT), …)

Operators

Math and comparison

Math operations

= assignment operator
+, -, *, / add, subtract, multiply, divide
% modulo: rest of integer division, e.g. 5%2=1, 5%3=2
+=, -=, *=, /= operation and assignment
++, -- increment, decrement (by 1), !precedence → next slide
<<, >> bit shift left, right

Comparison and logical

== is equal (don't mix up with assignment)
!= is not equal
! logical negation
<, >, <=, >= is smaller, is larger, is smaller or equal, is larger or equal
&& logical and
|| logical or

This list is not exhaustive, operators might have different meaning, e.g. for classes (see later)
More can be found e.g. here: http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operators

Precedence

- Operators are executed in a certain order, they have a certain 'priority'. This is called precedence
- The precedence can be made explicit using () like in mathematics, e.g. + and *
- C(++) follows mathematical rules like * (mult) before + (add)

Don't confuse with addition/ subtraction!

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`++ --`	Suffix/postfix increment and decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
3	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new, new[]`	Dynamic memory allocation	
	`delete, delete[]`	Dynamic memory deallocation	

Taken from: http://de.cppreference.com/w/cpp/language/operator_precedence

Operators
Precedence 2

4	`.* ->*`	Pointer to member	Left-to-right
5	`* / %`	Multiplication, division, and remainder	
6	`+ -`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=` `> >=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	`== !=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`?:`
	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		Ternary conditional
		Direct assignment (provided by default for C++ classes)
		Assignment by sum and difference
		Assignment by product, quotient, and remainder
		Assignment by bitwise left shift and right shift
		Assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

- In addition there is a direction (associativity) in which they are interpreted
- Examples: a=b=c is equal to a=(b=c) but different from (a=b)=c

  a<<b+1 is different from (a<<b)+1

Taken from: http://de.cppreference.com/w/cpp/language/operator_precedence

Control structures

Types

- Control structures are used to steer the program flow
  - Flow control: if else, switch
  - Loops: while, do while, for
  - Jumps: break, continue

Control structures

if, else, else if

Control the program flow following some conditions, where condition means a value of '0' or non '0'.

```
If ( <condition1> ) {
  Code Block 1;
} else if ( <condition 2> ) {
  Code Block 2;
} else if ( <condition 3>) {
  Code Block 3;
} else {
  If nothing of the before is
  fulfilled execute this code block;
}
```

Control structures

if, else, else if – example

```
// every condition other than '0' is interpreted as fulfilled (true)
if (0) {
  cout << "This code will never be executed" << endl;
}

if (1) {
  cout << "This code will always be executed" << endl;
}

// it is recommended to use a code block {} after if
// if this is not done, only the next line will be handled by if
if (0)
  cout << "this line is handled by the if and will not be executed" << endl;
cout << "this line is not handled by the if" << endl;

int valueToBeCheckedInIf=10;

if ( valueToBeCheckedInIf<10 ) {
  cout << "Value is smaller than 10" << endl;
} else if ( valueToBeCheckedInIf>=10 && valueToBeCheckedInIf<20 ) {
  cout << "Value is between 10 and 20" << endl;
} else {
  cout << "Value is larger than 20" << endl;
}
```

control_structures.cpp

Control structures

switch

Switch between several possibilities. Only possible for integer type values.

```
switch (<statement>) {
    case <n1>:
        execute this if <statement> is <n1>;
        break;
    case <n2>:
        execute this if <statement> is <n2>;
        break;
    default:
        By default execute this;
}
```

Be careful, everything after a case statement is executed until break

Control structures

switch – example

```
int value=2;

switch (value) {
  case 0:
    cout << "value is 0" << endl;
    break; // leave the switch structure
  case 1:
    cout << "value is 1" << endl;
  case 2:
    cout << "value is 1 or 2" << endl;
    break;
  default:
    cout << "value is none of 0,1,2" << endl;
}
```

control_structures.cpp

Control structures

for loop

Loops are used to execute a part of a code several times

```
for (<initialisation>; <condition>; <end of loop>) {
  execute this code;
}
```

<initialisation>	is executed when the loop is entered
<condition>	the loop is executed as long as this is fulfilled
<end of loop>	is executed at the end of each loop iteration

Control structures

while, do while loop

Loops are used to execute a part of a code several times

```
while (<condition>) {
  execute this code;
}
```

The loop is entered and executed if and as long the condition is true

```
do {
  execute this code;
} while (<condition>);
```

The loop is entered **at least once** and executed as long the <condition> is true

Used only seldomly

Be careful about the logic in while loops in order to avoid infinite loops

Control structures

break and continue a loop (for, while, do..while)

- The execution of a loop can be stopped anywhere inside the loop using the 'break' statement.
- The execution of the loop can be continued anywhere inside the loop using the 'continue' statement.

```
while (<condition>) {
  execute this code;
  If (<condition2>) break;
  This code will not be
  reached any longer if
  <condition2> is fulfilled
}
```

```
while (<condition>) {
  execute this code;
  If (<condition2>) continue;
  This code will not be
  reached as long as
  <condition2> is fulfilled
}
```

Control structures

Examples 1

```
for (int i=0; i<10; ++i) {
  cout << "Iteration " << i << "inside the for loop" << endl;
}
```

```
int counter=0;

// this loop is not executed
while ( counter>=2 && counter<10 ) {
  cout << "1st while loop counter: " << counter << endl;
  ++counter;
}

counter=2;
// this loop is executed
while ( counter>=2 && counter<10 ) {
  cout << "2nd while loop counter: " << counter << endl;
  ++counter;
}
```

```
counter=0;

do{
  cout << "do loop counter: " << counter << endl;
  ++counter;
} while ( counter>=2 && counter<10 );
```

If you deal with counters use
++counter **instead of**
counter++ or
counter+=1 or
counter=counter+1
(reason: better performance
in some cases, e.g., if counter
is an instance of a c++ class)

control_structures.cpp

Control structures

Examples 2

```
counter=0;

// without the break statement in the if clause, the loop would go up to 10
while ( counter<10 ) {
  if (counter==5) break;
  cout << "while loop counter (with break): " << counter << endl;
  ++counter;
}

counter=0;

// without the continue statement in the if clause,
//   the loop would write all values up to 10
while ( counter<10 ) {
  if ( counter>2 && counter<8 ) {
    ++counter; // we need to increase the counter here as well
               // otherwise we would execute this loop infinitely
    continue;
  }
  cout << "while loop counter (with continue): " << counter << endl;
  ++counter;
}
```

control_structures.cpp

Functions

Basics

- Functions are used to encapsulate tasks inside the code

```
<type> <name> (<parameter list>)
{ //function block
  <function code>

  return <return value>;
}
```

<type>	basic type, void (no return value), class type
<name>	name of the function
<parameter list>	list of values passed to the function
<return value>	value returned by the function. Needs to be of <type>, or nothing in case of void
return	keyword to leave the function, can be issued at any point inside the function code

Functions

Parameter list

- Parameters are separated with a ','

```
int myFunction (int par1, int par2, float par3)
```

- Parameters can have default values
  - They are assigned with 'par=value'
  - Default values are only possible starting from the last parameter
  - If a parameter has a default value, it is optional in the function call

```
int myFunction (int par1, int par2, float par3=3.2)      //ok
int myFunction (int par1, int par2=1, float par3=3.2)    //ok
int myFunction (int par1, int par2=1, float par3)        //will not compile
```

Functions

Example

```cpp
float average(float values[], int arraySize)
{
  //
  // this function calculates the average of all values in the array 'values'
  // the the number of values in the array is 'arraySize'
  //

  //sanity check if there is anything to calculate
  if (arraySize==0) return 0.;

  // sum up all values
  float sum=0;
  for (int iValue=0; iValue<arraySize; ++iValue) {
    sum+=values[iValue];
  }

  // build average
  sum/=arraySize;

  return sum;
}

int main()
{
  //define values
  float values[5]={1.,1.5,2.5,2.8,3.2};
  //get the average of the values using the function 'average'
  float valuesAverage = average(values,5);
  //print the result
  cout << "The average of the values: ";
  for (int iValue=0; iValue<5; ++iValue) cout << values[iValue] << ", ";
  cout << "is: "<< valuesAverage << endl;

  return 0;
}
```

functions.cpp

Pointers

Basics 1

- Pointers are variables which point to the memory address of a variable or object
- Pointers are very powerful, but also very dangerous. Use with care!

```
<type> * <pointer_name>;
```

<type>	variable (or object) type, e.g. int
*	denotes that this variable is of pointer type
<pointer_name>	name of the variable

It is a very good practice to initialise pointers!

e.g.

```
int * myptr = 0x0;
```

Initialise it at least to the NULL pointer, using 0, 0x0 or NULL (equivalent)

Pointers

Basics 2

- To get the pointer of a variable use the reference operator (&)

```
int myInt=2;
int * myIntPtr=&myInt;
```

- To access the value of the address a pointer is pointing to use the dereference operator (*)

```
int myInt2 = *myIntPtr;
```

Pointers

Example

```cpp
int myInt1 = 2;
int myInt2 = 8;

int * myIntPtr = 0x0;

// using the reference operator (&)
// assign the memory address of myInt1 to myIntPtr
myIntPtr = &myInt1;
printPtrInfo(myIntPtr);

// assign the memory address of myInt1 to myIntPtr
myIntPtr = &myInt2;
printPtrInfo(myIntPtr);

// using the dereference operator (*)
// assign a value to myInt2 via the pointer
*myIntPtr = 4;
cout << "New value of myInt2 is " << myInt2 << endl;
printPtrInfo(myIntPtr);
```

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./pointers
The memory address is 0x7fff02899504
The value stored is 2
The memory address is 0x7fff02899508
The value stored is 8
New value of myInt2 is 4
The memory address is 0x7fff02899508
The value stored is 4
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

pointers.cpp

References

Basics

- References are similar to pointers, they point to the same memory as a variable, they are a sort of alias to the variable
- Safer to use
- A reference can only reference one object with which it was initialised
- Very convenient for passing results through the parameter list of a function (avoid the copy)

```
<type> & <reference_name> = <reference_variable>;
```

<type>	variable (or object) type, e.g. int
&	denotes that this variable is a reference
<reference_name>	name of the reference variable
<reference_variable>	the variable to be referenced

References
Example

```cpp
int myInt3     = 7;
// create a reference to myInt3
int & myInt3Ref = myInt3;

//print the values of myInt3 and the reference to it
// references are accessed like simple variables
cout << "The value of myInt3 is " << myInt3 << endl;
cout << "The value of the reference is " << myInt3Ref << endl;

// change the value of myInt3 via its reference
myInt3Ref=1000;
//print the values of myInt3 and the reference to it
cout << "The value of myInt3 is " << myInt3 << endl;
cout << "The value of the reference is " << myInt3Ref << endl;
```

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code

File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$ ./pointers
The value of myInt3 is 7
The value of the reference is 7
The value of myInt3 is 1000
The value of the reference is 1000
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code$
```

pointers.cpp

Dynamic memory allocation

Basics

- C++ programs can make use of two different types of memory
  - The 'Stack': used for standard variables in the program where the memory allocation is known at compile time
  - The 'Heap': dynamic memory which can be allocated free in size during run time
- ATTENTION: dynamically allocated memory needs to be freed by the user himself! Otherwise the program might run into memory problems (leak)
- Only use the heap if really necessary. The Stack is safer and heap allocation takes more time.

  Example: cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory

Dynamic memory allocation

The 'new' and 'delete' operator

- Memory is allocated dynamically with the 'new' operator
- The new operator returns a 'pointer' to the allocated space in memory

```
int *ptrToInt = new int;
```

- Dynamically allocated memory is 'freed' using the 'delete' operator

```
delete ptrToInt; ptrToInt = 0x0;
```

- Every 'new' needs a 'delete'
- After a delete the pointer should be set to NULL

Dynamic memory allocation

The 'new' and 'delete' operator 2

- Arrays are dynamically allocated using the [] behind the basic type

```
int *ptrToInt = new int[10]; // integer array of size 10
```

- The memory of an array is freed using [] after the 'delete' operator

```
delete [] ptrToInt; ptrToInt = 0x0; // delete the array
```

Classes

Basics – introduction

- Classes are structures to encapsulate data and operations on the data
- In order to use a class you have to create an instance of this class called 'object'

Classes

Basics – class definition

```
class <class_name> {
<access_specifier_1>:
    <member_1>;
<access_specifier_2>:
    <member_2>;
};

<class_name> myObject;
```

<class_name>	name of the class
<access_specifier>	private: members are only accessible inside the class itself
	public: members are accessible anywhere
	protected: members are accessible in derived classes
<member>	member of the class, either a variable or a function

Classes

Basics – class definition

```
class <class_name> {
<access_specifier_1>:
    <member_1>;
<access_specifier_2>:
    <member_2>;
};
```

Important: don't forget the ';' at the end of the class definition

```
<class_name> myObject;
```

<class_name>	name of the class
<access_specifier>	private: members are only accessible inside the class itself
	public: members are accessible anywhere
	protected: members are accessible in derived classes
<member>	member of the class, either a variable or a function

Classes

Basics – data members I

- Classes should decouple the variables used internally for operations from the user
  - private data members
  - public interface to access the data members
- E.g. imagine a car, the public interface are the pedals, gear selector, steering wheel, etc. The private part are all the gearwheels, the engine, electronics, etc.

Classes

Basics – data members II

- Data members should be initialised in the 'constructor' of the class (see below)
- If pointer data members exist and the class allocated dynamic memory, the memory must be freed in the 'destructor' of the class (see below)

Classes

Constructors

- Constructors are special functions. They don't have a return type and have the same name as the class itself
- The constructor is called once an object of a class is instantiated and should initialise all data members (usually in the 'initialisation list')
- The 'default constructor' has no arguments
- Constructors (like functions) can be overloaded (implement different argument lists)
- A copy constructor should be implemented (as well as the assignment operator – not discussed here)

Classes

Constructors – example

```
class Point2D {
public:
  //default constructor
  Point2D()                 : fx(0.), fy(0.) {;}
  //custom constructor
  Point2D(float x, float y) : fx(x ), fy(y ) {;}
  //copy constructor
  Point2D(const Point2D &point) : fx(point.fx), fy(point.fy) {;}
```

- The 'initialisation list' follows the declaration, separated by a ':'
- It allows to call the 'constructors' of the data members
- All data member should be initialised in the order they are implemented in the class definition

classes.cpp

Classes

Destructors

- The destructor, like the constructor, is a special function.
- It is called once the object goes out of scope or is destroyed (delete)
- It carries the same name as the class, but has a '~' in front

```
~Point2D();
```

Classes

Example

initialisation list

```
class Point2D {
public:
  //default constructor
  Point2D()                 : fx(0.), fy(0.) {;}
  //custom constructor
  Point2D(float x, float y) : fx(x ), fy(y ) {;}
  //copy constructor
  Point2D(const Point2D &point) : fx(point.fx), fy(point.fy) {;}

  // setter functions
  void SetPoint(float x, float y) { fx=x; fy=y; }
  // getter functions
  void GetPoint(float &x, float &y) const { x=fx; y=fy; }
  float GetX() const { return fx; }
  float GetY() const { return fy; }

  // operations
  float DistanceToPoint(const Point2D &point);
  void  Print() { cout << "Point2D with coordinates (" << fx <<","<<fy<<")"<<endl; }

private:
  float fx;      // x-coordinate of the point
  float fy;      // y-coordinate of the point
};

// function implementations
float Point2D::DistanceToPoint(const Point2D &point)
{
  // calculate distance of this point to another 'point'
  float x=0.,y=0.;
  point.GetPoint(x,y);

  return sqrt( (x-fx)*(x-fx) + (y-fy)*(y-fy) );
}
```

NB: to use “sqrt” you need to include the “cmath” library – see include in the file

Most functions are usually implemented outside the actual class definition. This is possible if the 'scope' (class_name::) is specified in front of the function

classes.cpp

Classes

Header and implementation files

- The class definition (interface) is stored in a 'header file'
  - Conventionally the file ends with '.h' (for header)
  - This file is what is needed in the code development to inform a program about the 'interface'
  - Inside another program the header is included with the #include <headerfile> directive
- The class implementation is stored in the 'implementation file'
  - Conventionally the file ends with '.cxx' or '.cpp'

Not treated in this primer

- Inheritance, polymorphism, friendship
- Name spaces
- Templates
- Forward declarations
- …
- Some of the concepts will become clearer during the course

Do and Don't

http://en.wikipedia.org/wiki/KISS_principle

KISS principle

From Wikipedia, the free encyclopedia

"K-I-S-S" redirects here. For other uses, see Kiss (disambiguation).

KISS is an acronym for "**Keep it simple, stupid**" as a design principle noted by the U.S. Navy in 1960.[1][2] The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided. The phrase has been associated with aircraft engineer Kelly Johnson (1910–1990).[3] The term "KISS principle" was in popular use by 1970.[4] Variations on the phrase include "Keep it Simple, Silly", "keep it short and simple", "keep it simple and straightforward"[5] and "keep it small and simple".[6]

- First think about what your code should do, e.g. make a sketch on paper
- Always have an expectation “a priori” on the results produced of the code, do not accept without thinking any number the code gives you
- Document your code well! (use many comments)
- Don't use meaningless names for variables
- Use functions whenever possible (if you find recurring lines of code think about putting it in a function)
- Be careful with pointer → always initialise them
- Prefer Stack memory over Heap
- Remember to free allocated memory

Exercises

- Write a small program that internally uses different data types, performing mathematical operations and printing the results. Try using references and pointers to change the results. Play with arrays and loop over them
- Write a function that calculates mean and standard deviation (sigma) of a double array. The results should be stored in variables that are passed as references in the parameter list of the function *(Solution: exercises/day01/meanSigma.cpp)*
- Write a class (Rectangle) that uses two Point2D to define a rectangle (bottom left, top right corner)
  - What functions and constructors could be useful?
  - Implement at least the functions: GetWidth, GetHeight, GetArea, GetPoint1, GetPoint2

  (Solution: exercises/day01/Rectangle.cpp)

More remarks - classes

```
class Point2D {
public:
  //default constructor
  Point2D()               : fx(0.), fy(0.) {;}
  //custom constructor
  Point2D(float x, float y) : fx(x ), fy(y ) {;}
  //copy constructor
  Point2D(const Point2D &point) : fx(point.fx), fy(point.fy) {;}

  // setter functions
  void SetPoint(float x, float y) { fx=x; fy=y; }
  // getter functions
  void GetPoint(float &x, float &y) const { x=fx; y=fy; }
  float GetX() const { return fx; }
  float GetY() const { return fy; }

  // operations
  float DistanceToPoint(const Point2D &point);
  void  Print() { cout << "Point2D with coordinates (" << fx <<","<<fy<<")"<<end
l; }

private:
  float fx;      // x-coordinate of the point
  float fy;      // y-coordinate of the point
};

// function implementations
float Point2D::DistanceToPoint(const Point2D &point)
{
  // calculate distance of this point to another 'point'
  float x=0.,y=0.;
  point.GetPoint(x,y);

  return sqrt( (x-fx)*(x-fx) + (y-fy)*(y-fy) );
}
```

- Name of the class

  followed by the Scope operator "::" is used to specify that the function belongs to the class

classes.cpp

classes

```
class Point2D {
public:
  //default constructor
  Point2D()                    : fx(0.), fy(0.) {;}
  //custom constructor
  Point2D(float x, float y) : fx(x ), fy(y ) {;}
  //copy constructor
  Point2D(const Point2D &point) : fx(point.fx), fy(point.fy) {;}

  …

  }
```

After a class is defined (in this case class "Point2D"), you can use it to instantiate objects of this class. So:

//without arguments: will use the default constructor
Points2D myObjectPoint;

// with argument: will initialize his private members to 2 and 3
Points2D myObjectPoint(2,3);

classes.cpp

classes

- After you create object you can access its functions using the dot operator (.)
- In case you access them through a pointer to the object, use the arrow operator (→)

```
int main()
{
  //create object p1 of type Point2D
  Point2D p1(2,5);
  //call its function
  p1.Print();

  //define a pointer to the object
  Point2D *p = &p1;//pointer to p1
  //call the function through the pointer : will give the same result
  p->Print();
}
```

Passing arguments as reference

```
void function1( float x, float y)
{
  ++x;
  ++y;
}

//same function but arguments are passed as reference
void function2( float &x, float &y)
{
  ++x;
  ++y;
}

int main()
{

  float myX = 5.;
  float myY = 6.;
  cout<<myX<< " "<<myY<<endl;

  //  no change in the value of the variables
  function1 (myX, myY);
  cout<<myX<< " "<<myY<<endl;

  //passing argument via reference change their values
  function2 (myX, myY);
  cout<<myX<< " "<<myY<<endl;

  //note that noting changed in the call to the function
  //only the declaration of the function above changed
  return 0;
}
```

Extra

- In computing, floating point is the formulaic representation that approximates a real number so as to support a trade-off between range and precision. A number is, in general, represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent in some fixed base; the base for the scaling is normally two, ten, or sixteen.
- Example:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{-4}^{\text{exponent}}}$$

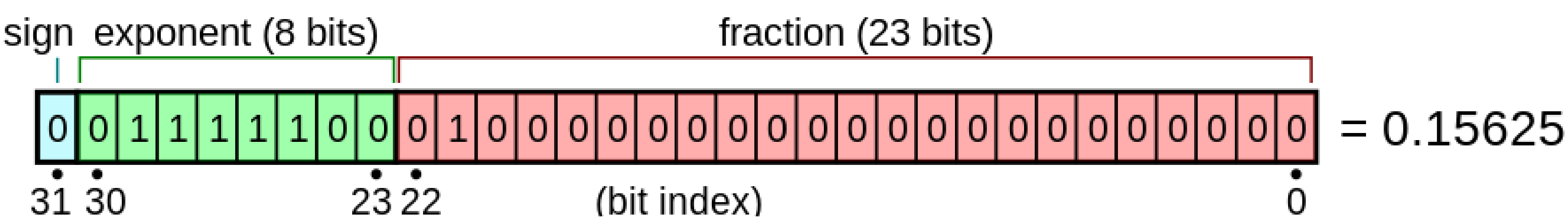
Significand also called “mantissa”

- Example of representation (32 bits)

Sign bit: 1 bit

Exponent width: 8 bits

Significand precision: 24 bits (23 explicitly stored- the first is 1)



https://en.wikipedia.org/wiki/Floating_point
https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

Taken from: http://www.cplusplus.com/doc/tutorial/variables/

More info on floating point numbers and their industry standards:
https://en.wikipedia.org/wiki/IEEE_754-1985

Basic types

Type sizes are expressed in bits;
the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory

level	width	range at full precision	precision*
single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	approx. 7 decimal digits
double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	approx. 16 decimal digits

* Precision: The number of decimal digits precision is calculated via number_of_mantissa_bits * $\text{Log}_{10}(2)$.

Ex:
With 32 bits: 24 bits of mantissa, defining the precision
Precision: 2^24 ; Translating in base 10: Log_10 (2^24) ~ 7
Max range: 2^(2^(8-1)) ~10^38

Taken from: http://www.cplusplus.com/doc/tutorial/variables/