Introduction to C++

Andreas Mussgiller*
DESY Summer Student Lectures
26/07/2011

* many slides courtesy of B. List

HELMHOLTZ ASSOCIATION

DESY

Introduction to C++

- Created by Bjarne Stroustrup in 1983
- Based on the programming language „C“
- Many extensions compared to „C“
  - Object-oriented (OO) - classes
  - Operator overloading
  - References
  - Templates
  - Easily extendable - classes
  - ...
- Became ISO standard in 1998
- Currently THE language in HEP
  - Many things would become to complicated without OO
- A new standard is being developed C++0x
  - many new features that make the life of a programmer easier

Introduction to C++ - The Main Message Today

- C++ is a very powerful programming language
- C++ is one of the most complicated programming languages
- The best way to learn a programming language is by looking at code
  - You will see many examples throughout this talk
- Not every written piece of code is good programming practice
- Many physicists think that physicists are good programmers by definition
  - There are many physicists that are excellent programmers
  - Most of us are physics result oriented programmers which mostly yields horrible code
- You will not learn C++ nor will you become an expert today
- I will cover the basics and a few common topics that you might find useful
- Follow the coding conventions of your project
- Please never use a statement like this: „I know the code looks horrible, but it works“
  - Make sure that other people are able to understand the code you have written in a reasonable amount of time

Introduction to C++ - This Talk

- I added links to online references where possible
  - http://www.cplusplus.com/reference
  - Hyperlinks will be underlined
- All examples are available for download via
  - http://mussgill.web.cern.ch/mussgill/public_files/CPPIntro.tgz
  - The names of the files in question are shown on the slides
  - On your DESY afs account do the following

```
$> wget http://mussgill.web.cern.ch/mussgill/public_files/CPPIntro.tgz
$> tar -xvzf CPPIntro.tgz
CPPIntro/area.cc
CPPIntro/area.h
...
$> cd CPPIntro
$> make
```

Our First C++ Program

HelloWorld.cc

```
#include <iostream>

int main()
{
  std::cout << "Hello, World!!!" << std::endl;
  return 0;
}
```

On the command line:

```
$> g++ -o HelloWorld HelloWorld.cc
$> ./HelloWorld
Hello, World!!!
$>
```

- Green boxes will be used for example code
- Gray boxes for commands to be executed on the command line

- ***cout*** is declared in ***iostream***
- Function ***main*** is the entry point for the operating system
- C++ is case sensitive: cout, Cout and COUT are not the same
- ***std::*** is a namespace qualifier
- Every statement has to be terminated with a semicolon
- Function ***main*** must return an integer

- *g++* is the C++ compiler
- *HelloWorld* is the executable produced by the compiler

Functions

area.h

```
#ifndef area_h_
#define area_h_

double area(double radius);

#endif // area_h_
```

area.cc

```
#include <cmath>

#include "area.h"

double area(double radius)
{
  double result = M_PI * radius * radius;
  return result;
}
```

- Almost everything is done via functions
  - see ***main*** function
- Functions have to be declared
  - e.g. in area.h
  - here: function area takes one argument (radius) and returns a double
- ***#ifndef***, #***define*** and #***endif*** ensures that function is only declared once
- Comments start with *//*
- Comments can be enclosed in */* ... */*
- Functions have to be defined
  - e.g. in area.cc
- ***M_PI*** is declared in ***cmath***
- Indirection via double variable ***result*** not necessary
- Indentation makes code readable

Using Functions

calcarea.cc

```
#include <iostream>

#include "area.h"

int main()
{
  double radius;
  std::cout << "Please enter radius: ";
  std::cin >> radius;
  std::cout << "Area is "
            << area(radius)
            << std::endl;
  return 0;
}
```

```
$> g++ -o calcarea calcarea.cc area.cc
$> ./calcarea
Please enter radius: 2.0
Area is 12.5664
$>
```

- Inclusion of *area.h* makes compiler aware of whatever is declared in *area.h*
  - *area.h* can declare more than one function
- ***cin*** reads from standard input
  - declared in ***iostream***

Intermezzo - Compiler Warnings and Errors

calcarea.cc

```
#include <iostream>

#include "area.h"

int main()
{
  double radius;
  std::cout << "Please enter radius: ";
  std::cin >> radius
  ...
```

```
$> g++ -o calcarea calcarea.cc area.cc
calcarea.cc: In function ‘int main()’:
calcarea.cc:10: error: expected `;' before ‘std’
$>
```

- Compiler warning and error messages can give a very good idea of what is going wrong
- They might be cryptic sometimes
- Don´t be afraid of them

```
$> g++ -o calcarea calcarea.cc
Undefined symbols for architecture x86_64:
  "area(double)", referenced from:
      _main in cc5DZqXF.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
$>
```

- Declaration of function area is not enough
- Executable also needs actual definition of function
- This is the message you get on a 64Bit MacBook with g++ 4.2.1

Built-in Types

Type	Meaning	Size	Range	Resolution
int	integer	32 Bits	± 2147483648	1
long	long integer	≥ 32 Bits	± 2147483648	1
short	short integer	≥ 16 Bits	± 32768	1
char	character	8 Bits	± 127	1
float	single precision floating point	32 Bits	$\pm 3 \cdot 10^{\pm 38}$	$1.2 \cdot 10^{-7}$
double	double precision floating point	64 Bits	$\pm 2 \cdot 10^{\pm 308}$	$2.2 \cdot 10^{-16}$
bool	boolean	8 Bits	0/1, true/false	1
unsigned short	unsigned short integer	16 Bits	0 -> 65535	1

- Size depends on actual system architecture
  - long: 64Bit MacBook: 64 bit
- ISO standard defines limits that have to be guarantied by systems/compilers
- All integer types are also available as unsigned

from limits.cc:

```
#include <limits>
...
  std::cout << "Range:     " << std::numeric_limits<float>::min() << "   "
...
```

Operators I - Arithmetic Operators

Operator	Meaning
-	sign change
*	multiplication
/	division
%	modulus
+	addition
-	subtraction

```
$> g++ -o operatorsI operatorsI.cc
$> ./operatorsI
a = 1
a = -2
b = 12
b = 9
a = 13
a = 1
$>
```

- Evaluate right side of **=**
- Assign result to left side of **=**

operatorsI.cc

```
#include <iostream>

int main()
{
  int a = 1;
  std::cout << "a = " << a << std::endl;

  a = -2;
  std::cout << "a = " << a << std::endl;

  float b = 0;
  b = 4 * 3;
  std::cout << "b = " << b << std::endl;
  b = b - 3;
  std::cout << "b = " << b << std::endl;

  ...

  return 0,
}
```

Operators II

operatorsII.cc

```
#include <iostream>

int main()
{
  int a = 1;

  a -= 3;

  int b = 1;
  b++;
  b--;

  a = ++b;
  a = b++;

  return 0,
}
```

same as ***a = a -3;*** *a* is now *-2*

same as ***b = b + 1;*** b is now 2

same as ***b = b - 1;*** b is again 1

pre-increment

same as ***b = b + 1;***

the new value of b is assigned to a

a = 2; b = 2

post-increment

same as ***b = b + 1***

the old value is assigned to a

a = 2; b = 3

a += 2 is the same as ***a = a + 2***
a *= 2 is the same as ***a = a * 2***
a /= 2 is the same as ***a = a / 2***

Operators III - Relational Operators

Operator	Meaning
==	equal
!=	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

```
$> g++ -o operatorsIII \
          operatorsIII.cc
$> ./operatorsIII
a != 0
b == 0
a >= b
b != 0 : b = 3
b == 0 : b = 0
$>
```

- Relational operators evaluate to a boolean
  - 0 or 1, false or true
- Beware: **==** is a comparison whereas **=** is an assignment
- Assignments also have a value

operatorsIII.cc

```
#include <iostream>

int main()
{
  int a = 1, b = 0;

  if (a!=0) std::cout << "a != 0" << std::endl;
  if (b==0) std::cout << "b == 0" << std::endl;

  if (a>=b) std::cout << "a >= b" << std::endl;

  if ((b=3)) std::cout << "b != 0 : b = " << b << std::endl;
  if ((b=0)==0) std::cout << "b == 0 : b = " << b << std::endl;

  return 0;
}
```

Operators IV - Logical Operators

Operator	Meaning
!	not
!=	exclusive or
||	or
&&	and

```
$> g++ -o operatorsIV \
          operatorsIV.cc
$> ./operatorsIV
a != 0
!b
a != 0 && b == 0
a != 0 || b != 0
a != 0 != b != 0
$>
```

operatorsIV.cc

```
#include <iostream>

int main()
{
  int a = 1, b = 0;

  if (!(a==0)) std::cout << "a != 0" << std::endl;

  if (!b) std::cout << "!b" << std::endl;

  if (a!=0 && b==0)
    std::cout << "a != 0 && b == 0" << std::endl;

  if (a!=0 || b!=0)
    std::cout << "a != 0 || b != 0" << std::endl;

  if (a!=0 != b!=0)
    std::cout << "a != 0 != b != 0" << std::endl;

  return 0;
}
```

Operators V - Bitwise Operators

Operator	Meaning
~	compliment
&	and
|	or
^	exclusive or

- Bit by bit operation on integer types
- Don´t confuse logical operators and bitwise operators

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
4	0	0	0	0	0	1	0	0
10	0	0	0	0	1	0	1	0

- 4 && 10 = true (any non-zero value is true)
- 4 & 10 = 0

```
$> g++ -o operatorsV \
          operatorsV.cc
$> ./operatorsV
a & b = 0
a | b = 7
a & c = 4
a | c = 7
...
$>
```

operatorsV.cc

```
int main()
{
  int a = 0x4; // bit pattern 0100
  int b = 0x3; // bit pattern 0011
  int c;

  std::cout << "a & b = " << (a & b) << std::endl;
  std::cout << "a | b = " << (a | b) << std::endl;
  c = a | b;
  std::cout << "a & c = " << (a & c) << std::endl;
  std::cout << "a | c = " << (a | c) << std::endl;
...
```

Operators VI - Input and Output Operators

operatorsVI.cc

```
#include <iostream>

int main()
{
  int i;
  double d;
  std::cout << "Please enter an integer : ";
  std::cin >> i;
  std::cout << "Please enter a double   : ";
  std::cin >> d;

  std::cout << "The integer is " << i
            << " and the double is " << d
            << std::endl;

  std::cerr << "no error\n";

  return 0;
}
```

- Every program has three default input and output streams
  - ***cin*** is the so-called standard input
  - ***cout*** is the standard output
  - ***cerr*** is the error output
- **<<** is the output operator
- **>>** is the input operator
- more on file input and output follows later
- Depending on your shell you can redirect ***cout*** and ***cerr*** to files, i.e. a log file

```
$> g++ -o operatorsVI \
          operatorsVI.cc
$> ./operatorsVI
Please enter an integer : 3
Please enter a double   : 2
The integer is 3 and the double is 2
no error
$>
```

Numerical Functions

Function	Meaning	Remark
sin(x)	sine	
cos(x)	cosine	
tan(x)	tangent	
asin(x)	arc sine	
acos(x)	arc cosine	
atan(x)	arc tangent	π/2 < result < π/2
atan(x, y)	arc tangent x/y	-π < result < π
exp(x)	exponential	
log(x)	natural logarithm	
log10(x)	logarithm with base 10	
abs(x)	absolute value	
fabs(x)	absolute value	
sqrt(x)	sqare root	
pow(x, y)	x to the power of y	only for x > 0
pow(x, i)	x to the integer power i	also for x < 0

- Available via ***#include <cmath>***
- Don´t forget the ***std::***

Type Conversions

- C/C++ has many pre-defined type conversions that are applied automatically
  - integer types (int, short, char, long long) to floating point types (float, double)
    - values may be truncated
    - see table with types and corresponding ranges
  - floating point types to integer types
    - the number is truncated towards zero: 1.7 ➟ 1, -2.3 ➟ -2
  - number types to bool: 0 ➟ false, non-zero ➟ true
  - arithmetic expressions between integers result in integers
    - 3/2 ➟ 1, 7/8 ➟ 0
  - arithmetic expressions between floats (and integers) result in floats
    - 1.5*2.0 ➟ 3, 4.0/5 ➟ 0.8, 4/5.0 ➟ 0.8
- Explicit conversion from one type to the other: casts

```
double d = 3.45;
int i = static_cast<int>(d); // explicit cast from double to integer
```

  - other casts exist: ***dynamic_cast*** and ***reinterpret_cast***
  - the ***<int>*** indicates that ***static_cast*** is a template function
  - some details on templates follow later

Control Structures - If - Else

```
double maximum1(double a, double b)
{
  double result;
  if (a > b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}

double maximum2(double a, double b)
{
  double result = (a > b) ? a : b;
  return result;
}

double maximum3(double a, double b)
{
  return (a > b) ? a : b;
}
```

```
...
  double result = std::sqrt( (v >= 0) ? v : -v );
...
```

- Code in block after ***if*** is executed if condition is ***true***
  - condition in brackets after ***if***
- A single statement after ***if(...)***, ***else if(...)*** or ***else*** does not require curly braces but improves readability
- Multiple statements that are to be executed conditionally must be enclosed in curly braces
- ***? :*** is a special operator that allows inline conditional code
  - no temporary variable needed in the example
  - can be used also as argument to a function

Control Structures - While, Do - While Loops

```
double power(double x, int n)
{
  double result = 1;
  int i = 0;
  while (i<n) {
    result *= x;
    ++i;
  }
  return result;
}

double exponential(double x)
{
  /*
    exp(x) = 1 + x + x^2/2 + ... x^i/i!
  */
  double result = 1, xx = 1;
  int i = 1;
  do {
    xx *= x/i;
    result += xx;
    ++i;
  } while (xx > 0.0000001 * result);
  return result;
}
```

The block is executed as long as ***i < n***

The block may be skipped in case ***n = 0***

The block is executed as long as condition is true

The block is executed at least once

Control Structures - For Loops

```
double power(double x, int n)
{
  double result = 1;
  int i = 0;
  while (i<n) {
    result *= x;
    ++i;
  }
  return result;
}

double power(double x, int n)
{
  double result = 1;
  for (int i = 0; i<n; ++i) {
    result *= x;
  }
  return result;
}
```

- A ***for***-loop is the same as a ***while***-loop
- ***for (initialization; condition; increment) ...***
- ***initialization*** part can have multiple comma-separated statements
- ***increment*** part can have multiple comma-separated statements

Classes - The Basics

Vector3D.h

```
#ifndef Vector3D_h_
#define Vector3D_h_

class Vector3D
{
  public:
    double x_, y_, z_;
};

#endif // Vector3D_h_
```

- A class can be considered a new type
- A class can have member variables
- A class can have member functions and operators
- public means accessible from everywhere
- member variables are typically named such that they can be distinguished from local variables
  - here: _
  - Your software projects coding conventions will tell you what to use

calcVectorLength.h

```
#ifndef calcVectorLength_h_
#define calcVectorLength_h_

#include "Vector3D.h"

double calcVectorLength(Vector3D);

#endif // calcVectorLength_h_
```

calcVectorLength.cc

```
#include <cmath>

#include "calcVectorLength.h"

double calcVectorLength(Vector3D v)
{
  return std::sqrt(v.x_*v.x_ + v.y_*v.y_ + v.z_*v.z_);
}
```

Classes - The Basics cont.

useVector3D.cc

```
#include <iostream>

#include "Vector3D.h"
#include "calcVectorLength.h"

int main()
{
  Vector3D v;
  v.x_ = 2.7;
  v.y_ = 4.2;
  v.z_ = 9.8;
  std::cout << "The length of Vector3D v is "
            << calcVectorLength(v) << std::endl;
  Vector3D w = v;
  std::cout << "The length of Vector3D w is "
            << calcVectorLength(w) << std::endl;
  return 0;
}
```

Create a ***Vector3D*** with variable name ***v***

Set the x, y, z components

Create another ***Vector3D*** with variable name ***w*** and assign ***Vector3D v*** to it

```
$> g++ -o useVector3D useVector3D.cc
$> ./useVector3D
The length of Vector3D v is 10.9986
The length of Vector3D w is 10.9986
$>
```

Classes - Constructors & Destructor

Vector3D.h

```
#ifndef Vector3D_h_
#define Vector3D_h_

class Vector3D
{
  public:
    Vector3D();
    Vector3D(double x, double y, double z);
    ~Vector3D();
    double x_, y_, z_;
};

#endif // Vector3D_h_
```

Vector3D.cc

```
#include "Vector3D.h"

Vector3D::Vector3D()
  :x_(0), y_(0), z_(0) { }

Vector3D::Vector3D(double x, double y, double z)
  :x_(x), y_(y), z_(z) { }

Vector3D::~Vector3D() { }
```

- A class can have several constructors
- The one without argument is called the default constructor
- All initialization is done in or triggered by the constructor
- A class has one destructor
- The destructor is called when the object is deleted
- Cleanup should be done in the destructor

Intermezzo - Pointers

Pointers.cc

```
#include <iostream>

int main()
{
  int i = 1234;
  int * p = &i;
  std::cout << "i    = " << i << std::endl;
  std::cout << "p    = " << p << std::endl;
  std::cout << "*p   = " << *p << std::endl;

  *p = 4321;
  std::cout << "i    = " << i << std::endl;
  std::cout << "p    = " << p << std::endl;
  std::cout << "*p   = " << *p << std::endl;

  int **pp = &p;
  std::cout << "pp   = " << pp << std::endl;
  std::cout << "*pp  = " << *pp << std::endl;
  std::cout << "**pp = " << **pp << std::endl;

  **pp = 5678;
  std::cout << "i    = " << i << std::endl;
  std::cout << "*p   = " << *p << std::endl;
  std::cout << "**pp = " << **pp << std::endl;

  return 0;
}
```

- Every object (int, double, instance of a class) is located somewhere in memory
- Pointers point to the address of the object in memory
- Pointers are objects themselves and therefore have a location in memory
- Pointers have knowledge of the type of object they point to
- Pointers can point to other pointers

```
$> g++ -o Pointers Pointers.cc
$> ./Pointers
i    = 1234
p    = 0x7fff5fbfdb9c
*p   = 1234
i    = 4321
p    = 0x7fff5fbfdb9c
*p   = 4321
pp   = 0x7fff5fbfdb90
*pp  = 0x7fff5fbfdb9c
**pp = 4321
i    = 5678
*p   = 5678
**pp = 5678
$>
```

Intermezzo - References

References.cc

```
#include <iostream>

int main()
{
  int i = 1234;
  int & r = i;

  std::cout << "i = " << i << std::endl;
  std::cout << "r = " << r << std::endl
              << std::endl;

  r = 4321;
  std::cout << "i = " << i << std::endl;
  std::cout << "r = " << r << std::endl;

  return 0;
}
```

- A reference is like the object itself
- It is just a reference to the object, not a copy
- References are very similar to pointers
- References have knowledge of the type of object they reference
- References did not exist in standard C

```
$> g++ -o References References.cc
$> ./References
i = 1234
r = 1234

i = 4321
r = 4321
$>
```

Intermezzo - Passing Arguments

useVector3D_2.cc

```
#include <iostream>

int main()
{
  Vector3D v(2.7, 4.2, 9.8);
  Vector3D *p = &v;
  Vector3D &r = v;

  std::cout << "start of test"
            << std::endl;

  std::cout << "by value" << std::endl;
  calcVectorLength(v);

  std::cout << "by pointer"
            << std::endl;
  calcVectorLengthByPointer(p);

  std::cout << "by reference"
            << std::endl;
  calcVectorLengthByReference(r);

  std::cout << "end of test"
            << std::endl;

  return 0;
}
```

- Arguments can be passed to functions in three different ways
  - by value
  - by pointer
  - by reference
- Passing by value is always creating a copy of the object
- Use references if possible

```
$> g++ -o useVector3D_2 useVector3D_2.cc
$> ./useVector3D_2
Vector3D::Vector3D(double x, double y, double z)
start of test
by value
Vector3D::Vector3D(const Vector3D &v)
Vector3D::~Vector3D()
by pointer
by reference
end of test
Vector3D::~Vector3D()
$>
```

Classes - Member Functions

Vector3D.h

```
#ifndef Vector3D_h_
#define Vector3D_h_

class Vector3D
{
  public:
    double length();
    double x_, y_, z_;
};

#endif // Vector3D_h_
```

Vector3D.cc

```
#include <cmath>

#include "Vector3D.h"

Vector3D::length()
{
  return std::sqrt(x_*x_ + y_*y_ + z_*z_);
}
```

- Calculating the length of a vector is not nicely implemented
- It is not done in a C++ kind of way
- The vector knows how its length is calculated
- We don´t need an extra function, instead we should be able to ask the ***Vector3D*** for its length
- Add a so-called member function ***length()*** that does the job
- Member functions have direct access the member variables

Classes - Getter & Setter Functions

Vector3D.h

```
#ifndef Vector3D_h_
#define Vector3D_h_

class Vector3D
{
  public:
    void setX(double x) { x_ = x; }
    double getX() const { return x_; }
    ...
    double length();

  private:
    double x_, y_, z_;
};

#endif // Vector3D_h_
```

Vector3D.cc

```
#include <cmath>

#include "Vector3D.h"

Vector3D::length()
{
  return std::sqrt(x_*x_ + y_*y_ + z_*z_);
}
```

- Member variables should be hidden from direct public access
- Make ***x_***, ***y_*** and ***z_*** private
- Does not change implementation of ***length()*** member variables are always accessible within member functions
- Getter and setter functions are provided to allow public access to the vector components
- ***const*** in getter function means that calling the function does not change the object itself
- By using getter and setter functions one decouples the functionality of the class from the actual member variables
  - ***Vector3D*** could use polar coordinates for storage and the user would not have to bother about it
  - The data members can change without having an effect on the user as long as the interfaces stay as they are

Classes - Constness

Vector3D.h

```
#ifndef Vector3D_h_
#define Vector3D_h_

class Vector3D
{
  public:
    void setX(double x) { x_ = x; }
    void getX() const { return x_; }
    ...
    double scalarProduct(const Vector3D & v);
    ...
};

#endif // Vector3D_h_
```

Vector3D.cc

```
double Vector3D::scalarProduct(const Vector3D & v)
{
  return getX() * v.getX() + getY() * v.getY() + getZ() * v.getZ();
}
```

- A (member) function taking a reference to an object as argument can in principle modify the object
- This can be prevented by passing a ***const*** reference
- Within the function, only member functions declared as ***const*** can then be used
- ***const*** after the function declaration tells the compiler that the function may only be used for constant objects
- ***const*** is a promise that the object is not changed
- ***void setX(double x) const { x_ = x; }*** does not make any sense and results in a compilation error

Intermezzo - Object Lifetime, Scope, Heap & Stack

Scope.cc

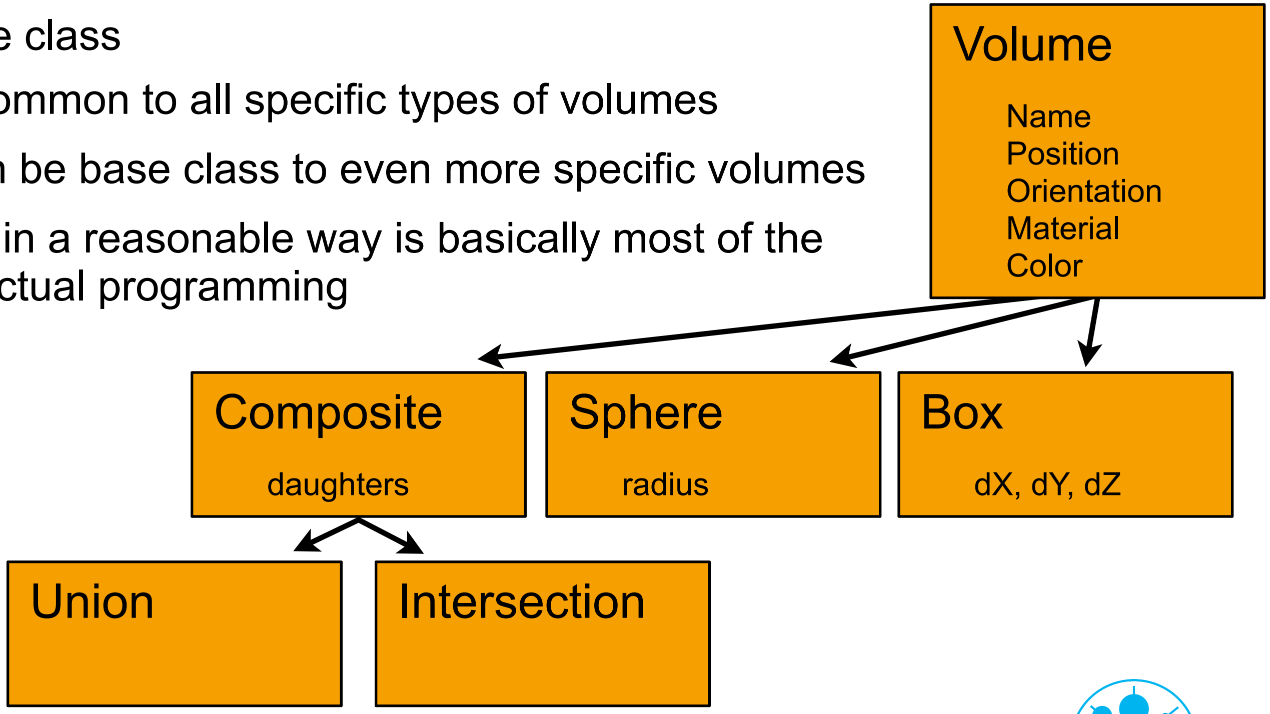
```
void simpleFunction1()
{
  Vector3D v;
}

void loopFunction()
{
  for (int i=0;i<2;i++) {
    Vector3D v;
  }
}

int main()
{
  simpleFunction1();
  loopFunction();

  Vector3D v1;
  Vector3D * v2;
  Vector3D * v3;

  v2 = new Vector3D();
  v3 = new Vector3D();

  delete v2;

  return 0;
}
```

- ***Vector3D v*** is created on the stack
- Objects created on the stack are automatically deleted when the variable goes out of scope
- ***Vector3D * v2 = new Vector3D()*** is created on the heap
- An object created on the heap lives until it is explicitly deleted
- Not deleting results in a memory leak

```
$> g++ -o Scope Scope.cc
$> ./Scope
Vector3D::Vector3D()
Vector3D::~Vector3D()
Vector3D::Vector3D()
Vector3D::~Vector3D()
Vector3D::Vector3D()
Vector3D::~Vector3D()
Vector3D::Vector3D()
Vector3D::Vector3D()
Vector3D::Vector3D()
Vector3D::~Vector3D()
Vector3D::~Vector3D()
$>
```

Class Hierarchies - Introduction

- In HEP we use Monte Carlo simulations to study physics processes
  - Need a description of the detector geometry - this will be our example
- The most generic object within a geometry is a ***Volume***
  - Volume will be our base class
  - It contains everything common to all specific types of volumes
- Specific types of volumes (***Box***, ***Sphere***, etc.) inherit functionality and members from base class
  - ***Volume*** will be our base class
  - It contains everything common to all specific types of volumes
- ***Box***, ***Sphere***, etc can in turn be base class to even more specific volumes
- Defining the class hierarchy in a reasonable way is basically most of the work and does not involve actual programming



Class Hierarchies - Volume Base Class

Volume.h

```
class Volume
{
 public:
  Volume(const std::string &name);
  virtual ~Volume() { }

  const std::string & getName() const { return name_; }
  const Vector3D& getPosition() const
  { return position_; }

  virtual double getVolume() const = 0;
  virtual bool isInside(double x, double y, double z) const
  { return false; }

 protected:
  void setPosition(const Vector3D& p) { position_ = p; }
  void setPosition(double x, double y, double z) {
    position_.setX(x);
    position_.setY(y);
    position_.setZ(z); }

 private:
  std::string name_;
  Vector3D position_;
};
```

- Base class contain all common member variables and member functions
  - name and position is common to all volumes
- strings will be covered later
- keyword ***virtual*** allows for reimplementation of member function in derived class
- ***virtual ... = 0*** makes class purely virtual
  - Instance of ***Volume*** can not be constructed
  - An instance of a derived class is only constructible if ***getVolume*** is implemented
- ***protected*** allows access within derived class but prevents public access

Class Hierarchies - Box Class

Box.h

```
class Box : public Volume
{
 public:
  Box(const std::string &name,
      double dx, double dy, double dz,
      double px, double py, double pz);
  virtual ~Box() { }
  ...
  double getVolume() const;
  bool isInside(double x, double y, double z) const;

 private:
  double dX_, dY_, dZ_;
};
```

- ***Box*** is derived from ***Volume*** and inherits common member variables and member function
- ***Box*** re-implements ***isInside***
- ***Box*** implements ***getVolume***
- The constructor takes the dimensions and position of the box
- The position has to be set from within ***Box*** since ***Volume::setPosition*** is ***protected***

testGeometry.cc

```
#include "Box.h"

int main()
{
  Volume v("test");
  Box b("Box",
         1, 2, 3,
         10, 11, 12);
  b.setPosition(0,0,0);
```

```
$> g++ -o testGeometry testGeometry.cc
error: cannot declare variable ‘v’ to be of
abstract type ‘Volume’
...
error: ‘void Volume::setPosition(double,
double, double)’ is protected
$>
```

Class Hierarchies - Box Class cont.

Box.cc

```
Box::Box(const std::string &name,
          double dx, double dy, double dz,
          double px, double py, double pz)
  :Volume(name), dX_(dx), dY_(dy), dZ_(dz) {
  setPosition(px, py, pz);
}

double Box::getVolume() const
{
  return getDX()*getDY()*getDZ();
}

bool Box::isInside(double x, double y, double z) const
{
  return ((getPosition().getX() - getDX()/2.0 < x &&
           getPosition().getX() + getDX()/2.0 > x) &&
          (getPosition().getY() - getDY()/2.0 < x &&
           getPosition().getY() + getDY()/2.0 > x) &&
          (getPosition().getY() - getDZ()/2.0 < x &&
           getPosition().getY() + getDZ()/2.0 > x));
}
```

- ***position_(px, py, pz)*** would not work because ***Volume::position_*** is ***private***
- One has to use ***setPosition(...)***

- ditto here
- ***Volume::position_*** being private forces one to use ***getPosition()***

Class Hierarchies - Composite Class

Composite.h

```
class Composite : public Volume
{
 public:
  Composite(const std::string &name,
            double px=0, double py=0, double pz=0);
  virtual ~Composite();

  void addDaughter(Volume * daughter);

 private:
  int nDaughters;
  Volume * daughter_[10];
};
```

Composite.cc

```
Composite::~Composite()
{
  for (int i=0;i<nDaughters;++i) {
    delete daughter_[i];
  }
}

void Composite::addDaughter(Volume * daughter)
{
  daughter_[nDaughters++] = daughter;
}
```

- positions ***px***, ***py*** and ***pz*** default to 0 - constructor only needs the name
- ***getVolume*** and ***isInside*** are not implemented
  - An instance of ***Composite*** can not be constructed
  - Only derived classes ***Union*** and ***Intersection*** can know how to implement ***getVolume*** and ***isInside***
- Daughter volumes are stored in an ***array*** with maximum size of 10
  - In a real implementation one would use something dynamic
- ***Composite*** takes over ownership of daughter volumes
  - One has to take care of deleting daughter volumes in destructor

Class Hierarchies - Composite Class cont.

testGeometry.cc

```
int main()
{
  Composite comp("Composite");
  comp.addDaughter(new Box("Box1",
                              1, 1, 1,
                              -0.5, 0, 0));

  Box * box2 = new Box("Box2",
                        1, 1, 1,
                        +0.5, 0, 0);
  comp.addDaughter(box2);

  delete box2;

  return 0;
}
```

```
$> g++ -o testGeometry testGeometry.cc
$> ./testGeometry
testGeometry(45765) malloc: *** error for object
0x100100200: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap
$>
```

- Daughter volumes are owned by ***Composite***
- Double delete is possible
- A good design should prevent this

- Reminder: Passing by reference is better than passing by pointer
- What about ***addDaughter*** ?
- Passing by reference is not a solution because we want to pass ownership to ***Composite***
- Any idea?

Templates - Introduction

Array.h

```
template <typename T, int size>
class Array
{
 public:
  Array() {
    for (int i=0;i<size;++i) storage_[i] = 0;
  }
  ~Array() {
    for (int i=0;i<size;++i) delete storage_[i];
  }
  void addAt(T* object, unsigned int position) {
    if (position>=size) return;
    if (storage_[position]!=0)
      delete storage_[position];
    storage_[position] = object;
  }
  T* at(unsigned int position) {
    if (position>=size) return 0;
    return storage_[position];
  }

 protected:
  T* storage_[size];
};
```

- The template mechanism allows to pass type names as a parameter to a function or a class
- Example: trivial array template class
  - Ownership of objects is taken over by ***Array***
  - ***Array*** has to delete elements
  - Implementation actually has a memory leak - any idea?

useArray.cc

```
int main()
{
  Array<Vector3D,10> array;

  array.addAt(new Vector3D, 2);
  array.addAt(new Vector3D, 1);

  array.addAt(new Vector3D, 10);
}
```

The Standard Template Library (STL) Containers

- A container is an object that contains other objects
- The C++ standard library provides containers and iterators (STL framework)
- Containers can be lists, vectors, maps, etc. which optimized for specific task and performance
- The type of object stored in the container is one of the parameters for the container template class

testVector.cc

```
#include <vector>
#include <iostream>

void printVector(const std::vector<int> &v) {
  for (unsigned int i=0;i<v.size();++i)
    std::cout << v[i] << " ";
  std::cout << std::endl;
}

int main()
{
  std::vector<int> v(5); // reserve space for 5 ints
  printVector(v);
  v.push_back(4); // add one more int
  printVector(v);
  v[0] = 1; // set new value at index 0
  printVector(v);
}
```

```
$> g++ -o testVector testVector.cc
$> ./testVector
0 0 0 0 0
0 0 0 0 0 4
1 0 0 0 0 4
$>
```

STL Container Overview

Name	Description	Comment
vector	Vector	an array with dynamic size
deque	Double ended queue	fast insertion of elements at any position
list	Double linked list	efficient insertion and movement of elements
stack	Last In - First Out (LIFO)	
queue	First In - FIrst Out (FIFO)	
set	associative container with values as keys	unique element values
multiset	associative container with values as keys allows multiple keys per value	
map	associative container with key/value storage	unique key values
multimap	associative container with key/value storage and multiple elements per key	
bitset	container for storing sequences of bits	

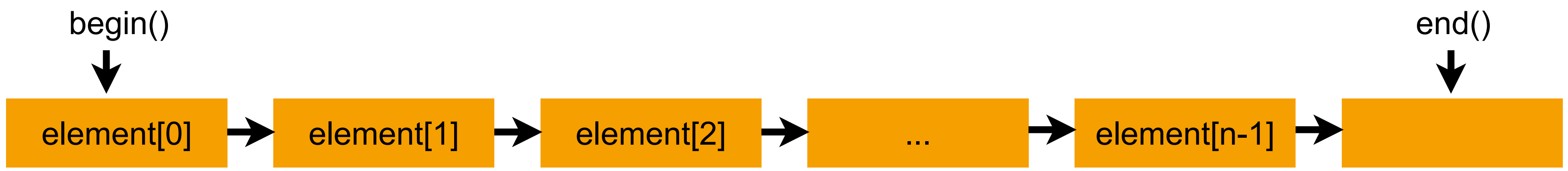
- Grouped in sequence containers, container adaptors and associative containers
- Basically same functionality provided by all containers - chose container for best performance

STL Container Function Overview

Operation	vector	list	queue	set	map	Description
push_back	X	X	X			add element at end of container
push_front		X				add element at beginning of container
insert	X	X		X	X	insert element at position
operator[]	X				X	return reference to element
at	X					return reference to element
front	X	X	X			returns ref. to first element
back	X	X	X			returns ref. to last element
pop_back	X	X				remove last element from container
pop_front		X	X			remove first element from container
erase	X	X		X	X	erase element from container
clear	X	X		X	X	remove all elements
size	X	X	X	X	X	returns the number of elements in the container
find				X	X	find an element via key
begin	X	X		X	X	returns iterator referring to the beginning of the cont.
end	X	X		X	X	returns iterator referring to the end of the container
rbegin	X	X		X	X	returns reverse iterator referring to the beginning
rend	X	X		X	X	returns reverse iterator referring to the end

STL Container - Iterators

- An iterator is an abstraction of a pointer to an element in a container
- Iterators can return the element in the collection they currently point to
- Iterators know where to find the next (and previous) element in the collection
- Iterators understand operations (++, --, +=, -=) and comparisons (==, !=)



testSTLIterators.cc

```
int main()
{
  std::vector<int> v;
  v.push_back(3); v.push_back(4); v.push_back(5);

  for (std::vector<int>::iterator it = v.begin();
       it!=v.end();
       ++it) std::cout << *it << " ";
  std::cout << std::endl;

  for (std::vector<int>::reverse_iterator it = v.rbegin();
       it!=v.rend();
       ++it) std::cout << *it << " ";
  std::cout << std::endl;
}
```

```
$> g++ -o testSTLIterators \
          testSTLIterators.cc
$> ./testSTLiterators
3 4 5
5 4 3
$>
```

STL Map Example

testSTLMap.cc

```
#include <map>
#include <iostream>

void printMap(const std::map<int,float> &m) {
  std::cout << m.size() << " elements: ";
  for (std::map<int,float>::const_iterator it = m.begin();
       it != m.end();
       ++it)
    std::cout << "(" <<it->first << "," << it->second << ") ";
  std::cout << std::endl;
}

int main()
{
  std::map<int,float> m;
  m[6] = 6.2;
  m[2] = 13.5;
  printMap(m);

  m[2] = 2.1;
  printMap(m);

  std::map<int,float>::iterator it = m.find(5);
  if (it != m.end()) std::cout << "value for key 5 is "
                                         << it->second << std::endl;
  it = m.find(6);
  if (it != m.end()) std::cout << "value for key 6 is "
                                         << it->second << std::endl;
}
```

Fill the map by key/value assignment

re-assignment

find keys

- A ***map*** is an associative container that stores key-value pairs
- Keys in the container are unique
- Container is sorted by keys
- The key can be accessed via ***iterator::first***
- The value can be accessed via ***iterator::second***
- ***find*** returns an iterator that points to the found element or is ***map::end()***

```
$> g++ -o testSTLMap testSTLMap.cc
$> ./testSTLMap
2 elements: (2,13.5) (6,6.2)
2 elements: (2,2.1) (6,6.2)
value for key 6 is 6.2
$>
```

STL Container - Algorithms

- The STL provides various algorithms that can be used on the container classes
- As an example we sort a vector of ***Vector3D*** by length

testSTLSort.cc

```
bool sortFunction(const Vector3D& v1, const Vector3D& v2) {
  return (v1.length()<v2.length());
}

int main()
{
  std::vector<Vector3D> v;
  v.push_back( Vector3D(3.1, 4.2, 6.1) );
  v.push_back( Vector3D(2.7, 1.3, 3.9) );
  v.push_back( Vector3D(7.3, 2.8, 5.2) );

  std::cout << "before sorting:" << std::endl;
  for (std::vector<Vector3D>::iterator it = v.begin();
       it!=v.end();
       ++it) it->print();

  std::sort(v.begin(), v.end(), sortFunction);

  std::cout << "after sorting:" << std::endl;
  for (std::vector<Vector3D>::iterator it = v.begin();
       it!=v.end();
       ++it) it->print();
}
```

Pass ***Vector3D*** by reference

Fill the vector

Sort the vector using ***sortFunction***

```
$> g++ -o testSTLSort testSTLSort.cc \
          Vector3D.cc
$> ./testSTLSort
before sorting:
Vector3D: (3.1, 4.2, 6.1)  length = 8.0287
Vector3D: (2.7, 1.3, 3.9)  length = 4.91833
Vector3D: (7.3, 2.8, 5.2)  length = 9.38989
after sorting:
Vector3D: (2.7, 1.3, 3.9)  length = 4.91833
Vector3D: (3.1, 4.2, 6.1)  length = 8.0287
Vector3D: (7.3, 2.8, 5.2)  length = 9.38989
$>
```

Random Notes - Strings

- At some point you will certainly have to print some sort of text from one of your programs
- A string is a sequence of characters
- The standard library provides a class ***string*** that helps to deal with character strings

Function	Description
append	append other string at the end
at	returns character at certain position
begin	returns iterator to the first character
end	returns iterator to the end of string
clear	empties the string

Function	Description
compare	compares to another string
erase	erases characters from the string
find	find another string within the string
insert	insert another string into the string
length	returns the number of characters

Strings1.cc

```
#include <string>
#include <iostream>

int main()
{
  std::string s = "Hello, World!";
  std::cout << s << std::endl;

  size_t pos = s.find("World");
  s.replace(pos, 6, "Hamburg!");
  std::cout << s << std::endl;
}
```

```
$> g++ -o Strings1 Strings1.cc
$> ./Strings1.cc
Hello, World!
Hello, Hamburg
$>
```

Random Notes - String Composition & Formatting

- Typically you want to compose a string with some proper formatting
  - Decimal point always at the same position
  - fixed number of digits after the decimal point ...
- The header files ***<sstream>*** (stringstream) and ***<iomanip>*** are your friends
- To many features to list them all

Strings2.cc

```
#include <string>
#include <sstream>
#include <iostream>

int main()
{
  std::ostringstream os;

  os << "Hello!!!\n\n";
  os << "pi = " << 3.1415926534 << "\n";
  os.precision(9);
  os << "pi = " << std::fixed << 3.1415926534 << "\n\n";

  os << "column 1 | column 2\n";
  os.width(8); os << 13 << " | ";
  os.width(8); os << 12.4 << "\n";
  os.width(8); os << 1 << " | ";
  os.width(8); os << 345.2 << "\n";

  std::string s = os.str();
  std::cout << s << std::endl;
}
```

```
$> g++ -o Strings2 Strings2.cc
$> ./Strings2
Hello!!!

pi = 3.14159
pi = 3.141592653

column 1 | column 2
      13 |   12.400
       1 |  345.200

$>
```

Random Notes - File I/O

- At some point you will have to either write data to a file or retrieve data from a file
- File I/O is covered by the header file ***<fstream>***
- For output to file there is a class ***ofstream*** that behaves just like ***cout***
- For input from file there is a class ***ifstream*** that behaves just like ***cin***
- What about reading data from a table in a file

ReadFile.cc

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
  ifstream ifile("data.txt");

  char buffer[80];
  while (ifile.peek()=='#') {
    ifile.getline(buffer, 80);
  }

  cout.precision(6);
  float x, y;
  while (ifile >> x >> y) {
    cout.width(12); cout << fixed << x;
    cout.width(12); cout << fixed << y << endl;
  }
}
```

```
$> g++ -o ReadFile ReadFile.cc
$> cat data.txt
# comments start with a '#'
#   x          y = sin(x)
0.000000     0.000000
0.314159     0.309017
...
$>
$>
$> ./ReadFile
    0.000000    0.000000
    0.314159    0.309017
...
$>
```

What is new in the example?

Random Notes - Namespaces

- Namespaces can be used to logically group things together
- Up to now you have seen the namespace ***std*** in this presentation
- ***using namespace std;*** allows to omit the ***std::*** before everything inside the namespace
- However, unless you know what you are doing, I would stick to the explicit ***std::***
- ***using namespace XXX;*** in header files can cause problems down the road

myVector.h

```
template<class T>
class vector
{
 public:
  vector() {}
};
```

someHeader.h

```
#include <vector>

using namespace std;

void foo() { /* do something */ }
```

testNamespace.cc

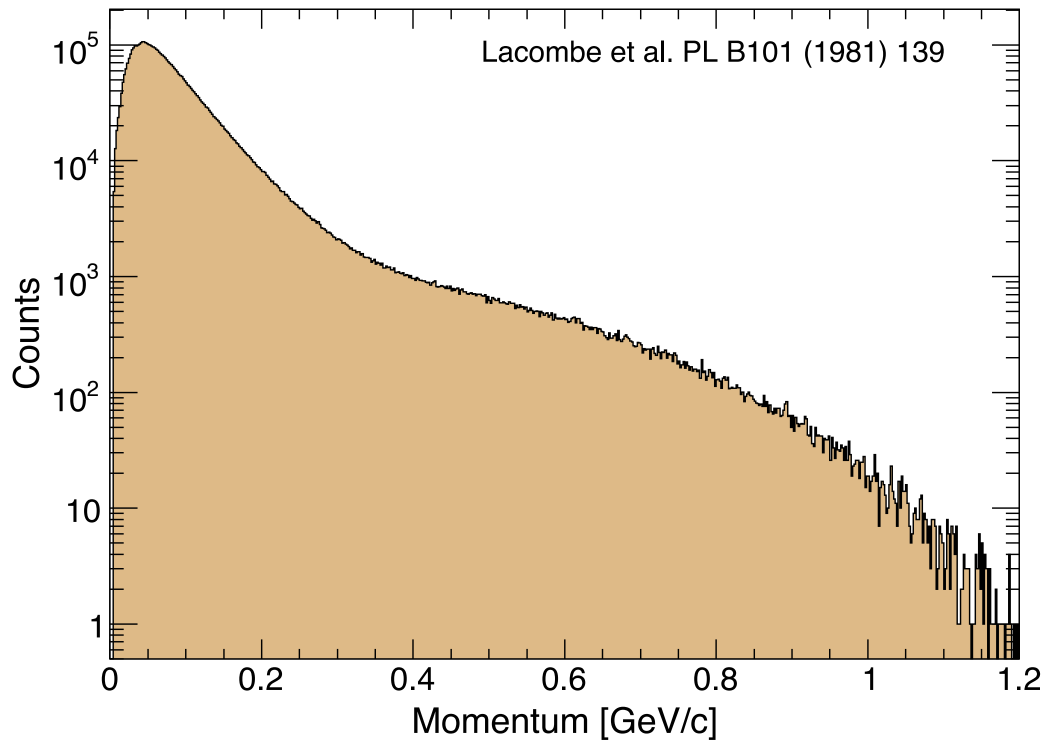
```
#include "myVector.h"
#include "someHeader.h"

int main()
{
  vector<int> x;
  return 0;
}
```

```
$> g++ -o testNamespace testNamespace.cc
testNamespace.cc: In function ‘int main()’:
testNamespace.cc:6: error: reference to ‘vector’ is ambiguous
...
$>
```

- Consider ***someHeader.h*** is not under our control
- ***someHeader.h*** tries to forces us to use ***std::vector***
- Code does not compile and it is not the fault of ***myVector.h*** and ***testNamespace.cc***
- If at all, use ***using namespace XXX*** inside a function definition

Putting Everything Together

- In HEP we love fancy plots
  - e.g. a histogram:



- In principle you have all C++ tools at hand to get something like this done
- How would you implement a simple histogram in C++ ?
  - without the fancy graphics
  - text output to screen is perfectly fine

Putting Everything Together - A Simple Histogram Class

Histo1D.h

```
class Histo1D
{
 public:
  Histo1D(int nBins, double min, double max);
  ~Histo1D();

  void fill(double x);
  void print() const;

 protected:

  int findBin(double x) const;
  double binEdgeMin(int bin) const;
  double binEdgeMax(int bin) const;

 private:
  int nBins_;
  double min_;
  double max_;
  std::map<int,int> content_;
};
```

- A Histogram has a finite number of bins, a upper and a lower edge
- Filling the histogram, i.e. incrementing the number of counts for a certain bin
- Printing the histogram onto the screen
- When filling the histogram we need to find the correct bin for a value on the x-axis
- For printing we need the upper and lower edges for each bin on the x-axis
- We need to store the size of the histogram in order to do the calculations (findBin etc)
- Content of histogram is stored in a map
- One could easily extend the class such that reading and writing the histo is possible
  - ***writeToFile(const std::string & filename)***
  - ***readFromFile(const std::string & filename)***

Testing the Simple Histogram Class

testHisto1D.h

```
int main()
{
  Histo1D h(20, 0.0, M_PI);

  for (int i=0;i<10000;++i) {
    double v = std::rand();
    double x = std::acos(2.0*(v/RAND_MAX)-1.0);
    h.fill(x);
  }

  h.print();

  return 0;
}
```

- Histogram is filled with ***acos*** taking [-1,1] as argument
- ***rand()*** returns a pseudo-random integer in the range [0,***RAND_MAX***]
- Please never use ***rand()*** for any scientific work
- There are real random number generators out there

```
$> g++ -o testHisto1D testHisto1D.cc \
          Histo1D.cc
$> ./testHisto1D
Number of entries: 10000
Maximum:           817
Underflows:        0
Overflows:         0
    0.00 -     0.16 : ##
    0.16 -     0.31 : ######
    0.31 -     0.47 : ########
    0.47 -     0.63 : ##########
    0.63 -     0.79 : #############
    0.79 -     0.94 : ###############
    0.94 -     1.10 : ################
    1.10 -     1.26 : ##################
    1.26 -     1.41 : ####################
    1.41 -     1.57 : ####################
    1.57 -     1.73 : ###################
    1.73 -     1.88 : ###################
    1.88 -     2.04 : #################
    2.04 -     2.20 : #################
    2.20 -     2.36 : ###############
    2.36 -     2.51 : #############
    2.51 -     2.67 : ###########
    2.67 -     2.83 : ########
    2.83 -     2.98 : #####
    2.98 -     3.14 : ##
$>
```

Summary

- C++ is a complicated but very powerful programming language
- Object-oriented programming allows
  - close coupling between data and functions that manipulate the data
  - hiding of the internal details while providing an interface
  - splitting the code into small pieces (classes) that are individually simple and easy to maintain
- C++ is easily extendable
  - Write your own class and you have extended C++ by a new type
  - STL is mighty powerful
- I have not covered quite a few things
  - Exceptions
  - Operator overloading
  - Design patters ...
- Reminder
  - The best way to learn a programming language is by looking at code
  - Not every written piece of code is good programming practice
  - When writing code, please do it such that also other people can understand
  - Stick to the coding conventions of the project you work on
- http://www.cplusplus.com is a good source for documentation and reference