Basic syntax in C++

Shahram Rahatlou

Computing Methods in Physics

`http://www.roma1.infn.it/people/rahatlou/cmp/`

Anno Accademico 2020/21

SAPIENZA
UNIVERSITÀ DI ROMA

Brief History of C

- C was developed in 1967 mainly as a language for writing operating systems and compilers
  - Think about the gcc compiler and Linux today
  - You can compile the gcc compiler yourself
  - You can get the latest linux kernel (core of the linux operating system) from www.kernels.org and compile it yourself

- C was the evolution of two previous languages: B and BCPL
  - Both used to develop early versions of UNIX at Bell Labs

- C became very popular and was ported to variety of different hardware platforms
  - C was standardized in 1990 by International Organization for Standardization (ISO) and American National Standards Institute (ANSI)
    - ANSI/ISO 9899: 1990

Object Oriented Programming and Birth of C++

- By 1970's the difficulties of maintaining very large software projects for companies and businesses had lead to structured programming
  - From Wikipedia:

    Structured programming can be seen as a subset or subdiscipline of procedural programming, one of the major programming paradigms. It is most famous for removing or reducing reliance on the GOTO statement (also known as "go to")

- By late 70's a new programming paradigm was becoming trendy: object orientation

- In early 1980's Bjarne Stroustrup developed C++ using features from C but adding capabilities for object orientation

What is 'Object Oriented Programming` anyway?

- Objects are software units modelled after entities in real life
  - Objects are entities with attributes: length, density, elasticity, thermal coefficient, color
  - Objects have a behavior and provide functionalities
    - A door can be opened
    - A car can be driven
    - A harmonic oscillator oscillates
    - A nucleus can decay
    - A planet moves in an orbit
- Object orientation means writing your program in terms of well defined units (called objects) which have attributes and offer functionalities
  - Program consists in interaction between objects using methods offered by each of them

C++ is not C !

- Don’t be fooled by the name!

- C++ was developed to overcome limitations of C and improve upon it
  - C++ looks like C but feels very differently
  - C++ shares many basic functionalities but improves upon many of them
    - For example input/output significantly better in C++ than in C

- C excellent language for structural programming
  - Focused around actions on data structures
  - Provides methods which act on data and create data

- C++ focused on inter-action between objects
  - Objects are ‘smart’ data structures: data with behavior!

What You Need to compile your C++ Program?

- On Linux machines you should have the g++ compiler installed by default
- On Windows you can use the C++ compiler provided by the free version of Visual Studio
- On Mac OS, you can install the g++ compiler via XCode, available for free on Mac App Store

- The easiest way is to use the virtual box available on the course website

Structure of a C++ Program

```
// your first C++ application!
#include <iostream> // required to perform C++ stream I/O

// function main begins program execution
int main() {
   return 0; // indicate that program ended successfully

} // end function main
```

Precompiler/Preprocessor Directives

What is the preprocessor? What does it do?

```
// your first C++ application!
#include <iostream> // required to perform C++ stream I/O

// function main begins program execution
int main() {
   return 0; // indicate that program ended successfully

} // end function main
```

`iostream` will be included before compiling this code!

What does the Preprocessor do?

```
// Foo.h
class Foo {
public:
 Foo() {};
 Foo(int a) { x_ = a; };

private:
 int x_;
};
```

```
// ExamplePreprocessor.cpp
#include "Foo.h"


int main() {

  return 0;
}
```

Pre-compile only

```
$ g++ -E ExamplePreprocessor.cpp > prep.cc
$ cat prep.cc
# 1 "ExamplePreprocessor.cpp"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "ExamplePreprocessor.cpp"

# 1 "Foo.h" 1
class Foo {
public:
 Foo() {};
 Foo(int a) { x_ = a; };

private:
 int x_;
};
# 3 "ExamplePreprocessor.cpp" 2

int main() {

  return 0;
}
```

- Replace user directives with requested source code
  - Foo.h is included in ExamplePreprocessor.cpp

Comments in C++

```
// your first C++ application!
#include <iostream> // required to perform C++ stream I/O

// function main begins program execution
int main() {
   return 0; // indicate that program ended successfully

} // end function main
```

- Comments preceded by //
  - Can start anywhere in the program either at the beginning or right after a statement in the middle of the line

Compiling a C++ application

```
$ g++ -o Welcome Welcome.cpp
```

Name of the binary output

C++ file to compile and link

```
$ ls -l
-r--r--r--  1 rahatlou None   1379 Apr 18 22:55
Welcome.cpp
-rwxr-xr-x  1 rahatlou None 476600 Apr 18 22:57
Welcome
```

- We will be using the free compiler gcc throughout the examples in this course

Some basic aspects of C++

- All statements must end with a semi-colon ;
  - Carriage returns are not meaningful and ignored by the compiler

- Comments are preceded by //
  - Comments can be an entire line or in the middle of the line after a statement

- Any C++ application must have a **`main`** method

- **`main`** must return an **`int`**
  - Return value can be used by user/client/environment
  - E.g. to understand if there was an error condition

What about changing a different type of **`main`**?

```
// VoidMain.cpp
#include <iostream>
using namespace std;

void main(){

  // no return type

} // end function main
```

```
$ g++ -o VoidMain VoidMain.cpp
VoidMain.cpp:6: error: `main' must return `int'
```

- Compiler requires **`main`** to return an **`int`** value!
- Users must simply must satisfy this requirement
  - If you need a different type there is probably a mistake in your design!

Typical Compilation Errors So Far

```
// BadCode1.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   int nIterations;

   cout << "How many
           iterations? "; // cannot break in the middle of the string!

   cin >> nIteration; // wrong name! the s at the end missing

   // print message to STDOUT
   cout << "Number of requested iterations: " << nIterations << endl;

   return 0 // ; is missing!

} // end of main
```

```
$ g++ -o BadCode1 BadCode1.cpp
BadCode1.cpp: In function `int main()':
BadCode1.cpp:9: error: missing terminating " character
BadCode1.cpp:10: error: `iterations' undeclared (first use this function)
BadCode1.cpp:10: error: (Each undeclared identifier is reported only once for each function
 it appears in.)
BadCode1.cpp:10: error: missing terminating " character
BadCode1.cpp:12: error: `nIteration' undeclared (first use this function)
BadCode1.cpp:12: error: expected `:' before ';' token
BadCode1.cpp:12: error: expected primary-expression before ';' token
BadCode1.cpp:19: error: expected `;' before '}' token
```

Some C reminders

Always initialise your variables!

```
// tinput_bad2.cc
#include <iostream>
using namespace std;

int main() {

  int iters;
  cout << "iters before cin: " << iters << endl;

  cout << "iterations? ";
  cin >> iters;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

Random value since not initialized!

```
$ g++ -Wall -o tinput_bad2 tinput_bad2.cc
$ ./tinput_bad2
iters before cin: 134514841
iterations? 3
requested 3 iterations
$ ./tinput_bad2
iters before cin: 134514841
iterations? er
requested 134514841 iterations
```

Arrays (same as in C)

```
// vect3.cc
#include <iostream>
using namespace std;

int main() {


   float vect[3] = {0.4,1.34,56.156}; // vector of int
   float v2[3];
   float v3[] = { 0.9, -0.1, -0.65}; // array of size 3

   for(int i = 0; i<3; ++i) {
     cout << "i: " << i << "\t"
          << "vect[" << i << "]: " << vect[i] << "    \t"
          << "v2[" << i << "]: " << v2[i] << " \t"
          << "v3[" << i << "]: " << v3[i]
           << endl;
   }

   return 0;
}
```

Index of arrays starts from 0 !!

v2[0] is the first elements of array **v2** of size 3.

v2[2] is the last element of **v2**

What happened to **v2**?

```
$ g++ -o vect3 vect3.cc
$ ./vect3
i: 0    vect[0]: 0.4          v2[0]: 5.34218e+36      v3[0]: 0.9
i: 1    vect[1]: 1.34         v2[1]: 2.62884e-42      v3[1]: -0.1
i: 2    vect[2]: 56.156       v2[2]: 3.30001e-39      v3[2]: -0.65
```

Arrays and Pointers

- The name of the array is a pointer to the first element of the array

```
// array.cpp
#include <iostream>
using namespace std;

int main() {

   int vect[3] = {1,2,3}; // vector of int
   int v2[3]; //what is the default value?
   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;
   int* c = vect;
   int* e = v2;

   for(int i = 0; i<5; ++i) {
     cout << "i: " << i << ", d = " << d << ", *d: " << *d;
     ++d;
     cout << ", c = " << c << ", *c: " << *c;
     ++c;
     cout << ", e = " << e << ", *e: " << *e << endl;
     ++e;
   }
   return 0;
}
```

What happened to `e`?

```
$ g++ -o array array.cc
$ ./array
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecc, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eeec, *e: 2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

Another bad example of using arrays

```
// vect2.cc
#include <iostream>
using namespace std;

int main() {


   float vect[3] = {0.4,1.34,56.156}; // vector of int
   float v2[3]; // use default value 0 for each element
   float v3[] = { 0.9, -0.1, -0.65, 1.012, 2.23, -0.67, 2.22 }; // array of size 7

   for(int i = 0; i<5; ++i) {
     cout << "i: " << i << "\t"
          << "vect[" << i << "]: " << vect[i] << "  \t"
          << "v2[" << i << "]: " << v2[i] << "  \t"
          << "v3[" << i << "]: " << v3[i]
           << endl;
   }

   return 0;
}
```

Accessing out of range component!

```
$ g++ -o vect2 vect2.cc
$ ./vect2
i: 0    vect[0]: 0.4            v2[0]: 5.34218e+36      v3[0]: 0.9
i: 1    vect[1]: 1.34           v2[1]: 2.62884e-42      v3[1]: -0.1
i: 2    vect[2]: 56.156         v2[2]: 3.30001e-39      v3[2]: -0.65
i: 3    vect[3]: 5.60519e-45    v2[3]: 1.57344e+20      v3[3]: 1.012
i: 4    vect[4]: 1.72441e+20    v2[4]: 0.4              v3[4]: 2.23
```

Example of Bad non-initialized Arrays

```
// vect1.cc
#include <iostream>
#include <cmath>

using namespace std;


int main() {

   float vect[3]; // no initialization

   cout << "printing garbage since vector not initialized" << endl;
   for(int i=0; i<3; ++i) {
     cout << "vect[" << i << "] = " << vect[i]
          << endl;
   }

   vect[0] = 1.1;
   vect[1] = 20.132;
   vect[2] = 12.66;

   cout << "print vector after setting values" << endl;
   for(int i=0; i<3; ++i) {
     cout << "vect[" << i << "] =    " << vect[i] << "\t"
          << "sqrt( vect[" << i << "] ) = " << sqrt(vect[i])
          << endl;
   }

   return 0;
}
```

```
$ ./vect1
printing garbage since vector not initialized
vect[0] = 2.62884e-42
vect[1] = NaN
vect[2] = 0
print vector after setting values
vect[0] =    1.1        sqrt( vect[0] ) = 1.04881
vect[1] =    20.132     sqrt( vect[1] ) = 4.48687
vect[2] =    12.66      sqrt( vect[2] ) = 3.55809
```

Control Statements in C++

```
// SimpleIf.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   if( 1 == 0 ) cout << "1==0" << endl;

   if( 7.2 >= 6.9 ) cout << "7.2 >= 6.9" << endl;

   bool truth = (1 != 0);
   if(truth) cout << "1 != 0" << endl;

   if( ! ( 1.1 >= 1.2 ) ) cout << "1.1 < 1.2" << endl;

   return 0;
} // end of main
```

```
$ g++ -o SimpleIf SimpleIf.cpp
$ ./SimpleIf
7.2 >= 6.9
1 != 0
1.1 < 1.2
```

Declaration and Definition of Variables

```
// SimpleVars.cpp
#include <iostream>
using namespace std;

int main() {

   int samples; // declaration only

   int events = 0; // declaration and assignment

   samples = 123; // assignment

   cout << "How many samples? " ;
   cin >> samples; // assigment via I/O

    cout << "samples: " << samples
         << "\t"  // insert a tab in the printout
         <<  "events: " << events
         << endl;

   return 0;
} // end of main
```

```
$ g++ -o SimpleVars SimpleVars.cpp
$ ./SimpleVars
How many samples? 3
samples: 3      events: 0
```

Loops and iterations in C++

```
int main() { // main begins here

   int nIterations;
   cout << "How many iterations? ";
   cin >> nIterations;

   int step;
   cout << "step of iteration? " ;
   cin >> step;

   for(int index=0; index < nIterations; index+=step) {
     cout << "index: " << index << endl;
   }
   return 0;
} // end of main
```

Maximum

Starting value

step

```
$ g++ -o SimpleLoop SimpleLoop.cpp
$ ./SimpleLoop
How many iterations? 7
step of iteration? 3
index: 0
index: 3
index: 6
```

Namespace, Pointers and References, Constants
Introduction to Class

Shahram Rahatlou

Computing Methods in Physics

`http://www.roma1.infn.it/people/rahatlou/cmp/`

Anno Accademico 2019/20

SAPIENZA
UNIVERSITÀ DI ROMA

Output with **iostream**

```
// SimpleIO.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   // print message to STDOUT
   cout << "Moving baby steps in C++!" << endl;

   return 0;

} // end of main
```

End of line
start a new line!

```
$ g++ -o SimpleIO SimpleIO.cpp
$ ./SimpleIO
Moving baby steps in C++!
```

- **iostream** provides output capabilities to your program

Input with `iostream`

```
// SimpleInput.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   int nIterations;

   cout << "How many iterations? ";
   cin >> nIterations;

   // print message to STDOUT
   cout << "Number of requested iterations: " << nIterations << endl;

   return 0;
} // end of main
```

Put content of cin into variable nIterations

```
$ g++ -o SimpleInput SimpleInput.cpp
$ ./SimpleInput
How many iterations? 7
Number of requested iterations: 7
```

- **`iostream`** provides also input capabilities to your program

Problems with **cin**

```
// tinput_bad.cc
#include <iostream>
using namespace std;

int main() {

  cout << "iterations? ";

  int iters;
  cin >> iters;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

```
$ g++ -Wall -o tinput_bad tinput_bad.cc
$ ./tinput_bad
iterations? 23
requested 23 iterations
$ ./tinput_bad
iterations? dfed
requested 134514793 iterations
```

Checking **cin** success or failure

```
//tinput.cc
#include <iostream>
using namespace std;

int main() {

  cout << "iterations? ";

  int iters = 0;
  cin >> iters;

  if(cin.fail()) cout << "cin failed!" << endl;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

Fails if input data doesn't match expected data type

```
$ g++ -Wall -o tinput tinput.cc
$ ./tinput
iterations? 34
requested 34 iterations
$ ./tinput
iterations? sfee
cin failed!
requested 0 iterations
```

Scope of Variables

```
// scope.cc
#include <iostream>


double f1() {
  double y = 2;
  return y;
}

int main() {

   double x = 3;
   double z = f1();

   std::cout << "x: " << x << ", z: " << z << ", y: " << y
             << std::endl;
   return 0;
}
```

```
$ g++ -o scope scope.cc
scope.cc: In function `int main()':
scope.cc:16: error: `y' undeclared (first use this function)
scope.cc:16: error: (Each undeclared identifier is reported
 only once for each function it appears in.)
```

What is the difference between **`cout`** and **`std::cout`**?

- The scope of a name is the block of program where the name is valid and can be used
  - A block is delimited by { }
  - It can be the body of a method, or a simple scope defined by the user using { }

What is **namespace** ?

- A mechanism to group declarations that logically belong to each other

```
namespace physics {
  class vector;
  class unit;
  class oscillator;
  void sort(const vector& value);
}

namespace electronics {
  void sort(const vector& value);
  class oscillator;
}

namespace graphics {
  void sort(const vector& value);
  class unit;
}
```

- Provides an easy way for logical separation of parts of a big project

- Basically a ‘scope’ for a group of related declarations

How do I use **namespaces** ?

```
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

int main() {
   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   std::cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   double z2 = foobar::mean(x,y);
   std::cout << "foobar::mean(" << x << "," << y << ") = " << z2
             << std::endl;
   return 0;
}
```

physics::mean

foobar::mean

Use "::" to specify the namespace

```
$ g++ -o namespace1 namespace1.cc
$ ./namespace1
physics::mean(3,4) = 3.5
foobar::mean(3,4) = 12.5
```

Defined in iostream

Common Errors with **namespaces**

```
// namespaceBad.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

int main() {

   double x = 3;
   double y = 4;

   double z1 = mean(x,y); // forgot the namespace!
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   return 0;
}
```

If you forget to specify the namespace the compiler doesn't know where to find the method

```
$ g++ -o namespaceBad namespaceBad.cc
namespaceBad.cc: In function `int main()':
namespaceBad.cc:15: error: `mean' undeclared (first use this function)
namespaceBad.cc:15: error: (Each undeclared identifier is reported only
 once for each function it appears in.)
namespaceBad.cc:16: error: `cout' undeclared (first use this function)
```

using namespace directive

```
// namespace2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

using namespace std; // make all names in std namespace available!

int main() {

   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << endl;

   return 0;
}
```

Provide default namespace for un-qualified names

Same concepts used also in python

Compiler looks for **cout** and **endl** first

if not found looks for **std::cout** and **std::endl**;

```
$ g++ -o namespace2 namespace2.cc
$ ./namespace2.exe
physics::mean(3,4) = 3.5
```

Be careful with **`using`** directive!

```
// namespaceBad2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

using namespace foobar;
using namespace physics;
using namespace std;

int main() {
   double x = 3;
   double y = 4;

   double z1 = mean(x,y);
   double z2 = mean(x,y);

   return 0;
}
```

Ambiguous use of method **`mean`**!

Is it in **`foobar`** or in **`physics`**?

```
$ g++ -o namespaceBad2 namespaceBad2.cc
namespaceBad2.cc: In function `int main()':
namespaceBad2.cc:21: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:  double foobar::mean(const double&, const double&)
namespaceBad2.cc:25: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:  double foobar::mean(const double&, const double&)
```

Some tips on **using** directive

```
// namespace3.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

void printMean(const double& a, const double& b) {
  double z1 = physics::mean(a,b);

  using namespace std; // using std namespace within this method!
  cout << "physics::mean(" << a << "," << b << ") = " << z1 << endl;
}

int main() {

   double x = 3;
   double y = 4;
   printMean(x,y);

   cout << "no namespace available in the main!" << endl;
   return 0;
}
```

Namespace defined only within printMean

```
$ g++ -o namespace3 namespace3.cc
namespace3.cc: In function `int main()':
namespace3.cc:23: error: `cout' undeclared (first use this function)
namespace3.cc:23: error: (Each undeclared identifier is reported only
once for each function it appears in.)
namespace3.cc:23: error: `endl' undeclared (first use this function)
```

No default namespace in the main()

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes and functions we are going to use instead of entire namespace

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: ???
in local scope before int, x: ???
in local scope after int, x: ???
in main after local scope, x: ???
```

What do you think the output is going to be?

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes and functions we are going to use instead of entire namespace

Changed value of x from main scope

Define new variable in this scope

Back to the main scope

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: 1.2
in local scope before int, x: 2.2
in local scope after int, x: 4
in main after local scope, x: 2.2
```

Functions and Methods

- A function is a set of operations to be executed
  - Typically there is some input to the function
  - Usually functions have a return value
  - Functions not returning a specific type are **`void`**

```
// func1.cc
#include <iostream>

double pi() {
  return 3.14;
}

void print() {
  std::cout << "void function print()" << std::endl;
}

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}
```

```
$ g++ -o func1 func1.cc
$ ./func1
pi: 3.14
void function print()
```

Functions must be declared before being used

```
// func2.cc
#include <iostream>

double pi() {
  return 3.14;
}

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}

void print() {
  std::cout << "void function print()" << std::endl;
}
```

Compiler does not know what the name **`print`** stands for!

No declaration at this point!

```
$ g++ -o func2 func2.cc
func2.cc: In function `int main()':
func2.cc:11: error: `print' undeclared (first use this function)
func2.cc:11: error: (Each undeclared identifier is reported only
 once for each function it appears in.)
func2.cc: In function `void print()':
func2.cc:16: error: `void print()' used prior to declaration
```

Definition can be elsewhere

```
// func3.cc
#include <iostream>

double pi() {
  return 3.14;
}

extern void print(); // declare to compiler print() is a void method

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}

// now implement/define the method void print()
void print() {
  std::cout << "void function print()" << std::endl;
}
```

```
$ g++ -o func3 func3.cc
$ ./func3
pi: 3.14
void function print()
```

Pointers and References

- A variable is a label assigned to a location of memory and used by the program to access that location

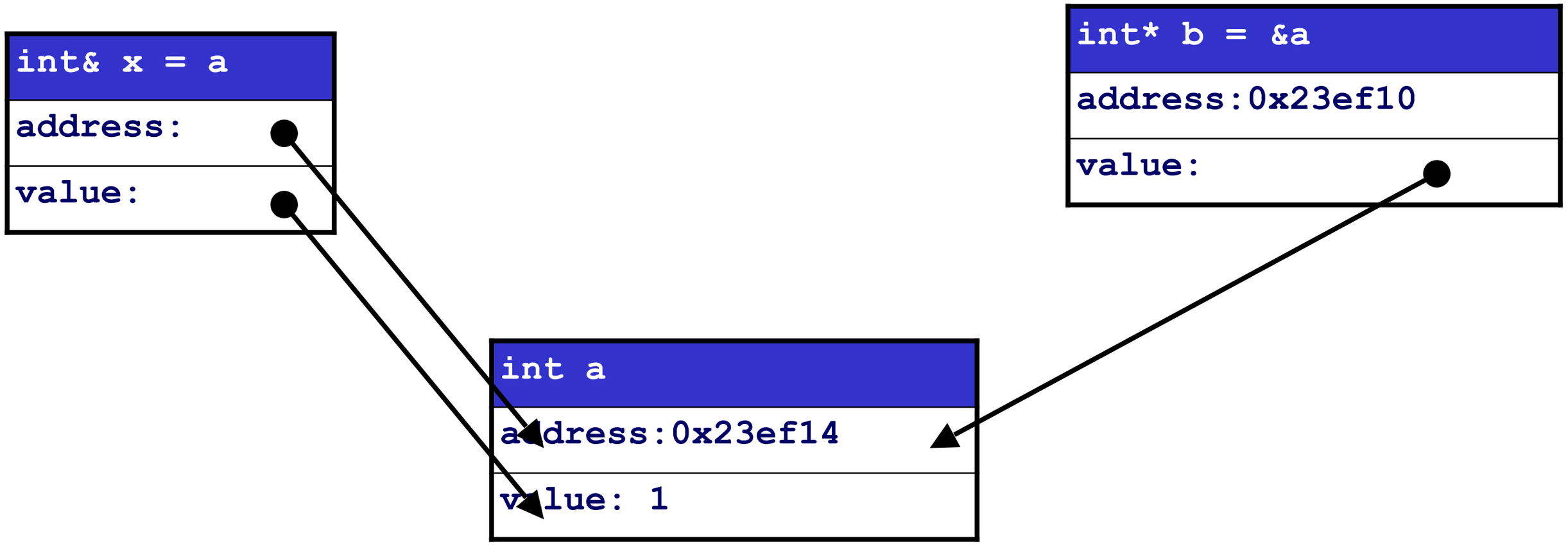
`int a` → 4 bytes==32bit of memory

```
// Pointers.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   int a; // a is a label for a location of memory dtor'ing an int value

   cout << "Insert value of a: ";
   cin >> a; // store value provided by user
             // in location of memory held by a

   int* b; // b is  a pointer to variable of
           // type a

   b  = &a; // value of b is the address of memory
            // location assigned to a

   cout << "value of a: " << a << endl;
   cout << "address of a: " << b << endl;

   return 0;
} // end of main
```

Same location in memory but different values!

```
$ g++ -o Pointers Pointers.cpp
$ ./Pointers
Insert value of a: 3
value of a: 3
address of a: 0x23ef14
$ ./Pointers
Insert value of a: 1.2
value of a: 1
address of a: 0x23ef14
```

Pointers and References



- **x** is a reference to **a**
  - A different name for the same physical location in memory
  - Using **x** or **a** is exactly the same!

- **b** is a pointer to location of memory named **x** or **a**

Pointers and References

```
// refs.cpp
#include <iostream>
using namespace std;

int main() {

   int a = 1;

   int* b; // b is  a pointer to variable of type int

   b  = &a; // value of b is the address of memory location assigned to a

   int& x = a; //

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   cout << "value of x: " << x
        << ", address of x, &x: " << &x
        << endl;

   cout << "value of b: " << b
        << ", address of b, &b: " << &b
        << ", value of *b: " << *b
        << endl;

   return 0;
}
```

```
$ ./refs
value of a: 1, address of a, &a: 0x23ef14
value of x: 1, address of x, &x: 0x23ef14
value of b: 0x23ef14, address of b, &b: 0x23ef10, value of *b: 1
```

Using pointers and references

```
// refs2.cpp
#include <iostream>
using namespace std;

int main() {

   int a = 1;

   int* b = &a;
   *b = 3;

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   int& x = a;
   x = 45;

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   return 0;
}
```

Change value of a with pointer b

Change value of a with reference x

```
$ g++ -o refs2 refs2.cc
$ ./refs2
value of a: 3, address of a, &a: 0x23ef14
value of a: 45, address of a, &a: 0x23ef14
```

Bad and Null Pointers

```
// badptr1.cpp
#include <iostream>
using namespace std;

int main() {

   int* b; // b is  a pointer to varible of type int

   int vect[3] = {1,2,3}; // vector of int

   int* c; // non-initialized pointer
   cout << "c: " << c << ", *c: " << *c <<endl;

   for(int i = 0; i<3; ++i) {
     c = &vect[i];
     cout << "c = &vect[" << i << "]: " << c << ", *c: " << *c << endl;
   }

   // bad pointer
   c++;
   cout << "c: " << c << ", *c: " << *c <<endl;

   // null pointer causing trouble
   c = 0;
   cout << "c: " << c << endl;
   cout << "*c: " << *c <<endl;

   return 0;
}
```

What is the size of an int in memory?

No problem compiling

```
$ g++ -o badptr1 badptr1.cc
$ ./badptr1
c: 0x7c90d592, *c: -1879046974
c = &vect[0]: 0x23eef0, *c: 1
c = &vect[1]: 0x23eef4, *c: 2
c = &vect[2]: 0x23eef8, *c: 3
c: 0x23eefc, *c: 1627945305
c: 0
Segmentation fault (core dumped)
```

Crash at runtime

Constants

- C++ allows to ensure value of a variable does not change within its scope
  - Can be applied to variables, pointers, references, vectors etc.
  - Constants must be ALWAYS initialized since they can't change at a later time!

```
// const1.cpp

int main() {

   const int a = 1;
   a = 2;

   const double x;

   return 0;
}
```

```
$ g++ -o const1 const1.cc
const1.cc: In function `int main()':
const1.cc:6: error: assignment of read-only variable `a'
const1.cc:8: error: uninitialized const `x'
```

Constant Pointer

Read from right to left:
`int * const b`:

`b` is a constant pointer to int

```
// const2.cpp

int main() {

   int a = 1;
   int * const b = &a; // const pointer to int

   *b = 5; // OK. can change value of what b points to

   int c = 3;
   b = &c; // Not OK. assign new value to c

   return 0;
}
```

```
$ g++ -o const2 const2.cc
const2.cc: In function `int main()':
const2.cc:11: error: assignment of read-only variable `b'
```

Pointer to Constant

`a` is not a constant!
But we can treat it as such when pointing to it

```
// const3.cpp

int main() {

   int a = 1;
   const int * b = &a; // pointer to const int

   int c = 3;
   b = &c; // assign new value to c ... OK!

   *b = 5; // assign new value to what c point to ... NOT OK!

   return 0;
}
```

Read from right to left:
`const int * b`:

`b` is a pointer to constant int

```
$ g++ -o const3 const3.cc
const3.cc: In function `int main()':
const3.cc:11: error: assignment of read-only location
```

NB: the error is different!

Constant Pointer to Constant Object

- Most restrictive access to another variable
  - Specially when used in function interface
- Can not change neither the pointer nor what it points to!

```
// const4.cpp

int main() {

   float a = 1;
   const float * const b = &a; // const pointer to const float

   *b = 5; // Not OK. can't change value of what b points to

   float c = 3;
   b = &c; // Not OK. can't change what b points to!

   return 0;
}
```

```
$ g++ -o const4 const4.cc
const4.cc: In function `int main()':
const4.cc:8: error: assignment of read-only location
const4.cc:11: error: assignment of read-only variable
```

Bad Use of Pointers

```
int vect[3] = {1,2,3};
int v2[3];
int v3[] = { 1, 2, 3, 4, 5, 6, 7 };
```

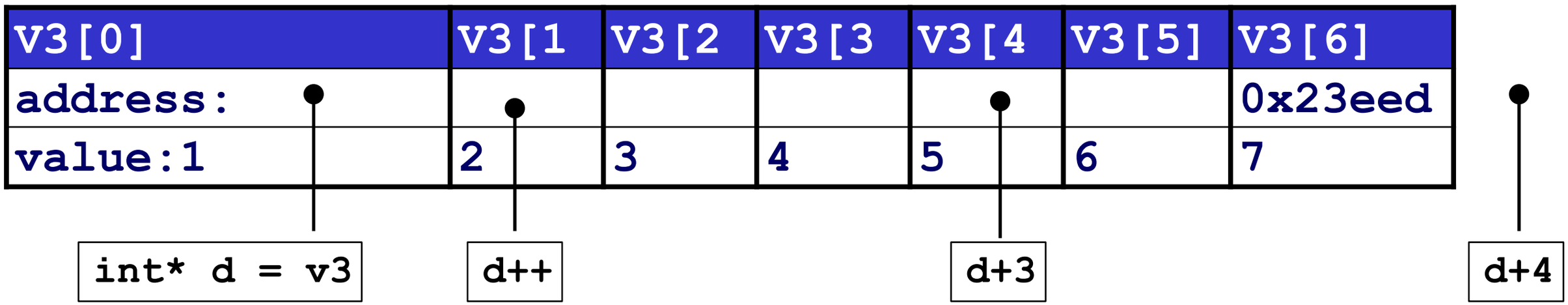
```
$ g++ -o array array.cc
$ ./array
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecc, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eeec, *e:
2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

V3[0]	V3[1	V3[2	V3[3	V3[4	V3[5]	V3[6]
address:						0x23eed
value:1	2	3	4	5	6	7

How many bytes in memory between v3[6] and v2[0] ?

V2[0]	V2[1]	V2[2]		vect[0]	vect[1]	Vect[2]
0x23eee	0x23eee	0x23eee	0x23eee	0x23eef	0x23eef	0x23eef
-1	2	3		1	2	3

Pointer Arithmetic




```
// ptr.cc
#include <iostream>
using namespace std;

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7
   int* d = v3;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d++;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d = d+3;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d = d+4;
   cout << "d = " << d << ", *d: " << *d <<endl;

   return 0;
}
```

```
$ g++ -o ptr ptr.cc
$ ./ptr
d = 0x23eef0, *d: 1
d = 0x23eef4, *d: 2
d = 0x23ef00, *d: 5
d = 0x23ef10, *d: 1628803505
```

+ and – operators with Pointers

```
// ptr2.cc
#include <iostream>
using namespace std;

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;
   int*c = &v3[4];
   cout << "d = " << d << ", *d: " << *d <<endl;
   cout << "c = " << c << ", *c: " << *c <<endl;

   //int* e = c + d;  // not allowed

   cout << "c-d: " << c - d << endl;
   cout << "d-c: " << d - c << endl;

   //int* e = c-d; // wrong!

   int f = c - d;
   float g = c - d;

   cout << "f: " << f << " g: " << g << endl;

   int *  h = &v3[6] + (d-c);
   cout << "int *  h = &v3[6] + (d-c): " << h << " *h: " << *h << endl;

   return 0;
}
```

Arguments of Functions

- Arguments of functions can be passed in two different ways

– By value
  - x is a local variable in f1()

– Pointer or reference
  - x is reference to argument used by caller

```
// funcarg1.cc
#include <iostream>

using namespace std;

void emptyLine() {
  cout
  << "\n--------------------------------\n"
       << endl;
}

void f1(double x) {
  cout << "f1: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f1: change value of x in f1(). x =
"
       << x << endl;
}

void f2(double& x) {
  cout << "f2: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x =
"
       << x << endl;
}
```

Pointers and References in Functions

```
int main() {

   double a = 1.; // define a

   emptyLine();
   cout << "main: before calling f1, a = " << a<< endl;
   f1(a);  // void function
   cout << "main: after calling f1, a = " << a<< endl;

   emptyLine();
   cout << "main: before calling f2, a = " << a<< endl;
   f2(a);  // void function
   cout << "main: after calling f2, a = " << a<< endl;

   return 0;
}
```

f1 has no effect on variables in main

Because a is passed by value

x is a copy of a

f2 modifies the value of the variable in the main!

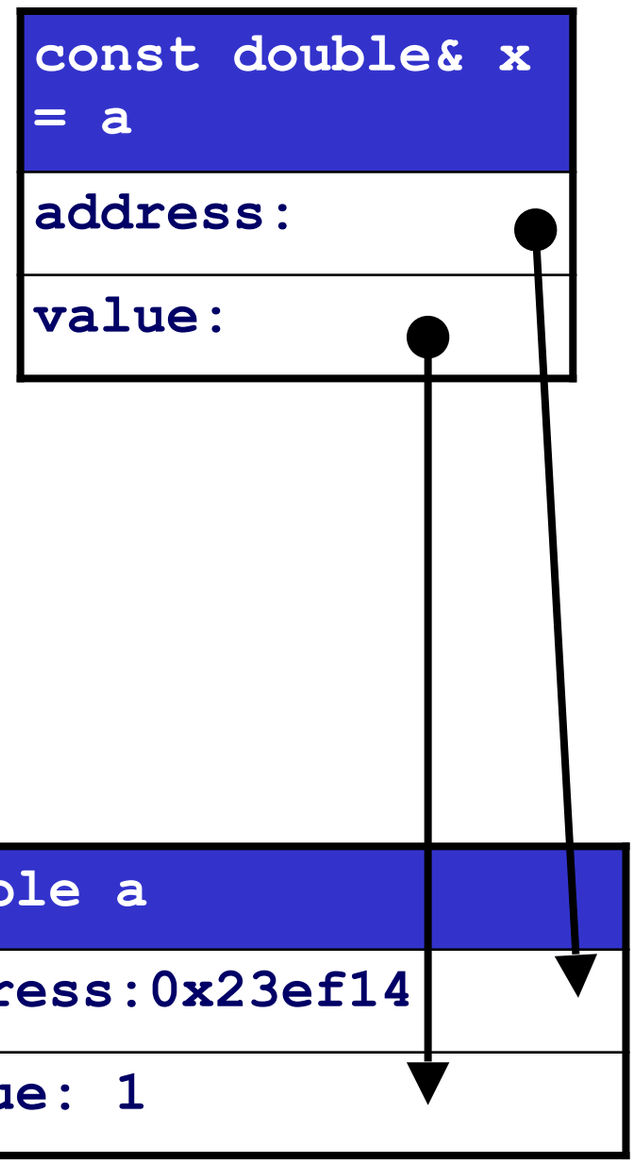
Because a is passed by reference

double& x = a;

```
$ ./funcarg1
-----------------------------------
main: before calling f1, a = 1
f1: input value of x = 1
f1: change value of x in f1(). x = 1.234
main: after calling f1, a = 1
-----------------------------------

main: before calling f2, a = 1
f2: input value of x = 1
f2: change value of x in f2(). x = 1.234
main: after calling f2, a = 1.234
```

Constant Pointers and References in Functions

```
// funcarg2.cc
#include <iostream>

using namespace std;

void f2(const double& x) {
  cout << "f2: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x = "
        << x << endl;
}

int main() {

   double a = 1.;
   f2(a);

   return 0;
}
```




```
$ g++ -o funcarg2  funcarg2.cc
funcarg2.cc: In function `void f2(const double&)':
funcarg2.cc:9: error: assignment of read-only reference `x'
```

Pointers, References and Passing by Value in Functions

```
// mean.cc
#include <iostream>
using namespace std;

void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}

int main() {

   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average;

   computeMean( pressure, 7, average );

   cout << "average pressure: "
        << average << endl;
   return 0;
}
```

```
$ g++ -o mean mean.cc
$ ./mean
data: 0x23eed0, *data: 1.2
data: 0x23eed8, *data: 0.9
data: 0x23eee0, *data: 1.34
data: 0x23eee8, *data: 1.67
data: 0x23eef0, *data: 0.87
data: 0x23eef8, *data: 1.04
data: 0x23ef00, *data: 0.76
average pressure: 1.11143
```

Closer Look at **computeMean()**

```
void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}
```

- Input data passed as constant pointer
  - Good: can't cause trouble to caller! Integrity of data guaranteed
  - Bad: No idea how many data points we have!

- Number of data pointer passed by value
  - Simple int. No gain in passing by reference
  - Bad: separate variable from array of data. Exposed to user error

- Very bad: void function with no return type
  - Good: appropriate name. computeMean() suggests an action not a type

New implementation with Return Type

```
double mean(const double* data, int nData) {
  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean
}
```

- Make function return the computed mean
- New name to make it explicit function returns something
  - Not a rule, but simple courtesy to users of your code
- No need for variables passed by reference to be modified in the function
- Still exposed to user error…

Possible Problems with use of Pointers

```
// mean2.cc
#include <iostream>
using namespace std;

double mean(const double* data, int nData) {

  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean;
}

int main() {
   double pressure[] = { 1.2, 0.6, 1.8 }; // only 3 elements
   double average = mean(pressure, 4); // mistake!
   cout << "average pressure: " << average << endl;

   return 0;
}
```

```
$ g++ -o mean2 mean2.cc
$ ./mean2
data: 0x23eef0, *data: 1.2
data: 0x23eef8, *data: 0.6
data: 0x23ef00, *data: 1.8
data: 0x23ef08, *data: 8.48798e-314
average pressure: 0.9
```

Simple luck!
Additional value not changing the average!

No protection against possible errors!

What about computing other quantities?

- What if we wanted to compute also the standard deviation of our data points?

```
void computeMean(const double* data, int nData, double& mean, double& stdDev) {
 // two variables passed by reference to void function
 // not great. But not harmful.
}


double meanWithStdDev(const double* data, int nData, double& stdDev) {
  // error passed by reference to mean function! ugly!! anti-intuitive
}

double mean(const double* data, int nData) {
  // one method to compute only average
}

double stdDev(const double* data, int nData) {
  // one method to compute standard deviation
  // use mean() to compute average needed by std deviation
}
```

What if we had a new C++ type?

- Imagine we had a new C++ type called **`Result`** including data about both mean and standard deviation

- We could then simply do the following

```
Result mean(const double* data, int nData) {
  Result result;
  // do your calculation
  return result;
}
```

- *This is exactly the idea of classes in C++!*

Classes in C++

- A class is a set of data and functions that define the characteristics and behavior of an object
  - Characteristics also known as attributes
  - Behavior is what an object can do and is referred to also as its interface

Interface
or
Member Functions

Data members or attributes

```
class Result {
  public:

    // constructors
    Result() { }
    Result(const double& mean, const double& stdDev) {
      mean_ = mean;
      stdDev_ = stdDev;
    }

    // accessors
    double getMean() { return mean_; };
    double getStdDev() { return stdDev_; };

  private:
    double mean_;
    double stdDev_;
};
```

Don’t’s forget ; at the end of definition!

Using class **Result**

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

class Result {
  public:

    // constructors
    Result() { };
    Result(const double& mean, const double& stdDev) {
      mean_ = mean;
      stdDev_ = stdDev;
    }

    // accessors
    double getMean() { return mean_; };
    double getStdDev() { return stdDev_; };

  private:
    double mean_;
    double stdDev_;
};
```

```
int main() {

  Result r1;
  cout << "r1, mean: " << r1.getMean()
       << ", stdDev: " << r1.getStdDev()
       << endl;

  Result r2(1.1,0.234);
  cout << "r2, mean: " << r2.getMean()
       << ", stdDev: " << r2.getStdDev()
       << endl;

  return 0;
}
```

```
$ g++ -o results2 result2.cc
$ ./results2
r1, mean: NaN, stdDev: 8.48798e-314
r2, mean: 1.1, stdDev: 0.234
```

`r1` is ill-defined. Why?

What is wrong with Result::Result() ?

C++ Data Types

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

- Size is architecture dependent!
  - Difference between 32-bit and 64-bit machines
  - Above table refers to typical 32-bit architecture
- **`int`** is usually has size of 'one word' on a given architecture

- Four integer types: **`char`**, **`short`**, **`int`**, and **`long`**
  - Each type is at least as large as previous one
    - **`size(char) <= size(short) <= size(int) <= size(long)`**

- Long **`int == int`**; similarly **`short int == short`**

Size of Objects/Types in C++

```
// cpptypes.cc
#include <iostream>
using namespace std;

int main() {
  char*       aChar = "c"; // char
  bool        aBool = true; // boolean
  short       aShort = 33; // short
  long        aLong  = 123421; // long
  int         anInt = 27; // integer
  float       aFloat = 1.043; // single precision
  double      aDbl = 1.243e-234; // double precision
  long double aLD = 0.432e245; // double precision

  cout << "char* aChar = " << aChar << "\tsizeof(" << "*char" << "): " << sizeof(*aChar) << endl;
  cout << "bool aBool = " << aBool << "\tsizeof(" << "bool" << "): " << sizeof(aBool) << endl;
  cout << "short aShort = " << aShort << "\tsizeof(" << "short" << "): " << sizeof(aShort) << endl;
  cout << "long aLong = " << aLong << "\tsizeof(" << "long" << "): " << sizeof(aLong) << endl;
  cout << "int aInt = " << anInt << "\tsizeof(" << "int" << "): " << sizeof(anInt) << endl;
  cout << "float aFloat = " << aFloat << "\tsizeof(" << "float" << "): " << sizeof(aFloat) << endl;
  cout << "double aDbl = " << aDbl << "\tsizeof(" << "double" << "): " << sizeof(aDbl) << endl;
  cout << "long double aLD = " << aLD << "\tsizeof(" << "long double" << "): " << sizeof(aLD) << endl;

  return 0;
}
```

```
$ g++ -o cpptypes cpptypes.cc
$ ./cpptypes
char* aChar = c                    sizeof(*char): 1
bool aBool = 1                     sizeof(bool): 1
short aShort = 33                  sizeof(short): 2
long aLong = 123421                sizeof(long): 4
int aInt = 27                      sizeof(int): 4
float aFloat = 1.043               sizeof(float): 4
double aDbl = 1.243e-234           sizeof(double): 8
long double aLD = 4.32e+244        sizeof(long double): 12
```

Topics

- Classes
  - data members and member functions
- Constructors
  - Special member functions
- private and public members
- Helper and utility methods
  - setters
  - getters
  - accessors

Classes in C++

- A class is a set of data and functions that define the characteristics and behavior of an object
  - Characteristics also known as attributes
  - Behavior is what an object can do and is referred to also as its interface

Interface
or
Member Functions

Data members or attributes

```
class Result {
 public:

  // constructors
  Result() { }
  Result(const double& mean, const double& stdDev) {
    mean_ = mean;
    stdDev_ = stdDev;
  }

  // accessors
  double getMean() { return mean_; };
  double getStdDev() { return stdDev_; };

 private:
  double mean_;
  double stdDev_;
};
```

Don’t’s forget ; at the end of definition!

Data Members (Attributes)

```
class Datum {
    double value_;
    double error_;
};
```

- Data defined in the scope of a class are called data members of that class
- Data members are defined in the class and can be used by all member functions
- Contain the actual data that characterise the content of the class
- Can be public or private
  - public data members are generally bad and symptom of bad design
  - More on this topic later in the course

Interface: Member Functions

- Member functions are methods defined inside the scope of a class
  - Have access to all data members

name_ is a datamember

No declaration of name_ in member functions!

name is a local variable only within setName()

```
// Student
#include <iostream>
#include <string>

class Student {
  using namespace std;
  public:
   // default constructor
   Student() { name_ = ""; }

   // another constructor
   Student(const string& name) { name_ = name; }

   // getter method: access to info from the class
   string name() { return name_; }

   // setter: set attribute of object
   void setName(const string& name) { name_ = name; }

   // utility method
   void print() {
     cout << "My name is: " << name_ << endl;
   }

  private:
   string name_; // data member
};
```

Arguments of Member Functions

- All C++ rules discussed so far hold
- You can pass variables by value, pointer, or reference
- You can use the constant qualifier to protect input data and restrict the capabilities of the methods
  - This has implications on declaration of methods using constants
  - We will discuss constant methods and data members next week
- **Member functions can return any type**
  - Exceptions! Constructors and Destructor
    - Have no return type
    - More on this later

Access specifiers **public** and **private**

- Public functions and data members are available to anyone
- Private members and methods are available ONLY to other member functions

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Datum {
 6   public:
 7     Datum() { }
 8     Datum(double val, double error) {
 9       value_ = val;
10       error_  = error;
11     }
12
13   double value() { return value_; }
14   double error() { return error_; }
15
16   void setValue(double value) { value_ = value; }
17   void setError(double error) { error_ = error; }
18
19   double value_; // public data member!!!
20
21   private:
22     double error_; // private data member
23 };
```

Access elements of an object through member selection operator "."

```
25 int main() {
26
27   Datum d1(1.1223,0.23);
28
29   cout << "d1.value(): " << d1.value()^
30        << " d1.error(): " << d1.error()
31        << endl;
32
33
34   cout << "d1.value_: " << d1.value_
35        << " d1.error_: " << d1.error_
36        << endl;
37
38   return 0;
39 }
```

Accessing private members is a compilation error!

```
$ g++ -o class1 class1.cc
class1.cc: In function `int main()':
class1.cc:22: error: `double Datum::error_' is private
class1.cc:35: error: within this context
```

private members

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

class Datum {
  public:
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value)
      { value_ = value; }
    void setError(double error)
      { error_ = error; }

    void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_; // private data member!!!
    double error_; // private data member
};
```

```
int main() {

  Datum d1(1.1223,0.23);
  // setter with no return value
  d1.setValue( 8.563 );

  // getter to access private data
  double x = d1.value();

  cout << "d1.value(): " << d1.value()
       << " d1.error(): " << d1.error()
       << endl;

  d1.print();

  return 0;
}
```

```
$ g++ -o class2 class2.cc
$ ./class2
d1.value(): 8.563 d1.error(): 0.23
datum: 8.563 +/- 0.23
```

private methods

- Can be used only inside other methods but not from outside

```
 1 // class3.cc
 2 #include <iostream>
 3 using namespace std;
 4
 5 class Datum {
 6   public:
 7     Datum() { reset(); } // reset data members
 8
 9     double value() { return value_; }
10     double error() { return error_; }
11
12     void setValue(double value) { value_ = value; }
13     void setError(double error) { error_ = error; }
14
15     void print() {
16       cout << "datum: " << value_ << " +/- "
               << error_ << endl;
17     }
18
19   private:
20     void reset() {
21       value_ = 0.0;
22       error_ = 0.0;
23     }
24
25     double value_;
26     double error_;
27 };
```

```
int main() {

  Datum d1;
  d1.setValue( 8.563 );
  d1.print();

  return 0;
}
```

```
$ g++ -o class3 class3.cc
$ ./class3
datum: 8.563 +/- 0
```

```
30 int main() {
31
32   Datum d1;
33   d1.setValue( 8.563 );
34   d1.print();
35   d1.reset();
36
37   return 0;
38 }
```

```
$ g++ -o class4 class4.cc
class4.cc: In function `int main()':
class4.cc:20: error: `void Datum::reset()' is private
class4.cc:35: error: within this context
```

Hiding Implementation from Users/Clients

- How to decide what to make public or private?

- Principle of Least Privilege
  - elements of a class, data or functions, must be private unless proven to be needed as public!

- Users should rely solely on the interface of a class

- They should never use the internal details of the class

- ***That's why having public data members is a VERY bad idea!***
  - name and characteristics of data members can change
  - Functionalities and methods remain the same
  - You must be able to change internal structure of the class without affecting the clients!

Bad Example of Public Data Members

```
int main() {

  Datum d1(1.1223,0.23);
  double x = d1.value();
  double y = d1.error_;
  cout << "x: " << x << "\t y: " << y << endl;

  return 0;
}
```

```
class Datum {
  public:
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
      cout << "datum: " << value_ << " +/- " << error_ << endl;
    }

  //private:        // all data are public!
    double value_;
    double error_;
};
```

application uses directly the data member!

```
$ g++ -o class6 class6.cc
$ ./class6
x: 1.1223        y: 0.23
```

Bad Example of Public Data Members

Same Application as before

Change the names of data members

No change of functionality so no one should be affected!

```
class Datum {
  public:
    Datum(double val, double error) {
      val_ = val;
      err_  = error;
    }

    double value() { return val_; }
    double error() { return err_; }

    void setValue(double value) { val_ = value; }
    void setError(double error) { err_ = error; }

    void print() {
      cout << "datum: " << val_ << " +/- " << err_ << endl;
    }

  //private:        // alla data are public!
    double val_;  // value_ → val_
    double err_;  // error_ → err_
};
```

```
28 int main() {
29
30   Datum d1(1.1223,0.23);
31   double x = d1.value();
32   double y = d1.error_;
33
34   cout << "x: " << x << "\t y: " << y << endl;
35
36   return 0;
37 }
```

Our application is now broken!

But Datum has not changed its behavior!

Bad programming!

Only use the interface of an object not its internal data!

Private data members prevent this

```
$ g++ -o class7 class7.cc
class7.cc: In function `int main()':
class7.cc:32: error: 'class Datum' has no member named `error_'
```

Constructors

```
class Datum {
  public:
    Datum() { }
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

  private:
    double value_; // public data member!!!
    double error_; // private data member
};
```

- Special member functions
  - Required by C++ to create a new object
  - MUST have the same name of the class
  - Used to initialize data members of an instance of the class
  - Can accept any number of arguments
    - Same rules as any other C++ function applies

- Constructors have no return type!

- There can be several constructors for a class
  - Different ways to declare and an object of a given type

Different Types of Constructors

- Default constructor
  - Has no argument
  - On most machines the default values for data members are assigned
- Copy Constructor
  - Make a new object from an existing one
- Regular constructor
  - Provide sufficient arguments to initialize data members

```
class Datum {
  public:
    Datum() { }


    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }


    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }


  private:
    double value_;
    double error_;
};
```

Using Constructors

```
// class5.cc
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum() { }

    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }

    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }

    void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_;
    double error_;
};
```

```
int main() {

  Datum d1;
  d1.print();

  Datum d2(0.23,0.212);
  d2.print();

  Datum d3( d2 );
  d3.print();


  return 0;
}
```

```
$ g++ -o class5 class5.cc
$ ./class5
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Default Constructors on Different Architectures

```
$ uname -a
CYGWIN_NT-5.1 lajolla 1.5.18(0.132/4/2) 2005-07-02 20:30 i686 unknown
unknown Cygwin
$ gcc -v
Reading specs from /usr/lib/gcc/i686-pc-cygwin/3.4.4/specs
…
gcc version 3.4.4 (cygming special) (gdc 0.12, using dmd 0.125)

$ g++ -o class5 class5.cc
$ ./class5
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Windows XP with CygWin

```
$ uname -a
Linux pccms02.roma1.infn.it 2.6.14-1.1656_FC4smp #1 SMP Thu Jan 5 22:24:06 EST
 2006 i686 i686 i386 GNU/Linux
$ gcc -v
Using built-in specs.
Target: i386-redhat-linux
…
gcc version 4.0.2 20051125 (Red Hat 4.0.2-8)
$ g++ -o class5 class5.cc
$ ./class5
datum: 6.3275e-308 +/- 4.85825e-270
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Fedora Core4

Default Assignment

```
// ctor.cc
#include <iostream>
using std::cout;
using std::endl;

class Datum {
  public:
    Datum(double x) { x_ = x; }
    double value()  { return x_; }
    void setValue(double x) { x_ = x; }
    void print() {
      cout << "x: " << x_ << endl;
    }

  private:
    double x_;
};
```

```
int main() {

  Datum d1(1.2);
  d1.print();

  // no default ctor. compiler error if uncommented
  //Datum d2;
  //d2.print();

  Datum d3 = d1; // default assigment by compiler
  d3.print();
  cout << "&d1: " << &d1
       << "\t &d3: " << &d3 << endl;
  return 0;
}
```

d3.x_ = d1.x_

done by compiler

```
$ g++ -o ctor ctor.cc
$ ./ctor
x: 1.2
x: 1.2
&d1: 0x23ef10    &d3: 0x23ef08
```

Question

- Can a constructor be private?
  - Is it allowed by the compiler?
  - How to instantiate an object with no public constructor?

- *Find a working example of a very simple class for next week*

Accessors and Helper/Utility Methods

- Methods that allow read access to data members
- Can also provide functionalities commonly needed by users to elaborate information from the class
  - for example formatted printing of data
- Usually they do not modify the objects, i.e. do not change the value of its attributes

```
class Student {
  public:

    // getter method: access to data members
    string name() { return name_; }

    // utility method
    void print() {
      cout << "My name is: " << name_ << endl;
    }

  private:
    string name_; // data member
};
```

```
class Datum {
  public:

    double value() { return value_; }
    double error() { return error_; }

    void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_; // public data member!!!
    double error_; // private data member
};
```

Getter Methods

- getters are helpers methods with explicit names returning individual data members
  - Do not modify the data members simply return them
  - Good practice: call these methods as getFoo() or foo() for member foo_
- Return value of a getter method should be that of the data member

```
class Datum {
  public:
    Datum(double val, double error) {
      val_ = val;
      err_  = error;
    }

    double value() { return val_; }
    double error() { return err_; }

    void setValue(double value) { val_ = value; }
    void setError(double error) { err_ = error; }

    void print() {
      cout << "datum: " << val_ << " +/- " << err_
           << endl;
    }

  private:
    double val_;
    double err_;
};
```

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

class Student {
  public:
   // default constructor
   Student() { name_ = ""; }

   // another constructor
   Student(const string& name) { name_ = name; }

   // getter method: access to info from the class
   string name() { return name_; }

   // setter: set attribute of object
   void setName(const string& name) { name_ = name; }

   // utility method
   void print() {
     cout << "My name is: " << name_ << endl;
   }

  private:
   string name_; // data member
};
```

Setter Methods

- Setters are member functions that modify attributes of an object after it is created
  - Typically defined as void
  - Could return other values for error handling purposes
  - Very useful to assign correct attributes to an object in algorithms
  - As usual abusing setter methods can cause unexpected problems

```
// class8.cc
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum(double val, double error) {
      value_ = val;
      error_  = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
      cout << "datum: " << value_ << " +/- "
           << error_ << endl;
    }

  private:
    double value_;
    double error_;
};
```

```
int main() {

  Datum d1(23.4,7.5);
  d1.print();

  d1.setValue( 8.563 );
  d1.setError( 0.45 );
  d1.print();

  return 0;
}
```

```
$ g++ -o class8 class8.cc
$ ./class8
datum: 23.4 +/- 7.5
datum: 8.563 +/- 0.45
```

Pointers and References to Objects

```
// app2.cpp
#include <iostream>
using std::cout;  // use using only for specific
classes
using std::endl;  // not for entire namespace

class Counter {
  public:
    Counter() { count_ = 0; x_=0.0; };
    int value()  { return count_; }
    void reset() { count_ = 0; x_=0.0; }
    void increment() { count_++; }
    void increment(int step)
       { count_ = count_+step; }
    void print() {
      cout << "---- Counter::print() ----" << endl;
      cout << "my count_: " << count_ << endl;
      // this is special pointer
      cout << "my address: " << this << endl;
      cout << "&x_ : " << &x_ << "  sizeof(x_): "
           << sizeof(x_) << endl;
      cout << "&count_ : " << &count_
      << "  sizeof(count_): "
      << sizeof(count_) << endl;
      cout << "---- Counter::print()----" << endl;
    }

  private:
    int count_;
    double x_; // dummy variable
};
```

```
void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  // ptr is a pointer to a Counter Object
  Counter* ptr = &counter;
  cout << "ptr = &counter: " << &counter << endl;

  // use . to access member of objects
  cout << "counter.value(): " << counter.value() << endl;

  // use -> with pointer to objects
  cout << "ptr->value(): " << ptr->value() << endl;

  printCounter( counter );
  printByPtr( ptr );

  ptr->print();

  cout << "sizeof(ptr): " << sizeof(ptr) << "\t"
       << "sizeof(counter): " << sizeof(counter)
       << endl;

return 0;
}
```

-> instead of . When using pointers to objects

Size and Address of Objects

gcc 3.4.4 on cygwin

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0x22ccd0
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0x22ccd0
&count_ : 0x22ccd0  sizeof(count_): 4
&x_ : 0x22ccd8  sizeof(x_): 8
---- Counter::print() : end ----
&i: 0x22ccc8
sizeof(ptr): 4  sizeof(counter): 16
sizeof(int): 4  sizeof(double): 8
```

gcc 4.1.1 on fedora core 6

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0xbf841e20
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0xbf841e20
&count_ : 0xbf841e20  sizeof(count_): 4
&x_ : 0xbf841e24  sizeof(x_): 8
---- Counter::print() : end ----
&i: 0xbf841e1c
sizeof(ptr): 4  sizeof(counter): 12
sizeof(int): 4  sizeof(double): 8
```

- Different size of objects on different platform!
  - Different configuration of compiler
  - Optimization for access to memory

- Address of object is address of first data member in the object

Classes and Applications

- So far we have always included the definition of classes together with the main application in one file

- The advantage is that we have only one file to modify

- Disadvantage are many
  - There is always ONE file to modify no matter what kind of modification you want to make
  - This file becomes VERY long after a very short time
  - Hard to maintain everything in only one place
  - We compile everything even after very simple changes

Example of Typical Application So Far

```
// app3.cpp
#include <iostream>
using std::cout;
using std::endl;

#include "Counter.h"

Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  Counter* ptr = &counter;

  cout << "counter.value(): " << counter.value()
       << endl;
  cout << "ptr = &counter: " << &counter << endl;
  cout << "ptr->value(): " << ptr->value() << endl;

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( c2 );

  cout << "sizeof(ptr): " << sizeof(ptr)
       << " sizeof(c2): " << sizeof(c2)
       << endl;

  return 0;

}
```

Separating Classes and Applications

- It's good practice to separate classes from applications

- Create one file with only your application
  - Use #include directive to add all classes needed in your application

  - Keep a separate file for each class

- Compile your classes separately

- Include compiled classes (or libraries) when linking your application

First Attempt at Improving Code Management

```
// Datum1.cc
// include all header files needed
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum() { }

    Datum(double x, double y) {
      value_ = x;
      error_ = y;
    }

    Datum(const Datum& datum) {
      value_ = datum.value_;
      error_ = datum.error_;
    }

    void print() {
      cout << "datum: " << value_
           << " +/- " << error_
           << endl;
    }

  private:
    double value_;
    double error_;
};
```

```
// app1.cpp
#include "Datum1.cc"

int main() {

  Datum d1;
  d1.print();

  Datum d2(0.23,0.212);
  d2.print();

  Datum d3( d2 );
  d3.print();

  return 0;
}
```

```
$ g++ -o app1 app1.cpp
$ ./app1
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Problems with Previous Example

- Although we have two files it is basically if we had just one!

- Datum1.cc includes not only the declaration but also the definition of class Datum
  - Implementation of all methods exposed to user

- When compiling app1.cpp we also compile class Datum every time!
  - We do not need any library because app1.cpp includes all source code!
  - When compiling and linking app1.cpp we also create compiled code for Datum to be used in our application

  - *Remember what #include does!*

Pre-Compiled version of `Datum1.cc`

- Our source file is only a few lines long

```
$ wc -l Datum1.cc
30 Datum1.cc

$ wc -l app1.cpp
16 app1.cpp

$ g++ -E -c Datum1.cc > Datum1.cc-precomoiled

$ wc -l Datum1.cc-precompiled
23740 Datum1.cc-precompiled
```

- The precompiled version is almost 24000 lines!
  - This is all code included in and by iostream

```
$ grep "#include"  /usr/lib/gcc/i686-pc-cygwin//3.4.4/include/c++/
iostream
 *  This is a Standard C++ Library header.  You should @c #include
this header
#include <bits/c++config.h>
#include <ostream>
#include <istream>
```

iostream

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

#pragma GCC system_header

#include <bits/c++config.h>
#include <ostream>
#include <istream>

namespace std
{
  /**
   *  @name Standard Stream Objects
   *
*/
  //@{
  extern istream cin;          ///< Linked to standard input
  extern ostream cout;         ///< Linked to standard output
  extern ostream cerr;         ///< Linked to standard error (unbuffered)
  extern ostream clog;         ///< Linked to standard error (buffered)

#ifdef _GLIBCXX_USE_WCHAR_T
  extern wistream wcin;        ///< Linked to standard input
  extern wostream wcout;       ///< Linked to standard output
  extern wostream wcerr;       ///< Linked to standard error (unbuffered)
  extern wostream wclog;       ///< Linked to standard error (buffered)
#endif
  //@}

  // For construction of filebuffers for cout, cin, cerr, clog et. al.
  static ios_base::Init __ioinit;
} // namespace std

#endif /* _GLIBCXX_IOSTREAM */
```

I have removed all comments from the file to make it fit in this slide

Additional code included by the header files in this file

How do you find **`iostream`** file on your computer?

Separating Interface from Implementation

- Clients of your classes only need to know the interface of your classes

- Remember:
  - Users should only rely on public members of your class
  - Internal data structure must be hidden and not needed in applications

- Compiler needs only the declaration of your classes, its functions and their signature to compile the application
  - Signature of a function is the exact set of arguments passed to a function and it return type

- The compiled class code (definition) is needed only at link time
  - Libraries are needed to link not to compile!

Header and Source Files

- We can separate the declaration of a class from its implementation
  - Declaration tells the compiler about data members and member functions of a class
  - We know how many and what type of arguments a function has by looking at the declaration but we don't know how the function is implemented

- Declaration of a class Counter goes into a file usually called Counter.h or Counter.hh suffix

- Implementation of methods goes into the source file usually called Counter.cc

Counter.h and Counter.cc

```
// Counter.h
// Counter Class: simple counter class.
// Allows simple or step
// increments and also a reset function

// include header files for types
// and classes used in the declaration

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

  private:
    int count_;
};
```

```
// Counter.cc
// include class header files
#include "Counter.h"

// include any additional header files
//  needed in the class
// definition
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
  count_ = 0;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment() {
  count_++;
}

void Counter::increment(int step) {
  count_ = count_+step;
}
```

Scope operator :: is used to tell methods belong to Class Counter

What is included in header files?

- Declaration of the class
  - Public and data members

- All header files for types and classes used in the header
  - data members, arguments or return types of member functions

- Sometimes when we have very simple methods these are directly implemented in the header file

- Methods implemented in the header file are referred to as inline functions
  - For example getter methods are a good candidate to become inline functions

What is included in source file?

- Header file of the class being implemented
  - Compiler needs the prototype (declaration) of the methods

- Implementation of methods declared in the header file
  - Scope operator :: must be used to tell the compiler methods belong to a class

- Header files for all additional types used in the implementation but not needed in the header!
  - Nota bene: header files include in the header file of the class are automatically included in the source file

Compiling Source Files of a Class

```
$ g++ Counter.cc
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../libcygwin.a(libcmain.o)::
undefined reference to `_WinMain@16'
collect2: ld returned 1 exit status
```

WinXP+ cygwin

```
$ g++ Counter.cc
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../crt1.o(.text+0x18):
In function `_start':: undefined reference to `main'
collect2: ld returned 1 exit status
```

Linux

- Do you understand the error?
- What does undefined symbol usually mean?
- Why we did not encounter this error earlier?

- Separating Interface
- Implementation of Classes
- Header and Source Files
- Dynamic Memory Management
- Class Destructors

Reminder about g++

- g++ by default looks for a main function in the file being compiled unless differently instructed

- The main function becomes the program to run when the compiler is finished linking the binary application
  - Compiling: translate user code in high level language into binary code that system can use
  - Linking: put together binary pieces corresponding to methods used in the main function
  - Application: product of the linking process

- Source files of classes do not have any main method

- We need to tell g++ (and other compilers) no linking is needed

Compiling without Linking

- g++ has a `-c` option that allows to specify only compilation is needed

- User code is translated into binary but no attempt to look for main method and creating an application

```
$ ls -l Counter.*
-rw-r--r--  1 rahatlou users 449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users 349 May 15 00:55 Counter.h

$ g++ -c Counter.cc

$ ls -l Counter.*
-rw-r--r--  1 rahatlou users  449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users  349 May 15 00:55 Counter.h
-rw-r--r--  1 rahatlou users 1884 May 15 01:23 Counter.o
```

By default g++ creates a .o (object file) for the .cc file

Using Header Files in Applications

```
// app2.cpp
#include <iostream>
using namespace std;

#include "Counter.h"


Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: "
       << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: "
       << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  Counter* ptr = &counter;

  cout << "counter.value(): "
       <<counter.value() <<
endl;
  cout << "ptr = &counter: "
       << &counter << endl;
  cout << "ptr->value(): "
       << ptr->value() << endl;


  Counter c2 = makeCounter();
  c2.increment();

  printCounter( c2 );

  return 0;
}
```

```
$ g++ -o app2 app2.cpp
/tmp/ccJuugJc.o:app2.cpp:(.text+0x10d): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x124): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x16e): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1dc): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1ef): undefined reference to `Counter::increment(int)'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x200): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x272): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x2b7): undefined reference to `Counter::increment()'
collect2: ld returned 1 exit status
```

Providing compiled Class Code at Link Time

- Including the header file is not sufficient!
  - It tells the compiler only about arguments and return type
  - But it does not tell him what to execute
  - Compiler doesn't have the binary code to use to create the application!

- We must use the compiled object file at link time
  - g++ is told to make an application called app2 from source code in app2.cpp and using also the binary file Counter.o to find any symbol needed in app2.cpp

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
counter.value(): 7
ptr = &counter: 0x23ef10
ptr->value(): 7
counter value: 1
```

Problem: Multiple Inclusion of Header Files!

- What if we include the same header file several times?
  - This can happen in many ways

- Some pretty common ways are
  - **`App.cpp`** includes both **`Foo.h`** and **`Bar.h`**
  - **`Foo.h`** is included in **`Bar.h`** and **`Bar.cc`**

```
// Bar.h

#include "Foo.h"

class Bar {

  // class goes here
  Bar(const Foo& afoo, double x);


}`
```

```
// App.cpp

#include "Foo.h"
#include "Bar.h"

int main() {

  // program goes here
  Foo f1;
  Bar b1(f1, 0.3);

  return 0;
}
```

Example of Multiple Inclusion

```
// app3.cpp
#include <iostream>
using namespace std;
#include "Counter.h"

Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}

#include "Counter.h"
int main() {
  Counter counter;
  counter.increment(7);

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( counter );
  printCounter( c2 );

  return 0;
}
```

Line 19

```
// Counter.h
// Counter Class: simple counter class. Allows simp
// increments and also a reset function

// include header files for types and classes
// used in the declaration

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

  private:
    int count_;
}
```

Line 8

```
$ g++ -o app3 app3.cpp Counter.o
In file included from app3.cpp:19:
Counter.h:8: error: redefinition of `class Counter'
Counter.h:8: error: previous definition of `class Counter'
```

#define, #ifndef and #endif directives

- Problem of multiple inclusion can be solved at pre-compiler level

1: if Datum_h is not defined follow the instruction until #endif

2: define a new variable called Datum_h

3: end of ifndef block

```
#ifndef Datum_h
#define Datum_h
// Datum.h

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value() { return value_; }
    double error() { return error_; }

  private:
    double value_;
    double error_;
};
#endif
```

Example: application using Datum

```
// app4.cpp
#include "Datum.h"
#include <iostream>

void print(Datum& input) {
  using namespace std;
  cout << "input: " << input.value()
       << " +/- " << input.error()
       << endl;
}

#include "Datum.h"

int main() {
  Datum d1(-1.4,0.3);
  print(d1);

  return 0;
}
```

```
$ g++ -c Datum.cc
$ g++ -o app4 app4.cpp Datum.o
$ ./app4
input: -1.4 +/- 0.3
```

Typical Errors

- Forget to use the scope operator :: in .cc files

```
#ifndef FooDatum_h
#define FooDatum_h
// FooDatum.h

class FooDatum {
  public:
    FooDatum();
    FooDatum(double x, double y);
    FooDatum(const FooDatum& datum);
    double value() { return value_; }
    double error() { return error_; }
    double significance();

  private:
    double value_;
    double error_;
};
#endif
```

```
#include "FooDatum.h"

FooDatum::FooDatum() { }

FooDatum::FooDatum(double x, double y) {
  value_ = x;
  error_ = y;
}

FooDatum::FooDatum(const FooDatum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}

double
significance() {
  return value_/error_;
}
```

```
$ g++ -c FooDatum.cc
FooDatum.cc: In function `double significance()':
FooDatum.cc:17: error: `value_' undeclared (first use this function)
FooDatum.cc:17: error: (Each undeclared identifier is reported only once
 for each function it appears in.)
FooDatum.cc:17: error: `error_' undeclared (first use this function)
```

- Functions implemented as global
- error when applying function as a member function to objects
- No error compiling the classes but error when compiling the application

Reminder: Namespace of Classes

- C++ uses namespace as integral part of a class, function, data member

- Any quantity declared within a namespace can be accessed ONLY by using the scope operator :: and by specifying its namespace

- When using a new class, you must look into its header file to find out which namespace it belongs to
  - There are no shortcuts!

- When implementing a class you must specify its namespace
  - Unless you use the using directive

Another Example of Namespace

```
#ifndef CounterNS_h_
#define CounterNS_h_
#include <string>

namespace rome {
  namespace didattica {

    class Counter {
      public:
        Counter(const std::string& name);
        ~Counter();
        int value();
        void reset();
        void increment(int step =1);
        void print();

      private:
        int count_;
        std::string name_;
    }; // class counter

  } // namespace didattica
} //namespace rome
#endif
```

```
#include "CounterNS.h"

int main() {
  rome::didattica::Counter c1("c1");
  c1.print();
  return 0;
}
```

```
// CounterNS.cc
#include "CounterNS.h"

// include any additional heade files needed in the class
// definition
#include <iostream> // needed for input/output
using std::cout;
using std::endl;
using namespace rome::didattica;

Counter::Counter(const std::string& name) {
  count_ = 0;
  name_ = name;
  cout << "Counter::Counter() called for Counter "<< name_
<< endl;
};

Counter::~Counter() {
  cout << "Counter::~Counter() called for Counter "<<
name_ << endl;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment(int step) {
  count_ = count_+step;
}

void Counter::print() {
  cout << "Counter::print(): name: " << name_ << "  value:
" << count_ << endl;
}
```

Class `std::vector<T>`

```
#include <iostream>
#include <vector>
#include "Datum.h"

int main() {


  std::vector<double> vals;
  vals.push_back(1.3);
  vals.push_back(-2.1);

  std::vector<double> errs;
  errs.push_back(0.2);
  errs.push_back(0.3);

  std::vector<Datum> data;
  data.push_back( Datum(1.3, 0.2) );
  data.push_back( Datum(-2.1, 0.3) );

 std:cout << "# dati:: " << data.size() << std::endl;

  // using traditional loop on an array
  int i=0;
  std::cout << "Using [] operator on vector" << std::endl;
  for(i=0; i< data.size(); ++i) {
    std::cout << "i: " << i
                    << "\t data: " << data[i].value() << " +/- " << data[i].error()
                    << std::endl;
  }

  // using vector iterator
  i=0;
  std::cout << "std::vector<T>::iterator " << std::endl;
  for(std::vector<Datum>::iterator d = data.begin(); d != data.end(); d++) {
    //std::cout << "d: " << d << std::endl;
    i++;
    std::cout << "i: " << i
                    << "\t data: " << d->value() << " +/- " << d->error()
                    << std::endl;
  }

    // using vector iterator
  i=0;
  std::cout << "C+11 extension feature " << std::endl;
  for(Datum dit : data) {
    i++;
    std::cout << "i: " << i
                    << "\t data: " << dit.value() << " +/- " << dit.error()
                    << std::endl;
  }

  return 0;

}
```

```
$ g++ -o app.exe vector1.cc Datum.cc
vector1.cc:45:17: warning: range-based for loop is a C++11 extension
      [-Wc++11-extensions]
  for(Datum dit : data) {
                ^
1 warning generated.
$ ./app.exe
# dati:: 2
Using [] operator on vector
i: 0            data: 1.3 +/- 0.2
i: 1            data: -2.1 +/- 0.3
std::vector<T>::iterator
i: 1            data: 1.3 +/- 0.2
i: 2            data: -2.1 +/- 0.3
C+11 extension feature
i: 1            data: 1.3 +/- 0.2
i: 2            data: -2.1 +/- 0.3
```

Interface of `std::vector<T>`

http://www.cplusplus.com/reference/vector/vector/
https://en.cppreference.com/w/cpp/container/vector

Member functions

(constructor)	constructs the `vector` (public member function)
(destructor)	destructs the `vector` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data` (C++11)	direct access to the underlying array (public member function)

Iterators

`begin` **`cbegin`**	returns an iterator to the beginning (public member function)
`end` **`cend`**	returns an iterator to the end (public member function)
`rbegin` **`crbegin`**	returns a reverse iterator to the beginning (public member function)
`rend` **`crend`**	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)

Using `std::vector<T>` in functions

```
#include <vector>
Using std::vector;

Datum average(vector<float>&  val,
vector<float>& err) {
  double mean = 0.;
  double meanErr(0.); // same as = 0.

  // loop over data
    // compute average

  Datum  res(mean, meanErr);
  return res;
}
```

Constructor is called with arguments
Same behavior for **`double`** and **`Datum`**

Object **`res`** is like any other variable **`mean`** or **`meanErr`**
`res` simply returned as output to caller

```
#include <vector>
Using std::vector;

Datum average(vector<float>&  val,
vector<float>& err) {
  double mean = 0.;
  double meanErr(0.); // same as =
0.

  // loop over data
    // compute average

  return Datum(mean, meanErr);
}
```

```
#include <vector>
Using std::vector;

double average(vector<float>&  val) {
  double mean = 0.;
    // loop over data
    // compute average

  return mean;
}
```

Since **`res`** not really needed within function
we can just create it while returning the function output

Dynamic Memory Allocation: **`new`** and **`delete`**

- C++ allows dynamic management memory at run time via two dedicated operators: **`new`** and **`delete`**

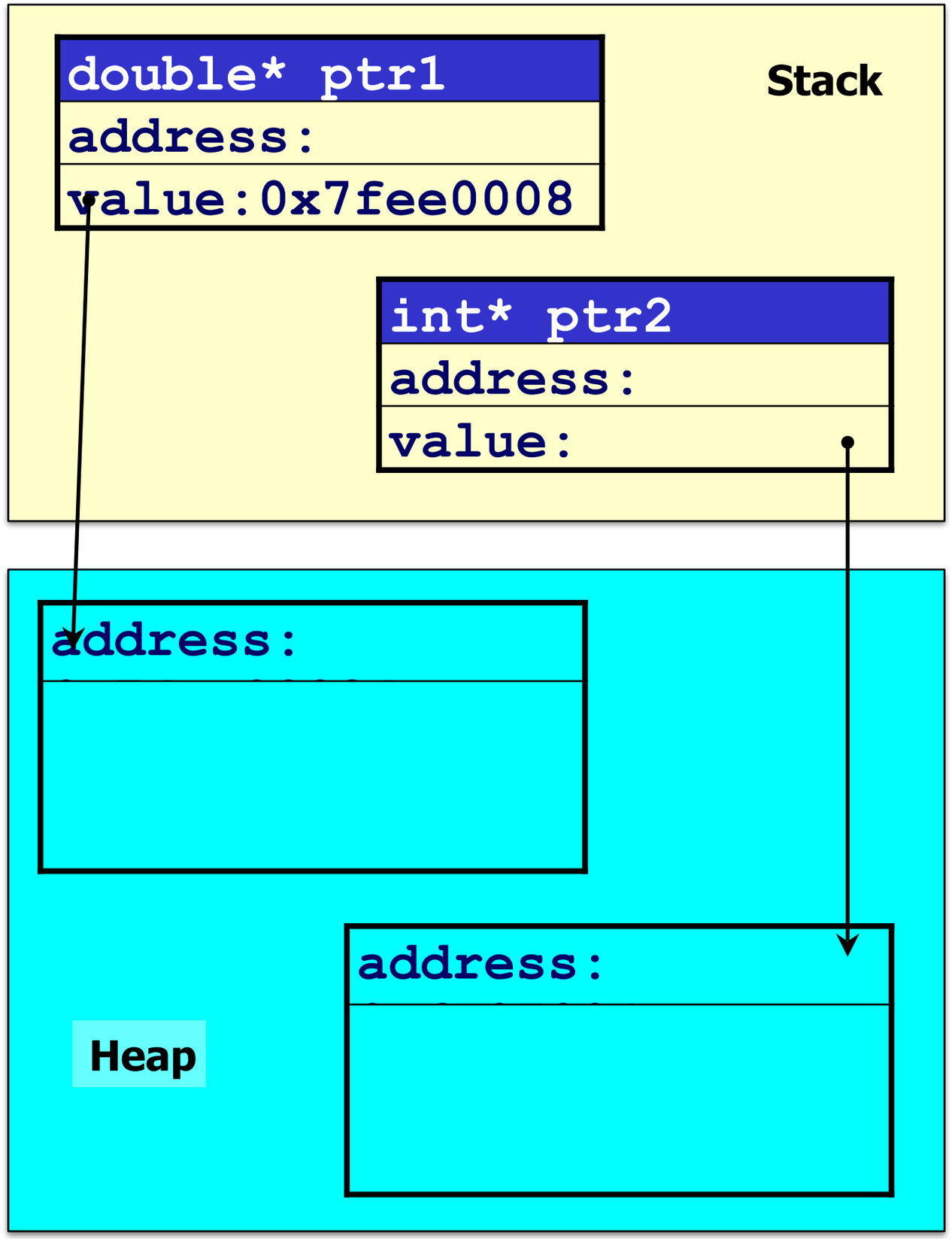
- **`new`**: allocates memory for objects of any built-in or user-defined type
  - The amount of allocated memory depends on the size of the object
  - For user-defined types the size is determined by the data members

- Which memory is used by **`new`**?
  - **`new`** allocated objects in the free store also known as heap
  - This is region of memory assigned to each program at run time
  - Memory allocated by **`new`** is unavailable until we free it and give it back to system via **`delete`** operator

- **`delete`**: de-allocates memory used by new and give it back to system to be re-used

Stack and Heap

```
// app7.cpp
#include <iostream>
using namespace std;

int main() {
    double* ptr1 = new double[100000];
    ptr1[0] = 1.1;

    cout << "ptr1[0]: " << ptr1[0]
         << endl;

    int* ptr2 = new int[1000];
    ptr2[233] = -13423;

    cout << "&ptr1: "<< &ptr1
     << "  sizeof(ptr1): " << sizeof(ptr1)
     << "  ptr1: " << ptr1 << endl;

    cout << "&ptr2: "<< &ptr2
     << "  sizeof(ptr2): " << sizeof(ptr2)
     << " ptr2: " << ptr2 << endl;
    delete[] ptr1;
    delete[] ptr2;
    return 0;
}
```



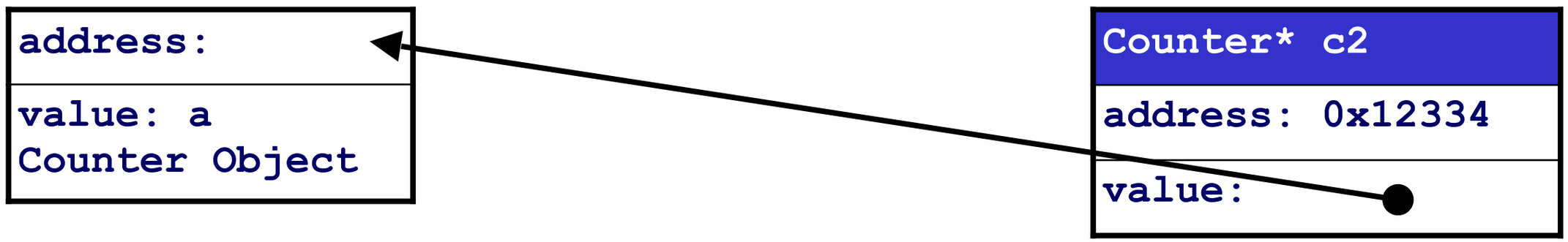

```
$ g++ -Wall -o app7 app7.cpp
$ ./app7
ptr1[0]: 1.1
&ptr1: 0x22cce4  sizeof(ptr1): 4  ptr1: 0x7fee0008
&ptr2: 0x22cce0  sizeof(ptr2): 4
ptr2: 0x6a0700
```

What does **new** do?

Dynamic object in the heap

```
Counter* c2 = new Counter("c2");

delete c2; // de-allocate memory!
```

Automatic variable in the stack



- new allocates an amount of memory given by **`sizeof(Counter)`** somewhere in memory
- returns a pointer to this location
- we assign **`c2`** to be this pointer and access the dynamically allocated memory
- delete de-allocates the region of memory pointed to by **`c2`** and makes this memory available to be re-used by the program

Memory Leak: Killing the System

- Perhaps one of the most common problems in C++ programming
- User allocates memory at run time with new but never releases the memory – forgets to call delete!
- Golden rule: every time you call **`new`** ask yourself
  - Do I really need to use new?
  - where and when **`delete`** is called to free this memory ?
- Even small amount of leak can lead to a crash of the system
  - Leaking 10 kB in a loop over 1M events leads to 1 GB of allocated and un-usable memory!

Simple Example of Memory Leak

```
// app6.cpp
#include <iostream>
using namespace std;

int main() {

  for(int i=0; i<10000; ++i){

    double* ptr = new double[100000];
    ptr[0] = 1.1;

    cout << "i: " << i
         << ", ptr: " << ptr
         << ", ptr[0]: " << ptr[0]
         << endl;

    // delete[] ptr; // ops! memory
leak!
  }
  return 0;
}
```

```
$ g++ -o leak1 leak1.cpp
$ ./leak1
i: 0, ptr: 0x4a0280, ptr[0]: 1.1
i: 1, ptr: 0x563bf8, ptr[0]: 1.1
…
i: 1381, ptr: 0x4247e178, ptr[0]: 1.1
i: 1382, ptr: 0x42541680, ptr[0]: 1.1
Abort (core dumped)
```

- At each iteration **`ptr`** is a pointer to a new (and large) array of 100k doubles!
- This memory is not released because we forgot the **`delete`** operator!
- At each turn more memory becomes unavailable until the system runs out of memory and crashes!

Advantages of Dynamic Memory Allocation

- No need to fix size of data to be used at compilation time
  - Easier to deal with real life use cases with variable and unknown number of data objects
  - No need to reserve very large but FIXED-SIZE arrays of memory
  - Example: interaction of particle in matter
    - How many particles are produced due to particle going through a detector?
    - Number not fixed a priori
    - Use dynamic allocation to create new particles as they are generated

- Disadvantage: correct memory management
  - Must keep track of **ownership** of **objects**
  - If not de-allocated can cause memory leaks which leads to slow execution and crashes
  - Most difficult part specially at the beginning or in complex systems

Destructor Method of a Class

- Constructor used by compiler to initialise instance of a class (an object)
  - Assign proper values to data members and allocate the object in memory

- Destructors are special member function doing reverse work of constructors
  - Do cleanup when object goes out of scope

- Destructor performs termination house keeping when objects go out of scope
  - No de-allocation of memory
  - Tells the program that memory previously occupied by the object is again free and can be re-used

- Destructors are ***FUNDAMENTAL*** when using dynamic memory allocation

Special Features of Destructors

- Destructors have no arguments

- Destructors do not have a return type
  - Similar to constructors

- Destructor of class Counter MUST be called **`~Counter()`**

```
#ifndef Counter_h_
#define Counter_h_
// Counter.h
#include <string>

class Counter {
  public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);
    void print();

  private:
    int count_;
    std::string name_;
};
#endif
```

Trivial Example of Destructor

Constructor initializes data members

```
#ifndef Counter_h_
#define Counter_h_
// Counter.h
#include <string>

class Counter {
  public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);
    void print();

  private:
    int count_;
    std::string name_;
};
#endif
```

Destructor does nothing

```
#include "Counter.h"
#include <iostream> // needed for input/output
using std::cout;
using std::endl;

Counter::Counter(const std::string& name) {
  count_ = 0;
  name_ = name;
  cout << "Counter::Counter() called for Counter "
       << name_ << endl;
};

Counter::~Counter() {
  cout << "Counter::~Counter() called for Counter "
       << name_ << endl;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment() {
  count_++;
}

void Counter::increment(int step) {
  count_ = count_+step;
}

void Counter::print() {
  cout << "Counter::print(): name: " << name_
       << "  value: " << count_ << endl;
}
```

Who and When Calls the Destructor?

Constructors are called by compiler when new objects are created

```
// app1.cpp
#include "Counter.h"
#include <string>

int main() {

  Counter c1( std::string("c1") );
  Counter c2( std::string("c2") );
  Counter c3( std::string("c3") );

  c2.increment(135);
  c1.increment(5677);

  c1.print();
  c2.print();
  c3.print();

  return 0;
}
```

Destructors are called implicitly by compiler when objects go out of scope!

Destructors are called in reverse order of creation

```
$ g++ -c Counter.cc
$ g++ -o app1 app1.cpp Counter.o
$ ./app1
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::print(): name: c1  value: 5677
Counter::print(): name: c2  value: 135
Counter::print(): name: c3  value: 0
Counter::~Counter() called for Counter c3
Counter::~Counter() called for Counter c2
Counter::~Counter() called for Counter c1
```

Create in order objects c1, c2, and c3

Destruct c3, c2, and c1

Another Example of Destructors

```
// app2.cpp
#include "Counter.h"
#include <string>

int main() {

  Counter c1( std::string("c1") );

  int count = 344;

  if( 1.1 <= 2.02 ) {
    Counter c2( std::string("c2") );

    Counter c3( std::string("c3") );
    if( count == 344 ) {
      Counter c4( std::string("c4") );
    }

    Counter c5( std::string("c5") );

    for(int i=0; i<3; ++i) {
      Counter c6( std::string("c6") );
    }
  }

  return 0;
}
```

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::Counter() called for Counter c4
Counter::~Counter() called for Counter c4
Counter::Counter() called for Counter c5
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::~Counter() called for Counter c5
Counter::~Counter() called for Counter c3
Counter::~Counter() called for Counter c2
Counter::~Counter() called for Counter c1
```

Who and When Calls the Destructor?

Constructors are called by compiler when new objects are created

```
// app1.cpp
#include "Counter.h"
#include <string>

int main() {

  Counter c1( std::string("c1") );
  Counter c2( std::string("c2") );
  Counter c3( std::string("c3") );

  c2.increment(135);
  c1.increment(5677);

  c1.print();
  c2.print();
  c3.print();

  return 0;
}
```

Destructors are called implicitly by compiler when objects go out of scope!

Destructors are called in reverse order of creation

```
$ g++ -c Counter.cc
$ g++ -o app1 app1.cpp Counter.o
$ ./app1
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::print(): name: c1  value: 5677
Counter::print(): name: c2  value: 135
Counter::print(): name: c3  value: 0
Counter::~Counter() called for Counter c3
Counter::~Counter() called for Counter c2
Counter::~Counter() called for Counter c1
```

Create in order objects c1, c2, and c3

Destruct c3, c2, and c1

Another Example of Destructors

```
// app2.cpp
#include "Counter.h"
#include <string>

int main() {

  Counter c1( std::string("c1") );

  int count = 344;

  if( 1.1 <= 2.02 ) {
    Counter c2( std::string("c2") );

    Counter c3( std::string("c3") );
    if( count == 344 ) {
      Counter c4( std::string("c4") );
    }

    Counter c5( std::string("c5") );

    for(int i=0; i<3; ++i) {
      Counter c6( std::string("c6") );
    }
  }

  return 0;
}
```

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::Counter() called for Counter c4
Counter::~Counter() called for Counter c4
Counter::Counter() called for Counter c5
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~Counter() called for Counter c6
Counter::~Counter() called for Counter c5
Counter::~Counter() called for Counter c3
Counter::~Counter() called for Counter c2
Counter::~Counter() called for Counter c1
```

Using **new** and **delete** Operators

```
// app6.cpp
#include "Counter.h"
#include "Datum.h"
#include <iostream>
using namespace std;

int main() {

  Counter c1("c1");

  Counter* c2 = new Counter("c2");
  c2->increment(6);

  Counter* c3 = new Counter("c3");

  Datum d1(-0.3,0.07);

  Datum* d2 = new Datum( d1 );
  d2->print();

  delete c2; // de-allocate memory!
  delete c3; // de-allocate memory!
  delete d2;

  return 0;
}
```

```
$ g++ -o app6 app6.cpp Datum.o Counter.o
$ ./app6
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
datum: -0.3 +/- 0.07
Counter::~Counter() called for Counter c2
Counter::~Counter() called for Counter c3
Counter::~Counter() called for Counter c1
```

Order of calls to destructors has changed!

delete calls explicitly the destructor of the object to de-allocate memory

Vital for objects holding pointers to dynamically allocated memory

Why no message when destructing d2 ?

`constant` Member Functions

- Enforce principle of least privilege
  - Give privilege ONLY if needed

- **`const`** member functions cannot
  - modify data members
  - cannot be called on non-constant objects

- **`const`** member functions tell user, the function only 'uses' the input data or data members but makes **no changes to data members**

- Pay attention which function can be called on which objects
  - Objects can be constant
    - You can not modify a constant object
    - calling non-constant methods on constant objects does not make sense!

Datum Class and const Member Functions

```
class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);

    double value() { return value_; }
    double error() { return error_; }
    double significance();
    void print();

    void setValue(double x) { value_ = x; }
    void setError(double x) { error_ = x; }

  private:
    double value_;
    double error_;
};
```

Which methods could become constant?

Datum Class with const Methods

All methods that only return a value and do not change the attributes of an object!

All getters can be constant

```
#ifndef Datum1_h
#define Datum1_h
// Datum1.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    void setValue(double x) { value_ = x; }
    void setError(double x) { error_ = x; }

  private:
    double value_;
    double error_;
};
#endif
```

what about setter methods?

```
#include "Datum1.h"
#include <iostream>
Datum::Datum() {
  value_ = 0.; error_ = 0.;
}
Datum::Datum(double x, double y) {
  value_ = x; error_ = y;
}
Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}
double
Datum::significance() const {
  return value_/error_;
}
void Datum::print() const {
  using namespace std;
  cout << "datum: " << value_
       << " +/- " << error_ << endl;
}
```

Typical error with constant methods

```
#ifndef Datum2_h
#define Datum2_h
// Datum2.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    void setValue(double x) const { value_ = x; }
    void setError(double x) const { error_ = x; }

  private:
    double value_;
    double error_;
};
#endif
```

setters can never be constant!

Setter method is used to modify data members

Similarly constructors and destructors can not be constant

```
$ g++ -c Datum2.cc
In file included from Datum2.cc:1:
Datum2.h: In member function `void Datum::setValue(double) const':
Datum2.h:18: error: assignment of data-member `Datum::value_' in read-only structure
Datum2.h: In member function `void Datum::setError(double) const':
Datum2.h:19: error: assignment of data-member `Datum::error_' in read-only structure
```

Example of Error using non-constant functions

```
#ifndef Datum4_h
#define Datum4_h
// Datum4.h
#include <iostream>
#include <string>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;

    void print(const std::string& comment) ;
    void setValue(double x) { value_ = x; }
    void setError(double x) { error_ = x; }
  private:
    double value_;
    double error_;
};
#endif
```

print MUST have been constant!

bad design of the class!

```
void Datum::print(const std::string& comment) {
  using namespace std;
  cout << comment << ": " << value_
       << " +/- " << error_ << endl;
}
```

```
// app1.cpp

#include "Datum4.h"

int main() {

  Datum d1(-67.03, 32.12 );
  const Datum d2(-67.03, 32.12 );

  d1.print("datum");

  d2.print("const datum");

  return 0;
}
```

```
$ g++ -o app1 app1.cpp Datum4.o
app1.cpp: In function `int main()':
app1.cpp:12: error: passing `const Datum' as `this'
argument of `void Datum::print(const std::string&)'
discards qualifiers
```

Default Values for Methods

- Functions (not only member functions in classes) might be often invoked with recurrent values for their arguments

- It is possible to provide default values for arguments of any function in C++
  - Default arguments must be provided the first time the name of the function occurs
    - In declaration if separate implementation
    - In definition if the function is declared and defined at the same time

- Only the right-most argument can be omitted
  - Including all arguments to the right of omitted argument

Example of Default Values

```
// Counter.h

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

  private:
    int count_;
};
```

Two increment() methods
but very similar functionality

increment() is a special case of
increment(int step) with step=1

Why two different methods?

```
// Counter.cc
// include class header files
#include "Counter.h"

// include any additional header files
//  needed in the class
// definition
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
  count_ = 0;
};


int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment() {
  count_++;
}

void Counter::increment(int step) {
  count_ = count_+step;
}
```

Default Value for `Counter::increment(int step)`

```
#ifndef Counter_Old_h_
#define Counter_Old_h_
// CounterOld.h

class Counter {
  public:
    Counter();
    int value();
    void reset();
    void increment(int step = 1);
  private:
    int count_;
};
#endif
```

Bad Practice!
Name of class different from name of file

```
// CounterOld.cc
#include "CounterOld.h"
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
  count_ = 0;
};

int Counter::value()  {
  return count_;
}

void Counter::reset() {
  count_ = 0;
}

void Counter::increment(int step) {
  count_ = count_+step;
}
```

```
// app3.cpp
#include "CounterOld.h" // old counter class
#include <iostream>
using namespace std;

int main() {

  Counter c1;

  c1.increment(); // no argument
  cout << "counter: " << c1.value() << endl;

  c1.increment(14); // provide argument, same function
  cout << "counter: " << c1.value() << endl;

  return 0;
}
```

```
$ g++ -c CounterOld.cc
$ g++ -o app3 app3.cpp
CounterOld.o
.$ ./app3
counter: 1
counter: 15
```

Ambiguous Use of Default Arguments

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    //Datum();
    Datum(double x=1.0, double
y=0.0);
    Datum(const Datum& datum);
    double value() { return value_; }
    double error() { return error_; }
    double significance();

  private:
    double value_;
    double error_;
};
#endif
```

Does it make sense to have default value and error?

```
$ g++ -c Datum.cc
$ g++ -o app4 app4.cpp Datum.o
$ ./app4
datum: -0.23 +/- 0.05
datum: 5.23 +/- 0
datum: 1 +/- 0
```

```
#include "Datum.h"

Datum::Datum(double x, double y) {
  value_ = x;
  error_ = y;
}

Datum::Datum(const Datum& datum) {
  value_ = datum.value_;
  error_ = datum.error_;
}

double
Datum::significance() {
  return value_/error_;
```

```
#include "Datum.h"
int main() {

  Datum d1(-0.23, 0.05); // provide arguments
  d1.print();

  Datum d2(5.23); // default error ...
  d2.print();

  Datum d3; // default value and error!
  d3.print();
  return 0;
}
```

Don’t Abuse Default Arguments!

- Default values must be used for functions very similar in functionality and with obvious default values

- If default values are not intuitive for user think twice before using them!

- Quite often different constructors correspond to DIFFERENT ways to create an object
  - Default values could be misleading

- If arguments are physical quantities ask yourself: is the default value meaningful and useful for everyone?

C++ Applications
command line arguments,
g++ options, I/O to/from file
External Libraries: ROOT

Options of g++

```
GCC(1)                                GNU                                GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [   |   |   ] [     standard]
           [   ] [     ] [   level]
           [   warn...] [                ]
           [   dir...] [   dir...]
           [   macro[=defn]...] [   macro]
           [   option...] [   machine-option...]
           [    outfile] [@file] infile...

       Only the most useful options are listed here; see below for the remainder.    accepts mostly the
       same options as     .

       When you invoke GCC, it normally does preprocessing, compilation, assembly and linking.  The
       "overall options" allow you to stop this process at an intermediate stage.  For example, the
       option says not to run the linker.  Then the output consists of object files output by the
       assembler.

       Other options are passed on to one stage of processing.  Some options control the preprocessor and
       others the compiler itself.  Yet other options control the assembler and linker; most of these are
       not documented here, since you rarely need to use any of them.

       Most of the command line options that you can use with GCC are useful for C programs; when an option
       is only useful with another language (usually C++), the explanation says so explicitly.  If the
       description for a particular option does not mention a source language, you can use that option with
       all supported languages.

       The     program accepts options and file names as operands.  Many options have multi-letter names;
       therefore multiple single-letter options may not be grouped:     is very different from    .
```

Some Already Familiar Options

```
$ ls -l color.*
-rw-r--r--  1 rahatlou None 601 May 22 13:10 color.cpp
```

- -E : stop after running pre-compiler to resolve pre-compiler directives
  - Don't compile nor link the binary

```
$ g++ -E -o color.pre-compiler color.cpp
$ ls -l color.*
-rw-r--r--  1 rahatlou None    601 May 22 13:10 color.cpp
-rw-r--r--  1 rahatlou None 681002 May 23 11:19 color.pre-compiler
```

- -c : stop after compilation
  - Doesn't link so no executable is produced

```
$ g++ -c color.cpp
$ ls -lrt color.*
-rw-r--r--  1 rahatlou None    601 May 22 13:10 color.cpp
-rw-r--r--  1 rahatlou None 681002 May 23 11:19 color.pre-compiler
-rw-r--r--  1 rahatlou None  29489 May 23 11:21 color.o
```

- -o : specify name of the output

```
$ g++ -c color.cpp
$ ls -lrt
-rw-r--r--  1 rahatlou None    601 May 22 13:10 color.cpp
-rw-r--r--  1 rahatlou None 681002 May 23 11:19 color.pre-compiler
-rw-r--r--  1 rahatlou None  29489 May 23 11:21 color.o
-rwxr-xr-x  1 rahatlou None 524423 May 23 11:22 a.out
```

Default name of binary

Increasing Warning Level

```
// app11.cpp
#include <string>
#include <iostream>
int index() {
  int i = 27;
}

std::string name() {
  std::string str("test of g++ options");
  return str;

  // text after return
  int j = 56;
}

int main() {
  int i = index();
  std::string st = name();
  std::cout << "i: " << i
            << "st: " << st
            << std::endl;
  return 0;
}
```

Very often simple warnings are a clear signal there is something seriously wrong

```
$ g++ -o app1 app1.cpp

$ g++ -o app1 -Wall app1.cpp
app1.cpp: In function `int index()':
app1.cpp:6: warning: unused variable 'i'
app1.cpp:7: warning: control reaches end of non-void function
app1.cpp: In function `std::string name()':
app1.cpp:14: warning: unused variable 'j'
$ ./app1
i:0      st: test of g++ options
```

Value of index() is always 0! ignores completely you implementation!

Debug Symbols with -g

- Produce debugging information to be used by debuggers, e.g. GDB
  - larger binary
- Extremely useful when first developing your code
  - You can see the high level labels (your function names and variables)
- It slows down a bit the code but might pay off in development phase
- Once code fully tested you can remove this option and fully optimize

-g Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, -g enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use -gstabs+, -gstabs, -gxcoff+, -gxcoff, or -gvms (see below).

Unlike most other C compilers, GCC allows you to use -g with -O. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

Libraries of Compiled Code

- Libraries are simple archives that contain compiled code (object files)

- Two types of libraries
  - Static Library: used by linker to include compiled code in the executable at linking time.
    - Larger executable since includes ALL binary code run during execution
    - Does not require presence of libraries at runtime since all code already included in the executable
  - Shared Library: used by executable at runtime
    - The binary holds only references to functions in the libraries but the code is not included in the executable itself
    - Smaller executable size but REQUIRES library to be available at runtime
    - We might discuss shared libraries in a future lecture

Creating and Using Static Libraries

```
$ g++ -c Datum.cc
$ g++ -c Result.cc
$ g++ -c InputService.cc
$ g++ -c Calculator.cc


$ ar -r libMyLib.a Datum.o Result.o InputService.o Calculator.o
ar: creating libMyLib.a


$ ar tv libMyLib.a
rw-r--r-- 1003/513   1940 May 23 12:21 2006 Datum.o
rw-r--r-- 1003/513    748 May 23 12:21 2006 Result.o
rw-r--r-- 1003/513   4482 May 23 12:21 2006 InputService.o
rw-r--r-- 1003/513  22406 May 23 12:21 2006 Calculator.o


$ g++ -o wgtavg wgtavg.cpp -lMyLib -L.


$ ./wgtavg
```

Commonly Used g++ Options with External Libraries

- Usually when using external libraries you are provided with
  - path to directory where you can find include files
  - path to directory where you can find librararies
  - NO access to source code!
    - But you don't need the source code to compile. Only header files. Remember only interface matters!

- -L : path to directory containing libraries
  - **`-L /usr/local/root/5.08.00/lib`**

- -I : path to directory containing header files
  - **`-I /usr/local/root/5.08.00/include`**

- -l : specify name of libraries to be used at link time
  - **`-l Core -lHbook`**
  - you don't have to specify the prefix "lib" nor the extension ".a"

Optimizing Your Executable

- g++ offers many options to optimize your executable and reduce execution time
  - Compiler analyzes your code to determine the best execution path
  - Takes longer to compile with optimization
  - It's harder to debug an optimized program
  - Remember: your optimized and non-optimized executables MUST give the same results or you have a bug!

Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Using the -funit-at-a-time flag will allow the compiler to consider information gained from later functions in the file when compiling a function. Compiling multiple files at once to a single output file (and using -funit-at-a-time) will allow the compiler to use information gained from all of the files when compiling each of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed.

Levels of Optimization

-O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O turns on the following optimization flags: -fdefer-pop -fmerge-constants -fthread-jumps -floop-optimize -fif-conversion -fif-conversion2 -fdelayed-branch -fguess-branch-probability -fcprop-registers

-O also turns on -fomit-frame-pointer on machines where doing so does not interfere with debugging.

-O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

-O2 turns on all optimization flags specified by -O. It also turns on the following optimization flags: -fforce-mem -foptimize-sibling-calls -fstrength-reduce -fcse-follow-jumps -fcse-skip-blocks -frerun-cse-after-loop -frerun-loop-opt -fgcse -fgcse-lm -fgcse-sm -fgcse-las -fdelete-null-pointer-checks -fexpensive-optimizations -fregmove -fschedule-insns -fschedule-insns2 -fsched-interblock -fsched-spec -fcaller-saves -fpeephole2 -freorder-blocks -freorder-functions -fstrict-aliasing -funit-at-a-time -falign-functions -falign-jumps -falign-loops -falign-labels -fcrossjumping

Please note the warning under -fgcse about invoking -O2 on programs that use computed gotos.

-O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -fweb and -frename-registers options.

-O0 Do not optimize. This is the default.

You should notice the difference in your application when using –O3

Passing Arguments to C++ Applications

```
// app12.cpp

#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {

  cout << "# of cmd line arguments argc: " << argc << endl;
  cout << "argv[0]: " << argv[0] << endl;

  cout << "Running " << argv[0] << endl;

  return 0;
}
```

```
$ g++ -o app2  app2.cpp
$ ./app2
# of cmd line arguments argc: 1
argv[0]: ./app2
Running ./app2
```

- **`argc`** is number of command line arguments
  - it includes the name of the application as well!

- **`argv`** is vector of pointers to characters
  - interprets each set of disjoint characters as a token

Passing non-string values

```
// args.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "# of cmd line arguments argc: " << argc << endl;
  cout << "argv[0]: " << argv[0] << endl;

  if(argc < 4 ) {
    cout << "Error... not enough arguments!" << endl;
    cout << "Usage: args <integer> <double> <string>" << endl;
    cout << "now exiting..." << endl;
    return -1; // can be used by user to determine error condition
  }

  int index = atoi( argv[1] );
  double mean = atof( argv[2] );
  std::string name( argv[3] );

  cout << "Running " << argv[0]
       << " with "
       << "index: " << index
       << ", mean: " << mean
       << ", name: " << name
       << endl;

  return 0;
}
```

`atoi`: converts char to int

`atof`: converts char to double

User responsibility to check validity of arguments provided at runtime

```
$ ./args
# of cmd line arguments argc: 1
argv[0]: ./args
Error... not enough arguments!
Usage: args <integer> <double> <string>
now exiting...
$ ./args 34
# of cmd line arguments argc: 2
argv[0]: ./args
Error... not enough arguments!
Usage: args <integer> <double> <string>
now exiting...
$ ./args 34 3
# of cmd line arguments argc: 3
argv[0]: ./args
Error... not enough arguments!
Usage: args <integer> <double> <string>
now exiting...
$ ./args 34 3.1322 sprogrammazione
# of cmd line arguments argc: 4
argv[0]: ./args
Running ./args with index: 34, mean: 3.1322, name: sprogrammazione
```

Input from file with ifstream

```
// readfile.cc

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {

  // file name
  const char filename[30] = "input.txt";

  // create object for input file
  ifstream infile(filename); //input file object

  // string to hold each line
  string line;

  // make sure input file is open otherwise exit
  if(!infile.is_open()) {
    cerr << "cant open inut file" << endl;
    return -1;
  }
```

Parsing input lines with **sscanf**

```
// readfile.cc -- continued

  // variables to read in from file at each line
  char nome[30];
  double val, errpos;
  float errneg;

  // loop over file until end-of-file
  while(! infile.eof() ) {
     // get current line
     getline(infile,line);
     if( line == "\n" || line == "" ) continue;

     // parse line with the provided format and put data in variables
     // NB: USING POINTERS TO VARIABLES
     // format:  %s string    %f  float   %lf  double
     sscanf(line.c_str(),"%s %lf %lf %f",nome,&val,&errpos, &errneg);

     // print out for debug purposes
     cout << "nome: " << nome
          << "\tvalore: " << val << "\terr pos: " << errpos
          << "\terr neg: " << errneg << endl;
  } // !eof

  infile.close(); // close input file before exiting
  return 0;
}
```

ROOT for data analysis

ROOT: An object oriented data analysis framework

ROOT Data Analysis Framework

http://root.cern.ch

Google Custom Search

Download Documentation News Support About Development Contribute

Getting Started | Reference Guide | Forum | Gallery

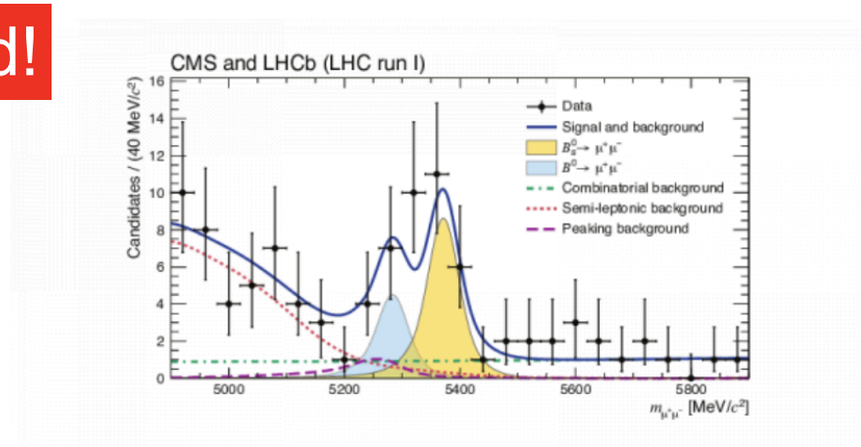
Reference guide is all you need!

ROOT is ...

A modular scientific software toolkit. It provides all the functionalities needed to deal with big data processing, statistical analysis, visualisation and storage. It is mainly written in C++ but integrated with other languages such as Python and R.

Start from examples or **try it in your browser!**

Download or Read More ...



Previous Resume Next

Under the Spotlight

2018-01-17 **ROOT Users' Workshop 2018**

The ROOT team would like to invite you to the 11th ROOT Users' Workshop.

2017-08-03 **The ROOT Docker container (alpha version)**

Do you like Docker? Would you like to use ROOT? We provide an *alpha* version of the ROOT Docker container!

2016-09-05 **Get the most out of the ROOT tutorials!**

Other News

2018-07-11 RDataFrame session at CHEP 2018

2018-06-15 ROOT::RWhy!

2017-03-08 Development release 6.09/02 is out!

2016-04-16 The status of reflection in C++

Latest Releases

Release 6.14/04 - 2018-08-23

ROOT Reference Guide

ROOT 6.15/01
Reference Guide

https://root.cern/doc/master/

ROOT Home | Main Page | Tutorials | Functional Parts | Namespaces | All Classes | Files | Release Notes

TH

The ROOT Mathematical Libraries
The Dataset Stager
th ROOT::R::TRInterface
TH1
TH1.cxx
TH1.h
TH1C
TH1C.h
TH1D
TH1D.h
TH1DModel
TH1Editor
TH1Editor.cxx
TH1Editor.h

ROOT Reference Documentation

Introduction

Welcome to ROOT

This documentation describes the software classes and functions that makes up the ROOT software system as well a
introduction of ROOT, for this please refer to the ROOT User Guides and Manuals. This documentation is generated c
kept up to date. The version of ROOT corresponding to this documentation is indicated at the page heading. You may also find in reference documentation page pointers to reference manuals for other ROOT versions.

How to use this reference documentation

The User's Classes in the top bar provides the user API, mainly the list of main Users' classes organized by module or functionality. The full list of classes, both for the public API and for the implementation details are available under the All Classes tab. A classification of classes based on their C++ namespace can be found under the Namespace tab. The fully indexed list of all source code is available under the tab Files.

ROOT provides other types of documentation:

- A general Users Guide is provided for a more in depth explanation of concepts and functionality available in the ROOT system.
- A number of topical User Guides and Manuals for various components of the system.
- A rich set of ROOT tutorials and code examples are offered to developers to exercise specific functionality.
- A rich set of HowTo's is also present to discuss issues commonly faced by ROOT users.

Caveat

We have moved recently to generate the documentation with Doxygen. To achieve this the comments in the source code needed to be formatted and written specifically for Doxygen to generated proper documentation. If you find missing documentation or inaccuracies please report them to our bug tracker. Detailed instructions on how to submit a bug can be found here.

ROOT 6.15/01 - Reference Guide Generated on Tue Oct 9 2018 05:58:41 (GVA Time) using Doxygen 1.8.13.

1D Histogram of floats: TH1F

List of all members | Public Member Functions | Protected Member Functions | Friends | List of all

TH1F Class Reference

Histogram Library

1-D histogram with a float per channel (see TH1 documentation)}

Definition at line 571 of file TH1.h.

Public Member Functions

	TH1F () Constructor. More...
	TH1F (const char *name, const char *title, Int_t nbinsx, Double_t xlow, Double_t xup) Create a 1-Dim histogram with fix bins of type float (see **TH1::TH1** for explanation of parameters) More...
	TH1F (const char *name, const char *title, Int_t nbinsx, const Float_t *xbins) Create a 1-Dim histogram with variable bins of type float (see **TH1::TH1** for explanation of parameters) More...
	TH1F (const char *name, const char *title, Int_t nbinsx, const Double_t *xbins) Create a 1-Dim histogram with variable bins of type float (see **TH1::TH1** for explanation of parameters) More...
	TH1F (const **TVectorF** &v) Create a histogram from a TVectorF by default the histogram name is "TVectorF" and title = "". More...
	TH1F (const **TH1F** &h1f) Copy Constructor. More...
virtual	**~TH1F** () Destructor. More...
virtual **void**	**AddBinContent** (**Int_t** bin) Increment bin content by 1. More...
virtual **void**	**AddBinContent** (**Int_t** bin, **Double_t** w) Increment bin content by a weight w. More...
virtual **void**	**Copy** (**TObject** &hnew) const Copy this to newth1. More...

Using ROOT

- Use classes from root in your application
- Use 1D histograms to plot your data
- Use canvas provided by root to store the histogram as output in a file (eps or gif)
- Use root functionalities to make your plot nicer
  - Change color, labels, names, fonts
- Become familiar with using external libraries without access to source files

Interface and Libraries are All You Need!

Public Member Functions

TH1F ()
Constructor. More...

https://root.cern/doc/master/classTH1F.html

TH1F (const char ***name**, const char *title, **Int_t** nbinsx, **Double_t** xlow, **Double_t** xup)
Create a 1-Dim histogram with fix bins of type float (see **TH1::TH1** for explanation of parameters) More...

TH1F (const char ***name**, const char *title, **Int_t** nbinsx, const **Float_t** *xbins)
Create a 1-Dim histogram with variable bins of type float (see **TH1::TH1** for explanation of parameters) More...

TH1F (const char ***name**, const char *title, **Int_t** nbinsx, const **Double_t** *xbins)
Create a 1-Dim histogram with variable bins of type float (see **TH1::TH1** for explanation of parameters) More...

TH1F (const **TVectorF** &**v**)
Create a histogram from a TVectorF by default the histogram name is "TVectorF" and title = "". More...

TH1F (const **TH1F** &h1f)
Copy Constructor. More...

virtual **~TH1F** ()
Destructor. More...

virtual void **AddBinContent** (**Int_t** bin)
Increment bin content by 1. More...

virtual void **AddBinContent** (**Int_t** bin, **Double_t** w)
Increment bin content by a weight w. More...

virtual void **Copy** (**TObject** &hnew) const
Copy this to newth1. More...

TH1F & **operator=** (const **TH1F** &**h1**)
Operator =. More...

virtual void **Reset** (**Option_t** *option="")
Reset. More...

virtual void **SetBinsLength** (**Int_t** n=-1)
Set total number of bins including under/overflow Reallocate bin contents array. More...

◆ TH1F() [2/6]

```
TH1F::TH1F ( const char *  name,
             const char *  title,
             Int_t         nbinsx,
             Double_t      xlow,
             Double_t      xup
           )
```

Create a 1-Dim histogram with fix bins of type float (see **TH1::TH1** for explanation of parameters)

Definition at line **9349** of file **TH1.cxx**.

Installing ROOT

- Information for downloading ROOT available at https://root.cern/downloading-root
- Binaries provided for variety of architecture and OS
  - See latest release 6.22/02: https://root.cern/releases/release-62202/

Binary distributions

Platform	Files	Size
CentOS 7	root_v6.22.02.Linux-centos7-x86_64-gcc4.8.tar.gz	186M
Fedora 30	root_v6.22.02.Linux-fedora30-x86_64-gcc9.3.tar.gz	225M
Fedora 31	root_v6.22.02.Linux-fedora31-x86_64-gcc9.3.tar.gz	225M
Fedora 32	root_v6.22.02.Linux-fedora32-x86_64-gcc10.2.tar.gz	227M
Ubuntu 16	root_v6.22.02.Linux-ubuntu16-x86_64-gcc5.4.tar.gz	200M
Ubuntu 18	root_v6.22.02.Linux-ubuntu18-x86_64-gcc7.5.tar.gz	218M
Ubuntu 19	root_v6.22.02.Linux-ubuntu19-x86_64-gcc9.2.tar.gz	223M
Ubuntu 20	root_v6.22.02.Linux-ubuntu20-x86_64-gcc9.3.tar.gz	224M
macOS 10.13 Xcode 10	root_v6.22.02.macosx64-10.13-clang100.pkg	315M
macOS 10.13 Xcode 10	root_v6.22.02.macosx64-10.13-clang100.tar.gz	200M
macOS 10.14 Xcode 10	root_v6.22.02.macosx64-10.14-clang100.pkg	314M
macOS 10.14 Xcode 10	root_v6.22.02.macosx64-10.14-clang100.tar.gz	200M
macOS 10.15 Xcode 11	root_v6.22.02.macosx64-10.15-clang110.pkg	314M
macOS 10.15 Xcode 11	root_v6.22.02.macosx64-10.15-clang110.tar.gz	200M

preview Windows Visual Studio 2019 (debug)	root_v6.22.02.win32.vc16.debug.exe	155M	
preview Windows Visual Studio 2019 (debug)	root_v6.22.02.win32.vc16.debug.zip	227M	
<!– REMOVED due to virus false positive:	**preview** Windows Visual Studio 2019	root_v6.22.02.win32.vc16.exe] (https://root.cern/download/root_v6.22.02.win32.vc16.exe)	84M – >
preview Windows Visual Studio 2019	`root_v6.22.02.win32.vc16.zip`	114M	

Using root libraries and header files

```
// app13.cc

#include  "TH1F.h"

int main() {

  TH1F h1;
  h1.Print();

  return 0;
}
```

Using ROOT libraries

```
csh> setenv ROOTSYS /home/rahatlou/root/5.11.02

bash> export ROOTSYS=/home/rahatlou/root/5.11.02
```

Needed at runtime to find libraries

```
csh> setenv LD_LIBRARY_PATH $ROOTSYS/lib

bash> export LD_LIBRARY_PATH=$ROOTSYS/lib
```

Provide path to header files and libraries

```
$ g++ -o app1 app1.cpp `$ROOTSYS/bin/root-config --cflags --libs`

$ ./app1
TH1.Print Name  = , Entries= 0, Total sum= 0
```

root-config

```
> $ROOTSYS/bin/root-config --cflags --libs
-pthread -stdlib=libc++ -std=c++11 -m64 -I/Users/
rahatlou/Library/root/v6.14.00/include -L/Users/rahatlou/
Library/root/v6.14.00/lib -lCore -lImt -lRIO -lNet -lHist
-lGraf -lGraf3d -lGpad -lROOTDataFrame -lROOTVecOps
-lTree -lTreePlayer -lRint -lPostscript -lMatrix
-lPhysics -lMathCore -lThread -lMultiProc -lpthread
-stdlib=libc++ -lm -ldl
```

- provides you with all options needed to compile and/or link your application

- Use at on command line with ` quotes instead of writing manually

- We will soon use makefiles to make such settings easier for users

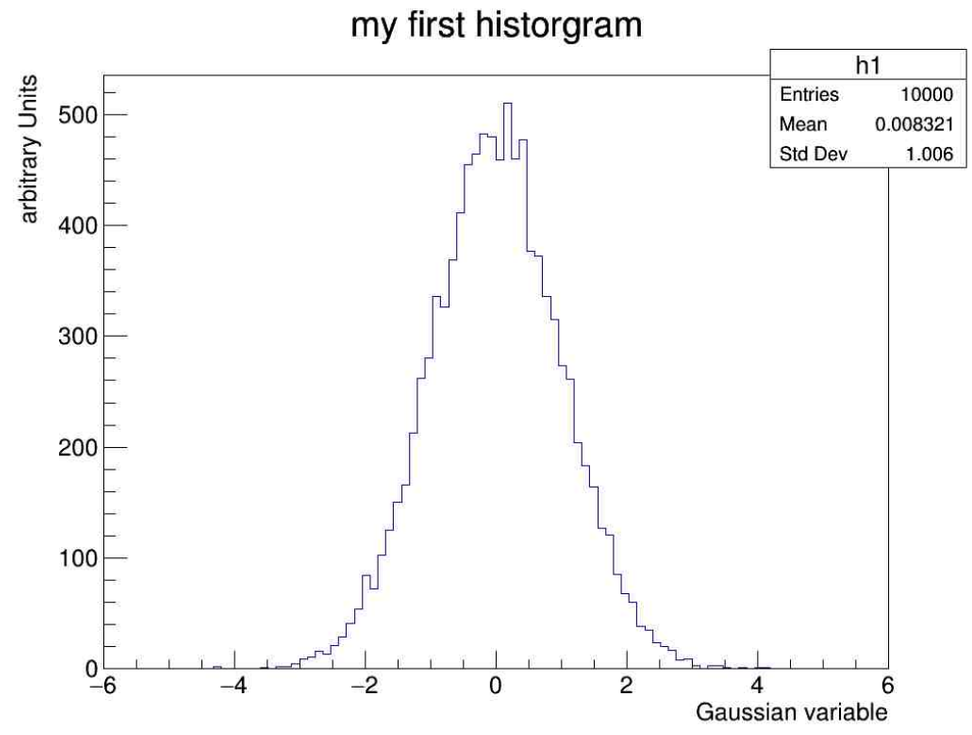
First classes to use

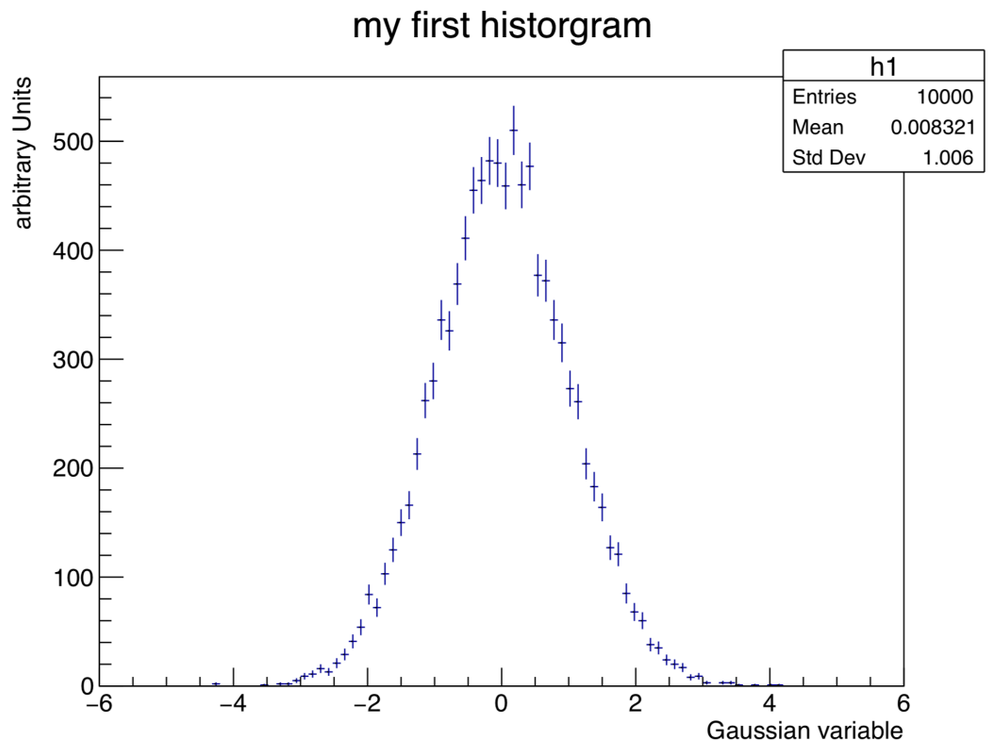
- **`TH1F`**: 1D histogram
  - look at constructors
  - public methods to add data to histogram
  - public methods to add comments or change labels of axes

- **`TCavnvas`**: canvas to draw your histogram
  - how to make one
  - changing properties such as color
  - drawing 1D histogram on a canvas
  - storing the canvas as a graphic file, e.g. eps or gif

Simple Example with TH1

```
// app14.cc
#include "TH1F.h"
#include "TCanvas.h"

int main() {

  // create histogram
  TH1F h1("h1","my first historgram",100,-6.0,6.0);

  // fill histogram with 10000 random gaussian numbers
  h1.FillRandom("gaus",10000);

  // add labels to axis
  h1.GetXaxis()->SetTitle("Gaussian variable");
  h1.GetYaxis()->SetTitle("arbitrary Units");

  // create a canvas to draw tour histogram
  TCanvas c1("c1","my canvas”,1024,800);

  // draw the histogram
  h1.Draw();

  // save canvas a JPG file
  c1.SaveAs("canvas.jpg");

  // change fill color to blue
  h1.SetFillColor(kBlue);

  // draw again the histogram
  h1.Draw();
  c1.SaveAs("canvas-blue.jpg");

  // dra histograms as points with errors
  h1.Draw("pe");

  // save canvas a PDF file
  c1.SaveAs("canvas-points.pdf");
  return 0;
}
```






```
$ g++ -Wall -o app14 app14.cc `root-config --libs --cflags`
$ ./app14
```

A Few Tips about Using ROOT

- Look at the reference guide to find out what is provided by the interface

- Start by simply creating new objects and testing them before making fancy use of many different classes

- Operators in C++
- Overloading operators
- special pointer **`this`**
- **`Examples`**
  - **`Class Datum`**
- More on dynamically allocated data members

Operators

Operation between Datum Objects

- Since Datum represents user data we could imagine having

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1.plus( d2 );

Datum d4 = d1.minus( d2 );

Datum d5 =
d1.product( d2 );
```

- These functions are easy to implement and provide behavior similar to doubles, ints, floats

- But they are functions not operators! They look different from what we are used to do with simple numbers

Operators

- C++ has a variety of built-in operators for built-in types

```
int i =8;
int j = 10;

int l = i + j;
int k = i * j;
```

- C++ allows you to implement such built-in operators also for user-defined types (classes!)

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1 + d2;
```

- This is called **overloading of operators**
  - We need to tell the compiler what to do when adding two Datum objects!

C++ Operators

- Binary operators require two operands
  - right-hand and left-hand operands

+	+=	<<=
-	-=	==
*	*=	!=
/	/=	<=
%	%=	>=
^	^=	&&
&	&=	||
|	|=	,
>	>>	()
<	<<	[]
=	>>=	->*

- Unary operators

+
-
*
&
->
~
!
++
--

Example of Overloaded Operator

```
class Datum {
  public:
    // interface same as before

    Datum operator+( const Datum& rhs ) const;

  private:
    // same data members
};
#endif
```

```
// app1.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );
  cout << "input data d1 and d2: " << endl;
  d1.print();
  d2.print();

  Datum d3 = d1 + d2;

  cout << "output d3 = d1+d2 " << endl;
  d3.print();

  Datum d4 = d1.operator+( d2 );
  d4.print();

  return 0;
}
```

```
#include "Datum.h"
#include <iostream>
#include <cmath>

// other member functions same as before

Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
$ g++ -Wall -o app1 app1.cpp Datum.cc
$ ./app1
input data d1 and d2:
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
output d3 = d1+d2
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

Understanding Overloading of Operators: the syntax

```
Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

- **`operator+`** is a member function of class Datum
  - it returns a Datum object in output by value
  - it has one argument called rhs
  - it is a constant function: can not modify the object it is applied to

- In this example we assume data points are not correlated
  - values are added
  - error on the sum is the sum in quadrature of the errors

Using Operators with Objects

- Operators can be called on objects exactly like any other member function of a class

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d4 = d1.operator+( d2 );
```

  - **`operator+`** is called on object **`d3`** with argument **`d2`** and result is stored in **`d4`**

- However, since they are operators, they can also be used like the operators for the built-in C++ types

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d3 = d1 + d2;
```

Operator versus Function

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x=1.0, double y=0.0);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    Datum sum( const Datum& rhs ) const;


  private:
    double value_;
    double error_;
};
#endif
```

```
Datum Datum::operator+( const Datum& rhs) const {

  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}


Datum Datum::sum( const Datum& rhs) const {

  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  Datum d4 = d1.sum( d2 );
  d3.print();
  d4.print();

  return 0;
}
```

```
$ g++ -Wall -o app2 app2.cpp Datum.cc
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

Why is operator+ constant?

- As usual, if not declared constant you can't call it constant objects

```
Datum Datum::operator+( const Datum& rhs ) {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
// app3.cpp
#include <iostream>
using namespace std;

#include "Datum1.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  const Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  d3.print();

  return 0;
}
```

```
$ g++ -Wall -o app3 app3.cpp Datum1.cc
app3.cpp: In function `int main()':
app3.cpp:12: error: passing `const Datum' as `this' argument of `
Datum Datum::operator+(const Datum&)' discards qualifiers
```

- Adding constant objects is perfectly reasonable
  - Your mistake! operator+ MUST be constant!

Rules of the Game: What You Can or Cannot Do

- You can overload any of the built-in C++ operators for your classes

- Overload operators for classes should mimic functionality of built-in operators for built-in types
  - operator * should not be implemented as a division!
  - Purpose of overloading operators is to extend the C++ language for custom user types (classes)
    - Overload only operators that are meaningful
    - What is the meaning of ++ operator for class Datum ?

- You CANNOT
  - create new operators but only overload existing ones
  - change meaning of operators for built-in types
  - change parity of operators: a binary operator can not be overloaded to become a unary operator

Lecture 7

20 oct 2020

Lecture 7

- More on operator +
- Assignment operator =
  - special **`this`** pointer
- Operations between custom class and built-in types
  - Complex with double
  - Datum with double
- Friend Methods
- Overloading operators for built-in types
- Global functions as a way of operator overloading
- Static data members and methods

Why is operator+ constant?

- As usual, if not declared constant you can't call it constant objects

```
Datum Datum::operator+( const Datum& rhs ) {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
// app13cpp
#include <iostream>
using namespace std;

#include "Datum1.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  const Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  d3.print();

  return 0;
}
```

```
$ g++ -Wall -o app13 app13.cpp Datum1.cc
app3.cpp: In function `int main()':
app3.cpp:12: error: passing `const Datum' as `this' argument of `
Datum Datum::operator+(const Datum&)' discards qualifiers
```

- Adding constant objects is perfectly reasonable
  - Your mistake! operator+ MUST be constant!

Assignment Operator `Datum::operator=(const Datum& rhs)`

```
class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    Datum sum( const Datum& rhs ) const;

    const Datum& operator=( const Datum& rhs );

  private:
    double value_;
    double error_;
};
```

```
const Datum& Datum::operator=(const Datum& rhs) {
  value_ = rhs.value_;
  error_ = rhs.error_;

  return *this;
}
```

remember `this` ?

```
$ g++ -Wall -o app14 app14.cpp Datum.cc
$ ./app14
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
```

operator= cannot be a constant method
We need to modify the object it is applied to!

```
// app14.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1;
  d3.print();

  Datum d4;
  d4.operator=(d2);
  d4.print();

  return 0;
}
```

Another Example of Use of Assignment Operator

```
// app15.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  Datum d1( 1.2, 0.3 );
  const Datum d2 = d1; // OK.. init the constant

  Datum d3( -0.2, 1.1 );
  d2 = d3; // error!

  return 0;
}
```

```
const Datum& Datum::operator=(const Datum& rhs) {
  value_ = rhs.value_;
  error_ = rhs.error_;

  return *this;
}
```

First assignment OK

Cannot modify a constant

```
$ g++ -Wall -o app15 app15.cpp Datum.cc
app5.cpp: In function `int main()':
app5.cpp:13: error: passing `const Datum' as `this' argument of
`const Datum& Datum::operator=(const Datum&)' discards qualifiers
```

Special Pointer `this` in a Class

- Special pointer provided in C++
- Allows an object to get a pointer to itself from within any member function of the class
- Useful when an object (instance of a class) has to compare itself with other objects
- Particularly useful for overloading operators
  - many operators are used to modify an object: =, +=, *=, etc.
  - All these operators should return an object of the type of the class
  - When overloading you want an object to modify itself AND return itself

One More Example of **this**

```
// this.cpp
#include <iostream>
#include <string>
using namespace std;

class Example {
  public:
   Example() { name_ = ""; }
   Example(const string& name);
   void printSelf() const;
  private:
   string name_;
};

Example::Example(const string& name) {
  name_ = name;
}


void
Example::printSelf() const {
  cout << "name: " << name_
       << "\t this: " << this
       << endl;
}
```

```
int main() {
  Example ex1("ex1");
  ex1.printSelf();

  cout << "&ex1: " << &ex1 <<
endl;

  return 0;
}
```

```
$ g++ -o this this.cpp
$ ./this
name: ex1          this: 0x23eef0
&ex1: 0x23eef0
```

this is the reference of ex1 accessible from within ex1

Exercise

- Complete class Datum with remaining operators and make sure errors are treated correctly (assuming no correlation)
  - for example * and /
  - add operator to multiply Datum by float

```
Datum d1(-1.1, 0.2);
Datum d2 = d1 * 3.5;
```

  - How can you take into account correlations between Datum objects?

Division and Multiplication of **Datum**

```
// app1.cc
#include <iostream>
using namespace std;

#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -3.4, 0.7 );
  d1.print();
  d2.print();

  Datum d3 = d1 * d2;
  Datum d4 = d1.operator*(d2);

  d3.print();
  d4.print();

  Datum d5 = d1 / d2;
  Datum d6 = d2/d1 ;
  d5.print();
  d6.print();

  return 0;
}
```

```
$ g++ -Wall -o app1 app1.cc Datum.cc
$ ./app1
datum: 1.2 +/- 0.3
datum: -3.4 +/- 0.7
datum: -4.08 +/- 1.32136
datum: -4.08 +/- 1.32136
datum: -0.352941 +/- 0.114305
datum: -2.83333 +/- 0.917613
```

```
Datum operator*( const Datum& rhs ) const;
Datum operator/( const Datum& rhs ) const;
```

```
Datum Datum::operator*(const Datum& rhs) const {
  double val = value_*rhs.value_;

  // propagate correctly the error for x*y
  double err = sqrt( rhs.value_*rhs.value_*error_*error_ +
                     rhs.error_*rhs.error_*value_*value_ );
  return Datum(val,err);
}

Datum Datum::operator/(const Datum& rhs) const {
  double val = value_ / rhs.value_;

  // propagate correctly the error for x / y
  double err = fabs(val) * sqrt( (error_/value_)*(error_/value_) +
                                 (rhs.error_/rhs.value_)*
                                 (rhs.error_/rhs.value_) );

  return Datum(val,err);
}
```

To be meaningful you must compute correctly the error for the result as expected by the user

Otherwise your class is incorrect

Interactions between **Datum** and **double**

- It's intuitive to multiply a Datum by a double
- No problem… overload the * operator with necessary signature

```
// app2.cc

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d2 = d1 * 1.5;
  d2.print();

  return 0;
}
```

```
class Datum {
  public:
    Datum operator*( const double& rhs ) const;
  // …

};

Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
$ g++ -Wall -o app2 app2.cc Datum.cc
$ ./app2
datum: 1.2 +/- 0.3
datum: 1.8 +/- 0.45
```

What about `double * Datum` ?

- Of course it is natural to do also
  - No reason to limit users to multiply always in a specific way
  - Not natural and certainly not intuitive

```
// app3.cc

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

- But this code does not compile
  - Do you understand why?

```
$ g++ -Wall -o app3 app3.cc Datum.cc
app3.cpp: In function `int main()':
app3.cpp:10: error: no match for 'operator*' in '5.0e-1 * d1'
```

- Whose operator must be overloaded?
  - operator * of class Datum ?
  - operator * of type double ?

More on What about **double*Datum**

- The following statement

```
double x = 0.5
Datum d3 = x * d1;
```

  is equivalent to

```
double x = 0.5
Datum d3 = x.operator*( d1 );
```

- This means that we need operator * of type double to be overloaded, something like

```
class double {
  public:
   Datum operator*( const Datum& rhs );

};
```

- This is not allowed!
  - Remember: We can not overload operators for built in types!

- So? should we define a new double **MyDouble** just for this? Seems crazy!
  - How many times we might need such functionality?

A new Global Function

- We can define a global function to do what we need
  - Declaration in header file OUTSIDE class scope
  - Implementation in source file
- It works but not as natural to use

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
Datum productDoubleDatum(const double& lhs, const Datum& rhs);
#endif
```

```
// Datum.cc
#include "Datum.h"
// implement all member functions

// global function!
Datum productDoubleDatum(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

```
$ g++ -Wall -o app3 app3.cc Datum.cc
$ ./app3
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

```
// app3.cc

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = productDoubleDatum(0.5,d1);
  d3.print();

  return 0;
}
```

Overloading Operators as Global Functions

- We can define a global operator to do exactly what we need
  - Declaration in header file OUTSIDE class scope
  - Implementation in source file. No scope operator needed
    - Not a member function

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
Datum operator*(const double& lhs, const Datum& rhs);
#endif
```

```
// Datum.cc
#include "Datum.h"
// implement all member functions

// global function!
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

```
// app4.cc

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

```
$ g++ -Wall -o app4 app4.cc Datum.cc
$ ./app3
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

Another Example: Overloading **operator<<()**

```
// Datum.cc
#include "Datum.h"
// implement all member functions

// global functions
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << rhs.value() << " +/- "
     << rhs.error();
  return os;
}
```

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
ostream& operator<<(ostream& os, const Datum& rhs);
#endif
```

```
// app5.cc
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();
  cout << d3 << endl;

  return 0;
}
```

```
$ g++ -Wall -o app5 app5.cc Datum.cc
$ ./app5
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
0.6 +/- 0.15
```

Overhead of operator overloading with global functions

```
// Datum.cc
#include “Datum.h”
// implement all member functions

// global functions
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value() << " +/- “
     << rhs.error() << endl;
  return os;
}
```

- Global functions don’t have access to private data of objects
- Necessary to call public methods to access information
  - Two calls for each cout or even simple product
- Overhead of calling functions can become significant if a frequently used operator is overloaded via global functions

Overloading `bool Datum::operator<(const Datum& rhs)`

```
class Datum {
  public:

    bool operator<(const Datum& rhs) const;

   // …
}
```

return type is boolean

constant method since does not modify the object being applied to

```
bool Datum::operator<(const Datum& rhs) const {
  return ( value_ < rhs.value_ );
}
```

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d3( -0.2, 1.1 );
  cout << "d1: " << d1 << endl;
  cout << "d3: " << d3 << endl;

  if( d1 < d3 ) {
    cout << "d1 < d3" << endl;
  } else {
    cout << "d3 < d1" << endl;
  }

  return 0;
}
```

Comparison based on the value_

error_ does not affect the comparison do you agree?

```
$ g++ -Wall -o app7 app7.cc Datum.cc
$ ./app7
d1:
datum: 1.2 +/- 0.3
d3:
datum: -0.2 +/- 1.1
d3 < d1. d3 is:
```

`friend` Methods

```
#ifndef DatumNew_h
#define DatumNew_h
// DatumNew.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
   // … other methods

    const Datum& operator=( const Datum& rhs );
    bool operator<(const Datum& rhs) const;

    Datum operator*( const Datum& rhs ) const;
    Datum operator/( const Datum& rhs ) const;

    Datum operator*( const double& rhs ) const;

    friend Datum operator*(const double& lhs, const Datum& rhs);
    friend ostream& operator<<(ostream& os, const Datum& rhs);

  private:
    double value_;
    double error_;
};
#endif
```

```
// DatumNew.cc
#include "DatumNew.h"
// implement all member functions

// global functions
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value_, lhs*rhs.error_ );
}

ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value_ << " +/- "
     << rhs.error_; // NB: no endl!
  return os;
}
```

global methods declared **`friend`** within the class can access private members without being a member functions

```
$ g++ -o app6 app6.cc DatumNew.cc
$ ./app6
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
0.6 +/- 0.15
```

Typical Error: Operators += and <=

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d3( -0.2, 1.1 );

  d1 += d3;

  if( d1 <= d3 ) {
    cout << "d1 < d3. d1 is:" << endl;
  } else {
    cout << "d3 < d1. d3 is:" << endl;
  }


  return 0;
}
```

```
$ g++ -Wall -o app8 app8.cc Datum.cc
app8.cc: In function `int main()':
app8.cc:12: error: no match for 'operator+=' in 'd1 += d3'
app8.cc:14: error: no match for 'operator<=' in 'd1 <= d3'
```

Having defined =, +, and < separately does not provide automatically += and <=

These must be overloaded explicitly by the user

Tip:
Use < to quickly implement > and >= as well

Datum::operator+=()

```
class Datum {
 //…
   Datum operator+( const Datum& rhs ) const;
   const Datum& operator+=( const Datum& rhs );

};
```

```
// app9.cc
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( 3.1, 0.4 );

  d1 += d2;
  d1.print();

  return 0;
}
```

```
// Datum.cc

const Datum& Datum::operator+=(const Datum& rhs) {
  value_ += rhs.value_;
  error_ = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return *this;
}
```

```
$ g++ -o app9 app9.cc Datum.cc
$ ./app9
d1: 1.2 +/- 0.3 d2: 3.1 +/- 0.4
d1+d2 = 4.3 +/- 0.5
```

Why `const& Datum operator+=()` ?

- Why not return by value?

```
Datum Datum::operator+=(const Datum& rhs) {
  double value = value_ + rhs.value_;
  double error = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return Datum(value,error);
}
```

- Why not return simple non-const reference?
  - non-const will also work almost always
  - use cases why const is needed not very common

```
Datum& Datum::operator+=(const Datum& rhs) {
  value_ += rhs.value_;
  error_ = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return *this;
}
```

Problem with Returning by-value

```
class Foo {

public:
  Foo() { name_ = ""; x_= 0; }
  Foo(const std::string& name, const double x) { name_ = name; x_
= x; }
  double value() const { return x_; }
  std::string name() const { return name_; }

  Foo operator=(const Foo& rhs) {
    Foo aFoo(rhs.name_,rhs.x_);
    cout << "In Foo::operator=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  Foo operator+=(const Foo& rhs) {
    Foo aFoo(std::string(name_+"+"+rhs.name_), x_ + rhs.x_);
    cout << "In Foo::operator+=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  void reset() {
    x_ = 0.;
    name_ = "";
  }

private:
  double x_;
  std::string name_;
};

// global function
ostream&  operator<<(ostream& os, const Foo& foo) {
  os << "Foo name: "  << foo.name() << " value: " << foo.value()
     << " address: " << &foo;
  return os;
}
```

```
// fooapp3.cc
int main() {
  Foo f1("f1",1.);
  Foo f2("f2",2.);
  Foo f3("f3",3.);

  cout << " before f1+=f2 " << endl;
  f1 += f2;
  cout << "after f1+=f2\n" << f1 << endl;

  cout << " before f1 = f3 " << endl;
  f1 = f3;
  cout << "after f1 = f3\n" << f1 << endl;

  return 0;
}
```

```
$ g++ -o fooapp3 fooapp3.cc
$ ./fooapp3
 before f1+=f2
In Foo::operator+=: value: 3, name: f1+f2, &aFoo: 0x7ffeec53e6e8
after f1+=f2
Foo name: f1 value: 1 address: 0x7ffeec53e7a0
 before f1 = f3
In Foo::operator=: value: 3, name: f3, &aFoo: 0x7ffeec53e6c8
after f1 = f3
Foo name: f1 value: 1 address: 0x7ffeec53e7a0
```

Assignment never happens! the left-hand-side is never modified by the operators

static data and methods

Shared data between Objects

- Objects are instances of a class
  - Each object has a copy of data members that define the attributes of that class
  - Attributes are initialized in the constructors or modified through setters or dedicated member functions

- What if we wanted some data to be shared by ALL instances of class ?
  - Example: keep track of how many instances of a class are created

- How can we do the book keeping?
  - External registry or counter.
    - Where should such a counter live?
    - how can it keep track of ANYBODY creating objects?
    - How to handle the scope problem?

Examples of Sharing Data between Objects

- High energy physics
  - Production vertex for particles in a collision

- Perhaps more interesting example for you… Video Games!
  - Think about any of the flavors of WarCraft, StarCraft, Command and Conquer, Civilization, Halo, Clash of Clans, Fortnite, etc
  - The humor and courage of your units depend on how many of them you have
    - If there are many soldiers you can easily conquer new territory
    - If you have enough resources you can build new facilities or many new manpower

- How can you keep track of all units and facilities present in all different parts of a complex game?
  - static might just do it!

Tolerance for comparing Datum

- Comparison between two Datum objects

```
Datum d1(1.01, 0.131);
Datum d2(0.99, 0.128);

if( d1 == d2 ) {
 // do something
  .. ..
}
```

- When should == be true ?

- In a physics problem you often define a numerical tolerance or a detector precision when comparing measurements
  - All Datum objects could share a same tolerance for comaparison

`static` Data Members

- static data member is common to ALL instances of a class
  - All object use **exactly** the same data member
  - There is really **only one copy** of static data members accessed by all objects

```
#ifndef Unit_h
#define Unit_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
     operator<<(std::ostream& os,
                const Unit& unit);

    static int counter_;

  private:
    std::string name_;
};
#endif
```

```
#include "Unit.h"
using namespace std;

// init. static data member.
// NB: No static keyword necessary.
// Otherwise... compilation error!
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << unit.name_ << " Total Units: "
     << unit.counter_;
  return os;
}
```

Example of **`static`** data member

```
#include "Unit.h"
using namespace std;

// init. static data member.
// NB: No static keyword necessary.
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os,
           const Unit& unit) {
  os << unit.name_ << " Total Units: "
     << unit.counter_;
  return os;
}
```

```
int main() {
  Unit john("John");
  cout << john << endl;

  cout << "&john.counter_: "
       << &john.counter_ << endl;

  Unit* fra = new Unit("Francesca");
  Unit pino("Pino");
  cout << "&pino.counter_: "
       << &pino.counter_ << endl;

  cout << "&(fra->counter_): “
       << &(fra->counter_) << endl;
  cout << pino << endl;

  delete fra;

  cout << pino << endl;

  return 0;
}
```

All objects use the same variable!

constructor and destructor in charge of bookkeeping

```
$ g++ -Wall -o static1 static1.cpp Unit.cc
$ ./static1
John Total Units: 1
&john.counter_: 0x449020
&pino.counter_: 0x449020
&(fra->counter_): 0x449020
Pino Total Units: 3
Pino Total Units: 2
```

Using member functions with static data

```
#ifndef Unit2_h
#define Unit2_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
    operator<<(std::ostream& os,
               const Unit& unit);

    int getCount() { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
#include "Unit2.h"
using namespace std;

// init. static data member
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
}

Unit::~Unit() {
  counter_--;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << "My name is " << unit.name_
     << "! Total Units: " << unit.counter_;
  return os;
}
```

- All usual rules for functions, arguments etc. apply

- Nothing special about public or private static members or functions returning static members

Does it make sense to ask objects for static data?

```
// static2.cpp

#include <iostream>
#include <string>
using namespace std;
#include "Unit2.h"


int main() {
  Unit john("John");
  Unit* fra = new Unit("Francesca");
  cout << "john.getCount(): " << john.getCount() << endl;
  cout << "fra->getCount(): " << fra->getCount() << endl;

  delete fra;

  return 0;
}
```

```
$ g++ -Wall -o static2 static2.cpp Unit2.cc
$ ./static2
john.getCount(): 2
fra->getCount(): 2
```

- counter_ is not really an attribute of any objects
  - It is mostly a general feature of all objects of type Unit

- In principle we would like to know how many Units we have regardless of a specific Unit object

- But how can we use a function if no object has been created?

static member functions

`static` member functions

- static member functions of a class can be called without having any object of the class!

- Mostly (but not only) used to access static data members
  - static data members exist before and after and regardless of objects
  - static functions play the same role

- Common use of static functions is in utility classes which have no data member
  - Some classes are mostly place holders for commonly used functionalities
    - we will see a number of such classes in ROOT for mathematical

Example of **`static`** Member Function

```
#ifndef Unit3_h
#define Unit3_h
#include <string>
#include <iostream>
class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream&
    operator<<(std::ostream& os,
               const Unit& unit);

    static int getCount() { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
int main() {
  cout << "units: " << Unit::getCount() << endl;

  Unit john("John");
  Unit* fra = new Unit("Francesca");

  cout << "john.getCount(): " << john.getCount() << endl;
  cout << "fra->getCount(): " << fra->getCount() << endl;
  delete fra;

  cout << "units: " << Unit::getCount() << endl;

  return 0;
}
```

```
#include "Unit3.h"
using namespace std;

// init. static data member
int Unit::counter_ = 0;

Unit::Unit(const std::string& name) {
  name_ = name;
  counter_++;
  cout << "Unit(" << name
       <<") called. Total Units: "
       << counter_ << endl;
}

Unit::~Unit() {
  counter_--;
  cout << "~Unit() called for "
       <<  name_ << ". Total Units: "
       << counter_ << endl;
}

ostream&
operator<<(ostream& os, const Unit& unit) {
  os << "My name is " << unit.name_
     << "! Total Units: " << unit.counter_;
  return os;
}
```

```
$ g++ -Wall -o static3 static3.cpp Unit3.cc
$ ./static3
units: 0
Unit(John) called. Total Units: 1
Unit(Francesca) called. Total Units: 2
john.getCount(): 2
fra->getCount(): 2
~Unit() called for Francesca. Total Units: 1
units: 1
~Unit() called for John. Total Units: 0
```

Common Error with **`static`** Member Functions

```
#ifndef Unit3_h
#define Unit3_h

#include <string>
#include <iostream>

class Unit {
  public:
    Unit(const std::string& name);
    ~Unit();

    std::string name() const { return name_; }
    friend std::ostream& operator<<(std::ostream& os, const Unit& unit);

    static int getCount() const { return counter_; }

  private:
    static int counter_;
    std::string name_;
};
#endif
```

```
$ g++ -Wall -c Unit3.cc
In file included from Unit3.cc:1:
Unit3.h:15: error: static member function `static int
Unit::getCount()'
 cannot have `const' method qualifier
```

Typical error! static functions can not be **`const`**! Since they can be called without any object no reason to make them constant

Features of static methods

- They cannot be constant
  - static functions operate independently from any object
  - They can be called before and after any object is created

- They can not access non-static data members of the class
  - non-static data members characterize objects
  - how can data members be modified if no object created yet?

```
class Unit {
  public:

    static int getCount() {
      name_ = "";
      return counter_;
    }

};
```

```
$ g++ -c  Unit4.cc
In file included from Unit4.cc:1:
Unit4.h: In static member function `static int Unit::getCount()':
Unit4.h:21: error: invalid use of member `Unit::name_' in
static member function
Unit4.h:15: error: from this location
```

- No access to **`this`** pointer in static functions
  - Recall: **`this`** is specific to individual objects

`static` Methods in Utility Classes

- Classes with no data member and (only) static methods are often called utility classes

```
#ifndef Calculator_h
#define Calculator_h

#include <vector>
#include "Datum.h"

class Calculator {
 public:
  Calculator();

  static Datum
    weightedAverage(const std::vector<Datum>& dati);
  static Datum
    arithmeticAverage(const std::vector<Datum>& dati);
  static Datum
    geometricAverage(const std::vector<Datum>& dati);
  static Datum
    fancyAverage(const std::vector<Datum>& dati);

};
#endif
```

Example of Application

- Application to compute weighted average and error
  - Application must accept an arbitrary number of input data
  - Each data has a central value x and uncertainty
  - Compute weighted average of input data and uncertainty on the average
- Possible extensions
  - Provide different averaging methods
  - Uncertainties could be also asymmetric ($x\ ^{+\sigma 1}\ _{-\sigma 2}$)
  - Consider also systematic errors
  - Compute correlation coefficient and take it into account when computing the average and its uncertainty
  - Use ROOT to make histogram of data points and plot a coloured band to indicate the average and its uncertainty overlaid on the histogram

Possible implementation

```
// wgtavg.cc
#include <vector>
#include <iostream>

#include “Datum.h"// basic data object
#include “InputService.h" // class dedicated to handle input of data
#include “Calculator.h” // implements various algorithms

using std::cout;
using std::endl;

int main() {

 std::vector<Datum> dati = InputService::readDataFromUser();

  Datum r1 = Calculator::weightedAverage(dati);
  cout << “weighted average: “ << r1 << endl;

  Datum r2 = Calculator::arithmeticAverage(dati);

  return 0;
}
```

Interface of Classes

```
#ifndef Calculator_h
#define Calculator_h

#include <vector>
#include "Datum.h"

class Calculator {
 public:
  Calculator();

  static Datum
    weightedAverage(const std::vector<Datum>& dati);
  static Datum
    arithmeticAverage(const std::vector<Datum>& dati);

};
#endif
```

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value() ;
    double error() ;
    double significance();
  private:
    double value_;
    double error_;
};
#endi
```

```
#ifndef InputService_h
#define InputService_h
#include <vector>
#include "Datum.h"

class InputService {
 public:
   InputService();
   static std::vector<Datum> readDataFromUser();
 private:
};
#endif
```

You see the interface but don't know how the methods are implemented!

Application for Weighted Average

```
// wgtavg.cc
#include <vector>
#include <iostream>

#include “Datum" // basic data object
#include "InputService" // class dedicated to handle input of data
#include "Calculator" // impelments various algorithms

using std::cout;
using std::endl;

int main() {

  std::vector<Datum> dati = InputService::readDataFromUser();

  Datum r1 = Calculator::weightedAverage(dati);
  cout << “weighted average: “ << r1 << endl;

  Datum r2 = Calculator::arithmeticAverage(dati);

  return 0;
}
```

```
$ g++ -c InputService.cc
$ g++ -c Datum.cc
$ g++ -c Calculator.cc
$ g++ -o wgtavg wgtavg.cpp InputService.o Datum.o Calculator.o
```

Questions

- What about reading a file of data?
  - how to communicate the file name and where?
    - in main or in InputService?
- Do you need any arguments for these functions?
- Who should compute correlation?
  - should be stored?
    - if yes, where?
  - should the data become an attribute of some object?
    - If yes, in which class?
- what about generating pseudo-data to test our algorithms?
  - where would this generation happen?
  - in the main() method or in some class?

Lecture 8

23 oct 2020

Lecture 8

- Operator == for Datum
  - Static Data Members

- Enumeration

- Use of `std::pair`, `std::vector`, `std::map`

Class **Datum**

- Use static data member to implement operator == for Datum
  - Implement also <= and >= with similar logic

```
class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    const Datum& operator+=( const Datum& rhs );

    Datum sum( const Datum& rhs ) const;

    const Datum& operator=( const Datum& rhs );


    bool operator==(const Datum& rhs) const;
    bool operator<(const Datum& rhs) const;

    Datum operator*( const Datum& rhs ) const;
    Datum operator/( const Datum& rhs ) const;

    Datum operator*( const double& rhs ) const;



    friend Datum operator*(const double& lhs, const Datum& rhs);
    friend std::ostream& operator<<(std::ostream& os, const Datum& rhs);

    static void setTolerance(double val) { tolerance_ = val; };

  private:
    double value_;
    double error_;
    static double tolerance_;
};
```

```
#include "Datum.h"
#include <iostream>
#include <cmath>
using std::cout;
using std::endl;
using std::ostream;

double Datum::tolerance_ = 1e-4;

// functions …

bool Datum::operator==(const Datum& rhs) const {
  return (fabs(value_-rhs.value_)< tolerance_ &&
          fabs(error_-rhs.error_)< tolerance_    );
}
```

Using Datum::tolerance_

```
// app1.cc
#include "Datum.h"
#include <iostream>
using std::cout;
using std::endl;

int main() {

    Datum d1(-1.1,0.1);
    Datum d2(-1.0, 0.2);
    Datum d3(-1.11, 0.099);
    Datum d4(-1.10001, 0.09999999);

    cout << "d1: " << d1 << endl;
    cout << "d2: " << d2 << endl;
    cout << "d3: " << d3 << endl;
    cout << "d4: " << d4 << endl;


    for(double eps = 0.1; eps > 1e-8; eps /= 10) {
      Datum::setTolerance( eps );
      cout << "Datum tolerance  = " << eps << endl;

      if( d1 == d2 ) cout << "\t d1 same as d2" << endl;
      if( d1 == d3 ) cout << "\t d1 same as d3" << endl;
      if( d1 == d4 ) cout << "\t d1 same as d4" << endl;
    }
    return 0;
}
```

```
$ g++ -o /tmp/app app1.cc Datum.cc
$ /tmp/app
d1: -1.1 +/- 0.1
d2: -1 +/- 0.2
d3: -1.11 +/- 0.099
d4: -1.10001 +/- 0.1
Datum tolerance  = 0.1
          d1 same as d3
          d1 same as d4
Datum tolerance  = 0.01
          d1 same as d4
Datum tolerance  = 0.001
          d1 same as d4
Datum tolerance  = 0.0001
          d1 same as d4
Datum tolerance  = 1e-05
          d1 same as d4
Datum tolerance  = 1e-06
Datum tolerance  = 1e-07
Datum tolerance  = 1e-08
```

IO manipulators

```
//app2.cc
#include "Datum.h"
#include <iostream>
#include <iomanip>       // std::setprecision

using std::cout;
using std::endl;

int main() {

    Datum d1(-1.1,0.1);
    Datum d2(-1.0, 0.2);
    Datum d3(-1.101, 0.099);
    Datum d4(-1.10001, 0.09999999);

    cout << "d1: " << std::setprecision(9) << d1 << endl;
    cout << "d2: " << std::setprecision(9) << d2 << endl;
    cout << "d3: " << std::fixed << d3 << endl;
    cout << "d4: " << std::fixed << d4 << endl;


    for(double eps = 0.1; eps > 1e-8; eps /= 10) {
      Datum::setTolerance( eps );
      cout << "Datum tolerance  = " << std::scientific << eps << endl;

      if( d1 == d2 ) cout << "\t d1 same as d2" << endl;
      if( d1 == d3 ) cout << "\t d1 same as d3" << endl;
      if( d1 == d4 ) cout << "\t d1 same as d4" << endl;
    }
    return 0;
}
```

```
$ g++ -o /tmp/app app2.cc Datum.cc
$ /tmp/app
d1: -1.1 +/- 0.1
d2: -1 +/- 0.2
d3: -1.101000000 +/- 0.099000000
d4: -1.100010000 +/- 0.099999990
Datum tolerance  = 1.000000000e-01
          d1 same as d3
          d1 same as d4
Datum tolerance  = 1.000000000e-02
          d1 same as d3
          d1 same as d4
Datum tolerance  = 1.000000000e-03
          d1 same as d4
Datum tolerance  = 1.000000000e-04
          d1 same as d4
Datum tolerance  = 1.000000000e-05
          d1 same as d4
Datum tolerance  = 1.000000000e-06
Datum tolerance  = 1.000000000e-07
Datum tolerance  = 1.000000000e-08
```

Enumerators

Enumerators

- Enumerators are set of integers referred to by identifiers
- There is natural need for enumerators in programming
  - Months: Jan, Feb, Mar, …, Dec
  - Fit Status: Successful, Failed, Problems, Converged
  - Shapes: Circle, Square, Rectangle, …
  - Colors: Red, Blue, Black, Green, …
  - Coordinate system: Cartesian, Polar, Cylindrical
  - Polarization: Transverse, Longitudinal
- Enumerators make the code more user friendly
  - Easier to understand human identifiers instead of hardwired numbers in your code!
- You can redefine the value associated to an identifier w/o changing your code

Example of Enumeration

```
// enum1.cc
#include <iostream>
using namespace std;

int main() {
  enum FitStatus  { Succesful, Failed, Problems, Converged };

  FitStatus status;

  status = Succesful;
  cout << "Status: " << status << endl;

  status = Converged;
  cout << "Status: " << status << endl;

  return 0;
}
```

By default the first identifier is assigned value 0

Don’t forget this one!

```
$ g++ -o /tmp/enum1 enum1.cc
$ /tmp/enum1
Status: 0
Status: 3
```

enums can be used as integers but not vice versa!

Another Example of Enumeration

- You can use arbitrary integer values for each of your identifiers
  - for example use RGB codes for main colours

```
// enum2.cc
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;


  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;


  return 0;
}
```

```
$ g++ -o /tmp/app enum2.cc
$ /tmp/app
Color: 1
Color: 342
```

Common errors with enumuration

```
// enum3.cc
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;

  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;

  col = 45; //assign int to enum

  int i = Red;

  return 0;
}
```

Can’t assign an int to an enum!

But you can assign an enum to an int

```
$ g++ -o /tmp/app enum3.cc
enum3.cc:16:9: error: assigning to 'Color' from incompatible type 'int'
  col = 45; //assign int to enum
        ^~
1 error generated.
```

Enumeration in Classes

- Use complete qualifier including namespace and class to use **public** enumerators

```
#ifndef Fitter_h_
#define Fitter_h_
// Fitter.h
namespace analysis {

  class Fitter {
    public:
      enum Status { Succesful=0,
                    Failed,
                    Problems };

      Fitter() { };

      Status fit() {
        return Succesful;
      }
    private:
  }; // class Fitter
} //namespace
#endif
```

```
//enum4.cc
#include "Fitter.h"
#include <iostream>
using namespace std;

int main() {

  analysis::Fitter myFitter;

  analysis::Fitter::Status stat =
                                myFitter.fit();

  if( stat == analysis::Fitter::Succesful ) {
    cout << "fit succesful!" << endl;
  } else {
    cout << "Fit had problems ... status = "
         << stat << endl;
  }

  return 0;
}
```

```
$ g++ -o /tmp/app enum4.cc
$ /tmp/app
fit succesful!
```

Enumerators and strings

- No automatic conversion from enumeration to strings
- You can use vectors of strings or std::map to assign string names to enumeration states

```
// color.cc

#include <iostream>
#include <map>
using std::cout;
using std::endl;

int main() {
  enum Color  { Red=1, Blue=45,
                Yellow=17, Black=342 };

  Color col;

  // using std::map
  std::map<int,std::string> colname;
  colname[Red] = std::string("Red");
  colname[Black] = std::string("Black");

  col = Red;
 cout << "Color: " << colname[col] << endl;

  return 0;
}
```

```
$ g++ -o /tmp/app color.cc
$ /tmp/app
Color: Red
```

std::map

http://www.cplusplus.com/reference/map/map/

https://en.cppreference.com/w/cpp/container/map

class template

std::map <map>

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

std::pair

https://en.cppreference.com/w/cpp/utility/pair

http://www.cplusplus.com/reference/utility/pair/

std::pair

Defined in header <utility>

```
template<
    class T1,
    class T2
> struct pair;
```

std::pair is a struct template that provides a way to store two heterogeneous objects as a single unit. A pair is a specific case of a std::tuple with two elements.

If std::is_trivially_destructible_v<T1> && std::is_trivially_destructible_v<T2> is true (since C++17), the destructor of pair is trivial.

Template parameters

T1, T2 - the types of the elements that the pair stores.

Application with **map**, **pair**, **vector**

```
// map1.cc
#include<iostream>
#include<vector>
#include<map>
#include <utility>      // std::pair, std::make_pair
#include<string>

#include "Student.h"

int main() {

  // pair object to associate two different types of data
  std::pair< std::string, int> grade = std::make_pair("MQR", 24);

  // grades of a student stored in a vector
  std::vector< std::pair< std::string, int> > grades; //
  grades.push_back( std::make_pair("MQR", 26) );
  grades.push_back( std::make_pair("Phys Lab", 27) );
  grades.push_back( std::make_pair("Cond Matt", 23) );

  Student gino("Gino", 110998);


  // databases of grades of all students
  //    key: student     value: grades
  std::map<Student,  std::vector< std::pair< std::string, int> >  > exams;
  exams[gino] = grades;

  Student tina("Tina", 121001);
  grades.clear(); // delete all previous values in the vector
  grades.push_back( std::make_pair("MQR", 29) );
  grades.push_back( std::make_pair("Phys Lab", 28) );
  grades.push_back( std::make_pair("Cond Matt", 25) );

  exams[tina] = grades;

  // loop over entries in the map
  for(std::map<Student,  std::vector< std::pair< std::string, int> >  >::iterator it = exams.begin(); it != exams.end(); it++ ) {

    // print out student data
    std::cout << "Student name: " << (it->first).name() << "\t id: " << (it->first).id() << std::endl;

    // loop over list of exams
    for(std::vector< std::pair< std::string, int> >::iterator vit = (it->second).begin(); vit != (it->second).end(); vit++) {
      // print name of each exams and relative grade
      std::cout << "\t Subject: " << vit->first << "\t grade: " << vit->second << std::endl;
    } // end: loop over grades

  } // end: loop over students

  return 0;

}
```

```
#ifndef Student_h
#define Student_h

#include<string>

class Student {
public:
  Student(const std::string& name, int id) {
    name_ = name;
    id_ = id;
  }

  bool operator<(const Student& rhs) const {
    return id_ < rhs.id_;
  }

  std::string name() const {
    return name_;
  }

  int id() const {
    return id_;
  }

private:
  std::string name_;
  int id_;
};
#endif
```

```
$ g++ -o /tmp/app map1.cc
$ /tmp/app
Student name: Gino  id: 110998
        Subject: MQR         grade: 26
        Subject: Phys Lab  grade: 27
        Subject: Cond Matt         grade: 23
Student name: Tina  id: 121001
        Subject: MQR         grade: 29
        Subject: Phys Lab  grade: 28
        Subject: Cond Matt         grade: 25
```

Class Vector 3D

- How many and what type of data members?
- How can we handle different coordinate systems?
  - are the classes different?
  - do you need different attributes?
  - is it only a setup problem?
  - How do you distinguish polar vector from cartesian?
  - can you ask phi() and theta() to a cartesian vector?

Lecture 9

27 oct 2020

Object Oriented Programming: Inheritance Polymorphism

Shahram Rahatlou

Computing Methods in Physics

http://www.roma1.infn.it/people/rahatlou/

Anno Accademico 2020/21

SAPIENZA
UNIVERSITÀ DI ROMA

Lecture 9

- Introduction to elements of object oriented programming (OOP)
  - Inheritance
  - Polymorphism

- Base and Derived Classes

- Inheritance as a mean to provide common interface

What is Inheritance?

- Powerful way to reuse software without too much re-writing

- Often several types of object are in fact special cases of a basic object
  - keyboard and files are different types of an input stream
  - screen and file are different types of output stream
  - Resistors and capacitors are different types of circuit elements
  - Circle, square, ellipse are different types of shapes
  - In StarCraft, engineers, builders, soldiers are different types of units

- Inheritance allows to define a “base” class that provides basic functionalities to “derived” classes
  - Derived classes can extend the base class by adding new data members and functions

Inheritance: Student "is a" Person

```
// example1.cpp
#include <string>
#include <iostream>
using namespace std;


class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is "
           << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) :
      Person(name) {
      id_ = id;
      cout << "Student(" << name
           << ", " << id << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for name:"
           << name() << " and id: " << id_
           << endl;
     }

     int id() const { return id_; }

  private:
    int id_;
};
```

Example of Inheritance in Use

```
// example1.cpp

int main() {

  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);

  susan->print();
  cout << "name: " << susan->name() << " id: " << susan->id() << endl;

  delete john;
  delete susan;

  return 0;
}
```

```
$ g++ -o /tmp/app example1.cpp
$ /tmp/app
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am a Person. My name is Susan
name: Susan id: 123456
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

Student "behaves as" Person

```
Person* john = new Person("John");
john->print();

Student* susan = new Student("Susan", 123456);
susan->print();
cout << "name: " << susan->name()
     << " id: " << susan->id()
     << endl;

  delete john;
  delete susan;

  return 0;
}
```

`print()` and `name()` are methods of `Person`

`id()` is a method of `Student`

- Methods of **`Person`** can be called with an object of type **`Student`**
  - Functionalities implemented for Person available for free
  - No need to re-implement the same code over and over again
  - If a functionality changes, we need to fix it just once!

Student is an "extension" of **Person**

```
class Student : public Person {
  public:

     int id() const { return id_; }

  private:
    int id_;
};
```

id() is a method of Student

```
Person* john = new Person("John");
john->print();

Student* susan = new Student("Susan", 123456);
susan->print();
cout << "name: " << susan->name()
     << " id: " << susan->id()
     << endl;

delete john;
delete susan;

return 0;
}
```

- Student provides all functionalities of Person **and more**
- Student has additional data members and member functions
- Student is an extension of Person but not limited to be the same

Typical Error: **Person** is not **Student**!

```
// bad1.cpp

int main() {

  Person* susan = new Student("Susan", 123456);
  cout << "name: " << susan->name() << endl;
  cout << "id: " << susan->id() << endl;

  delete susan;

  return 0;
}
```

susan is a pointer to Person
but initialized by a Student ctor!

OK… because a Student is also a Person!
elements of polymorphism

```
$ g++ -o /tmp/app bad1.cpp
bad1.cpp:53:28: error: no member named 'id' in 'Person'
  cout << "id: " << susan->id() << endl;
                  ~~~~~  ^
1 error generated.
```

- You can not use methods of Student on a Person object
  - Inheritance is a one-way relation
  - Student knows to be derived from Person
  - Person does not know who could be derived from it

- You can treat a Student object (*susan) as a Person object

Public Inheritance

```
class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is "
           << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) :
      Person(name) {
      id_ = id;
      cout << "Student(" << name
           << ", " << id << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for
name:"
           << name() << " and id: " << id_
           << endl;
      }

      int id() const { return id_; }

  private:
    int id_;
};
```

Student can use only public methods and data of Person
like anyone else (public inheritance)

No special access privilege… as usual access can be granted not taken

`public` and `private` in public inheritance

- Student is derived from Person through public inheritance

```
class Student : public Person {
  public:

  private:
};
```

private and protected inheritance are also possible but will not be discussed here

- All public members of Person become public members of Student as well
  - Both data and functions

- Private members of Person **remain** private and not accessible directly by Student
  - Access provided only through public methods (getters)
- You don't need to access source code of a class to inherit from it!
  - *Use public inheritance and add new data members and functions*

protected members

- protected members become protected members of derived classes
  - Protected is somehow between public and private

```
// example2.cpp
class Person {
  public:
    Person(const string& name, int age) {
      name_ = name;
      age_ = age;
      cout << "Person(" << name << ", "
           << age << ") called" << endl;
    }
    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }
    int age() const { return age_; }
    void print() {
      cout << "I am a Person. name: " << name_
           << " age: " << age_ << endl;
    }

  private:
    string name_;

  protected:
    int age_;

};
```

```
class Student : public Person {
  public:
    Student(const string& name, int age,
            int id) :
                        Person(name,age) {
      id_ = id;
      cout << "Student(" << name << ", "
           << age  << ", " << id
           << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for name:"
           << name()
           << " age: " << age_ << " and id: "
           << id_ << endl;
    }

    int id() const { return id_; }

  private:
    int id_;
};
```

protected members can be used by derived classes

Constructors of Derived Classes

- Compiler calls default constructor of base class in constructors of derived class ***unless*** you call explicitly a specific constructor
  - NB: constructors are not inherited by constructed by compiler
- Necessary to insure data members of the base class **always** initialised when creating instance of derived class

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

private:
    int id_;
};
```

Bad Programming!

Constructor of Student does not call constructor of Person

Compiler is forced to call Person() to make sure name_ is intialised correctly

Bad: we rely on default constructor to do the right thing

Common Error with Missing Constructors

```
class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }
    ~Person() {
      cout << "~Person() called for "
          << name_  << endl;
     }

private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

  private:
    int id_;
};
```

```
$ g++ -o /tmp/app bad2.cpp
bad2.cpp:32:5: error: constructor for 'Student' must explicitly initialize
the base class 'Person' which does not
      have a default constructor
    Student(const string& name, int id) {
    ^
bad2.cpp:7:7: note: 'Person' declared here
class Person {
      ^
1 error generated.
```

```
// bad2.cpp

int main() {

  Person anna("Anna");

  Student* susan =
        new Student("Susan", 123456);
  susan->print();
  delete susan;

  return 0;
}
```

No default constructor implemented for Person

Compiler can use a default one to make anna

But gives error dealing with derived classes.

You need to provide a default constructor or call one of the implemented constructors

Bad Working Example

```
class Person {
  public:
    Person() { } // default constructor
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name << ") called"
           << endl;
    }

};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

};
```

```
// bad3.cpp

int main() {

  Student* susan =
       new Student("Susan", 123456);
  susan->print();

  delete susan;

  return 0;
}
```

```
$ g++ -o /tmp/app bad3.cpp
$ /tmp/app
Student(Susan, 123456) called
I am a Person. My name is
~Student() called for name: and id: 123456
~Person() called for
```

- Default constructor is called by compiler
- No name assigned to student by default
  - Ask yourself: why name is not used in the constructor?
- Code compiles, links, and runs but bad behavior

Destructors

- Similar to constructors

- Compiler calls the default destructor of base class in destructor of derived class

- No compilation error if destructor of base class not implemented
  - Default will be used but… bad things can happen!

- Extremely important to implement correctly the destructors to avoid memory leaks!

Member Functions of Derived Classes

- Derived classes can also overload functions provided by the base class
  - Same signature but different implementation

```
class Person {
  public:
    void print() {
      cout << "I am a Person. My name is " << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    void print() {
      cout << "I am Student "
           <<  name()
           << " with id " << id_
           << endl;
    }

  private:
    int id_;
};
```

Overloading Methods from Base Class

```
// example3.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {

  Person* john = new Person("John");
  john->print();  // Person::print()

  Student* susan = new Student("Susan", 123456);
  susan->print(); // Student::print()
  susan->Person::print(); // Person::print()

  Person* p2 = susan;
  p2->print(); // Person::print()

  delete john;
  delete susan;

  return 0;
}
```

Compiler calls the correct version of print() for Person and Student

We can use Person::print() implementation for a Student object by specifying its scope

Remember: a function is uniquely identified by its namespace and class scope

```
$ g++ -o example3 example3.cpp
$ ./example3
Person(John) called
I am a Person. My name is John

Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan

I am a Person. My name is Susan

~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

Undesired limitation

```
// example3.cpp
int main() {

  Person john("John");
  john.print();  // Person::print()

  Student susan("Susan", 123456);
  susan.print(); // calls Student::print()
  susan.Person::print(); // explicitly call Person::print()

  // using base class pointer
  cout << "-- using base class pointer" << endl;
  Person* p2 = &susan;
  p2->print(); // calls Person::print()

  //using derived class pointer
  cout << "-- using derived class pointer" << endl;
  Student* sp = &susan;
  sp->print(); // calls Student::print()

  // using base class reference
  cout << "-- base class reference" << endl;
  Person& p3 = susan;
  p3.print(); // calls Person::print()

  // behavior of print() depends on the type
  // of pointer determined at compilation time

  return 0;
}
```

```
$ g++ -o /tmp/app example4.cpp
$ /tmp/app
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan
-- using base class pointer
I am a Person. My name is Susan
-- using derived class pointer
I am Student Susan with id 123456
-- base class reference
I am a Person. My name is Susan
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

Polymorphism

Polymorphism

- Polymorphism with inheritance hierarchy
- virtual and pure virtual methods
  - When and why use virtual or/and pure virtual functions
- virtual destructors
- Abstract and Pure Abstract classes
  - Providing common interface and behavior

Polymorphism

- Ability to treat objects of an inheritance hierarchy as belonging to the base class
  - Focus on common general aspects of objects instead of specifics

- Polymorphism allows programs to be general and extensible with little or no re-writing
  - resolve **different objects** of same inheritance hierarchy **at runtime**
  - Recall videogame with polymorphic objects Soldier, Engineer, Technician of same base class Unit
  - Can add new 'types' of Unit without rewriting application

- Base class provides interface common to all types in the hierarchy
- Application uses base class and can deal with new types not yet written when writing your application!

Examples of Polymorphism

- Application for graphic rendering
  - Base class **`Shape`** with **`draw()`** and **`move()`** methods
  - Application expects all shapes to have such functionality

- **`Function`** in Physics
  - We'll study this example in detail
  - **`Guassian`**, **`Breit-Wigner`**, **`polynomials`**, **`exponential`** are all functions
  - A **`Function`** must have
    - **`value(x)`**
    - **`integral(x1,x2)`**
    - **`primitive(x)`**
    - **`derivative(x)`**
  - Can write a fit application that can handle existing or not-yet implemented functions using a base class Function

Reminders about Inheritance

- Inheritance is a is-a relationship
  - Object of derived class ‘is a’ base class object as well

- Can treat a derived class object as a base class object
  - call methods of base class on derived class
  - can point to derived class object with pointer of type base class

- Base class does not know about its derived classes
  - Can not treat a base class object as a derived object

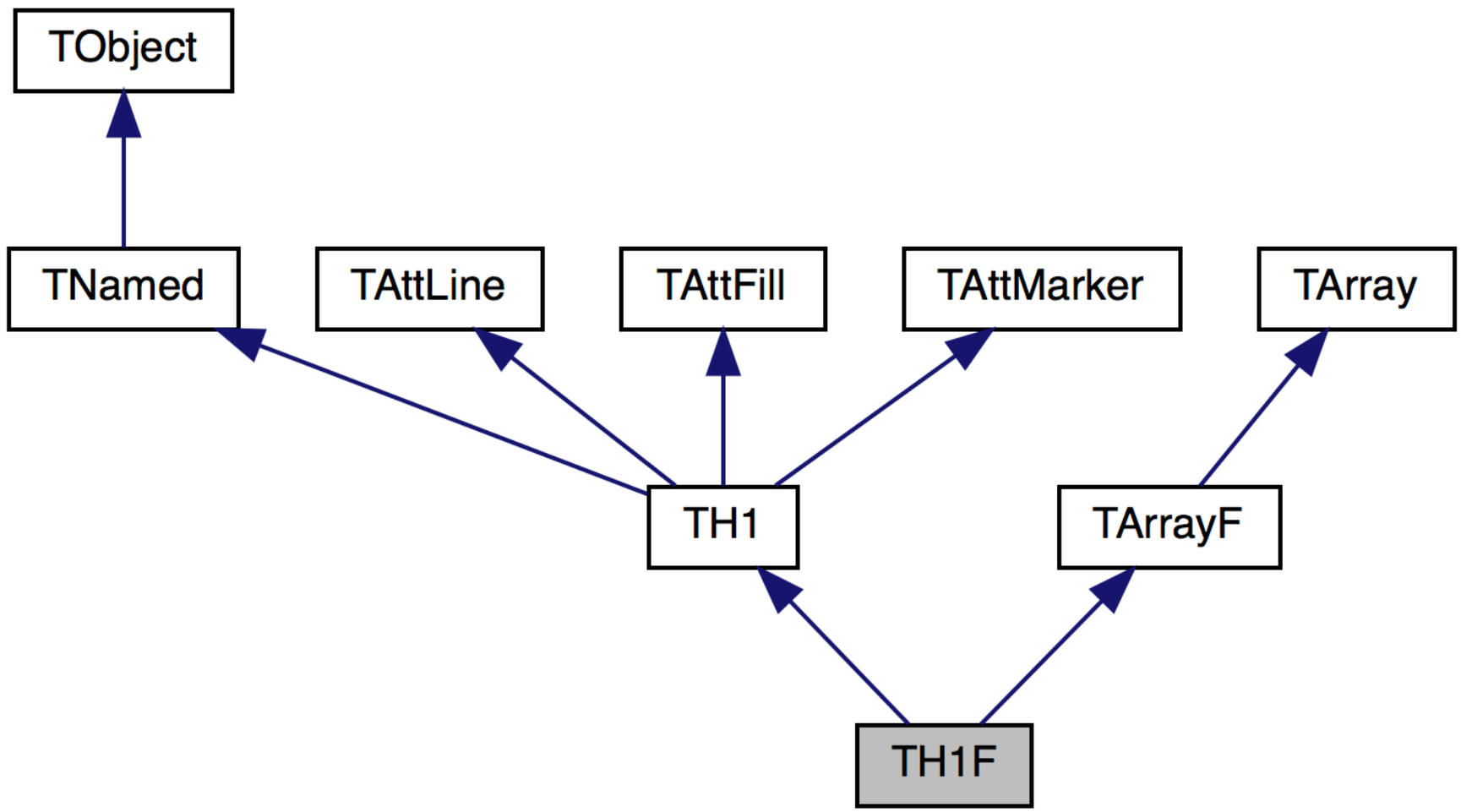
- Methods of base class can be redefined in derived classes
  - Same interface but different implementation for different types of object in the same hierarchy

Example from ROOT: TH1F

https://root.cern.ch/doc/master/classTH1F.html

```
#include <TH1.h>
```

Inheritance diagram for TH1F:



TObject and TNamed

ROOT » CORE » BASE » TObject

https://root.cern.ch/root/html526/TObject.html

class TObject

TObject

Mother of all ROOT objects.

The TObject class provides default behaviour and protocol for all objects in the ROOT system. It provides protocol for object I/O, error handling, sorting, inspection, printing, drawing, etc. Every object which inherits from TObject can be stored in the ROOT collection classes.

ROOT » CORE » BASE » TNamed

https://root.cern.ch/root/html526/TNamed.html

class TNamed: public TObject

TNamed

The TNamed class is the base class for all named ROOT classes
A TNamed contains the essential elements (name, title)
to identify a derived object in containers, directories and files.
Most member functions defined in this base class are in general
overridden by the derived classes.

TH1

https://root.cern.ch/root/html526/TH1.html

class TH1: public TNamed, public TAttLine, public TAttFill, public TAttMarker

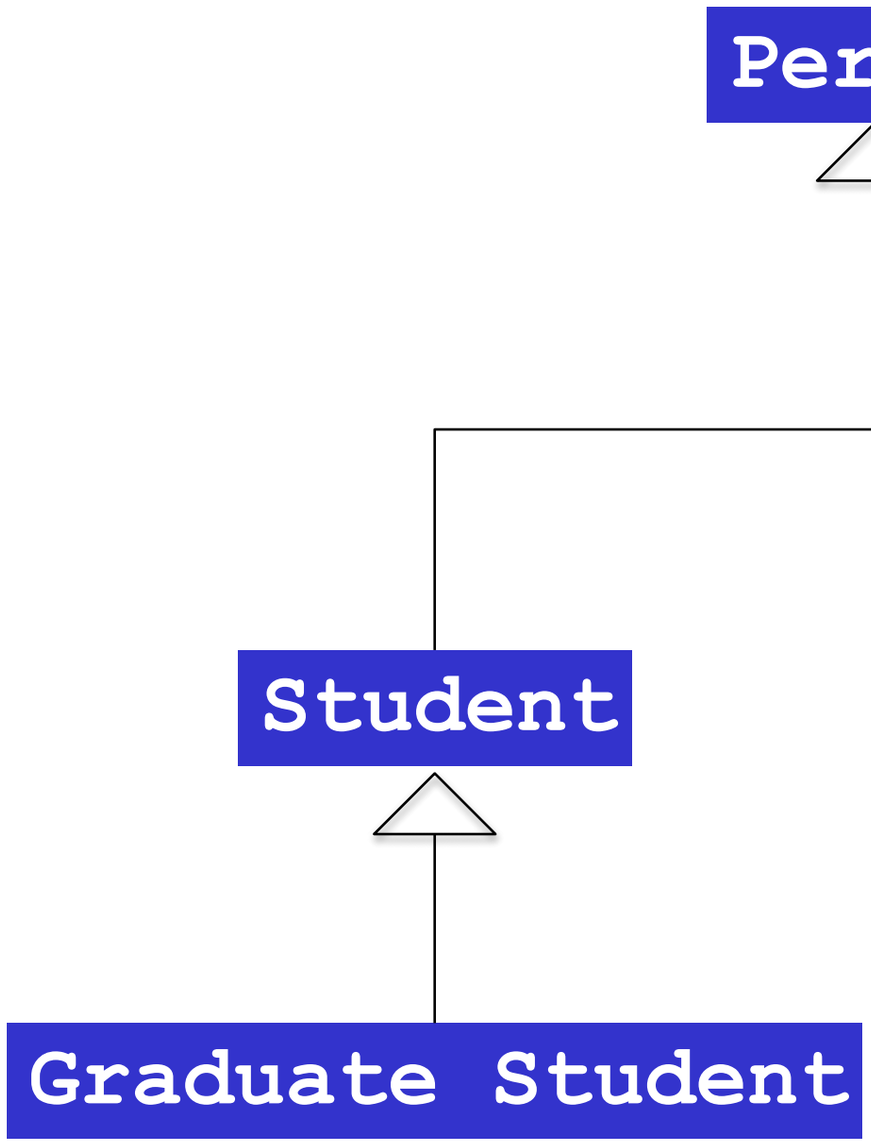
The Histogram classes

ROOT supports the following histogram types:

- 1-D histograms:
  - TH1C : histograms with one byte per channel. Maximum bin content = 127
  - TH1S : histograms with one short per channel. Maximum bin content = 32767
  - TH1I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH1F : histograms with one float per channel. Maximum precision 7 digits
  - TH1D : histograms with one double per channel. Maximum precision 14 digits
- 2-D histograms:
  - TH2C : histograms with one byte per channel. Maximum bin content = 127
  - TH2S : histograms with one short per channel. Maximum bin content = 32767
  - TH2I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH2F : histograms with one float per channel. Maximum precision 7 digits
  - TH2D : histograms with one double per channel. Maximum precision 14 digits
- 3-D histograms:
  - TH3C : histograms with one byte per channel. Maximum bin content = 127
  - TH3S : histograms with one short per channel. Maximum bin content = 32767
  - TH3I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH3F : histograms with one float per channel. Maximum precision 7 digits
  - TH3D : histograms with one double per channel. Maximum precision 14 digits
- Profile histograms: See classes TProfile, TProfile2D and TProfile3D. Profile histograms are used to display the mean value of Y and its RMS for each bin in X. Profile histograms are in many cases an elegant replacement of two-dimensional histograms : the inter-relation of two measured quantities X and Y can always be visualized by a two-dimensional histogram or scatter-plot; If Y is an unknown (but single-valued) approximate function of X, this function is displayed by a profile histogram with much better precision than by a scatter-plot.

All histogram classes are derived from the base class TH1

`Person` Inheritance Hierarchy



Student and GraduateStudent

```
class Person {
  public:
    Person(const std::string& name);
    ~Person();
    std::string name() const { return name_; }
    void print() const;

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

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    int id() const { return id_; }
    void print() const;

  private:
    int id_;
};
```

```
class GraduateStudent : public Student {
  public:
    GraduateStudent(const std::string& name, int id,
                    const std::string& major);
    ~GraduateStudent();
    std::string getMajor() const { return major_; }
    void print() const;

  private:
    std::string major_;
};
```

Proper Constructor

```
// Person.cc

Person::Person(const std::string& name) {
  name_ = name;
  cout << "Person(" << name << ") called" << endl;
}
```

```
// Student.cc

Student::Student(const std::string& name, int id) : Person(name) {
  id_ = id;
  cout << "Student(" << name << ", " << id << ") called" << endl;
}
```

```
// GraduateStudent.cc

GraduateStudent::GraduateStudent(const std::string& name, int id, const std::string& major) :
  Student(name,id) {
  major_ = major;
  cout << "GraduateStudent(" << name << ", " << id << "," << major << ") called" << endl;
}
```

- **`Person::Person(name)`** assigns value to name_
- **`Student(name,id)`** assigns id_ and calls **`Person::Person(name)`**
- **`GraduateStudent(name,id,major)`** assigns major, calls **`Student::Student(name,id)`**, which calls **`Person::Person(name)`**

Example

```
// example11.cpp

int main() {

  Person* john = new Person("John");
  john->print();  // Person::print()

  Student* susan = new Student("Susan", 123456);
  susan->print(); // Student::print()
  susan->Person::print(); // Person::print()

  Person* p2 = susan;
  p2->print(); // Person::print()

  GraduateStudent* paolo =
    new GraduateStudent("Paolo", 9856, "Physics");
  paolo->print();

  Person* p3 = paolo;
  p3->print();


  delete john;
  delete susan;

  return 0;
}
```

Can point to **`Student`** or **`GraduateStudent`** object with a pointer of type **`Person`**

Can treat **`paolo`** and **`susan`** as **`Person`**

Depending on the pointer type different **`print()`** methods are called

No delete for **`paolo`**!!
Memory Leak!

```
$ g++  -o /tmp/app example11.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan
I am Student Susan with id 123456
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

Problem with Previous Example

```
// example11.cpp

int main() {

  Person* john = new Person("John");
  john->print();  // Person::print()

  Student* susan = new Student("Susan", 123456);
  susan->print(); // Student::print()
  susan->Person::print(); // Person::print()

  Person* p2 = susan;
  p2->print(); // Person::print()

  GraduateStudent* paolo =
    new GraduateStudent("Paolo", 9856, "Physics");
  paolo->print();

  Person* p3 = paolo;
  p3->print();


  delete john;
  delete susan;

  return 0;
}
```

```
$ g++  -o /tmp/app example11.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan
I am Student Susan with id 123456
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

- Call to method **`print()`** is resolved base on the type of the pointer
  - **`print()`** methods is determined by pointer not the actual type of object
- Desired feature: use generic **`Person*`** pointer but call appropriate **`print()`** method for **`paolo`** and **`susan`** based on **actual type** of these objects

Desired Feature: Resolve Different Objects at Runtime

- We would like to use the same **`Person*`** pointer but call different methods based on the type of the object being pointed to
- We **do not** want to use the scope operator to specify the function to call

```
Person* john = new Person("John");
john->print();  // Person::print()

Student* susan = new Student("Susan", 123456);
Person* p2 = susan;
p2->print(); // Person::print()

GraduateStudent* paolo =
  new GraduateStudent("Paolo", 9856, "Physics");

Person* p3 = paolo;
p3->print();
```

Same **`Person*`** pointer used for three different types of object in the same hierarchy

Same code used by types solved at runtime

```
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456

Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
```

Polymorphic Behavior

```
// example12.cpp
int main() {

 vector<Person*> people;

  Person* john = new Person("John");
  people.push_back(john);

  Student* susan = new Student("Susan", 123456);
  people.push_back(susan);

  GraduateStudent* paolo = new GraduateStudent("Paolo", 9856, "Physics");
  people.push_back(paolo);

  for(int i=0;
     i< people.size(); ++i) {
    people[i]->print();
  }

  delete john;
  delete susan;
  delete paolo;

  return 0;
}
```

vector of generic type Person
No knowledge about specific types

Different derived objects stored in the vector of Person

Generic call to print()

Different functions called based on the real type of objects pointed to!!

How? virtual functions!

```
$ g++ -o /tmp/app example12.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~GraduateStudent() called for name:Paolo id: 9856 major: Physics
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
```

`virtual` functions

```
class Person {
  public:
    Person(const std::string& name);
    ~Person();
    std::string name() const { return name_; }
    virtual void print() const;

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

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    int id() const { return id_; }
    virtual void print() const;

  private:
    int id_;
};
```

- Virtual methods of base class are **overridden not redefined** by derived classes
  - if not overridden, base class function called
- Type of objects pointed to determine which function is called
- Type of pointer (also called handle) has no effect on the method being executed
- **`virtual`** allows polymorphic behavior and generic code without relying on specific objects

```
class GraduateStudent : public Student {
  public:
    GraduateStudent(const std::string& name, int id, const std::string& major);
    ~GraduateStudent();
    std::string getMajor() const { return major_; }
    virtual void print() const;

  private:
    std::string major_;
};
```

Static and Dynamic (or late) binding

- Choosing the correct derived class function at run time based on the type of the object being pointed to, regardless of the pointer type, is called dynamic binding or late binding

- Dynamic binding works only with pointers and references not using dot-member operators
  - static binding: function calls resolved at compile time

```
// example13.cpp

int main() {

  Person john("John");
  Student susan("Susan", 123456);
  GraduateStudent paolo("Paolo",
      9856, "Physics");

  john.print();
  susan.print();
  paolo.print();

  return 0;
}
```

static binding

```
$ g++ -o /tmp/app example13.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~GraduateStudent() called for name:Paolo id: 9856 major: Physics
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

Example of Dynamic Binding

```
// example14.cpp

  Person* john = new Person("John");
  Person* susan = new Student("Susan", 123456);
  Person* paolo = new GraduateStudent("Paolo", 9856, "Physics");

  (*john).print();
  (*susan).print();
  (*paolo).print();

  john->print();
  susan->print();
  paolo->print();
```

```
$ g++ -o /tmp/app example14.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Person() called for Susan
~Person() called for Paolo
```

Example: **`virtual`** Function at Runtime

```
int main() {

  Person* p = 0;
  int value = 0;
  while(value<1 || value>10) {
    cout << "Give me a number [1,10]: ";
    cin >> value;
  }
  cout << flush; // write buffer to output
  cout << "make a new derived object..." << endl;
  if(value>5) p = new Student("Susan", 123456);
  else        p = new GraduateStudent("Paolo", 9856, "Physics");

  cout << "call print() method ..." << endl;

  p->print();

  delete p;
  return 0;
}
```

Type of object decided at runtime by user

Compiler does not know what object will be used

```
$ g++ -o /tmp/app example15.cpp {Person,Student,GraduateStudent,Professor}.cc
$ /tmp/app
Give me a number [1,10]: 6
make a new derived object...
Person(Susan) called
Student(Susan, 123456) called
call print() method ...
I am Student Susan with id 123456
~Person() called for Susan

$ /tmp/app
Give me a number [1,10]: 2
make a new derived object...
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
call print() method ...
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for Paolo
```

Virtual methods allow dynamic binding at runtime

Default for Virtual Methods

```
// Professor.h
class Professor : public Person {
  public:
    Professor(const std::string& name,
              const std::string& department);
    ~Professor();
    std::string department() const { return department_; }
    //virtual void print() const; // will use Person::Print()

  private:
    std::string department_;
};
```

`print()` not overriden in `Professor`

```
// example16.cpp

int main() {

  Person john("John");
  Student susan("Susan", 123456);
  GraduateStudent
   paolo("Paolo", 9856, "Physics");
  Professor
      bob("Robert", "Biology");

  john.print();
  susan.print();
  paolo.print();
  bob.print();

  return 0;
}
```

```
$ g++ -o /tmp/app example16.cpp {Person,Student,GraduateStudent,Professor}.cc
$ /tmp/app
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
Person(Robert) called
Professor(Robert, Biology) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is Robert
~Professor() called for name:Robert and department: Biology
~Person() called for Robert
~GraduateStudent() called for name:Paolo id: 9856 major: Physics
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

`Person::print()` used by default

Abstract Class

Pure virtual Functions

- virtual functions with no implementation
  - All derived classes ***are required*** to implement these functions
- Typically used for functions that can't be implemented (or at least in an unambiguous way) in the base case
- Class with at least one pure virtual method is called an "Abstract" class

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;

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

= 0 is called pure specifier

```
#include "Function.h"

Function::Function(const std::string& name) {
  name_ = name;
}
```

ConstantFunction

```
#ifndef ConstantFunction_h
#define ConstantFunction_h

#include <string>
#include "Function.h"

class ConstantFunction : public Function {
  public:
    ConstantFunction(const std::string& name, double value);
    virtual double value(double x) const;
    virtual double integrate(double x1, double x2) const;

  private:
    double value_;
};
```

```
#include "ConstantFunction.h"

ConstantFunction::ConstantFunction(const std::string& name, double value) :
  Function(name) {
  value_ = value;
}

double ConstantFunction::value(double x) const {
  return value_;
}

double ConstantFunction::integrate(double x1, double x2) const {
  return (x2-x1)*value_;
}
```

Typical Error with Abstract Class

```
// func1.cpp
#include <string>
#include <iostream>
using namespace std;

#include "Function.h"

int main() {

  Function* gauss  = new Function("Gauss");

  return 0;
}
```

Cannot make an object of an Abstract class!

Pure virtual methods not implemented and the class is effectively incomplete

```
$ g++ -o /tmp/app func1.cpp Function.cc
func1.cpp:10:22: error: allocating an object of abstract class type 'Function'
  Function* gauss  = Function("Gauss");
                     ^
./Function.h:9:20: note: unimplemented pure virtual method 'value' in 'Function'
    virtual double value(double x) const = 0;
                   ^
./Function.h:10:20: note: unimplemented pure virtual method 'integrate' in 'Function'
    virtual double integrate(double x1, double x2) const = 0;
                   ^
1 error generated.
```

`virtual` and pure `virtual`

- No default implementation for pure virtual
  - Requires explicit implementation in derived classes

- Use pure virtual when
  - Need to enforce policy for derived classes
  - Need to guarantee public interface for all derived classes
  - You expect to have certain functionalities but too early to provide default implementation in base class
  - Default implementation can lead to error
    - User forgets to implement correctly a virtual function
    - Default implementation is used in a meaningless way

- Virtual allows polymorphism

- Pure virtual forces derived classes to ensure correct implementation

Abstract and Concrete Classes

- Any class with at least one pure virtual method is called an Abstract Class
  - Abstract classes are incomplete
    - At least one method not implemented
    - Compiler has no way to determine the correct size of an incomplete type
  - ***Cannot instantiate an object of Abstract class***

- Usually abstract classes are used in higher levels of hierarchy
  - Focus on defining policies and interface
  - Leave implementation to lower level of hierarchy

- Abstract classes used typically as pointers or references to achieve polymorphism
  - Point to objects of sub-classes via pointer to abstract class

Example of Bad Use of `virtual`

```
class BadFunction {
  public:
    BadFunction(const std::string& name);
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

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

Default dummy implementation

```
class Gauss : public BadFunction {
  public:
    Gauss(const std::string& name, double mean, double width);

    virtual double value(double x) const;
    //virtual double integrate(double x1, double x2) const;

  private:
    double mean_;
    double width_;
};
```

Implement correctly `value()` but use default `integrate()`

We can use ill-defined `BadFunction` and wrongly use `Gauss`!

```
// func2
int main() {

  BadFunction f1  = BadFunction("bad");
  Gauss g1("g1",0.,1.);
  cout << "g1.value(2.): " << g1.value(2.) << endl;
  cout << "g1.integrate(0.,1000.): "
       << g1.integrate(0.,1000.) << endl;
  return 0;
}
```

```
$ g++ -o /tmp/app func2.cpp
        {BadFunction,Gauss,Function}.cc
$ /tmp/app
g1.value(2.): 0.0540047
g1.integrate(0.,1000.): 0
```

Function and BadFunction

```
class BadFunction {
  public:
    BadFunction(const std::string& name);
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

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

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;

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

```
// func3.cpp
int main() {

  BadFunction f1  = BadFunction("bad");
  Function f2("f2");

  return 0;
}
```

Cannot instantiate Function because abstract

Bad Function can be used

```
$ g++ -o /tmp/app func3.cpp {BadFunction,Function}.cc
func3.cpp:13:12: error: variable type 'Function' is an abstract class
  Function f2("f2");
           ^
./Function.h:9:20: note: unimplemented pure virtual method 'value' in 'Function'
    virtual double value(double x) const = 0;
                   ^
./Function.h:10:20: note: unimplemented pure virtual method 'integrate' in 'Function'
    virtual double integrate(double x1, double x2) const = 0;
                   ^
1 error generated.
```

Use of **`virtual`** in Abstract Class **`Function`**

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;
    virtual void print() const;
    virtual std::string name() const { return name_; }

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

```
#include "Function.h"
#include <iostream>

Function::Function(const std::string& name) {
  name_ = name;
}

void
Function::print() const {
  std::cout << "Function with name "
            << name_ << std::endl;
}
```

Default implementation of name()

Unambiguous functionality: user will always want the name of the particular object regardless of its particular subclass

print() can be overriden in sub-classes to provide more details about sub-class but still a function with a name

Concrete Class **Gauss**

```
#include "Gauss.h"
#include <cmath>
#include <iostream>
using std::cout;
using std::endl;

Gauss::Gauss(const std::string& name,
             double mean, double width) :
  Function(name) {
  mean_ = mean;
  width_ = width;
}

double Gauss::value(double x) const {
  double pull = (x-mean_)/width_;
  double y = (1/sqrt(2.*3.14*width_)) * exp(-pull*pull/2.);
  return y;
}

double Gauss::integrate(double x1, double x2) const {
  cout << "Sorry. Gauss::integrate(x1,x2) not implemented yet..."
       << "returning 0. for now..." << endl;
  return 0;
}

void
Gauss::print() const {
  cout << "Gaussian with name: " << name()
       << " mean: " << mean_
       << " width: " << width_
       << endl;
}
```

```
#ifndef Gauss_h
#define Gauss_h

#include <string>
#include "Function.h"

class Gauss : public Function {
  public:
    Gauss(const std::string& name,
     double mean, double width);

    virtual double value(double x) const;
    virtual double integrate(double x1,
                             double x2) const;
    virtual void print() const;

  private:
    double mean_;
    double width_;
};
#endif
```

```
int main() {

  Function* g1  = new Gauss("gauss",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  return 0;
}
```

```
$ g++ -o /tmp/app func4.cpp {Gauss,Function}.cc
$ /tmp/app
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
```

Problem with destructors

- We now properly delete the Gauss object

```
// func5.cpp
int main() {

  Function* g1  = new Gauss("gauss",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  delete g1;

  return 0;
}
```

```
$ g++ -o /tmp/app func5.cpp {Gauss,Function}.cc
$ g++ -o /tmp/app func5.cpp {Gauss,Function}.cc
func5.cpp:15:3: warning: delete called on 'Function' that is abstract but has non-virtual destructor
      [-Wdelete-non-virtual-dtor]
  delete g1;
  ^
1 warning generated.
$ /tmp/app
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
Illegal instruction
```

- In general with polymorphism and inheritance it is a VERY GOOD idea to use virtual destructors
- Particularly important when using dynamically allocated objects in constructors of polymorphic objects

Revisit Person and Student

```
// example7.cpp
int main() {

 Person* p1  = new Student("Susan", 123456);
 Person* p2  = new GraduateStudent("Paolo", 9856, "Physics");


  delete p1;
  delete p2;

  return 0;
}
```

```
$ g++ -o /tmp/app example7.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
~Person() called for Susan
~Person() called for Paolo
```

```
Person::~Person() {
  cout << "~Person() called for " << name_ << endl;
}
```

```
Student::~Student() {
  cout << "~Student() called for name:" << name()
<< " and id: " << id_ << endl;
}
```

```
GraduateStudent::~GraduateStudent() {
  cout << "~GraduateStudent() called for name:" << name()
       << " id: " << id()
       << " major: " << major_ << endl;
}
```

Note that ~Person() is called and not that of the sub class!

We did not declare the destructor to be virtual

destructor called based on the pointer and not the object! Non-polymorphic behaviour

virtual destructors

- Derived classes might allocate dynamically memory
  - Derived-class destructor (if correctly written!) will take care of cleaning up memory upon destruction

- Base-class destructor will not do the proper job if called for a sub-class object

- Declaring destructor to be virtual is a simple solution to prevent memory leak using polymorphism

- virtual destructors ensure that memory leaks don’t occur when delete an object via base-class pointer

Simple Example of **virtual** Destructor

```
// noVirtualDtor.cc
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x) {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  ~Base() {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x) : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  ~Derived() {
    cout << "~Derived() called" << endl;
  }
};

int main() {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Destructor Not virtual

```
$ g++ -Wall -o /tmp/noVirtualDtor noVirtualDtor.cc
$ /tmp/noVirtualDtor
Base(1.2) called
Derived(1.2) called
~Base() called
```

```
// virtualDtor.cc
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x) {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  virtual ~Base() {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x) : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  virtual ~Derived() {
    cout << "~Derived() called" << endl;
  }
};

int main() {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Virtual Destructor

```
$ g++ -Wall -o /tmp/VirtualDtor VirtualDtor.cc
$ /tmp/VirtualDtor
Base(1.2) called
Derived(1.2) called
~Derived() called
~Base() called
```

Revised Class **Student**

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    void addCourse(const std::string& course);
    virtual void print() const;

    int id() const { return id_; }
    const std::vector<std::string>* getCourses() const;
    void printCourses() const;

  private:
    int id_;
    std::vector<std::string>* courses_;
};
```

```
void Student::addCourse(const std::string&
course) {
  courses_->push_back( course );
}

void
Student::printCourses() const {
  cout << "student " << name()
       << " currently enrolled in following
courses:"
       << endl;

  for(int i=0; i<courses_->size(); ++i) {
    cout << (*courses_)[i] << endl;
  }
}

const std::vector<std::string>*
Student::getCourses() const {
  return courses_;
}
```

```
Student::Student(const std::string& name,
int id) :
  Person(name) {
  id_ = id;
  courses_ = new
std::vector<std::string>();
  cout << "Student(" << name << ", " << id
<< ") called"
       << endl;
}

Student::~Student() {
  delete courses_;
  courses_ = 0; // null pointer
  cout << "~Student() called for name:" <<
name()
       << " and id: " << id_ << endl;
}

void Student::print() const {
  cout << "I am Student " << name()
       << " with id " << id_ << endl;
  cout << "I am now enrolled in "
       << courses_->size() << " courses."
<< endl;
}
```

Example of Memory Leak with **Student**

```
// example8.cpp

int main() {

  Student* p1  = new Student("Susan", 123456);
  p1->addCourse(string("algebra"));
  p1->addCourse(string("physics"));
  p1->addCourse(string("Art"));
  p1->printCourses();

  Student* paolo   = new Student("Paolo", 9856);
  paolo->addCourse("Music");
  paolo->addCourse("Chemistry");

  Person* p2 = paolo;

  p1->print();
  p2->print();

  delete p1;
  delete p2;

  return 0;
}
```

Memory leak when deleting paolo because nobody deletes courses_

Need to extend polymorphism also to destructors to ensure that object type not pointer determine correct destructor to be called

```
$ g++ -o /tmp/app example8.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
I am Student Susan with id 123456
I am now enrolled in 3 courses.
I am Student Paolo with id 9856
I am now enrolled in 2 courses.
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for Paolo
```

virtual Destructor for **Person** and **Student**

```
class Person {
  public:
    Person(const std::string& name);
    virtual ~Person();
    std::string name() const { return name_; }
    virtual void print() const;

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

Correct destructor is called using the base-class pointer to Student

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    virtual ~Student();
    void addCourse(const std::string& course);
    virtual void print() const;

    int id() const { return id_; }
    const std::vector<std::string>* getCourses() const;
    void printCourses() const;

  private:
    int id_;
    std::vector<std::string>* courses_;
};
```

```
// example9.cpp

int main() {

  Student* p1  = new Student("Susan", 123456);
  p1->addCourse(string("algebra"));
  p1->addCourse(string("physics"));
  p1->addCourse(string("Art"));
  p1->printCourses();

  Student* paolo   = new Student("Paolo", 9856);
  paolo->addCourse("Music");
  paolo->addCourse("Chemistry");
  Person* p2 = paolo;

  delete p1;
  delete p2;

  return 0;
}
```

```
$ g++ -o /tmp/app example9.cpp
                        {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
```