Introduction to C++

andrea.rizzi@unipi.it

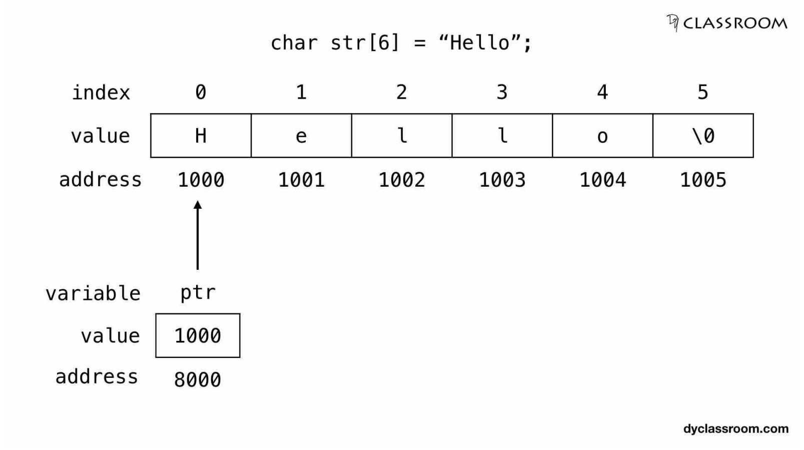
1. Reminder of C concepts and intro to C++
    a. Hello world, basic gcc compile,libraries, linking
    b. Memory handling, pointers and references
    c. Classes, polymorphism, function overload
    d. Templates meta programming, traits, specific templates
2. Exercise with Classes and Objects
    a. Basic gcc compile,libraries, linking
    b. E.g 4-vector, particle, decay chain
    c. Example templates (make the internal representation, traits)
3. C++ standards and C++ nightmares
    a. STL: Standard algorithms, containers, iterators
    b. string handling, file parsing, object persistence
    c. Singletons, strict type checking, smart pointers, thread-safety
4. C++11/C++14/C++17 and Vectorization
    a. auto, lambda, variadic template rvalue refs, move semantic.
    b. Data structures: vector<object> vs object<vector> and vectorization of operations
5. Introduction to ROOT
    a. Classi e funzioni di root, IO, memory handling, basic plotting
    b. ROOT C++: persistenza, dizionari,
6. Advanced exercise
    a. Root exercise
    b. Profiling
7. Computing infrastructure in HEP: scientific Grid and cloud computing
8. Numerical optimizations
    a. Roofit, python minimizers
9. Advanced HEP analysis tools
    a. ROOT: RDataFrame, TMVA
    b. Python: PyRoot, uproot, root-numpy

Why C++ in particle physics ?

- Python is good for analysis,how about other tasks?
    - Data Acquisition (DAQ): synchronous processing, interaction with custom electronics (e.g. with direct memory access), state machines, development of firmware for FPGA or other programmable electronics
    - Simulation: quick and efficient simulation of passage of particles through matter, creation of secondary particles, showers, ionization, etc…
    - Reconstruction: convert detector readout information into higher level processed information on particle trajectories, energy deposits, decay vertices, etc…
    - Trigger: quick decisions on events to read and save based on partial information with limited latency
- C++ remains de-facto a standard for many of the above applications
    - Allow to write fast code
    - Allow access to low level hardware features
    - High level of abstraction to implement complex algorithms in modular ways

What we assume your are familiar with

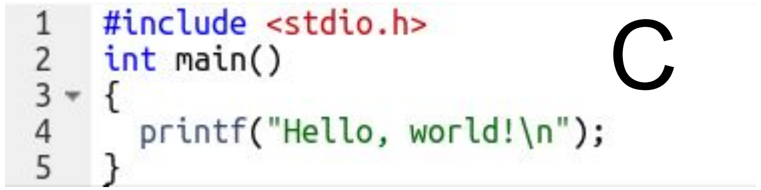
- Basic C programming
  - Builtin types (int, float, long, char, double, void)
  - Variable declaration and initialization
  - Function declaration " int f(); " and definition "int f() {return 2;}"
  - Concept of pointer
  - If, else, for , while, operators, {...} blocks , etc..



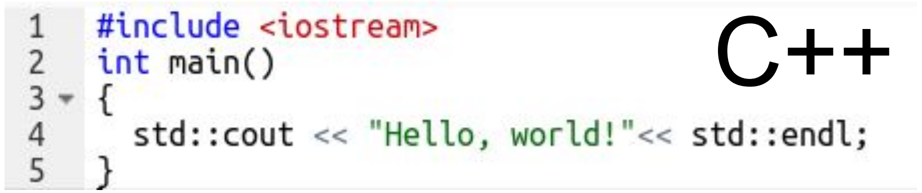
- High level programming concepts discussed in the first part of the course (with python examples)
  - Containers, standard algorithms
- If you have basic C++ skills, the hope is to answer many of the questions you may have in your mind or explain why things are done the way they are (beyond the "I cut and paste it from an example")

An example C++ program

- We can test some of the simple programs in online c++ enviroments (similar to colab/jupyter but for c++), e.g. http://cpp.sh
- For the next lecture/exercise we instead need a linux shell and gcc
  - Do you have the course VirtualBox image installed or a linux installation available?

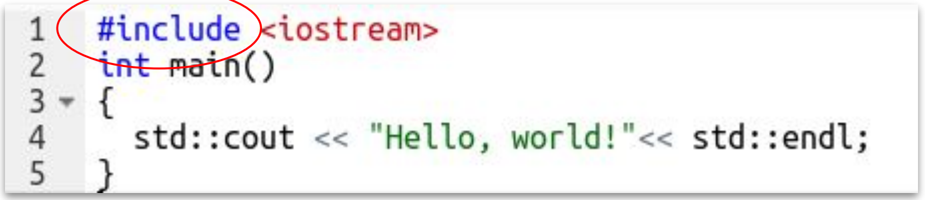


```c
#include <stdio.h>
int main()
{
  printf("Hello, world!\n");
}
```

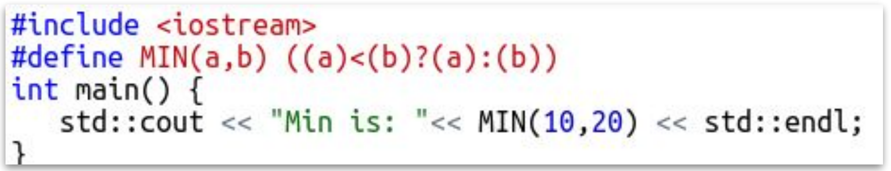


```cpp
#include <iostream>
int main()
{
  std::cout << "Hello, world!"<< std::endl;
}
```

Preprocessor directives

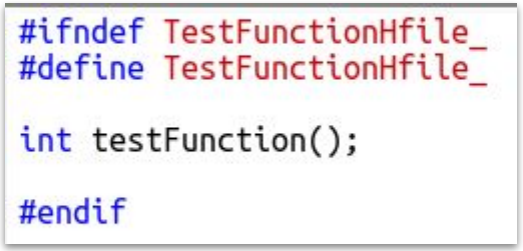


```
#include <iostream>
int main()
{
    std::cout << "Hello, world!"<< std::endl;
}
```



```
#include <iostream>
#define MIN(a,b) ((a)<(b)?(a):(b))
int main() {
    std::cout << "Min is: "<< MIN(10,20) << std::endl;
}
```

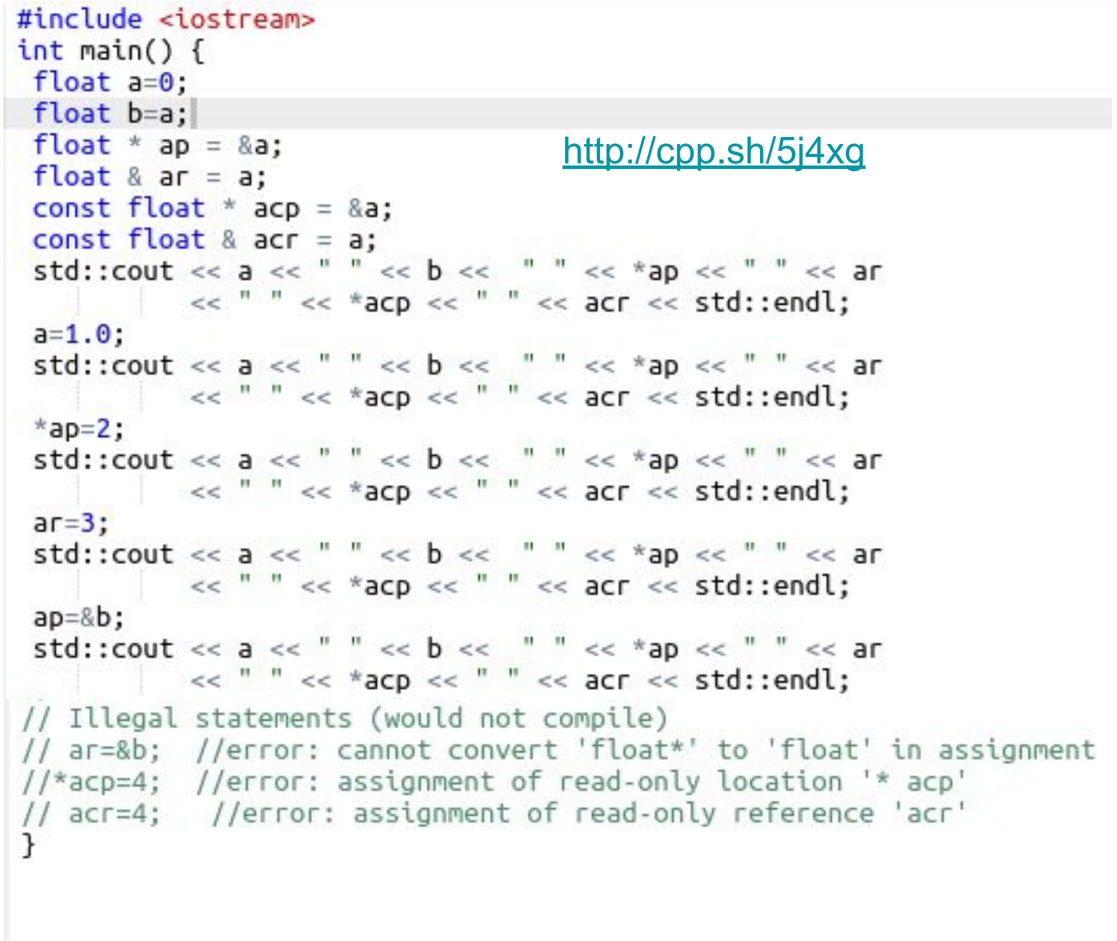
- Preprocessor directive starts with **#**
- The "preprocessor" is run on the code before the actual compiler
  - It replaces the effective source code seen by the compiler
  - It is useful to combine pieces of C++ code or to switch on/off the compilation of some pieces of code
  - It also allow some very simple "meta programming"
- Often used to avoid including multiple time the same file



```
#ifndef TestFunctionHfile_
#define TestFunctionHfile_

int testFunction();

#endif
```

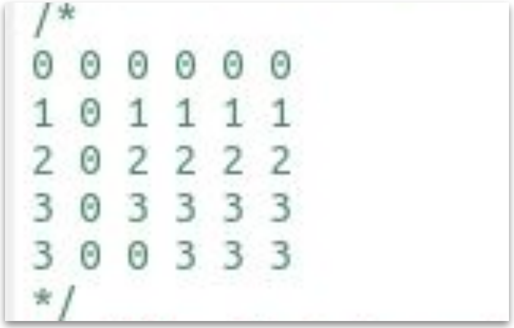
1. `#include`
2. `#define`
3. `#undef`
4. `#ifdef`
5. `#ifndef`
6. `#if`
7. `#else`
8. `#elif`
9. `#endif`
10. `#error`
11. `#pragma`

Reminder on variables, pointers, referencese



```
#include <iostream>
int main() {
 float a=0;
 float b=a;
 float * ap = &a;
 float & ar = a;
 const float * acp = &a;
 const float & acr = a;
 std::cout << a << " " << b <<  " " << *ap << " " << ar
           << " " << *acp << " " << acr << std::endl;
 a=1.0;
 std::cout << a << " " << b <<  " " << *ap << " " << ar
           << " " << *acp << " " << acr << std::endl;
 *ap=2;
 std::cout << a << " " << b <<  " " << *ap << " " << ar
           << " " << *acp << " " << acr << std::endl;
 ar=3;
 std::cout << a << " " << b <<  " " << *ap << " " << ar
           << " " << *acp << " " << acr << std::endl;
 ap=&b;
 std::cout << a << " " << b <<  " " << *ap << " " << ar
           << " " << *acp << " " << acr << std::endl;
// Illegal statements (would not compile)
// ar=&b;  //error: cannot convert 'float*' to 'float' in assignment
//*acp=4;  //error: assignment of read-only location '* acp'
// acr=4;   //error: assignment of read-only reference 'acr'
}
```

http://cpp.sh/5j4xq

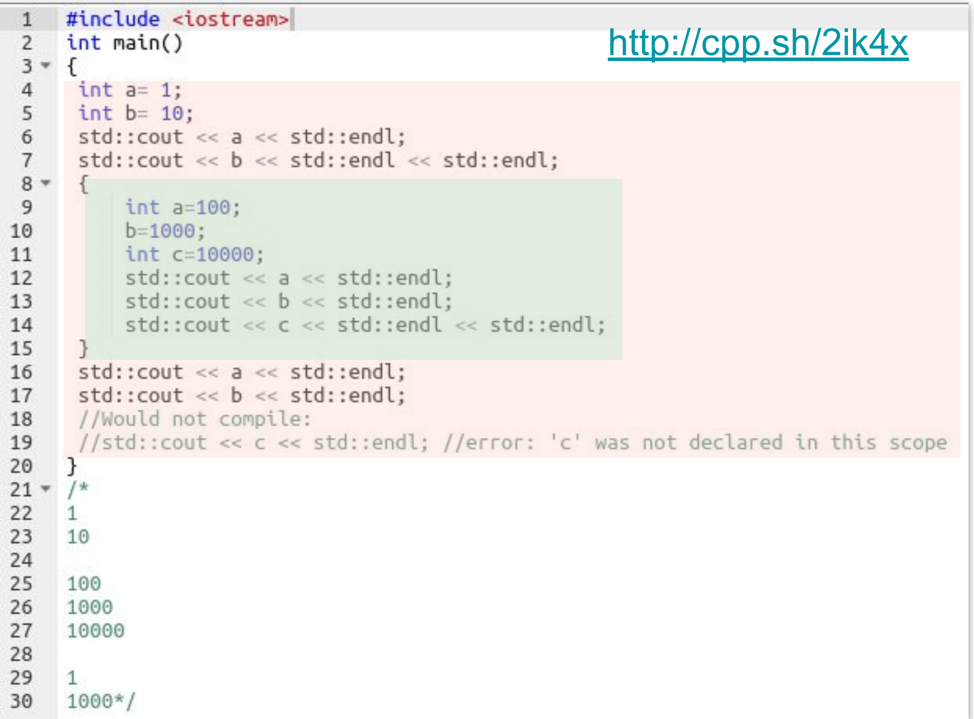


```
/*
0 0 0 0 0 0
1 0 1 1 1 1
2 0 2 2 2 2
3 0 3 3 3 3
3 0 0 3 3 3
*/
```

- Pointers and references **know the location in memory** where the data is, rather than the value
- Reference are used with a syntax similar to the variable

Variable scope. Namespaces

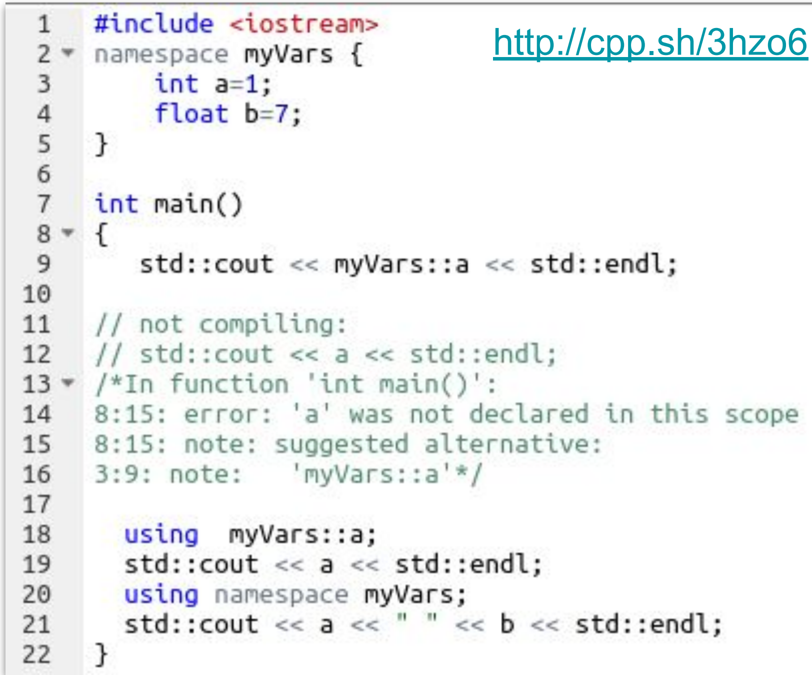
- Variables do not exist outside the scope they were declared in
- In a new scope the same variable hides the existing one



http://cpp.sh/2ik4x

```
#include <iostream>
int main()
{
 int a= 1;
 int b= 10;
 std::cout << a << std::endl;
 std::cout << b << std::endl << std::endl;
 {
     int a=100;
     b=1000;
     int c=10000;
     std::cout << a << std::endl;
     std::cout << b << std::endl;
     std::cout << c << std::endl << std::endl;
 }
 std::cout << a << std::endl;
 std::cout << b << std::endl;
 //Would not compile:
 //std::cout << c << std::endl; //error: 'c' was not declared in this scope
}
/*
1
10

100
1000
10000

1
1000*/
```

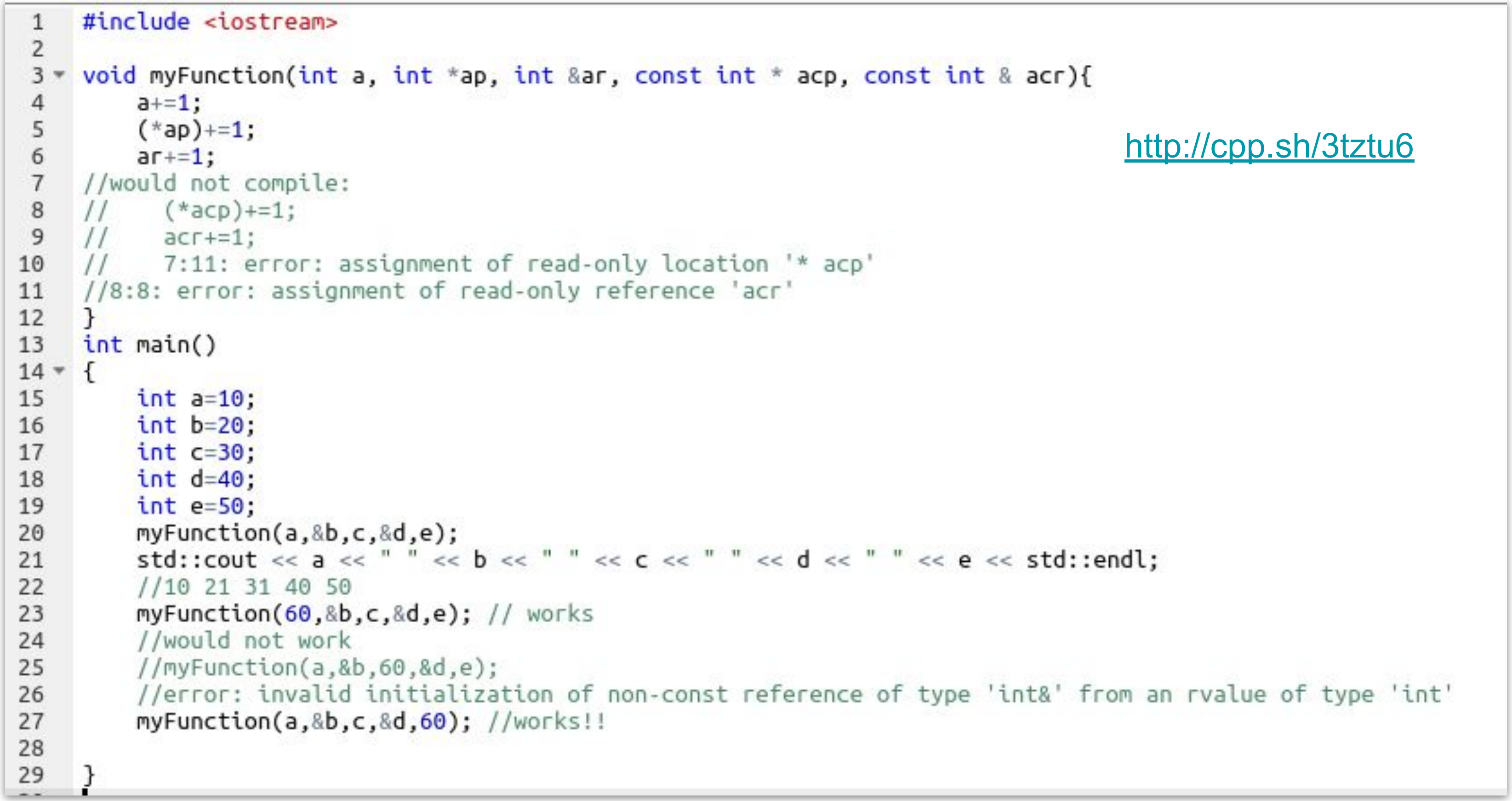
- Variables can be grouped in “namespaces”
- Namespaces should be explicitly specified unless it is said that you implicitly **use** them or that a given variable should be taken from there



http://cpp.sh/3hzo6

```
#include <iostream>
namespace myVars {
     int a=1;
     float b=7;
}

int main()
{
    std::cout << myVars::a << std::endl;

// not compiling:
// std::cout << a << std::endl;
/*In function 'int main()':
8:15: error: 'a' was not declared in this scope
8:15: note: suggested alternative:
3:9: note:   'myVars::a'*/

  using  myVars::a;
  std::cout << a << std::endl;
  using namespace myVars;
  std::cout << a << " " << b << std::endl;
}
```

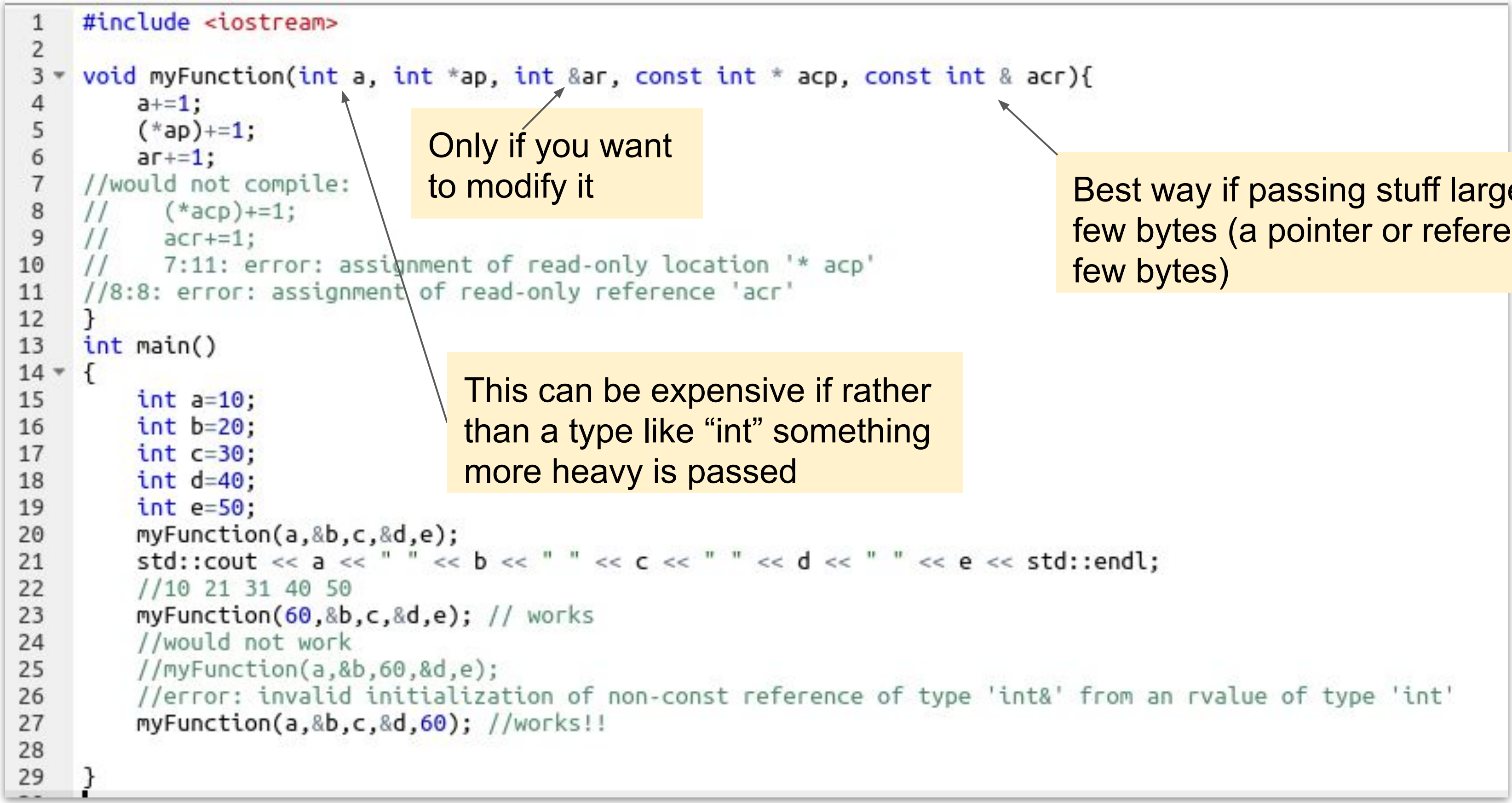
Functions: passing by reference/pointers vs value



```
#include <iostream>

void myFunction(int a, int *ap, int &ar, const int * acp, const int & acr){
    a+=1;
    (*ap)+=1;
    ar+=1;
//would not compile:
//    (*acp)+=1;
//    acr+=1;
//    7:11: error: assignment of read-only location '* acp'
//8:8: error: assignment of read-only reference 'acr'
}
int main()
{
    int a=10;
    int b=20;
    int c=30;
    int d=40;
    int e=50;
    myFunction(a,&b,c,&d,e);
    std::cout << a << " " << b << " " << c << " " << d << " " << e << std::endl;
    //10 21 31 40 50
    myFunction(60,&b,c,&d,e); // works
    //would not work
    //myFunction(a,&b,60,&d,e);
    //error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
    myFunction(a,&b,c,&d,60); //works!!

}
```

http://cpp.sh/3tztu6

Functions: passing by reference/pointers vs value



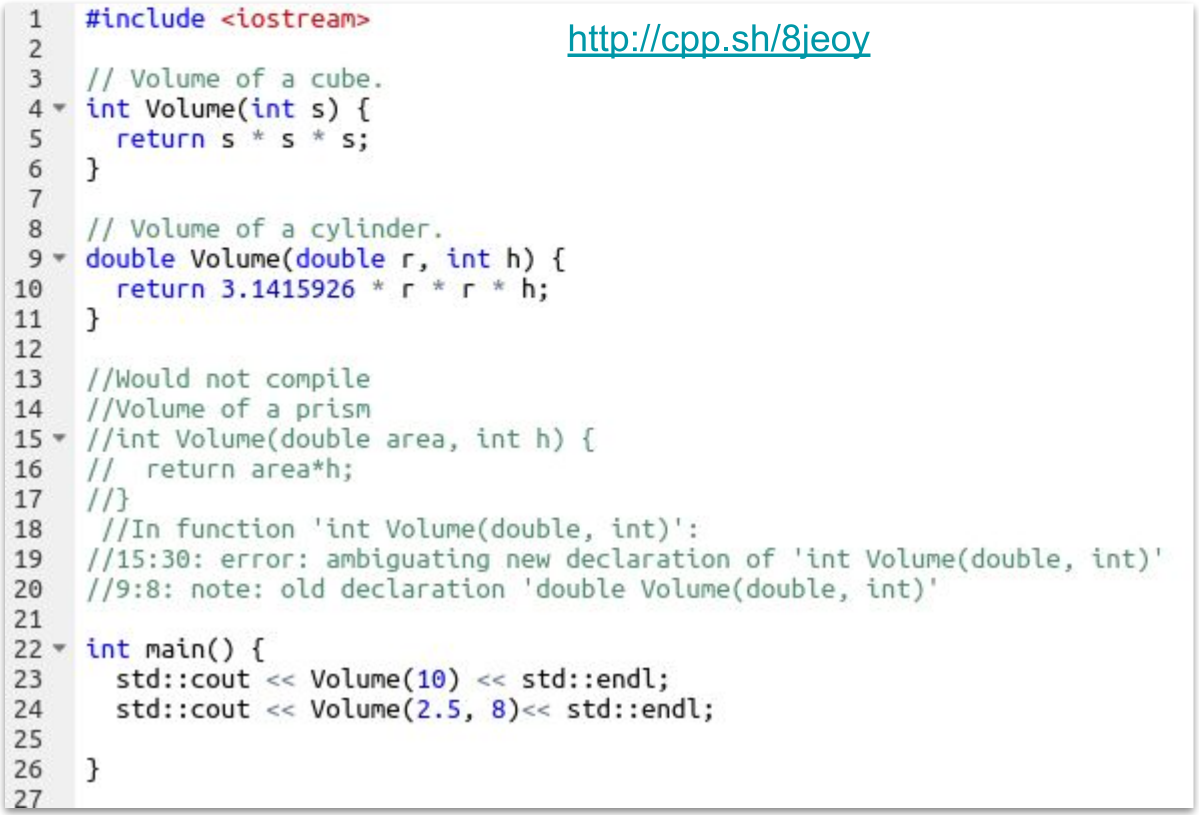
```
#include <iostream>

void myFunction(int a, int *ap, int &ar, const int * acp, const int & acr){
    a+=1;
    (*ap)+=1;
    ar+=1;
//would not compile:
//    (*acp)+=1;
//    acr+=1;
//    7:11: error: assignment of read-only location '* acp'
//8:8: error: assignment of read-only reference 'acr'
}
int main()
{
    int a=10;
    int b=20;
    int c=30;
    int d=40;
    int e=50;
    myFunction(a,&b,c,&d,e);
    std::cout << a << " " << b << " " << c << " " << d << " " << e << std::endl;
    //10 21 31 40 50
    myFunction(60,&b,c,&d,e); // works
    //would not work
    //myFunction(a,&b,60,&d,e);
    //error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
    myFunction(a,&b,c,&d,60); //works!!

}
```

Only if you want to modify it

Best way if passing stuff larger than few bytes (a pointer or reference is few bytes)

This can be expensive if rather than a type like “int” something more heavy is passed

Function overloading



```
 1  #include <iostream>
 2
 3  // Volume of a cube.
 4  int Volume(int s) {
 5    return s * s * s;
 6  }
 7
 8  // Volume of a cylinder.
 9  double Volume(double r, int h) {
10    return 3.1415926 * r * r * h;
11  }
12
13  //Would not compile
14  //Volume of a prism
15  //int Volume(double area, int h) {
16  //  return area*h;
17  //}
18   //In function 'int Volume(double, int)':
19  //15:30: error: ambiguating new declaration of 'int Volume(double, int)'
20  //9:8: note: old declaration 'double Volume(double, int)'
21
22  int main() {
23    std::cout << Volume(10) << std::endl;
24    std::cout << Volume(2.5, 8)<< std::endl;
25
26  }
27
```

http://cpp.sh/8jeoy

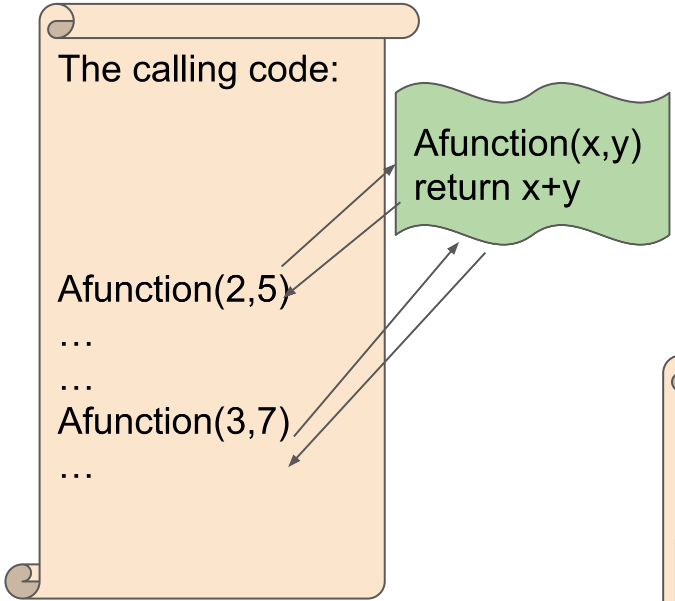
A **function's signature** includes the **function's** name and the number, order and type of its formal parameters. Two overloaded **functions** must not have the same **signature**. The return value is not part of a **function's signature**.

Some parameters can have **default values** and can be omitted in the call

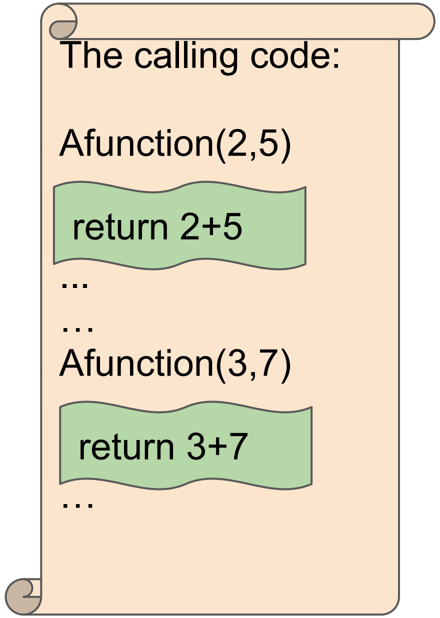
```
int diag(float x1, float x2, float x3=0){
    return sqrt(x1*x1+x2*x2+x3*x3)
}
```

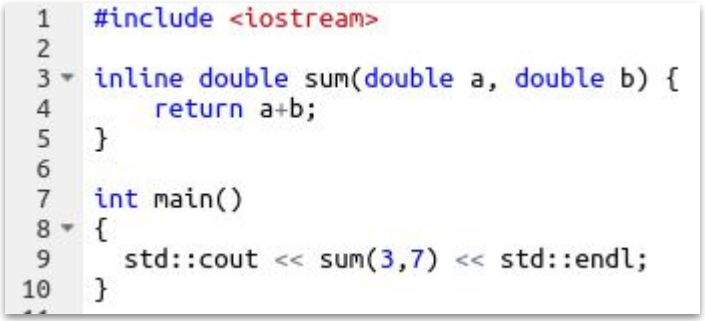
Inline functions

- The code of inline functions is repeated in each place where the function is called
- Non inline functions code is in a single place and execution jumps to the right place at each call
- Inline can be "suggested" to the compiler with "inline" keyword
  - And in other cases discussed later (class members)
- Inline make sense for small functions (code size grows) to avoid "jump" overhead



INLINE

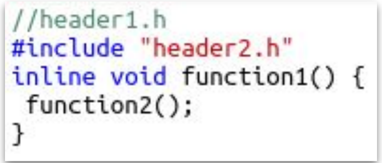




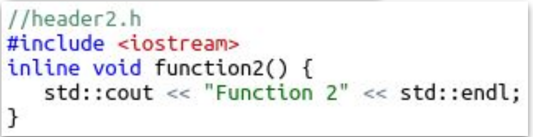
```cpp
#include <iostream>

inline double sum(double a, double b) {
    return a+b;
}

int main()
{
  std::cout << sum(3,7) << std::endl;
}
```

Separate definition and declaration

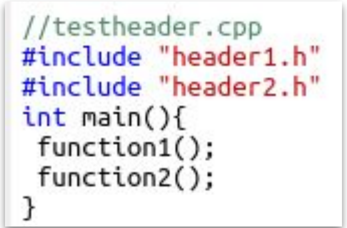
- Function declaration (the function exists) and function definition (what the function does) can be separated
  - Allow e.g. to have two functions calling each other
- Declarations of functions (and classes, see next slides) are often separated in a separated file named “header file” or “include file” or simply “.h file”
- Header files can then be included in source code
  - If the declaration of the function is given in the .h file, it must be an inline function as otherwise the code for the function would be generated multiple times
  - Header files can include other header files, the #ifndef mechanism is used to protect against multiple includes



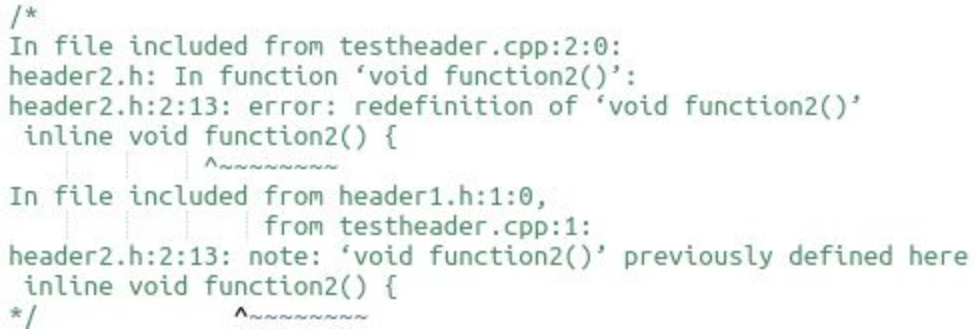
```
//header1.h
#include "header2.h"
inline void function1() {
 function2();
}
```



```
//header2.h
#include <iostream>
inline void function2() {
   std::cout << "Function 2" << std::endl;
}
```



```
//testheader.cpp
#include "header1.h"
#include "header2.h"
int main(){
 function1();
 function2();
}
```



```
/*
In file included from testheader.cpp:2:0:
header2.h: In function ‘void function2()’:
header2.h:2:13: error: redefinition of ‘void function2()’
 inline void function2() {
             ^~~~~~~~~
In file included from header1.h:1:0,
                 from testheader.cpp:1:
header2.h:2:13: note: ‘void function2()’ previously defined here
 inline void function2() {
             ^~~~~~~~~
*/
```

Separate definition and declaration

- Function declaration (the function exists) and function definition (what the function does) can be separated
  - Allow e.g. to have two functions calling each other
- Declarations of functions (and classes, see next slides) are often separated in a separated file named “header file” or “include file” or simply “.h file”
- Header files can then be included in source code
  - If the declaration of the function is given in the .h file, it must be an inline function as otherwise the code for the function would be generated multiple times
  - Header files can include other header files, the #ifndef mechanism is used to protect against multiple includes

```
//header1.h
#ifndef HEADER_1_
#define HEADER_1_
#include "header2.h"
inline void function1() {
 function2();
}
#endif
```

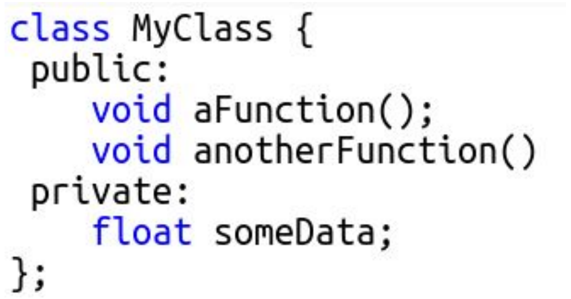
```
//header2.h
#ifndef HEADER_2_
#define HEADER_2_
#include <iostream>
inline void function2() {
    std::cout << "Function 2" << std::endl;
}
#endif
```

```
//testheader.cpp
#include "header1.h"
#include "header2.h"
int main(){
 function1();
 function2();
}
```

“...”
vs
<...>

Classes

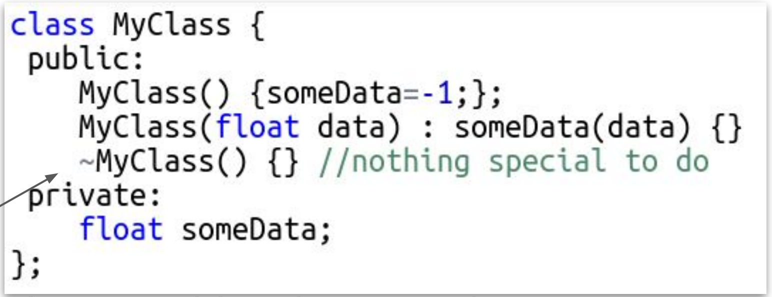
- In C++ you can define new types (classes) that combine
  - Data
  - Functions to act on the data
- The functions and data of a class are named “members” of the class
  - Data members
  - Function members (can be declared “**const**”, in that case they cannot modify the data members)
- Members can be private or public
  - Public members can be accessed outside the class
  - Private members are accessed only by member functions
    - This is useful to do “data hiding” i.e. do not expose the user with the “raw content” of the class but always access via dedicated function members (aka “accessors”)



```
class MyClass {
 public:
    void aFunction();
    void anotherFunction()
 private:
    float someData;
};
```

An instance of a class, i.e. an object

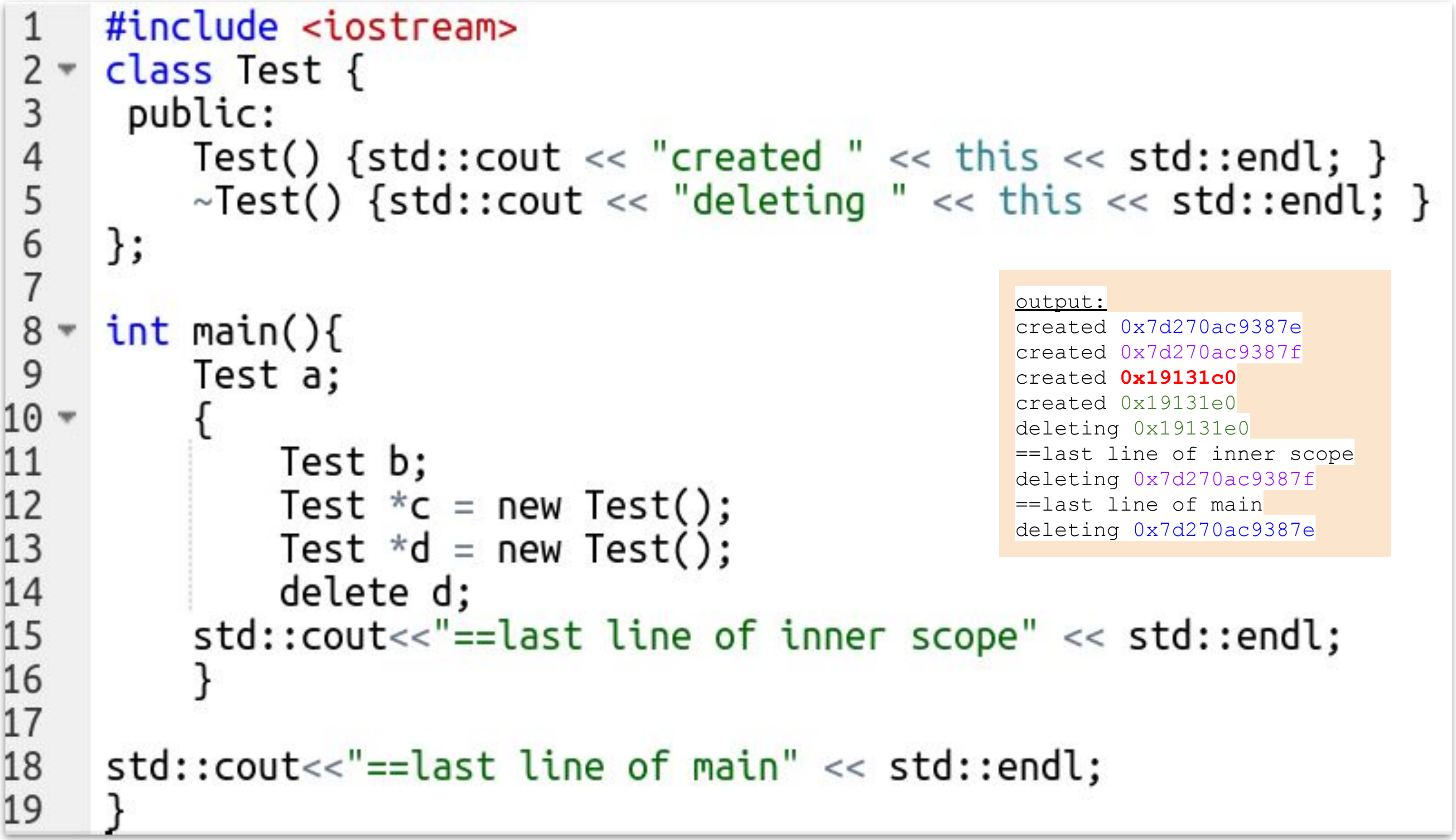
- If classes are new types, the variable you can create with those new types are called objects or instances
- When a new object is created a special function in the class is called, the "constructor"
  - Constructor has the same name of the class
  - It does not "return" and has no return type, should do init ops
  - It may have multiple signatures
  - Even if you do not specify any, **default constructor** (no arguments) and **copy constructor** are anyhow available
- When the object is deleted a "destructor" is called
- In general members of a class can be accessed only when an object is created
  - Unless they are declared "static"
    - Static function members are function that cannot access non static data members
    - Static data members are shared among all instances
- The "this" keyword can be used inside a (non static) function member to access the object pointer
- In c++ objects can also be create with "new" operator
  - In that case you need to "delete" them



```
class MyClass {
 public:
    MyClass() {someData=-1;};
    MyClass(float data) : someData(data) {}
    ~MyClass() {} //nothing special to do
 private:
    float someData;
};
```

http://cpp.sh/3m3tj

New&delete vs variables in scope

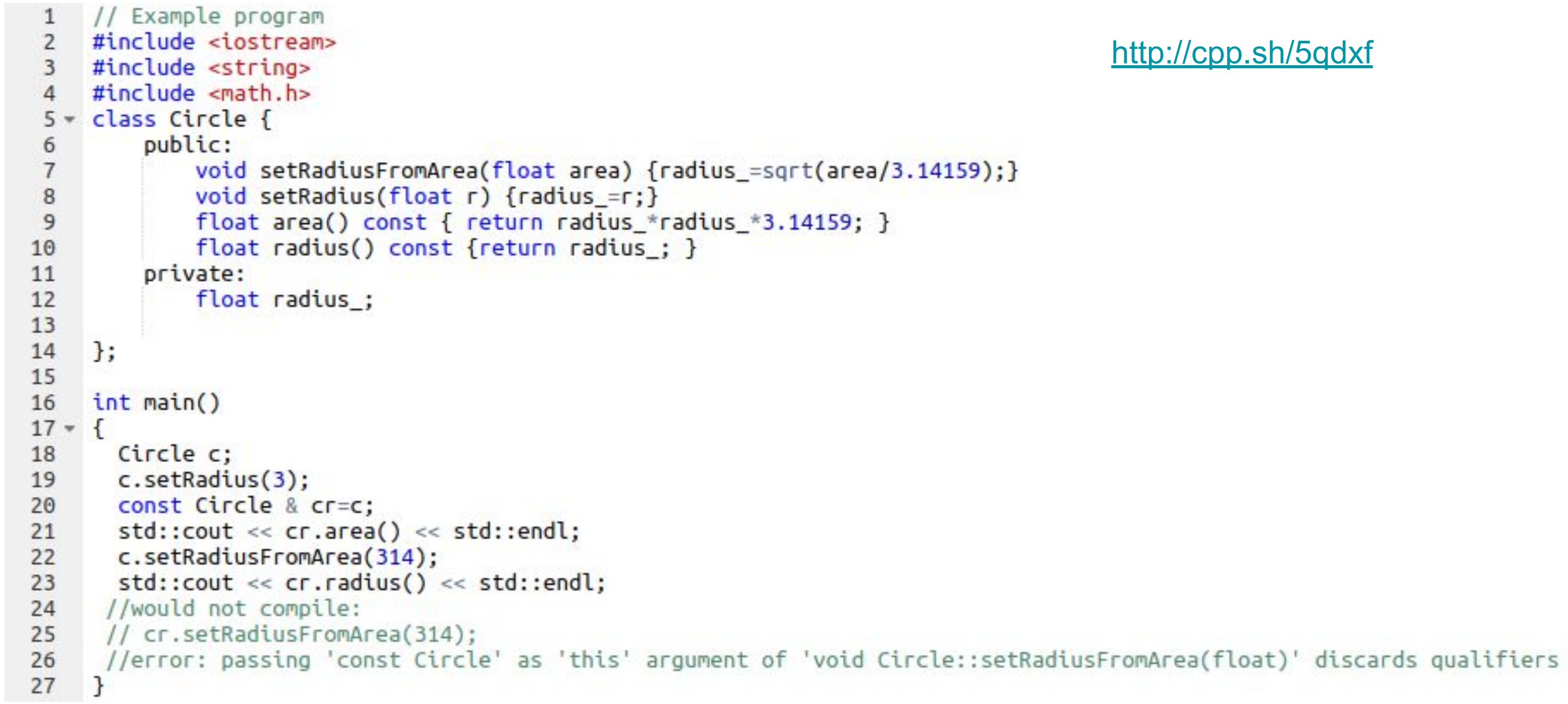


```cpp
#include <iostream>
class Test {
 public:
    Test() {std::cout << "created " << this << std::endl; }
    ~Test() {std::cout << "deleting " << this << std::endl; }
};

int main(){
    Test a;
    {
        Test b;
        Test *c = new Test();
        Test *d = new Test();
        delete d;
    std::cout<<"==last line of inner scope" << std::endl;
    }

std::cout<<"==last line of main" << std::endl;
}
```

```
output:
created 0x7d270ac9387e
created 0x7d270ac9387f
created 0x19131c0
created 0x19131e0
deleting 0x19131e0
==last line of inner scope
deleting 0x7d270ac9387f
==last line of main
deleting 0x7d270ac9387e
```

A class declaration

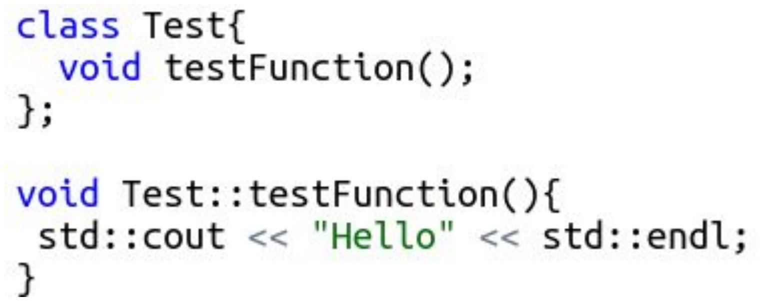
http://cpp.sh/5qdxf



```cpp
// Example program
#include <iostream>
#include <string>
#include <math.h>
class Circle {
    public:
        void setRadiusFromArea(float area) {radius_=sqrt(area/3.14159);}
        void setRadius(float r) {radius_=r;}
        float area() const { return radius_*radius_*3.14159; }
        float radius() const {return radius_; }
    private:
        float radius_;

};

int main()
{
  Circle c;
  c.setRadius(3);
  const Circle & cr=c;
  std::cout << cr.area() << std::endl;
  c.setRadiusFromArea(314);
  std::cout << cr.radius() << std::endl;
 //would not compile:
 // cr.setRadiusFromArea(314);
 //error: passing 'const Circle' as 'this' argument of 'void Circle::setRadiusFromArea(float)' discards qualifiers
}
```

Access to class functions and members

- "ClassName::Member" is used to refer to a member of a class outside the class declaration
    - E.g. for the function definition!



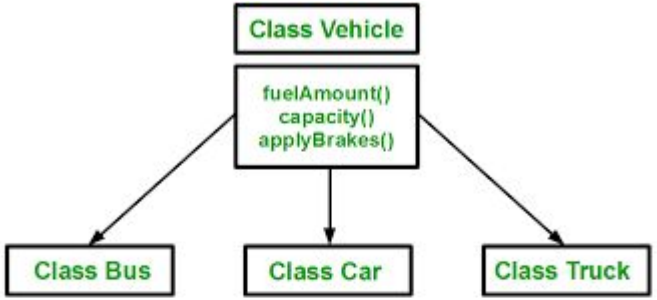
```cpp
class Test{
  void testFunction();
};

void Test::testFunction(){
 std::cout << "Hello" << std::endl;
}
```

- A member of an object can be accessed with ".", e.g.
    - object.aMemberFunction()
    - object.aDataMember
- Given a pointer to an object "." is replaced with "->"
    - objPointer->aMemberFunction()
    - objPointer->aDataMember

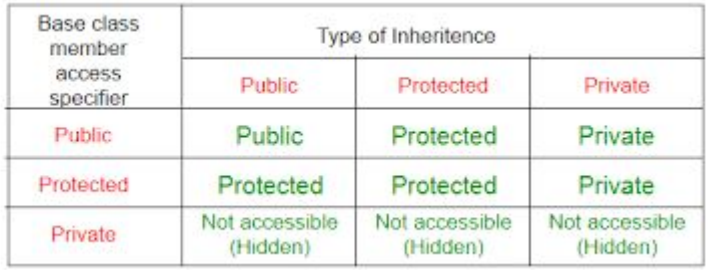
Inheritance and polymorphism

- Classes can be organized in hierarchy

```
class DerivedClass : public BaseClass {
    DerivedClass(int a, float b) :
                BaseClass(a) {}
    //more derived class stuff
};
```



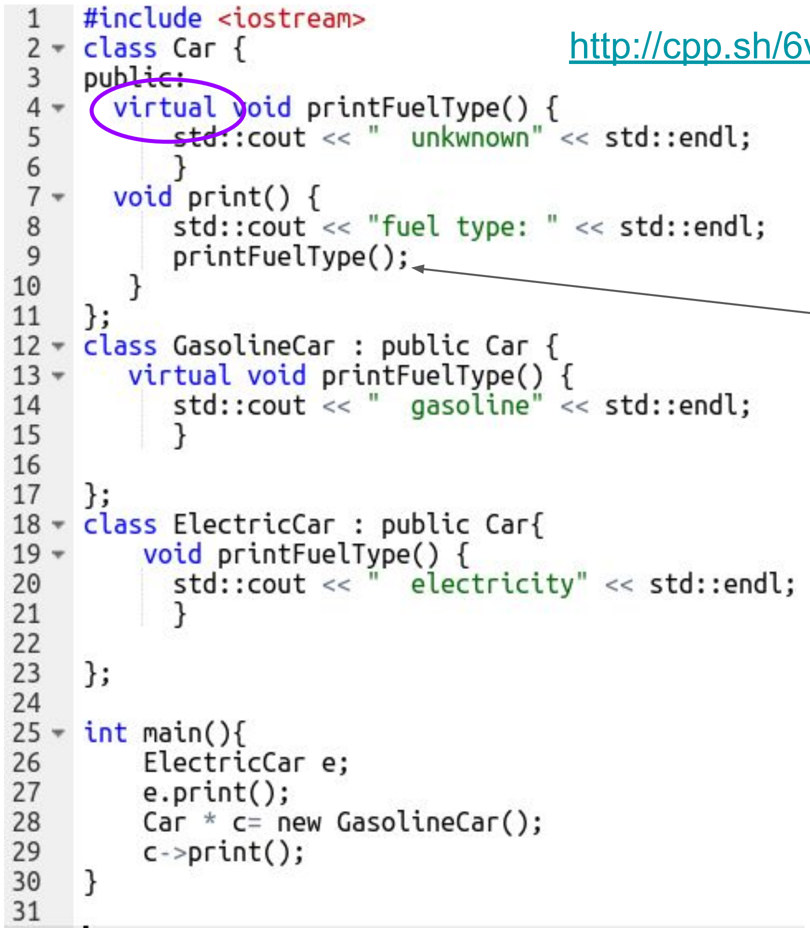
- Members of a class are inherited in derived classes
    - Access can be “protected” instead of “private” to let the inherited classes have access
    - Inheritance can be public/protected/private
- Functions that are declared “virtual” can be re-implemented in the derived class in a specialized version
- A pointer to a base class can effectively contain a derived class instance



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

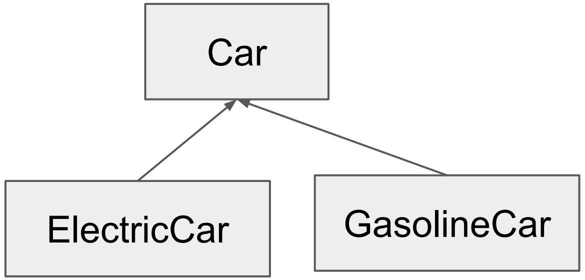
```
object->aDataMember;
object->aFunctionMember();
```

Virtual functions



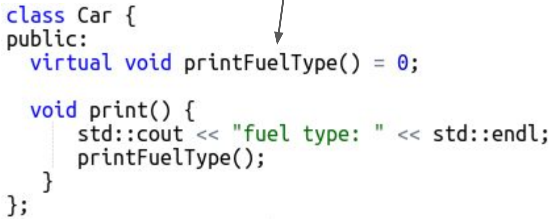
```
#include <iostream>
class Car {
public:
  virtual void printFuelType() {
      std::cout << "  unkwnown" << std::endl;
      }
  void print() {
      std::cout << "fuel type: " << std::endl;
      printFuelType();
    }
};
class GasolineCar : public Car {
   virtual void printFuelType() {
      std::cout << "  gasoline" << std::endl;
      }

};
class ElectricCar : public Car{
    void printFuelType() {
      std::cout << "  electricity" << std::endl;
      }

};

int main(){
    ElectricCar e;
    e.print();
    Car * c= new GasolineCar();
    c->print();
}

```

http://cpp.sh/6vfg



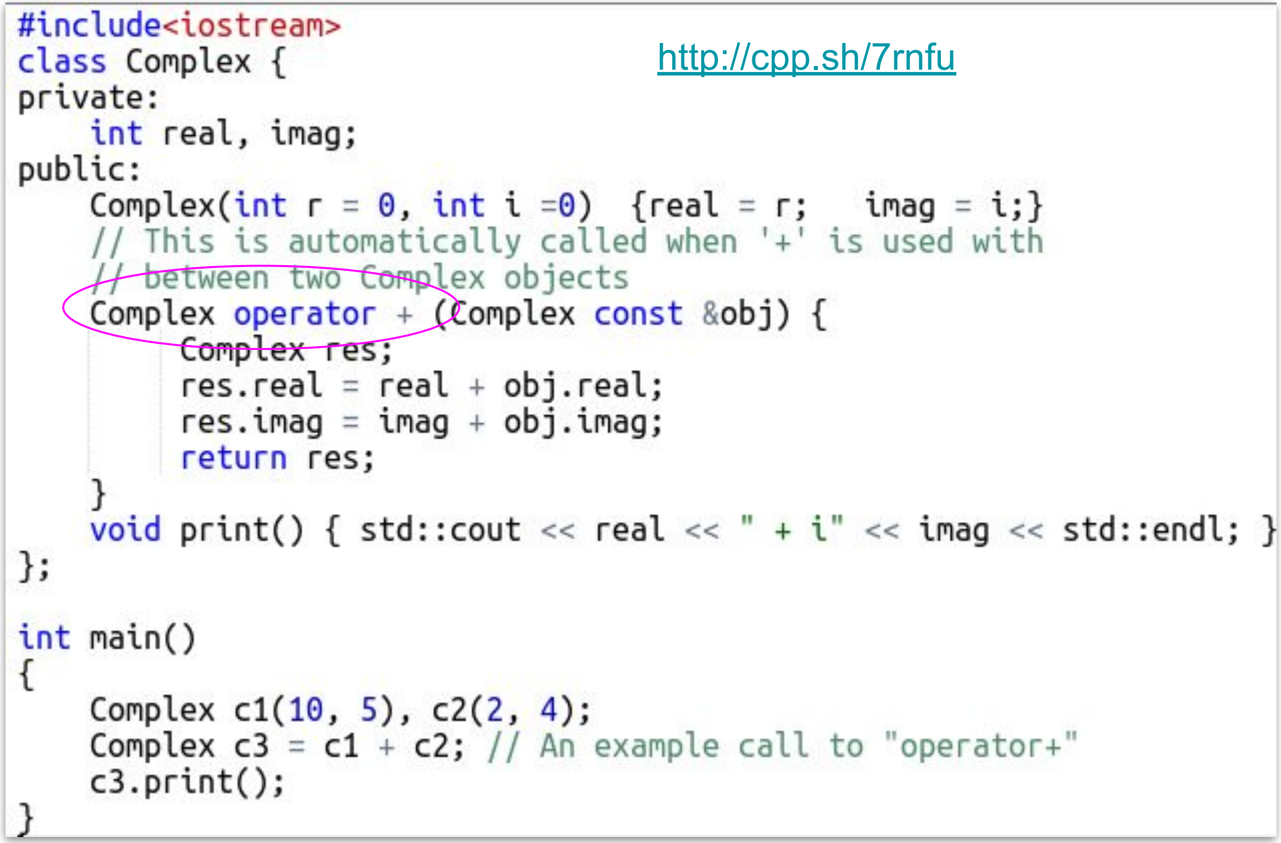
Because printFuelType is virtual the function implemented in the concrete object will be called

The base class can also not implement a function (i.e. "pure virtual")
… but then no object of the base class can be created



```
class Car {
public:
  virtual void printFuelType() = 0;

  void print() {
      std::cout << "fuel type: " << std::endl;
      printFuelType();
    }
};
```

Operators overloading



http://cpp.sh/7rnfu

```
#include<iostream>
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { std::cout << real << " + i" << imag << std::endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

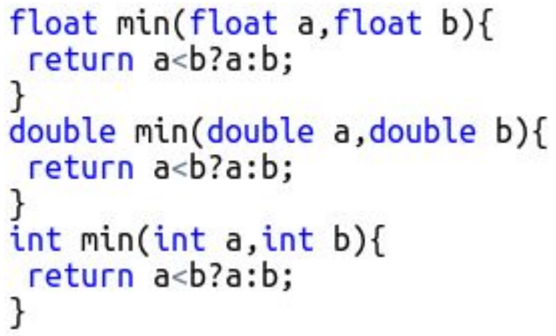
The "operator" keyword, followed by an operator (+,-,* etc..) can be used to define functions to use when using operators with objects operands

First assignment (for next week)

- Create a 4-vector class
  - storing internally cartesian representation (px, py, pz, E)
  - Hide data and define accessors
  - Add functions that returns invariant mass and transverse momentum (and more if you want phi, theta, eta, rapidity, gamma, beta, etc...)
- Create operators that sum 4-vectors or multiply them by scalars
- Create derived class “Particle”
  - Adds particle “id” and particle decay time
- Create derived class “TwoBodiesDecayedParticle”
  - It is still a Particle
  - Internally store the two decay products
  - At constructors it computes the 4-vector from the decay products

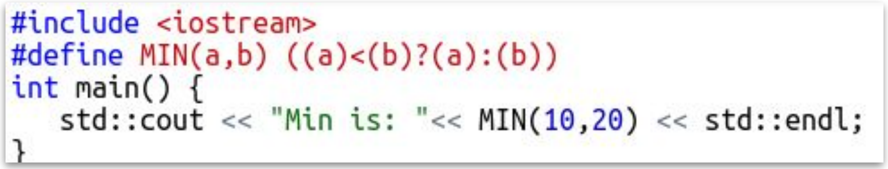
Templates and meta programming

- Functions allow to avoid rewriting exactly the same code multiple types
  - But if you want to change the type of the input you need a new function



```
float min(float a,float b){
 return a<b?a:b;
}
double min(double a,double b){
 return a<b?a:b;
}
int min(int a,int b){
 return a<b?a:b;
}
```

Example meta programming with preprocessor



```
#include <iostream>
#define MIN(a,b) ((a)<(b)?(a):(b))
int main() {
   std::cout << "Min is: "<< MIN(10,20) << std::endl;
}
```

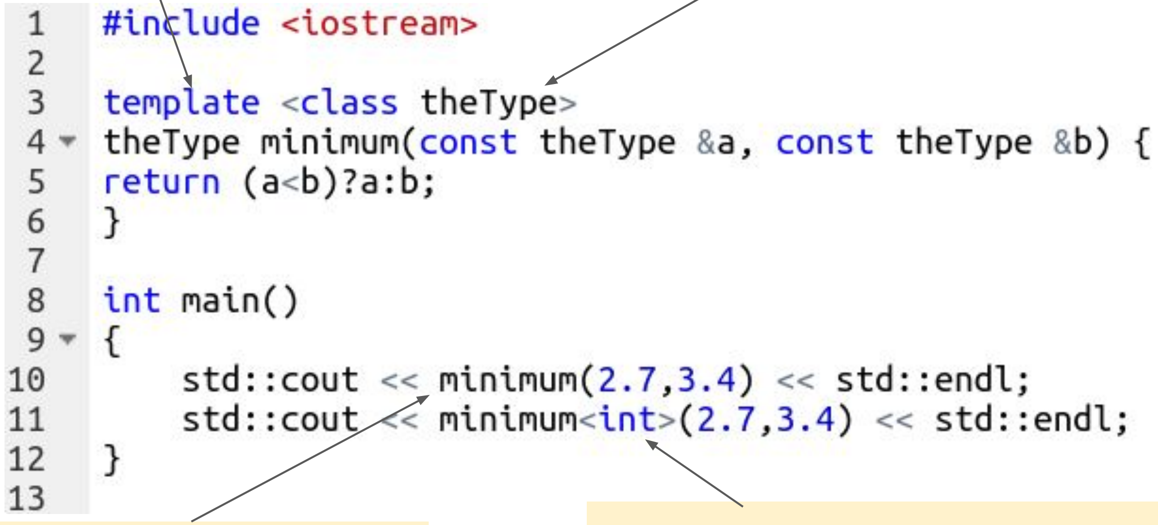
- Templates in c++ allow much more complex metaprogramming
- Templates allow to write code independently of the concrete types of some objects
- Templates are translated to actual code at compile time based on what the compiled code effectively needs

A template function

http://cpp.sh/8rvf

template keyword

Placeholder name for the actual class name



```cpp
#include <iostream>

template <class theType>
theType minimum(const theType &a, const theType &b) {
return (a<b)?a:b;
}

int main()
{
    std::cout << minimum(2.7,3.4) << std::endl;
    std::cout << minimum<int>(2.7,3.4) << std::endl;
}
```

Here it deduces the type of “theType”

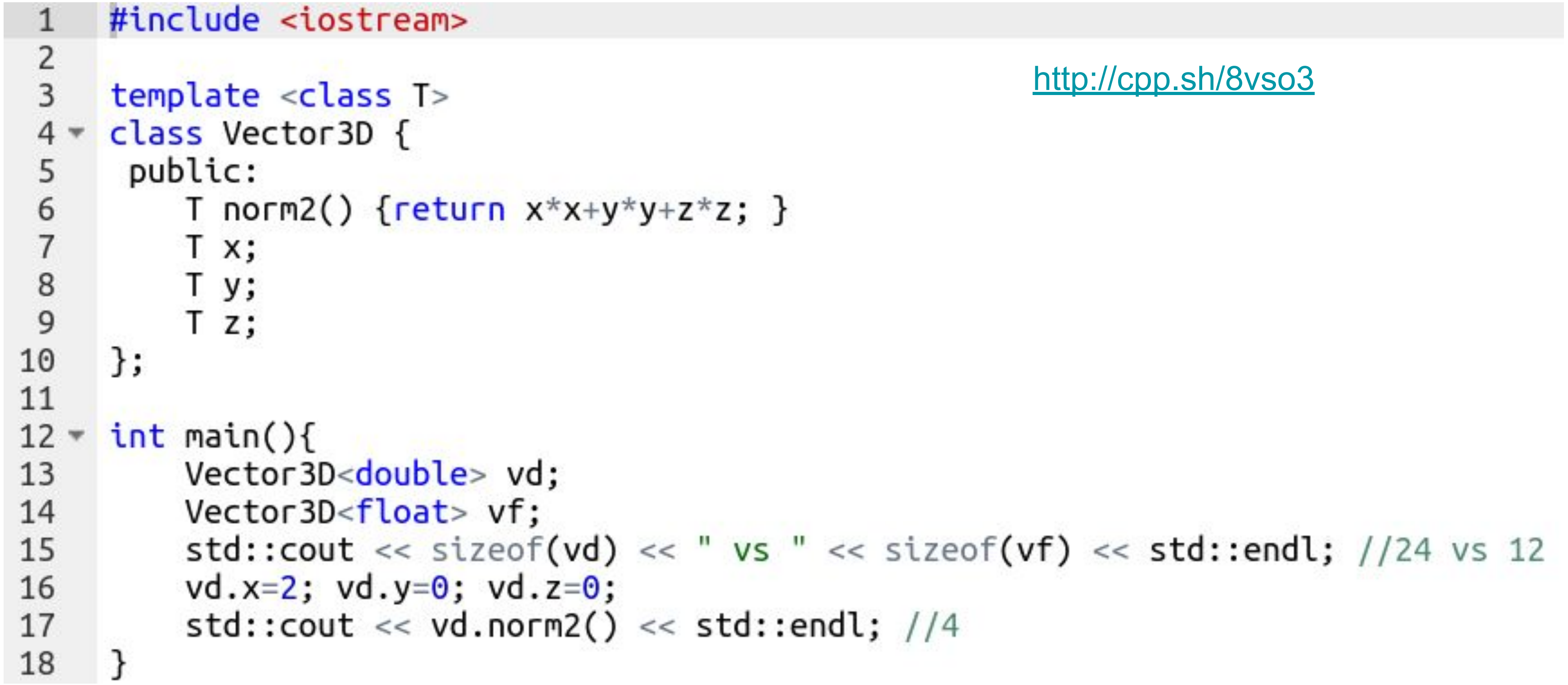
Here we force the “int” version, hence the floats are casted and precision is lost

output:

```
2.7
2
```

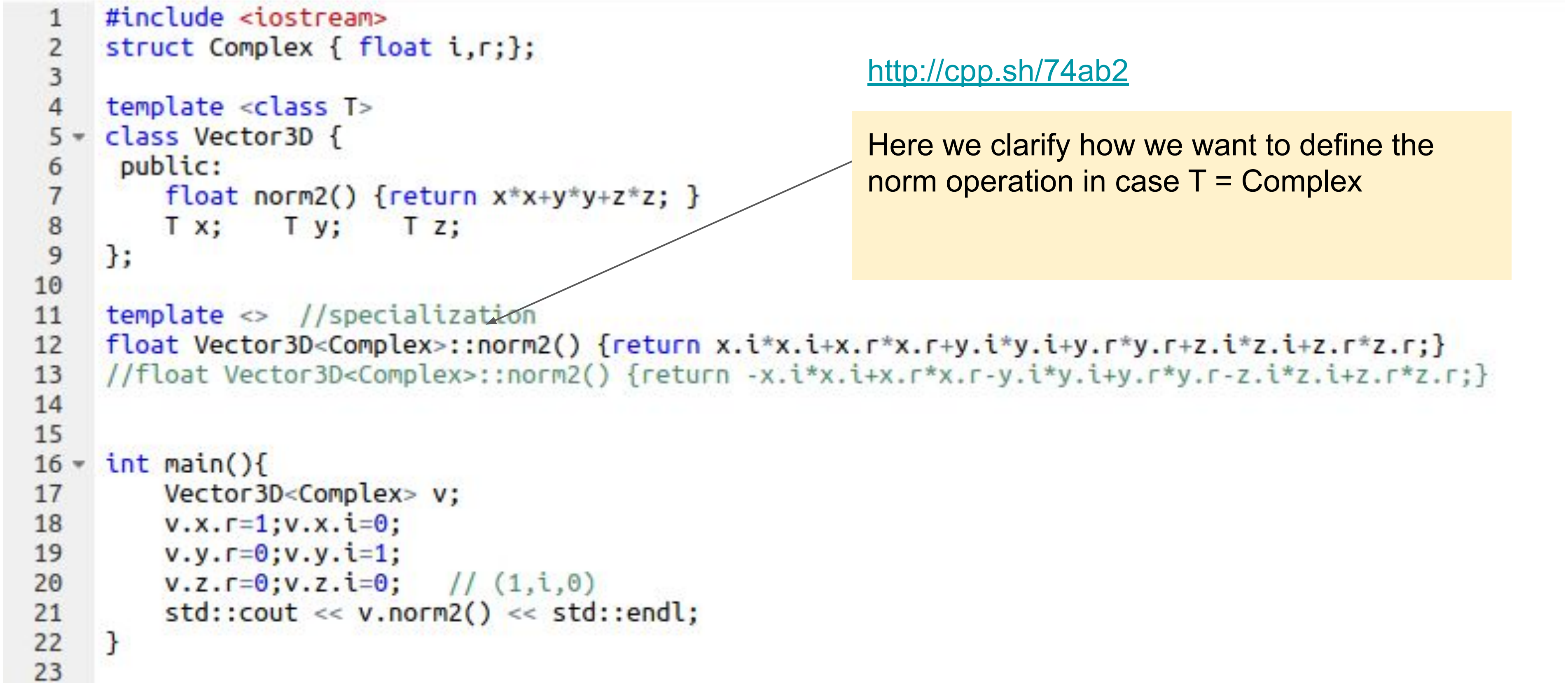
A template class

http://cpp.sh/8vso3



```cpp
#include <iostream>

template <class T>
class Vector3D {
 public:
    T norm2() {return x*x+y*y+z*z; }
    T x;
    T y;
    T z;
};

int main(){
    Vector3D<double> vd;
    Vector3D<float> vf;
    std::cout << sizeof(vd) << " vs " << sizeof(vf) << std::endl; //24 vs 12
    vd.x=2; vd.y=0; vd.z=0;
    std::cout << vd.norm2() << std::endl; //4
}
```

Template specialization



```cpp
#include <iostream>
struct Complex { float i,r;};

template <class T>
class Vector3D {
 public:
    float norm2() {return x*x+y*y+z*z; }
    T x;    T y;    T z;
};

template <>  //specialization
float Vector3D<Complex>::norm2() {return x.i*x.i+x.r*x.r+y.i*y.i+y.r*y.r+z.i*z.i+z.r*z.r;}
//float Vector3D<Complex>::norm2() {return -x.i*x.i+x.r*x.r-y.i*y.i+y.r*y.r-z.i*z.i+z.r*z.r;}


int main(){
    Vector3D<Complex> v;
    v.x.r=1;v.x.i=0;
    v.y.r=0;v.y.i=1;
    v.z.r=0;v.z.i=0;   // (1,i,0)
    std::cout << v.norm2() << std::endl;
}
```

http://cpp.sh/74ab2

Here we clarify how we want to define the norm operation in case T = Complex

Code conventions

- Always respect coding rule of the project you write code for!
- Always respect coding rule of the project you write code for!
- Example (common) coding rules
  - Functions start with lower case, e.g. **anExampleFunction()**
  - Classes start with upper case, e.g. **MyNewClass**
  - Data member have a trailing “_” or a “m_” prefix, e.g. **radius_** or **m_radius**
  - Classes with pure virtual functions should say it in the name, e.g. **Vehicle<u>Base</u>**
  - Accessors (aka getters) and setters should follow a common schema, e.g.
    - radius() (some project prefers getRadius())
    - setRadius(r)
  - … read coding rules documents!...
- Always respect coding rule of the project you write code for!
- Always respect coding rule of the project you write code for!
- Always respect coding rule of the project you write code for!

Why OO is not cool anymore

- ObjectOriented programming was very trendy in the 90ies
  - Every task was thought in term of interacting objects
  - Factories, Dispatchers, Observers, Builder, …
- Not clear where to write the code that let the objects interact
- Often resulting in an inefficient data representation

How to compile a cpp file

- gcc/g++ compile and/or link your code
  - **Compile** = transform the C++ into code that can be executed
  - **Link** = put together different pieces of compiled code, either consolidating in a single executable (static linking) or with multiple files being loaded at runtime (shared libraries)
- `sh> g++ filename.cpp   #this will create a.out executable`
- `sh> g++ filename.cpp -o exename #this creates exename executable`
- `sh> g++ filename.cpp  -I headersDirectory/ #where to search for .h files`
- `sh> g++ filename.cpp  -L librariesDirectory/ -lmytools`
- `Creating an using a shared library`
  - `sh> gcc -c -fpic librarycode.c -o librarycode.o`
  - `sh> gcc -shared -o libtools.so librarycode.c`
  - `sh> g++ application.cpp -L. -l tools`
  - `sh> g++ otherapp.cpp -L. -l tools`
  - `Run with: sh> LD_LIBRARY_PATH=. ./a.out`
- Step zero for this exercise session: open a terminal, create simple hello world program and compile it. Then put a function in a shared library and link it from a program defined in a different cpp file

Second assignment

- Template the 4-vector definition (e.g. to use with float vs double)
- Template the 4-vector internal representation PtEtaPhiMass vs PxPyPzE
- Implement seamless algebra

Afternoon exercise session

First assignment

- Create a 4-vector class
  - storing internally cartesian representation (px, py, pz, E)
  - Hide data and define accessors
  - Add functions that returns invariant mass and transverse momentum (and more if you want phi, theta, eta, rapidity, gamma, beta, etc...)
  - Make a program showing the functionalities of the class
- Create operators that sum 4-vectors or multiply them by scalars
  - Print (p4_a + p4_b).m() and pt()
- Create derived class “Particle”
  - Adds particle “id” and particle decay time
- Create derived class “TwoBodiesDecayedParticle”
  - It is still a Particle
  - Internally store the two decay products
  - At constructors it computes the 4-vector from the decay products
  - Make a print member function that summarize all the class info

Second assignment

- Template the 4-vector definition (e.g. to use with float vs double)
  - Be sure that sum can be done between FourVector<double> and FourVector<float>
- Implement a PtEtaPhiMassVector
- Create member functions to convert from one class to the other
  - You can overload the cast operator!:

```
operator AClassName() const { return ...something... }
```

- Create operators to sum from different types