COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2026

C++

Object-oriented programming is a programming style that captures the behavior of the real world in a way that hides detailed implementation, and it allows the problem solver to think in terms of the problem domain. C++ was created with two goals: to make it compatible with ordinary C, and to extend C with OO constructs.

FAQ:http://www.parashift.com/c++-faq-lite/

Object-oriented programming is a data-centered view of programming, in which data and behavior are strongly linked. Data and behavior are conceived of as classes whose instances are objects. Objects are class variables, and object-oriented programming allows abstract data structures to be easily created and used.

A C++ program is a collection of declarations and functions that begin with executing the function `main()`. The program is compiled with a first phase executing any preprocessor directives, such as

```
#include <string>
```

Namespaces were introduced to provide a scope that allows different code providers to avoid global name clashes. The

```
namespace std;
```

is reserved for use with the standard libraries. The `using` declaration allows the identifiers found in the standard library to be used without

being qualified. Without this declaration the program would have to use `std::cout` instead of `cout`.

In OO terminology, a variable is called an *object*. A constructor is a member function whose job is to initialize an object of its class. Constructors are invoked whenever an object of its class is created. A destructor is a member function whose job is to finalize a variable of its class. The destructor is called implicitly when an automatic object goes out of scope.

Native types

The simple native types in C++ are `bool`, `double`, `int` and `char`. These types have a set of values and representation that is tied to the underlying machine architecture on which the compiler is running. C++ simple types can be modified by the keywords `short`, `long`, `signed` and `unsigned` to yield further simple types.

Fundamental data types		
bool		
char	signed char	unsigned char
wchar_t		
short	int	long
unsigned short	unsigned	unsigned long
float	double	long double

A variable declaration associates a type with a variable name. A declaration of a variable constitutes a definition, if storage is allocated for it. The definition can be thought as creating an object. Int:

```
int i;
```

A definition can also initialize the value of the variable

```
int i = 0;
int i = 0,j;
```

C++ declarations are themselves statements and they can occur throughout a block.

An automatic type conversion can occur across assignments and mixed statements. A promotion (widening) is usually well behaved, but demotions (narrowing) can lose information. In mixed statements the type conversion follows promotion rules.

In addition to implicit conversions, there are explicit conversions called `casts`.

```
static_cast<double>(i)
const_cast<double>(constVar)
```

Older C++ systems allow an unrestricted form of cast with

```
(type)expression or
type(expression)
```

Expressions

The arithmetic operators in C++ are +,-,*,/,%. To raise to a power one can use the function **pow**. Arithmetic expressions are consistent with the expected practise, with one important difference: division operator. The result of the division operator depends on its arguments

```
a = 3 / 2;
```

C++

```
a = 3 / 2.0;
```

Relational (<, >, <=, >=), equality (==,!=) and logical (!,&&,||) operators work as expected. In the evaluation of expressions that are the operands of && and ||, the evaluation process stops as soon as the outcome is known. If expr1 is false, then in

```
expr1 && expr2
```

expr2 will not be evaluated. Similarly, if expr1 is true, then in

```
expr1 || expr2
```

expr2 will not be evaluated since the value of the logical expression is already determined.

The conditional operator ?: is a ternary operator, taking three expressions as operands. In a construct such as

```
expr1 ? expr2 : expr3
```

expr1 is evaluated first. If it is true, expr2 is evaluated, and that is the value of the conditional expression as a whole. If expr1 is false then expr3 is evaluated. For example

```
x = (y < z) ? y : z;
```

assigns the smaller of the two values to x.

C++ provides also assignment operators that combine an assingment and some operator like

```
a += b; // a = a + b
a *= a + b; // a = a*(a+b)
i++; // i = i + 1
i- -; // i = i - 1
j = i++; // j = i; i = i + 1
```

j = ++i; // i = i + 1; j = i

Notice that the two last examples are not equivalent.

Statements

The general forms of basic control flow statements are

if (condition) statement

if (condition) statement1 else statement2

while (condition) statement

for (init; condition; increment) statement

do statement while (condition)

switch (condition) statement

To interrupt the normal flow of control within a loop, two special statements can be used: `break` and `continue`

Example: a simple for loop

```
for(int i = 0; i < 3; i++){
  cout << i << endl;
}
```

Example: a while loop

```
while (getline(file,line)){
  cout << line.length() << endl;
}
```

Functions

A C++ program is made of one or more functions, one of which is `main()`. The form for a function definition is

```
type name(parameter-list) {
  statements
}
```

C++

The type specification that precedes the function name is the return type, and the value is returned with statement `return`. If the function does not return any value, the return type of the function is `void`.

The parameters in the parameter list can be given default arguments

```
void myFunction(int i, int j = 1){...}
```

It is possible to overload functions

```
void myFunction(int i){...}
void myFunction(double d){...}
```

Pointers

Pointers are used to reference variables and machine addresses. They can be used to access memory and manipulate addresses. Pointer variables can be declared in programs and then used to take addresses as values. The declaration

```
int* p;
```

declares p to be of type pointer to int. The legal range of values for any pointer always includes the special address 0.

```
int* p = &i;
```

The variable p here can be thought of as referring to i or pointing to i or containing the address of i.

The dereferencing or indication operator * can be used to return the value of the variable the pointer points to

```
int j = *p;
```

Here *p returns the value of variable i.

Arrays

An array declaration is of the form

```
int a[size];
```

The values of an array can be accessed by a[expr], where expr is an integer expression getting values from 0 to (size-1). Arrays can be initialized by a comma separated list

```
int a[3] = {1, 2, 3};
```

or element by element a[0] = 1; a[1] = 2; a[2] = 3;. Arrays can be thought of constant pointers. Pointer arithmetic provides an alternative to array indexing

```
int *p = a;
int *p = &a[0];
```

Arrays can be of any type, including arrays of arrays. Multidimensional arrays can be formed by adding bracket pairs

```
int a[2][2]; // Not a[2,2]!
int b[2][5][200];
```

Abstract Data Structures

ADT implementations are used for user-defined data types. A large part of the OO program design process involves thinking up the appropriate data structures for a problem.

The `structure` type allows the programmer to aggregate components into a single named variable. A structure has components, called members, that

are individually named. Since the members of a structure can be of various types, the programmer can create aggregates suitable for describing complicated data.

Example: complex numbers

```
struct complex {
  double re,im;
};

complex c;
c.re = 1; c.im = 2;
```

Member functions

The concept of `struct` is augmented in C++ to allow functions to be members. The function declaration is included in the structure declaration.

```
struct complex {
  void reset(){
    re = 0;
    im = 0;
  }
  double re,im;
};
```

The member functions can be `public` or `private`. Public members are visible outside the data structure, while private can be accessed only within the structure. Very often the data is hidden, and accessed only by functions built for that purpose.

Hiding data allows more easily debugged and maintained code because errors and modifications are localized.

C++

Classes

Classes are introduced by a keyword `class`. They are a form of struct whose default privacy specification is `private`. Thus struct and class can be used interchangeably with the appropriate access specifications.

```
class complex {
 public:
   void reset(){
     re = 0;
     im = 0;
   }
 private:
   double re,im;
};
```

Class adds a new set of scope rules to those of the kernel language. The scope resolution operator :: comes in two forms:

```
::i // refers to external scope
foo_bar::i // refers to class scope
```

Classes can nest. This means that there can be classes within classes. Referencing the outer or inner variables is done with the scope resolution operator.

```
class X {
  int i; // X::i
  public:
  class Y {
    int i; // X::Y::i
  };
};
```

The definition of a purely local class is unavailable outside their local

scope.

Static variables are shared by all variables of that class and stored in only one place.

They keyword `this` denotes an implicitly declared self-referential pointer.

```
class X {
  public:
    X clone(){return (*this);}
};
```

The member functions can also be `static` and `const`. A static member function can exist independent of any specific variables of the class type being declared. Const member functions cannot change the values of the data members. Writing out const member functions and const parameter declarations is called const-correctness. It makes the code more robust.

```
class X {
  static int Y();
  int Z() const;
};
int X::Y(){}
int X::Z() const {}
```

Input/Output

Output is inserted into an object of type `ostream`, declared in the header file `iostream.h`. An operator << is overloaded in this class to perform output conversions

C++

from standard types
  std::cout << "Hello world!" << std::endl;
Input is inserted in `istream`, and operator >> is overloaded to perform input conversions to standard types.
  std::cin >> i;
Here "`using namespace std;`" would allow using these commands without the scope resolution operator.
File i/o is handled by including `fstream.h`. To read a variable *var* from a file, first open the file:
ifstream inFile("filename",ios::in);
inFile >> var;
A while loop can be made to read until EOF is reached:
while(!inFile.eof()){}
It is also possible to read a whole line using
`getline(ifstream,string)`.

Example: MET from high pt objects
File SimpleMET.h:

```
#ifndef SIMPLEMET_H
#define SIMPLEMET_H

#include "MyEvent.h"

class SimpleMET {
  public:
    SimpleMET();
    ~SimpleMET();

    void Add(MyTrack);
    void Add(MyJet);

    MyMET GetMET();
    double Value();
```

C++

```
    private:
      MyMET etmiss;
};
#endif
```

File SimpleMET.cc:

```
#include "SimpleMET.h"

SimpleMET::SimpleMET(){
      etmiss.x = 0;
      etmiss.y = 0;
}
SimpleMET::~SimpleMET(){}

void SimpleMET::Add(MyTrack
track){
      etmiss.x -= track.Px();
      etmiss.y -= track.Py();
}

void SimpleMET::Add(MyJet jet){
      etmiss.x -= jet.Px();
      etmiss.y -= jet.Py();
}

MyMET SimpleMET::GetMET(){
      return etmiss;
}

double SimpleMET::Value(){

      return etmiss.Value();
}
```

Object Creation and Destruction

An object requires memory and some initial value. C++ provides this through declarations that are definitions.

```
int n = 5;
```

The int object n gets allocated off the run-time system stack, and initialized to the value 5.

In creating complicated aggregates, the user will expect similar management of a class defined object. The class needs a mechanism to specify object creation and destruction, so that a client can use objects like native types.

A *constructor* constructs values of the class type. It is a member function whose name is the same as the class name. This process involves initializing data members and, frequently, allocating free store using *new*.

A *destructor* is a member function whose purpose is to destroy values of the class type. It is a member function whose name is preceded by the tilde character.

Constructors can be overloaded. A constructor is invoked when its associated type is used in a definition.

Destructors are invoked implicitly when an object goes out of scope.

Constructors and destructors do not have return types and cannot use

return statements.

```
class Example {
  public:
    Example();
    Example(int);

   ~Example();
};

Example e;
Example * ep = new Example;
delete ep;
Example e(5);
Example * ep = new Example(5);
```

A constructor requiring no arguments is called the default constructor. This can be a constructor with an empty argument list or a constructor where all arguments have default values. It has the special purpose of initializing arrays of objects of its class.

If a class does not have a constructor, the system provides a default constructor. If a class has constructors, but does not have a default constructor, array allocation causes a syntactic error.

Constructor initializer is a special syntax for initializing subelements of objects with constructors. Constructor initializers can be specified in a comma-separated list that follows the parameter list and precedes the body. Assuming the class Example has a data member i:

```
Example::Example() : i(0);
```

C++

Constructors of a single parameter are automatically conversion functions unless declared with the keyword *explicit*.

```
Example::Example(int);
```

This is automatically a type conversion from int to Example. It is available both explicitly and implicitly:

```
Example e;
int i = 123;
e = static_cast<Example>(i);
e = i;
```

With explicit keyword the conversion can be disabled. The constructor would be

```
explicit Example::Example(int);
Example e = 123; // illegal
Example e = Example(123); // ok
Example e(123); // ok
e = 124; // illegal
e(125); // illegal
```

A *copy constructor* is called whenever a new variable is created from an object. If there is no copy constructor defined for the class, C++ uses the default copy constructor which copies each field.

```
Example e(0); // constructor to build e
Example f(e); // copy constructor to build f
Example f = e; // copy constr., init. in decl.
```

If shallow copies are ok, dont write copy constructors, but use the default.

C++

Shallow copy is a copy where a change to one variable would affect the other. If the copy is in a different memory location, then it's a deep copy.

i = e.i; // shallow copy
Example e,f; f = e; // deep copy

If you need a copy constructor, you most probably also need a destructor and operator=

The syntax for the copy constructor is

Example::Example(const Example&){...}

Inheritance

Inheritance is a mechanism of deriving a new class from an old one. An existing class can be added to or altered to create the derived class.

C++ supports virtual member functions. They are functions declared in the base class and redefined in the derived class. A class hierarchy that is defined by public inheritance creates a related set of user types, all of whose objects may be pointed by a base class pointer. The appropriate function definition is selected at run-time.

Inheritance should be designed into software to maximize reuse and allow a natural modelling for problem domain.

C++

A class can be derived from an existing class using the form
class name :
(public|protected|private) bclass{};

The keywords public, protected and private are used to specify how the base class members are accessible to the derived class. The keyword protected is introduced to allow data hiding for members that must be available in the derived classes, but otherwise act like private members.
See also:
www.parashift.com/c++-faq-lite/strange-inheritance.html

A derived class inherits the public and protected members of a base class. Only constructors, its destructor and any member function operator=() cannot be inherited.

Member functions can be overridden.

Virtual functions in the base class are often dummy functions. They have an empty body in the base class, but they will be given specific meanings in the derived class. A pure virtual function is a virtual member function whose body is normally undefined.

virtual $< type > func\_prototype() = 0;$

Templates allow the same code to be used with respect to different types.

The syntax of a template class

declaration is prefaced by
template < class *identifier* >

Example:

```
template <class T> class stack {...};
stack<char*> s_ch;
stack<int> s_i;
```

This mechanism saves us from rewriting class declaration where the only variation would be the type declaration.

Function templates can be used for functions which have the same body regardless of type.

Example:

```
template <typename T> T myMax
(T x, T y) {return (x > y) ? x : y;}
cout << myMax<int>(3, 7) << endl;
```

Containers and Iterators

Container classes are used to hold a large number of individual items. Many of the operations on container classes involve the ability to visit individual elements conveniently.

Useful containers in C++ are e.g.

- vector
- pair
- map
- list
- queue
- stack
- set

Containers come in two major families: sequence and associative.

C++

Sequence containers include vectors and lists; they are ordered by having a sequence of elements. Associative containers include sets and maps, they have keys for looking up elements.

An iterator is a pointer which is used for navigating over the containers.

Example:

```
vector<int> intVec;
intVec.push_back(123);
intVec.push_back(456);
vector<int>::const_iterator i;
for(i = intVec.begin();
     i != intVec.end(); i++){
   cout << *i << endl;
}
```

Example:

```
map<string,int> cut;
cut["pt"] = 20;
cut["deltaPhi"] = 175;
if(track.pt() < cut["pt"] ) continue;
map<string,int>::const_iterator i =
find("pt");
```

Example:

In function tag() :

```
return pair<JetTag,Coll>(tag,tauIp);
```

Using function tag() :

```
pair<JetTag,Coll> ip = algo.tag();
JetTag tag = ip.first;
Coll c = ip.second;
```

Polymorphism

Polymorphism is a means of giving different meanings to the same message. OO takes advantige of

polymorphism by linking behavior to the object's type.

Overloading a function gives the same function names different meanings.

Overloading operators gives them new meanings.

The overloaded meaning is selected by matching the argument list of the function call to the argument list of the function declaration.

Example: filling 1D and 2D histograms

```
  void
MyHistogram::fill(string,double);
  void MyHis-
togram::fill(string,double,double);
```

When an overloaded function is invoked, the compiler selection algorithm picks the appropriate function. It depends on what type conversions are available. An exact match is the best. Casts can be used to force such a match.

Overloading operators allows infix expressions (operator between operands) of both ADT's and built-in types to be written. It is an important notational convenience, and in many instances leads to shorter and more readable programs.

Operators can be overloaded as non-static member functions.

C++

Example: An operator for matrix*vector

```
Vector Vector::operator*(
        const Matrix& m,
        const Vector& v){...};
Vector v; Matrix m;
Vector result = m * v;
```

Example: Two dimensional point addition

```
Point Point::operator + (const
Point& p) const{
    Point point;
    point.x = x + p.x();
    point.y = y + p.y();
    return point;
}
```

Example: MyVertex class inheriting MyGlobalPoint

```
class MyGlobalPoint {
  public:
    MyGlobalPoint();
    ~MyGlobalPoint();

    double getX() const;
    double getY() const;
    double getZ() const;

    double getXerror() const;
    double getYerror() const;
    double getZerror() const;

    double value() const;
    double error() const;
    double significance() const;
    double getPhi() const;

    double MyGlobalPoint operator +
    (const MyGlobalPoint&) const;
    double MyGlobalPoint operator -
    (const MyGlobalPoint&) const;
  protected:
    double x,y,z,
```

```
                dxx,dxy,dxz,
                    dyy,dyz,
                        dzz;
};

#include <vector>
#include "MyTrack.h"

using namespace std;

class MyVertex : public
MyGlobalPoint{
  public:
    MyVertex();
    ~MyVertex();

    double Eta() const;
    double Phi() const;
    double eta() const;
    double phi() const;

    MyVertex operator + (
        const MyVertex&) const;
    MyVertex operator - (
        const MyVertex&) const;

    vector<MyTrack> tracks();

  private:
    vector<MyTrack> assocTracks;
};
```

Example: Saving events in a ROOT file

File MyRootTree.h:

```
#ifndef MYROOTTREE_H
#define MYROOTTREE_H

#include "TROOT.h"
#include "TFile.h"
#include "TTree.h"
#include "MyEvent.h"
```

C++

```
class MyRootTree {

  public:
    MyRootTree();
    ~MyRootTree();

    void fillTree(MyEvent*);

  private:
    TTree* rootTree;
    MyEvent* myEvent;
    TFile* rootFile;
};
#endif
```

File MyRootTree.cc:

```
#include "MyRootTree.h"
MyRootTree::MyRootTree(){
  rootFile = new

TFile("analysis.root","RECREATE");
  rootTree = new
    TTree("rootTree","events");
  myEvent = new MyEvent();
  int bufsize = 256000;
  int split = 1;
  rootTree->Branch("MyEvent",
    "MyEvent",
    &myEvent,bufsize,split);
}
MyRootTree::~MyRootTree(){
  rootFile->cd();
  rootTree->Write();
  rootTree->ls();
  rootFile->Close();
  delete rootFile;
}
MyRootTree::fillTree(MyEvent*
event){
  myEvent = event;
  rootFile->cd();
```

C++

```
  rootTree->Fill();
}
class MyEventAnalyser : public
edm::EDAnalyzer {
...
   private:
...
      MyRootTree* userRootTree;
};

void MyEventAnalyser::beginJob() {
...
  userRootTree = new MyRootTree();
}

void MyEventAnalyser::analyze(const
edm::Event& iEvent){
// Event reconstruction here,
// reconstructing primary
// vertex, electrons, muons, etc.

  MyEvent* ev = new MyEvent;
  ev->primaryVertex = PV;
  ev->electrons = recElectrons;
  ev->muons = recMuons;
  ev->mcParticles = mcParts;

userRootTree->fillTree(saveEvent);

void MyEventAnalyser::endJob() {
  delete saveEvent;
  delete userRootTree;
}
```

The MyRootTree type pointer is added in the MyEventAnalyser class definition, in the initialization the MyRootTree class object is created, in the event loop the TTree is filled, and when the destructor is called (delete MyRootTree class object), the TTree is stored in a ROOT file.