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);
```

C++

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);
```

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.

C++

```
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. 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.

A class can be derived from an existing class using the form

```
class name : (public|protected|private) bclass{};
```

C++

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 $< type > function\_prototype() = 0;$

Templates

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 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.

```
template<class T1, class T2>
void copy(T1 a[], T2 b[], int n){
  for(int i = 0; i< n; i++) a[i] = b[i];
}
```

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. 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.

```
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;
}
```

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

In function tag() :

```
return pair<JetTag,Collection>(theTag,tauIp);
```

Using function tag() :

```
pair<JetTag,Collection> ipInfo = algo.tag();
JetTag tag = ipInfo.first;
Collection c = ipInfo.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 MyHistogram::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.

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;
  }
```

C++

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> associatedTracks;
};
```

C++

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"

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();
  rootTree->Fill();
}
```

C++

```
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* saveEvent = new MyEvent;
   saveEvent->primaryVertex = PV;
   saveEvent->electrons = recElectrons;
   saveEvent->muons = recMuons;
   saveEvent->mcParticles = mcParticles;
   userRootTree->fillTree(saveEvent);
   delete saveEvent;
}
```

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

In CMSSW the reconstruction can be done e.g. using a class derived from EDAnalyzer. The MyRootTree type pointer is added in the MyEventAnalyser class definition, in the initialization (beginJob) the MyRootTree class object is created, in the event loop (analyze) the TTree is filled, and when the destructor is called (delete MyRootTree class object), the TTree is stored in a ROOT file.