C++: notes

January 19, 2016

Contents

Part I
Language

1 Basics

1.1 Compilation

```
c++ -Wall -o exec_name file_list
```

`-Wl,--no-as-needed` loads everything even if doesn't needed. For example, useful when the dependence is not explicit.

With Valgrind `c++ -Wall -g3 -ggdb -gdwarf-3 -fsanitize=address -o exec_name file_list`

1.2 Types

- signed integers: `int, short, long, long long`

- unsigned integers
- enumerators `enumerators`
- floating point
- characters
- C-strings
- logicals
- `enum`, C++11`enum class`

1.2.1 Declaration

- Unmodifiable: `const`
- C++11`constexpr` known at compile time
- C++11`auto, decltype`
- `typedef /type/ /name/` (useful to modify by changing just one line)

1.2.2 Type cast

- C-style cast: `i=(int)x; j=int(y)`
- `i=static_cast<int>(x)`
- `*const_cast<int*>(p)=2;` force the modification of a non-`const` variable through a pointer to `const`.
- `int* pi = reinterpret_cast<int*>(pf);` no checks.

1.3 Operators

- unary
- binary
- ternary
- others: `(), new, delete, sizeof(), ...`

Precedence and associativity table. Priority:

```
++ --;
* / %;
+ -;
< > <= >= == !=;
= *= /= %= += -=
```

Division between integers is an `int` **!!**

`k=(i+3)*(++i);` is undefined `float x=5.7/(j=i);` is defined: **assignment operators are expressions !!**

1.3.1 Logical and bitwise operators

- logical (decreasing priority): `& ^ && || &= ^= |=`
- bitwise (decreasing priority): `<< >> <<= >>=`

`if(((i*i)<0)&&((j+=2)>10))` j not incremented

1.4 Flux control

Conditional statements: `if` Loop statements: `for, while, do while`; instructions: `continue, break`. Choice statements: `switch (int-expr) list-of-cases` .

2 Functions

2.1 `main` function

Returns an integer: 0 for no errors `main(int argc,char* argv[])` words in the command line.

2.2 Recursive functions

```
unsigned int fact(unsigned int n) \{
 if(n)return n*fact(n-1);
 return 1;
\}
```

2.3 `inline` functions

Not possible for recursive functions.

2.4 Arguments

Pass by value, reference, pointer. Overloading possible.

2.4.1 Default

`int f(int i,int j=1,int k=2);` If an argument has a default value, all the following ones must have one.

2.5 Predefined functions

Mathematical functions (in math.h): sqrt(double) , pow(double,double) sin(double) , acos(double) , ... atan2(double,double) exp(double) , log(double) fabs(double) : abs. value lround(double) , llround(double) : rounding Add a trailing l to use with long doubles Utility functions (in stdlib.h): random() : random int between 0 and 2 31 1 (RAND_MAX0x7fffffff) srandom(unsigned int) : set the seed for the random generation exit(int) : stop the execution immedately

3 Pointers, arrays, references

3.1 Pointers

A pointer is the memory address of the object it points to.

```
int* p=\&i; // "p" is the address of "i"

*p=24; // "i" is now 24

p=&j; // "p" is now the address of "j"

const int* p=&i;

void* p=&i // Requires reinterpret_cast
```

A null pointer (`=0`) is always invalid (C++11`nullptr`)**!!**
In declaration `int* p, q` just p is pointer.

3.1.1 Arrays

```
*p      i[0] , *(p + n)      i[n] , p + n      \&i[n]
```

C++11**array loop** `for (int& j:  i) j=2*k++;`
Prevention of "*narrowing*": `int j=43.1;` gives an error.
C++11initialization: `int j[3]{14,25,37};` .
Examples

```
Particle**
const Particle* const * partList
```

C-strings are arrays of `chars`, with a `\0` as last element.

3.2 References

Can be seen as a new name for an existing variable or object. Actually theyre pointers, with the * embedded.

```
int* i = new int(3);       //"i" is a pointer to an int whose value is "3"
float* f = new float[12]; // "f" is an array of 12 float
delete, delete[]
```

Caution Only pointer or reference to persistent objects can be returned**!!**

```
int* f(int i) {
  int j=i*2; // local variable, destroyed
             // when "f" returns
  return &j; // unvalid pointer returned
}
```

pointer/reference to const declared as `const int* p=&i;`: variable not modifiable through it.

pointer to void Can contain address of any variable or object. Cannot be dereferenced, deleted. Has to be used with a `reinterpret_cast`.

Dynamic memory handling `new, delete` operators; dynamic variables are not bound to a scope. `float* f = new float[12];` When pointers go out of scope: **memory leak**. When pointers are empty, but still existing: **dangling references** and a second `delete` operation cannot be performed. Deleting a null pointer has no effect.

4 Input-output

4.1 To/from file

```
#include <fstream>
std::ifstream file("inputfile");
```

```
std::ifstream file( "inputfile", std::ios::binary );
file.read( reinterpret_cast<char*>(&i), sizeof(i) );
```

5 Linkage and storage

5.1 Extern linkage

Variable declared outside all functions has external linkage. To use it: `extern int i;`

5.2 Libraries

5.2.1 Static libraries

```
c++ -c file2.cc
ar -r libTestS.a file2.o
c++ -o exec file1.cc -L. -lTestS
```

And include definitions everywhere And include guards:

```
#ifndef Lib.h
#define Lib.h
#endif
```

6 Composite types

6.1 Classes

6.1.1 Interface

```
public:    // can be accessed by anything
protected: // can be accessed by the same class and the derived ones
private:   // can be accessed only from the class

friend classes
```

Constructors and destructors `function(...) const;`

```
Point::Point(float cxi, float yi):
  xp(cxi),
  yp(yi)\{
\}
Point::~Point()\{
\}
```

Arguments initialized in declaration order **Default constructor**: constructor for each member. **Default deconstructor**: deconstructor for each member. **Default copy and assignment**: simply copied. Default only if not otherwise defined.

Methods Function members: direct access to member data of the object.

```
float Point::dist() const\{
  return sqrt(pow(xp-p.xp,2) + pow(yp-p.yp,2));
\}
```

If using another class, header is `class Name;`.

Shared members `static` (different meaning!) Initialization: `float Line::tolerance=1.0e-05;`
Pointer `*this` to the class itself

`mutable` : modifiable by a `const` function calling a const function modifying the variable.

Operators `Vector2D operator+(const Vector2D& v);`
With write/read objects:

```
std::ostream\& operator<< (std::ostream\& os,
                           const Vector2D\& v) \{
   os << v.getX() << "␣" << v.getY();
   return os;
};
```

Type conversions Implicit vs. `explicit`. With operators (C++ explicit cast).

Deep/shallow copy Default: shallow copy. Does not prevent "double free". Empiristic rule: deep copy necessary when destructor declaration is necessary. Lowest-level protection: prevent object copying putting under `private`

```
private:
 FloatArray(const FloatArray& a);
 FloatArray& operator=(const FloatArray& a);
```

or write `= delete` after declaration (**C++11, better**). **Deep copy**: declaring `public` when copies are necessary

```
FloatArray(const FloatArray& a);
FloatArray& operator=(const FloatArray& a);
private:
void copy(const FloatArray& a);

void FloatArray::copy(const FloatArray& a) \{
if(cont==a.cont)return; // skip "a=a" cases
delete[] cont; // delete old content
cont=new float[eltn=a.eltn];
// copy size and allocate new memory
float* pr=a.cont+eltn; // copy elements
loat* pl= cont+eltn; // one by one
while(pl>cont)*--pl=*--pr;
return;
\}
FloatArray::FloatArray(const FloatArray& a):
cont(0) \{ // set "cont" at 0 so that a
copy(a); // "delete[]" has no effect
\}
const FloatArray& FloatArray::operator=(
const FloatArray& a) \{
copy(a);
return *this; // return the just-copied
\}
// object as value
```

6.2 Namespaces

```
namespace Geom {
 class Line;
 class Point {
  ...
 };
};
```

`typedef Geom::Point point;`
`using Geom::Point;`
`using namespace Geom;` Do not use it in header files: it affects all the translation unit

6.3 Exceptions

`throw try catch`

7 Templates

```
  template<class T>
  void perm(T& x,T& y,T& z) \{
   // make a cyclic permutation of x,y,z
   T t = x;
   x = y;
   y = z;
   z = t;
   return;
  \}
```

or `template<typename T>`. **Full definition** before usage!

7.1 Class templates

```
  template<class T> // T replaces "float"
  class Array \{
  // everywhere
  public:
  // in the class definition
  Array(int n);

  ~Array();
  ...
  private:
  int eltn;
  T* cont;
  void copy(const Array<T>& a);
  \};
```

The implementation must come before the class is used: its usually coded in implementation files (.hpp) included at the end of the header. Construction: `Array<float>`
Nested classes When calling from outside, use `typename A<T>::C`

```
void print() {
 return;
}

template<typename T>
void print(T t) {
     //prints t
}

template<typename T, typename... R>
void print(T t, R... r) {
 cout << t <<"␣";
 print(r...);
 return;
}
```

```
template<class T, unsigned int N>
FixedSizeArray<T,N>
```

Template specialization In addition to the generic one :. template¡¿ Array¡int¿::Array(int n): eltn(n), cont(new int[eltn]) cout ¡¡ "create an Array of int" ¡¡ endl; ..

8 Standard Template Library

8.1 Strings

Creation:

```
string s("...");
```

Initialization: `string s = charvect` makes a copy. Conversion in C-string:

```
const char* c = s.c_str();
```

Copy (=) and concatenation (+). Compare (==).

```
s.insert(n, "...")
```

Get m characters from the nth `s.substr(n,m)` `s.length()` `s.find("...")` returns the number of characters before the substrings, or `string::npos` if not found.

```
std::stringstream s;
s.clear();
s.str("12");
s >> i;
```

9 Containers

- Sequences: contents stored in sequential order.
  - Queues: contiguous memory location, random access supported, linear time to access data.
  - Lists: data linked to the previous/next one, random access not supported, constant time to access data.
- Associations: contents stored with a key.
  - Sets
  - Maps

9.1 Vectors

`std:vector` Index `n` is an `unsigned int`!

Methods Constructors:

```
vector<T> v // create an empty vector
vector<T> v(n)// create a vector with n elements
vector<T> v(n,t)//
create a vector with n elements initialized at t
vector<T> v(u)// create a vector by copying vector u
v[i]// reference to element at position i (unchecked range)
v.at(i)// reference to element at position i (checked range: if empty returns an exce
v.reserve(n)// allocate memory for n elements
v.push\_back(t)// add an element at the end. !!Every push back over the allocated mem
v.pop\_back()}: remove an element at the end
v.resize(n)} , \code{v.resize(n,t)// set the number of
elements by adding or removing elements at the end and
leaving unchanged the others
v.clear()// remove all elements (memory remains allocated)
v.size()// number of elements (returns just the non-empty allocated memory)
v.capacity()// available memory
v.empty()// true if the number of elements is 0
v.front()// reference to the first element
v.back()// reference to the last element
```

```
v[i]// reference to element at position i
(unchecked range)
v.at(i)// reference to element at position i
(checked range)
```

```
vector<T>::iterator i=v.begin() // an iterator pointing to the first element
vector<T>::iterator i=v.end() // an iterator pointing to the next-to-last element
*i // reference to the pointed element
++i  i++ // move the iterator to the next element
--i  i-- // move the iterator to the previous element
vector<...>::iterator ii=it+n //create an iterator ii pointing n positions after it
vector<...>::iterator ii=it-n //create an iterator ii pointing n positions before it
it+=n  it-=n //move forward or backward of n positions
int d = distance(ip,in) //compute how much ip must be advanced to reach in
advance(it,n) // move it forward of n positions
v.insert(it,t) // insert the element t at the position it
v.insert(it,ib,ie) // insert the elements pointed by [ib,ie) at the position it
v.erase(it)  v.erase(ib,ie) // erase the element(s) pointed by it or [ib,ie) after an
the sequence are invalidated
const vector<T> v // a vector whose size and elements cannot be modified
vector<T>::const_iterator // analogous to  pointer to const
vector<T>::reverse_iterator // allow the scan in the backward direction
v.rbegin() , v.rend() // begin and end of the reversed vector
```

9.2 Lists

```
#include <list>
```

9.3 sets

```
#include <set>
```

```
Elements automatically ordered.

%%%%
\subsection{\code{map}s}
\code{\#include <map>}
\begin{lstlisting}map<Key,T> m(comp)
```

Each element is a

```
\code{first} with type Key
\code{second} with type T
\begin{lstlisting}
m.insert(make_pair(k,x)) //insert x with key k

m[k]; //returns/create a reference to element whose key is k
```

9.4 Smart pointers

```
#include <memory>
auto_ptr<A> a(new A)
```

Each copy gets the possession of the object pointed. Deprecated in C++98, better variations in C++11:

```
 unique_ptr shared_ptr
```

.

10 Algorithms

```
#include <algorithms>
```

10.1 Sorting

$\mathcal{O}(N \ln(N))$

Using '¡' `sort(v.begin(),v.end());`

Using '()' (to be defined) `sort(first,last,comp);`

Using lambda function :

```
sort(first,last,
             [](Vector2D* vl,Vector2D* vr){
                return (pow(vl->getX(),2)+
                pow(vl->getY(),2))<
                (pow(vr->getX(),2)+
                pow(vr->getY(),2));
              });
```

Variables in the environment can be captured: `[]` capture nothing `[&]` capture all by reference `[=]` capture all by value `[=,&i]` capture all by value, but i by reference

10.2 `sort`

```
sort(v.begin(), v.end()
```

(`last` is actually one after the last one). **Using function object**

```
sort(first, last, comp)
```

(using the `comp::bool operator()` operator of the class). **Lambda functions** can also be used:

```
       sort(first,last
                     [] (Vector2D* vl, Vector2D* vr){
                                                ... });
```

10.3 Binary search

```
iter=lower_bound(first,last,i)
```

: the first element such that it and the following ones are not smaller than i

```
such that it and the following ones are bigger than i
\begin{lstlisting}
iter=lower_bound(first,last,i,comp)
iter=upper_bound(first,last,i,comp)
```

11 Inheritance and polymorphism

11.1 Inheritance

Public inheritance: new class can be used anywhere in place of its base.

```
class Vector3D: public Vector2D{ ... };

Vector3D u(3.4,4.5,7.2);
Vector2D v= u; //"slicing"
```

The declaration of a function in a derived `class` hides all the functions with the same name in the base `class` , also the ones with different parameters.

```
using vector2D::transform(...) //C++11 only
```

11.2 Polymorphism

Functions of a base class can be declared to be automatically replaced by functions of the derived class even when called through the base class interface Functions behaving in this way are called `virtual` functions. The "virtual table" used in the call of virtual functions inside the base class functions, too.

```
class Shape {
 public:
 ...
 virtual float perimeter() const;
 virtual float area() const;
 ...
};
class Triangle: public Shape {
 public:
 ...
 virtual float perimeter() const; //"virtual" is not mandatory
```

```
 virtual float area() const;
 ...
};

Shape* t = new Triangle(...);
```

Call by pointer: slicing

```
const Shape s = *t; //doesn't keep Triangle structure
```

Pure `virtual` functions Functions to be necessarily (re)implemented in derived classes.

```
virtual float perimeter() const = 0;
```

```
Shape* s;
...
Square* q = dynamic_cast<Square*>(s);
if(q!=0)cout << q->side() << endl;
```

If failed, returned a null pointer.

Construction and destruction sequence The base class constructor is run before the derived class constructor. The base class destructor is run after the derived class destructor. **!!**The base class constructor and destructor cannot call the derived class functions!

Multiple inheritance In case of ambiguity:

```
Derived* d;
...
d->f(); // ambiguous
d->BaseA::f();
```

A common base can be declared to be shared by all its derived classes: "direct base" for all derived classes in the inheritance chain.

```
class Base {
...
};
class IntermediateA: public virtual Base {
...
};
class IntermediateB: public virtual Base {
...
};
```

Part II
Design patterns

12 Creational

12.1 Singleton

- can be instantiated only once
- instance globally accessible
- usually created at its first use

```
class ObjS {
 public:
  static ObjS* instance();
  ...
 private:
  ObjS();
  ~ObjS();
  ObjS(const ObjS& x);
  ObjS& operator=(const ObjS& x);
};
```

"Pointer saving" implementation When an object, derived from `Base`, is created and `obj` is null, its pointer is stored in `obj`.

```
Base::Base() {
  Base*& i=instance();
  if(i==0)i=this;
}
Base*& Base::instance() {
  static Base* obj=0;
  return obj;
}
```

12.2 Factory

A Factory is a class that creates an instance of another class from a family of derived classes.

- a (usually static) function returning a pointer to a base class
- The client does not depend on the derived classes.
- The client needs not knowing all the informations needed to create the objects.

```
class ShapeFactory {
 public:
  static Shape* create(...);
 ...
};
```

Abstract factory An Abstract Factory is an interface to a concrete Factory, so that the objects actually created depend on the actual factory object.
The function to create objects is declared virtual: to be reimplemented in every factories.

12.3 Builder

A Builder is a class that creates complex objects step by step Used to encapsulate the operations needed to create a complex object. A Builder has functions to specify how the objects is to be built, and a function to actually create the result. A Builder usually specify only an interface, while a Concrete Builder actually performs the operations. The client can create different objects by using similar operations.

12.4 Prototype

A Prototype is a class that creates new objects by cloning an initial object
The object to clone can be obtained from a manager.
As with Factory, the client need not knowing all the concrete object types.
The Prototype manager can allow the registration of new objects at runtime.
New objects can be registered by loading new dynamic libraries at runtime by using the dlopen function.

```
#include <dlfcn.h>
...
 void* p=dlopen("libName.so",RTLD_LAZY);
...
```

13 Structural

14 Behavioural

Part III
Root

15 Root language

```
~> c++ -Wall   root -config --cflags   \
? -o prog prog.cc   root -config --libs
~> setenv LD_LIBRARY_PATH \
? ${LD_LIBRARY_PATH}":${ROOTSYS}/lib"
```

15.1 Histograms

Histograms are object of type `TH1F`

```
TH1F* h=new TH1F(name,title,
nbin,xmin,xmax);
```

Use C-strings.

```
h->Fill(x) //fills the histogram
int n=h->GetNbinsX() //gives the number of bins.
float c=h->GetBinContent(i); //gives the content of bin i :
                     //i=0 gives the underflow content,
                     //i=n+1 gives the overflow content.
float e=h->GetBinError(i); //gives the error on content of bin i
h->SetBinContent(i,c); //set the content of bin i at c ;
h->SetBinError(i,e);   //set the error on content of bin i at e ;
int i=h->FindBin(x);   // gives the bin whose interval contains x .
```

To store histograms:

```
TDirectory* currentDir=gDirectory;
TFile* file=new TFile(name,mode); //C-string;
          /*mode="CREATE" or "NEW" : create a new file and open it for writing; if th
                 "RECREATE" :create a new file and open it for writing; if the file a
                 "UPDATE" open an ecxisting file for writing; if the file does not ec
                 "READ" (default) open an ecxisting file for reading.*/
h->Write();
delete file;
currentDir->cd();
```

To retrieve histograms:

```
TDirectory* currentDir=gDirectory;
TFile* file=new TFile(f_name,mode);
```

```
currentDir->cd();
TH1F* h = dynamic_cast<TH1F*>(
file->Get(h_name)->Clone() );
delete file;
```

16 Sources

Profs. Garfagnini, Ronchese