Scuola Superiore Sant'Anna

Real-Time System Laboratory

Advanced course on C++

Giuseppe Lipari

Scuola Superiore S. Anna

Introduction

Perfection is achieved only on the point of collapse

- C. N. Parkinson

What is C++

- A programming language, built on top of C
- It provides:
  - object oriented programming
  - generic programming
- It is very different from C !
  - Strongly typed
  - Very powerful (sometimes too much)

Features

- C++ is an extensible language
- The user can define
  - new types (classes)
  - generalizations (with templates)
- The user can also re-define
  - operators
  - memory allocation strategies
  - … and many more features

Complexity

- C++ is a complex language
  - it is considered a *difficult* language
  - it takes years of experience to be able to manage all the different aspects of the language
- Don’t be scared!
  - You don’t need to know every aspect of C++ to be able to start programming
  - You will probably never use some aspect of the language
  - I like C++ because it is challenging!!

Advices

- Focus on concepts; do not get lost in technical details.
- You don't have to know every detail of C++ to write good programs.
- Never say "I know C++ perfectly"; there is always something to learn.

From C to C++

"Don't reinvent the wheel:

use libraries"

- B. Stroustrup

Declarations and definitions

- A *declaration* introduces the name and the type of an "entity" to the compiler

```
int func(int a);                // function declaration (prototype)
```

  - a declaration does not imply a memory allocation

- A *definition* says to the compiler: here is the entity

```
int func(int a)
{
    return a+1;
}
```

  - a definition implies a memory allocation

Declarations and definitions

- Sometimes a declaration can also be a definition
- It happens with variables

```
extern int a;  // declaration
int a;         // definition and declaration
```

- In C/C++ *extern* means “this is just a declaration and not a definition
- For functions, extern is not needed

Functions

- Syntax

  ret_type   fun_name(arg1_type arg1_name,
                                   arg2_type arg2_name, …);

- Warning:

      int f();

  in C, it means   ->   int f(int arg);

  in C++, it means ->   int f(void);

- The return type is always mandatory!

Function declaration

- Function declaration is not essential in C.
  - It is possible in C to call a non-declared function. The compiler will “guess” the prototype
  - However, if the compiler make a wrong guess, only the linker can find the problem, maybe!!
  - Functions should always be declared in C!
- Function declaration is essential in C++
  - Using a non-declared function is a compiler error

Translation unit

- Each separate .c or .cpp file is a *translation unit*
  - It means that it is compiled separately from the other files to produce an *object file* (.o or .obj)
- So, whatever you declare in a .c or .cpp file is not visible to the compiler when it compiles other .c or .cpp files
- The *linker* will then put together all *object files* for making the *executable*
  - if there are inconsistencies, the linker can find them

Problems with C

- As said, in C function declaration can be omitted

```
int a;
int b;

b = func(a);
```

file1.c
The compiler *guesses* that there must be a function
int func(int);

```
void func(double) {
  ...
}
```

file2.c
The compiler produces code for a function
void func(double)
it does not know that another module will call a function
int func(int)

The linker is not able to find out this problem

This is a subtle error!

Header files

- All declarations related to a certain part of a program (module) are often collected in *header* files
  - Header files define the *interface* of a module
  - Especially useful for libraries
  - They are used with the *#include* < > directive

```
#include <stdlib.h>
#include “mymodule.h”
```

Using header files

- Headers are a way to ensure consistency in the declarations
- They also facilitate the documentation of a module, by collecting all interface definition in a single file

Variables in C++

- In C, variables must be defined global or at the beginning of a function definition
- In C++ they can be defined everywhere

```
double f(double b) {
    int i,j,k=0;
    …
    for (i=0; i<100; ++i) {
        j = i*4 - 1;
        if (j % i == 31) k = i+j;
    }

    return k * b;
}
```

```
double f(double b) {
    int k=0;
    …
    for (int i=0; i<100; ++i) {
        int j = i*4 -1;
        if (j % i == 31) k = i+j;
    }
    // we cannot use i or j here
    return k * b;
}
```

Scope

- The scope of an object is the piece of program between its declaration and the end of the block where the declaration is done

```
int x;                  // global x

void f() {
    int x;              // local x
    x = 1;              // assign to local x
    {
        int x;          // another x
        x = 2;          // assign to second local x
    }
    x = 3;              // assign to first local x
    ::x = 4;            // assign to global x
}
```

Boolean values

- In C, every expression that evaluates to 0 is considered false, every other expression is considered true
- In C++, there is a Boolean type and two Boolean constants:

```
bool var;
var = true;

if (var == false)    …
if (!var) …
```

Pointers

- A *pointer* is a variable that holds a memory address
- Pointers have type:

```
int *p;              // pointer to integer
double *p2;          // pointer to double
struct data *p3;     // pointer to structure
```

We can obtain the address of a variable using **&var;**

We can obtain the value of the memory location pointed by a pointer with ***p;**

```
int a = 5;
int *p = &a;

cout << “a = “ << a << endl;
cout << “*p = “ << *p << endl;
```

Array

- An array is a set of consecutive locations in memory

```
int arrayOfInt[10];                // 10 integers (from 0 to 9)
double arrayOfDouble[25];   // 25 doubles (from 0 to 24)
struct MyData {
  int a;
  int b;
} arrayOfStruct[50];               // 50 structs (from 0 to 49)
```

```
for (int i=0; i<10; ++i)
   arrayOfInt[i] = i*2;

cout << arrayOfStruct[7].a << " - " << arrayOfStruct[7].b << endl;
```

Array II

- The size of the array must be a constant expression

```
void f(int i)
{
    int v[i];                  // this is an error!!
    vector<int> v(i)    // ok
}
```

```
int v1[] = {1,2,3,4};
char v2[] = “Ciao ragazzi!”;
```

Array initializer

```
char v2[14];
v2 = “Ciao ragazzi!”;
```

No assignment!

Array and pointers

- The name of an array can be used like a constant pointer

```
void fun1(int *a);
void fun2(int a[]);
```

equivalent declarations

```
arrayOfDouble[5];
*(arrayOfDouble + 5);
```

equivalent expressions

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

for (int i=0; i<10; ++i, ++p)
   cout << *p << endl;
```

another way of going through an array

Pointers II

Pointers can be *incremented/decremented*. The number of location a pointer is incremented by depends on the pointer type

```
int *p;            // pointer to integer
double *p2;        // pointer to double
MyData *p3; // pointer to structure
```

```
p++;          // incremented by sizeof(int)
p2 += 2;      // incremented by 2*sizeof(double)
p3--;         // decremented by sizeof(MyData);
```

Structs

- A structure is a collection of variables

```
struct Entry {
   string name;
   string surname;
   int phone_number;
   string address;
};

Entry phone_book[1000];
```

There is a big difference btw C and C++ structs

- in C++ structs can also contain functions and operators. They are *almost* like classes

Structs

- To indicate a variable inside a struct, we use the .

```
Entry entry;

entry.name = “Giuseppe”;
entry.surname = “Lipari”;
entry.number = 1234;
entry.address = “Via Carducci, 40”;

phone_book[12] = entry;
```

We can also initialize a struct with {}

```
Entry entry = {“Giuseppe”, “Lipari”, 1234, “Via Carducci, 40”};
```

Re-declaration

- In C and C++, it is not allowed to declare a structure (and a class in C++) more than once
  - however, it is possible to declare functions more than once, if they match
- In a complicated program, however, it can happen that a header file is included twice
  - so, unexpectedly, a struct can be declared twice
- To avoid this problem, programmers use *guards*

Header file guards

- Suppose we have a myheader.h file:

```
#ifndef  __MYHEADER_H__
#define __MYHEADER_H__

…    // declarations here

#endif
```

This technique is also called *conditional inclusion*

Pointers to structs

- To reference a variable inside a struct with a pointer to the struct, use operator ->

```
Entry *p;
p = &phone_book[0];

for (int i=0; i<50; ++i,++p)
   cout << p->name << " - " << p->surname << endl;
```

Passing parameters to functions

- In C, we can pass parameters by value or by pointer

```
void my_func(int a, int *b)
{
    a += 5;
    *b = a+1;
}
```

i is *passed by value*: it is not modified by my_func

j is *passed by pointer*: it is modified by my_func

```
int i = 2;
int j = 3;

my_func(i, &j);
```

References

- In C++, there is another way of referencing variables

```
void my_func(int a, int &b)
{
   a += 5;
   b = a+1;
}
```

notice how b is declared !

i is *passed by value*: it is <u>not</u> modified by my_func

j is *passed by reference*: it is modified by my_func

```
int i = 2;
int j = 3;

my_func(i, j);
```

References

- A reference is *an alternative name* for an object
  - Another definition: *a pointer that is automatically de-referenced*

```
void f ()
{
    int i = 1;
    int &r = i;
    int x = r;        // now x = 1;
    r = 2;            // now i = 1;
}
```

```
int i = 1;
int &r1 = i;      // ok
int &r2;          // wrong !!!

r1 ++;            // now i = 2
```

A reference must always be initialized!

A reference <u>is not</u> a pointer!

References vs. pointers

- Pointers are more general
  – References have a clear syntax
- It is possible to have pointers to void: **void *p**
  – It is not possible to have references to void
- It is possible to do pointer arithmetic
  – No reference arithmetic

- Try to use references whenever you can!

References vs. Pointers II

- Another difference: structs (and classes)

```
void my_func(struct data *pd){
    pd->a = pd->b / 2;
    pd->b = pd->a + 10;
}
```

passing by pointer (C style)

passing by reference (C++-style)

```
void my_func(struct data &rd)
{
    rd.a = rd.b / 2;
    rd.b = rd.a + 10;
}
```

Pointers to functions

- The portion of memory where the code of a function resides has an address;
- we can define a pointer to this address:

```
void (*funcPtr)();          // pointer to  void f();
int (*anotherPtr)(int)     // pointer to    int f(int a);
```

Assigning

```
void f(){…};

funcPtr = &f();     // now funcPtr points to f()
```

```
(*funcPtr)();
```

invoking

Arrays of function pointers

- It is also possible to define arrays of pointers to functions:

```
void f1(int a) {…}
void f2(int a) {…}
void f3(int a) {…}
...
void (*funcTable []) (int) = {f1, f2, f3}
...
for (int i =0; i<3; ++i) (*funcTable[i])(i + 5);
```

Constants

- Constants in C

```
#define  PI   3.14159
```

There is no type checking!

Constants in C++

```
const double pi = 3.14159;
```

In C++ *const* is a *type modifier*

It is not only a directive, but “modifies” the meaning of the type, by saying “this cannot be changed”

A const must always have an initial value!

Using *const*

- *const* is often used when passing a parameter by reference;

```
int f(const MyClass &p);
```

- It means: variable p will not be modified by this function
  - In fact, passing a parameter by reference does not mean automatically that we want to modify it! Maybe we want just to save time and space…
  - There is no way to understand from the prototype if the function will modify the parameter or not, unless we use const. So, you should always use const if the function does not modify the parameter!

Casting

- Sometime we want to assign a variable of type T a value of another type

```
int a = 4;
double c = 3.5;

a = c;     // implicit casting     now a = 3;         compiler issues a warning
c = a;     // implicit casting     now c = 3.0;       compiler does not warn

bool b = (a < c);     // no casting involved

int b1 = (a == c);    // implicit casting             compiler does not warn
```

Explicit cast

- We can force an explicit cast with the () operator

```
int a = 4;
double c = 3.5;

a = (double) c;      // C style       no compiler warning
a = double(c);       // C++ style     no compiler warning
```

Cast between pointers:

```
struct MyData {
    double a, b;
};

MyData data;
void *p = &data;        // implicit casting     no compiler warning
```

Casting

- Casting is dangerous

```
struct MyData {
    double a;
    double b;
};

void *m = malloc(10);
MyData *p = (MyData *) m;      // explicit cast
                   // this is an error! m points to a memory buffer of 10 bytes;
                   // p points to a data structure of 16 bytes!
                   // soon, a segmentation fault...
```

There is no way for the compiler to check this problem

C++ cast operators

- static_cast<>
  - it is analogous to the old cast; it is easier to find in a program. For “safe” casts.
- const_cast<>
  - to get rid of the const type modifier
- reinterpret_cast<>
  - to cast to a completely different meaning; very dangerous!
- dynamic_cast<>
  - for type safe downcasting

A tour of the standard library

“No significant program is written in just a bare programming language.

First a set of supporting libraries are developed.

These then form the basis for further work”

- B. Stroustrup

Introduction

- Here we introduce the basic classes of the C++ std library
- You will need them when writing your programs and exercise
- Don’t panic: you don’t need to understand how these objects are implemented, but only how they can be used

A few words on namespaces

- In C, there is the *name-clashing* problem
  - cannot declare two entities with the same name
- One way to solve this problem in C++ is to use namespaces
  - A name space is a collection of declarations
  - We can declare two entities with the same name in different namespaces
  - All the standard library declarations are inside namespace std;

Using entities inside namespaces

- There are two ways:
  - Using the scope resolution operator ::
  - the *using namespace xx* directive

```
std::string a;       // declaring an object of type
                     // string from the std namespace

mylib::string b;     // declaring an object of type
                     // string from the mylib namespace
```

```
using namespace std;    // from now on use std

string a;               // declaring an object of type
                        // string from the std namespace
```

Basic input/output

```
#include <iostream>
int main()
{
   std::cout << “Hello world!”;
}
```

- Basic I/O function are declared within *iostream*
- *cout* is the standard output stream
- *std::cout* means that the *cout* object is contained in a namespace called *std::*
- all the *std* library is contained in *std*
- we can also use the *using* directive

Basic I/O

```
#include <iostream>
using namespace std;
int main()
{
   cout << “Hello world!\n”;
}
```

- operator << sends its right part to the stream to the left
- it can send many kinds of variables or constants:

```
int age = 30;

cout << “I am “ << age << “ years old\n”;
```

Basic I/O

```
#include <iostream>
using namespace std;
int main()
{
  int age;
   cout << “Enter your age:”;
   cin >> age;
   cout << “Next year your age will be “ << (age + 1) << “\n”;
}
```

- *cin* is used for input;
- operator >> can read many kinds of variables;

Strings

- the std library provides a string type

```
#include <string>
using namespace std;
int main()
{
   string s1 = “Hello”;
   string s2 = “world”;

   string s3 = s1 + “ “ + s2;

   cout << s3 << “\n”;
}
```

```
void respond(const string &answer)
{
    if (answer == “yes”) {...}
    else if (answer == “no”) {...}
    else cout << “Please answer y/n\n”;
}
```

Strings

- Some useful function with strings:

```
string name = “Giuseppe Lipari”;

void substitute() {
    string s = name.substr(9,6);
    name.replace(0,8, “Roberto”);     // name becomes “Roberto Lipari”
}
```

```
cout << name[0] << name[1] << name[2] << “\n”;    // prints “Rob”
```

```
void f() {
    printf(“name : %s\n”, name.c_str());
}
```

Strings

- String can be compared with std operators;
- The order is alphabetical

```
string a = "Peppe";
string b = "Gianni";
string c = "Gianni";

void cmp(const string &s1, const string &s2) {
    cout << s1;
    if (s1 == s2) cout << " == ";
    else if (s1 < s2) cout << " < ";
    else cout << " > ";
    cout << s2 << "\n";
}

cmp(a,b);       // prints   "Peppe > Gianni"
cmp(b,c);       // prints   "Gianni == Gianni"
cmp(c,a);       // prints   "Gianni < Peppe"
```

Input/Output and strings

- reading a word

```
int main () {
    string str;
    cout << “please, enter your name ”;
    cin >> str;
    cout << “Hello “ << str << “!\n”;
}
```

- reading the entire line

```
int main () {
    string str;
    cout << “please, enter your name ”;
    getline(cin, str);
    cout << “Hello “ << str << “!\n”;
}
```

Files

- An input file can be opened with *ifstream*
- then, it can be used as *cin*
- For output file, use *ofstream*, that can be used as *cout*

```
int main () {
    ifstream in(“input.txt”);
    ofstream out(“output.txt”);

    string str;
    while (in >> str)  out << str;
}
```

Containers: vector

- sometimes we do not know how many element an array will contain

```
struct Entry {
    string name;
    int number;
};

Entry phone_book[1000];

void print_entry(int i) {
    cout << phone_book[i].name << ‘ ‘ <<phone_book[i].number << “\n”;
}
```

what if phone_book overflows?

Containers: vector

- we can use the vector<Entry> container

```
struct Entry {
    string name;
    int number;
};

vector<Entry> phone_book(10);       // initially, only 10 elements

void print_entry(int i) {
    cout << phone_book[i].name << ' ' << phone_book[i].number << "\n";
}

void add_entry(const Entry &e) {
    phone_book.push_back(e);         // after 10 elements, expands automatically
}
```

Containers: vector

- What is the push_back() function?
  - inserts a new element at the end of the vector. If there is not enough space, the vector is enlarged
- How can we know the actual number of elements?
  - using the size() function

```
void add_entry(const Entry &e) {
    phone_book.push_back(e);        // expands automatically
    cout << “Now the numer of elements is “ << phone_book.size() << “\n”;
}
```

Containers: vector

- for efficiency reasons, operator [] is not checked for out-of-range
- however, we can use the function at() instead of []

```
// this causes a segmentation fault if i is out of range
void print_entry(int i) {
    cout << phone_book[i].name << ‘ ‘ << phone_book[i].number << “\n”;
}

// this throws an out_of_range exception
void print_entry_with_exc(int i) {
    cout << phone_book.at(i).name << ‘ ‘ << phone_book.at(i).number <<
“\n”;
}
```

First example

- We will write a program that:
  - reads a file line by line
  - stores each line in a vector;
  - outputs the file upside/down (from the last line to the first) into another file

Reading the command line

- A program can read the command line through its main function

```
int main(int argc, char* argv[])
{
  cout << "Num of args: " << argc << "\n";
  for (int i =0; i<argc; ++i)
    cout << argv[i] << "\n";
}
```

```
$> ./args joe 5.0 12 india
Num of args: 5
./args
joe
5.0
12
india
```

argc contains the number of args+ 1
argv[i] contains the i-th argument
argv[0] is always equal to the name of the program

Now the code...

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main(int argc, char *argv[])
{
  if (argc < 3) {
    cout << "Usage: ";
    cout << argv[0] << " <input file> <output_file>" << endl;
    exit(-1);
  }
  ifstream in(argv[1]);
  ofstream out(argv[2]);
```

Now the code...

```
…
vector<string> lines;

 string str;
 while (getline(in, str)) lines.push_back(str);

 int n = lines.size();
 cout << "The size of the input file is " << n << " lines\n";
 for (int i=n; i > 0; --i)
   out << lines[i-1] << endl;

 cout << "Done!!" << endl;

}
```

Containers: map

- what if we want to search the phone_book by name?
- we have to perform a linear search

```
int get_number(const string &name)
{
  for (int i=0; i<phone_book.size(); ++i)
     if (phone_book[i].name == name) break;

  if (i== phone_book.size()) {
    cout << “not found!!\n”;
    return 0;
  }
  else return phone_book[i].number;
}
```

Containers: map

- Another (more optimized) way is to use map<string, int>

```
map<string, int> phone_book;

void add_entry(const string &name, int number)
{
    phone_book[name] = number;
}

int get_number(const string &name)
{
    int n = phone_book[name];
    if (n == 0) cout << "not found!\n";

    return n;
}
```

Containers: map

- You can think of map<> as an associative array
  - in our example, the index is a string, the content is an integer
- How map is implemented is not our business!
  - Usually implemented as hash tree, or red-black tree
  - linear search in a vector is O(n)
  - searching a map is O(log(n))
- Very useful!!

Iterators

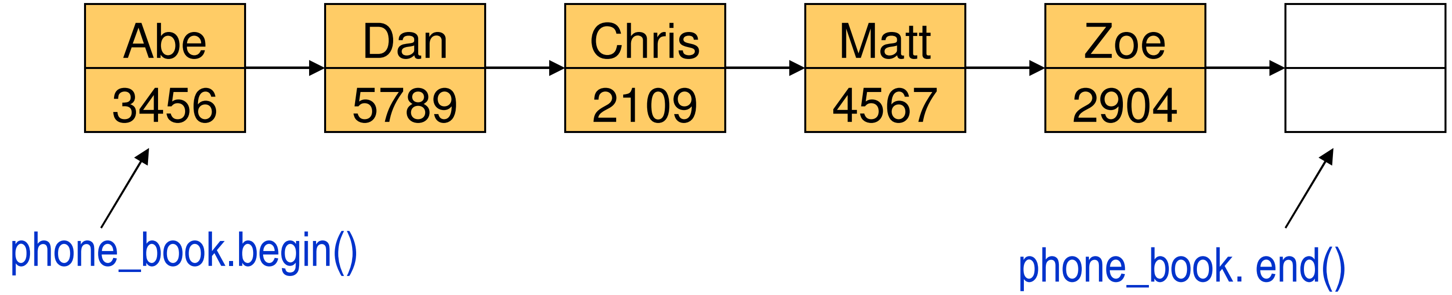
- What if we want to print all elements of a map?
- we need an iterator...

```
map<string, int> phone_book;

void print_all()
{
    map<string, int>::iterator i;

    for (i = phone_book.begin(); i != phone_book.end(); ++i);
        cout << “Name : “ << (*i).first << “ “;
        cout << “Number : “ << (*i).second << “\n”;
    }
}
```

What the ?@#$ is an iterator?

- An iterator is an object for dealing with sequence of objects inside containers
- You can think of it as a special pointer

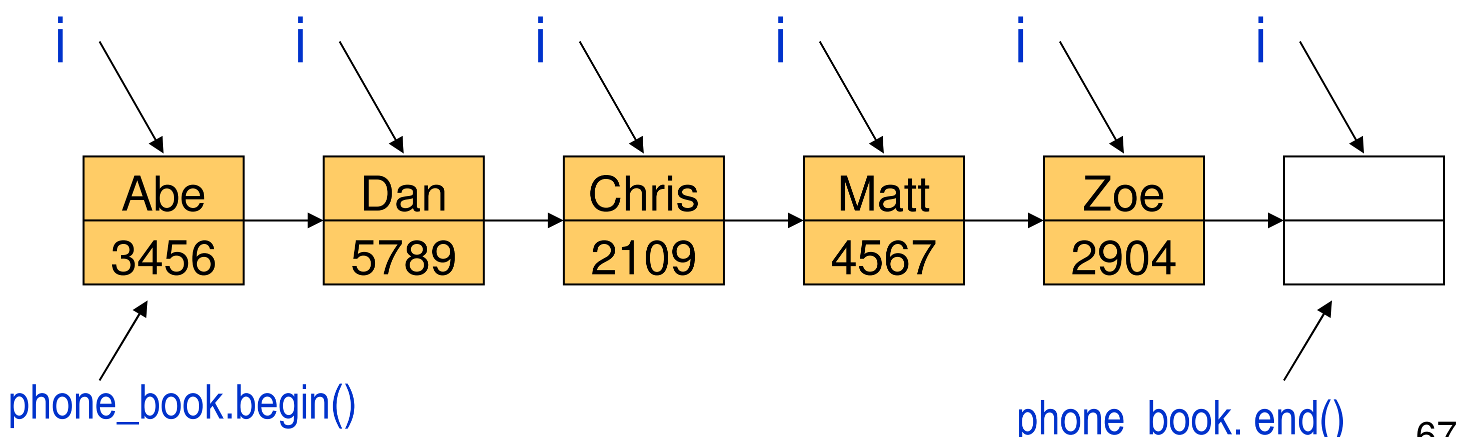
```
phone_book.begin();        // the beginning of the sequence
phone_book.end();          // the end of the sequence
```



Iterators

- Here is how the for() works:

```
void print_all() {
    map<string, int>::iterator i;
    for (i = phone_book.begin(); i != phone_book.end(); ++i);
        cout << "Name : " << (*i).first << " ";
        cout << "Number : " << (*i).second << "\n";
    }
}
```



Iterators

- There are iterators for all containers
  - vector, string, list, map, set, etc.
  - all support begin() and end()
- Iterators are also used for generic algorithms on containers
  - find, foreach, sort, etc.

sort()

- Let's get back to the vector example

```
struct Entry {
    string name;
    int number;
};

vector<Entry> phone_book(10);     // initially, only 10 elements
```

what if we want to order the entries alphabetically ?

In the old C / C++ programming, we would take a good book of algorithms (like "The art of computer programming" D. Knuth) and write perhaps a shell-sort

With the standard library, this has already been done by someone else and it is fast and optimized; all we have to do is to customize the algorithm for our purposes.

sort()

- We have to specify an ordering function
  - the algorithm needs to know if a < b
  - we re-use operator < on strings

```
bool operator <(const Entry &a, const Entry &b)
{
    return a.name < b.name;
}
```

Now we can use the sort algorithm:

```
template<class Iter> void sort(Iter first, Iter last);
```

```
sort(phone_book.begin(), phone_book.end());
```

The complete program

```
bool operator < (const Entry &a, const Entry &b) { return a.name < b.name;}

void add_entry(const string &n, int num) {
  Entry tmp;
  tmp.name = n; tmp.number = num;
  phone_book.push_back(tmp);
}

int main() {
  add_entry("Lipari Giuseppe", 1234);
  add_entry("Ancilotti Paolo", 2345);
  add_entry("Cecchetti Gabriele", 3456);
  add_entry("Domenici Andrea", 4567);
  add_entry("Di Natale Marco", 5678);
  sort(phone_book.begin(), phone_book.end());

}
```

Generic algorithms

- sort is an example of generic algorithm
  - to order objects, you don't really need to know what kind of objects they are, nor where they are contained
  - all you need is how they can be compared
  - (the < operator)
- So, to customize the sort algorithm, you have to specify what does it mean A < B
- You will learn later how to write a generic algorithm, that does not rely on the type of objects

Generic algorithms

- Another example: for_each()

```
void print_entry(const Entry &e)
{
    cout << e.name << " \t " << e.number << "\n";
}

int main(){
   …
   for_each(phone_book.begin(),phone_book.end(),print_entry);
}
```

Try to change the container from vector<> to map<>.

The for_each does not need to be changed!

for_each() works as long as it has a couple of iterators

Another example

- Suppose we want to print only the first 5 elements of the sequence:

```
for_each(phone_book.begin(),
         phone_book.begin()+min(3,phone_book.size()),
         print_entry);
```

It is all that simple!

We will show in the next lessons how it is possible to combine these objects to do almost everything.

Exercises

- Write a program that reads a file line by line, add a line number at the beginning of each line, and outputs the results on a new file.
- Write a program that reads a file line by line, reverts each line and output the results on a new file
- Write a simple phone book program using map<> and string: it should allow to
  - add a new entry,
  - look for a number, given a name,
  - look for a name given a number.

Exercises

- Let us begin to build the first brick of our project: a simple parser
- The program has data structures (you decide which type) to hold
  - a set of verbs with their past tense: take/taken, drop/dropped, move/moved, use/used, open, opened, etc.
  - a set of objects
- The program reads from the std input a sentence verb+object and responds with object+past-tense
  - If the verb is not found, say “what should I do with the <object>?”
  - If the object is not found, say “I don’t see any <object>”?
  - If nothing is found say a random phase like “say it again” or “what?”

Makefiles

- When building a large program with several files, we can use the *make utility*
  - see “Thinking in C++”, page 202