Introduction to C++

Memory management, Exceptions, Templates, standard containers and algorithms

Marco Frailis

INAF – Osservatorio Astronomico di Trieste

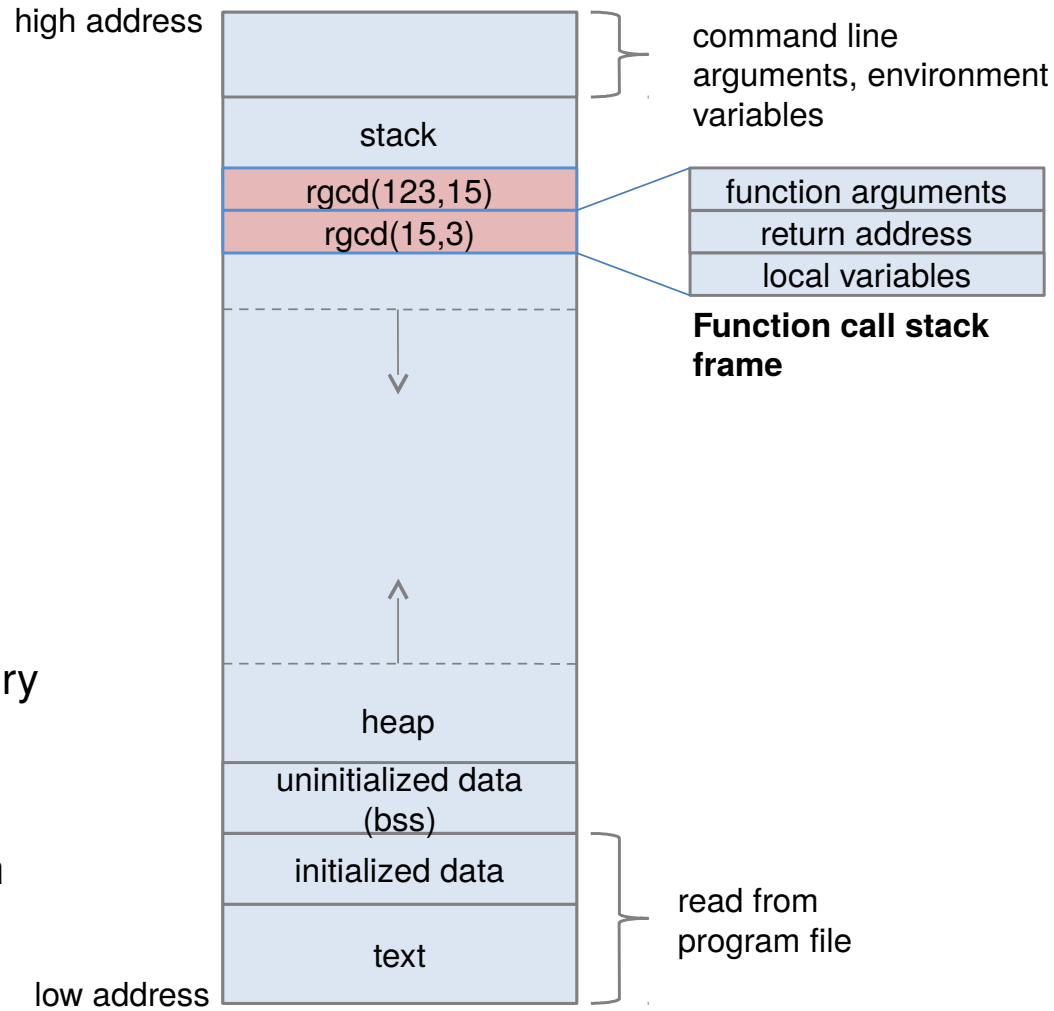
Contributions from:

Stefano Sartor (INAF)

Memory layout of a C/C++ program

```
// recursive greatest common divisor
int rgcd(int v1, int v2)
{
  if (v2 != 0)
    return rgcd(v2, v1 % v2);
  return v1;
}
```

- Text segment: machine instructions
- Initialized data segment: statically allocated global variables explicitly initialized
- bss area: statically allocated or global variables initialized by default to zero
- Heap memory: where dynamic memory allocation takes place
- Stack memory: non-static local variables and the information for each function call



Function overloading

ex09

```
void pretty_print(ostream& os, const vector<double>& v)
{
  os << "[ ";
  for (decltype(v.size()) i = 0; i < v.size(); ++i) {
    if (i != 0)
      os << ", ";
    os << v[i];
  }
  os << " ]" << endl;
}

void pretty_print(ostream& os, const vector<double>& v, int prec)
{
  ...
}

void pretty_print(ostream& os, vector<double>& v)
{
  // same body as first function
}

void pretty_print(ostream& os, vector<int>& v)
{
  ...
}
```

decltype tells you the name's or the expression's type of the argument

- Two functions that appear in the same scope are overloaded if they have the same name but have different parameter lists (different number of parameters or different parameter types)

Calling overloaded functions

- The compiler matches a call to a function automatically with the functions available in the overloaded set

```
const vector<double> a{4.5, 7.7};
vector<double> b{5.6, 7.6, 1.23};
vector<int> c{1,2,3,4,5};

pretty_print(cout, a);
pretty_print(cout, a, 10);
pretty_print(cout, b);
pretty_print(cout, c);
```

- The overloading can also distinguish between an reference parameter and a const reference parameter
  - But in the previous example, the function
    `void pretty_print(ostream& os, vector<double>& v)`
    is redundant
- In case a call is ambiguous (more than one match available), the programmer should cast one or more arguments to resolve the ambiguity

Function declaration

- A function must be declared before it is used
  - the compiler verifies the correspondence of arguments, so that it can give back error or warning messages if needed
- We can declare a function separately from its definition. Functions should be declared in header files and defined in source files
- A function declaration consists of a return type, the function name and the parameter list. The function body is replaced by a semicolon. These three elements form the function prototype
- In a function declaration, we can eliminate the names of the parameters and just keep the types
- Let's suppose that we have defined a function calculating the median of a vector in a source file named stats.cpp. The corresponding header file, that we call stats.h, can be the following

Header file

ex10

```
#ifndef GUARD_STATS_H
#define GUARD_STATS_H

#include <vector>

// Function returning the median of the input vector
double median(std::vector<double>);

// Function returning the mean of the input vector
double mean(std::vector<double>&);

#endif // end of GUARD_STATS_H
```

- User defined header files generally can have the extension ".h" or ".hpp" or ".hxx"
- In a header file, **`using`** directives should be avoided
- Header files can include other header files. To avoid multiple inclusion of the same header we should use the so called "header guards"
  - After the first inclusion, the name GUARD_STATS_H is defined and hence a second inclusion will not pass the #ifndef preprocessor condition

Header file inclusion

ex10

- The source file that defines a function should include the header file where the function is declared
- Any other source file that uses that function should include such header
- User defined headers are searched by the compiler starting from the directory in which the source file including it is located

```
#include "stats.h"
#include <iostream>
using namespace std;


int main()
{
  vector<double> v{1,2,3,4,5,6,7,8,9,10};

  cout << "v median: " << median(v) << endl;

  return 0;
}
```

Functions and default parameters

ex11

- For one or more function parameters, we can also specify a default value
- Default values should be provided in the function declaration and only to consecutive parameters starting from the last one

```
// Reads lines from a Comma-separated Value file
std::vector<std::vector<std::string>>
readCSVLines(std::istream&, char fieldSep = ',',
             char lineSep = '\n');
```

- A function that provides a default parameter can be invoked with or without an argument for that parameter

```
#include "utils.h"
#include <fstream>
using namespace std;

int main()
{
  ifstream infile("data.csv");
  auto lines = readCSVLines(infile); // default field and line separator
  infile.clear();                    // clear fail and eof bits
  infile.seekg(0, ios::beg);         // back to the start
  lines = readCSVLines(infile, ' '); // use space as field sep
}
```

Inline functions

- Calling a function is slower than evaluating the equivalent expression
- We can optionally qualify a (small) function definition as inline, asking the compiler to expand calls to the function “inline” when appropriate
- The compiler needs to be able to see the function definition: inline functions are usually defined in header files

```
inline void swap(int& v1, int& v2)
{
  int tmp = v2;
  v2 = v1;
  v1 = tmp:
}
```

- In general, inline should be used to optimize small (with few lines) functions that are called frequently

Compiling and linking multiple source files

- C++ supports the notion of separate compilation, which allows us to store our program into separate source files and compile each of these files independently
- The source files are compiled separately and then linked together (a two stage process):
  1. A source file is compiled without creating an executable. The result is an object file, with '.o' suffix
  2. The object files are merged together by a separate program called linker. The result is an executable

```
$ g++ -Wall -std=c++11 -c stats.cpp          # generates stats.o
$ g++ -Wall -std=c++11 -c test_stats.cpp     # generates test_stats.o
$ g++ test_stats.o stats.o -o test_stats
```

- The first two commands create the object files. The last command calls the linker (an external program, such as ld) and generates the executable

Exceptions

ex12

- In the first lecture, slide 43, we have used a C++ language feature, called exceptions, to signal an error during program execution:

```
double mean(const vector<double>& samples)
{
  ...
  if (size == 0)
    throw domain_error("mean of an empty vector");
  ...
}
```

- When a program throws an exception, the execution passes to another part of the program, along with an exception object
- An exception object contains information that the caller can use to act on the exception

<stdexcept>

logic_error	runtime_error
domain_error	range_error
invalid_argument	overflow_error
length_error	underflow_error
out_of_range	

Exception handling

ex12

- When a group of statements can throw exceptions, we can include that group of statements in a try statement

```
double mean = 0, median = 0;
try {
  mean = mean(samples);
  median = median(samples);
  cout << "sample statistics: mean=" << mean
       << " median=" << median << endl;
} catch (domain_error& e) {
  cerr << e.what() << endl;
}
```

- The try statement tries to execute the block following the try keyword. If a domain_error occurs, it stops the execution of the block and executes the block in the catch clause
- In this example, the `what` member function provided by the standard exceptions returns the string used in the initialization of the exception object
- Multiple catch clauses can follow a try statement, each one checking a different type of exception

Defining a template function 1/2

ex13

- To show some template function concepts, let's define a function that calculates the maximum value of a vector, for any element type T that provides the < operator

```
template <class T>
T max(const std::vector<T>& v)
{
  auto size = v.size();
  if (size == 0)
    throw std::domain_error("Error: max of an empty vector");

  T max_value = v[0];
  for (decltype(size) i = 1; i != size; ++i)
    if (max_value < v[i])
      max_value = v[i];

  return max_value;
}
```

- the template header, **`template <class T>`**, tells the compiler that we are defining a template function that will take a single type parameter T

Defining a template function 2/2

ex13

- Type parameters define names that can be used within the scope of the function. But they refer to type names, not to variables
- In the example, the parameter type T is used to specify the type of the vector elements, and the return type of the function. We also declare the local variable max_value of type T
- When we call max, passing as argument a vector<double>, the compiler will effectively create and compile an instance of the function that replaces every occurrence of T with double
  - Calling the template function:

```
vector<double>
v{3,5,2,9,4,1}:

double max_value = max(v);
```

No need to specify the template parameter for function templates

```
double max_value = max<double>(v);
```

Template compilation

ex13

- Template functions are different from ordinary functions. The compiler must have access to the source code that defines the template function when it finds a call to that function
- All compilers support the so called “inclusion model”, according to which the template function is declared in a header file, but the header file then includes the source file where the function is defined
- For instance, suppose that the max template function is declared in the tstats.h header and defined in the tstats.cpp source file, then:

tstats.h

```
#ifndef GUARD_TSTATS_H
#define GUARD_TSTATS_H

#include <vector>

template <class T> T max(const std::vector<T>&);

#include "tstats.cpp"

#endif // end of GUARD_TSTATS_H
```

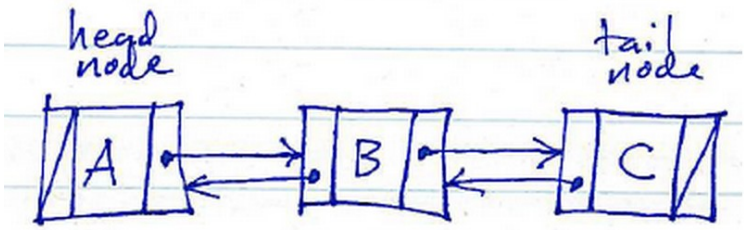
tstats.cpp

```
#ifndef GUARD_TSTATS_CPP
#define GUARD_TSTATS_CPP

#include "tstats.h"
#include <stdexcept>

template <class T>
T max(const std::vector<T>& v)
{
  // definition as before
}

#endif // end of GUARD_TSTATS_CPP
```

Additional containers: std::list 1/2

ex14

- std::list is a template data structure which allows to efficiently add or delete elements from the middle of the container.
- An std::list is implemented as a doubly linked list. The elements are not contiguous in memory




```
#include <iostream>
#include <algorithm>    // std::lower_bound
#include <list>

inline bool is_even(int i){
  return !(i%2);
}

using namespace std;
int main(){
  list<int> l1{1,2,3,4,5,14,15};
  list<int> l2{10,12,13};

  // find the first element in l1 which is not less than 10
  auto low = lower_bound(l1.begin(),l1.end(),10);

  // insert the list l2 in l1 starting from low index
  l1.splice(low,l2);
```

Additional containers: std::list 2/2

ex14

```
  // remove even elements using the function previously defined
  l1.remove_if(is_even);

  for(auto e : l1)
    cout << e << " ";
  cout << std::endl;

  return 0;
}
```

- std::list does not provide random access to its elements. Missing member functions: subscripting [], capacity() and reserve().

Associative containers: std::map 1/2

ex15

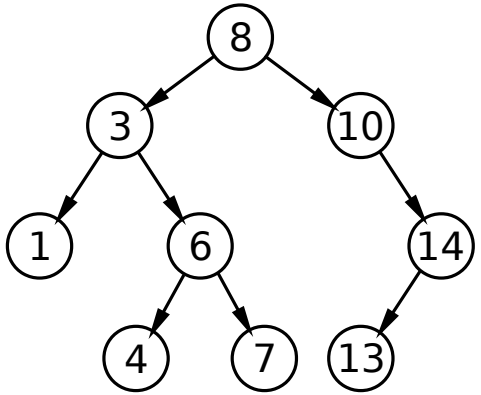
- `map` is an associative data structure that stores (key, value) pairs and lets us insert and retrieve elements quickly based on their keys
- Each element in a map is really a `std::pair`. A pair is a simple data structure that holds two elements, which are named first and second

```
#include <map>
#include <iostream>

using namespace std;

int main()
{
  // Constructing a map from an initializer list
  map<string, string> m1 = {{"TELESCOP", "PLANCK"},
                            {"INSTRUME", "LFI"},
                            {"APID", "1538"}};

  m1["TYPE"] = "3";   // adding a new key-value pair {"TYPE","3}

  // A pair is the really the type of each element in the map
  pair<string,string> p{"SUBTYPE","25"};
  // Attributes of a pair type
  cout << "pair elements: " << p.first << " " << p.second << endl;
```

Associative containers: std::map 2/2

ex15

```
  // Alternative isertion methods
  m1.insert(p);
  m1.insert({"PI1_VAL","102"});
  m1.emplace("PI2_VAL", "0");

  cout << "m1 elements: [ ";
  for (const auto& x : m1)
    cout << "(" << x.first << "," << x.second << ") ";
  cout << " ]" << endl;

  // Searching a key within the map m1
  auto it = m1.find("APID");
  if (it != m1.end())
    cout << "key \"APID\" found: "
         << "(" << it->first << "," << it->second << ")"
         << endl;
}
```



- map is usually implemented as a balanced binary search tree
- So, elements are automatically kept sorted based on the key values
  - Time to access an element: logarithmic in the total number of elements of the container
- Alternative: unordered_map, implemented with hash tables

Built-in array

ex16

- An array is a kind of container that is part of the core language
- An array contains a sequence of one or more objects of the same type
- It has no function members.
- According to the C++ standard, the number of elements in the array must be known at compile time and cannot change dynamically

```
// defining an array of doubles: 1024 elements
double buffer[1024];

// the size of the array can be specified using const variables
const auto BSIZE = 1024;
double buffer2[BSIZE];

// size deduced from the initialization list length
double filter[] = {0.020, 0.230, 0.498, 0.230, 0.020};

// set all elements to 0.0
for(auto i=0; i<BSIZE; i++)
  buffer2[i] = 0;
```

Array of char

- A string literal is an array of const char with an extra character (the null character, '\0'):

```
const char hello[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

has the same meaning as the string literal "Hello". In general, C style strings are defined as:

```
// two examples of C-style strings
char* msg1 = "Please, insert your name:";
char msg2[] = "Parameter 1:";
```

- In the above example, the null character is added automatically by the compiler. This character is used by the function strlen, defined in the `cstring` header, to calculate the length of the string

```
size_t len = strlen(msg1);
```

std::array

ex16

- Provided by the C++11 standard as an alternative to built-in arrays
  - It is a fixed-size sequence of elements where the number of elements is specified at compile time
  - But it provides a subset of methods used by other std containers:

    array::begin(), array::end(), array::size(), array::data(), etc.

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
  array<int, 5> a = {0,5,2,6,9};

  // key is a value to search in the unsorted a
  int key = 6;
  bool found = false;

  size_t i = 0;
  while (i < a.size() and !found) {
    if (a[i] == key)
      found = true;
    ++i;
  }

  // ... print result
}
```

Non-type template parameter

Only constant integral values

Iterators and standard algorithms

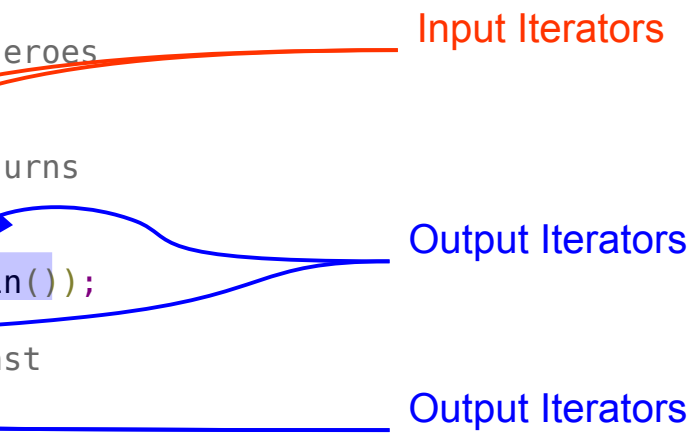
- An Iterator is an object which allows to iterate (hence its name) through the elements of a container, providing at least two operations:
  - (++) **increment operator**. Move the iterator to the next element.
  - (*) **dereference operator**. Access the current element.
- According to the peculiarities of the container, different Iterator categories, which provide different operations, exist:
  - **Input iterator**: Sequential access in one direction, read only
  - **Output iterator**: Sequential access in one direction, write only
  - **Forward iterator**: Sequential access in one direction, read and write
  - **Bidirectional iterator**: Sequential access in both directions, read and write
  - **Random-access iterator**: Efficient access to any element, read and write

Iterators and standard algorithms

ex17

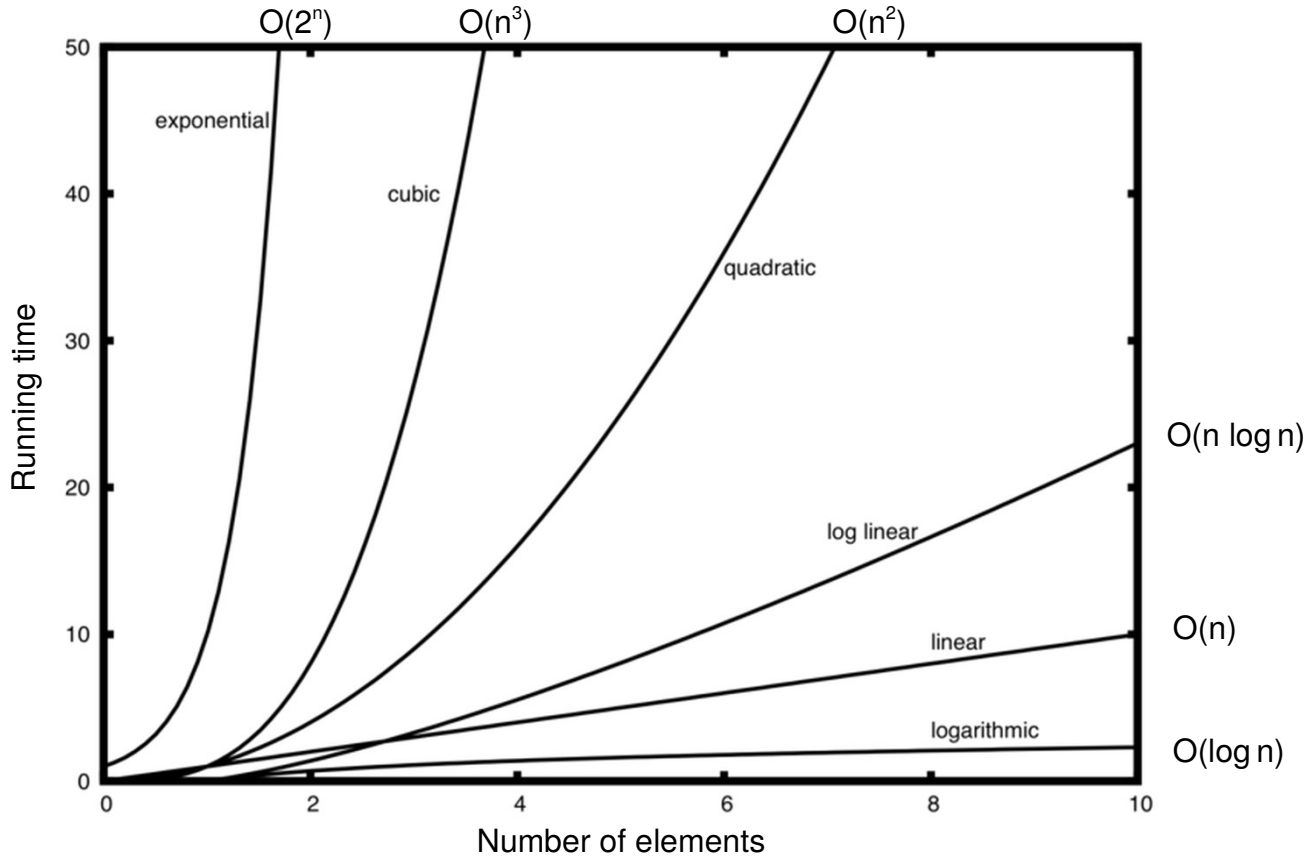
- std containers provide member functions begin(), returning an iterator to the first element, and end(), returning an iterator to the last plus one element.
- The standard library provides a few algorithms and utils which accept iterators as arguments (regardless the container type).

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
int main(){
  vector<int> v(8); // v is initialized with 8 zeroes
  int a[] = {2,5,7,1,0,45,30,3};

  /* copies first 4 elements from a to v and returns
   * the iterator to the next element in v
   */
  vector<int>::iterator it = copy(a, a+4, v.begin());

  /* fills with -5 starting from it until the last
   * element of v
   */
  fill(it, v.end(),-5);
}
```



Computational complexity in the standard library

- Big-O notation expresses the running time of an algorithm as a function of a given input of size n



Sorting

ex18

- **sort**: sorts the elements using operator < by default (relative order of equal elements may not be preserved). *O(N·log(N))*
- **nth_element**: rearranges the elements in such a way that the element at the nth position is the element that would be in that position in a sorted sequence. None of the elements preceding nth are greater than it, and none of the elements following it are less. *O(N)*

```
#include <vector>
#include <algorithm>
using namespace std;
int main(){
  int a[] =       {11, 45, 6, 7, 8, 10, 9, 2};
  vector<int> v = {11, 45, 6, 7, 8, 10, 9, 2};

  // sort the elements of v
  sort(v.begin(),v.end());

  // set the 4th element in the position as array a was sorted
  nth_element(a,a+3,a+8);
  /*
    v == 2 6 7 8  9 10 11 45
    a == 6 2 7 8 11 10  9 45
   */
}
```

Lower bound and find

ex19

- **lower_bound**: returns, in an ordered container, an iterator to the first element equal or grater then the compared one. *O(log(N))*
- **find**: returns an iterator to the first element equal to the compared one if present; returns the end iterator otherwise. *O(N)*

```
#include <algorithm>
#include <vector>

using namespace std;
int main(){
  vector<int> v = {1,5,6,8,11,20};
  vector<int>::iterator lb, fi;

  // find the first element equal to or grater then 7
  lb = lower_bound(v.begin(),v.end(),7);

  // find the first element equal to 7
  fi = find(v.begin(),v.end(),7);

  /*
   *lb == 8
   fi == v.end()
   */
}
```

Sort: comparison operation

ex20

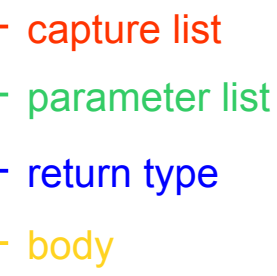
- sort, nth_element and lower_bound, accept a callable object as final optional argument to specify a different order relation that guides the behavior of the comparison operation (by default the usual <)

```
#include <vector>
#include <algorithm>
#include <iostream>

// function to sort elements in descending order
bool gt (int x, int y) { return x > y;}

using namespace std;
int main(){
  vector<int> v = {11, 45, 6, 7, 8, 12, 9, 2};
  int a[] =       {11, 45, 6, 7, 8, 12, 9, 2};

  sort(v.begin(),v.end(),gt);

  auto it = lower_bound(v.begin(),v.end(),10,gt);

  nth_element(a,a+3,a+8,gt);

  /*
   v == 45 12 11 9 8 7 6 2
   a == 11 45 12 9 8 7 6 2
   *it == 9
  */
}
```

Lambda expressions 1/2

ex21

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
int main(){
  vector<int> v = {0,1,7,5,6,3,5,2,3,4};

  int val = 5;

  // lambda expressions can be stored in variables
  auto lambda  = [val] (int x) -> bool { return x == val;};

  // count how many elements are equal to val
  int found = count_if(v.begin(),v.end(),lambda);

  cout << "found " << found << " elements equal to " << val << endl;

  // print the elements using a lambda expression
  for_each(v.begin(),v.end(),[](int x){cout << x << " ";});

  cout << endl;
  return 0;
}
```



Lambda expressions 2/2

- Lambda expressions are anonymous functions, mostly used to improve code readability when passing functions as arguments
  - **[]** *capture list*: possibly empty list of names defined in outer scope used by the expression
  - **()** *optional parameter list*: specify the arguments the lambda expression requires
  - **-> *type*** : (optional) specify the return type
  - **{}** *body* : body of the lambda expression, specifying the code to be executed

Classes: struct definition

ex22

- The struct keyword lets us define a new data type as an aggregate of different attributes (a record)
- In the following example, we define two types:
  - A Point type, to store spherical coordinates (RA, DEC)
  - A Galaxy type, to store galaxy properties

```
struct Point {
  double ra;
  double dec;
};

struct Galaxy {

  // ID of galaxy instance
  unsigned long id;
  // Galaxy coordinates (RA, DEC)
  Point coord;
  // Redshif of Galaxy instance
  double z;
  // Velocity of Galaxy instance
  double v;
};
```

Struct members

ex22

- Initializing and accessing a struct attributes

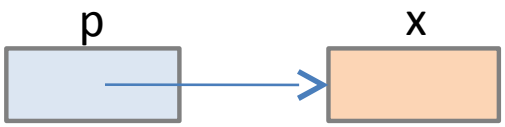
```
int main()
{
  Galaxy g1{1, {349.18372, -0.070794291}, 0.527313,
            1/(1 + 0.527313)};

  Galaxy g2; // default initialized galaxy instance
             // in this case all elements initialized to zero

  g2.id = 2;
  g2.coord.ra = 348.3452;
  g2.coord.dec = 0.0653423;
  g2.z = 0.5135289;
  g2.v = 1/(1+g2.z);

  double ra = g1.coord.ra;
  double dec = g1.coord.dec;

  cout << "galaxy g1 RA-DEC: (" << ra << ", " << dec << ")" << endl;

  std::vector<Galaxy> galaxy_cluster;
  galaxy_cluster.push_back(g1);
  galaxy_cluster.push_back(g2);
}
```

- In C++, a struct is in fact a class where by default all members are public, i.e. directly accessible with the dot operator

Pointers 1/2

- In the previous example, a galaxy instance requires **40 bytes** of memory
- If we need to read in memory millions of galaxy instances and sort them by their coordinates, this would require millions of expensive copy operations when swapping elements in a `std::vector<Galaxy>`
- We need a lighter type to support such operations, since we cannot create a vector of references.
- A pointer is a value that represents the memory address of an object

Pointer 2/2



- If x is an object , than &x is the address of that object
  - The & in &x is the address operator, distinct from the & used to define reference types
- If p is the address of an object, then *p is the object itself
  - The * is the dereference operator, analogous to the one applied to iterators
  - We say that p is a pointer that points to x
  - If v is a member of the object pointed by p, then we access v from p with two alternative notations:

    p->v or (*p).v
- The address of an object of type T has type “pointer to T”, written as T*

Using pointers

ex23

- Pointer definition

```
int *p, q;  // p is a pointer to int, q is an int variable
```

or

```
int* p, q;  // equivalent but misleading (q is not a pointer)
```

or

```
int* p;
int q;
```

- Pointers should be initialized to `nullptr`

```
double* p = nullptr;
```

- Pointer usage:

```
double x = 0.5;
cout << "x = " << x << endl;

// dp points to x
double* dp = &x ;
// change the value of x through dp
*dp += 0.5;
cout << "x = " << x << endl ;

// xref is a reference to x
double& xref = *dp;
xref += 0.5;
cout << "x = " << x << endl;
```

Pointer arithmetic

- Built-in arrays and pointers are strictly related: when we use the name of an array as a value, that name represents a pointer to the initial element of the array

```
double buffer[1024];
*buffer = 1;  // the first element of buffer is now set to 1.0
```

- A pointer is a kind of random-access iterator: if p points to the mth element of an array, then (p + n) points to the (m + n)th element of the array and (p - n) points to the (m – n)th element (if they exists)
- If p and q are pointer to elements of the same array, the integer number of elements between them is given by (p – q) (it might be negative)

Dynamic memory allocation and deallocation

- If T is a type, it is possible to allocate at run-time an object of type T using the **new** operator. The result is a pointer to the allocated object
- To allocate a single object, the new operator can be used in three forms:

```
new T          new T(args)          new T{args}
```

- With the first form, the object is default-initialized; the second and third forms let us provide the arguments to initialize the object

```
double *pval1 = new double;  // uninitialized value
double *pval2 = new double(1.0);
```

- When we have finished using a dynamically allocated object, we must delete it, using the delete operator

```
delete pval1;        pval1 = nullptr;
delete pval2;        pval2 = nullptr;
```

- After applying delete, the memory occupied by *pval1 and *pval2 is freed and the pointers become invalid

Array allocation and deallocation

- There is another form of the new operator that can be used to allocate, at run-time, an entire array of objects:

```
new T[n]
```

- It creates an array of n objects of type T and returns a pointer to the first element of the array. Each element is default-initialized
- As an example, suppose that a file contains a sequence of floating-point values and that this sequence is preceded by an integer specifying the number of values:

```
std::size_t n;
ifile >> n;  // first reading the total number of values
// Allocating the necessary memory buffer
double* samples = new double[n];
```

- To later free the memory allocated, we must use the **delete[]** operator:

```
delete[] samples;
```

Sorting with a vector of pointers

ex24

```
int main()
{
  // filling a vector of galaxy pointers with dynamically allocated instances
  vector<Galaxy*> vg;
  vg.push_back(new Galaxy{1, {349.18372, -0.070794291}, 0.527313});
  vg.push_back(new Galaxy{2, {348.3452, 0.0653423}, 0.5135289});
  vg.push_back(new Galaxy{3, {346.29340, 0.034823}, 0.5126848});

  // lambda expression to sort galaxy pointers by RA
  auto sort_by_ra = [] (const Galaxy* x, const Galaxy* y)
                      {return x->coord.ra < y->coord.ra;};

  sort(vg.begin(), vg.end(), sort_by_ra);

  cout << "galaxies ordering by RA: ";
  for (const auto x : vg)
    cout << x->id << " ";
  cout << endl;

  // deleting all the dynamically allocated galaxies within the vector
  for (auto x : vg) {
    delete x;
    x = nullptr;
  }

  return 0;
}
```