# Introduction to C++

Basic program building blocks, control structures, functions

Marco Frailis

INAF – Osservatorio Astronomico di Trieste

## The C++ language 1/2

- C++ is designed to be a statically typed, general purpose language, to be as compatible with the C language as possible
- It is designed to support multiple programming styles:
  - Procedural programming
  - Data abstraction
  - Object-oriented programming (OOP)
  - Generic programming (with templates)
  - Functional programming (to some extent, e.g. lambda functions)
- C++ is a compiled language
  - Source text files processed by a compiler producing object files
  - Object files combined by a linker yielding an executable program
  - The executable program is created for a specific hardware/system combination

## The C++ language 2/2

- It provides a standard library, extending the core language, including:
  - Input/output classes
  - Data structures: dynamic arrays, linked lists, binary trees, hash tables
  - Several algorithms: searching, sorting, counting, manipulating
- Memory management
  - Low level: pointers and raw arrays, allocation, deallocation
  - Higher level: allocation and smart pointers

# Object-oriented programming

- An **object** packages both data and procedures that operate on the data
- Such procedures are called **member functions** or methods or operations
- The implementation of an object is specified in its **class**, which defines the internal data (or **attributes**) of the object and the **operations** that the object can perform
- New classes can be defined as **subclasses** of a **parent class**, **inheriting** its attributes and operations
- An **abstract class** is one whose main purpose is to define a **common interface** for its subclasses
- When inheriting from an abstract class, we speak about **polymorphism**, since the same interface is associated to different implementations through its subclasses.

## Generic programming

- Generic programming parameterizes algorithms so that they work for a **variety** of **types** and **data structures**
- For instance, data structures holding a collection of elements of some kind, such as vectors, lists, queues, associative containers, are general concepts and should be independent of the type of elements
- Also, algorithms to sort, copy or search a sequence of elements should be independent of the particular type of the elements or type of the container
- They can be defined as **templates**, parameterized by the types to which type are applied
- In C++, templates are a **compile time** mechanism, avoiding any run-time overhead

## C++ standard evolution 1/2

- First ISO standard in 1998 (C++98)
- New revision of the standard in 2003 (C++03)
  - No new language features, just a *bug fix* release for compiler writers
- C++ Technical Report 1 (TR1) in 2005
  - A document proposing additions to the standard library

## C++ standard evolution 2/2

- New C++ ISO standard in 2011 (C++11)
  - Additions to the core language, including:
    - Deducing the type of an object from its initializer (**auto**)
    - Lambda expressions
    - Move semantics
    - The range-for statement
    - Type aliases
  - Additions to the standard library, including
    - Hashed containers
    - Basic concurrency library (threads)
    - Regular expression library
    - tuple library
    - Unique and shared pointers
- Last C++ standard revision in 2014 (C++14)
  - Minor revision with small improvements, e.g. function return type deduction

## C++ compilers (with C++11 conformance)

- GCC (GNU Compiler Collection), version 4.8 or greater
  - Linux, Mac OS X, Windows (with MinGW or Cygwin), iOS, Android
  - ISO support status: https://gcc.gnu.org/onlinedocs/libstdc+/manual/status.html#status.iso.2014
- Clang, version 3.3 or greater
  - Linux, Mac OS X, Windows (with MinGW or Cygwin), iOS, Android
- Intel(R) C++ Compiler (commercial), version 13.0 or greater
  - Linux, Mac OS X, Windows, Android
- Microsoft Visual C++, version in Visual Studio 2013 or greater
  - Free version (with license limitations): Visual Studio Community 2013
  - Windows, Android, iOS

# C++ in Astronomy

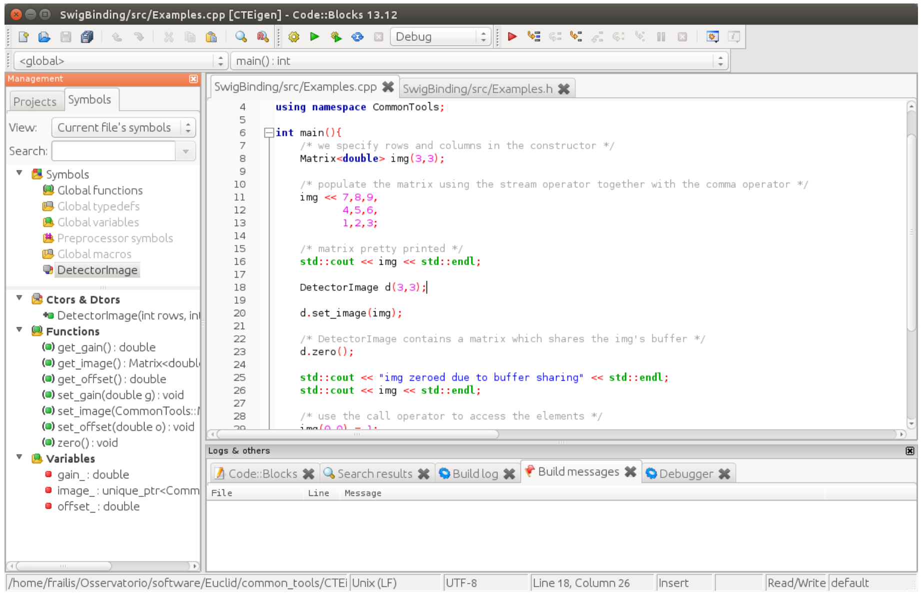
- Planck
  - Most of the processing pipelines developed in C++ (e.g. telemetry processing, calibration)
- LSST
  - Core processing developed in C++
- ALMA
  - ACS software: C++ implementation for the control system
- MAGIC telescopes
  - MARS, the analysis and reconstruction software is based on ROOT and written in C++
- Euclid
  - C++ and Python selected for developing all the processing levels
- FITS standard
  - cfitsio (C)
  - CCfits (C++)

## References

- On-line references and FAQ:
  - http://www.cplusplus.com
  - http://en.cppreference.com/w/
  - https://isocpp.org/faq
  - http://www.stroustrup.com/C++11FAQ.html
- On-line tutorials
  - http://www.tutorialspoint.com/cplusplus/index.htm
  - http://www.cplusplus.com/doc/tutorial/
  - http://www.learncpp.com/
  - http://www.cprogramming.com/tutorial/c++-tutorial.html
- Books (in suggested order)
  - Accelerated C++: Practical Programming by Example, by A. Koenig and B. E. Moo
  - The C++ Standard Library: A Tutorial and Reference, 2nd edition, by N. M. Josuttis
  - C++ Primer (5th edition), by S. B. Lippman, J. Lajoie and B. E. Moo
  - The C++ Programming Language, 4th edition, B. Stroustrup
- More advanced books
  - Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, S. Meyers
  - C++ Templates: The Complete Guide, by D. Vandevoorde and N. M. Josuttis
  - Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, by E. Sutter
  - More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions, by E. Sutter

## Integrated Development Environments (IDEs)

- Code::Blocks (http://www.codeblocks.org/)
  - A lightweight IDE (it is provided in the virtual machine created for this course)



```
using namespace CommonTools;

int main(){
    /* we specify rows and columns in the constructor */
    Matrix<double> img(3,3);

    /* populate the matrix using the stream operator together with the comma operator */
    img << 7,8,9,
           4,5,6,
           1,2,3;

    /* matrix pretty printed */
    std::cout << img << std::endl;

    DetectorImage d(3,3);

    d.set_image(img);

    /* DetectorImage contains a matrix which shares the img's buffer */
    d.zero();

    std::cout << "img zeroed due to buffer sharing" << std::endl;
    std::cout << img << std::endl;

    /* use the call operator to access the elements */
```

- Eclipse CDT (https://eclipse.org/cdt/)
- NetBeans IDE (https://netbeans.org/features/cpp/)
- Emacs (http://www.gnu.org/software/emacs/)
- Coding Ground (http://www.tutorialspoint.com/compile_cpp11_online.php)
  - An on-line web IDE, for short tests and examples
- Others:
  - QT Creator, Xcode (Mac OS X), Visual Studio (Windows)

## C++ program: `main` function example

ex01

```cpp
#include <cmath>
#include <iostream>

using namespace std;

const double PI = 3.1415926535897931;

const double RAD2DEG = 180.0/PI;

// Convert Cartesian coordinates to spherical coordinates
int main()
{
  // Coordinates to be converted
  double x=10.3, y=-5.2, z=36;

  // Norm
  double r = sqrt(x*x + y*y + z*z);

  // Longitude
  double phi = atan2(y, x);
  phi = phi < 0 ? phi + 2*PI : phi;

  // Colatitude
  double theta = acos(z/r);

  cout << "Longitude = " << phi*RAD2DEG << " deg, Colatitude = "
       << theta*RAD2DEG << " deg, Radius = " << r << endl;

  return 0;
}
```

Including standard library functions and classes

Constants definitions

Main function definition for the executable

statements for coordinates conversion

Printing output

## Compiling with GNU g++

- C++ source code should be given one of the valid C++ file extensions: .cpp, .cc, .cxx or .C, while the .c extension is reserved to C programs
- Saving the previous source code in file “coord.cpp”, in order to compile it and create an executable, we can use the GNU g++ compiler:

```
$ g++ -std=c++11 coord.cpp -o coord
```

- This command compiles the source code to machine code, saving it in the executable file “coord”
- The option “-std=c++11” enables compiler support for the 2011 ISO C++ standard

## The `#include` preprocessor[1] directive

- Many C++ fundamental facilities, such as I/O, are not part of the core language
- They are part of the standard library. These facilities are requested using the **`#include`** directive
- Example: requesting the I/O standard library facilities and standard strings

  ```
  #include <iostream>
  #include <string>
  ```

  where `iostream` and `string` are called a standard headers
- Including a header file produces the same results as copying the header file into each source file that needs it
- Normally the include directives are inserted at the beginning of a source file

[1]The preprocessor is a macro processor, called by the compiler to transform the program before compilation

# Variable definition and initialization

- A variable (also called object in C++) provides a named storage (a region of memory) that our program can manipulate
- Each variable definition starts with a type specifier, followed by a comma separated list of one or more names (identifiers)

```
int day, month, year;
double salary;
```

- A definition may also provide an initial value for the object (initialization)
- C++ supports four forms of variable initialization: copy-initialization, direct-initialization, list-copy-initialization, list-direct-initialization

```
int month = 9, year = 2012, day;          // copy-initialization
double salary(1250.23);                   // direct-initialization
```

OK for built-in types

C++11

```
std::complex<float> c = {-1.0, 0.0};      // list-copy-initialization
long int count{35600};                    // list-direct-initialization
```

Recommended for class types

- It is safer to initialize every object of built-in type

# C++ built-in types

ex02

```
// Boolean type
bool empty = true;

// Character types
char single_letter = 'D';

// Integer types
char tiny_value = 127;
short small_value = 32767;
int value = 2147483647;
long large_value = 2147483647;
long long huge_value = -9.22E18LL;

// Unsigned integer types
unsigned char r = 255;
unsigned short small_pvalue = 65535;
// ...etc.

//Floating point types
float temperature = 2.323787;
double theta = 2.663832728147556;
long double ltheta = 2.6638327281475567373L;
```

- Additional character types: `wchar_t` and `char16_t` and `char32_t`, for Unicode characters (UTF-16 and UTF-32)

## Literals

- Literal integer constants
  - Notations:

```
242   /* decimal */    0362  /* octal */    0xF2   /* hexadecimal */
```

  - Integer literal types: int or long by default, depending on value. Use of suffix to force type

```
128U  /* unsigned int */    12ULL /* unsigned long long */
```

- Floating point literals: double by default

```
2.323787F  /* float */   2.66383272814755673 73L  /* long double */
```

- Character literals (char type, ASCII): `'a'  '3'`
  - Nonprintable characters: use of escape sequence

```
'\n'  /* newline */   '\t'  /* horizontal tab */  '\?'  /* question mark */
'\''  /* single quote */  '\"'  /* double quote */  '\0'  /* null character */
```

- String literals are *arrays* of characters

```
"\tHello World\n"            ""  /* empty string */
```

# Expression: Arithmetic, relational and logical operators

- An expression is composed of one or more operands and (usually) an operator; it produces a result
- Arithmetic operators:

```
a + b                    a - b
a * b                    a / b
                         a % b
```

Reminder after division a / b

- Relational operators:

```
a < b                    a <= b
a > b                    a >= b
a == b                   a != b
```

- Logical operators

| Operator | Meaning | Example |
|---|---|---|
| && | and | `(a >= 0) && (a < 5)` |
| \|\| | or | `(a < 0) \|\| (a >= 5)` |
| ! | not | `!((a >= 0) && (a < 5))` |

# Bitwise operators

- Bitwise operators:

| Operator | Meaning | Example |
|---|---|---|
| << | left shift | `a << 3` |
| >> | right shift | `a >> 2` |
| & | bitwise and | `a & mask` |
| \| | bitwise or | `a \| mask` |
| ^ | bitwise xor | `a ^ mask` |
| ~ | bitwise not | `~mask` |

```
unsigned char a = 20, mask = 1; // i.e. a = 00010100, mask = 00000001


a >> 2          // result = 00000101
mask << 3       // result = 00001000
~mask           // result = 11111110

a & (mask << 4) // result = 00010000 (check if bit 4 is set)
a | (mask << 1) // result = 00010110 (set bit 1)
a ^ (mask << 3) // result = 00011100 (flip bit 3)
```

# Assignment and compound assignment operators

```
// definition and initialization
int index = 0 , step = 5;
double deltaT = 0;

// assignment
deltaT = 0.3;
index = step * deltaT ;  // here an implicit truncation is performed
```

- The result of an assignment is the left-hand operand
- We often apply an operator to an object and reassign the result to the same object:

```
index = index + step * deltaT;
```

- C++ provides compound-assignment operators as a shorthand, for arithmetic and bitwise operators

```
index += step * deltaT;  // equivalent to the previous statement
```

## Increment and decrement operators

- Increment (++) and decrement (- -) operators can be used as a shorthand for adding or subtracting 1 from an object
- There are two forms of these operators: prefix and postifix
- The prefix form increment (decrement) its operand and yelds the changed value as its result
- The postfix form increment (decrement) its operand but yelds a copy of the original, unchanged value as its result

```
int i = 0, j;

j = ++i;    // j = 1 , i = 1;
j = i++;    // j = 1 , i = 2;
```

# Implicit type conversions

- In expressions with operands of mixed types, the types are converted to a common type by the compiler
- For arithmetic conversions, the rules define a hierarchy of conversions in which operands are converted to the widest type in the expression
- The conversion rules are defined so as to preserve the precision of the values involved
- Conversion to/from **`bool`**:
  - Integral and floating-point values can be converted to **`bool`**. If the value is 0, the resulting bool is **`false`**, otherwise it is **`true`**.
  - **`bool`** can be converted to other types: **`true`** is converted to 1, **`false`** to 0

# Explicit conversions

- An explicit conversion is spoken of as a **cast**
- C old-style cast:

```
int x = 37, y = 6;
float result = (float) x / y;   // or float(x) / y
```

- Casts should be discouraged: they turn off normal type-checking
- C++ tries to make casts more visible providing named cast operators:

```
result = static_cast<float>(x) / y;
```

- Additional C++ named cast operators:

```
dynamic_cast, const_cast, reinterpret_cast
```

## Strings in C++

- The Standard Library provides the `string` type to support variable length character strings
- The library takes care of managing the memory associated with storing the characters
- It provides several ways to initialize a `string` variable:

```
#include <string>

int main()
{
  std::string s1;                 // s1 is an empty string
  std::string s2(s1);             // s2 is initialized as a copy of s1
  std::string s3("swing")         // s3 is a copy of the string literal
  std::string s4(5,'-');          // s4 = "-----"
}
```

- `string` is part of the **`std`** namespace
- A namespace is a collection of related names: the standard library uses **`std`** to contain all the names that it defines
- `std::string` is a qualified name, using the scope operator (`::`)

# String operations

ex03

```
#include <string>

using namespace std;

int main()
{
  string s1;
  string s2 = "Hell0";
  string s3 = s2 + " World";              // string + literal concatenation

  bool isEmpty = s1.empty();              // check if s1 is an empty string
  char first = s3[0];                     // get the first character
  s3[4] = 'o';                            // modify the 5th character in s3
  bool isLess = s2 < s1;                  // comparison operator between strings
                                          // (lexicographic order)
  s1 = s2;                                // copy s2 in s1, replacing s1 characters
  string s4 = s3.substr(6, 5);            // get the substring starting at index 6
                                          // and taking 5 chars (s4 = "World")
  string::size_type size = s4.size();     // get string size in bytes
}
```

Type defined in the string class:
synonym of an unsigned int, but more portable

# C++ Input/Output

- The standard library defines a family of types that support I/O to and from devices such as files and console windows. Additional types allow strings to act like I/O streams
- The I/O types are defined in three separate **headers**:

  **`<iostream>`** defines the types used to read from and write to a console window

  **`<fstream>`** defines the types used to read from and write to named files

  **`<sstream>`** defines the types used to read from and write to in-memory strings

- The **`iostream`** header includes the definition of three I/O **objects**:
  - The `istream` object named `cin`, also referred to as *standard input*
  - The `ostream` object named `cout`, also referred to as *standard output*
  - The `ostream` object named `cerr`, also referred to as *standard error*

## Standard I/O example

ex04

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

int main()
{
  cout << "Please, insert two numbers:" << endl;
  float v1, v2;

  cin >> v1 >> v2;

  cout << "The sum of " << v1 << " and " << v2
       << " is " << v1 + v2 << endl;
  return 0;
}
```

- The stream operators (<< and >>) are left associative and return the left operand, i.e. the stream object)
- So we can chain the stream operations

## Formatting floating point values

ex05

I/O format state (parametric) manipulators

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

int main()
{
  const double PI = 3.1415926535897931;

  cout << PI << endl;
  cout << setprecision(15) << PI << endl;
  cout << scientific << PI << endl;
  cout << fixed << setw(20) << PI << endl;     // setw sets the field width
  cout << resetiosflags(ostream::floatfield);  // revert to default notation

  cout << PI << endl;
}
```

Output:

```
3.14159
3.14159265358979
3.141592653589793e+00
   3.141592653589793
3.14159265358979
```

## Statements

- Most statements in C++ end with a semicolon
- We have already seen some expression statements
  - i.e. an expression followed by a semicolon
  - Commonly expression statements affect the program's state: assignment, increment, input/output operators, declaration statements
- A compound statement, or block, is a (possibly empty) sequence of statements surrounded by a pair of curly braces
- Compound statements can be used where the rules of the language require a single statement

## Scope of a name

- Every name in a C++ program must refer to a unique entity (e.g. a variable, function, type, etc.)
- A name can be reused, also with different meanings, as long as it is used in different context
- Such context is the **scope**, i.e. a region of the program
- There are different kinds of scope:
  - The statements between a pair of matching braces form a scope. The body of the `main` or the body of every function and the scopes nested inside a function (such as a **block**) form **local scopes**
  - Names defined outside any function have **global scope**. They are accessible from anywhere in the program
  - **Namespaces** partition the global namespace. A namespace is a scope, as we have already seen with the standard library namespace `std`
  - Every **class** defines its own new scope. An examples is given by the type `size_type` of the `string` class

# Scope example

```cpp
#include <iostream>

using namespace std;

const string s = "---------";

int main()
{
  unsigned int s = 0;

  {
     unsigned int s = 1;
     cout << "Inner s: " << s << endl;
  }

  cout << "Outer s: " << s << endl;
  cout << "Global s: " << ::s << endl;

  return 0;
}
```

Global const variable

s is a variable in the main function local scope

In this block, s definition hides the outer one

- Output:

```
Inner s: 1
Outer s: 0
Global s: ---------
```

## if statement

Condition

```
if (rawVal > 90) {
  cout << "Warning: hard limit exceeded" << endl;
  ++hardLimit ;
}
else if (rawVal > 70)
  ++softLimit;
else {
  sum += rawVal;
  ++numSamples;
}
```

Statement 1

Statement 2

- Second **if** form: without the **else** branch

```
if (i < j)
  ++i;
```

# **while** and **do while** statements

while statement

Condition

```cpp
while (i > 0 && v[i] < key) {
   v[i+1] = v[i];
   --i;
}
```

Statement

do-while statement

```cpp
~~double threshold = -1;~~
~~do {~~
~~   cout << "Please enter a threshold level in the range [0, 1]"~~
~~        << endl;~~
~~   cin >> threshold;~~
~~} while (threshold < 0 || threshold > 1);~~
```

Statement executed at least once

```cpp
int i = 0;
while (i < size) {
   // operations that don't
   // change the value of i
   ++i;
}
```

Recurrent pattern: `for` loop

## for and range-based for loops

for statement

init-statement condition expression statement

```
for (int i = 1, j = 0; i < v.size(); i += 2, ++j) {
  s[j] = (v[i-1] + v[i])/2;
  d[j] = v[i-1] - s[j];
}
```

Executed after body of for loop

range-for statement

```
for (int x : v) {
  sum += x;
  sum2 += x*x;
}
```

for each value x in range v

C++11

Infinite loop until event

```
for (;;) {
  cout << "Processing next event" << endl:
  checkEventQueue();
  if (stopExecution)
    break;
}
```

ends the nearest enclosing loop

```
for (int i = 0; i < v.size(); ++i)
{
  if (v[i] < 0 || v[i] > 5)
    continue;
  sum += v[i];
}
```

skip rest of iteration

## `switch-case` statement

```
int apId = packet.apId();

switch (apId) {
  case 1536:
  case 1538:
    ++lfiHkCounter;  // LFI housekeeping telemetry found
    break;
  case 1540:
    ++lfiSciCounter; // LFI scientific telemetry found
    break;
  default:
    cout << "Unknown telemetry packet" << endl;
}
```

expression with integral result

- It provides a more convenient way to write deeply nested if/else logic
- The result of the expression is compared with the value associated with each **`case`**
- Execution starts with the first statement following the matching label, till a **`break`** is found

## Standard vector 1/3

ex06

- The standard library provides a type, named vector, that holds a sequence of values of a given type and grows as needed
- It is defined using a language feature called template classes

Template parameter

C++11

```
#include <iostream>
#include <vector>
#include <complex>

using namespace std;

int main()
{

  int n = 10;
  vector<float> a(n, 5.0);      // a has n elements that are copies of 5.0
  vector<int> b = {3,2,7,11,23}; // b contains the elements provided in the list

  cout << "First and last elements of b: "
       << b[0] << " " << b[b.size()-1] << endl;

  // vector from a list of numbers gathered from the standard input
  double value;
  vector<double> c;
  cout << "Please insert some values (use Ctrl-D to end): " << endl;

  // The user ends inputting numbers with the EOF character
  while (cin >> value)
    c.push_back(value);  // appends a new element at the end of vector c
```

Continue ...

# Standard vector 2/3

ex06

```
  // Print vector elements
  cout << "c elements: ";
  for (double x : c)
    cout << x << " ";
  cout << endl;

  vector<double> d(c);    // defines d as a copy of c

  // e holds a copy of the first half of d
  vector<double> e(d.begin(), d.begin() + d.size()/2);

  c.clear();  // removes all elements in c; c.size() == 0

  // erase elements of first half of d
  d.erase(d.begin(), d.begin() + d.size()/2);

  c = e;  // c is now a copy of e

  // prepend elements of a in e
  e.insert(e.begin(), a.begin(), a.end());

  vector<complex<double>> f;
  f.push_back({1, -1});       // analogous to push_back(complex<double>(1,-1)), but shorter
                              // thanks to initializer list
  f.emplace_back(1, -1);      // emplace_back is analogous to push_back but it just needs the
                              // arguments to construct the value, which is created in-place
}
```

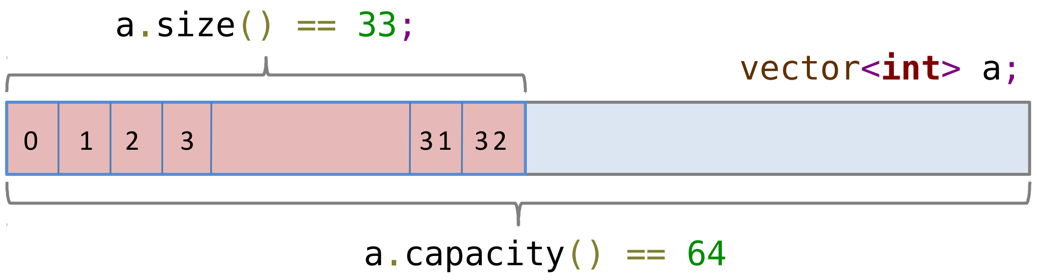
C++11

## Standard `vector` 3/3

ex06

- vector is one of the sequential containers provided by the standard library
- To support fast random access to the elements, vector elements are stored contiguously
- Vectors grow dynamically: they allocate capacity beyond what is immediately needed, as a reserve for new elements
  - When the capacity is exceeded, a new block of contiguous memory is automatically allocated (e.g. by a factor of 2 larger than the previous one)
  - Vector class provides two member functions concerning its memory block: vector::capacity() and vector::reserve(n)

```cpp
vector<int> a;
for (int i = 0; i < 33; ++i) {
  a.push_back(i);
  cout << "a capacity: "
       << a.capacity()
       << endl;
}
```



`a.reserve(n)` ← lets us set the initial vector capacity

# The **auto** type specifier and type aliases

ex07

- When a definition of a variable has an initializer, we can let the variable have the type of the initializer using the `auto` type "placeholder"

C++11

```
vector<complex<double>> v = {{1,0}, {1,-1}, {0,1}, {0,-1}};

auto size = v.size();

cout << "v elements: ";
for (auto x : v)
  cout << x << " ";
cout << endl;
```

The `size` variable has type `vector<complex<double>>::size_type`

x has type `complex<double>`

  - When defining a variable with `auto`, prefer the "=" syntax

- Sometime we need a new name for a type, e.g. because it is too long or complex or because we need different types to have same name

C++11

```
using vec_compd = vector<complex<double>>;

vec_compd v = {{1,0}, {1,-1}, {0,1}, {0,-1}};
```

Older syntax using typedef

```
typedef vector<complex<double>> vec_compd;
```

# Functions

- Functions are named computations. We can identify important parts of our problems and create named computations corresponding to those parts
- A function is defined by specifying its return type, followed by the function name and then by a parameter list enclosed in () and finally the function body, which is enclosed in {}
- A function is uniquely represented by its name and the list of parameter types. They form the so called function signature

```
int gcd(int v1, int v2)  // return the greatest common divisor
{
  while (v2) {
    int temp = v2;
    v2 = v1 % v2;
    v1 = temp;
  }
  return v1;
}
```

## Argument passing

- In the previous example, parameters are **passed by value**, i.e. they are initialized by copying the corresponding argument
- When we pass built-in types by value, the cost of copying them into the function parameters is negligible
- However, if we need to pass a long vector or string, copying them can be time consuming
- Often, we also need to define functions that modify the arguments passed, or we need them to return additional information to the caller

# References and const references

- A **reference** to an object is another name, or synonym for that object

```
vector<double> samples;
vector<double>& s = samples;  // s is a synonym for samples
```

- In the example above, variable `s` is defined as a reference (synonym) to `samples`
- Any operation performed on `s` is equivalent to doing the same to `samples`, and vice versa
- A reference always refers to the object to which it was initialized (no null references)
- The main use of references is for specifying arguments and return values for functions in general
- When we add a **`const`** to a reference, we can only use the reference to read values from the original object, but we cannot change its values (no write access)

# Argument passing: const references

ex08

```
double mean(const vector<double>& samples)
{
  auto size = samples.size();

  if (size == 0)
    throw domain_error("mean of an empty vector");

  double sum = 0;
  for (const auto& x : samples)
    sum += x;
  return sum/size;
}
```

samples is a synonym for the actual argument (a vector) passed in the function call

each element in samples is passed to x as const reference

- The type that we specify for the function argument is called "reference to `const vector` of `double`"
  - **No copy** of the actual argument passed with the function call (efficient)
  - The actual argument is passed as a read-only object
- In the range-for within the function body, x is a const reference to each element in samples (**no copy performed**)

## Argument passing: non-const reference

ex08

- There are situations where passing parameters by value or using const references don't work. For instance, in a simple function that swaps two values we need to use non-const references:

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

int main()
{
  int i = 5, j = 20;
  swap(i,j);    // After this call: i = 20, j = 5

}
```

- This function has no return value. Its return type is **`void`**, a C/C++ built-in type used in a few restricted ways, e.g. to name a return type
- The purpose of the swap function is to change its argument values. So its parameters are declared as references

## Argument passing: summary

- When function parameters are built-in types, they can be passed by value if the function does not need to change their value
- Whenever the function does not need to change the parameter's value, and the parameter has a type that can be time consuming to copy, then the parameter should be a const reference
- When the function intends to change the arguments values, the corresponding parameters must be declared as non-const references
  - Arguments passed as non-const reference parameters must be non-temporary objects (lvalues)