# Introduction to C++

## Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1 Introduction to C++
- C++ language

# What is C++?

- ISO/IEC 14882:2011, ISO/IEC 14882:2014, ISO/IEC 14882:2017, ISO/IEC 14882/2020.
- *“A lightweight programming abstractions language”* (B. Stroustrup).
- Designed by Bjarne Stroustrup at Bell Labs: 1979-1983.
  - Constant evolution and improvement during last decade.
  - Evolving a language with millions lines of code is different from designing a new programming language.
- If somebody tells you that he has a perfect programming language:
  - Either he is a naïve.
  - Or he is a salesman.

# Where does it come from?

- High-level influences.
  - General purpose abstractions languages.
    - Simula.
  - Domain specific abstraction languages.
    - FORTRAN, COBOL.
- Low-level influences.
  - Direct hardware mapping.
    - Assembler.
  - Minimal hardware abstraction.
    - BCPL.
    - C.
- Influences on other programming languages.
  - Java.
  - C#.
  - C (C11).

# Evolution

- 1979: "C with classes" starts to be designed.
- 1983: First version of C++.
- 1989: C++ 2.0.
- 1998: C++98 → ISO/IEC 14882:1998
- 2003: C++03 → ISO/IEC 14882:2003.
  - Technical Corrigendum
- 2005: C++ TR1 → ISO/IEC TR 19768.
  - C++ Library Extensions
- 2011: C++11 → ISO/IEC 14882:2011.
- 2014: C++14 → ISO/IEC 14882:2014.
- 2017: C++17 → ISO/IEC 14882:2017.
- 2020: C++20 → ISO/IEC 14882:2020.

## 2 Basic aspects

- A first program
- Basic input/output
- Vectors
- Functions and argument passing
- Exceptions

# Hello

- Header file: **iostream**.
- Main program: **main**.
  - It is program entry point.
- Import from namespace: **std**
- Standard output stream: **cout**.
  - Is a global variable
- Output operator: **<<**.
  - Operator to send data to stream.
  - Defined for most types.
- End of line: **endl** (like **"\n"**).
- Exit code: **0** (returned to OS).

**hello.cpp**

```cpp
#include <iostream>

int main() {
  using namespace std;

  cout << "Hello" << endl;
  cerr << "Error\n";

  return 0;
}
```

## 2 Basic aspects

- A first program
- **Basic input/output**
- Vectors
- Functions and argument passing
- Exceptions

# Standard input/output

- Header: **<iostream>**.
- Namespace: **std**.
- Global objects
  - **cin**: Standard input.
  - **cout**: Standard output
  - **cerr**: Standard error.
  - **clog**: Standard log.
- Operators:
  - Dump data to a stream:

    ```
    std::cout << "Values: " << x << " , " << y << std::endl;
    ```

  - Value reading:

    ```
    std::cin >> x >> y;
    ```

# Input/output example

## Reading a name

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

int main() {
  std::cout << "Enter your name: \n";

  std::string name;
  std::cin >> name;

  std::cout << "Hello, " << name << "!\n";
  return 0;
}
```

# Input/output example

## Name reading

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

int main() {
  using namespace std;
  cout << "Enter your name: \n";

  string name;
  cin >> name;

  cout << "Hello, " << name << "!\n";
  return 0;
}
```

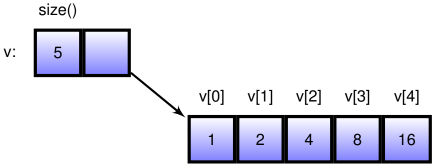
- **using namespace** avoids repeating **std::** qualifications.

## 2 Basic aspects

- A first program
- Basic input/output
- Vectors
- Functions and argument passing
- Exceptions

# Collection of values

- **vector** allows storing and processing a set of values from the same type.
  - C++ also has *arrays* but they are too limited and simple.
- A **vector**:
  - Has a sequence of elements.
  - Elements can be accessed by index.
  - Includes size information




- Alternative to direct use of arrays.

# Basic use

## Using a vector

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

int main() {
  using namespace std;

  vector<int> v(5);
  v[0] = 1;
  v[1] = 2;
  v[2] = 4;
  v[3] = 8;
  v[4] = 16;

  cout << v[3] << endl;

  return 0;
}
```

- Header file: **<vector>**
- Must specify element type.
  - All from the same type.
- Constructor argument: Initial size.
- Indices beyond size (included) cannot be accessed.

# Vectors and types

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

int main() {
  using namespace std;

  vector<string> v(2);
  v[0] = "Daniel";
  v[1] = "Carlos";

  vector<int> w(2);
  w[0] = 1969;
  w[1] = 2003;

  cout << v[0] << " : " << w[0] << endl;
  cout << v[1] << " : " << w[1] << endl;

  return 0;
}
```

# Vectors and initialization

- A vector with size initializes all its values to the default value for the element type.
  - Numeric values: **0**
  - String values: **“”**

- If no initial value is provided, vector has a size of **0**.

```
vector<double> v; // Vector with 0 elements
```

- A different initial value can be provided.

```
vector<double> v(100, 0.5); // 100 elements initialized to 0.5
```

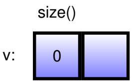
## Initializing at definition

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

int main() {
  using namespace std;

  vector<string> v { "Daniel", "Carlos" };

  vector<int> w { 1969, 2003 };

  cout << v[0] << " : " << w[0] << endl;
  cout << v[1] << " : " << w[1] << endl;

  return 0;
}
```

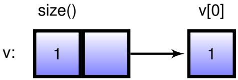
# Growing vectors

- A **vector** may *grow* when new elements are added.
  - Operation **push_back()**: Adds an element at the end of the vector.

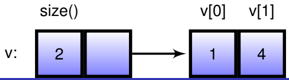
```
vector<int> v;
```




```
v.push_back(1);
```




```
v.push_back(4);
```



# Traversing a vector

- Size value can be obtained from a vector through **size** *member function*.

```
cout << v.size();
```

- **size()** allows defining a loop to traverse elements in a vector.

```
for (int i=0; i<v.size(); ++i) {
  cout << "v[" << i << "] = " << v[i] << endl;
}
```

# Range based traversal

- A vector can be traversed with a range based loop.

```
vector<int> v1 { 1, 2, 3, 4 };
for (auto x : v1) {
  cout << x << endl;
}

vector<string> v2 { "Carlos", "Daniel", "José", "Manuel" };
for (auto x : v2) {
  cout << x << endl;
}
```

# Example: Statistics

- Goal: Read from standard input a sequence of students marks and dump to standard output minimum, maximum and average mark.
  - Finish reading when reaching to end-of-file.
  - Finish reading if a value cannot be correctly read (e.g. letters instead of numbers).
  - Number of values is unknown (and not asked).

## marks.cpp

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

int main() {
  using namespace std;

  vector<double> marks;

  double x;
  while (cin >> x) { // x OK?
    marks.push_back(x);
  }

  double average = 0.0;
  double max_val = marks[0];
  double min_val = marks[0];

...
```

## marks.cpp

```
...

  for (auto x : marks) {
    media += x;
    max_val = (x>max_val) ? x : max_val;
    min = (x<min_val) ? x : min_val;
  }
  average /= marks.size();

  cout << "Average: " << average << "\n";
  cout << "Máx: " << max_val << "\n";
  cout << "Mín: " << min_val << "\n";

  return 0;
}
```

# Example: Unique words

- Goal: Dump the sorted list of unique words from a text.
  - Text is read from standard input until end-of-file.
  - The list of words is printed to standard output.

## unique.cpp

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

int main() {
  using namespace std;

  vector<string> dic;
  string p;

  while (cin >> p) {
    dic.push_back(p);
  }

...
```

## unique.cpp

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

  cout << endl;
  cout << dic[0] << endl;
  for (unsigned i=1; i<dic.size(); ++i) {
    if (dic[i-1] != dic[i]) {
      cout << dic[i] << endl;
    }
  }
  return 0;
}
```

## 2 Basic aspects

- A first program
- Basic input/output
- Vectors
- **Functions and argument passing**
- Exceptions

# Functions

- **Declaration**: Includes parameters and return type.
  - Two alternate syntaxes.

```
double area(double width, double height);
auto area(double width, double height) -> double;
```

- **Definition**: Allows return type deduction.

```
auto area(double width, double height) {
  return width * height;
}
```

# Pass by value

- Only valid argument passing in C.
- Function is passes a copy from the argument in the call.

```
int increment(int n) {
  ++n;
  return n;
}

void f() {
  int x = 5;
  int a = increment(x);
  int b = increment(x);
  int c = increment(42);
}
```

# Pass by constant reference

- Passes the object address but disallows modifications within function body.
  - Conceptually equivalent to pass by value.
  - Physically equivalent to pass by pointer.

```
double maxref(const std::vector<double> & v) {
  if (v.size() == 0) return std::numeric_limits<double>::min();
  double res = v[0];
  for (int i=1; i<v.size(); ++i) {
    if (v[i]>res) res = v[i];
  }
  return res;
}

void f() {
  vector<double> vec(1000000);
  // ...
  cout << "Max: " << maxref(vec) << "\n";
}
```

# Pass by reference

- Removes constraint of not modifying parameter within function.

```
void fill (std::vector<int> & v, int n) {
  for (int i=0; i<n; ++i) {
    v.push_back(i);
  }
}

void f() {
  using namespace std;
  vector<int> v;     // v.size() == 0
   fill(v, 100); // v.size() == 100
}
```

- Copy not passed, but address.
  - Access to the argument object.

## 2 Basic aspects

- A first program
- Basic input/output
- Vectors
- Functions and argument passing
- Exceptions

# Exceptions

- C++ exception model exhibits differences with other languages.
- An exception may be any type.

```
class negative_time {};
```

- When a function detects an exceptional situation throws (**throw**) an exception.

```
void print_velocity (double s, double t) {
  if (t > 0.0) {
    cout << s/t << "\n";
  }
  else {
    throw negative_time{};
  }
}
```

# Exception handling

- Caller may handle an exception with a **try-catch** block.

```
void f() {
  double s = get_space();
  double t = get_time();
  try {
    print_velocity(s,t);
  }
  catch (negative_time) {
    cerr << "Error: Negative time\n";
  }
}
```

- If an exception is not handled → it propagates to next level.

```
void f() {
  double s = get_space(), t = get_time();
  print_velocity(s,t);
}
```

# Standard exceptions

- Several predefined exceptions in the standard library.
  - **out_of_range**, **invalid_argument**, ...
  - All inheriting from **exception**.
  - All have a **what()** member function.

```
int main()
  try {
    f();
    return 0;
  }
  catch (out_of_range & e) {
    cerr << "Out of range:" << e.what() << "\n";
    return −1;
  }
  catch (exception & e) {
    cerr << "Exception: " << e.what() << "\n";
    return −2;
  }
}
```

## 3 Dynamic memory

- The free store
- Smart pointers

# Free store memory

- The **free store** holds the memory that can be acquired and released.
- **IMPORTANT**: C++ is not a language with automated resource management.
  - If you acquire a resource you must free it.
  - Acquired memory must be released.

*C++ is my favorite garbage collected language because it generates so little garbage.*
***Bjarne Stroustrup***

# Memory allocation operator

- Operator **new** allows allocating memory from the free store.

```
int * p = new int; // Allocates memory for an int
char * q = new char[10]; // Allocates memory for 10 char
```

- *Effect*:
  - Operator **new** returns a pointer to start of allocated memory.
  - An expression **new T** returns a value of type **T***.
  - An expression **new T[sz]** returns a value of type **T***.

# Access problems

- A pointer variable is not automatically initialized to a value.
  - Dereferencing an uninitialized pointer leads to undefined behavior.

```
int * p;
*p = 42; // Undefined behavior
p[0] = 42; // Undefined behavior
```

- A pointer variable initialized to a sequence can only be accessed within the sequence limits.

```
int * v = new int[10];
v[0] = 42; // OK
x = v[-1]; // Undefined
x = v[15]; // Undefined
v[10] = 0; // Undefined
```

# Null pointer

- A pointer can be initialized to the *null pointer value* to signal that points to no object.
  - **nullptr** literal.

```
int * p = nullptr;
char * q = nullptr;
if (p != nullptr) { /* ... */ }
if (q == nullptr) { /* ... */ }
```

# Memory allocation and initialization

- Operator **new** does not initialize the allocated object.

```
int * p = new int;
x = *p; // x has an unknown value
```

  - Initial value can be specified in curly braces.

```
p = new int{42}; // *p == 42
p = new int{}; // *p == 0
```

- If a sequence is allocated with **new**, no object is initialized.

```
int * v = new int[10];
```

  - Initial values can be specified in curly braces.

```
v = new int [4]{1,2,3,4}; // v[0] = 1, v[1] = 2, v[2] = 3, v[3] = 4
v = new int[4]{1, 2};     // v[0] = 1, v[1] = 2, v[2] = 0, v[3] = 0
v = new int [4]{};        // v[0] = 0, v[1] = 0, v[2] = 0, v[3] = 0
v = new int [4]{1,2,3,4,5}; // Error. Too many initializers .
```

# Memory deallocation operator

- Operator **delete** allows to release memory and mark it as non-allocated.
- It can only be applied to:
  - Memory returned by a call to operator **new** and currently allocated.
  - A null pointer value.

```
int * p = new int{10};
*p = 20;
delete p; // Release p
```

- It is an error to invoke twice **delete** on the same pointer value.

```
int * p = new int{10};
delete p; // Release p
delete p; // Undefined behavior
```

# Array deallocation

- There is a different version of **delete** to deallocate *arrays*.

```
int * p = new int{10};
int * v = new int[10];
delete p;    // Release p
delete []v;  // Release v
```

- **Important**: The right deallocation version must be used.
  - If memory is allocated with **new T** it must be released with **delete**.
  - If memory is allocated with **new T[n]** it must be released with **delete[]**.

```
int * p = new int{10};
int * v = new int[10];
delete []p;  // Undefined behavior
delete v;    // Undefined behavior
```

# Reasons to deallocate

- If memory is allocated and not released it remains allocated.

```
void f() {
  int * v = new int[1024*1024];
  // ...
}
```

  - Each time **f()** is invoked 8 MB are lost (assuming **sizeof(int)==8**).
- Problems with **memory leaks**:
  - Each memory allocation might require more time.
  - If program runs for a long time, memory could be exhausted.
- If memory is exhausted **bad_alloc** exception is thrown.

## 3 Dynamic memory

- The free store
- Smart pointers

# Why a smart pointer?

- A **smart pointer** encapsulates a pointer and automatically manages the associated memory.
  - Its destructor automatically releases the associated memory.

- Types of smart pointers:
  - **unique_ptr**: Pointer to object with no copying.
  - **shared_ptr**: Pointer with associated reference counter.
  - **weak_ptr**: Auxiliary pointer for **shared_ptr**.

# Reference counting

- A **shared_ptr** keeps a reference counter:
  - When pointer is copied, reference counter is incremented.
  - When pointer is destroyed, reference counter is decremented.
  - If counter reaches to zero, the pointed object is destroyed.

```
void f() {
  shared_ptr<string> p1{new string{"Hello"}};
  shared_ptr<string> p2{p1}; // references -> 2

  auto n = p1->size(); // string::size(). p1 used as ptr
  *p1 = "Bye";
  if (p2) {
    cerr << "Busy\n";
  }

  p1 = nullptr; // references -> 1
  // ...
} // references -> 0 ==> Destruction
```

# Unique pointers

- **unique_ptr** offers a non-shared pointer that cannot be copied.

```
void f(string & s, int n) {
  unique_ptr<int> p = new int{50};

  string tmp = s; // Might throw an exception
  if (n<0) return;

  *p = 42;
} // Release p
```

# Simplified creation

- Creation function:

```
auto p = std::make_shared<record>("Daniel", 42);
auto q = std::make_unique<string>("Hello");
```

- Allocate object and meta-data in a single operation.

## 4 User defined types

- Classes
- Constructors
- Destructor

# Classes in C++

- A class can be defined with **struct** or **class**.
  - The only difference is by default visibility.

```
struct date {
  // public by default
};

class date {
  // private by default
};
```

# Member functions

- Can only be invoked for an object of the defined type.

## Point

```
struct point {
  double x, y;
  double modulo();
  double move_to(double cx, double cy);
}
```

## Using a point

```
void f() {
  point p{2.5, 3.5};
  p.move_to(5.0, 7.5);
  cout << p.modulo() << "\n";
}
```

# Visibility

- Visibility levels for class members:
  - **public**: Anybody may access.
  - **private**: Only for class members.
  - **protected**: Derived classes members may also access.

```
class date{
public:
   // Public members
protected:
   // Protected members
private:
   // Private members
};
```

4 User defined types

- Classes
- Constructors
- Destructor

# Constructor

- A **constructor** is a special member function.
  - Used to initialize object from the type defined by a class.
  - Syntax ensures constructor invocation.

## Definition

```
class point {
public:
  point(double cx, double cy) :
    x{cx}, y{cy} {}
  // ...
private:
  double x;
  double y;
};
```

## Use

```
void f() {
  point p{1.5, 1.5}; // Construct point
  point q;           // Error: Missing args
  point r{p};        // OK. Copy
}
```

4 User defined types

- Classes
- Constructors
- Destructor

# Objects destruction

- A **destructor** is a special member function is executed **automatically** when an object exits scope.
  - Has no return type.
  - Does not take any parameter.
  - Class name with prepended .

## Definition

```
class vector {
public:
  // ...

  vector(int n) : size{n}, vec{new double[size]}
    {}

  ~vector() { delete [] vec; }

private:
  int size;
  double * vec;
};
```

# Destructor invocation

- Destructor is automatically invoked.

## Automatic invocation

```
void f() {
  vector v(100);
  for (int i=0; i<100; ++i) {
    v[i] = i;
  }
  // ...
  for (int i=0; i<100; ++i) {
    cout << v[i] << "\n";
  }
} // Destructor invocation
```

# Books

- **Programming – Principles and Practice Using C++**. 2nd Edition. Bjarne Stroustrup. Addison-Wesley, 2014.
- **A Tour of C++**. 2nd Edition. Bjarne Stroustrup. Addison-Wesley, 2018.
- **The C++ Programming Language**. 4th Edition. Bjarne Stroustrup. Addison Wesley, 2013.
- **The C++ Standard Library: A Tutorial and Reference** 2nd Edition. Nicolai Josutis. Addison Wesley, 2012.

# Other resources

- C++ Reference. **http://en.cppreference.com/w/cpp**.
- ISO C++ Foundation. **https://isocpp.org/**.
- C++ Super-FAQ. **https://isocpp.org/faq**.
- C++ Core Guidelines. **http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines**.

# Introduction to C++

## Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid