uc3m | Universidad Carlos III de Madrid

Introducción a C++

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)

Departamento de Informática
Universidad Carlos III de Madrid

1 Introducción a C++

2 Aspectos básicos

3 Memoria dinámica

4 Tipos definidos por el usuario

5 Referencias

1 Introducción a C++
- El lenguaje C++

uc3m | Universidad Carlos III de Madrid

C++

- Un lenguaje de **programación de sistemas**:
  - Abstracciones ligeras.
  - Genera código binario.
  - Altamente eficiente.
  - Utilizado en muchos dominios.
- Diversos **estilos** soportados:
  - Abstracciones de datos.
  - Programación orientada a objetos.
  - Programación genérica.
  - Programación funcional.
  - Programación asíncrona.
  - Concurrencia.

- Diseñado por **Bjarne Stroustrup**.

ISO International Organization for Standardization

C++

- Serie ISO/IEC 14882.

1 Introducción a C++

2 Aspectos básicos

3 Memoria dinámica

4 Tipos definidos por el usuario

5 Referencias

2 Aspectos básicos
- Un primer programa
- Entrada/salida básica
- Vectores
- Funciones y paso de parámetros
- Excepciones

Hola

hola.cpp

```
#include <iostream>

int main() {
  using namespace std;

  cout << "Hello C++" << endl;
  cerr << "Error message\n";

  return 0;
}
```

- Archivo de cabecera: **iostream**.
- Importación de espacio de nombres: **std**.
- Programa principal: **main**.
  - Es el punto de entrada al programa.
- Flujo de salida estándar: **cout**.
  - Es una variable global.
- Operador de salida: **<<**.
  - Envia datos a la salida estándar.
  - Definido para la mayoría de los tipos.
- Salto de línea: **endl** como **"\n"**.
- Código de salida: **0** (devuelto a SO).

2 Aspectos básicos
- Un primer programa
- Entrada/salida básica
- Vectores
- Funciones y paso de parámetros
- Excepciones

Entrada salida estándar

- Cabecera: **<iostream>**.
- Espacio de nombres: **std**.
- Objetos globales:
  - **cin**: Entrada estándar.
  - **cout**: Salida estándar.
  - **cerr**: Salida de errores.
  - **clog**: Salida de log.
- Operadores:
  - Volcado de un dato en un flujo:

```
std::cout << "Valores: " << x << " , " << y << "\n";
```

  - Lectura de valores:

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

Ejemplo de entrada/salida

Lectura de nombre

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

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

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

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

Ejemplo de entrada/salida

Lectura de nombre

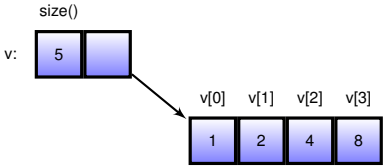
```
#include <iostream>
#include <string>

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

  string name;
  cin >> name;

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

- **using namespace** evita repetir cualificaciones **std::**.

2 Aspectos básicos
- Un primer programa
- Entrada/salida básica
- Vectores
- Funciones y paso de parámetros
- Excepciones

Colecciones de valores

- **vector** permite almacenar y procesar un conjunto de valores de un mismo tipo.
- Un **vector**:
  - Tiene una secuencia de elementos.
  - Se puede acceder a los elementos por su índice.
  - Incluye información de su tamaño.



- Alternativa a usar arrays directamente.

Uso básico

Uso de vector

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

int main() {
  using namespace std;

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

  cout << v[2] << "\n";
}
```

- Archivo de cabecera: **<vector>**
- Se debe indicar el tipo del elemento.
  - Todos del mismo tipo.
- Parámetro del constructor: **Tamaño inicial**.
- No se puede acceder a indices más allá del tamaño (inclusive).
  - **Comportamiento no definido**.

Vectores y tipos

```
#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] << "\n";
  cout << v[1] << " : " << w[1] << "\n";
}
```

Vectores e iniciación

- Un vector con tamaño inicia todos sus valores al valor por defecto del tipo.
  - Valores numéricos: **0**
  - Valores de cadena: **“”**

- Si no se indica tamaño inicial, el vector tiene tamaño **0**.

```
vector<double> v; // Vector con 0 elementos
```

- Se puede suministrar un valor inicial distinto.

```
vector<double> v(100, 0.5); // 100 posiciones iniciadas a 0.5
```

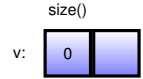
Iniciación en la declaración

```
#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] << "\n";
  cout << v[1] << " : " << w[1] << "\n";
}
```

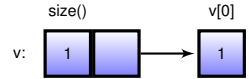
Vectores que crecen

- Un **vector** puede *crecer* cuando se añaden elementos.
  - Operación **push_back()**: Añade un elemento al final del vector.

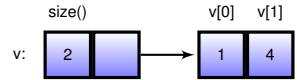
```
vector<int> v;
```




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




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



Recorrido de un vector

- Se puede consultar el tamaño de un vector mediante la *función miembro* **size**.

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

- **size()** permite definir un bucle para recorrer los elementos de un vector.

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

Recorrido basado en rango

- Se puede usar un recorrido basado en rango para un vector.

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

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

Ejemplo: Estadísticas

- Objetivo: Leer de la entrada estándar una secuencia de calificaciones y volcar en la salida estándar la calificación mínima, la máxima y la calificación media.
  - Finalizar la lectura si se llega a fin de fichero.
  - Finalizar la lectura si no se lee un valor correctamente (p. ej. letras en lugar de números).
  - Se desconoce (y no se pregunta) el número de valores.

notas.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];

...
```

notas.cpp

```
...
  for (auto m: marks) {
    average += m;
    max_val = (m >max_val) ? m : max_val;
    min_val = (m <min_val) ? m : min_val;
  }
  average /= static_cast<double>(marks.size());

  cout << "Average: " << average << "\n";
  cout << "Max: " << max_val << "\n";
  cout << "Min: " << min_val << "\n";
}
```

Ejemplo: Palabras únicas

- Objetivo: Volcar la lista ordenada de palabras únicas de un texto.
  - El texto se lee de la entrada estándar hasta fin de fichero.
  - La lista de palabras se imprime en la salida estándar.

unique.cpp

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

int main() {
  using namespace std;

  vector<string> words;
  string w;

  while (cin >> w) {
    words.push_back(w);
  }

...
```

unique.cpp

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

  cout << "\n";
  cout << words[0] << "\n";
  for (std::size_t i=1; i<words.size(); ++i) {
    if (words[i-1] != words[i]) {
      cout << words[i] << "\n";
    }
  }
}
```

2 Aspectos básicos
- Un primer programa
- Entrada/salida básica
- Vectores
- Funciones y paso de parámetros
- Excepciones

Funciones

- **Declaración**: Incluye parámetros y tipo de retorno.
  - Dos sintaxis alternativas.

```
double area(double ancho, double alto);
auto area(double ancho, double alto) -> double;
```

- **Definición**: Permite deducción automática de tipo de retorno

```
auto area(double ancho, double alto) {
  return ancho * alto;
}
```

Paso por valor

- Único paso de parámetros válido en C.
- Se pasa a la función una copia del argumento especificado en la llamada.

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

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

Paso por referencia constante

- Pasa la dirección del objeto pero impide su alteración dentro de la función.
  - Conceptualmente equivale a paso por valor.
  - Físicamente equivalente a paso de un puntero.

```
double maxref(const std::vector<double> & v) {
  double res = std::numeric_limits<double>::min();
  for (auto x : v) {
    if (x>res) {
      res = x;
    }
  }
  return res;
}
```

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

Paso por referencia

- Elimina la restricción de no modificar el parámetro dentro de la función.

```
void rellena(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
  rellena(v, 100); // v.size() == 100
}
```

- No se pasa una copia.
  - Se tiene acceso al propio objeto.

2 Aspectos básicos
- Un primer programa
- Entrada/salida básica
- Vectores
- Funciones y paso de parámetros
- Excepciones

Excepciones

- El modelo de excepciones de C++ presenta diferencias con otros lenguajes.
- Una excepción puede ser cualquier tipo definido por el usuario.

```
class tiempo_negativo {};
```

- Cuando una función detecta una situación excepcional lanza (**throw**) una excepción.

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

Tratamiento de excepciones

- El llamante puede tratar una excepción con un bloque **try-catch**.

```
void f() {
  double s = lee_espacio();
  double t = lee_tiempo();
  try {
    imprime_velocidad(s,t);
  }
  catch (tiempo_negativo) {
    cerr << "Error: Tiempo negativo\n";
  }
}
```

- No es necesario tratar una excepción → se propaga.

```
void f() {
  double s = lee_espacio(), t = lee_tiempo();
  imprime_velocidad(s,t);
}
```

Excepciones estándar

- Varias excepciones predefinidas en la biblioteca estándar.
  - **out_of_range**, **invalid_argument**, ...
  - Todos heredan de **exception**
  - Todos tienen una función miembro **what()**.

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

1 Introducción a C++

2 Aspectos básicos

3 Memoria dinámica

4 Tipos definidos por el usuario

5 Referencias

3 Memoria dinámica
- El almacén libre
- Punteros inteligentes

Memoria del almacén libre

- El **almacén libre** contiene la memoria que se puede adquirir y liberar.

- **IMPORTANTE**: C++ no es un lenguaje con gestión automática de recursos.
  - Si se adquiere un recurso, se debe liberar.
  - La memoria adquirida hay que liberarla.

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

Operador de asignación de memoria

- El operador **new** permite asignar memoria del almacén libre.

```
int * p = new int; // Asigna memoria para un int
char * q = new char[10]; // Asigna memoria para 10 char
```

- *Efecto*:
  - El operador **new** devuelve un puntero al inicio de la memoria asignada.
  - Una expresión **new T** devuelve un valor de tipo **T***.
  - Una expresión **new T[sz]** devuelve un valor de tipo **T***.

Problemas de acceso

- Una variable de tipo puntero no se inicia de forma automática a ningún valor.
  - Si se desreferencia un puntero no iniciado se tiene un comportamiento no definido.

```
int * p;
*p = 42; // Comportamiento no definido.
p[0] = 42; // Comportamiento no definido.
```

- Una variable de tipo puntero iniciada a una secuencia solamente puede accederse dentro de sus límites establecidos.

```
int * v = new int[10];
v[0] = 42; // OK
x = v[-1]; // No definido
x = v[15]; // No definido
v[10] = 0; // No definido
```

El puntero nulo

- Se puede iniciar un puntero al valor *puntero-nulo* para indicar que no apunta a ningún objeto.
  - Literal **nullptr**.

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

Asignación de memoria e iniciación

- El operador **new** no inicia el objeto asignado.

```
int * p = new int;
x = *p; // x tiene un valor desconocido
```

  - Se puede indicar el valor inicial entre llaves.

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

- Si se reserva una secuencia con **new** no se inicia ninguno de los objetos.

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

  - Se puede indicar los valores iniciales entre llaves.

```
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 demasiados iniciadores
```

Operador de desasignación de memoria

- El operador **delete** permite liberar memoria y marcarla como no asignada.
- Pude aplicarse solamente a:
  - Memoria devuelta por el operador **new** y actualmente asignada.
  - El puntero nulo.

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

- Es un error invocar dos veces a **delete** sobre un mismo puntero.

```
int * p = new int{10};
delete p; // Libera p
delete p; // Comportamiento no definido
```

Desasignación de arrays

- Existe una versión diferente para liberar *arrays*.

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

- **Importante**: Se debe usar la versión correcta de desasignación.
  - Si se reserva memoria con **new T** debe liberarse con **delete**.
  - Si se reserva memoria con **new T[n]** debe liberarse con **delete[]**.

```
int * p = new int{10};
int * v = new int[10];
delete []p; // Comportamiento no definido
delete v; // Comportamiento no definido
```

Razones para desasignar

- Si se reserva memoria y no se libera esta queda asignada.

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

  - Cada vez que se invoca a **f()** se pierden 8 MB (si **sizeof(int)==8**).
- Problemas con los goteos de memoria:
  - En cada asignación de memoria puede requerirse más tiempo.
  - Si el programa se ejecuta durante mucho tiempo, puede acabar agotándose la memoria.
- Se se agota la memoria se lanza la excepción **bad_alloc**.

3 Memoria dinámica
- El almacén libre
- Punteros inteligentes

Punteros

- Un **puntero inteligente** encapsula un puntero y gestiona de forma automática la gestión de la memoria asociada.
  - Su destructor libera automáticamente la memoria asociada.

- Tipos de punteros inteligentes:
  - **unique_ptr**: Puntero a un objeto que no admite copias.
  - **shared_ptr**: Puntero con contador de referencias asociado.
  - **weak_ptr**: Puntero auxiliar para **shared_ptr**.

Cuenta de referencias

- Un **shared_ptr** mantiene un contador de referencias:
  - Cuando se copia se incrementa el contador de referencias.
  - Cuando se destruye se decrementa el contador.
  - Si el contador llega a cero el objeto se destruye.

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

  auto n = p1->size(); // string::size(). p1 usado como ptr
  *p1 = "Adios";
  if (p2) { cerr << "Ocupado\n"; }

  p1 = nullptr; // referencias -> 1
  // ...
} // referencias -> 0 ==> Destrucción
```

Punteros únicos

- **unique_ptr** ofrece un puntero no compartido que no se puede copiar.

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

  string tmp = s; // Podría lanzar excepción
  if (n<0) return;

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

Creación simplificada

- Función de creación.

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

- Asigna el objeto y los meta-datos en una única operación.

1 Introducción a C++

2 Aspectos básicos

3 Memoria dinámica

4 Tipos definidos por el usuario

5 Referencias

4 Tipos definidos por el usuario
- Clases
- Constructores
- Destructor

Clases en C++

- Una clase se puede definir con **struct** o **class**.
  - La única diferencia es la visibilidad por defecto.

```
struct fecha {
  // Visibilidad pública por defecto
};

class fecha {
  // Visibilidad privada por defecto
};
```

Función miembro

- Solamente se puede invocar para un objeto del tipo definido.

Punto

```
struct punto {
  double x, y;
  double modulo();
  double mover_a(double cx, double cy);
};
```

Usando un punto

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

Visibilidad

- Niveles de visibilidad de los miembros de una clase:
  - **public**: Cualquiera puede acceder.
  - **private**: Solamente por miembros de la clase.
  - **protected**: Miembros de clases derivadas pueden acceder.

```
class fecha {
public:
   // Miembros públicos
protected:
   // Miembros protegidos
private:
   // Miembros privados
};
```

4 Tipos definidos por el usuario
- Clases
- Constructores
- Destructor

Constructor

- Un **constructor** es una función miembro especial.
  - Se usa para iniciar objetos del tipo definido por la clase.
  - La sintaxis obliga a invocar al constructor.

Definición

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

Uso

```
void f() {
  punto p{1.5, 1.5}; // Construye punto
  punto q;           // Error: faltan args
  punto r{p};        // OK. Copia
}
```

4 Tipos definidos por el usuario
- Clases
- Constructores
- Destructor

Destrucción de objetos

- Un **destructor** es una función miembro especial que se ejecuta **de forma automática** cuando un objeto sale de alcance.
  - No tiene tipo de retorno.
  - No toma parámetros.
  - Nombre de clase precedido de carácter ~.

Definición

```
class numvector {
public:
  // ...

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

  ~numvector() { delete []vec; }

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

Invocación de destructor

- El destructor se invoca de forma automática.

Invocación automática

```
void f() {
  numvector v(100);
  for (int i=0; i<100; ++i) {
    v[i] = i;
  }
  // ...
  for (int i=0; i<100; ++i) {
    cout << v[i] << "\n";
  }
} // Invocación de destructor
```

1 Introducción a C++

2 Aspectos básicos

3 Memoria dinámica

4 Tipos definidos por el usuario

5 Referencias

Libros

- **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.

Otros recursos

- 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**.

uc3m | Universidad Carlos III de Madrid

Introducción a C++

Arquitectura de Computadores

J. Daniel García Sánchez (coordinador)

Departamento de Informática
Universidad Carlos III de Madrid