Introducción a la programación genérica con C++

F. Palomo Lozano I. Medina Bulo A. García Domínguez

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Cádiz

Práctica 1

Índice

Características

- C++ fue inventado por Bjarne Stroustrup (1985)
- El nombre del lenguaje deriva del operador de incremento de C
- C++ es un compromiso entre el bajo y el alto nivel
- C++ es un lenguaje multiparadigma:
  - El paradigma de la programación estructurada
  - El paradigma de la programación orientada a objetos
  - El paradigma de la programación genérica

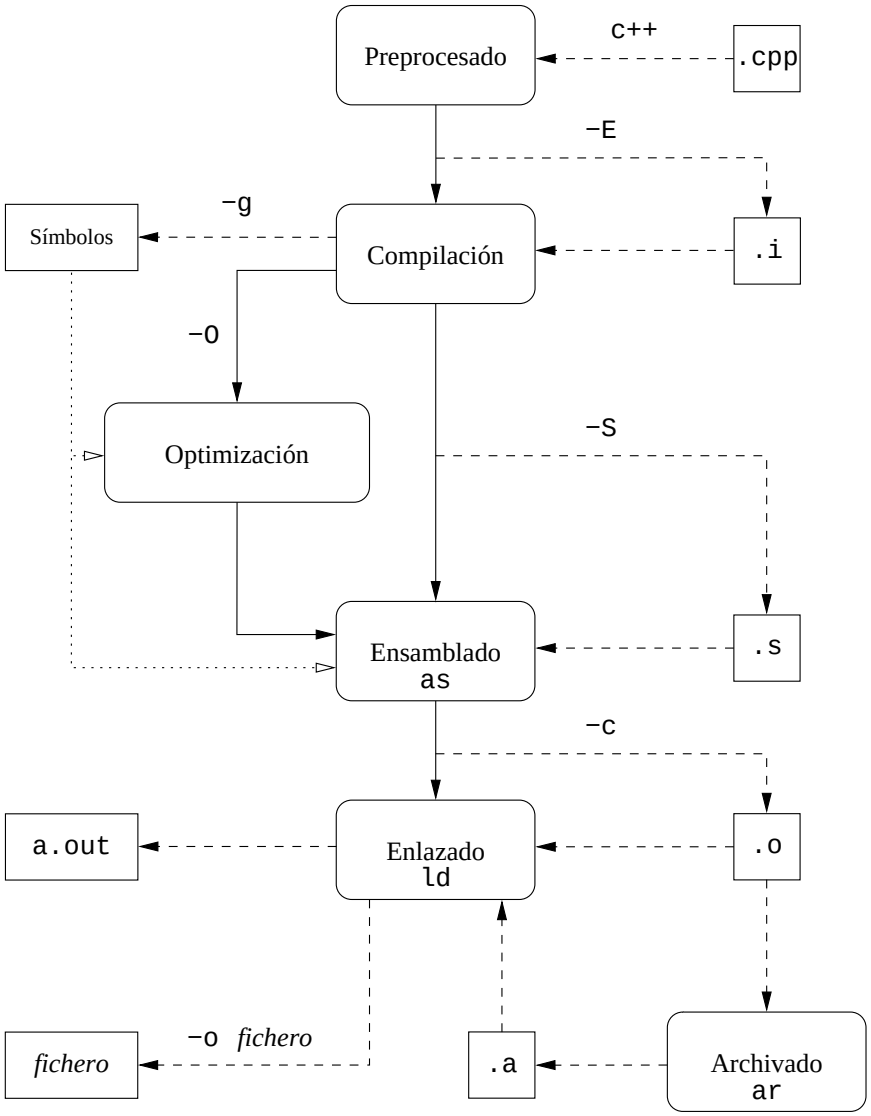
Un primer programa

```
#include <iostream>

using namespace std;

int main()
{
  cout << "¡Hola a todos!" << endl;
}
```

Otra versión:

```
int main()
{
  std::cout << "¡Hola a todos!" << std::endl;
}
```

Pasos en la traducción de un programa



Compatibilidad con C

C++

- C es un subconjunto de C++
- C++ mejora diversos aspectos de C

Por un lado, prohíbe explícitamente

- La declaración implícita de funciones
- El empleo de `int` como tipo por omisión
- La redefinición de un objeto de datos
- La inicialización de un vector con más elementos de los declarados
- La conversión implícita de entero a enumerado
- La conversión implícita de puntero genérico a otro tipo puntero

Compatibilidad con C

Por otro lado, modifica algunas características

- Incorpora `bool`: `false` y `true`
- Redefine el tamaño de los literales de carácter a `sizeof(char)`
- El tamaño de las enumeraciones depende del rango de valores declarado
- El nombre de una estructura oculta a cualquier otro nombre idéntico externo al ámbito de su declaración
- Exige que se emplee la sintaxis de prototipos en la declaración y definición de funciones
- Especifica que si en el prototipo de una función no aparece ningún parámetro, ésta no puede recibir parámetros
- Aparecen nuevas palabras reservadas

Mejoras en el lenguaje

Constantes

```
const double pi = 4.0 * atan(1.0);
```

- Modifica el enlace de las constantes de externo a interno, haciendo factible que aparezcan en los ficheros de cabecera
- Permite definir constantes cuyo inicializador no sea una expresión constante conocida en tiempo de compilación
- No se usan las macros para especificar valores constantes

Funciones en línea

```
inline double cuadrado(double x) { return x * x; }

inline double distancia(double x0, double y0,
                        double x1, double y1)
{ return sqrt(cuadrado(x1 - x0) + cuadrado(y1 - y0)); }
```

Mejoras en el lenguaje

Declaraciones

- C++ permite declaraciones casi en cualquier lugar

```
for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      double d = p[i][k] + p[k][j];
      if (d < p[i][j])
        p[i][j] = d;
    }
```

- Los nombres de las enumeraciones, estructuras y uniones son realmente tipos

```
struct fecha { unsigned short d, m, a; };
```

Mejoras en el lenguaje

Referencias

```
void actualiza(double& e, double& v, double a, double t)
{
  e += v * t + 0.5 * a * t * t;
  v += a * t;
}
```

y se emplea de la siguiente forma:

```
double e = 20.0, v = 0.0;
actualiza(e, v, 9.8, 1.0);
```

Sobrecarga

- C++ permite emplear un mismo nombre de función para realizar acciones distintas
- Las funciones se diferencian por el número o el tipo de los parámetros empleados

```
void actualiza(double& e, double v, double t) { e += v * t;}
```

Mejoras en el lenguaje

Plantillas

- Podemos utilizar la sobrecarga para crear versiones especializadas de una misma función que trabajen con distintos tipos de datos

```
inline int cuadrado(int x) { return x * x; }
inline double cuadrado(double x) { return x * x; }
```

- Pero C++ permite definir la función cuadrado de forma que permita calcular el cuadrado de cualquier objeto para el que se defina el operador $*$

```
template <typename T>
inline T cuadrado(T x) { return x * x; }
```

- Conviene utilizar referencias

```
template <typename T>
inline T cuadrado(const T& x) { return x * x; }
```

Mejoras en el lenguaje

Clases

- Se introducen en C++ como una generalización de las estructuras
- Una clase puede contener *miembros de datos* y *funciones miembro*

```
class C {
  int i;
  int f (int n) { return ++n; }
};
```

- Se especifica qué miembros son visibles desde el exterior (la *interfaz*)

```
class movil {
public:
  movil(double m);
  double espacio() const;
  double tiempo() const;
  double velocidad() const;
  double aceleracion() const;
  void aplicaFuerza(double f, double dt);
private:
  double m, e, t, v, a;
};
```

Mejoras en el lenguaje

Clases

- Constructor

```
movil::movil(double m):
  m(m), e(0.0), t(0.0), v(0.0), a(0.0)
{}
```

- Funciones observadoras

```
double movil::espacio() const { return e; }
double movil::velocidad() const { return v; }
```

- Función modificadora

```
void movil::aplicaFuerza(double f, double dt)
{
  a = f / m;
  e += v * dt + 0.5 * a * dt * dt;
  v += a * dt;
  t += dt;
}
```

- Ejemplo

```
movil proyectil(1000.0);
proyectil.aplicaFuerza(1000.0, 150.0);
cout << "Velocidad de impacto: " << proyectil.velocidad();
```

Ejemplo cronómetro

```
#include <ctime>
using std::clock;
using std::clock_t;

class cronometro {
public:
  cronometro(): activo(false) {}
  void activar() { activo = true; t0 = clock(); }
  void parar() { if (activo) { t1 = clock(); activo = false; } }
  double tiempo() const
   { return ((activo ? clock() : t1) - t0) / pps; }
private:
  bool activo;     // estado de actividad del cronómetro.
  clock_t t0, t1;  // tiempos inicial y final.
  static const double pps = CLOCKS_PER_SEC;
};

cronometro c;
c.activar();
for (long i = 0; i < 1000000000L; ++i)
  ;
c.parar();
cout << c.tiempo() << " s" << endl;
```

Entrada/Salida

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

int main()
{
  int n = 0;
  double s = 0.0;
  cout << "Datos numéricos:\n" << endl;
  while (cin) {
    cout << 'x' << n << " = ";
    double x;
    if (cin >> x) {
      s += x;
      ++n;
    }
  }
  cout << "\nLa media aritmética es " << (n ? s / n : 0)
       << '.' <<  endl;
  return 0;
}
```

Cadenas

`string` está en `<string>`

```
string completo;
string nombre = "Pepito", apellido = "Pérez";
apellido += " García";
completo = apellido + ", " + nombre;

for (string::size_type i = 0; i < nombre.size(); ++i)
  nombre[i] = toupper(nombre[i]);
```

`getline()` es muy útil

```
int main()
{
  cout << "Nombre completo: ";
  string nombre;
  getline(cin, nombre);
  cout << "Hola, " << nombre << '.' << endl;
}
```

STL

- STL: Standard Template Library
- Se compone de: *clases contenedoras* y *algoritmos genéricos*
- Un *contenedor almacena* una colección de elementos y permite realizar *operaciones* con ellos
- Los *iteradores* son una *abstracción* del concepto de *puntero*: permiten *recorrer* una colección de elementos y *acceder* a ellos individualmente.
- Siempre es posible usar un puntero donde se requiera un iterador
- Los contenedores proporcionan iteradores de una determinada categoría

Iteradores

Categorías

NOMBRE	OPERACIONES
Entrada	`*`, `->` (ambos sólo para lectura), `++`, `==` y `!=`
Salida	`*` (sólo para escritura) y `++`
Monodireccionales	Las de los de entrada y de salida
Bidireccionales	Las de los monodireccionales más `--`
Acceso directo	Las de los bidireccionales más `[]`, `<`, `<=`, `>`, `>=` `+` (con entero) y `-` (con entero, o con dos iter.)

Contenedores

Secuencias

Nombre	Cabecera	Descripción
`vector`	`vector`	Vectores
`deque`	`deque`	Colas dobles
`list`	`list`	Listas

Adaptadores de secuencias

Nombre	Cabecera	Descripción
`stack`	`stack`	Pilas
`queue`, `priority_queue`	`queue`	Colas simples y de prioridades

Contenedores

Contenedores asociativos ordenados

NOMBRE	CABECERA	DESCRIPCIÓN
`set`, `multiset`	`set`	Conjuntos y multiconjuntos
`map`, `multimap`	`map`	Aplicaciones mono y multivalor

Contenedores

Características

- Por omisión construyen contenedores vacíos
- Se pueden copiar, asignar y comparar de forma natural
- Tienen definiciones públicas de tipos
- Tienen métodos para construir iteradores

```
C<T> c1, c2;
c1 = c2;
if (c1 == c2) ...
f(c1);

C<T> c;             <- C::value_type
c.empty();
c.begin();          <- C::iterator, C::const_iterator
c.end();            <- [c.begin(), c.end())
c.size()            <- C::size_type
```

Contenedores

Secuencias: `c.push()`, `c.pop()`, `c.top()`, `c.front()` y `c.back()`

```
vector<int> t;
vector<int> u(10);
vector<int> v(10, 1);
t = u = v;
vector<int> w(v);
vector<int> x = w;
deque<int> y(x.begin(), x.end());
list<int> z(y.begin(), y.begin() + y.size() / 2);

list<int> c;
for (list<int>::iterator i = c.begin(); i != c.end(); ++i)
  *i = 0;
for (list<int>::const_iterator i=c.begin(); i!=c.end(); ++i)
  cout << *i << endl;
```

Contenedores asociativos ordenados

```
set<string> c;
set<string>::size_type n = c.size();
for (set<string>::size_type i = 0; i < n; ++i)
```

Programación genérica

```
template <typename T> bool ordenado(const vector<T>& v) {
  typedef typename vector<T>::size_type I;
  I i = 0, n = v.size();
  if (n != 0)
    for (I k = 1; k < n; ++k) {
      if (v[k] < v[i])
        return false;
      i = k;
    }
  return true;
}
template <typename I> bool ordenado(I i, I j) {
  if (i != j) {
    I k = i;
    while (++k != j) {
      if (*k < *i)
        return false;
      i = k;
    }
  }
  return true;
}
```