ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

Funciones de orden superior

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Ejercicio

Característica de los lenguajes funcionales

Usar funciones como parámetros de otras funciones.

- Función que recibe una lista de enteros y elimina los números pares de la misma.

```
bool es_par(int x) { return x % 2 == 0; }

void eliminar_pares(std::list<int> &elems) {
  auto it = elems.begin();
  while (it != elems.end()) {
    if (es_par(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

Usando iteradores

si es par lo elimina

Nosotros queremos de alguna forma una función que pueda eliminar el tipo de dato que queramos si se cumple cierta condición.

Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
eliminar_pares(v1);
std::cout << v1 << std::endl;
```

```
[1, 5, 9]
```

Nos quedan los impares de la lista que teníamos.

Ejercicio

- Función que recibe una lista de enteros y elimina los números **impares** de la misma.

```
bool es_impar(int x) { return x % 2 == 1; }

void eliminar_impares(std::list<int> &elems) {
  auto it = elems.begin();
  while (it != elems.end()) {
    if (es_impar(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

Llama a erase si el valor apuntado por el iterador es impar.

¿Pero yo reflexionando puedo pensar... No será mejor de alguna forma hacer como con los genéricos?

Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);
std::cout << v1 << std::endl;

eliminar_impares(v2);
std::cout << v2 << std::endl;
```

```
[1, 5, 9]
```

```
[6, 10, 20]
```

Ejercicio

- Función que recibe una lista de enteros y elimina los números **positivos** de la misma.

```
bool es_positivo(int x) { return x > 0; }

void eliminar_positivos(std::list<int> &elems) {
  auto it = elems.begin();
  while (it != elems.end()) {
    if (es_positivo(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

La función tiene la misma forma, pero cambia los métodos de dentro. Podemos observar que hay mucha duplicación de código

Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);
std::cout << v1 << std::endl;

eliminar_impares(v2);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3);
std::cout << v3 << std::endl;
```

```
[1, 5, 9]
```

```
[6, 10, 20]
```

```
[-2, -6]
```

¡Cuánta duplicación!

Recibir en parámetros aquello en lo que se diferencian.

```
void eliminar_positivos(std::list<int> &elems) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (es_positivo(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

```
void eliminar_impares(std::list<int> &elems) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (es_impar(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

```
void eliminar_pares(std::list<int> &elems) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (es_par(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

- La solución para unificar estas tres funciones es **parametrizarlas** en aquello en lo que se diferencian.
- ¡Pero aquí se diferencian en una **función**!
- ¿Es posible pasar funciones como parámetros en C++?

POR SUPUESTO QUE... SIIII!!!!!

Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

```
void eliminar_positivos(std::list<int> &elems) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (es_positivo(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```



```
void eliminar(std::list<int> &elems, ??? func) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (func(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

¿TIPO?

De forma que si en este caso queremos eliminar positivos tendríamos que comprobar si es positivo. Entonces le tendríamos que pasar como parámetro la función es_positivo.

Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

- **Puntero a función**
  - Mecanismo heredado de C.
- **Variable plantilla** LOS TEMPLATES DE C++.
  - Utiliza el mecanismo de plantillas de C++.
  - Dejamos que el compilador infiera el tipo.
  - Compatible con objetos función.

Siguiente vídeo

Uso de variable de plantilla

No se si en este caso con que pusiéramos que fuera de tipo bool porque todas devuelven un booleano true o false. Pero si quisiéramos otro tipo de función o es obligatorio que tenga la plantilla.

```
template <typename T>
void eliminar(std::list<int> &elems, T func) {
  auto it = elems.begin();
  while (it != elems.end()) {
    if (func(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }


std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);
std::cout << v1 << std::endl;

eliminar_impares(v2);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3);
std::cout << v3 << std::endl;
```

En vez de tener estas 3 funciones especificas, tendremos lo de la siguiente diapositiva.

Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }


std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);                    ──► eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar_impares(v2);                  ──► eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3);                ──► eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

Recibe un parámetro más y sirve saber las funciones que tiene dentro.

Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }


std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

```
[1, 5, 9]
```

```
[6, 10, 20]
```

```
[-2, -6]
```

pero ahora solamente tenemos una función eliminar.

Orden superior

- Cuando una función o método `f` recibe otras funciones como parámetros, o devuelve una función como resultado, decimos que `f` es una función o método de **orden superior**.
- La función `eliminar` es de orden superior.

Cualquier función que reciba como parámetro otra función es una función de orden superior.

Una pequeña generalización

- Podemos hacer que eliminar funcione sobre listas de cualquier tipo; no solo sobre listas de `int`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
  auto it = elems.begin();
  while (it ≠ elems.end()) {
    if (func(*it)) {
      it = elems.erase(it);
    } else {
      ++it;
    }
  }
}
```

Ahora podemos aplicarla a listas de objetos de cualquier tipo, clase fecha, clase persona, enteros...

Ejemplo

```
std::list<Fecha> v4 = { {25, 12, 2010}, {10, 21, 2020}, {25, 12, 1900}, {1, 1, 2000} };
eliminar(v4, es_navidad);
std::cout << v4 << std::endl;
```

```
[10/21/2020, 01/01/2000]
```

Para un tipo de dato fecha.