Tema 6: Aspectos avanzados de C++

Tecnología de la Programación de Videojuegos 1

Grado en Desarrollo de Videojuegos

Curso 2023-2024

Rubén Rubio Cuéllar

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Callbacks y eventos

Callbacks: funciones como argumento

La función que las recibe las llamará cuando lo necesite o corresponda.

- Parámetros de un algoritmo

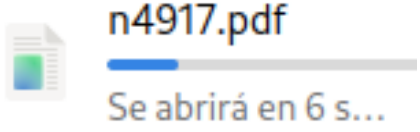
```
bool nombre(const Persona& izda,
            const Persona& dcha)
{ return izda.nombre < dcha.nombre; }
```

```
sort(lista.begin(),
     lista.end(),
     nombre);
```

- Botones y otros eventos en interfaces gráficas

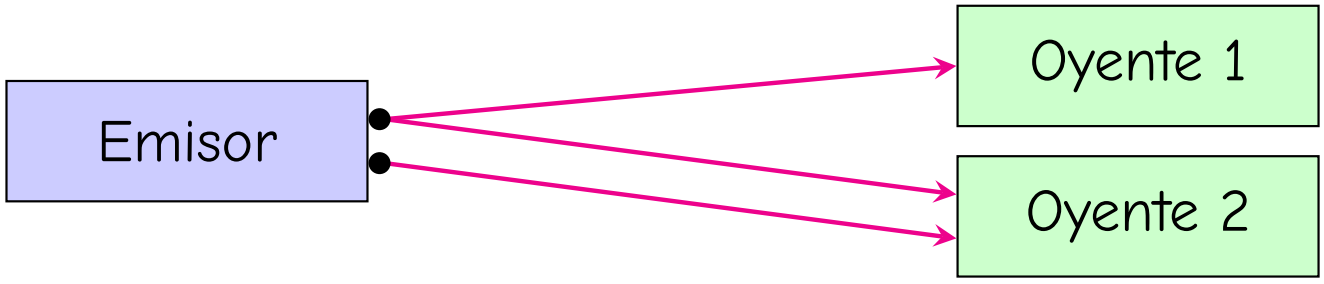

```
void clicked(Button* btn) { document.save(); }
```

- Operaciones asíncronas o prolongadas




```
download(connection, url, progress, finish);
```

Programación dirigida por eventos



- Ciertos componentes emiten eventos a los que otros se pueden suscribir o conectar (v.g. *clicked*, *loaded*, *mouseenter*, etc.)
- Utilizado habitualmente para programar interfaces de usuario
- Aparecen con nombres diversos en distintos contextos
  - `signal`/`raise` de la cabecera `csignal` (bajo nivel) en C/C++
  - **`event`** y **`delegate`** en C#
  - Disparadores (*triggers*) en bases de datos y SQL
  - `addEventListener` en Javascript y similares

Callbacks mediante métodos virtuales

Utilizando objetos oyentes de eventos (*event listener*) (habitual en Java...)

- Interfaz con los métodos que ha de implementar el usuario

```
class SDLEventListener // interfaz
{ public: virtual void handleEvent(const SDL_Event& e) = 0; };
```

- El usuario hereda de la interfaz e implementa sus métodos

```
class Cannon : public SceneObject, public SDLEventListener {
public:
   void handleEvent(const SDL_Event& e) override {
      if (event.type == SDL_KEYDOWN) { /* ... */ }
   }  // mejor en el .cpp                                  // [...]
};
```

Callbacks mediante métodos virtuales

- El llamante guarda referencias a los subscriptores

```
class Game {
   std::vector<SDLEventListener*> eventListeners;
public:
   void addEventListener(SDLEventListener* event);        // [...]
};
```

- Llama a sus métodos para difundir el evento

```
SDL_Event event;
while (SDL_PollEvent(&event)) {
   for (SDLEventListener* listener : eventListeners)
      listener->handleEvent(event);
}
```

Callbacks mediante métodos virtuales

Ejemplo sin eventos: un esqueleto para crear servidores web

```
class WebServer {
protected:
   virtual Response get(Request request) = 0;
   virtual Response post(Request request) = 0;                    // [...]
public:
   bool run(int port);                                            // [...]
};
```

```
class MyWebServer : public WebServer {
protected:
   Response get(Request request) override { /* ... */ };
   Response post(Request request) override { /* ... */ };         // [...]
};
```

Punteros a función

Las funciones son secuencias de código máquina cargadas en memoria, así que se pueden tomar punteros a ellas.

```
int suma(int a, int b) {
    return a + b;
}
```

`0x555555555120` `p = &suma`

```
0x555555555120 <suma(int, int)>:
    5120:   55                      push   %rbp
    5121:   48 89 e5                mov    %rsp,%rbp
    5124:   89 7d fc                mov    %edi,-0x4(%rbp)
    5127:   89 75 f8                mov    %esi,-0x8(%rbp)
    512a:   8b 45 fc                mov    -0x4(%rbp),%eax
    512d:   03 45 f8                add    -0x8(%rbp),%eax
    5130:   5d                      pop    %rbp
    5131:   c3                      ret
```

Punteros a función

- Declaración de variable o argumento de tipo (puntero a) función

```
int (*miOperacion)(int, int);
void filtra(vector<int>& v, bool (*pred)(int));
```

- Se pueden declarar alias de tipo para evitar la sintaxis compleja

```
using OpBinaria = int(*)(int, int);
OpBinaria miOperacion;
```

- Se llaman como si fueran una función concreta

```
miOperacion(2, 3);
(*miOperacion)(2, 3);  // es lo mismo
```

- Problema: ¿qué pasa si queremos pasar un puntero a método?

El tipo std::function

Envoltorio polimórfico de diversos objetos llamables, definido en la cabecera `functional`:

$$\texttt{std::function<}\texttt{tipo}_{res}\texttt{(}\texttt{tipo}_1\texttt{, ..., }\texttt{tipo}_n\texttt{)>}$$

- Punteros a función
- Punteros a método `void(Cannon::*p)(SDL_Event)`
- Funciones parcialmente evaluadas `std::bind_front(suma, 1)`
- Objetos llamables `operator()`
- Expresiones lambda `[](int a, int b) { return a + b; }`

El tipo std::function – ejemplo

El ejemplo anterior, utilizando callbacks funcionales. Por el emisor:

```
using SDLEventCallback = std::function<void(const SDL_Event&)>;

class Game {
    std::vector<SDLEventCallback> eventCallbacks;
public:
    void connect(SDLEventCallback cb);                    // [...]
};
```

```
SDL_Event event;
while (SDL_PollEvent(&event)) {
    for (const SDLEventCallback& cb : eventCallbacks)
        cb(event);
}
```

El tipo std::function – ejemplo

Por la parte del suscriptor:

```
class Cannon : public SceneObject {
public:
    void handleEvent(const SDL_Event& e);
};
```

```
Cannon::Cannon(Game* game, Point2D position) {
    // [...]
    game.connect([this](auto event) { handleEvent(event); });
}
void Cannon::handleEvent(const SDL_Event& event) {
    if (event.type == SDL_KEYDOWN) { /* ... */ }
};
```

Expresiones lambda

Son funciones anónimas definidas *in situ*, que además pueden capturar parte del contexto donde se definen (entre corchetes). El nombre hace referencia al λ-cálculo (Alonzo Church, 1936).

```
int var1 = 7;

auto func = [var1, &var2, this](int arg1, int arg2) {
    // Código de la función (puede utilizar la variable
    // copiada var1, la variable por referencia var2, y si
    // se ha puesto this y se ha definido dentro de un método,
    // los atributos y métodos de su clase).
    return var1 + arg1 + args2;
}

func(2, 3); // -> 12
```

Expresiones lambda – ejemplo

Ejemplo de la ordenación con expresiones lambda:

```
// Ejemplo con expresiones lambda
vector<Persona> listaNombre = lista;
vector<Persona*> listaEdad = aPunteros(lista);

sort(listaNombre.begin(), listaNombre.end(),
     [](const Persona& a, auto b) {
         return a.nombre < b.nombre;
});

sort(listaEdad.begin(), listaEdad.end(),
     [](Persona* a, Persona* b) -> bool {
         return a->fechaNacimiento < b->fechaNacimiento;
});
```

Recapitulación de callbacks

Se pueden implementar en C++ usando:

1. Métodos virtuales (objetos oyentes de eventos)
2. Objetos de tipo función (punteros a función o `std::function`)

Métodos como callback

- Usando la función `bind` o `bind_front` (C++20) con su **`this`**

```
std::bind(&Clase::método, this);  // y tal vez otros
```

- Usando una expresión lambda

```
[this](const SDL_Event& arg) { método(event); }
```

Punteros inteligentes

Punteros inteligentes

Recomendaciones sobre uso de punteros (de Bjarne Stroustrup)

1. Guarda el objeto como una variable o atributo sin puntero
2. Guarda múltiples objetos usando contenedores de la STL
3. Si no es posible (polimorfismo, etc.), usa punteros inteligentes
4. Si no eres el dueño del objeto, usa punteros normales

Punteros inteligentes de la STL

- Puntero único (`unique_ptr`)
  - Encapsula un puntero y libera la memoria al final de ámbito
- Puntero compartido (`shared_ptr`)
  - Lleva la cuenta del número de usuarios (copias del puntero)
  - Se liberará cuando no tenga ningún usuario
  - Como la *recolección de basura* de otros lenguajes, con un solo objeto

Punteros únicos

```
template<typename T>
class unique_ptr {
   T* ptr;
public:
   unique_ptr(T* ptr) : ptr(ptr) {}
   ~unique_ptr() { delete ptr; }              // libera ptr

   unique_ptr(const unique_ptr&) = delete;   // no se copia
   unique_ptr(unique_ptr&&) = default;       // se transfiere

   T& operator*() const { return *ptr; }     // *uptr
   T* operator->() const { return ptr; }     // uptr->fn()

   T& operator=(T* nuevo) { delete ptr; ptr = nuevo; }
};
```

Punteros compartidos

```
template<typename T>
class shared_ptr {
   struct Impl { T* ptr; long count; };
   Impl* impl;  // puntero y recuento de usuarios compartidos
public:
   shared_ptr(T* ptr) : impl(new Impl{ptr, 1}) {}
   ~shared_ptr() {
      if (--impl->count == 0) {   // si no hay usuarios
         delete impl->ptr;        // libera el objeto
         delete impl;             // libera el struct
      }
   }
   shared_ptr(const shared_ptr& orig) : impl(orig->impl) {
      orig->impl->count++;        // un usuario más
   }
```

Punteros compartidos

```
template<typename T>
class shared_ptr {
    struct Impl { T* ptr; long count; };
    Impl* impl;  // puntero y recuento de usuarios compartidos
public:
     // [sigue]

    shared_ptr(unique_ptr&&) = default;
    long use_count() const { return impl->count; }

    T& operator*() const { return *impl->ptr; }
    T* operator->() const { return impl->ptr; }
};
```

Biblioteca estándar y externas

Biblioteca estándar de C++

La biblioteca estándar de C++ incluye funciones útiles para diversas tareas, pero es más limitada que la de C#.

- Estructuras de datos (`vector`, `list`, `map`, `unordered_map`, etc.)
- Algoritmos (`sort`, `binary_search`, permutaciones, etc.)
- Operaciones matemáticas (`gcd`, `lcm`, `complex`, `valarray`, etc.)
- Números aleatorios (cabecera `random`)
- Expresiones regulares (cabecera `regex`)
- Concurrencia (`thread`, `promise`, `barrier`, `semaphore`, etc.) → TPV2
- Medición del tiempo y calendario (cabecera `chrono`)
- Manejo de archivos (cabecera `filesystem` y alguna función de C)

Biblioteca estándar de C++

Sin embargo, hay muchas funcionalidades habituales que no están en la biblioteca estándar y para las que hacen falta bibliotecas externas:

- Interfaces de usuario (Win32, Gtk, Qt, Cocoa, wxWidgets, ...)
- Sockets e internet (SDL_net, Win32, POSIX, Boost.ASIO, ...)
- Creación y comunicación entre procesos (Win32, POSIX, ...)
- Formatos de configuración o datos (XML, JSON, INI, ...)
- Manipulación de audio (SDL_mixer, portaudio, ...)
- Comunicación con bases de datos
- Etcétera...

Bibliotecas externas

Las bibliotecas externas se componen fundamentalmente de

- Una colección de archivos `.h`/`.hh` donde se declaran las funciones, clases, etc.
- El código compilado de la biblioteca
  - Uno o varios archivos `.lib` si la biblioteca es estática (va con el ejecutable)
  - Uno o varios archivos `.lib` y `.dll` si es dinámica (los `.dll` hay que distribuirlos con el programa)

En Visual Studio, hay que hacer todos esos archivos visibles al compilador cambiando la configuración del proyecto.