Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #7

HID and Error Handling

Lecture #7

Part I: HID

Introduction

- Games are interactive computer simulation
  - Management of user inputs is central
- Human interface devices (HID) for games
  - keyboard
  - mouse
  - joystick
  - joypad
  - track ball
  - multi-touch pad
  - remote controller
  - webcam
  - steering wheel, pedal, force plate, electric guitar
  - *and much more*

Introduction

- HID provides input to the game software
- Some HIDs allow to give feedback to the user
  - light, force feedback, vibration, sound
- Game engine reads and writes HID inputs and outputs
  - depends on the specific design, OS, and device

Interfacing with HID

- Two techniques
  - Polling
  - Interruption

Interfacing with HID

- Polling
  - Device state is checked by polling the hardware periodically
    - usually once per game loop iteration (or defined by the input manager update frequency)
    - explicit call to the reading of device state
    - by reading hardware registers, memory I/O port or higher software interface (driver)
  - Example: Microsoft's XInput API for Xbox 360 game pad for console and PC
    - call to XInputGetState() function at each update
    - returns a XINPUT_STATE containing joy pad information (buttons pushed, stick position *etc.*)

Interfacing with HID

- Interruption
  - Update game logic only when changes occur
  - No need to send a continuous stream of data when device is not pressed / released / moved
  - Communication with the game engine done via hardware interruption
    - electronic signal that suspends the game execution and calls an interrupt service routine (ISR)
    - ISR reads the HID state, updates its state in game engine and resumes the execution
    - game engine takes action(s) immediately or picks up the new state next time it is convenient to do so

Interfacing with HID

- Keyboard and mouse are the main devices for PC-based games
- Interfacing concepts for keyboard and mouse can be generalized to any HID

The keyboard

- Not particularly well suited for game control
  – set of buttons without direct spatial relationship with the virtual world
- On PC platform Windows
  – defined in header <winuser.h>
    - but include <windows.h>
  – [add library user32.dll in path]

Polling the keyboard

- To get the state of a specific key

```
short GetAsyncKeyState(int keycode);
```

  - if the key is a letter or digit (a => z, 0 => 9), keycode can use its ASCII value (0x41 => 0x5A, 0x30 => 0x39)
  - otherwise keycode is a virtual-key value (one out of the 256 entries, defined in winuser.h)
    - *e.g.* VK_BACK for backspace key, VK_TAB for tabulation, VK_RETURN, VK_SHIFT, VK_LEFT, ...

Polling the keyboard

- To get the state of a specific key

```
short GetAsyncKeyState(int keycode);
```

  - return value encodes the state of the key
    - Most significant bit set if key down
    - Least significant bit set if key pressed after previous call to GetAsyncKeyState
- Array of 256 bool usually used locally to maintain keyboard state

Polling the keyboard

- To get the state of a specific key

```
short GetKeyState(int keycode);
```

  - same input as GetAsyncKeyState
  - return value encodes the state of the key
    - Most significant bit set if key down, otherwise up
  - reports the state of the keyboard at the time of the generation of the keyboard-input message
    - GetKeyState always used in response to a message

Polling the keyboard

- GetAsyncKeyState vs. GetKeyState
  - At time t, the user Alt+LeftClick mouse
  - At time t+dt, the program responds to the click and checks the board state
  - Assuming the user released the Alt key in between
    - using GetAsyncKeyState will return that Alt is not down => at this very calling instant t+dt
    - using GetKeyState will return that Alt is down => at the time t the user clicked the mouse (event created)

Polling the keyboard

- Then each key requires a specific call
  - time consuming when testing a lot of keys
- The whole keyboard state can be queried

```
bool GetKeyboardState(PBYTE lpKeyState);
```

  - lpKeyState is a 256-byte array that receives the status for each virtual key
  - return value is true if call succeed
  - same behavior as GetKeyState on each key

The keyboard interruption

- To call a function only when key pressed
- On Windows PC platform, keyboard and mouse tracking can be done through the Windows Procedure of Win32 API
  - Notification/Message mechanism
  - Available on window-based applications via the WndProc function

The keyboard interruption

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
    {
    // ...
    switch (msg) {
          case WM_COMMAND:
                     // ...
                    break;
          case WM_DESTROY:
                    PostQuitMessage(0);
                    break;
          // ... other cases ...
          default:
                    return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

The keyboard interruption

- Messages are send when key state changes
- Examples:
  - WM_KEYDOWN
  - WM_KEYUP
  - *and more (see msdn.microsoft.com)*
- The wParam then contains the virtual-key code

The keyboard interruption

- Example

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_KEYDOWN: // do something when key pressed
            switch (wParam) {
                case VK_LEFT: // process left arrow
                case VK_RIGHT: // process right arrow
                case VK_F2: // process F2 key
                case 0x41: // process A key
                default: break;
            }
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

The keyboard interruption

- Polling is often used during a keyboard interruption to combine key/mouse events
- Examples
  - Shift+LeftArrow, in a game to strafe left
  - Ctrl+Alt+Delete, task manager
  - Alt+F4, to exit
  - Alt+Enter, to full screen
  - *etc.*

Polling the mouse

- To read mouse state (position)

```
bool GetCursorPos(LPPOINT point);
```

- LPPOINT is a pointer to a POINT structure
  - including two long int: x and y
- return nonzero if successful, zero otherwise
- Cursor position is specified in screen pixel coordinates
  - x (resp. y) from 0 to hor. (resp. vert.) max resolution
  - Screen coordinates can be converted to/from window coordinates by ScreenToClient / ClientToScreen functions

Polling the mouse

- Example

```
#include <windows.h> // -> includes <winuser.h>

POINT cursorPos;
GetCursorPos(&cursorPos);

cout << "Cursor position: ";
cout << "( " << cursorPos.x << " " << cursorPos.y << " )" ;
```

The mouse interruption

- To call a function when mouse is moved or button pressed (also considered as virtual keys)
- Same mechanism as keyboard: through Windows Procedure messages
  - WM_LBUTTONDBLCLK
  - WM_LBUTTONDOWN
  - WM_MOUSEHWHEEL
  - WM_MOUSEMOVE
  - *and more (see msdn.microsoft.com)*

The mouse interruption

- Each mouse related message has its own wParam and lParam contents
  - WM_MOUSEMOVE (and others)
    - horizontal and vertical position in lParam
    - buttons states in wParam
  - WM_MOUSEWHEEL
    - same plus the wheel-delta value in wParam

The mouse interruption

- Example

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_MOUSEMOVE: // do something when moving the mouse
                int xPos = GET_X_LPARAM(lParam);
                int yPos = GET_Y_LPARAM(lParam);
                convertToGameWorldLocation(xPos,yPos);
                if (wParam & MK_RBUTTON) // wParam contains buttons states
                        AttackAt(xPos,yPos); // attack while right clicking
                else if (wParam & MK_LBUTTON)
                        MoveTo(xPos,yPos); // move while left clicking
                else
                        LookAt(xPos,yPos); // look at otherwise
                break;
        default:
                return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

Input manager in game engine

- Usually the input manager is in charge of
  - calling the entities that are willing to take actions regarding input events
  - providing polling functions to them
- These entities register themselves to the game engine
- The Windows notifications from WndProc are forwarded to the input manager
  - which selects the user input related messages
  - and notifies the entities

Hardware abstraction

- More and more HIDs are available
- How to enable the use of any HID to control a game without any impact on the engine?
- Hardware abstraction specifies a virtual controller
  - any HID that conforms to the abstract profile of the controller can then be used
  - write a pure abstract class for generic controller handler
  - at run time only the selected HID controller is created

HID system functionalities

- Usually game engine HID system provides
  - data zones validity and filtering
    - due to an analog noisy signal, the input may have to be rounded in order to stay in the min/max limits and to have a steady rest configuration
    - due to signal noise ratio, the input is filtered (smoothed) using a low pass filter
  - event detection
    - interruption routines compatible with OS

HID system functionalities

- Usually game engine HID system provides
  - detection of chords and sequences
    - when a specific group of inputs is fired or when a sequence of inputs is realized, the system can trigger a special action
    - examples
      - CTRL-ALT-DELETE in Windows to start Task Manager
      - button sequence A-B-B-A-A-B-R-L-A in Street Fighter Turbo Speed 2 to activate a super hyper mega kick
  - management of multiple HIDs for multiple players
    - to route devices to the right player in game
    - involved a bidirectional player to controller mapping
    - needs also to take care of HID disconnection (unplugged, out of battery etc.) in the gameplay

HID system functionalities

- Usually game engine HID system provides
  - multiplatform HID support
    - by conditional compilation wherever platform specific functions are used
    - or by adding an abstraction layer
  - controller input re-mapping
    - action mapping table is used to translate raw inputs into logical game actions
    - to enable the re-assignment of the controller’s functions
    - examples: up/down direction in mouse and joystick (for flight games), OPQA vs. arrow pad vs. WASD

HID system functionalities

- Usually game engine HID system provides
  - context-sensitive inputs
    - when the same input triggers different actions according to the context
    - can be implemented with simple state machine to a get priority focus
    - examples
      - the ‘E’ use button in adventure games where it means talk to if NPC selected, and pick up if object in sight, and open if door in front etc.
      - HID to control character or vehicle or camera or 2D menu navigation
  - the ability to temporarily disable inputs
    - using disable mask on inputs or interpreting in the game logic
    - examples
      - disabling user inputs during in-game cinematic
      - disable camera moving in constrained environment

Lecture #7

Part II: Error Handling

Dealing with errors

- An error condition (or just "error") is a condition occurring during runtime, that is not executed by the normal flow
  - alternative way to recover safely from an error
  - not the same as a bug
- Error conditions in a function might be
  - Prevented (ensure always valid calling)
  - Handled in the function
  - Left to the user of the function to deal with it

Error handling

- Different approaches
  - to terminate the program
  - to return error codes or error indicators
  - to call an error handler function
  - to throw exceptions

Assertions

- An assertion checks an expression
  - if true nothing happens
  - if false a message is printed and the program is stopped
- Used as “land-mine”
  - as soon as a modification of the code violates the assertion, an error will be shown
  - usually only during development process
    - often used to check pointer validity (!= NULL)

Assertions

- Implemented with #define macro

```
#if ASSERTIONS_ENABLED
 #define ASSERT(expression) \
   if (expression) { } \
   else reportAssertionFailure(#expression,__FILE__,__LINE__); \
#else
 #define ASSERT(expression)
#endif
```

Assertions

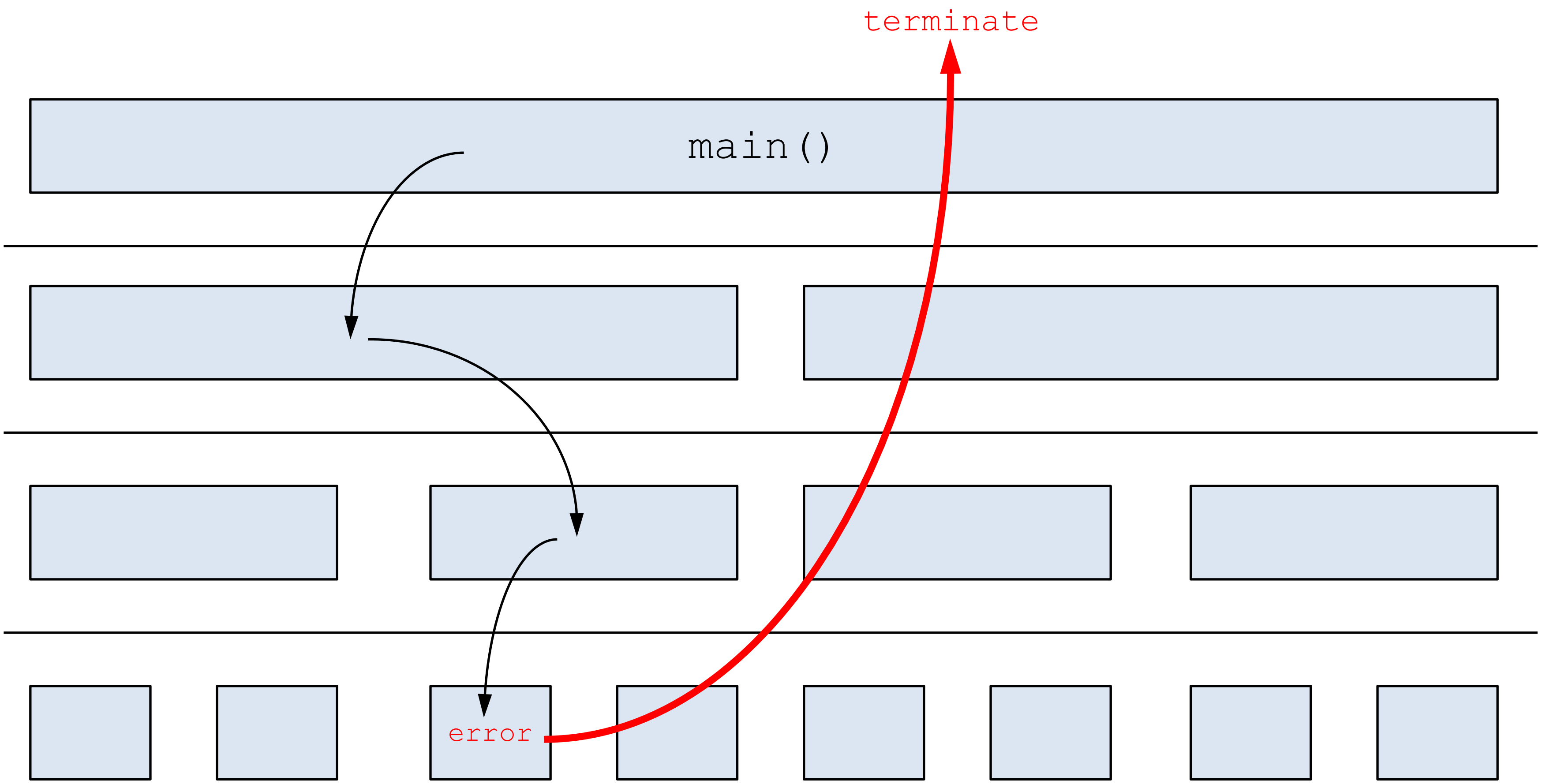
- Default C/C++ library
  - expression is written, then abort is called, terminating the program
  - asserts are ignored if NDEBUG is defined
    - designed to capture programming errors not user or running errors

```
#include <assert.h>

int * ptrValue;
// ...
assert(ptrValue != NULL);
```

```
Assertion failed: ptrValue != NULL, file main.cpp, line 5
```

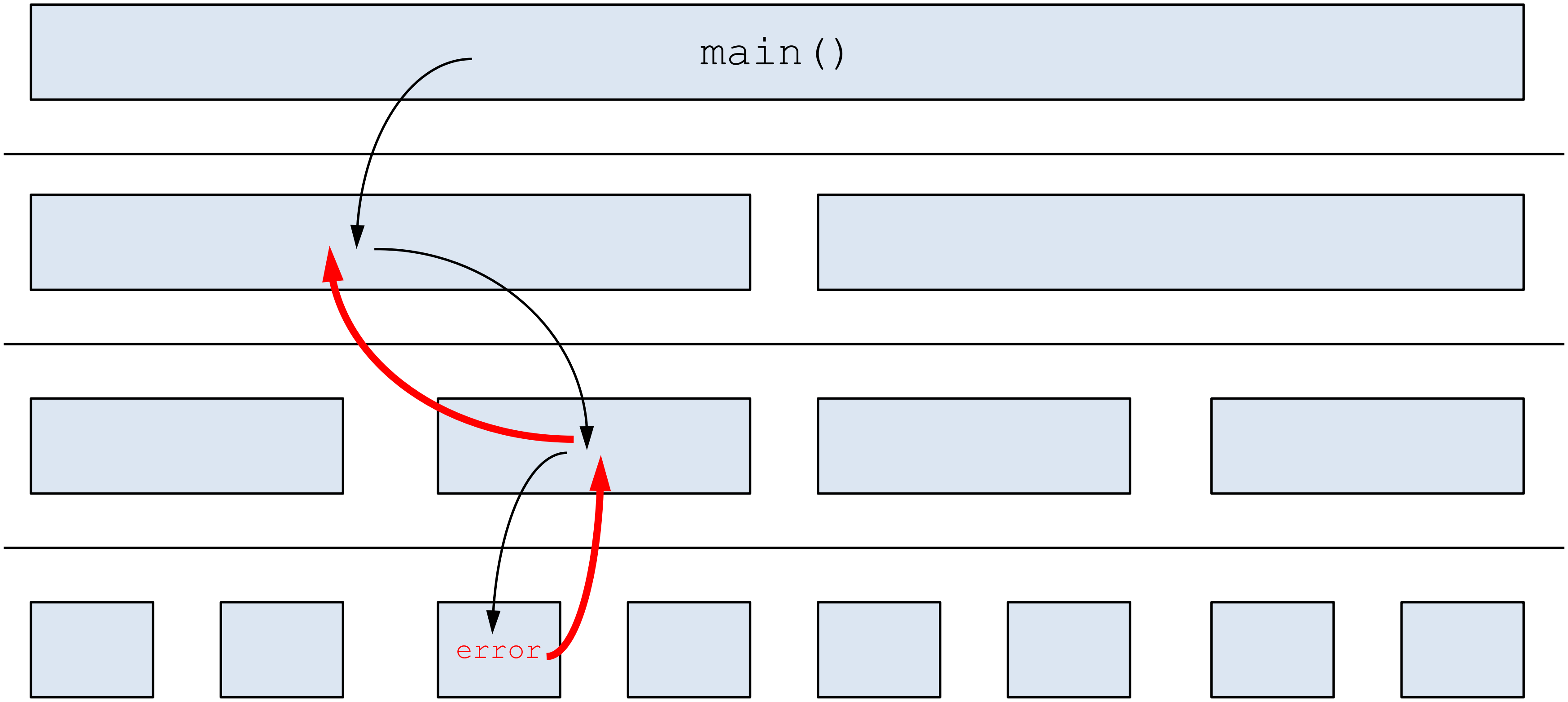
Assertions



Error codes and indicators

- Returning fail/pass code from the function in which the problem is first detected
  - boolean value
  - legal but “impossible” value of returned type or out of range (ex: NULL, -1, “”)
  - code (ex: 0 = ok, 1 = error 1, 2 = error 2 ...), usually in an enum
- Error indicator as reference parameter (usually last parameter, also called flag)
- Calling function intercept and interpret the error
  - solved directly or passed to the calling function

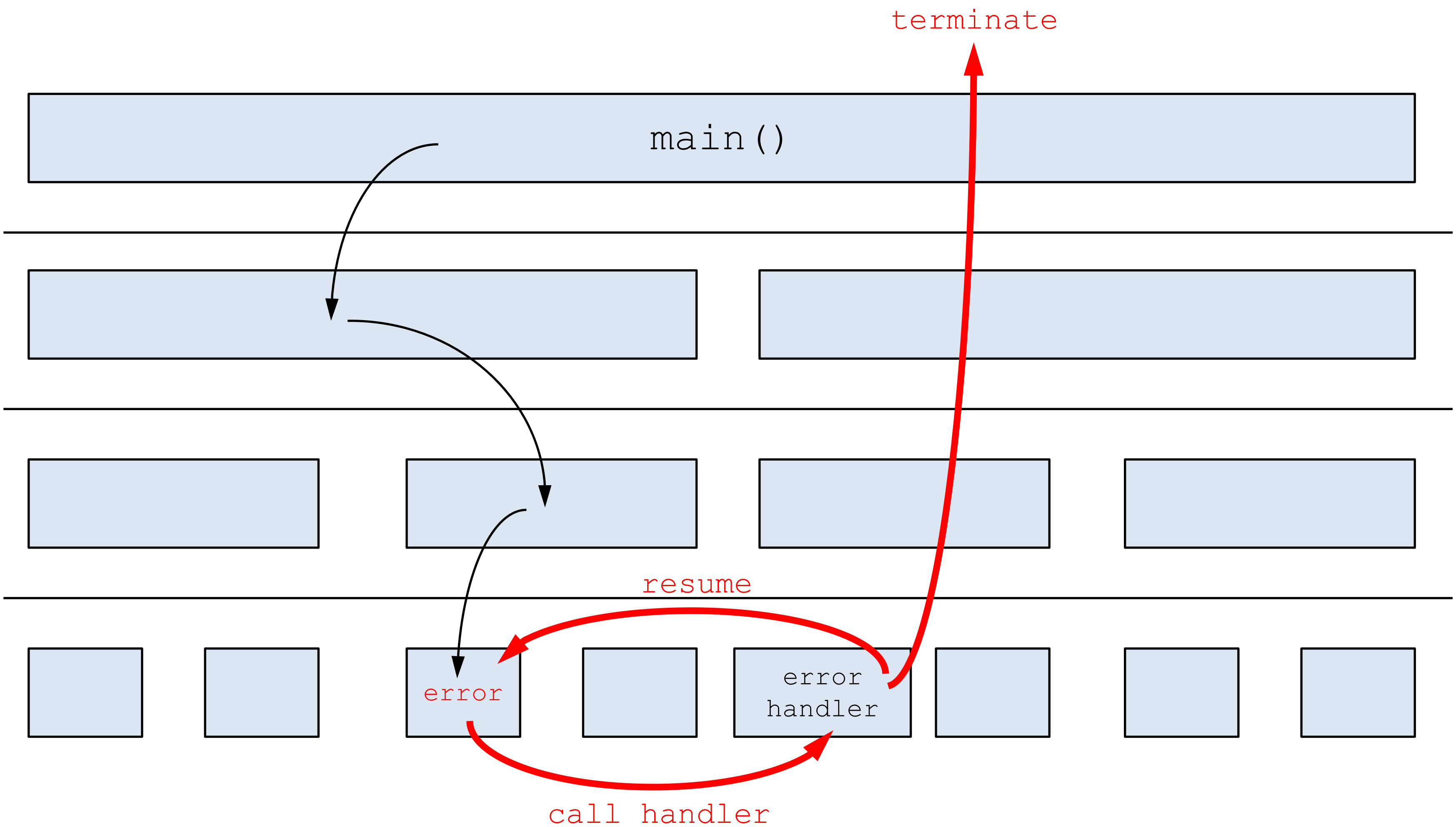
Error codes and indicators



Error handler function

- A function that is especially designed to deal with errors
  - chooses to stop program or resumes execution
- Might need access to many information to make the decisions
  - central organ of the code

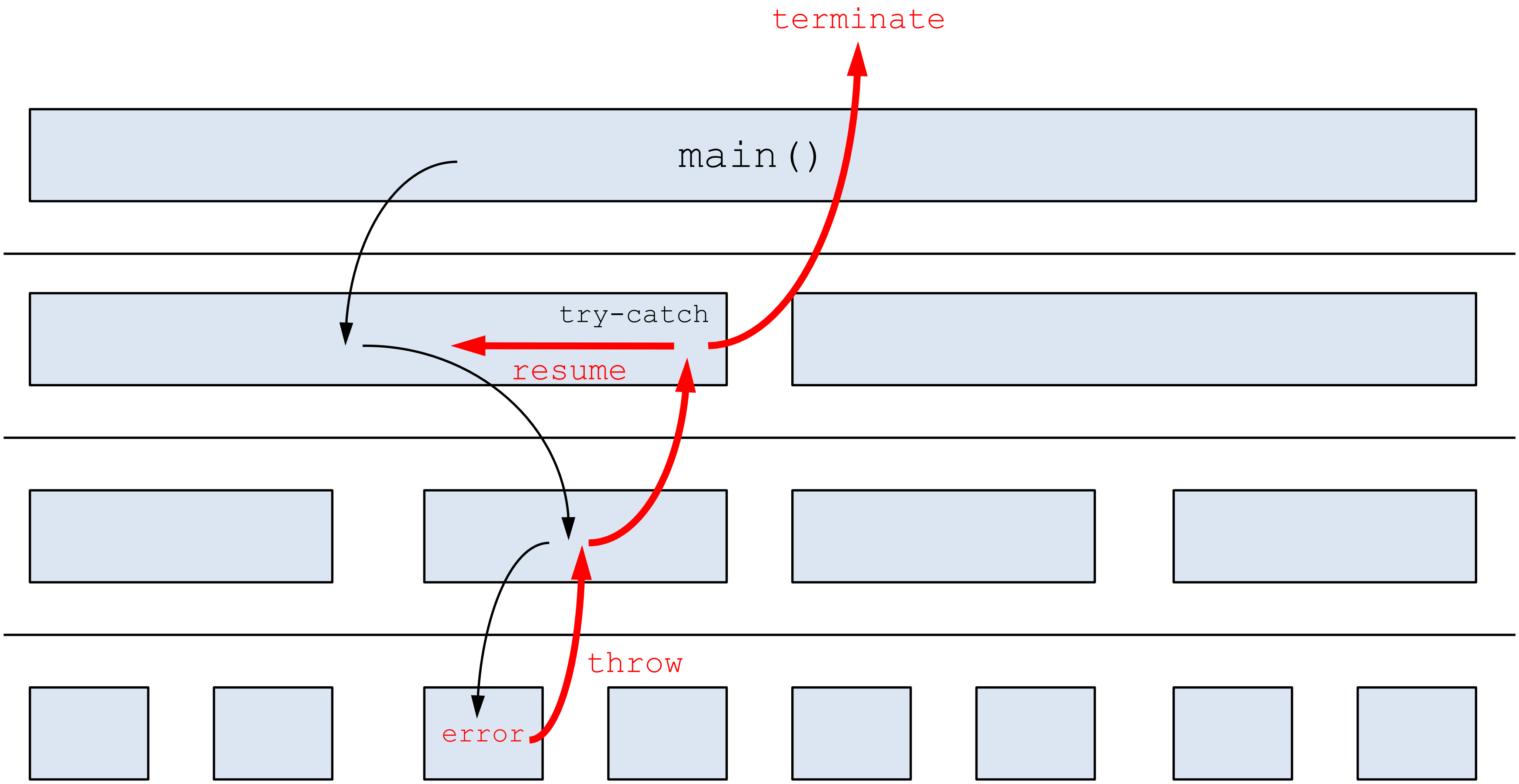
Error handler function



Exceptions

- Throwing exception allows a function to communicate an error to the rest of the code without information on the handling function
  - the rest of the code in the throwing function is not evaluated
- Creation of an exception object containing the information about the error
- The function that explicitly catches that exception deals with the error
  - try-catch block

Exceptions



Exceptions

- Advantages
  - Less messy code, easier to maintain
  - Flexibility (different handling for different errors)
  - Better error information handling
  - Error reporting in constructors (error codes are not possible)
  - No large error code table required, easier debugging
  - Exceptions are a uniform way of indicating errors in C++
- But very costly
  - memory for the unwinding process information
  - time (2-3x) to unwind the stack
  - implementation of try-catch 'everywhere'

Exception handling in C++

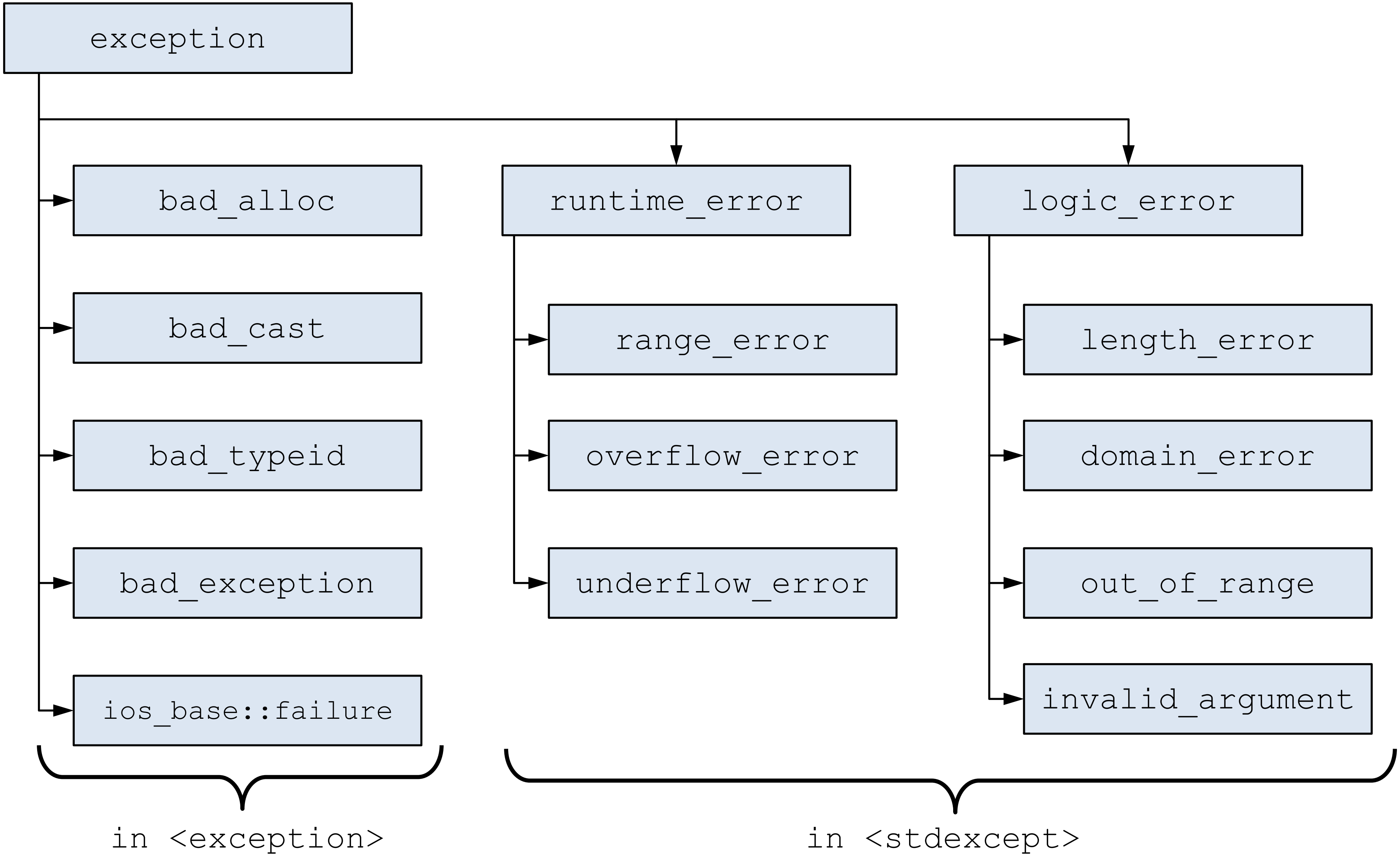
- Syntax for catching an error

```
try {
   // code here that could throw one or more exceptions
} catch (exceptions_parameter) {
   // deal with the error(s) here
}
```

- Syntax for throwing an error

```
throw errorException ;
```

Standard exception hierarchy



Exception handling in C++

- Catching example from constructor

```
try {
   Player* player = new Player();
} catch (std::bad_alloc& e) {
    // memory allocation didn’t succeed!
} catch (std::out_of_range& e) {
    // some array was accessed out of range!
} catch (std::runtime_error& e) {
    // range_error, overflow_error or underflow_error detected!
    // use dynamic_cast to determine exact error type
} catch (...) {
    // some other indeterminate exceptions occurred!
}
```

Exception handling in C++

- Throwing examples

```
Item* Inventory::getItem(int i) {
   if (i < 0 || i > amount_)
        throw std::range_error("Inventory:index out of range");
        // will be catched with range_error exception
   return items_[i];
}
```

```
Item* Inventory::getItem(int i) {
   if (i < 0 || i > amount_)
        throw -1;
        // will be catched with “catch (int e)”
   return items_[i];
}
```

Exception handling in C++

- Re-throwing an exception to the calling function

```
try {
    executeAExceptionFunction();
} catch (std::runtime_error& e) {
    doTheMostToSolve();
    throw;
}
```

- This will throw the same runtime error exception again
  - Useful for resolving local problems and passing the exception on to caller for further necessary actions

Exception specifications

- We can limit the exception type of a function
  - directly or indirectly thrown
  - by appending a throw suffix to the declaration

```
float FunctionOne (char parameter) throw (int);
int FunctionTwo (int parameter) throw (std::out_of_range);
```

- Permission to throw exceptions can also be specified with the throw specification

```
int F1 (int param) throw ();  // no exception allowed
int F2 (int param);           // all exceptions allowed
```

Exception handling in C++

- The exception is directly thrown to the calling function if no try-catch
- ⚠ But exceptions cannot be ignored!

```
void TestReady (Player& p) throw (std::runtime_error) {
   if (!PlayerReady(p))
        throw std::runtime_error("Player should be ready!");
}

void Run() throw (std::runtime_error) {//enable to throw directly
   Player p;
   TestReady(p); // does not required try-catch, throws to caller
   SpeakWithPlayer(p);
}
```

Exception handling in C++

- What if an exception is never caught?
  - Unwinding until the main function
  - If still no catch, call to std::terminate()
  - Possibility to change the terminate function

```
void myTerminate() {
   std::cout << "Unexpected exception not caught.\n";
}

int main()
{
   std::set_terminate(myTerminate);
   throw "error";
   return 0;
}
```

Custom exception class

- You can create your own exception class

```
class MyException {...}
class MyException : public std::exception {...}
```

- Most standard exception classes have a string member to use as a message
  - As parameter to the constructor

```
throw MyException("That's not acceptable!");
```

  - Accessible through the what() member function

```
try {...} catch (MyException& e) {cout << e.what();}
```

What can throw?

- The following can throw in C++
  - “throw” throws
  - “new” may throw std::bad_alloc if it cannot allocate the requested memory
  - A function that
    1. calls a function that throws
    2. does not catch an exception
  - Functions written by others may throw
    - See their doc’s.

What does not throw?

- The following cannot throw
  - Default operations on primitive types (including operator[])
  - The default version of “delete”
  - C++ Standard I/O libraries (by default)

Guidelines

- Identify all statements where an exception can appear
  - solve it or throw it up if one caller can solve it
- Identify all problems that can occur in presence of an exception
  - write handler to be able to
    - resume the program
    - re-do operation differently
    - allow a caller to solve the problem
    - terminate the program in last case
  - indicate in header that an exception might be thrown

Guidelines

- Ideally, leave your object in the state it was when the function was entered
  - catch exceptions and restore the initial state
- Do not catch exceptions if you do not know how to (partially) handle them
- If you cannot ignore propagated exceptions, use a catch-all (...) clause
- Do not throw strings as all exceptions will have the same type string
- Keep your objects destructible
  - do not leave dangling pointer in your objects

End of lecture #7

Next lecture

Template and Serialization