WELCOME TO TUTORIAL 6

EXCEPTIONS, RAII & SMART POINTERS
TODAY

- Exceptions
  - Throwing Exceptions
  - Catching Exceptions
  - C++ Standard Exceptions
  - Define New Exceptions
- RAII
  - Three levels of Guarantee
  - What’s RAII
- Smart Pointers
  - Basic Information
  - How to use them
EXCEPTIONS

• Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

• **throw** – A program throws an exception when a problem shows up. This is done using a **throw** keyword.

• **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

• **try** – A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
EXCEPTION HANDLING

• Instead of using C-style strategies of handling errors, we use exceptions in C++ to control the behavior of the program when errors arise.

• If an exception is raised/thrown, the program will terminate if there is no handler (that is, a catch block) for it.

• Recall that we catch exceptions with try and catch blocks:

```cpp
try { throw 42; }

catch (...) {
    // "..." will accept any type of error
    cerr << "Caught something" << endl;
}
```

• The try block will run as normal, but any errors will be handled by the catch block underneath.
EXCEPTION HANDLING

• Note: if the catch(...) is used, it must be the very last catch clause.
• The try and catch blocks must be used together.
• We can throw anything, including exception objects, strings, integers, etc.
• We can also throw what we just caught in the catch block:
  • try { throw 42; }
  • catch (...) { throw;
    // will throw the same exception that was caught }
EXCEPTION AND INHERITANCE

• Consider the code “tryCatch.cc”

• What will be the output?
  • Since BadNumber inherits from BadInput, it will be caught by the first error handler.

• Good practice:
  – Throw by value, and catch by reference.
    – catch blocks needs to be ordered from the most specific to the least specific.

• For example, if there is an inheritance relationship between the exception classes, you should always catch the subclass exceptions first.

• catch(...) must always be the last catch statement.
EXCEPTIONS

• Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows —

```java
try { // protected code }
catch( ExceptionName e1 ) { // catch block }
catch( ExceptionName e2 ) { // catch block }
catch( ExceptionName eN ) { // catch block }
```
THROWING EXCEPTIONS

• Exceptions can be thrown anywhere within a code block using `throw` statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

• Following is an example of throwing an exception when dividing by zero condition occurs —

```java
double division(int a, int b) {
    if (b == 0) {
        throw "Division by zero condition!";
    }
    return (a/b);
}
```
CATCHING EXCEPTIONS

• The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try { // protected code }
catch( ExceptionName e ) { // code to handle ExceptionName exception }
```

Above code will catch an exception of **ExceptionName** type.

• If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows —

```
try { // protected code }
catch(...) { // code to handle any exception }
```
The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b) {
    if (b == 0) {
        throw "Division by zero condition!";
    } return (a/b);
}

int main () {
    int x = 50;
    int y = 0;
    double z = 0;
    try {
        z = division(x, y);
        cout << z << endl;
    } catch (const char* msg) {
        cerr << msg << endl;
    }
    return 0;
}
```
C++ provides a list of standard exceptions defined in `<exception>` which we can use in our programs. These are arranged in a parent-child class hierarchy shown below —
STD::EXCEPTION

- **std::exception** - An exception and parent class of all the standard C++ exceptions.
- **std::bad_alloc** - This can be thrown by `new`.
- **std::bad_cast** - This can be thrown by `dynamic_cast`.
- **std::bad_exception** - This is useful device to handle unexpected exceptions in a C++ program.
- **std::bad_typeid** - This can be thrown by `typeid`.
- **std::logic_error** - An exception that theoretically can be detected by reading the code.
- **std::domain_error** - This is an exception thrown when a mathematically invalid domain is used.
STD::EXCEPTION

- `std::invalid_argument` - This is thrown due to invalid arguments.
- `std::length_error` - This is thrown when a too big `std::string` is created.
- `std::out_of_range` - This can be thrown by the 'at' method, for example a `std::vector` and `std::bitset<>::operator[]()`.
- `std::runtime_error` - An exception that theoretically cannot be detected by reading the code.
- `std::overflow_error` - This is thrown if a mathematical overflow occurs.
- `std::range_error` - This is occurred when you try to store a value which is out of range.
- `std::underflow_error` - This is thrown if a mathematical underflow occurs.
DEFINE NEW EXCEPTIONS

You can define your own exceptions by inheriting and overriding `exception` class functionality. Following is the example, which shows how you can use `std::exception` class to implement your own exception in standard way —

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

struct MyException : public exception {
    const char * what () const throw () {
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        // Other errors
    }
}
```
THREE LEVELS OF GUARANTEE

- **Basic guarantee**: if an exception is thrown, data will be in a valid but unspecified state.
- **Strong guarantee**: if an exception is thrown, no side-effects will have occurred, your program's state is as if the function was never called.
- **No-throw guarantee**: an exception is never thrown.
RAII

- **RAII**: Resource Acquisition in Initialization
- RAII is vital to writing exception-safe code in C++.
- **Mechanism**: When acquiring a resource you should also be initializing a statically allocated object with it.
- When the objects lifetime is over its destructor should free the resource.
- e.g. files (fstreams), memory (smart pointers), and more.
SMART POINTERS

- unique_ptr is for sole ownership, only one unique_ptr can point to the same block of heap memory.
- shared_ptr is for shared ownership, allows many pointers that all point to the same block of heap memory and only deletes that memory when no other shared_ptrs point to it.
- weak_ptr is for non-owning referencing, created with an existing shared_ptr, used mainly to break cyclical references between shared_ptrs.
- Example: smartptr.cc
SMART POINTERS

- In most cases use smart pointers as you would raw pointers.

```cpp
class MyClass {
    unique_ptr<Impl> pImpl;
    ...
};
```
SHARED POINTERS

- `shared_ptr` works by maintaining a reference count to an object.
- For each `shared_ptr` you create to the same object the reference count is incremented for that piece of data.
- When the destructor of a `shared_ptr` executes it decrements the reference count, if at that point the reference count is 0 then it frees the memory.
- This only works if the `shared_ptr` class is used properly. Let’s consider how we might implement `shared_ptr` for ints.
WEAK POINTER

- `weak_ptr` does not work by maintaining a reference count to an object.
- For each `weak_ptr` you create to the same object the reference count is not incremented for that piece of data.
- When the destructor of a `weak_ptr` executes it does not change the reference count.