Slides

Development > Programming Languages > C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

4.7 ★★★★☆

Created by Daniel Gakwaya

Section : Exceptions

Slide intentionally left empty

Exception handling : Introduction

A C++ built in mechanism to bring problems to the surface and possibly handle them

- static_assert
- assert
- std::cout

```
try{

        throw 0;

}catch(int ex){

}
```

Slide intentionally left empty

try and catch blocks

Syntax

```
int a{10};
int b{0};

try{
    Item item; // When exception is thrown, control immediately exits the try block
               // an automatic local variables are released
               // But pointers may leak memory.
    if( b == 0 )
        throw 0;
    a++; // Just using a and b in here, could use them to do anything.
    b++;
    std::cout << "Code that executes when things are fine" << std::endl;

}catch(int ex){
    std::cout << "Something went wrong. Exception thrown : " << ex <<  std::endl;
}

std::cout << "END." << std::endl;
```

Throwing pointers to locals : BAD!

```
int c{0};
try{
    int var{55};
    int* int_ptr = &var;
    if(c == 0)
        throw int_ptr;
    std::cout << "Keeping doing some other things..." << std::endl;
}catch(int* ex){
    std::cout << "Something went wrong. Exception thrown : " <<*ex << std::endl;
}
std::cout << "END." << std::endl;
```

Pointers in try block : LEAKS!

```
int d{0};
try{
    Item * item_ptr = new Item();
    if(d == 0)
        throw 0;
    std::cout << "Keeping doing some other things..." << std::endl;
}catch(int ex){
    std::cout << "Something went wrong. Exception thrown : "<< ex << std::endl;
}
std::cout << "END." << std::endl;
```

Unhandled exceptions : CRASH!

```
//If you throw an exception and it's not handled anywhere in your code,
//you'll get a hard crash

throw 0;
std::cout << "Doing something after we throw" << std::endl;

std::cout << "END." << std::endl;
```

Thrown objects must be copyable

```
// If copy constructor is either deleted, protected or private, the
// object can't be thrown as exception. You'll get a compiler error.

try{
    MyException e;
    throw e; //

}catch(MyException ex){

}
std::cout << "END." << std::endl;
```

Slide intentionally left empty

The need for exceptions

Integer division by 0

```
//int division by 0 : CRASH

const int a{45};
const int b{0};
int result = a/b;
```

Failing downcast

```
//Downcast using dynamic_cast with references
//The hierarchy of Animal->Dog has to be polymorphic to be able to do
//this. This throws an exception because animal has no dog part so the
//cast won't really work.

Animal a;
Dog& d{dynamic_cast<Dog&>(a)};
```

Recovering from integer division by zero

```
//Integer division
const int a{45};
const int b{0};
int result;

try{
    if(b == 0)
        throw 0;
   result = a/b;
}
catch(int ex){
    std::cout << "Integer division by zero detected" << std::endl;
}
std::cout<< "END." << std::endl;
```

Thrown type must the same as the catch parameter

```
//Integer division
const int a{45};
const int b{0};
int result;

try{
    if(b == 0)
        throw 0;
   result = a/b;
}
catch(std::string ex){
    std::cout << "Integer division by zero detected" << std::endl;
}

std::cout<< "END." << std::endl;
```

Thrown type must the same as the catch parameter

```
//Integer division
const int a{45};
const int b{0};
int result;

try{
    if(b == 0)
        throw 0;
   result = a/b;
}
catch(std::string ex){
    std::cout << "Integer division by zero detected" << std::endl;
}
catch(int ex){
    //Some processing
}

std::cout<< "END." << std::endl;
```

Exceptions thrown in functions

```
/*  Function throws const char* exception when par_b is zero  */
void process_parameters( int par_a , int par_b){

    // Do some processing

    if(par_b == 0)
        throw "Potential division by 0 detected";

    int result = par_a / par_b;

    //Some other processing.
}
```

Handling exceptions thrown in functions

```
//Exceptions thrown out of other parts of code written by you
//or somebody else
try{
    process_parameters(10,0);
}catch(const char* ex){
    std::cout <<"Exception: " <<  ex << std::endl;
}
```

Slide intentionally left empty

Handling Exceptions at different levels

Chained calls

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
Ending f3()
Ending f2()
Ending f1()
END.
```

Execution Thrower

```
void exception_thrower(){
    std::cout << "starting execution_thrower()" << std::endl;
    throw 0;// Execution will halt from here
    std::cout << "ending execution_thrower()" << std::endl;
}
```

Exception thrown in f3(). Not handled

```
//Exception thrown from f3() scope : Not handled anywhere
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    exception_thrower();
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
terminate called after throwing an instance of 'int'
```

Exception thrown in f3(). Handled in f3()

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    try{
        exception_thrower();
    }catch(int ex){
        std::cout << "Handling execution in f3()" << std::endl;
    }
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
Handling execution in f3()
Ending f3()
Ending f2()
Ending f1()
END.
```

Exception thrown in f3(). Handled in f2()

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    try{
        f3();
    }catch(int ex){
        std::cout << "Handling execution in f2()" << std::endl;
    }
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    exception_thrower();
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
Handling execution in f2()
Ending f2()
Ending f1()
END.
```

Exception thrown in f3(). Handled in f1()

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    try{
        f2();
    }catch(int ex){
        std::cout << "Handling execution in f1()" << std::endl;
    }
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    exception_thrower();
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
Handling execution in f1()
Ending f1()
END.
```

Exception thrown in f3(). Handled main()

```
//Exception thrown from f3() scope : Not handled anywhere
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    exception_thrower();
    std::cout << "Ending f3()" << std::endl;
}
```

Handling the exception in main()

```
int main(int argc, char **argv)
{
    try{
        f1();
    }catch(int ex){
        std::cout << "Handling execution in main()" << std::endl;
    }

    std::cout << "END." << std::endl;
    return 0;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
Handling execution in main()
END.
```

Facts

- You can choose to handle the exceptions at a level in your function call chain that makes sense for your application
- In the extreme case, you can handle all your exceptions in main by just wrapping all the code in a try block and following it with an appropriate series of catch blocks
- The further you choose to handle your exception from where it was thrown, the more you risk for functions to not terminate normally
- Functions that don't terminate normally are bad : they may leak memory or just not completely do what they were designed to do.

Guideline

- If you know (from the documentation or code) that the function you're about to call may throw an exception, you should wrap that call in a try block and handle the exception appropriately, that way your function doesn't run the risk of non terminating properly when an the exception is thrown.

Slide intentionally left empty

Multiple handlers for an exception

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    try{
        f2();
    }catch(int ex){
        std::cout << "Exception handled in f1()" << std::endl;
    }
    std::cout << "Ending f1()" << std::endl;
}
```

```
void f2(){
    std::cout << "Starting f2()" << std::endl;
    try{
        f3();
    }catch(int ex){
        std::cout << "Exception handled in f2()" << std::endl;
    }
    std::cout << "Ending f2()" << std::endl;
}
```

```
void f3(){
    std::cout << "Starting f3()" << std::endl;
    try{
        exception_thrower();
    }catch(int ex){
        std::cout << "Exception handled in f3()" << std::endl;
    }
    std::cout << "Ending f3()" << std::endl;
}
```

```
int main(int argc, char **argv)
{
    try{
        f1();
    }catch(int ex){
        std::cout << "Handling execution in main()" << std::endl;
    }
    std::cout << "main() finishing up" << std::endl;
    return 0;
}
```

Slide intentionally left empty

Nested try blocks

```
void some_function(){
    for(size_t i{} ; i < 15 ;++i){
        std::cout << "Iteration : " << i << std::endl;

        try{ // Outer try block
            //Exceptions thrown in this scope are
            //handled in outer catch blocks
            if(i ==2 ){
                throw "exception for int 2 thrown" ;// Throws const char*
            }

            try{ // Inner try block ...

        }
        catch(const char* ex){
            std::cout<< "Outer catch(const char*) block , cought  :" << ex  << std::endl;
        }
        catch(size_t ex){
            std::cout << "Outer catch(size_t) block, cought " << ex << std::endl;
        }
    }
}
```

Slide intentionally left empty

Throwing class objects

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string s)
        : m_message(s)
        {
        }

    //Copy Constructor
    SomethingIsWrong(const SomethingIsWrong& source){

    //Destructor
    ~SomethingIsWrong(){
        std::cout << "SomethingIsWrong destructor called" << std::endl;
    }
    std::string_view what()const{
        return m_message;
    }
private :
    std::string m_message;
};
```

```
void do_something(size_t i){
      if(i == 2){
          throw SomethingIsWrong("i is 2");
      }
      std::cout << "Doing something at iteration : " << i << std::endl;
}

int main(int argc, char **argv)
{
    for(size_t i{0}; i < 5 ; ++i){

      try{
          do_something(i);
      }
      catch(SomethingIsWrong& ex){
          std::cout << "Exception cought : " << ex.what() << std::endl;
      }
    }

    return 0;
}
```

Slide intentionally left empty

Exceptions as class objects & Inheritance hierarchies

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string& s) : m_message(s){}
     ~SomethingIsWrong(){}
     const std::string& what()const{return m_message;}
protected :
    std::string m_message;
};

class Warning : public SomethingIsWrong{
    public :
    Warning(const std::string& s) : SomethingIsWrong(s){}
};

class SmallError : public Warning{
    public :
    SmallError(const std::string& s) : Warning(s){}
};

class CriticalError : public SmallError{
    public :
    CriticalError(const std::string& s) : SmallError(s){}
};
```

```
void do_something(size_t i){
      if(i == 2){
          throw CriticalError("i is 2");
      }

      if(i == 3){
          throw SmallError("i is 3");
      }
      std::cout << "Doing something at iteration : " << i << std::endl;
}
```

```
int main(int argc, char **argv)
{
    for(size_t i{0}; i < 5 ; ++i){

      try{
          do_something(i);
      }
      catch(CriticalError& ex){
          std::cout << "CriticalError Exception cought : " << ex.what() << std::endl;
      }
      catch(SmallError& ex){
          std::cout << "SmallError Exception cought : " << ex.what() << std::endl;
      }
      catch(Warning& ex){
          std::cout << "Warning Exception cought : " << ex.what() << std::endl;
      }
      catch(SomethingIsWrong& ex){
          std::cout << "SomethingIsWrong Exception cought : " << ex.what() << std::endl;
      }
    }
    return 0;
}
```

Slide intentionally left empty

Polymorphic Exceptions

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string& s) : m_message(s){}
    virtual ~SomethingIsWrong(){}
    virtual const std::string& what()const{return m_message;}
protected :
    std::string m_message;
};

class Warning : public SomethingIsWrong{
    public :
    Warning(const std::string& s) : SomethingIsWrong(s){}
};

class SmallError : public Warning{
    public :
    SmallError(const std::string& s) : Warning(s){}
};

class CriticalError : public SmallError{
    public :
    CriticalError(const std::string& s) : SmallError(s){}
};
```

```
void do_something(size_t i){
      if(i == 2){
          throw CriticalError("i is 2");
      }

      if(i == 3){
          throw SmallError("i is 3");
      }
      std::cout << "Doing something at iteration : " << i << std::endl;
}
```

```
int main(int argc, char **argv)
{
    std::cout << "Object Exceptions with Class Inheritance Hierarchies" << std::endl;

    for(size_t i{0}; i < 5 ; ++i){

      try{
          do_something(i);
      }

      catch(SomethingIsWrong& ex){
          std::cout << "SomethingIsWrong Exception cought : " << ex.what() << std::endl;

          //Using typeid()
          //std::cout << typeid(ex).name()  << "Exception cought : " << ex.what() << std::endl;
      }
    }
    return 0;
}
```

Slide intentionally left empty

Rethrown exceptions

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string& s) : m_message(s){}
    virtual ~SomethingIsWrong(){}
    virtual const std::string& what()const{return m_message;}
    void set_message(const std::string& m) {m_message = m;}
protected :
    std::string m_message;
};

class Warning : public SomethingIsWrong{
    public :
    Warning(const std::string& s = "Warning Object") : SomethingIsWrong(s){}
};

class SmallError : public Warning{
    public :
    SmallError(const std::string& s = "SmallError object") : Warning(s){}
};

class CriticalError : public SmallError{
    public :
    CriticalError(const std::string& s = "CriticalError object" ) : SmallError(s){}
};
```

```
void do_something(size_t i){
    Warning w;
    SmallError s_e;
    CriticalError c_e;

    if(i == 1){
        w.set_message("Warning Object for iteration : 1 ");
        throw w;
    }

    if(i == 2){
        s_e.set_message("SmallError Object for iteration :  2 ");
        throw s_e;
    }

    if(i == 3){
        c_e.set_message("CriticalError Object for iteration :  3 ");
        throw c_e;
    }

      std::cout << "Doing something at iteration : " << i << std::endl;
}
```

```
for(size_t i{0}; i < 5 ; ++i){
    try{
        try{
            do_something(i);
        }
        catch(SomethingIsWrong& ex_inner){
            if(typeid(ex_inner) == typeid(Warning)){
                std::cout << typeid(ex_inner).name()  <<
                    " -Inner catch block ,Exception cought : " <<
                    ex_inner.what() << std::endl;
            }else{
                throw;
                //throw ex_inner;//This will do a copy, and there will be slicing.Beware.
            }
        }
    }
    catch(SomethingIsWrong& ex_outer){
        std::cout << typeid(ex_outer).name() <<
                " -Outer catch block, Exception cought : " <<
                    ex_outer.what() << std::endl;
    }
}//End of for loop
```

Slide intentionally left empty

Program custom termination

```
int main(int argc, char **argv)
{
    throw 1;

    return 0;
}
```

```
C++ is wired this way :
      . if an exception is not handled anywhere in the app,
         the function std::terminate() located in <exception>
          will be called

      . std::terminate will in turn call std::abort() located
          in <cstdlib> to actually kill the program
```

```
void our_terminate_function(){
    std::cout << "Our custom terminate function called" << std::endl;
    std::cout << "Program will terminate in 10s ..." << std::endl;
    //std::this_thread::sleep_for(std::chrono::milliseconds(10000));
    std::abort();
}

int main(int argc, char **argv)
{
    std::set_terminate(&our_terminate_function);
     /*
    std::set_terminate([](){
            std::cout << "Our custom terminate function called" << std::endl;
            std::cout << "Program will terminate in 10s ..." << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(10000));
            std::abort();
    });
    */

    throw 1;

    return 0;
}
```

If you don’t call std::abort() in your terminate function, the system will still kill your application

Slide intentionally left empty

Ellipsis catch all block

```
void some_function(size_t i){
    if(i == 0)
        throw 1;
    if(i ==2)
        throw "const char*-Hello there";
    if(i==3)
        throw CriticalError();
    if(i==4)
        throw std::string("std::string-Hello there");
}
```

```
int main(int argc, char **argv)
{
    try{
        for( size_t i{}; i < 5; ++i){
            try{
                some_function(i);
            }
            catch(int ex){
                std::cout << "int handler- Cought an integer" << std::endl;
            }
            catch(...){
                std::cout << "Inner... handler , Cought some exception" << std::endl;
               // throw;
            }
        }
    }
    catch(const std::string& ex){
        std::cout << "Cought some string exception" << std::endl;
    }
    catch(...){
        std::cout << "Outer ...handler cought some other exception" << std::endl;
    }
    return 0;
}
```

Slide intentionally left empty

noexcept specifier

noexcept method

```
class Item{
public :
        Item(){}

        void do_something_in_class() const noexcept{
            std::cout << "Doing something from class" << std::endl;
            try{
                throw 1;
            }
            catch(int ex){
                std::cout << "Handling exception in Item::do_something_in_class" << std::endl;
                //throw; // Rethrowing in noexcept function/method will terminate program
            }
        }
private :
        int m_member_var;
};
```

noexcept free function

```
void some_function() noexcept{

    try{
        throw 1;
    }
    catch(int ex){
        std::cout << "Handling int excpetion in free function some_function()" << std::endl;
        throw;
    }

}
```

Slide intentionally left empty

Exceptions in destructors

```
class Item{
public :
    Item(){}
    ~Item(){
    throw;
    }
};
```

```
class Item{
public :
    Item(){}
    ~Item() noexcept(false) {

        throw;

    }
};
```

Slide intentionally left empty

Standard exceptions

exception

logic_error	runtime_error	others
. invalid_argument . length_error . out_of_range	. overflow_error . underflow_error	. bad_alloc . bad_cast

std::exception

Defined in header <exception>

```
class exception;
```

Provides consistent interface to handle errors through the throw expression.

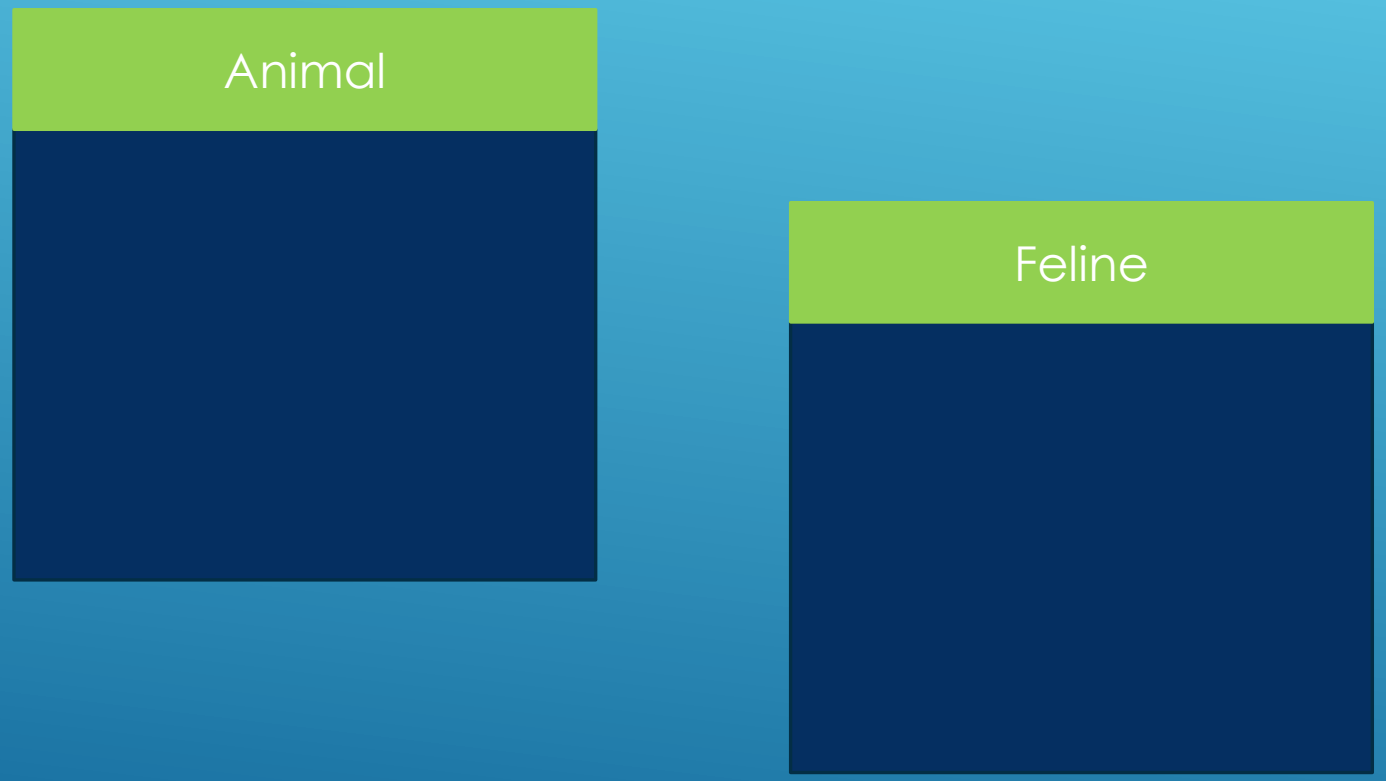
All exceptions generated by the standard library inherit from **std::exception**

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - system_error(C++11)

* Source : https://en.cppreference.com/w/cpp/error/exception

Slide intentionally left empty

Catching Standard Exceptions


Animal
Feline

```
int main(int argc, char **argv)
{
    Animal animal1;

    try{
        Feline feline {dynamic_cast<Feline&>(animal1)};
    }catch(std::exception& ex){
        std::cout << "Something is wrong : " << ex.what() << std::endl;
    }

    return 0;
}
```

Slide intentionally left empty

Throwing standard exceptions

```
class Students{
public :
    Students() = delete;
    Students(std::string_view s1, std::string_view s2, std::string_view s3,
                std::string_view s4, std::string_view s5){
                m_students[0] = s1;m_students[1] = s2;
                m_students[2] = s3;m_students[3] = s4;
                m_students[4] = s5;
    }
    ~Students() = default;

    std::string_view get_student(size_t index){
        const std::string message = "Index out of range, valid range["
                + std::to_string(0) + "," + std::to_string(4)
                + "]";

        if( (index < 0) || (index >= 5))
            throw std::out_of_range(message);
        return m_students[index];

    }
private :
    std::string m_students[5];
};
```

```
int main(int argc, char **argv)
{
    Students students("John Snow", "Terry Boomd",
                      "Nicholai Itchenko" , "Bilom Atunde" , "Lily Park");

    try{
        std::cout << students.get_student(2) << std::endl;
        std::cout << students.get_student(-2) << std::endl;
    }catch(std::exception& ex){

        std::cout << "Exception cought : " << ex.what() << std::endl;
    }

    return 0;
}
```

Slide intentionally left empty

Deriving From Standard Exceptions


```
class DivideByZeroException : public std::exception {
public :
    DivideByZeroException(int a, int b) noexcept : std::exception(),m_a(a),m_b(b){}

     virtual const char* what() const noexcept override {
         return "Divide by zero detected";
     }

     int get_a() const{
         return m_a;
     }

     int get_b() const{
         return m_b;
     }

private :
     int m_a{};
     int m_b{};

};
```

Slide intentionally left empty

Exceptions : Summary

A C++ built in mechanism to bring problems to the surface and possibly handle them

- static_assert
- assert
- std::cout

Try catch blocks : Syntax

```
int a{10};
int b{0};

try{
    Item item; // When exception is thrown, control immediately exits the try block
               // an automatic local variables are released
               // But pointers may leak memory.
    if( b == 0 )
        throw 0;
    a++; // Just using a and b in here, could use them to do anything.
    b++;
    std::cout << "Code that executes when things are fine" << std::endl;

}catch(int ex){
    std::cout << "Something went wrong. Exception thrown : " << ex <<  std::endl;
}

std::cout << "END." << std::endl;
```

The need for exceptions : Integer division by 0

```
//int division by 0 : CRASH

const int a{45};
const int b{0};
int result = a/b;
```

The need for exceptions : Failing downcast

```
//Downcast using dynamic_cast with references
//The hierarchy of Animal->Dog has to be polymorphic to be able to do
//this. This throws an exception because animal has no dog part so the
//cast won't really work.

Animal a;
Dog& d{dynamic_cast<Dog&>(a)};
```

Handling exceptions at different levels

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    f2();
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
Ending f3()
Ending f2()
Ending f1()
END.
```

Multiple handlers at different levels

```
void f1(){
    std::cout << "Starting f1()" << std::endl;
    try{
        f2();
    }catch(int ex){
        std::cout << "Handling execution in f1()" << std::endl;
    }
    std::cout << "Ending f1()" << std::endl;
}

void f2(){
    std::cout << "Starting f2()" << std::endl;
    f3();
    std::cout << "Ending f2()" << std::endl;
}

void f3(){
    std::cout << "Starting f3()" << std::endl;
    exception_thrower();
    std::cout << "Ending f3()" << std::endl;
}
```

C:\Windows\SYSTEM32\cmd.exe

```
Starting f1()
Starting f2()
Starting f3()
starting execution_thrower()
Handling execution in f1()
Ending f1()
END.
```

Nested try blocks

```
void some_function(){
    for(size_t i{} ; i < 15 ;++i){
        std::cout << "Iteration : " << i << std::endl;

        try{ // Outer try block
            //Exceptions thrown in this scope are
            //handled in outer catch blocks
            if(i ==2 ){
                throw "exception for int 2 thrown" ;// Throws const char*
            }

            try{ // Inner try block ...

        }
        catch(const char* ex){
            std::cout<< "Outer catch(const char*) block , cought  :" << ex  << std::endl;
        }
        catch(size_t ex){
            std::cout << "Outer catch(size_t) block, cought " << ex << std::endl;
        }
    }
}
```

Throwing class objects

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string s)
        : m_message(s)
        {
        }

    //Copy Constructor
    SomethingIsWrong(const SomethingIsWrong& source){

    //Destructor
    ~SomethingIsWrong(){
        std::cout << "SomethingIsWrong destructor called" << std::endl;
    }
    std::string_view what()const{
        return m_message;
    }
private :
    std::string m_message;
};
```

Class objects that are part of a inheritance hierarchy

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string& s) : m_message(s){}
     ~SomethingIsWrong(){}
     const std::string& what()const{return m_message;}
protected :
    std::string m_message;
};

class Warning : public SomethingIsWrong{
    public :
    Warning(const std::string& s) : SomethingIsWrong(s){}
};

class SmallError : public Warning{
    public :
    SmallError(const std::string& s) : Warning(s){}
};

class CriticalError : public SmallError{
    public :
    CriticalError(const std::string& s) : SmallError(s){}
};
```

Class objects that are part of a inheritance hierarchy

```
int main(int argc, char **argv)
{
    for(size_t i{0}; i < 5 ; ++i){

      try{
          do_something(i);
      }
      catch(CriticalError& ex){
          std::cout << "CriticalError Exception cought : " << ex.what() << std::endl;
      }
      catch(SmallError& ex){
          std::cout << "SmallError Exception cought : " << ex.what() << std::endl;
      }
      catch(Warning& ex){
          std::cout << "Warning Exception cought : " << ex.what() << std::endl;
      }
      catch(SomethingIsWrong& ex){
          std::cout << "SomethingIsWrong Exception cought : " << ex.what() << std::endl;
      }
    }
    return 0;
}
```

Polymorphic inheritance hierarchies

```
class SomethingIsWrong{
public :
    SomethingIsWrong(const std::string& s) : m_message(s){}
    virtual ~SomethingIsWrong(){}
    virtual const std::string& what()const{return m_message;}
protected :
    std::string m_message;
};

class Warning : public SomethingIsWrong{
    public :
    Warning(const std::string& s) : SomethingIsWrong(s){}
};

class SmallError : public Warning{
    public :
    SmallError(const std::string& s) : Warning(s){}
};

class CriticalError : public SmallError{
    public :
    CriticalError(const std::string& s) : SmallError(s){}
};
```

Polymorphic inheritance hierarchies

```
int main(int argc, char **argv)
{
    std::cout << "Object Exceptions with Class Inheritance Hierarchies" << std::endl;

    for(size_t i{0}; i < 5 ; ++i){

      try{
          do_something(i);
      }

      catch(SomethingIsWrong& ex){
          std::cout << "SomethingIsWrong Exception cought : " << ex.what() << std::endl;

          //Using typeid()
          //std::cout << typeid(ex).name()  << "Exception cought : " << ex.what() << std::endl;
      }
    }
    return 0;
}
```

Rethrown exceptions

```
for(size_t i{0}; i < 5 ; ++i){
    try{
        try{
            do_something(i);
        }
        catch(SomethingIsWrong& ex_inner){
            if(typeid(ex_inner) == typeid(Warning)){
                std::cout << typeid(ex_inner).name()  <<
                    " -Inner catch block ,Exception cought : " <<
                    ex_inner.what() << std::endl;
            }else{
                throw;
                //throw ex_inner;//This will do a copy, and there will be slicing.Beware.
            }
        }
    }
    catch(SomethingIsWrong& ex_outer){
        std::cout << typeid(ex_outer).name() <<
                " -Outer catch block, Exception cought : " <<
                    ex_outer.what() << std::endl;
    }
}//End of for loop
```

Program custom termination

```
void our_terminate_function(){
    std::cout << "Our custom terminate function called" << std::endl;
    std::cout << "Program will terminate in 10s ..." << std::endl;
    //std::this_thread::sleep_for(std::chrono::milliseconds(10000));
    std::abort();
}

int main(int argc, char **argv)
{
    std::set_terminate(&our_terminate_function);
     /*
    std::set_terminate([](){
            std::cout << "Our custom terminate function called" << std::endl;
            std::cout << "Program will terminate in 10s ..." << std::endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(10000));
            std::abort();
    });
    */

    throw 1;

    return 0;
}
```

Ellipsis catch all block

```
int main(int argc, char **argv)
{
    try{
        for( size_t i{}; i < 5; ++i){
            try{
                some_function(i);
            }
            catch(int ex){
                std::cout << "int handler- Cought an integer" << std::endl;
            }
            catch(...){
                std::cout << "Inner... handler , Cought some exception" << std::endl;
               // throw;
            }
        }
    }
    catch(const std::string& ex){
        std::cout << "Cought some string exception" << std::endl;
    }
    catch(...){
        std::cout << "Outer ...handler cought some other exception" << std::endl;
    }
    return 0;
}
```

noexcept specifier

```
void some_function() noexcept{

    try{
        throw 1;
    }
    catch(int ex){
        std::cout << "Handling int excpetion in free function some_function()" << std::endl;
        throw;
    }

}
```

Noexcept destructors

```
class Item{
public :
    Item(){}
    ~Item(){
    throw;
    }
};
```

Standard exceptions

```
exception

logic_error                runtime_error              others

    . invalid_argument         . overflow_error           . bad_alloc
    . length_error             . underflow_error          . bad_cast
    . out_of_range
```

Standard exceptions

```
exception

logic_error                runtime_error              others

    . invalid_argument         . overflow_error           . bad_alloc
    . length_error             . underflow_error          . bad_cast
    . out_of_range
```

- Catching/Handling
- Throwing
- Deriving

Slide intentionally left empty

