Maciej Sobieraj

Lecture 8

Outline

1. **Exceptions**
    1. To err is human
    2. Throw statement in detail
    3. Categorizing exceptions
    4. **Catching exceptions**
    5. Exceptions in action

Different catches for different purposes

- catch “catches” only these exceptions that are compatible in type with the catch header.
  - catch(string excp) { … }
- catches exceptions encapsulated inside objects of type string, and ignores all others.

Different catches for different purposes

- There's a specialized form of the *catch* that's able to catch literally all passing exceptions – it looks like:
  - catch(…) { … }
- but in contrast to the previous form, it can **neither identify the exception object, nor make any use of it**

Different catches for different purposes

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw out_of_range("0");
	case 1: throw overflow_error("1");
	case 2: throw domain_error("2");
	}
}
int main(void) {
	for(int i = 0; i < 3; i++) {
		try {
			function(i);
		}
		catch(...) {
			cout << "Exception caught!" << endl;
		}
	}
	return 0;
}
```

Different catches for different purposes

- The program produces the following output:
  - Exception caught!
  - Exception caught!
  - Exception caught!

Different catches for different purposes

- We've changed the *catch* header and added the "*exception ex*" instead of the ellipsis.
- The branch is allowed to catch all exceptions whose objects are compatible in type with the *exceptions* class.
- We can identify an object, name it locally (as *ex*) and make use of its properties and/or functions.
- We invoke the *what* function to find out what the object wants to say about itself.

Different catches for different purposes

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw out_of_range("0");
	case 1: throw overflow_error("1");
	case 2: throw domain_error("2");
	case 3: throw exception();
	}
}
int main(void) {
	for(int i = 0; i < 4; i++) {
		try {
			function(i);
		}
		catch(exception &ex) {
			cout << "Exception caught: " << ex.what() << endl;
		}
	}
	return 0;
}
```

Different catches for different purposes

- The modified program → produces the following output:
  - Exception caught: 0
  - Exception caught: 1
  - Exception caught: 2
  - Exception caught: Unknown exception

Different catches for different purposes

- If we're going to, or if we have to, provide different ways of handling different exceptions, **we're allowed to specify as many different catch branches as we want**

Different catches for different purposes

```
#include <iostream>
#include  <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
      switch(i) {
      case 0: throw out_of_range("0");
      case 1: throw overflow_error("1");
      case 2: throw domain_error("2");
      case 3: throw exception();
      }
}
int main(void) {
      for(int i = 0; i < 4; i++) {
            try {
                  function(i);
            }
            catch(out_of_range &ofr) {
                  cout << "Out of range: " << ofr.what() << endl;
            }
            catch(overflow_error &ovf) {
                  cout << "Overflow: " << ovf.what() << endl;
            }
            catch(domain_error &dmn) {
                  cout << "Domain: " << dmn.what() << endl;
            }
            catch(exception &ex) {
                  cout << "Exception: " << ex.what() << endl;
            }
      }
      return 0;
}
```

Different catches for different purposes

- The program outputs the following text:
  - Out of range: 0
  - Overflow: 1
  - Domain: 2
  - Exception: Unknown exception

Different catches for different purposes

- There’s no need to choose between “all or none”.
- We can selectively choose the exceptions we want to catch and handle carefully, and those that we want to handle very briefly.
- **some of the exceptions are caught individually** while **others go to the ellipsis**

Different catches for different purposes

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw out_of_range("0");
	case 1: throw overflow_error("1");
	case 2: throw domain_error("2");
	case 3: throw exception();
	case 4: throw "so bad";
	}
}
int main(void) {
	for(int i = 0; i < 5; i++) {
		try {
			function(i);
		}
		catch(out_of_range &ofr) {
			cout << "Out of range: " << ofr.what() << endl;
		}
		catch(overflow_error &ovf) {
			cout << "Overflow: " << ovf.what() << endl;
		}
		catch(domain_error &dmn) {
			cout << "Domain: " << dmn.what() << endl;
		}
		catch(exception &ex) {
			cout << "Exception: " << ex.what() << endl;
		}
		catch(...) {
			cout << "Something bad happened" << endl;
		}
	}
	return 0;
}
```

Different catches for different purposes

- The program outputs the following text:
  - Out of range: 0
  - Overflow: 1
  - Domain: 2
  - Exception: Unknown exception
  - Something bad happened

Order of the catch branches

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw out_of_range("0");
	case 1: throw overflow_error("1");
	case 2: throw domain_error("2");
	case 3: throw exception();
	case 4: throw "so bad";
	}
}
int main(void) {
	for(int i = 0; i < 5; i++) {
		try {
			function(i);
		}
		catch(overflow_error &ovf) {
			cout << "Overflow: " << ovf.what() << endl;
		}
		catch(out_of_range &ofr) {
			cout << "Out of range: " << ofr.what() << endl;
		}
		catch(domain_error &dmn) {
			cout << "Domain: " << dmn.what() << endl;
		}
		catch(exception &ex) {
			cout << "Exception: " << ex.what() << endl;
		}
		catch(...) {
			cout << "Something bad happened" << endl;
		}
	}
	return 0;
}
```

Order of the catch branches

- The program produces exactly the same output as the previous one:
  - Out of range: 0
  - Overflow: 1
  - Domain: 2
  - Exception: Unknown exception
  - Something bad happened

Order of the catch branches

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw out_of_range("0");
    case 1: throw overflow_error("1");
    case 2: throw domain_error("2");
    case 3: throw exception();
    case 4: throw "so bad";
    }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            function(i);
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(out_of_range &ofr) {
            cout << "Out of range: " << ofr.what() << endl;
        }
        catch(overflow_error &ovf) {
            cout << "Overflow: " << ovf.what() << endl;
        }
        catch(domain_error &dmn) {
            cout << "Domain: " << dmn.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

Order of the catch branches

- The program produces this output, which justifies the warnings:
  - Exception: 0
  - Exception: 1
  - Exception: 2
  - Exception: Unknown exception
  - Something bad happened

Order of the catch branches

- When the exception arrives at a set of *catch* branches, **the first compatible branch is chosen** (and only this one) as a target handler.
- This means that when **a more general type/class is placed before the more specific compatible type/class, the second branch will receive no exceptions at all**.

Order of the catch branches

```
#include<iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw domain_error("0");
	case 1: throw logic_error("1");
	case 2: throw exception();
	case 3: throw range_error("2");
	case 4: throw "so bad";
	}
}
int main(void) {
	for(int i = 0; i < 5; i++) {
		try {
			function(i);
		}
		catch(exception &ex) {
			cout << "Exception: " << ex.what() << endl;
		}
	}
	return 0;
}
```

Order of the catch branches

- The beginning of the output will look as follows:
  - Exception: 0
  - Exception: 1
  - Exception: Unknown exception
  - Exception: 2
- but immediately after this you'll see some alarming system messages and our program will terminate abnormally.
- The exception carried by the *string* type exception is, in a certain sense, **orphaned**: there's no *catch* branch wanting to receive it.

Order of the catch branches

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
	switch(i) {
	case 0: throw domain_error("0");
	case 1: throw logic_error("1");
	case 2: throw exception();
	case 3: throw range_error("2");
	case 4: throw "so bad";
	}
}
int main(void) {
	for(int i = 0; i < 5; i++) {
		try {
			function(i);
		}
		catch(exception &ex) {
			cout << "Exception: " << ex.what() << endl;
		}
		catch(...) {
			cout << "Something bad happened" << endl;
		}
	}
	return 0;
}
```

Order of the catch branches

- Our repaired program produces the following output:
  - Exception: 0
  - Exception: 1
  - Exception: Unknown exception
  - Exception: 2
  - Something bad happened

Order of the catch branches

- Can you predict its output?

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw domain_error("0");
    case 1: throw logic_error("1");
    case 2: throw exception();
    case 3: throw range_error("2");
    case 4: throw "so bad";
    }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            function(i);
        }
        catch(logic_error &le) {
            cout << "Logic error: " << le.what() << endl;
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

exception ← logic_error ← domain_error

exception ← runtime_error ← range_error

Order of the catch branches

- Your answer should look like this:
  - Logic error: 0
  - Logic error: 1
  - Exception: Unknown exception
  - Exception: 2
  - Something bad happened

exception ← logic_error ← domain_error

exception ← runtime_error ← range_error

Order of the catch branches

- Can you predict its output?

exception ← logic_error ← domain_error

exception ← runtime_error ← range_error

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw domain_error("0");
    case 1: throw logic_error("1");
    case 2: throw exception();
    case 3: throw range_error("2");
    case 4: throw "so bad";
    }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            function(i);
        }
        catch(logic_error &le) {
            cout << "Logic error: " << le.what() << endl;
        }
        catch(runtime_error &re) {
            cout << "Runtime error: " << re.what() << endl;
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

Order of the catch branches

- The answer is:
  - Logic error: 0
  - Logic error: 1
  - Exception: Unknown exception
  - Runtime error: 2
  - Something bad happened

exception ← logic_error ← domain_error

exception ← runtime_error ← range_error

Sharing the responsibility

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
      switch(i) {
      case 0: throw domain_error("0");
      case 1: throw logic_error("1");
      case 2: throw exception();
      case 3: throw range_error("2");
      case 4: throw "so bad";
      }
}
void broker(int i) {
      try { function(i); }
      catch(exception ex) { cout << "Broker - exception: " << ex.what() << endl; }
}
int main(void) {
      for(int i = 0; i < 5; i++) {
            try {
                  broker(i);
            }
            catch(logic_error &le) {
                  cout << "Logic error: " << le.what() << endl;
            }
            catch(runtime_error &re) {
                  cout << "Runtime error: " << re.what() << endl;
            }
            catch(exception &ex) {
                  cout << "Exception: " << ex.what() << endl;
            }
            catch(...) {
                  cout << "Something bad happened" << endl;
            }
      }
      return 0;
}
```

Sharing the responsibility

- Now **the handling process is dispersed over two levels**: lower (inside *broker*) and upper (inside *main*).
- The output of the program is as follows:
  - Broker - exception: 0
  - Broker - exception: 1
  - Broker - exception: Unknown exception
  - Broker - exception: 2
  - Something bad happened

Sharing the responsibility

- Can you predict its output?

```
#include <stdexcept>
#include <exception>
#include <iostream>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw domain_error("0");
    case 1: throw logic_error("1");
    case 2: throw exception();
    case 3: throw range_error("2");
    case 4: throw "so bad";
    }
}
void broker(int i) {
    try { function(i); }
    catch(logic_error &le) { cout << "Broker - logic_error: " << le.what() << endl; }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            broker(i);
        }
        catch(logic_error &le) {
            cout << "Logic error: " << le.what() << endl;
        }
        catch(runtime_error &re) {
            cout << "Runtime error: " << re.what() << endl;
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

Sharing the responsibility

- It’ll look like this:
  - Broker - logic error: 0
  - Broker - logic error: 1
  - Exception: Unknown exception
  - Runtime error: 2
  - Something bad happened

Sharing the responsibility

- A badly constructed broker may ruin the exception handling logic at higher levels.
- The broker’s decided to take control over all arriving exceptions.
- None of them will leave the broker.

Sharing the responsibility

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw domain_error("0");
    case 1: throw logic_error("1");
    case 2: throw exception();
    case 3: throw range_error("2");
    case 4: throw "so bad";
    }
}
void broker(int i) {
    try {
        function(i);
    }
    catch(...) {
        cout << "Broker swept problems under the carpet " << endl;
    }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            broker(i);
        }
        catch(logic_error &le) {
            cout << "Logic error: " << le.what() << endl;
        }
        catch(runtime_error &re) {
            cout << "Runtime error: " << re.what() << endl;
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

Sharing the responsibility

- The output of the program isn't really varied – this is how it looks:
  - Broker swept problems under the carpet
  - Broker swept problems under the carpet
  - Broker swept problems under the carpet
  - Broker swept problems under the carpet
  - Broker swept problems under the carpet

Sharing the responsibility

- The responsibility of handling exceptions may not only be **divided** – it may be **shared**, too.
- This means that **the handling of the same exceptions may be provided at more than one level**.
- Note that any of the *catch* branches might throw an exception too, and the exception won't be handled in the place where it was created, but at a higher level.

Sharing the responsibility

- Using the argument-less *throw* instruction means:
  - *throw the same exception you just got*
- there are no obstacles to using less anonymous variants, like this one:
  - **catch**(exception ex) {
  - **throw** ex;
  - }

Sharing the responsibility

- Note that you can throw another (new) exception instead of throwing the received exception.
- This might be a good idea when you want to change the category of the exception.
- Here's an example:
  - **catch**(logic_error err) {
  - **throw** "We have a problem";
  - }

Sharing the responsibility

```
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;
void function(int i) {
    switch(i) {
    case 0: throw domain_error("0");
    case 1: throw logic_error("1");
    case 2: throw exception();
    case 3: throw range_error("2");
    case 4: throw "so bad";
    }
}
void broker(int i) {
    try {
        function(i);
    }
    catch(...) {
        cout << "Broker swept problems under the carpet " << endl;
        throw;
    }
}
int main(void) {
    for(int i = 0; i < 5; i++) {
        try {
            broker(i);
        }
        catch(logic_error &le) {
            cout << "Logic error: " << le.what() << endl;
        }
        catch(runtime_error &re) {
            cout << "Runtime error: " << re.what() << endl;
        }
        catch(exception &ex) {
            cout << "Exception: " << ex.what() << endl;
        }
        catch(...) {
            cout << "Something bad happened" << endl;
        }
    }
    return 0;
}
```

Sharing the responsibility

- The example program produces the following output:
  - Broker swept problems under the carpet
  - Logic error: 0
  - Broker swept problems under the carpet
  - Logic error: 1
  - Broker swept problems under the carpet
  - Exception: Unknown exception
  - Broker swept problems under the carpet
  - Runtime error: 2
  - Broker swept problems under the carpet
  - Something bad happened

Outline

1. **Exceptions**
    1. To err is human
    2. Throw statement in detail
    3. Categorizing exceptions
    4. Catching exceptions
    5. **Exceptions in action**

Stack again

- We’ve introduced two important amendments compared to its previous incarnations:
  - the values stored in the stack are still located inside an array, but **the size of the array is defined dynamically by the constructor**; note the default stack size specification
  - as a consequence of the previous modification, we’ve had to add **a destructor responsible for removing the array** at the end of the stack’s life

Stack again

```
class Stack {
   private:
      int *stackstore;
      int stacksize;
      int SP;
   public:
      Stack(int size = 100);
      ~Stack();
      void push(int value);
      int pop(void);
};
Stack::Stack(int size) {
      stackstore = new int[size];
      stacksize = size;
      SP = 0;
}
Stack::~Stack(void) {
      delete []stackstore;
}
void Stack::push(int value) {
      stackstore[SP++] = value;
}
int Stack::pop(void) {
      return stackstore[--SP];
}

#include <iostream>
using namespace std;
int main(void) {
      Stack stk;
      stk.push(1);
      cout << stk.pop() << endl;
      return 0;
}
```

Stack again – new exceptions

- Let's try to identify all the "bad" surprises our stack may face in its life. We can see four of them right here:
  1. improper stack size specification (less or equal to zero)
  2. failure in allocating memory for the stack
  3. invoking push when the stack is full
  4. invoking pop when the stack is empty

Stack again – new exceptions

- We'll define our own exceptions for all these events. We think that:
  - #1 will be described by a new exception derived from the *length_error* class
  - #2 will be described by a new exception derived from *bad_alloc* class
  - #3 and #4 will be described by a new exception derived from *logic_error*

Stack again – new exceptions

1. stack_size_error
2. stack_bad_alloc
3. stack_overflow
4. stack_empty

Stack again – new exception classes

```
#include <iostream>
#include <exception>
#include <stdexcept>

class stack_size_error : public std::length_error {
public:
    explicit stack_size_error(const std::string &msg);
};
class stack_bad_alloc : public std::bad_alloc {
public:
    explicit stack_bad_alloc(void);
};
class stack_overflow : public std::logic_error {
public:
    explicit stack_overflow(const std::string &msg);
};
class stack_empty : public std::logic_error {
public:
    explicit stack_empty(const std::string &msg);
};
```

Stack again – new exception classes

```
stack_size_error::stack_size_error(const std::string &msg) : std::length_error(msg) {
};
stack_bad_alloc::stack_bad_alloc(void) : std::bad_alloc() {
};
stack_overflow::stack_overflow(const std::string &msg) : std::logic_error(msg) {
};
stack_empty::stack_empty(const std::string &msg) : std::logic_error(msg) {
};
```

Stack again – new stack declaration

- We expect that:
  - the constructor throws two exceptions: *stack_size_error* and *stack_bad_alloc*
  - the push function throws the *stack_overflow* exception
  - the pop function throws the *stack_empty* exception

Stack again – new stack declaration

```
class Stack {
   private:
      int *stackstore;
      int stacksize;
      int SP;
   public:
      Stack(int size = 100) throw(stack_size_error, stack_bad_alloc);
      ~Stack();
      void push(int value) throw(stack_overflow);
      int pop(void) throw(stack_empty);
};
```

Stack again – new stack declaration

- We've got two important things to do:
  - check whether the initial stack size isn't too low and will throw an exception in such a case
  - try to allocate memory for the stack and check if it was successful; we're going to re-throw our own exception in such a case

Stack again – new constructor

```
Stack::Stack(int size) throw(stack_size_error, stack_bad_alloc) {
    if(size <= 0)
        throw stack_size_error("size must be >= 0");
    try {
        stackstore = new int[size];
    } catch(std::bad_alloc ba) {
        throw stack_bad_alloc();
    }
    stacksize = size;
    SP = 0;
}
```

Stack again – new push

- Modifying the push function should be easy. We need to check if the *SP* hasn't exceeded its maximum allowable value (*stacksize* – 1) and we'll throw an event in such a case.

```
void Stack::push(int value) throw(stack_overflow) {
    if(SP == stacksize)
        throw stack_overflow("stack size exceeded");
    stackstore[SP++] = value;
}
```

Stack again – new pop

```
int Stack::pop(void) throw(stack_empty) {
        if(SP == 0)
                throw stack_empty("stack is empty");
        return stackstore[--SP];
}
```

Stack again – a header file for a new module

- First, we'll write the **header file** – a file containing all the necessary **declarations**.
- We've named it *mystack.h*.
- We've assumed that the file with all required definitions will be named *mystack.cpp*.
- The ***#ifndef*** directive is used by a pre-processor to check if **the compile-time symbol** is defined or not. In our example the checked symbol is named *__MYSTACK__*

Stack again – a header file for a new module

- If the symbol isn’t defined (*ndef*), the pre-processor will analyse the rest of the file, or skip it otherwise. Note that it doesn't skip the entire file content, but only the part nested between the ***#ifndef*** and ***#endif*** directives.
- The next directive, ***#define***, defines the *__MYSTACK__* symbol

Stack again – a header file for a new module

```
#ifndef __MYSTACK__
#define __MYSTACK__

#include <iostream>
#include <exception>
#include <stdexcept>

class stack_size_error : public std::length_error {
public:
      explicit stack_size_error(const std::string &msg);
};

class stack_bad_alloc : public std::bad_alloc {
public:
      explicit stack_bad_alloc(void);
};

class stack_overflow : public std::logic_error {
public:
      explicit stack_overflow(const std::string &msg);
};

class stack_empty : public std::logic_error {
public:
      explicit stack_empty(const std::string &msg);
};

class Stack {
   private:
      int *stackstore;
      int stacksize;
      int SP;
   public:
      Stack(int size = 100) throw(stack_size_error, stack_bad_alloc);
      ~Stack();
      void push(int value) throw(stack_overflow);
      int pop(void) throw(stack_empty);
};

#endif
```

Stack again - implementation

```
#include "mystack.h"

stack_size_error::stack_size_error(const std::string &msg) : std::length_error(msg) {
};

stack_bad_alloc::stack_bad_alloc(void) : std::bad_alloc() {
};

stack_overflow::stack_overflow(const std::string &msg) : std::logic_error(msg) {
};

stack_empty::stack_empty(const std::string &msg) : std::logic_error(msg) {
};

Stack::Stack(int size) throw(stack_size_error, stack_bad_alloc) {
    if(size <= 0)
        throw stack_size_error("size must be >= 0");
    try {
        stackstore = new int[size];
    } catch(std::bad_alloc ba) {
        throw stack_bad_alloc();
    }
    stacksize = size;
    SP = 0;
}

Stack::~Stack(void) {
    delete stackstore;
}

void Stack::push(int value) throw(stack_overflow) {
    if(SP == stacksize)
        throw stack_overflow("stack size exceeded");
    stackstore[SP++] = value;
}

int Stack::pop(void) throw(stack_empty) {
    if(SP == 0)
        throw stack_empty("stack is empty");
    return stackstore[--SP];
}
```

Stack again – main function

- Note that we've included the #include directive referring to the "mystack.h" header file. This is how the compiler learns about the stack and all of its components, as well as the exceptions we've jointly defined.
- The compilation process should look as follows.
  - the compiler compiles the "mystack.cpp" file and produces an object file (its name may be different on different platforms – some compilers may use "mystack.o", others "mystack.obj" – don't be surprised).
  - The compiler compiles the "main.cpp" file and produces an object file of a name, e.g. "main.obj"
  - The linker links both files, adding a code taken from standard libraries, and produces an executable file in the end.

Stack again – main function

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
	Stack stk;
	stk.push(1);
	cout << stk.pop() << endl;
	return 0;
}
```

Stack again – new, better main function

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
	try {
		Stack stk;
		stk.push(1);
		cout << stk.pop() << endl;
	}
	catch(stack_bad_alloc sba) {
		cout << "No room for the stack - sorry!" << endl;
	}
	catch(stack_size_error sse) {
		cout << "Stacks of that size don't exist - sorry!" << endl;
	}
	catch(stack_overflow se) {
		cout << "Stack is too small for that many pushes - sorry!" << endl;
	}
	catch(stack_empty su) {
		cout << "Stack is empty - sorry!" << endl;
	}
	return 0;
}
```

Stack again – crash tests

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
    try {
        Stack stk(0);
        stk.push(1);
        cout << stk.pop() << endl;
    }
    catch(stack_bad_alloc sba) {
        cout << "No room for the stack - sorry!" << endl;
    }
    catch(stack_size_error sse) {
        cout << "Stacks of that size don't exist - sorry!" << endl;
    }
    catch(stack_overflow se) {
        cout << "Stack is too small for that many pushes - sorry!" << endl;
    }
    catch(stack_empty su) {
        cout << "Stack is empty - sorry!" << endl;
    }
    return 0;
}
```

Stack again – crash tests

- First, we'll check if the constructor properly detects stack size values that are too low.
- Ok, it works fine – we've got:
  - Stacks of that size don't exist - sorry!

Stack again – crash tests

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
    try {
        Stack stk(2000000000);
        stk.push(1);
        cout << stk.pop() << endl;
    }
    catch(stack_bad_alloc sba) {
        cout << "No room for the stack - sorry!" << endl;
    }
    catch(stack_size_error sse) {
        cout << "Stacks of that size don't exist - sorry!" << endl;
    }
    catch(stack_overflow se) {
        cout << "Stack is too small for that many pushes - sorry!" << endl;
    }
    catch(stack_empty su) {
        cout << "Stack is empty - sorry!" << endl;
    }
    return 0;
}
```

Stack again – crash tests

- Next, we'll check if the constructor can handle our exorbitant demands on the stack size
- Ours can - we see:
  - No room for the stack - sorry!

Stack again – crash tests

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
	try {
		Stack stk(1);
		stk.push(1);
		stk.push(2);
		cout << stk.pop() << endl;
	}
	catch(stack_bad_alloc sba) {
		cout << "No room for the stack - sorry!" << endl;
	}
	catch(stack_size_error sse) {
		cout << "Stacks of that size don't exist - sorry!" << endl;
	}
	catch(stack_overflow se) {
		cout << "Stack is too small for that many pushes - sorry!" << endl;
	}
	catch(stack_empty su) {
		cout << "Stack is empty - sorry!" << endl;
	}
	return 0;
}
```

Stack again – crash tests

- Is our stack too-many-pushes-proof?
- Yes, it is – it says:
  - Stack is too small for that many pushes - sorry!

Stack again – crash tests

```
#include "mystack.h"
#include <iostream>

using namespace std;

int main(void) {
    try {
        Stack stk(1);
        stk.push(1);
        cout << stk.pop() << endl;
        cout << stk.pop() << endl;
    }
    catch(stack_bad_alloc sba) {
        cout << "No room for the stack - sorry!" << endl;
    }
    catch(stack_size_error sse) {
        cout << "Stacks of that size don't exist - sorry!" << endl;
    }
    catch(stack_overflow se) {
        cout << "Stack is too small for that many pushes - sorry!" << endl;
    }
    catch(stack_empty su) {
        cout << "Stack is empty - sorry!" << endl;
    }
    return 0;
}
```

Stack again – crash tests

- And what about too many pops?
- That’s okay too:
  - Stack is empty - sorry!