System Programming with C++

Exceptions

Exceptions

Exceptions: basic motivation (brief reminder)

- There are situations in the code where:
  - ✓ You can notice, that further execution of the code is impossible (will cause crash, UB or inconsistent object state)

Exceptions: basic motivation (brief reminder)

- There are situations in the code where:
  - ✓ You can notice, that further execution of the code is impossible (will cause crash, UB or inconsistent object state)
  - ✓ So, execution should be interrupted, and the problem should be somehow handled

Exceptions: basic motivation (brief reminder)

- There are situations in the code where:
  - ✓ You can notice, that further execution of the code is impossible (will cause crash, UB or inconsistent object state)
  - ✓ So, execution should be interrupted, and the problem should be somehow handled
  - ✓ Most probably in the caller function (you delegate that outside)

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

Do you see such situations here?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

What if Vector is empty?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

What if Vector is empty?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

What if idx is out of bounds?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

Can we fix such problems here?

What if idx is out of bounds?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

Can we fix such problems here?

Absolutely not.

What if idx is out of bounds?

Exceptions: basic motivation (brief reminder)

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

};
```

Can we fix such problems here?

Absolutely not. But we can notice them, interrupt the execution and somehow handle them outside of these methods.

What if idx is out of bounds?

Exceptions: basic syntax and mechanics

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    ...
    int& operator[](size_t idx) {
        return data_[idx];
    }

};
```

Exceptions: basic syntax and mechanics

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    ...
    int& operator[](size_t idx) {
        if (index >= size_) {
            throw "Out of bounds";
        }
        return data_[idx];
    }
};
```

Exceptions: basic syntax and mechanics

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    ...
    int& operator[](size_t idx) {
        if (index >= size_) {
            throw "Out of bounds";
        }
        return data_[idx];
    }
};
```

we've noticed a problem

Exceptions: basic syntax and mechanics

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    ...
    int& operator[](size_t idx) {
        if (index >= size_) {
            throw "Out of bounds";
        }
        return data_[idx];
    }
};
```

we've noticed a problem

we are interrupting the execution

Exceptions: basic syntax and mechanics

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    ...
    int& operator[](size_t idx) {
        if (index >= size_) {
            throw "Out of bounds";
        }
        return data_[idx];
    }
};
```

we've noticed a problem

we are interrupting the execution

here we create an exception object, this time it is const char*

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    std::cout << v[13] << std::endl;
    return 0;
}
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    std::cout << v[13] << std::endl;
    return 0;
}
```

```
terminate called after throwing an instance of 'char const*'
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    std::cout << v[13] << std::endl;
    return 0;
}
```

```
terminate called after throwing an instance of 'char const*'
```

this unhandled terminates an execution (somehow better than UB, but still bad situation, details later)

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout << v[13] << std::endl;
    } catch (const char* exception) {
        std::cout << "Something went wrong: " << exception << std::endl;
    }
    return 0;
}
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout << v[13] << std::endl;
    } catch (const char* exception) {
        std::cout << "Something went wrong: " << exception << std::endl;
    }
    return 0;
}
```

try-block

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout << v[13] << std::endl;
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    }
    return 0;
}
```

try-block

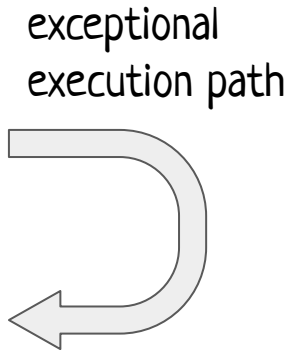
catch-block
(or handler)

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout << v[13] << std::endl;
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    }
    return 0;
}
```

try-block

catch-block
(or handler)



Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```

```
int getValue(Vector& v, size_t idx) {
   return v[idx];
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw "Out of bounds";
   }
   return data_[index];
}
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout <<getValue(v, 13);
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    }
    return 0;
}
```

```
int getValue(Vector& v, size_t idx) {
    return v[idx];
}
```

exceptional path

```
int& operator[](size_t index) {
    if (index >= size_) {
        throw "Out of bounds";
    }
    return data_[index];
}
```

Exceptions: basic syntax and mechanics

But this function can't handle the exception, so it delegates it

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout <<getValue(v, 13);
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    }
    return 0;
}
```

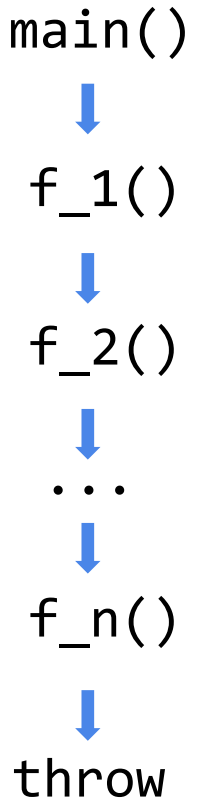
```
int getValue(Vector& v, size_t idx) {
    return v[idx];
}
```

exceptional path

```
int& operator[](size_t index) {
    if (index >= size_) {
        throw "Out of bounds";
    }
    return data_[index];
}
```

Exceptions: basic syntax and mechanics

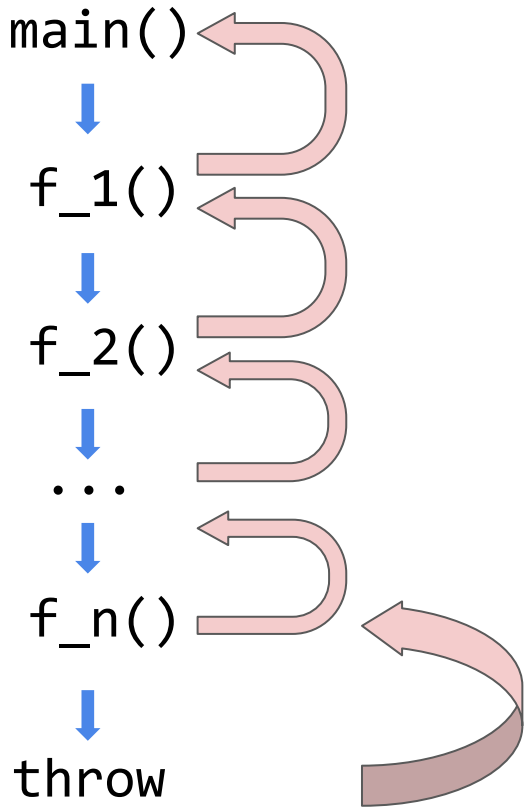
```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```



Exceptions: basic syntax and mechanics

Interrupts the execution till it finds corresponding handler

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```



Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```

Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   } catch (int code) {
      std::cout << code << std::endl;
   }
   return 0;
}
```

```
int getValue(Vector& v, size_t idx) {
   int result = v[idx];
   if (result > 13) {
       throw result;
   }
   return result;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw "Out of bounds";
   }
   return data_[index];
}
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout <<getValue(v, 13);
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    } catch (int code) {
        std::cout << code << std::endl;
    }
    return 0;
}
```

```
int getValue(Vector& v, size_t idx) {
    int result = v[idx];
    if (result > 13) {
        throw result;
    }
    return result;
}


int& operator[](size_t index) {
    if (index >= size_) {
        throw "Out of bounds";
    }
    return data_[index];
}
```

Exceptions: basic syntax and mechanics

```
int main() {
    Vector v;
    v.push(42);
    try {
        std::cout <<getValue(v, 13);
    } catch (const char* exception) {
        std::cout << "Something went wrong: "
                  << exception << std::endl;
    } catch (int code) {
        std::cout << code << std::endl;
    }
    return 0;
}
```

```
int getValue(Vector& v, size_t idx) {
    int result = v[idx];
    if (result > 13) {
        throw result;
    }
    return result;
}


int& operator[](size_t index) {
    if (index >= size_) {
        throw "Out of bounds";
    }
    return data_[index];
}
```

Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout << v[13];
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw "Out of bounds";
   }
   return data_[index];
}
```

Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout << v[13];
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```

```
int& operator[](size_t index) {
    if (index >= size_) {
        throw "Out of bounds";
    }
    return data_[index];
}
```

```
class OutOfBoundsEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       std::cout
       << "Out of bounds with index: "
       << index;
   }
};
```

Exceptions: basic syntax and mechanics

```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout << v[13];
   } catch (OutOfBoundsEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
    if (index >= size_) {
        throw OutOfBoundsEx(index);
    }
    return data_[index];
}
```

```
class OutOfBoundsEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       std::cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      cout << v[13];
   } catch (OutOfBoundsEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class OutOfBoundsEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
```

```
class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
```

```
class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

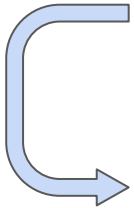
```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};
```

```
class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

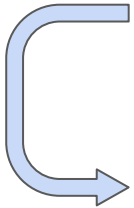
```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```



```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};


class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

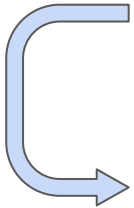


Substitution principle works here, so, we all derived exceptions from VectorEx will be catched.

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};

class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```



Substitution principle works here, so, we all derived exceptions from VectorEx will be catched.

This one is virtual, so different methods will be called

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};


class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. But will it be the same exception object there?

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. But will it be the same exception object there?

No, new object will be created! And of different type!

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. New VectorEx will be created with copy ctr,

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

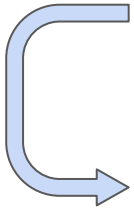
All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. New VectorEx will be created with copy ctr,
3. Which printDescription will be called?

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```



Substitution principle works here, so, we all derived exceptions from VectorEx will be catched.

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};

class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. New VectorEx will be created with copy ctr,
3. VectorEx::printDescription, not EmptyEx::printDescription

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx* exception) {
      exception->printDescription();
   }
   return 0;
}
```

```
int pop() {
   if (size_ == 0) {
       throw new EmptyEx();
   }
   return data_[--size_];
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx* exception) {
      exception->printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw new EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx* exception) {
      exception->printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw new EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx* exception) {
      exception->printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw new EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,
3. But now it is your responsibility to delete the exception object!!

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx* exception) {
      exception->printDescription();
      delete exceptiopn;
   }
   return 0;
}
```

```
int pop() {
   if (size_ == 0) {
       throw new EmptyEx();
   }
   return data_[--size_];
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,
3. But now it is your responsibility to delete the exception object!!

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,
3. But what about memory, what is lifetime of the exception object?

Object lifetime (third approximation)

When object dies?

If object is temporary =>

- if it is bound to some reference => lifetime extended to this reference;

- otherwise, end of the full statement;

Otherwise, depends on its storage duration:

- static => when program terminates
- automatic => at the end of the scope
- dynamic => when delete is called

Object lifetime (third approximation)

So, where is our exception object?

When object dies?

If object is temporary =>

- if it is bound to some reference => lifetime extended to this reference;

- otherwise, end of the full statement;

Otherwise, depends on its storage duration:

- static => when program terminates
- automatic => at the end of the scope
- dynamic => when delete is called

Object lifetime (third approximation)

So, where is our exception object?

When object dies? In some unspecified memory.

If object is temporary =>
- if it is bound to some reference => lifetime extended to this reference;
- otherwise, end of the full statement;

Otherwise, depends on its storage duration:
- static => when program terminates
- automatic => at the end of the scope
- dynamic => when delete is called

Object lifetime (third approximation)

So, where is our exception object?

When object dies?

In some unspecified memory. It is guaranteed to be alive during the catch block where it is used.

If object is temporary =>

- if it is bound to some reference => lifetime extended to this reference;

- otherwise, end of the full statement;

Otherwise, depends on its storage duration:

- static => when program terminates
- automatic => at the end of the scope
- dynamic => when delete is called

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

How should we throw and catch exceptions?

By value?
By pointer?
By reference?

All options are possible, but let's discuss them.

What will happen?

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

1. Well, this will work, catch block will be reached,
2. No copying and virtual methods work,
3. No problems with memory, the object will be deleted automatically after the catch.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

How should we throw and catch exceptions?

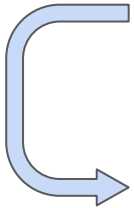
By value?
By pointer?
By reference? ✓

Usually catching by reference is the best option among all.

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   } catch (EmptyEx& exception) {
      cout << "empty one" << endl;
      exception.printDescription();
   }

   return 0;
}
```

```
int pop() {
   if (size_ == 0) {
       throw EmptyEx();
   }
   return data_[--size_];
}
```

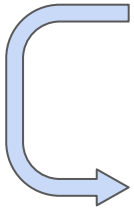
```
int main() {
    Vector v;
    v.push(42);
    try {
        v.pop();
        v.pop(); // <-- throws EmptyEx
        cout << v[13];
    } catch (VectorEx& exception) {
        exception.printDescription();
    } catch (EmptyEx& exception) {
        cout << "empty one" << endl;
        exception.printDescription();
    }

    return 0;
}
```



```
int pop() {
    if (size_ == 0) {
        throw EmptyEx();
    }
    return data_[--size_];
}
```

Both types are suitable, so, only the first catch block will be reached.

```
int main() {
    Vector v;
    v.push(42);
    try {
        v.pop();
        v.pop(); // <-- throws EmptyEx
        cout << v[13];
    } catch (EmptyEx& exception) {
        cout << "empty one" << endl;
        exception.printDescription();
    } catch (VectorEx& exception) {
        exception.printDescription();
    }

    return 0;
}
```



```
int pop() {
    if (size_ == 0) {
        throw EmptyEx();
    }
    return data_[--size_];
}
```

Both types are suitable, so, only the first catch block will be reached.

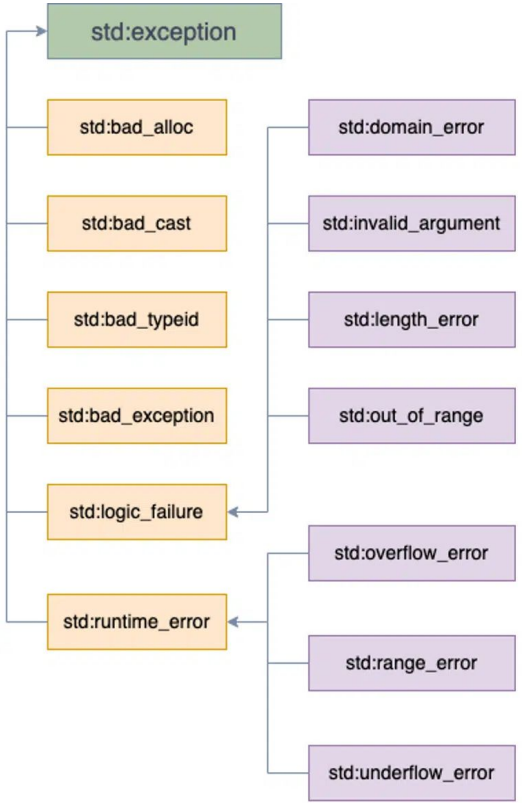
So, you place more specific handlers before more generic.

Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions

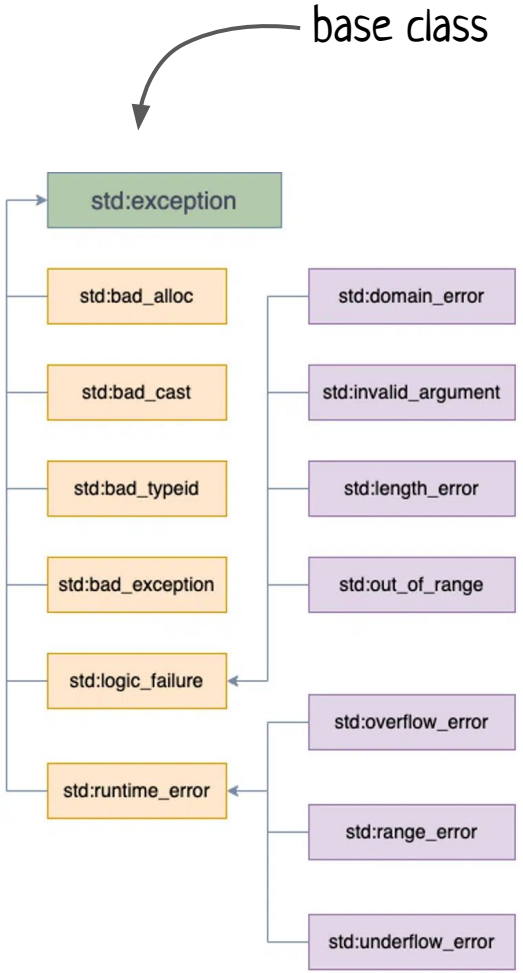
Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)



Exceptions: standard exceptions

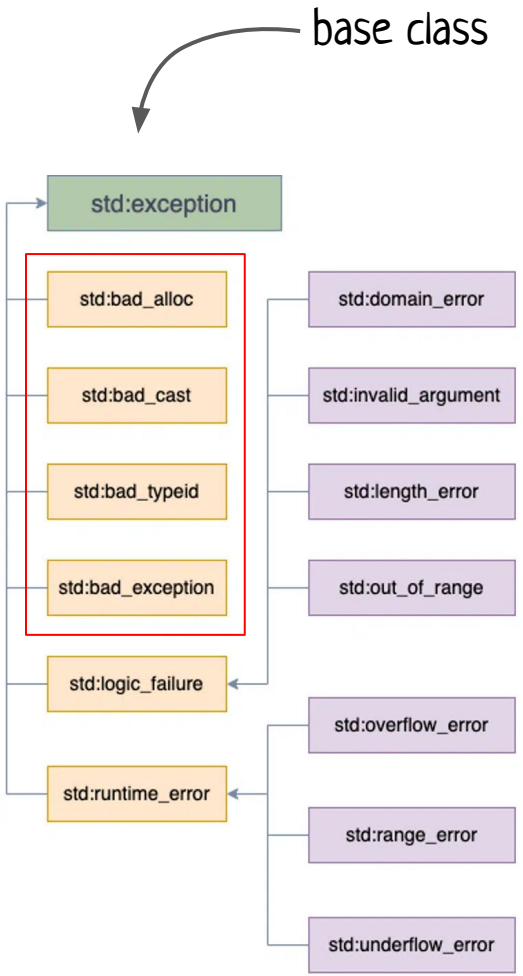
C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)



Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)

internal C++ runtime stuff: most probably
you shouldn't throw such exceptions
(but you can and should catch them!)

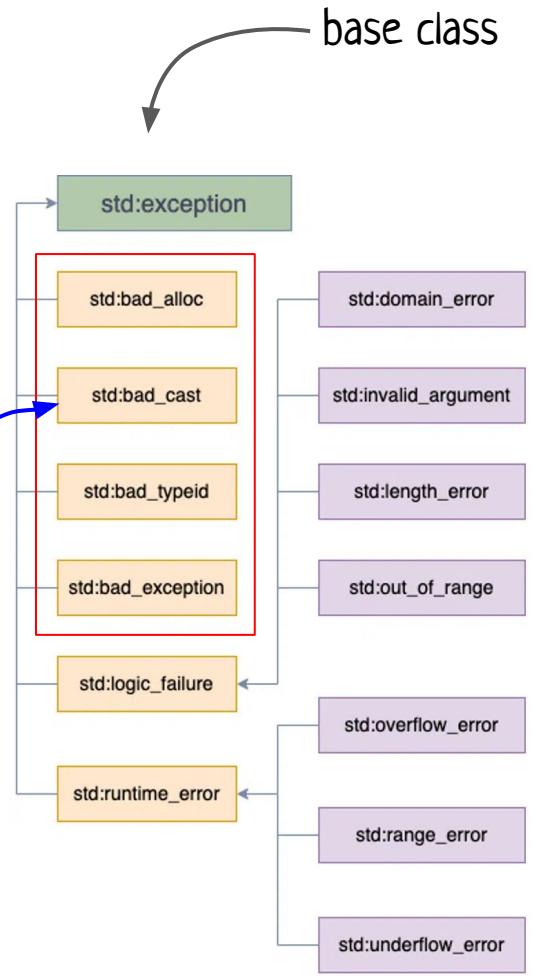


Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)

internal C++ runtime stuff: most probably
you shouldn't throw such exceptions
(but you can and should catch them!)

any idea who is this guy?



```
void check_if_student(Person* p) {
    p->print();

    Student* st = dynamic_cast<Student*>(p);
    if (st == nullptr) {
        std::cout << "cast failed"
                  << std::endl;
    } else {
        std::cout << "successfully casted"
                  << std::endl;
    }
}
```

```
void check_if_student(Person& p) {
    p.print();

    Student& st = dynamic_cast<Student&>(p);
    // how should we understand whether cast failed or not???
}
```

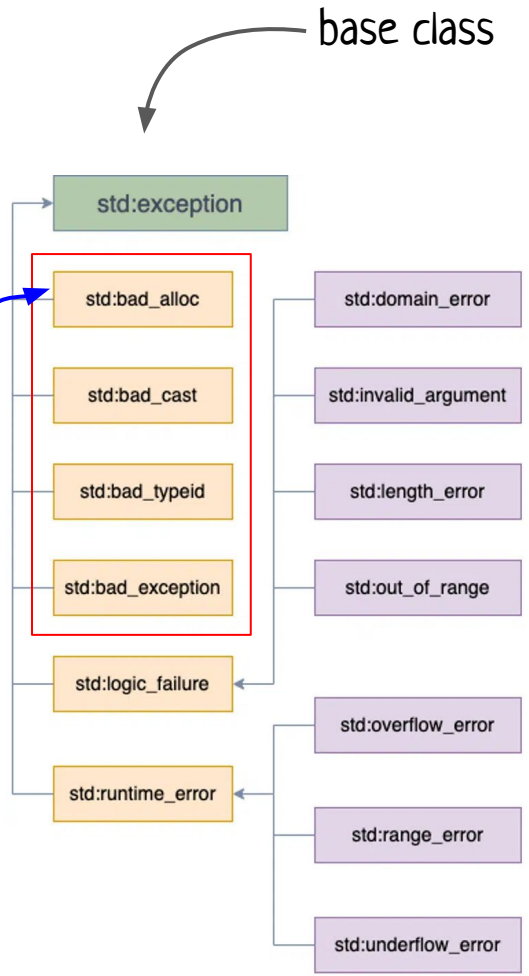
```
void check_if_student(Person& p) {
    p.print();

    try {
        Student &st = dynamic_cast<Student &>(p);
    } catch (std::bad_cast& ex) {
        std::cout << "it wasn't a student" << std::endl;
    }
}
```

Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)

internal C++ runtime stuff: most probably
you shouldn't throw such exceptions
(but you can and should catch them!)

any idea who is this guy?



```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```

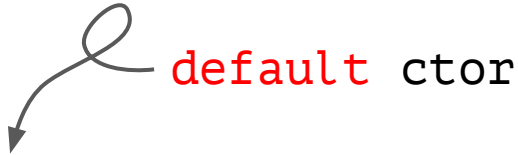
default ctor

New operator is C++ way to create objects in dynamic memory.

New operator:

- allocates memory,
- checks the result (throws a special exception on failure)
- initialize an object via calling ctor
- returns pointer to the initialized object

```
class Vector {
    ...
public:
    Vector(): Vector(16) { }

    Vector(size_t initial_capacity) {
        size_ = 0;
        capacity_ = initial_capacity;
        data_ = new int[capacity_];
    }
    ...
};
```



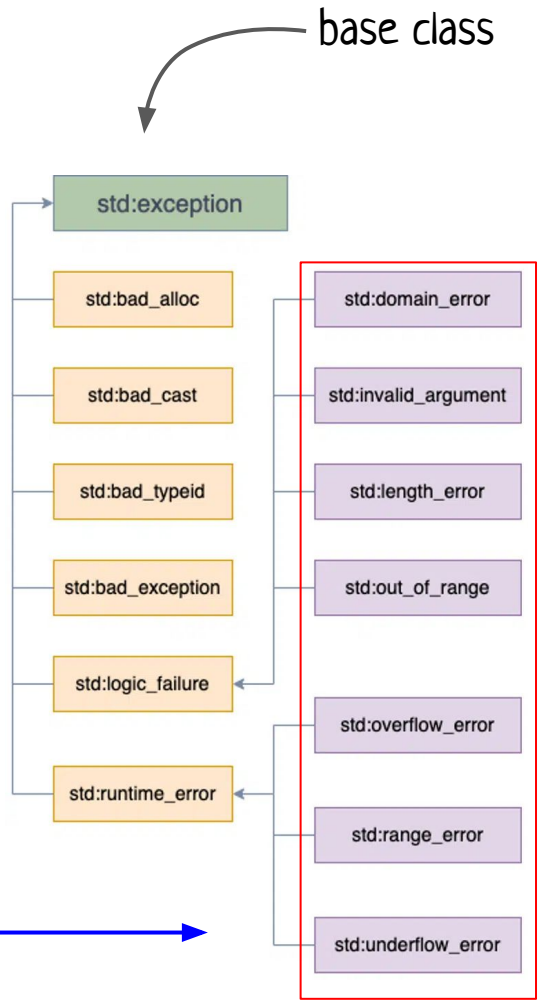
New operator is C++ way to create objects in dynamic memory.

New operator:

- allocates memory,
- checks the result (throws std::bad_alloc on failure)
- initialize an object via calling ctor
- returns pointer to the initialized object

Exceptions: standard exceptions

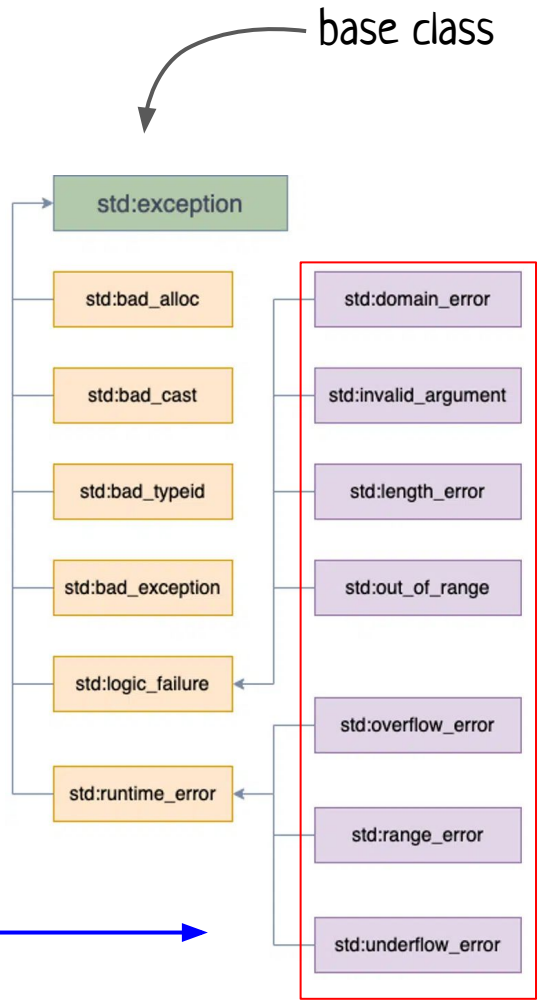
C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)



used in C++ std library, but you are very welcome to use it as well

Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)



used in C++ std library, but you are very
welcome to use it as well (and inherit from it!)

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class VectorEx {
public:
   virtual void printDescription() {
       cout << "Some problem with vector";
   }
};
class EmptyEx: public VectorEx {
public:
   void printDescription() {
      cout << "Vector is empty;
   }
};
```

```
class OutOfBoundsEx: public VectorEx {
   size_t index;
public:
   OutOfBoundsEx(size_t i): index(i) { }

   void printDescription() {
       cout
       << "Out of bounds with index: "
       << index;
   }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class OutOfBoundsEx: public std::exception {
    ...
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class exception {
public:
    exception(const exception&) = default;
    exception& operator=(const exception&) =
                                          default;
    exception(exception&&) = default;
    exception& operator=(exception&&) = default;

    virtual const char* what() const noexcept;

};
```

```
class OutOfBoundsEx: public std::exception {
    ...
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class exception {
public:
    exception(const exception&) = default;
    exception& operator=(const exception&) =
                                          default;
    exception(exception&&) = default;
    exception& operator=(exception&&) = default;

    virtual const char* what() const noexcept;

};
```

returns the description of an error

```
class OutOfBoundsEx: public std::exception {
    ...
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
     v.pop();
     v.pop(); // <-- throws EmptyEx
     cout << v[13];
   } catch (VectorEx& exception) {
      exception.printDescription();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class exception {
public:
    exception(const exception&) = default;
    exception& operator=(const exception&) =
                                        default;
    exception(exception&&) = default;
    exception& operator=(exception&&) = default;

    virtual const char* what() const noexcept;

};
```

returns the description of an error

ignore for now

```
class OutOfBoundsEx: public std::exception {
    ...
};
```

```cpp
int main() {
    Vector v;
    v.push(42);
    try {
        v.pop();
        v.pop(); // <-- throws EmptyEx
        cout << v[13];
    } catch (VectorEx& exception) {
        exception.printDescription();
    }
    return 0;
}
```

```cpp
int& operator[](size_t index) {
    if (index >= size_) {
        throw OutOfBoundsEx(index);
    }
    return data_[index];
}
```

```cpp
class exception {
public:
    exception(const exception&) = default;
    exception& operator=(const exception&) =
                                        default;
    exception(exception&&) = default;
    exception& operator=(exception&&) = default;

    virtual const char* what() const noexcept;

};
```

returns the description of an error

ignore for now

```cpp
class OutOfBoundsEx: public std::exception {
    std::string msg_;
public:
    OutOfBoundsEx(size_t i) {
        msg_ = "Out of bounds with index: ";
        msg_ += std::to_string(i);
    }
    const char* what() const noexcept {
        return msg_.c_str();
    }
};
```

```
int main() {
   Vector v;
   v.push(42);
   try {
      v.pop();
      v.pop(); // <-- throws EmptyEx
      cout << v[13];
   } catch (std::exception& e) {
      std::cout << e.what();
   }
   return 0;
}
```

```
int& operator[](size_t index) {
   if (index >= size_) {
       throw OutOfBoundsEx(index);
   }
   return data_[index];
}
```

```
class exception {
public:
    exception(const exception&) = default;
    exception& operator=(const exception&) =
                                        default;
    exception(exception&&) = default;
    exception& operator=(exception&&) = default;

    virtual const char* what() const noexcept;
};
```

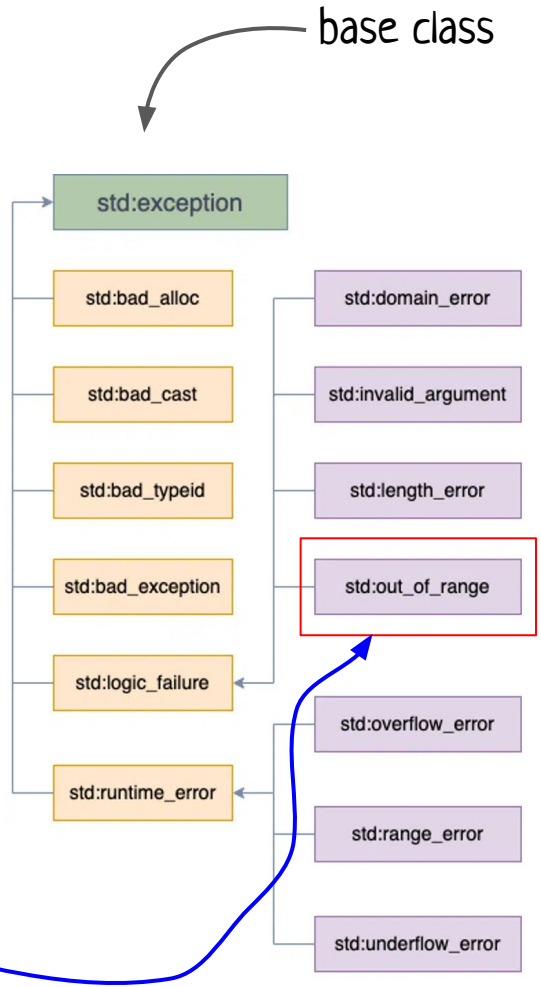
returns the description of an error

ignore for now

```
class OutOfBoundsEx: public std::exception {
   std::string msg_;
public:
   OutOfBoundsEx(size_t i) {
       msg_ = "Out of bounds with index: ";
       msg_ += std::to_string(i);
   }
   const char* what() const noexcept {
       return msg_.c_str();
   }
};
```

Exceptions: standard exceptions

C++ standard library already
has nice hierarchy of exceptions
(this is not the full hierarchy)

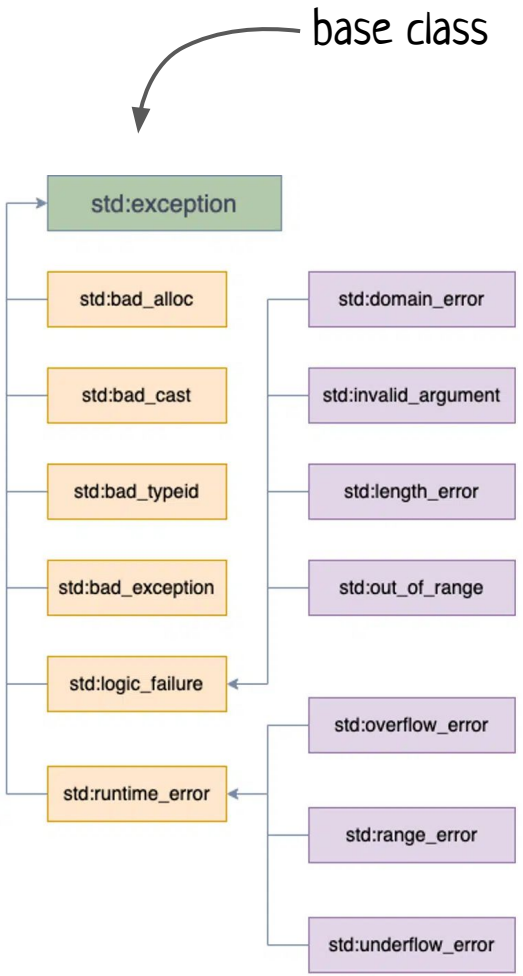


Or, you can just take this one and that's it!
(used in std::vector for checked "at" method)

Exceptions: standard exceptions

C++ standard library already has nice hierarchy of exceptions (this is not the full hierarchy)

Inheriting your exception from `std::exception` (or, more precisely from one of its derived classes) is a very good practice and the right way of adding new exceptions.

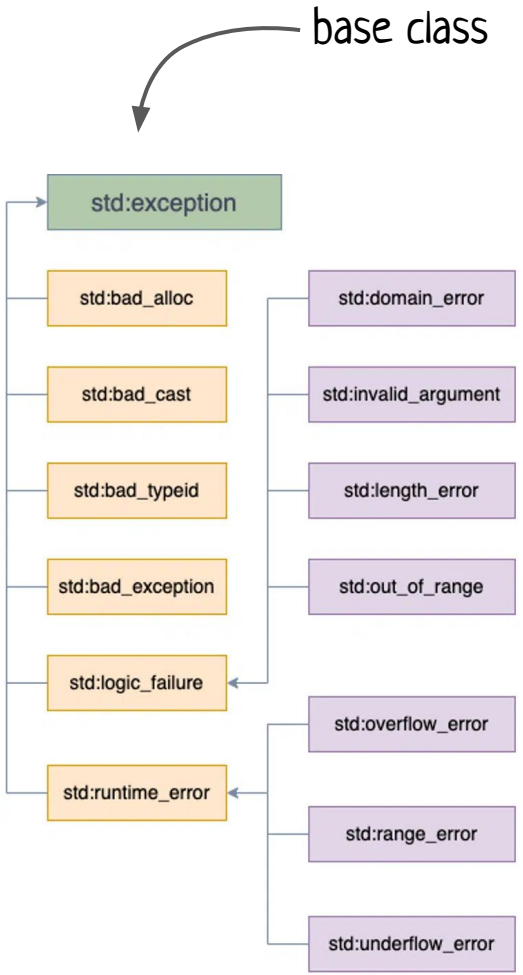


Exceptions: standard exceptions

C++ standard library already has nice hierarchy of exceptions (this is not the full hierarchy)

Inheriting your exception from std::exception (or, more precisely from one of its derived classes) is a very good practice and the right way of adding new exceptions.

It allows users of your library to use it without any investigation, in polymorphic way.



Exceptions: rules so far

- Catch exceptions by reference
- Most specific catch block comes first
- Inherit from std::exception

Exceptions: additional syntax

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
}
```

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
}
```

You can catch anything (that was not already caught) with `catch(...)`

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
}
```

You can catch anything (that was not already caught) with `catch(...)`, but why? Looks dangerous, you can break some internal mechanisms, exceptions from libraries and etc.

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
}
```

One of the reasonable approaches is to free some resources and rethrow the same exception further.

You can catch anything (that was not already caught) with catch(...), but why? Looks dangerous, you can break some internal mechanisms, exceptions from libraries and etc.

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
   throw;
}
```

One of the reasonable approaches is to free some resources and rethrow the same exception further.

You can catch anything (that was not already caught) with catch(...), but why? Looks dangerous, you can break some internal mechanisms, exceptions from libraries and etc.

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
   throw;
}
```

`throw` (without arguments is used to get current exception object and just throw it further.

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
   throw;
}
```

`throw` (without arguments is used to get current exception object and just throw it further. Can be used in any `catch` block (using out of it will crash your execution).

Exceptions: additional syntax

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
   throw;
}
```

`throw` (without arguments is used to get current exception object and just throw it further. Can be used in any `catch` block (using out of it will crash your execution).

Exceptions: additional syntax

the lifetime of exception object is prolonged till the next catch

```
Vector v;
v.push(42);
try {
   std::cout << v[13] << std::endl;
} catch (VectorEx& ex) {
   ex.printDescription();
} catch(...) {
   std::cout << "Well, something is wrong" << std::endl;
   throw;
}
```

throw (without arguments is used to get current exception object and just throw it further. Can be used in any catch block (using out of it will crash your execution).

Exceptions: additional syntax #2

Exceptions: additional syntax #2

```
void foo_(int arg) try {
   // do something dangerous
} catch (std::exception& ex) {
   std::cout << "foo was called with "<< arg
             << "argument, but: " << ex.what() << std::endl;
}
```

Exceptions: function-try-block

the whole body of a function is wrapped with try/catch

```
void foo_(int arg) try {
   // do something dangerous
} catch (std::exception& ex) {
   std::cout << "foo was called with "<< arg
             << "argument, but: " << ex.what() << std::endl;
}
```

Exceptions: function-try-block

the whole body of a function is wrapped with try/catch

```
void foo_(int arg) try {
   // do something dangerous
} catch (std::exception& ex) {
   std::cout << "foo was called with "<< arg
             << "argument, but: " << ex.what() << std::endl;
}
```

arg is still accessible here!

Exceptions: function-try-block

```
class Foo_ {
   Vector v;
public:
   Foo_(int s) try: v(s) {
       // some initialization
   } catch (std::exception& ex) {
       // handling exceptions including those
       // from member initialization list!
   }
};
```

Exceptions: function-try-block

```
class Foo_ {
   Vector v;
public:
   Foo_(int s) try: v(s) {
       // some initialization
   } catch (std::exception& ex) {
       // handling exceptions including those
       // from member initialization list!
   }
};
```

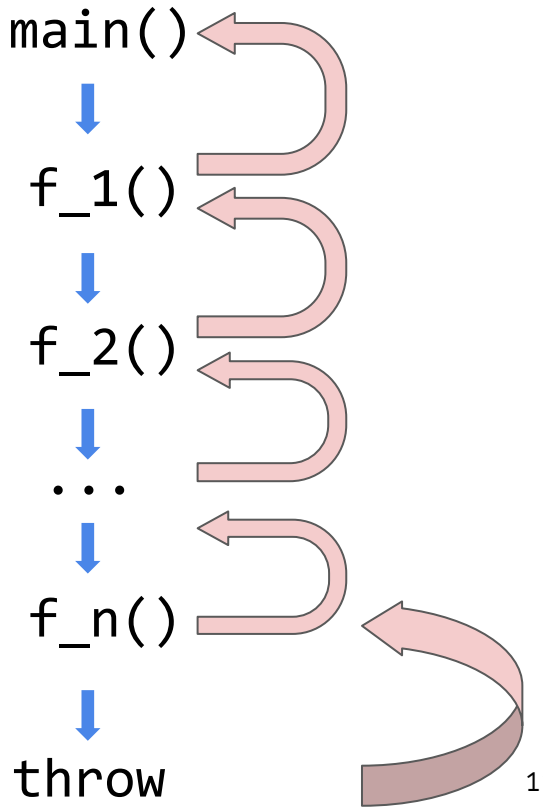
Works well with constructors. And basically this is the only way of handling exceptions from member initialization lists!

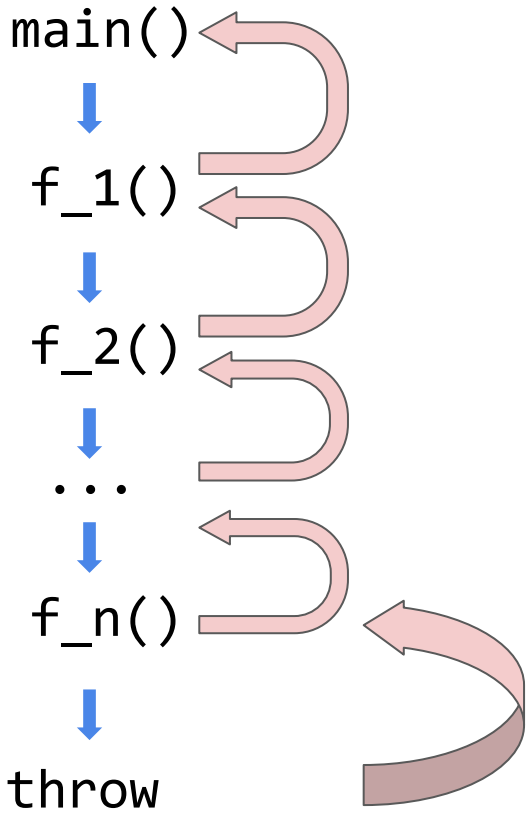
Exceptions: stack unwinding

Exceptions: basic syntax and mechanics

Interrupts the execution till it finds corresponding handler

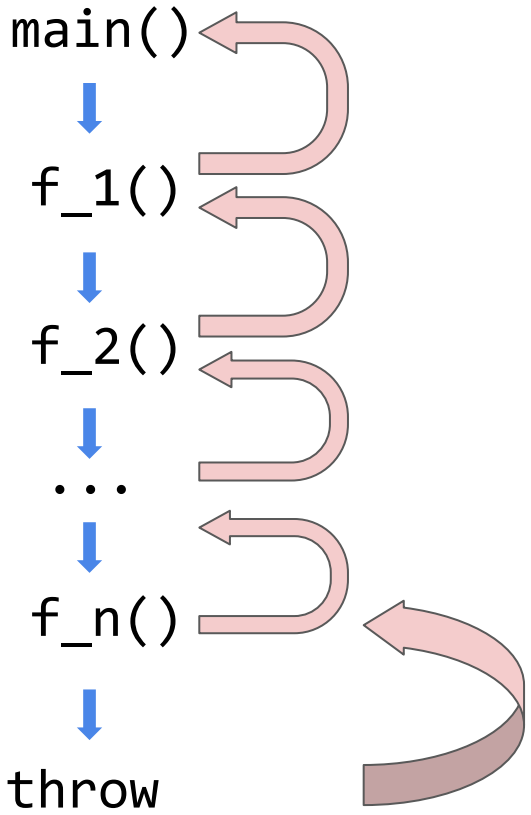
```
int main() {
   Vector v;
   v.push(42);
   try {
      std::cout <<getValue(v, 13);
   } catch (const char* exception) {
      std::cout << "Something went wrong: "
                << exception << std::endl;
   }
   return 0;
}
```





Interrupts the execution till it finds corresponding handler.

But what about local objects, placed on the stack of those interrupted methods?



Interrupts the execution till it finds corresponding handler.

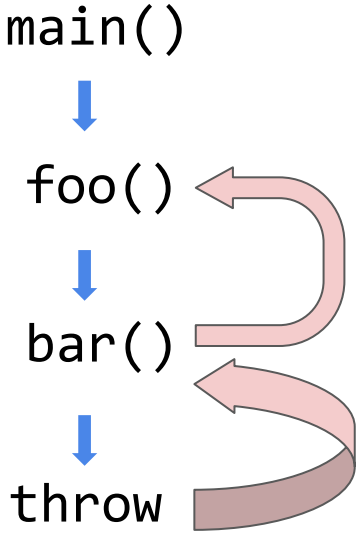
But what about local objects, placed on the stack of those interrupted methods?

They should be freed as well, their destructors must be called.
We base all our RAII approach on that.

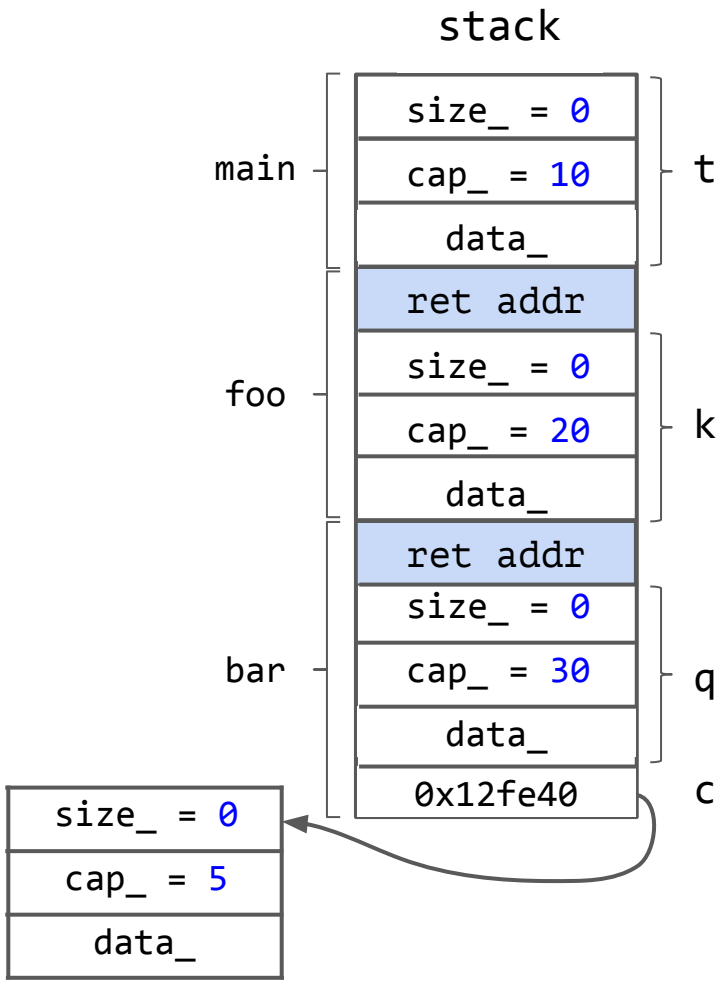
```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

    Vector::~Vector() {
        cout << "destructing vector of size " << size_ << endl;
        delete[] data_;
    }
};
```

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

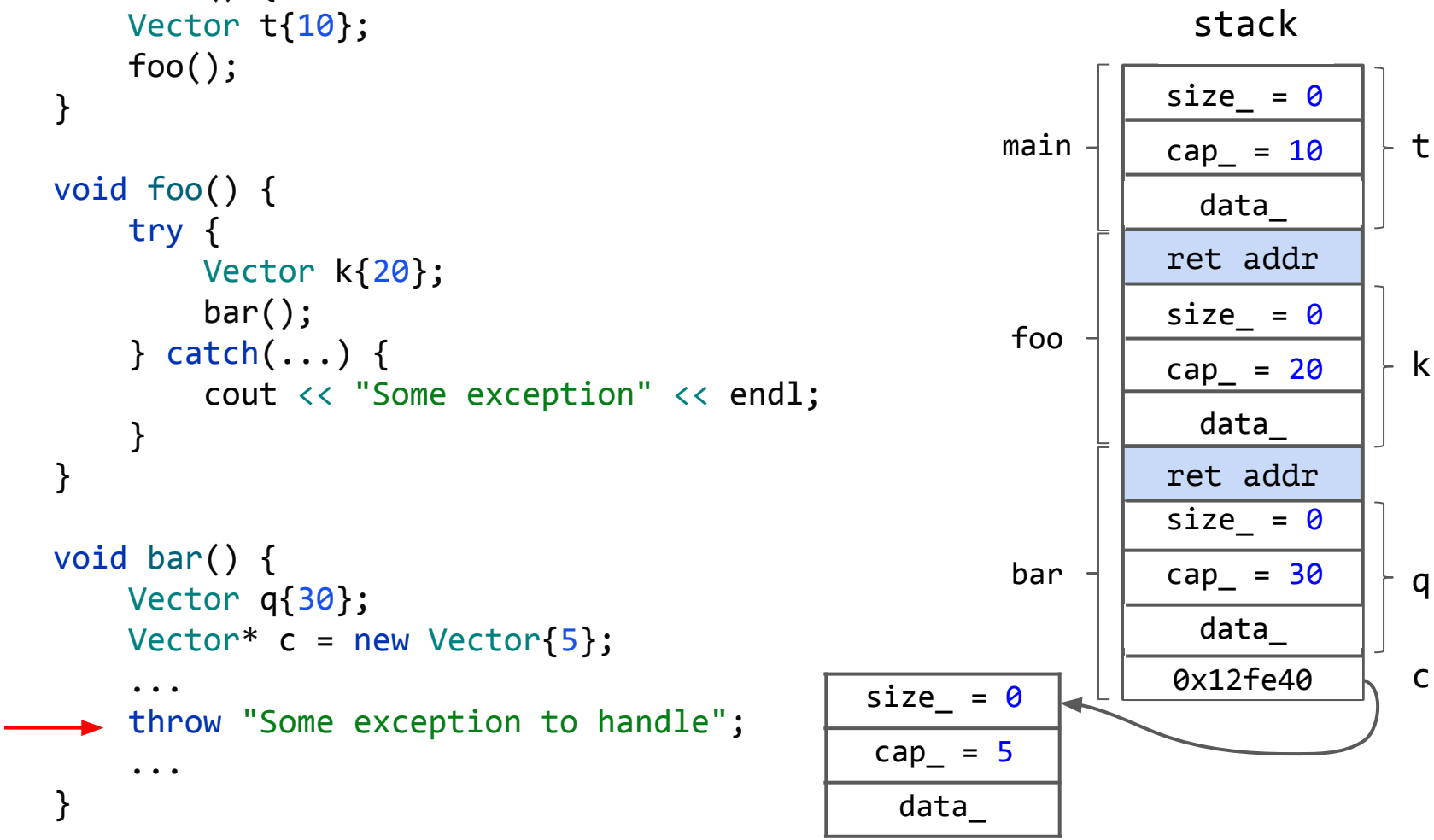
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



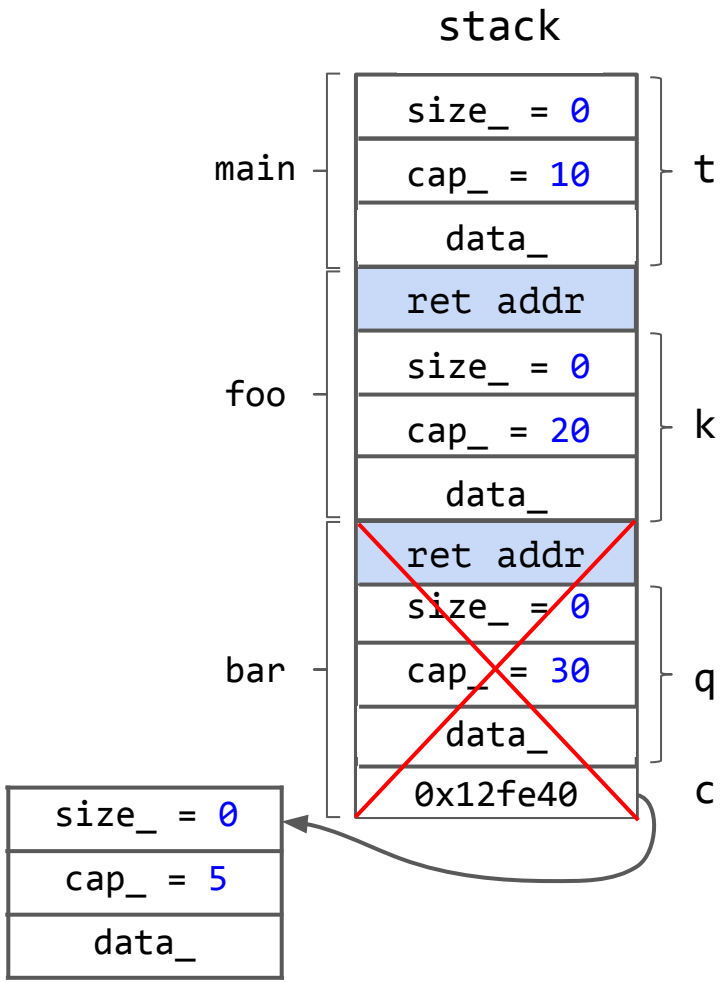
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



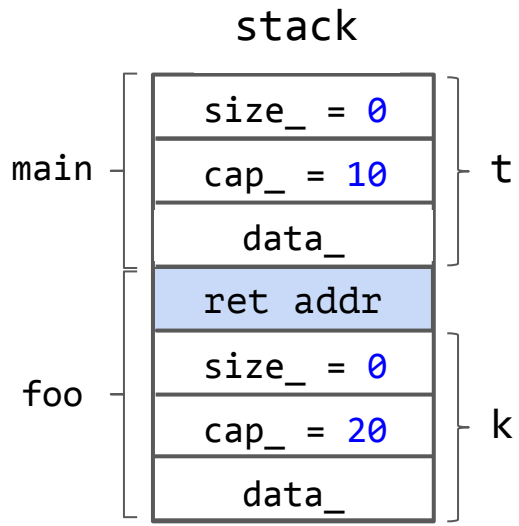
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



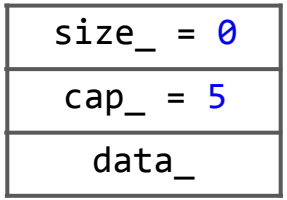
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



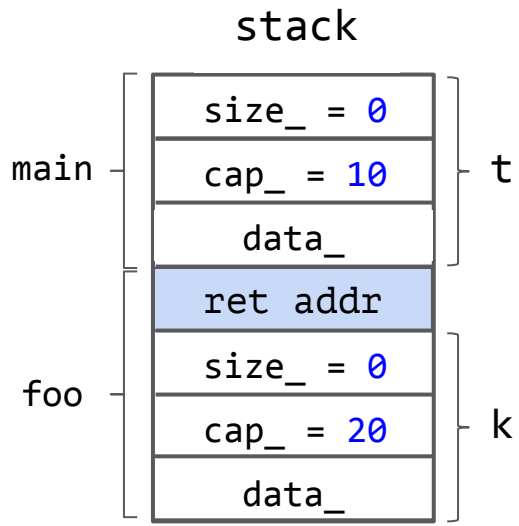
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



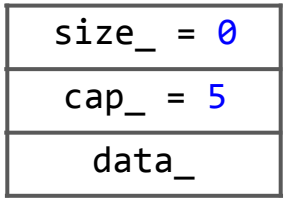
destructing vector of size 30



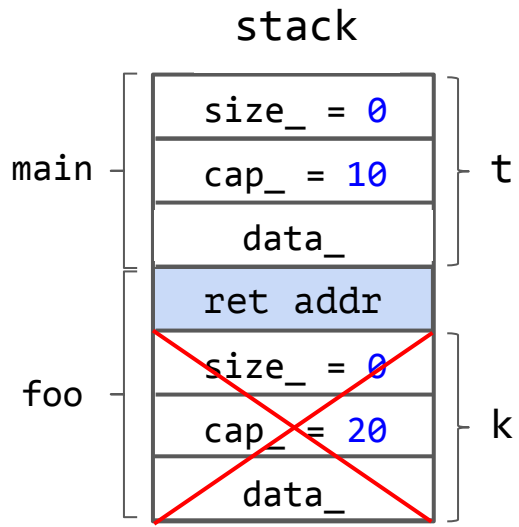
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



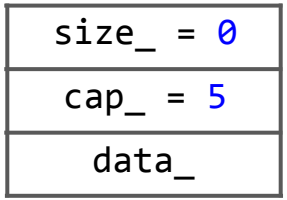
destructing vector of size 30



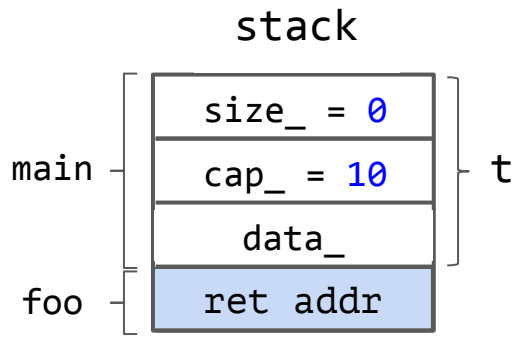
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



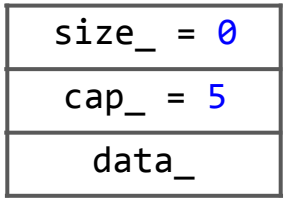
destructing vector of size 30



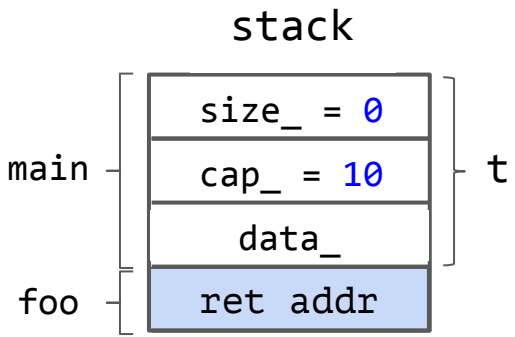
```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



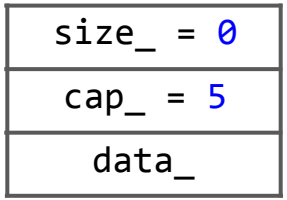

```
destructing vector of size 30
destructing vector of size 20
```

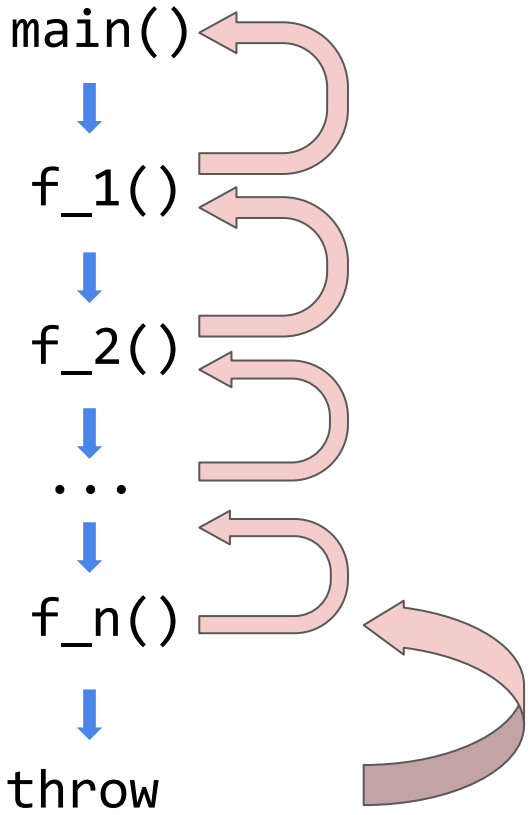


```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20}; ✓
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30}; ✓
    Vector* c = new Vector{5}; ✗
    ...
    throw "Some exception to handle";
    ...
}
```




```
destructing vector of size 30
destructing vector of size 20
```



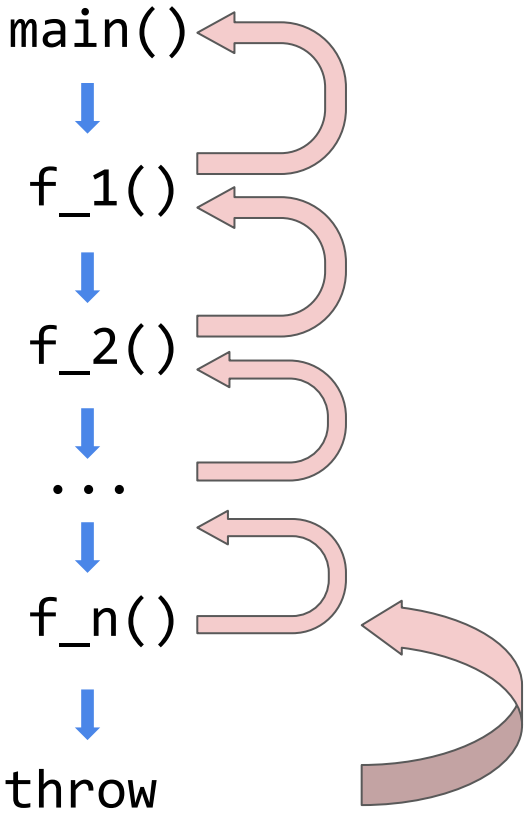


Interrupts the execution till it finds corresponding handler.

But what about local objects, placed on the stack of those interrupted methods?

They should be freed as well, their destructors must be called. We base all our RAII approach on that.

This process is called stack unwinding. And looks like it works as expected!



Interrupts the execution till it finds corresponding handler.

But what about local objects, placed on the stack of those interrupted methods?

They should be freed as well, their destructors must be called. We base all our RAII approach on that.

This process is called stack unwinding. And looks like it works as expected! Right?

Exceptions: stack unwinding (problems)

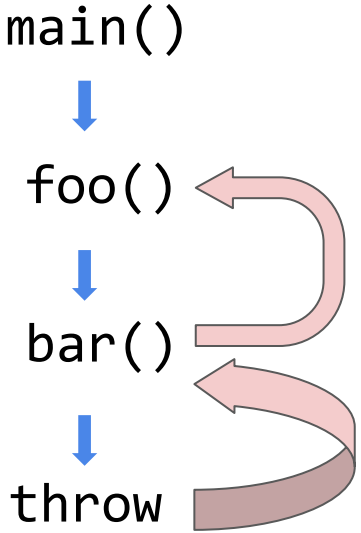
Exceptions: stack unwinding (problems)

So… destructors of local objects are called during unwinding.

But what if destructor itself throws an exception?

```
class Vector {
    size_t size_ = 0;
    size_t capacity_ ;
    int* data_ ;
public:
    Vector(size_t ic = 16) { … }
    ~Vector() { … }

    int pop() { return data_[--size_]; }
    int& operator[](size_t idx) { return data_[idx]; }

    Vector::~Vector() {
        cout << "destructing vector of size " << size_ << endl;
        delete[] data_;
        throw std::range_error("ooops");
    }
};
```

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```



```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

```
destructing vector of size 30
```

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

```
destructing vector of size 30

terminate called after throwing an
instance of 'std::range_error'
  what():  ooops
```

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

```
destructing vector of size 30

terminate called after throwing an
instance of 'std::range_error'
  what():  ooops
```

Yes, you can throw only one exception at the same time.

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

```
destructing vector of size 30

terminate called after throwing an
instance of 'std::range_error'
  what():  ooops
```

Yes, you can throw only one exception at the same time.

So! Destructors shouldn't throw an exception.

```
int main() {
    Vector t{10};
    foo();
}

void foo() {
    try {
        Vector k{20};
        bar();
    } catch(...) {
        cout << "Some exception" << endl;
    }
}

void bar() {
    Vector q{30};
    Vector* c = new Vector{5};
    ...
    throw "Some exception to handle";
    ...
}
```

```
destructing vector of size 30

terminate called after throwing an
instance of 'std::range_error'
  what():  ooops
```

Yes, you can throw only one exception at the same time.

So! Destructors shouldn't throw an exception.

Or, to be more precise: exceptions shouldn't leave destructors.

Exceptions: stack unwinding (problems)

Ok, it is clear with destructors.

Exceptions: stack unwinding (problems)

Ok, it is clear with destructors.

But can I safely throw exceptions from constructors?

Exceptions: stack unwinding (problems)

Ok, it is clear with destructors.

But can I safely throw exceptions from constructors?

Well…

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
public:
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
    }

    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
};
```

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
    }

    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
};
```

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
};
```

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

What will be printed?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
```

What will be printed?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
```

What will be printed?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
destructing of cap 16
destructing of cap 16
```

What will be printed?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
destructing of cap 16
destructing of cap 16
oops
```

What will be printed?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
destructing of cap 16
destructing of cap 16
oops
```

What will be printed?

Where is the third destructor??

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
destructing of cap 16
destructing of cap 16
oops
```

What will be printed?

Where is the third destructor??

The third vector was not yet initialized, so, no destructor for it...

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
    static int number_of_vectros;
public:
    Vector(): Vector(16) {};
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
        number_of_vectros++;
        if (number_of_vectros > 2) throw "too many";
    }
    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
```

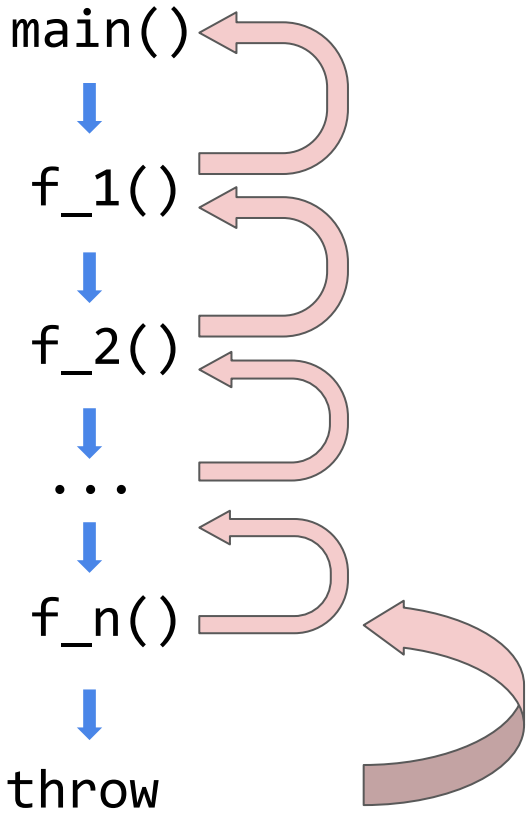
```
try {
    Vector arrays[10];
} catch(...) {
   cout << "oops" << endl;
}
```

```
constructing of cap 16
constructing of cap 16
constructing of cap 16
destructing of cap 16
destructing of cap 16
oops
```

What will be printed?

Where is the third destructor??

The third vector was not yet initialized, so, no destructor for it... memory leak is here.

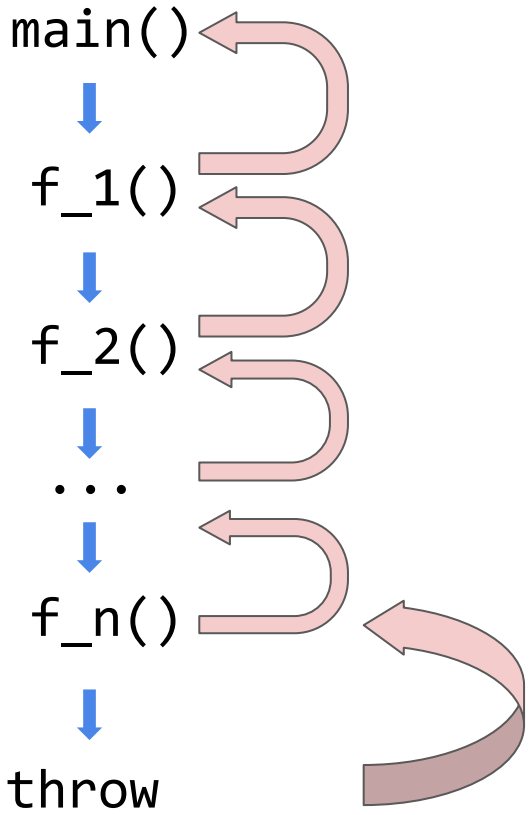


Interrupts the execution till it finds corresponding handler.

But what about local objects, placed on the stack of those interrupted methods?

They should be freed as well, their destructors must be called. We base all our RAII approach on that.

This process is called stack unwinding. And looks like it works as expected!



Interrupts the execution till it finds corresponding handler.

But what about local objects, placed on the stack of those interrupted methods?

They should be freed as well, their destructors must be called. We base all our RAII approach on that.

This process is called stack unwinding. And looks like it works as expected! (for objects which construction is finished).

Exceptions: stack unwinding (problems)

Ok, it is clear with destructors.

But can I safely throw exceptions from constructors?

Yes, but you should check, that no resources were allocated before it. Otherwise, there will be memory leaks

Exceptions: stack unwinding (problems)

Ok, it is clear with destructors.

But can I safely throw exceptions from constructors?

Yes, but you should check, that no resources were allocated before it. Otherwise, there will be memory leaks

Ok, but no more problems except constructors and destructors, right? Right?

```
class Vector {
    int* data_;
    size_t size_ = 0;
    size_t cap_;
public:
    Vector(size_t cap): cap_(cap) {
        cout << "constructing of cap " << cap_ << endl;
        data_ = new int[capacity_];
    }

    ~Vector() {
        cout << "destructing of cap " << cap_ << endl;
        delete[] data_;
    }
};
```

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

```
constructing of cap 20
constructing of cap 10
```

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

```
constructing of cap 20
constructing of cap 10
terminate called after throwing an instance of 'char const*'
```

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

```
constructing of cap 20
constructing of cap 10
terminate called after throwing an instance of 'char const*'
```

WTF?? Where are my destructors calls?

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

Well, it is guaranteed by the specification that destructors are called on the way from throw to catch.

```
constructing of cap 20
constructing of cap 10
terminate called after throwing an instance of 'char const*'
```

WTF?? Where are my destructors calls?

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

Well, it is guaranteed by the specification that destructors are called on the way from throw to catch.

If there were no catch => it is implementation defined whether destructors will be called or not.

```
constructing of cap 20
constructing of cap 10
terminate called after throwing an instance of 'char const*'
```

WTF?? Where are my destructors calls?

```
void foo() {
    Vector k{20};
    throw "Some exception not to handle";
}


int main() {
    Vector t{10};
    foo();
}
```

```
constructing of cap 20
constructing of cap 10
terminate called after throwing an instance of
```

WTF?? Where are my destructors calls?

Well, it is guaranteed by the specification that destructors are called on the way from throw to catch.

If there were no catch => it is implementation defined whether destructors will be called or not.

So, depends on the compiler!

Exceptions: stack unwinding (problems)

1) Do not throw an exception from the destructor,

Exceptions: stack unwinding (problems)

1) Do not throw an exception from the destructor,

2) Do not throw an exception from the constructor when some resources are already allocated (there will be no destructor for such object),

Exceptions: stack unwinding (problems)

1) Do not throw an exception from the destructor,

2) Do not throw an exception from the constructor when some resources are already allocated (there will be no destructor for such object),

3) Keep in mind, that stack unwinding (with all destructors calls) does only work when exception is caught.

Exceptions: stack unwinding (problems)

1) Do not throw an exception from the destructor,

2) Do not throw an exception from the constructor when some resources are already allocated (there will be no destructor for such object),

3) Keep in mind, that stack unwinding (with all destructors calls) does only work when exception is caught.

   So, if you really need to clear some resources there before the termination, do your best to catch it. (catch(...) to help).

Exceptions: safe exceptions philosophy

Exceptions: safe exceptions philosophy

```
Vector(size_t cap): cap_(cap) {
    cout << "constructing of cap " << cap_ << endl;
    data_ = new int[capacity_];
    number_of_vectros++;
    if (number_of_vectros > 2) throw "too many";
}
```

This constructor was bad and dangerous…

Exceptions: safe exceptions philosophy

```
Vector(size_t cap): cap_(cap) {
    cout << "constructing of cap " << cap_ << endl;
    data_ = new int[capacity_];
    number_of_vectros++;
    if (number_of_vectros > 2) throw "too many";
}
```

This constructor was bad and dangerous… because it throws after some resources are already allocated.

Exceptions: safe exceptions philosophy

```
Vector(size_t cap): cap_(cap) {
    cout << "constructing of cap " << cap_ << endl;
    data_ = new int[capacity_];
    number_of_vectros++;
    if (number_of_vectros > 2) throw "too many";
}
```

This constructor was bad and dangerous… because it throws after some resources are already allocated.

But isn't it the same for any instance method of a class?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = new int[capacity_];

    for (size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }

    return *this;
}
```

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = new int[capacity_];

    for (size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }

    return *this;
}
```

Do you see any problems here?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = new int[capacity_];

    for (size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }

    return *this;
}
```

Do you see any problems here?

what if std::bad_alloc is thrown here?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = new int[capacity_];

    for (size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }

    return *this;
}
```

Do you see any problems here?

what if std::bad_alloc is thrown here?

we will have an object in inconsistent state! as size_, capacity_ and data_ are already changed.

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = new int[capacity_];

    for (size_t i = 0; i < size_; i++) {
        data_[i] = other.data_[i];
    }

    return *this;
}
```

Do you see any problems here?

what if std::bad_alloc is thrown here?

we will have an object in inconsistent state! as size_, capacity_ and data_ are already changed.

how to fix?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    int* tmp = new int[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```

Now, no problems if allocation fails, the object is still correct.

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    int* tmp = new int[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```

Now, no problems if allocation fails, the object is still correct.

But do you see any more problems?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```

Now, no problems if allocation fails, the object is still correct.

But do you see any more problems?

Imagine that we have Vector of some generic types T.

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```

Now, no problems if allocation fails, the object is still correct.

But do you see any more problems?

Imagine that we have Vector of some generic types T.

In this case copying is also potentially dangerous!

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```


Now, no problems if allocation fails, the object is still correct.

But do you see any more problems?

Imagine that we have Vector of some generic types T.

In this case copying is also potentially dangerous! Copy assign operator for element can throw!

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    for (size_t i = 0; i < size_; i++) {
        tmp[i] = other.data_[i];
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;

    return *this;
}
```


Now, no problems if allocation fails, the object is still correct.

But do you see any more problems?

Imagine that we have Vector of some generic types T.

In this case copying is also potentially dangerous! Copy assign operator for element can throw! How to fix?

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

Well, it was basically not about inconsistent state of the object, but just about memory leak.

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }

    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

Well, it was basically not about inconsistent state of the object, but just about memory leak. Now fixed!

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }
    ---------------------------------------------
    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }
```

This code can fail (and throw exception). But it can't mutate the object.

```
    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }
```

This code can fail (and throw exception). But it can't mutate the object.

```
    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

This code can't fail!
No exceptions from here!

```
Vector& operator=(const Vector& other) {
    if (this == &other) {
        return *this;
    }

    T* tmp = new T[other.capacity_];
    try {
        for (size_t i = 0; i < size_; i++) {
            tmp[i] = other.data_[i];
        }
    } catch(...) {
        delete[] tmp;
        throw;
    }
```

This code can fail (and throw exception). But it can't mutate the object.

```
    size_ = other.size_;
    capacity_ = other.capacity_;

    delete[] data_;
    data_ = tmp;
    ...
```

This code can't fail!
No exceptions from here!
But it can mutate the object.

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

Can this code throw an exception?

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

Can this code throw an exception?
Depends on constructor and on foo()!

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    try {
        Vector k{20};
    } catch (...) {
        std::cout << "nothing bad happens here ;)" << std::endl;
    }
}
```

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

By this you are telling the compiler (and yourself): I swear, foo doesn't throw!

```
void foo() noexcept {
    try {
        Vector k{20};
    } catch (...) {
        std::cout << "nothing bad happens here ;)" << std::endl;
    }
}
```

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

By this you are telling the compiler (and yourself): I swear, foo doesn't throw!

Compiler can rely on it and generate code that can't throw an exception.

```
void foo() noexcept {
    try {
        Vector k{20};
    } catch (...) {
        std::cout << "nothing bad happens here ;)" << std::endl;
    }
}
```

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

By this you are telling the compiler (and yourself): I swear, foo doesn't throw!

Compiler can rely on it and generate code that can't throw an exception. Or it can ignore it completely. Up to him.

```
void foo() noexcept {
    try {
        Vector k{20};
    } catch (...) {
        std::cout << "nothing bad happens here ;)" << std::endl;
    }
}
```

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    throw std::exception("well, I'm a liar.")
}
```

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    throw std::exception("well, I'm a liar.")
}
```

You will most probably get a warning from IDE here.

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    throw std::exception("well, I'm a liar.")
}
```

You will most probably get a warning from IDE here.
But that's it, nobody stops you.

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    throw std::exception("well, I'm a liar.")
}
```

You will most probably get a warning from IDE here.
But that's it, nobody stops you.

But in runtime: `terminate called after throwing an instance of 'char const*'`

Exceptions: safe exceptions philosophy

But how can I check that some code doesn't throw an exception?

```
Vector t{10};
foo();
```

```
void foo() noexcept {
    throw std::exception("well, I'm a liar.")
}
```

You will most probably get a warning from IDE here.
But that's it, nobody stops you.

Well, maybe. Maybe not.

But in runtime: `terminate called after throwing an instance of 'char const*'`

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

`noexcept` can show your intention here.

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

noexcept can show your intention here.

You should respect that, otherwise there could (and most probably will) be a runtime crash.

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

noexcept can show your intention here.

You should respect that, otherwise there could (and most probably will) be a runtime crash.

But this is is not statically checked by the compiler!

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

noexcept can show your intention here.

You should respect that, otherwise there could (and most probably will) be a runtime crash.

But this is is not statically checked by the compiler!

Why? Why the hell it is not statically checked?

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

noexcept can show your intention here.

You should respect that, otherwise there could (and most probably will) be a runtime crash.

But this is is not statically checked by the compiler!

Why? Why the hell it is not statically checked?
Two reasons: backward compatibility and bad inference with move semantics feature.

Exceptions: *safe* exceptions philosophy

But how can I check that some code doesn't throw an exception?

noexcept can show your intention here.

You should respect that, otherwise there could (and most probably will) be a runtime crash.

But this is is not statically checked by the compiler!

Why? Why the hell it is not statically checked?
Two reasons: backward compatibility and bad inference with move semantics feature.

However, there is a way to statically check whether something can throw an exception or not. We will discuss it later.

Exceptions: usual disclaimers

Don’t forget that exceptions should be exceptional.

Exceptions: usual disclaimers

Don't forget that exceptions should be exceptional.

- They are expensive (when exception happens),
- They can lead to the spaghetty code (as it is actually a non-local jump),

Exceptions: usual disclaimers

Don't forget that exceptions should be exceptional.

- They are expensive (when exception happens),
- They can lead to the spaghetty code (as it is actually a non-local jump),
- There are C++ specific pitfalls as well.

Exceptions: usual disclaimers

Don't forget that exceptions should be exceptional.

- They are expensive (when exception happens),
- They can lead to the spaghetty code (as it is actually a non-local jump),
- There are C++ specific pitfalls as well.

Some companies just ban exceptions from their C++ code.

Exceptions: usual disclaimers

Don't forget that exceptions should be exceptional.

- They are expensive (when exception happens),
- They can lead to the spaghetty code (as it is actually a non-local jump),
- There are C++ specific pitfalls as well.

Some companies just ban exceptions from their C++ code. But in such case it would be really hard to implement (and use!) a library on C++.

Takeaways

- C++ exceptions syntax,
- General rules: catch by reference, most specific handler first, inherit from std::exception,

Takeaways

- C++ exceptions syntax,
- General rules: catch by reference, most specific handler first, inherit from std::exception,
- Stack unwinding and its pitfalls,
- Safe exceptions philosophy,

Takeaways

- C++ exceptions syntax,
- General rules: catch by reference, most specific handler first, inherit from std::exception,
- Stack unwinding and its pitfalls,
- Safe exceptions philosophy,
- noexcept and how to use it.