C++ Foundation

Control Flow, Iterators, and Exceptions

Control Flow, Iterators, and Exceptions

- throwing exceptions
- catching exceptions
- exceptions and object lifetime
- the RAII idiom
- predefined exception classes
- iterators
- iterator pairs to express a range
- iterator based algorithm examples

Puzzle

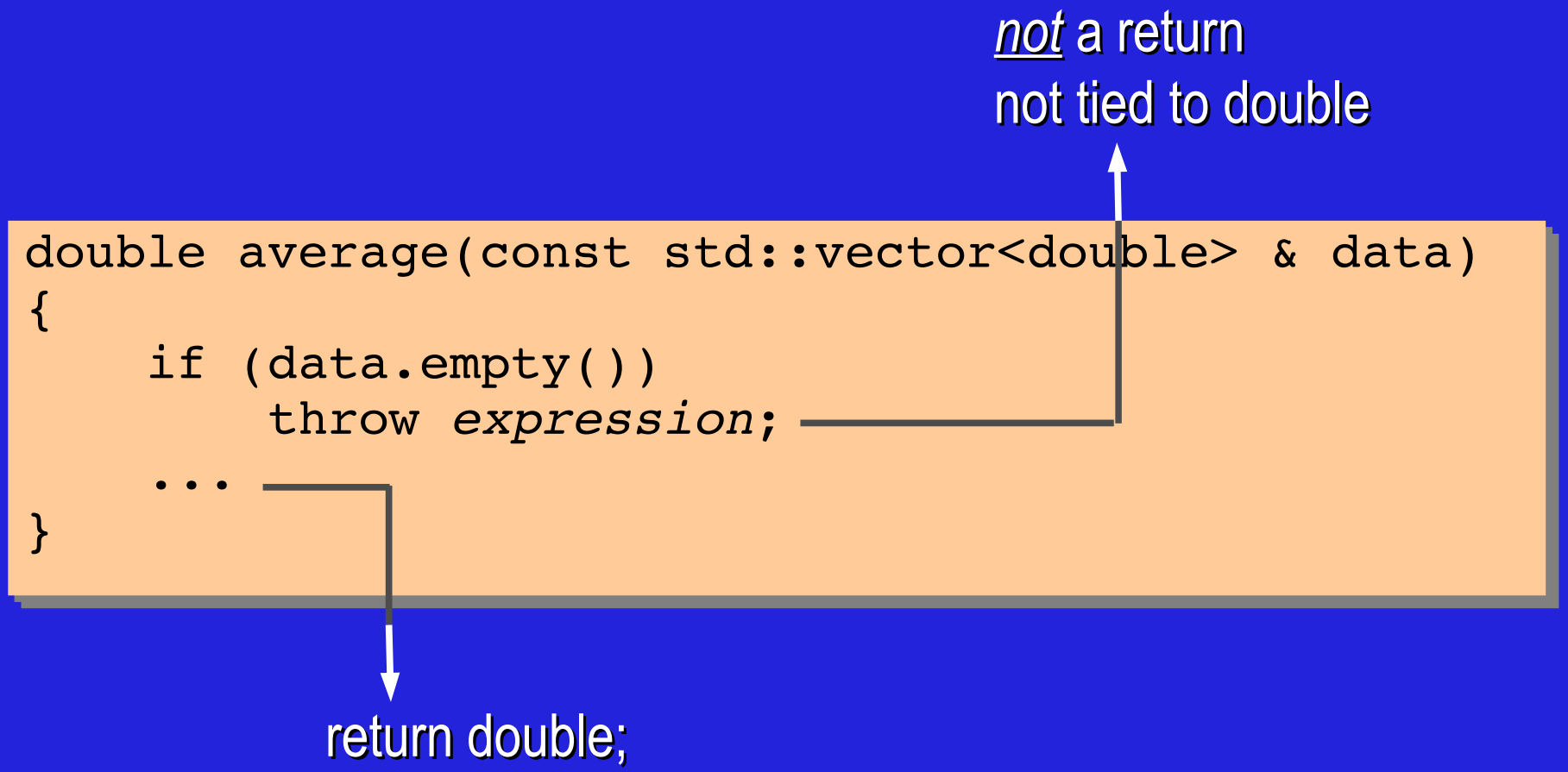
- Consider a function to find the average of a vector of doubles
- What should this function return if the vector is empty?

```
double average(const std::vector<double> & data);
```

```
void puzzle()
{
    std::vector<double> empty;
    assert(average(empty) == ???);
}
```

Introducing *throw*

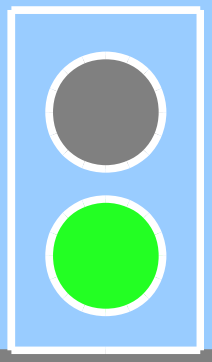
- Stops normal “forward” execution
- The program starts to unwind backwards!



Introducing *try* and *catch*

- This test passes if average throws any kind of exception

```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    bool caught = false;
    try
    {
        average(empty);
    }
    catch (...)
    {
        caught = true;
    }
    assert(caught);
}
```



test fails if no exception is thrown

catch-all the ... is part of the syntax and not ellision!

Refined test

```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    bool caught = false;
    try
    {
        average(empty);
    }
    catch (std::invalid_argument &)
    {
        caught = true;
    }
    catch (...)
    {
    }
    assert(caught);
}
```

test fails if no exception is thrown

test passes if invalid_argument is thrown

test fails if different exception is thrown

Standard Exception Classes

- Live in <stdexcept>

```
namespace std
{
    exception;
        bad_cast;
        bad_typeid;
        bad_alloc;
        bad_exception;
        logic_error;
            domain_error;
            invalid_argument;
            length_error;
            out_of_range;
        runtime_error;
            range_error;
            overflow_error;
            underflow_error;
        ...
}
```

thrown by dynamic_cast

thrown by typeid

thrown by new

errors in the internal logical of the program

errors that can only be determined at runtime

The exception Base Class

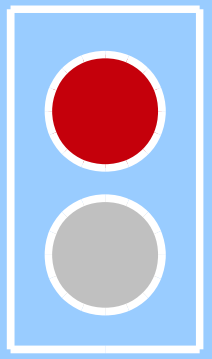
- Lives in <exception>

```
namespace std
{
    class exception
    {
    public:
        exception();
        virtual ~exception();
        exception(const exception &);
        exception & operator=(const exception &);
        virtual const char * what() const;
    private:
        ...
    };
}
```

Refined test

- This test passes only if a *specific* exception is thrown with a *specific* diagnostic string

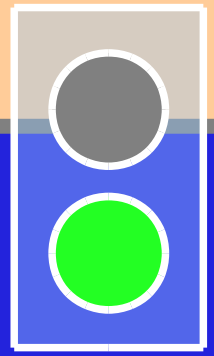
```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    try
    {
        average(empty);
        assert(false);
    }
    catch (std::invalid_argument & error)
    {
        assert(error.what() == std::string("empty"));
    }
    catch (...)
    {
        assert(false);
    }
}
```



Average

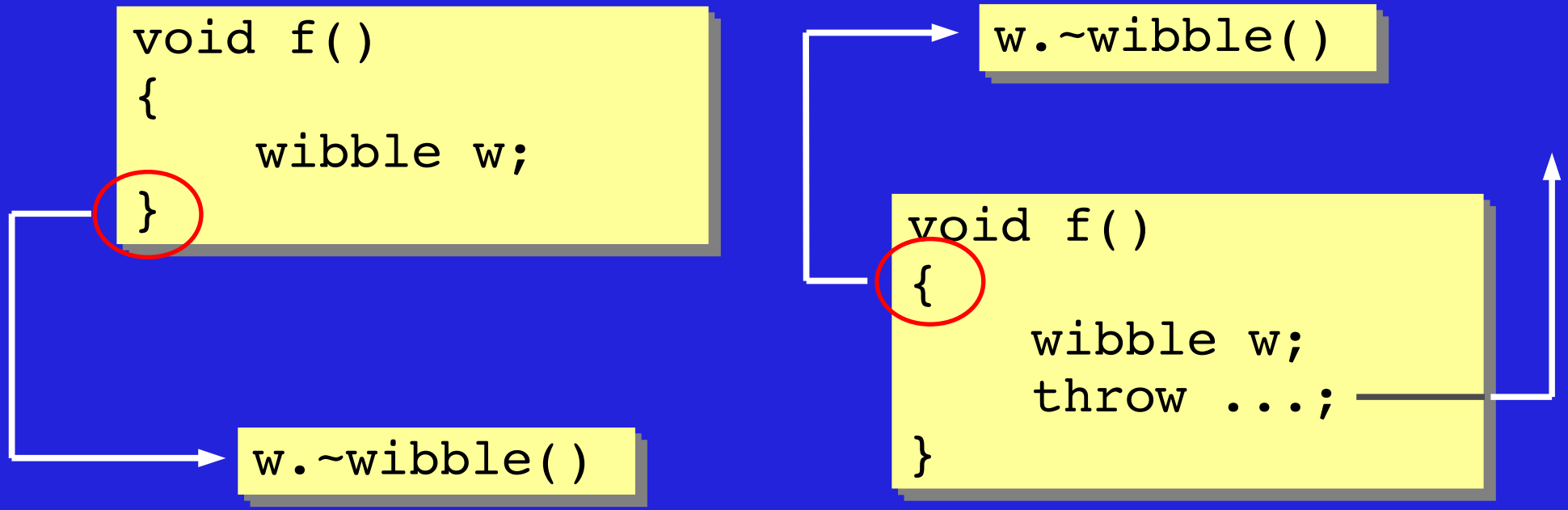
- Modified to make the test pass

```
#include <stdexcept>

double average(const std::vector<double> & data)
{
    if (data.empty())
        throw std::invalid_argument("empty");

    double sum = 0.0;
    for (size_t at = 0; at != data.size(); at++)
        sum += data[at];
    return sum / data.size();
}
```



Object Lifetime

- A fully constructed object *will* have it's destructor called automatically when it goes out of scope - regardless of how it goes out of scope



Resource Acquisition is Initialization

- Acquire a resource in a constructor so you can *automatically* release it in the destructor

```
class auto_file
{
public:
    auto_file(const std::string & name)
        : file(std::fopen(name))
    {
    }
    ~auto_file()
    {
        std::fclose(file);
    }
    ...
private:
    FILE * file;
};
```

```
void eg(const std::string & name)
{
    auto_file file(name);
    ...exception?...
}
```

file.~auto_file()

file.~auto_file()

Common Mistakes/Misunderstanding

throwing new'd objects (drop the new)

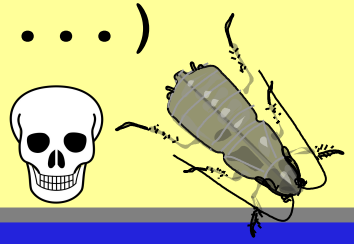
```
throw new std::invalid_argument("...");
```

```
catch (std::exception error)
{
    // ...
}
```

catching by copy
(catch by reference)

```
catch (std::exception & error)
{
}
```

```
catch (...)
{
}
```



catching an
exception and
doing nothing?

Iteration

- Two models for iteration...

```
int array[42];
for (int at = 0; at != 42; ++at)
{
     eg(array[at]);
}
```

random access

```
int array[42];
for (int * pos = &array[0];
     pos != &array[42];
     ++pos)
{
     eg(*pos);
}
```

sequential access

Iteration

- C++ iterators follow the sequential model

```
typedef std::list<int> container;

container values;
for (container::iterator pos = values.begin();
     pos != values.end();
     ++pos)
{
     eg(*pos);
     ...
}
```

```
class iterator
{
     ...
     operator*()
     operator++()
};
bool operator==(iterator, iterator);
bool operator!=(iterator, iterator);
```

begin() and end()

- Container classes offer begin() and end() member functions
- end() returns an iterator "*one-beyond-the-end*"

```
template<typename Type>
class list
{   ...
    class iterator { ... };
    class const_iterator { ... };

    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;
    ...
};
```

begin and end are overloaded on const

Iterator Pair == Range

- Using a pair of iterators to express a range is a dominant C++ idiom
- The standard library offers many algorithms based on iterator pairs

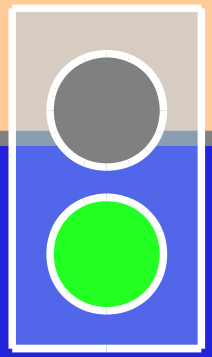
```
template<typename Iterator, typename Value>
Value accumulate(Iterator at, Iterator end,
                 Value sum)
{
    while (at != end)
    {
        sum += *at;
        ++at;
    }
    return sum;
}
```

#include <numeric>

Refactor

- average() implemented using accumulate

```
#include <numeric>

double average(const std::vector<double> & data)
{
    if (data.empty())
        throw std::invalid_argument("empty");

    return std::accumulate(
        data.begin(), data.end(), 0.0)
        / data.size();
}
```



std::sort

```
template<typename Iterator>
void sort(Iterator begin, Iterator end);
```

```
#include <algorithm>

void example(std::list<int> & values)
{
    ...
    std::sort(values.begin(), values.end());
    ...
}
```

std::for_each

```
template<typename Iterator, typename Function>
Function for_each(Iterator begin, Iterator end,
                  Function f);
```

```
#include <algorithm>

void print(int value)
{
    std::cout << value << ',';
}

void example(const std::list<int> & values)
{
   std::for_each(values.begin(),
                 values.end(),
                 print);
}
```