# 9. Handling of Exceptions



9.1. Assertions and debuggers

9.2. **`try`**, **`throw`**, **`catch`**

9.3. Function try blocks

9.4. **`noexcept`**

9.5. **`std::nested_exception`** and **`std::throw_with_nested`**

## 9.1. Assertions and debuggers

A way to ensure that a condition always should hold at some stage in the code:
If the expression supplied to **`assert()`** is false (0), the program *aborts* at the statement with an error.

example00.cpp

```
#include <iostream>  // use of std::cout, std::cin
#include <cstdlib>   // std::rand()
#include <ctime>     // std::time()
#include <cassert>   // assert()
int main() {
  std::srand( std::time(nullptr) );  // time seeds random generator (nullptr)
  double myValue = ( std::rand() % 4 ) - 2;  // gets a random value
  assert(myValue != 0);  // since we'll divide by myValue, it should not be zero
  myValue = 5 / myValue;
  std::cout << myValue << "\n";
  return 0;
}
```

## 9.1. Assertions and debuggers

Assert is a macro and depends on another macro, NDEBUG: If it is defined as a macro name at the point in the source code where <cassert> is included (i.e., `#define NDEBUG`), then assert will be disabled.

The program's state at particular points (e.g., after an assertion fails) can be checked in a ***debugger*** to allow watching the state of a running program. Examples: stop execution at a given code line (breakpoint), examine call stack, print / modify contents of variables, print type definitions, execute line-by-line

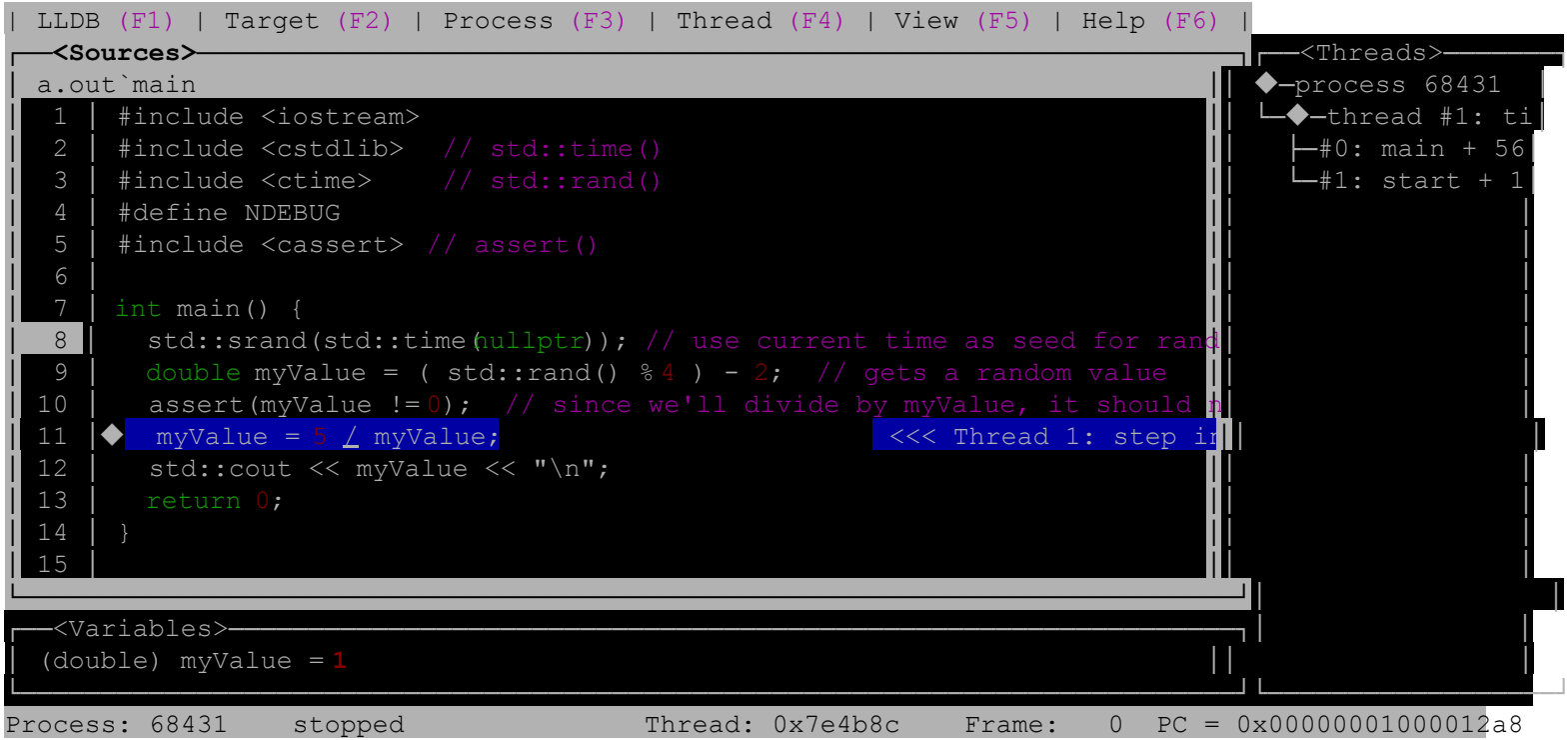
examples: ddd or gdbgui , both use gdb , or lldb → try on the previous code:

```
> g++ -g example00.cpp
> lldb a.out
```

## 9.1. Assertions and debuggers -- lldb example

```
> lldb a.out
bash-5.1$ lldb a.out
(lldb) target create "a.out"
Current executable set to '/Users/kvl/sciebo/UbiComp/Teaching/AdvancedCPP_43UCO1118V/a.out'
(x86_64).
(lldb) b main
Breakpoint 1: where = a.out`main + 15 at example00.cpp:8:14, address = 0x000000010000127f
(lldb) run
Process 68431 launched: '/Users/kvl/sciebo/UbiComp/Teaching/AdvancedCPP_43UCO1118V/a.out'
(x86_64)
Process 68431 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x000000010000127f a.out`main at example00.cpp:8:14
   5       #include <cassert>  // assert()
   6
   7       int main() {
-> 8         std::srand(std::time(nullptr)); // use current time as seed for random generator
   9         double myValue = ( std::rand() % 4 ) - 2;  // gets a random value
   10        assert(myValue != 0);  // since we'll divide by myValue, it should not be zero
   11        myValue = 5 / myValue;
Target 0: (a.out) stopped.
(lldb) gui
```

## 9.1. Assertions and debuggers -- lldb example



## 9.1. Assertions and debuggers

Assertions versus exceptions

Both detect run-time errors in a program, *but*:

- Assert() aborts the program, for the developer to fix their code
- Exceptions allow the program to recover and continue the execution from the first matching catch
  - Examples are any areas where variables obtain values outside the developer's control (e.g., others supply your code with file names which do not exist, or array sizes that do not fit in memory)
  - When no exception matches, the program will still abort

## 9.2. `try`, `throw`, `catch`

When an error occurs, functions or methods may *throw* an exception, to be handled later when *catch*ing the exception. If an exception is *throw*n in the *try*-block, the *try*-block is exited and the associated *catch*-block is executed. Exceptions that go uncaught will cause the program to halt.

```
try {
  throw 0.07f;  // throw an exception of type float
}
catch (float f) {  // float is thrown
  std::cout << "Exception: " << f << "\n";
}
```

## 9.2. `try`, `throw`, `catch`

**throw** supplies an instance of an exception class. This can be a built-in type, but more commonly is a class derived from the **std::exception** class:

```
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::cerr

int divBy(int a, int b) {
  if (b == 0) throw std::runtime_error("Divided by zero.");  // exception type runtime_error
  return a / b;
}

int main() {
  try {
    divBy(7, 0);  // this function throws an exception when b == 0
  }
  catch (const std::exception& e) {
    std::cerr << "Exception handled: " << e.what() << "\n";
  }
  return 0;
}
```

## 9.2. `try`, `throw`, `catch`

Throwing a *custom exception* requires a custom exception class, which inherits from **std::exception** and overrides its **what** method to return an error message:

example01.cpp

```
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::ce

class MyException : public std::exception {
 public:
  MyException(const char * msg) : message(msg) {}  // Constructor sets exception message
  const char * what() { return message.c_str(); }  // Override what() to return own message
 private:
  std::string message;
};

int main() {
  try { throw MyException("Oops, my bad."); }  // create and throw object of MyException
  catch (MyException& e) { std::cerr << "Exception handled: " << e.what() << "\n"; }
  return 0;
}
```

## 9.2. `try`, `throw`, `catch`

The **std::exception** class has many subclasses for specific exceptions:

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (since C++11)
- bad_typeid
- bad_cast
  - bad_any_cast (since C++17)
- bad_optional_access (since C++17)
- bad_expected_access (since C++23)
- bad_weak_ptr (since C++11)
- bad_function_call (since C++11)
- bad_alloc
  - bad_array_new_length (since C++11)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (since C++11)
  - system_error (since C++11)
  - ios_base::failure (since C++11)
  - filesystem::filesystem_error (since C++17)
  - tx_exception (TM TS)
  - nonexistent_local_time (since C++20)
  - ambiguous_local_time (since C++20)
  - format_error (since C++20)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (since C++17)

## 9.3. Function try blocks

Function try blocks allow to establish an exception handler around a function's body, instead of a block of code inside the function body. Function try blocks can catch both base and the current class exceptions:

```
#include <iostream>  // std::cout, std::runtime_error, std::exception, std::cerr
class Superclass {
 public:
  Superclass(int x) : x(x) { if (x < 0) throw 1; }  // an exception can be thrown here
  int x;
};
class Subclass : public Superclass {
 public:       // what if we want to catch Superclass's constructor thrown exception here? …
  Subclass(int x) : Superclass(x) {}
};
int main() {   // … instead of here?
  try { Subclass sub(-5); } catch (int) { std::cout << "Oops, my bad.\n"; }
  return 0;
}
```

## 9.3. Function try blocks

So instead the function try block can be wrapped around:

example02.cpp

```
#include <iostream>  // std::cout,std::runtime_error,std::exception,std::cerr
class Superclass {
 public:
  Superclass(int x) : x(x) { if (x < 0) throw 1; }  // an exception can be thrown here
  int x;
};

class Subclass : public Superclass {
 public:
  // exceptions from A's constructor are now caught here -- but note the throw --- ...
  Subclass(int x) try : Superclass(x) {}
                  catch (int) { std::cerr << "Oops, my bad."; throw; }
};

int main() {    // ... instead of here
  try { Subclass sub(-5); } catch (int) { std::cout << "Oops, my bad.\n"; }  return 0;
}
```

## 9.3. Function try blocks

Function try blocks for constructors are limited though: They *cannot* resolve the thrown exception:

```
class Subclass : public Superclass {
 public:
  // exceptions from A's constructor are now caught here -- but note the throw --- …
  Subclass(int x) try : Superclass(x) {}
                  catch (int) { std::cerr << "Oops, my bad."; throw; }
};
```

Once the end of the catch block is reached, exceptions will be implicitly re-thrown. Other methods and destructors, can throw, rethrow, or resolve the current exception via a return statement. Reaching the end of the catch block will implicitly resolve the exception for void-returning functions, and produces undefined behavior for value-returning functions (hence: avoid).

## 9.4. **noexcept**

Exception handling comes at a (small) cost. Since C++11, **noexcept** can be added for a class method or function declaration, to clarify that the function could throw exceptions (or not):

```
int funct() noexcept;  // funct() does not throw (same as noexcept(true))
void (*fp)() noexcept(false);  // fp points to a function that may throw
```

This allows the compiler to optimize the performance by skipping the processes associated with exception handling, resulting in faster execution of the program.

## 9.4. **noexcept** -- Example

example03.cpp

```
#include <iostream>  // use of std::cout, std::cin

int divBy(int a, int b) {  // divBy could throw exceptions (noexcept omitted)
  if (b == 0)
    throw std::runtime_error("Error: Division by zero");
  return a / b;
}

int safeDivBy(int a, int b) noexcept {  // safeDivBy won't throw exceptions (noexcept)
  if (b == 0) {
    std::cerr << "Division by zero in safeDivBy\n";
    std::terminate();
  }
  return a / b;
}

int main() {
  std::cout << "divBy: " << noexcept(divBy(7, 0)) << "\n";          // → "divBy: 0"
  std::cout << "safeDivBy: " << noexcept(safeDivBy(7, 0)) << "\n";  // → "safeDivBy: 1"
  return 0;
}
```

## 9.5. `std::nested_exception` and `std::throw_with_nested`

In C++11 and beyond: Nesting exceptions allow to recursively stack exceptions, generated at the point of the error, without runtime overhead.

example04.cpp

```
#include <iostream>  // std::cout
#include <fstream>   // std::ifstream

void run();  // catch exception + wrap it in nested exception
void open_file(const std::string& s);  // catch exception + wrap it

// Nested exception adds 'level' spaces and prints messages through recursion
void print_exception(const std::exception& e, int level = 0) {
  std::cerr << std::string(level, ' ') << "exception: " << e.what() << "\n";
  try { std::rethrow_if_nested(e); }
  catch (const std::exception& nestedException) {
    print_exception(nestedException, level+1);
  }
}
```

## 9.5. `std::nested_exception` and `std::throw_with_nested`

```
int main() {  // runs run() and prints the caught exception
  try { run(); } catch (const std::exception& e) { print_exception(e); }
  return 0;
}

void run() {  // catch exception + wrap it in nested exception
  try { open_file("nonExistentFile.txt"); }
  catch (...) { std::throw_with_nested(std::runtime_error("run() fail")); }
}

void open_file(const std::string& s) {  // catch exception + wrap it
  try {
    std::ifstream file(s);  // open file and create an IO fail:
    file.exceptions(std::ios_base::failbit);  // raise exception
  }
  catch (...) {
    std::throw_with_nested(std::runtime_error("file error: " + s));
  }
}
```

# 10. Streams and Container Classes



10.1. Streams

10.2. Container Classes

## 10.1. Streams

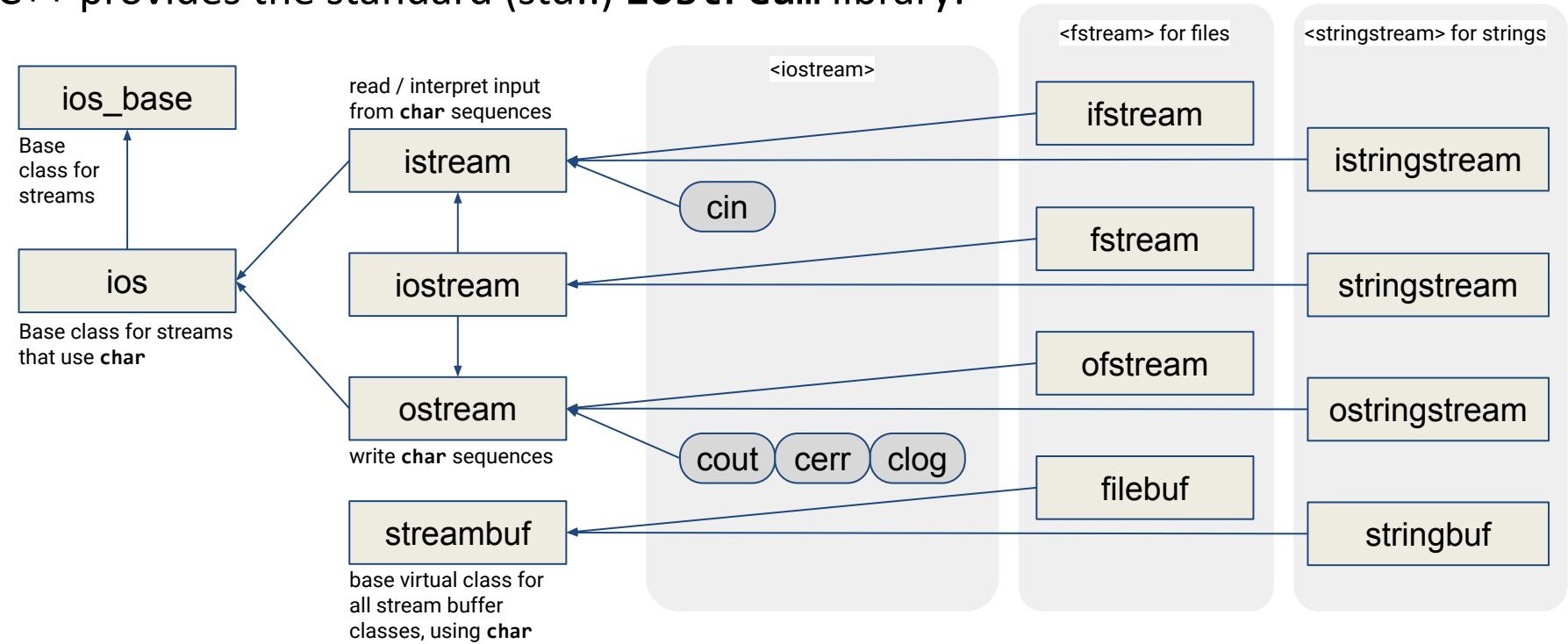
A stream class is a class which provides input and output functionality, as a standard abstraction across devices where input and output operations are performed. A stream can be represented as a source or destination, of characters of indefinite length.

For examples sending characters to and receiving characters from …

disk files, the keyboard and the console, a network connection

## 10.1. Streams

C++ provides the standard (std::) **`iostream`** library:



## 10.1. Streams

Streams have several common methods and operators, including:

- **`get`** / **`put`** a single character from/to a stream before returning
- **`read`** / **`write`** a certain amount of data from/to a stream before returning
- **`getline`** reads characters from an input stream until a delimiter character (usually **`'\n'`** is found) and places them into a string
- stream insertion operator **`<<`** for output to the stream
- stream extraction operator **`>>`** for input to the stream

## 10.1. Streams

Example 00 (difficulty level: 🌶🌶🌶): Install boost and compile:

```
/* change the following code to output the server reply straight to a local html file
   and add exception handling in case of file or connection problems */
#include <fstream>
#include <boost/asio.hpp>

int main() {
  const int bufferSize = 4096;
  boost::asio::ip::tcp::iostream socket("www.example.com", "http");  // socket stream
  std::ofstream outputFile("myTest.txt");  // stream to output file
  outputFile << "Reply of server:\n";
  char reply[bufferSize];
  socket << "GET / HTTP/1.1\r\nHost: www.example.com\r\nConnection: Close\r\n\r\n";
  socket << std::flush;
  socket.read(reply, bufferSize);
  outputFile << reply;  // output the reply of server to our text file
  return 0;
}
```

## 10.2. Container Classes

A container class is a class which implements a data structure containing objects of other classes, with well-defined access patterns (e.g., inserting, finding, removing, or sorting objects), independent of the type of objects stored inside.

Examples: Array, Stack, Queue, List, Tree

## 10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```
class Queue {  // Class for a queue of integers
 public:
  Queue(int size = 100) : maxSize(size), tail(0), head(0), filled(0) { items = new int[size]; }
  ~Queue() { delete[] items; items = nullptr; };
  void put(int data);
  int get();
  bool isFull() const { return filled == maxSize; }
  bool isEmpty() const { return filled == 0; }
  void clear() { filled = 0; head = 0; tail = 0; }  // clear whole queue
 private:
  int *items;    // array of integers
  int maxSize;   // size of items
  int tail;      // position in array to put
  int head;      // position in array to get from
  int filled;    // number of elements in queue
};
```

## 10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```
// put element at the tail of the queue, for example put(17) updates:
// items:  [  ][  ][  ][  ][  ][17][  ] … [  ][  ]
//              head              tail→
void Queue::put(int data) {   // put element at tail
  if (!isFull()) {
    items[tail] = data;
    tail = (tail+1) % maxSize;
    filled++;
  } else {
    throw std::runtime_error("queue: full on put");
  }
}
```

## 10.2. Container Classes

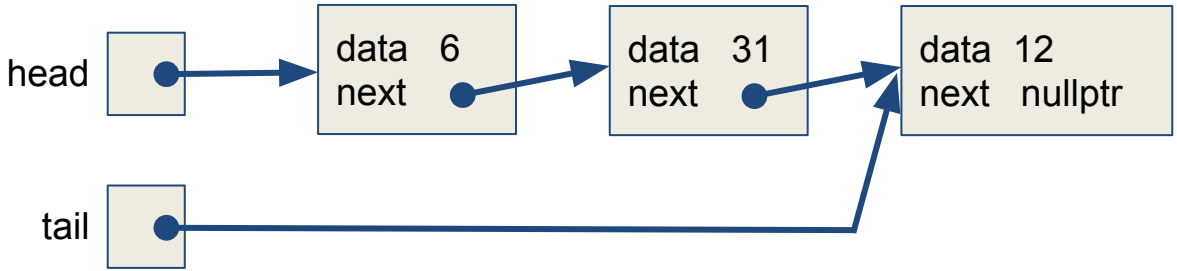
Illustration of a container: A queue of predefined size for integers

```
// gets element at the head of the queue, for example get() updates:
// items:  [  ][  ][  ][  ][  ][  ][  ] … [  ][  ]
//              head→            tail
int Queue::get() {  // get and remove element from head
  int retval;
  if (!isEmpty()) {
    retval = items[head];
    head = (head+1) % maxSize;
    filled--;
  } else {
    throw std::runtime_error("queue: empty on get");
  }
  return retval;
}
```

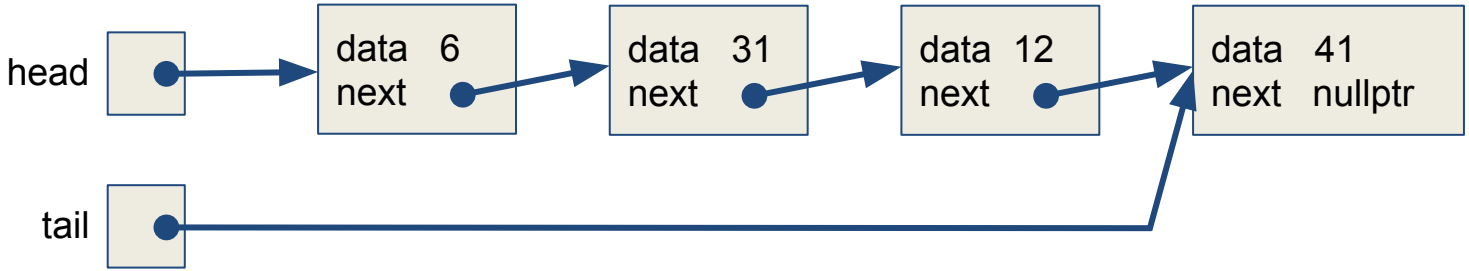
## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **`QueueElement`**s

```
UQueue q;
q.put(6);
q.put(31);
q.put(12);
```



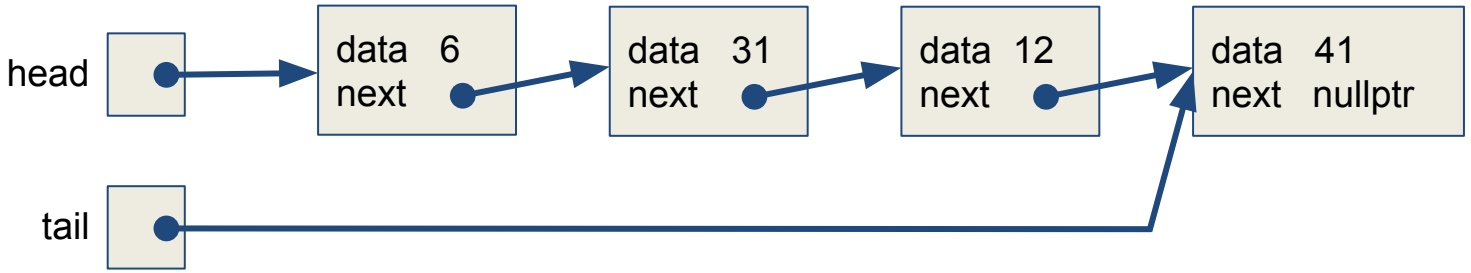

```
q.put(41);
```



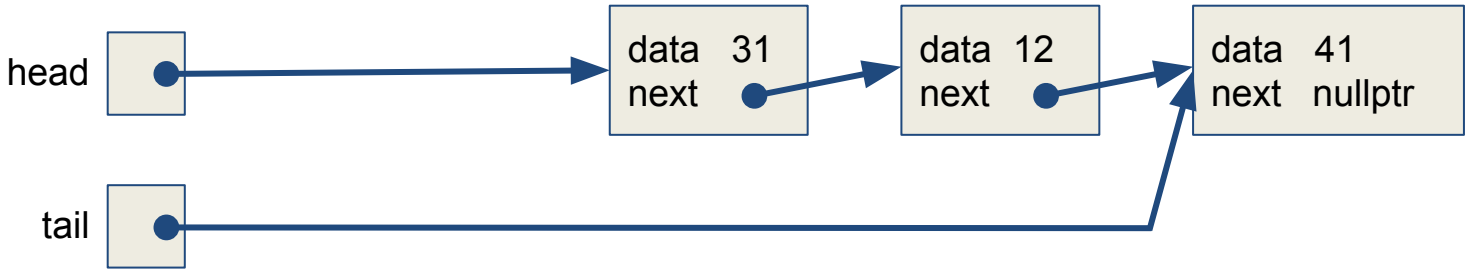
## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **`QueueElement`**s

```
q.put(41);
```




```
int v;
v = q.get();
```



## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
class QueueElement;  // declaration of the element class

class UQueue {  // Class for an unlimited queue of QueueElements
 public:
  UQueue() { head = tail = nullptr; }
  ~UQueue() { clear(); };
  void put(int data);
  int get();
  bool isEmpty() const;
  void clear();
 private:
  QueueElement * head;  // pointer to element to put
  QueueElement * tail;  // pointer to element to get from
};
```

## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
class QueueElement {  // element class, hidden from users
 public:
  QueueElement(int data) : data(data) , next(nullptr) {}
  int data;
  QueueElement * next;
};

void UQueue::clear() {  // iteratively clear the queue of all elements
  QueueElement * elem, * elem_next;
  for (elem = head; elem != nullptr; elem = elem_next) {
    elem_next = elem->next;
    delete elem;
  }
  head = tail = nullptr;
}
```

## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
bool UQueue::isEmpty() const {  // check whether the queue is empty
  return head == nullptr;
}

void UQueue::put(int data) {  // put in new data element at tail
  QueueElement *node = new QueueElement(data);
  if (isEmpty()) {
    head = tail = node;
  } else {
    tail = tail->next = node;  // tail is guaranteed to be valid
  }
}
```

## 10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
int UQueue::get() {  // get and remove element from head
  int retVal;
  if (!isEmpty()) {
    retVal = head->data;
    QueueElement *second = head->next;
    delete head;
    head = second;
  } else {
    throw std::runtime_error("uqueue: empty on get");
  }
  return retVal;
}
```