Efficient C++ Programming and Memory Management

F. Giacomini

INFN-CNAF

ESC'17 – Bertinoro, 22-28 October 2017

https://baltig.infn.it/giaco/cpp-memory-esc17

Outline

Outline

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed
- general-purpose

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed
- general-purpose
- multi-paradigm

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed
- general-purpose
- multi-paradigm
- good from low-level programming to high-level abstractions

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed
- general-purpose
- multi-paradigm
- good from low-level programming to high-level abstractions
- efficient (*"you don't pay for what you don't use"*)

What is C++

C++ is a complex and large programming language (and library)

- strongly and statically typed
- general-purpose
- multi-paradigm
- good from low-level programming to high-level abstractions
- efficient (*"you don't pay for what you don't use"*)
- standard

What is C++

C++ is a complex and large programming language (and library)

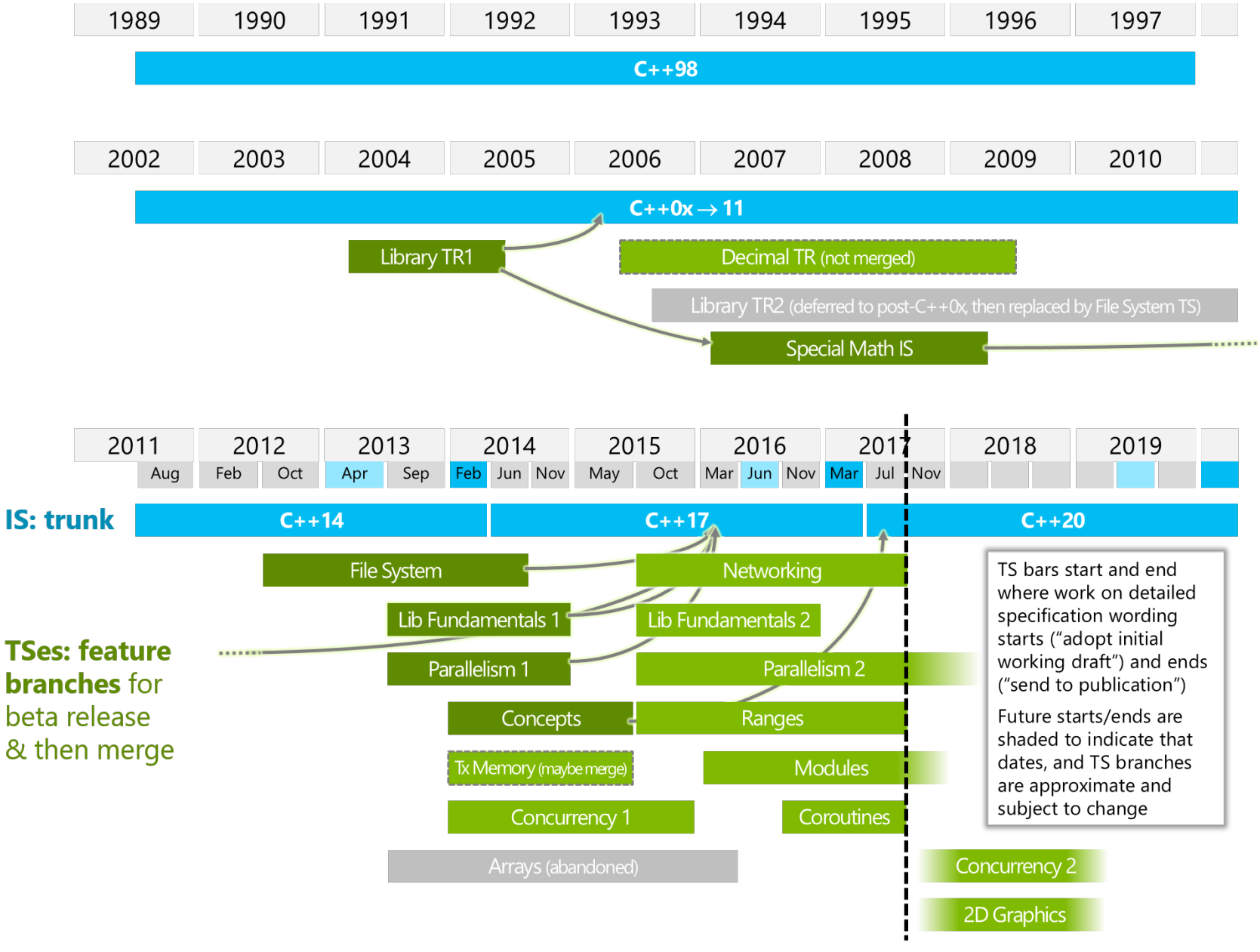
- strongly and statically typed
- general-purpose
- multi-paradigm
- good from low-level programming to high-level abstractions
- efficient (*"you don't pay for what you don't use"*)
- standard

The following material just scratches the surface

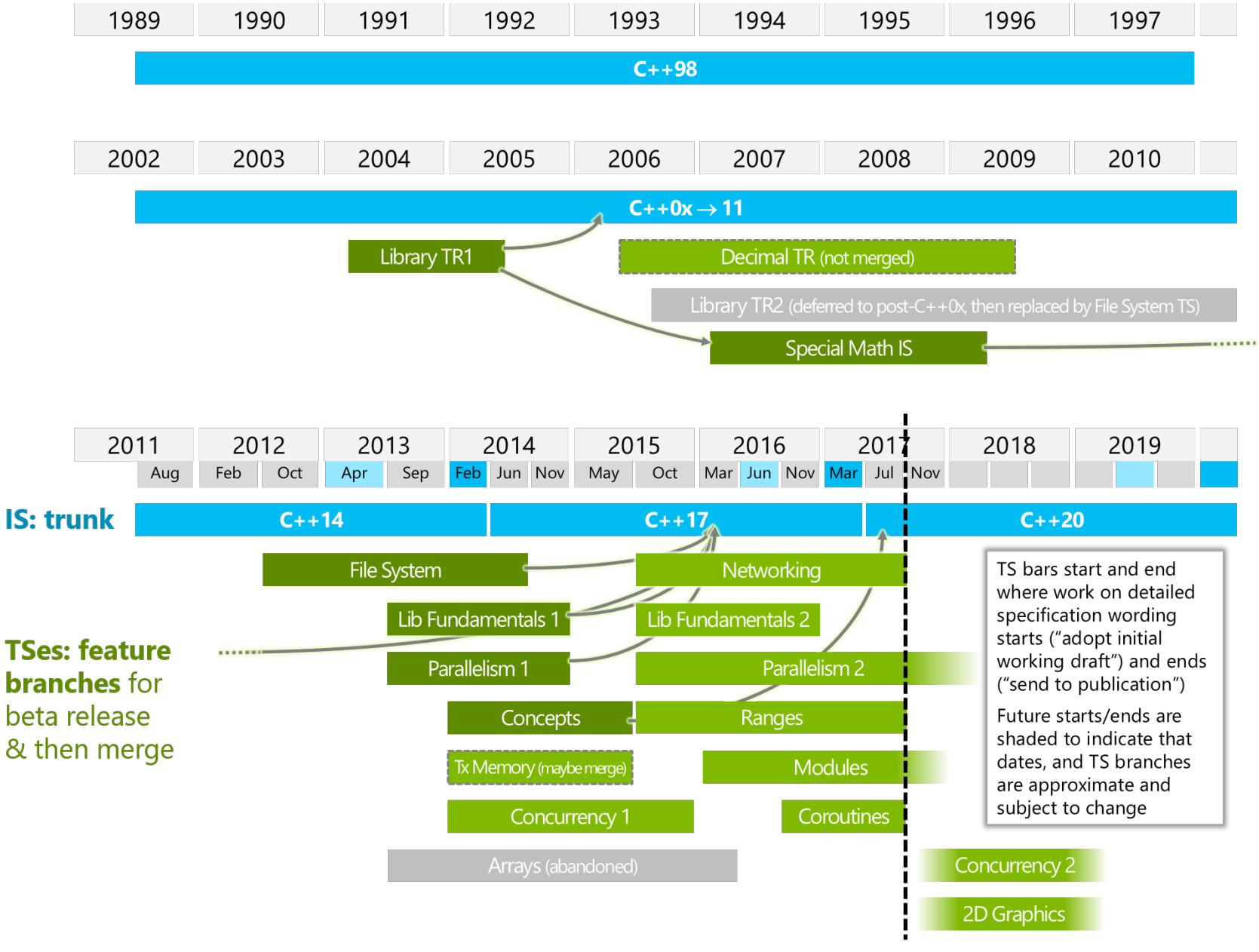
Where to go next

- Start from
  - `https://isocpp.org/`
  - `https://en.cppreference.com/`
  - `https://github.com/isocpp/CppCoreGuidelines`
- Main C++ conferences
  - `https://github.com/cppcon`, `https://youtube.com/cppcon`
  - `https://github.com/boostcon`, `https://youtube.com/boostcon`
- Standard Committee papers
  - `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`

C++ timeline


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
IS: trunk
C++14
C++17
C++20
File System
Networking
Lib Fundamentals 1
Lib Fundamentals 2
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (maybe merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
Concurrency 2
2D Graphics
TSes: feature branches for beta release & then merge
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")
Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change

C++ timeline



"Modern C++ feels like a new language"

Standards

Working drafts, almost the same as the final published document

- C++03 `http://wg21.link/n1905`
- C++11 `http://wg21.link/n3242`
- C++14 `http://wg21.link/n4296`
- C++17 `http://wg21.link/n4659`

For the LaTeXsources see
`https://github.com/cplusplus/draft`

Compilers

- The ESC machines provide gcc 7.2 and clang 5.0
- You can also edit and try your code online with multiple compilers at
  - `http://gcc.godbolt.org/`
  - `http://coliru.stacked-crooked.com/`

Outline

auto

Let the compiler deduce the type of a variable from the initializer

```
auto i = 0;                 // int
auto u = 0U;                // unsigned int
auto p = &i;                // int*
auto d = 1.;                // double
auto c = 'a';               // char
auto s = "a";               // char const*
```

auto

Let the compiler deduce the type of a variable from the initializer

```
auto i = 0;                    // int
auto u = 0U;                   // unsigned int
auto p = &i;                   // int*
auto d = 1.;                   // double
auto c = 'a';                  // char
auto s = "a";                  // char const*
auto t = std::string{"a"}; // std::string
std::vector<std::string> v;
auto it = std::begin(v);   // std::vector<std::string>::iterator
```

auto

Let the compiler deduce the type of a variable from the initializer

```
auto i = 0;                    // int
auto u = 0U;                   // unsigned int
auto p = &i;                   // int*
auto d = 1.;                   // double
auto c = 'a';                  // char
auto s = "a";                  // char const*
auto t = std::string{"a"}; // std::string
std::vector<std::string> v;
auto it = std::begin(v);   // std::vector<std::string>::iterator
using std::literals::chrono_literals;
auto u = 1234us;               // std::chrono::microseconds
```

auto

Let the compiler deduce the type of a variable from the initializer

```
auto i = 0;                    // int
auto u = 0U;                   // unsigned int
auto p = &i;                   // int*
auto d = 1.;                   // double
auto c = 'a';                  // char
auto s = "a";                  // char const*
auto t = std::string{"a"}; // std::string
std::vector<std::string> v;
auto it = std::begin(v);   // std::vector<std::string>::iterator
using std::literals::chrono_literals;
auto u = 1234us;               // std::chrono::microseconds
auto e;                        // error
```

`auto` and references

- `auto` never deduces a reference
- if needed, `&` must be added explicitly

```
T v;

auto  v1 = v;      // T  - v1 is a copy of v
auto& v2 = v;      // T& - v2 is an alias of v
auto  v3 = v2;     // T  - v2 is a copy of v
```

`auto` and `const`

- `auto` makes a mutable copy
- `auto const` (or `const auto`) makes a non-mutable copy
- `auto&` preserves `const`-ness

```
T v;

auto        v1 = v; // T        - v1 is a mutable copy of v
auto const  v2 = v; // T const  - v2 is a non-mutable copy of v
auto&       v3 = v; // T&       - v3 is a mutable alias of v
auto const& v4 = v; // T const& - v4 is a non-mutable alias of v
```

```
T const v;

auto        v1 = v; // T        - v1 is a mutable copy of v
auto const  v2 = v; // T const  - v2 is a non-mutable copy of v
auto&       v3 = v; // T const& - v3 is a non-mutable alias of v
auto const& v4 = v; // T const& - v4 is a non-mutable alias of v
```

How to check the deduced type?

- Trick by S. Meyers

```
template<typename T> struct D;

auto k = 0U;
D<decltype(k)> d; // error: aggregate 'D<unsigned int> d'...

auto const o = 0.;
D<decltype(o)> d; // error: aggregate 'D<const double> d'...

auto const& f = 0.f;
D<decltype(f)> d; // error: aggregate 'D<const float&> td'...

auto s = "hello";
D<decltype(s)> d; // error: aggregate 'D<const char*> d'...

auto& t = "hello";
D<decltype(t)> d; // error: aggregate 'D<const char (&)[6]> d'...
```

- `decltype` returns the type of an expression
  - at compile time

Outline

Functions

- A function associates a sequence of statements (the function *body*) with a name and a list of zero or more parameters

```
std::string cat(std::string const& s, int i)
{
  return s + '-' + std::to_string(i);
}

// cat(std::string{"XYZ"}, 5) returns std::string{"XYZ-5"}
```

- A function may return a value
- A function returning a `bool` is called a *predicate*

```
bool less(int n, int m) { return n < m; }
```

- Multiple functions can have the same name → *overloading*
  - different parameter lists

Functions

- A function associates a sequence of statements (the function *body*) with a name and a list of zero or more parameters

```
auto        cat(std::string const& s, int i)
{
  return s + '-' + std::to_string(i);
}

// cat(std::string{"XYZ"}, 5) returns std::string{"XYZ-5"}
```

- A function may return a value
- A function returning a `bool` is called a *predicate*

```
bool less(int n, int m) { return n < m; }
```

- Multiple functions can have the same name → *overloading*
  - different parameter lists

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying `all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying `all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`

Modifying `copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`

STL Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`

STL Algorithms

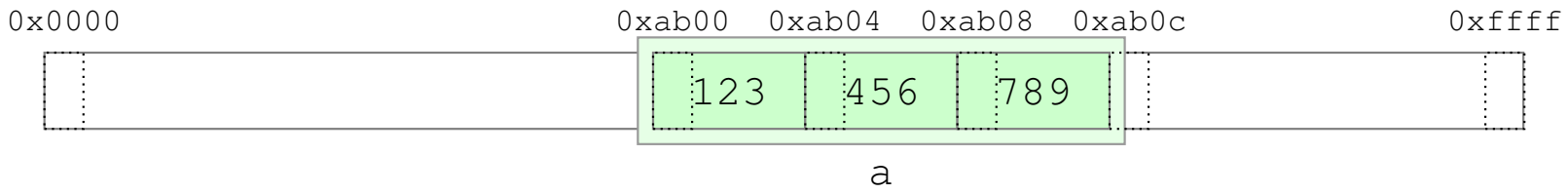
- Generic functions that operate on ranges of objects
- Implemented as function templates

Non-modifying	`all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...`
Modifying	`copy fill generate transform remove replace swap reverse rotate shuffle sample unique ...`
Partitioning	`partition stable_partition ...`
Sorting	`sort partial_sort nth_element ...`
Set	`set_union set_intersection set_difference ...`
Min/Max	`min max minmax lexicographical_compare clamp ...`
Numeric	`iota accumulate inner_product partial_sum adjacent_difference ...`

Range

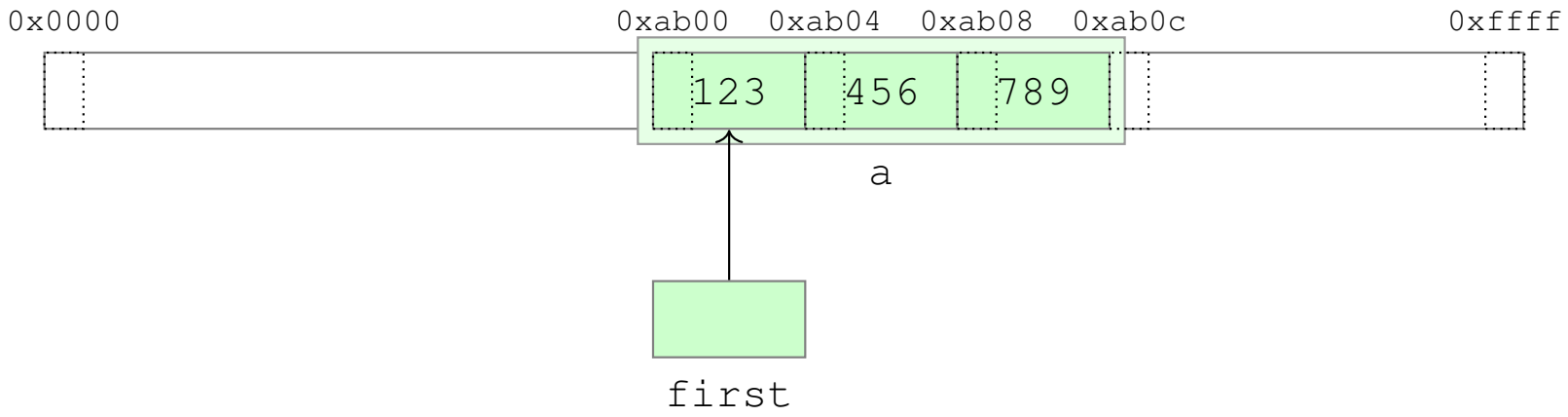
- A range is defined by a pair of iterators [*first*, *last*), with *last* referring to one past the last element in the range
  - the range is *half-open*
  - *first* == *last* means the range is empty
  - *last* can be used to return failure
- An iterator is a generalization of a pointer
  - it supports the same operations, possibly through overloaded operators
  - certainly `*` `++` `->` `==` `!=`, maybe `--` `+` `-` `+=` `-=` `<`
- Ranges are typically obtained from containers calling specific methods

Range (cont.)



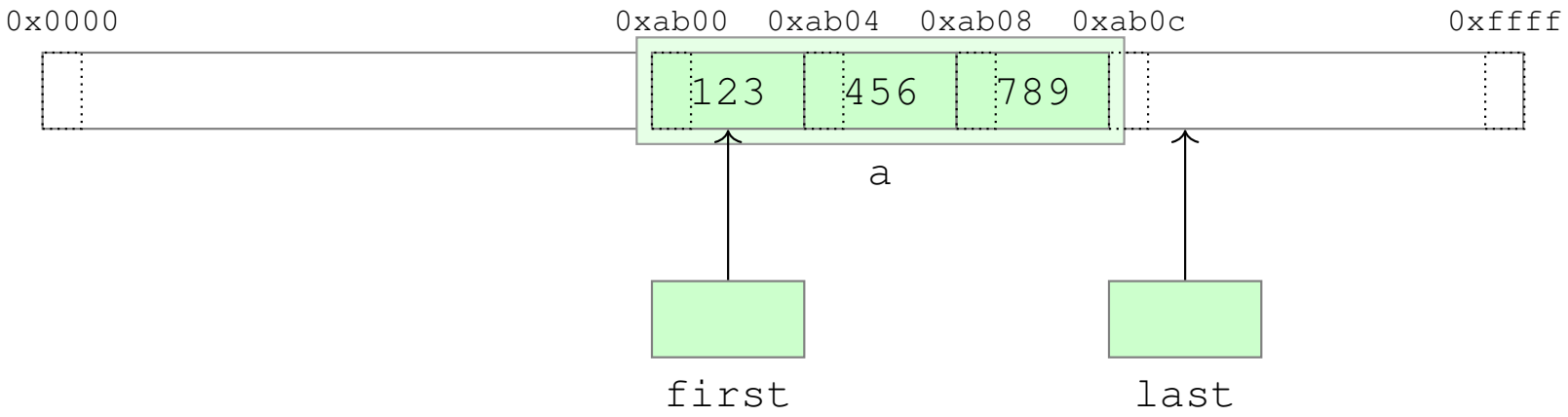

```
std::array<int,3> a = {123, 456, 789};
```

Range (cont.)



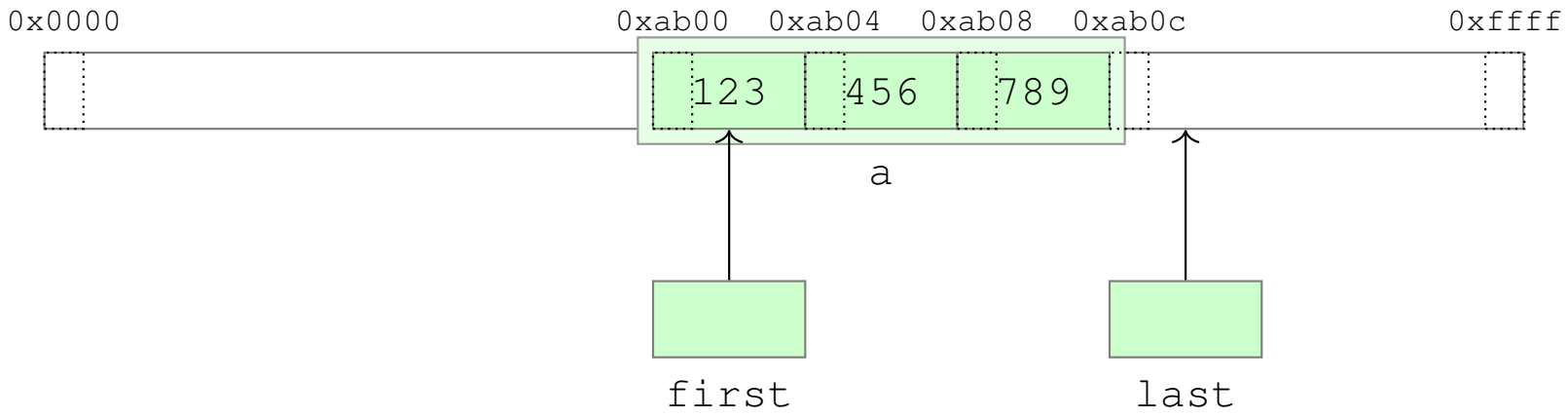

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();          // or std::begin(a)
```

Range (cont.)



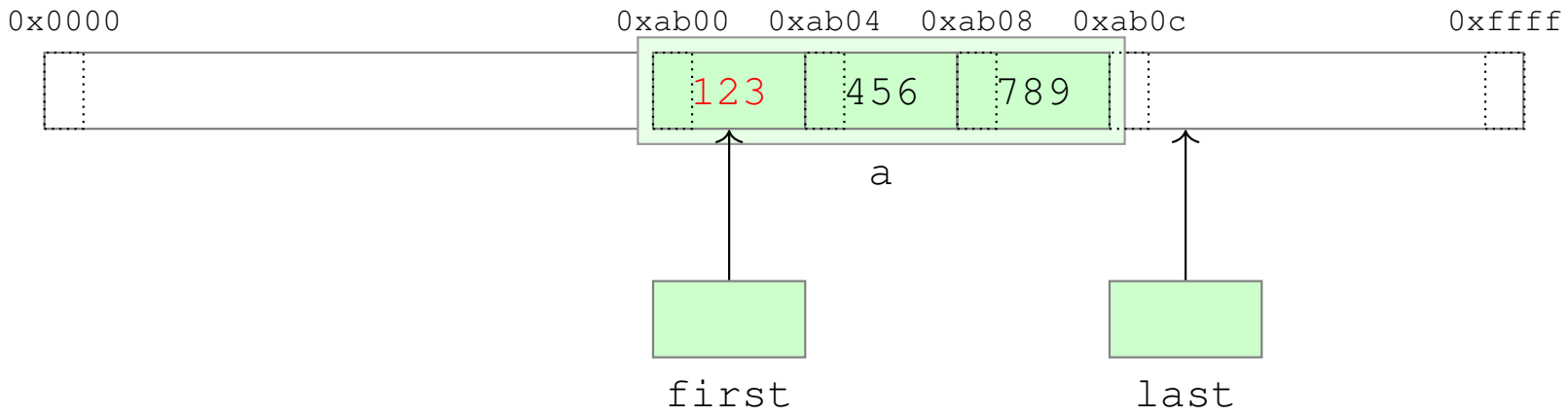

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();          // or std::begin(a)
auto last = a.end();             // or std::end(a)
```

Range (cont.)



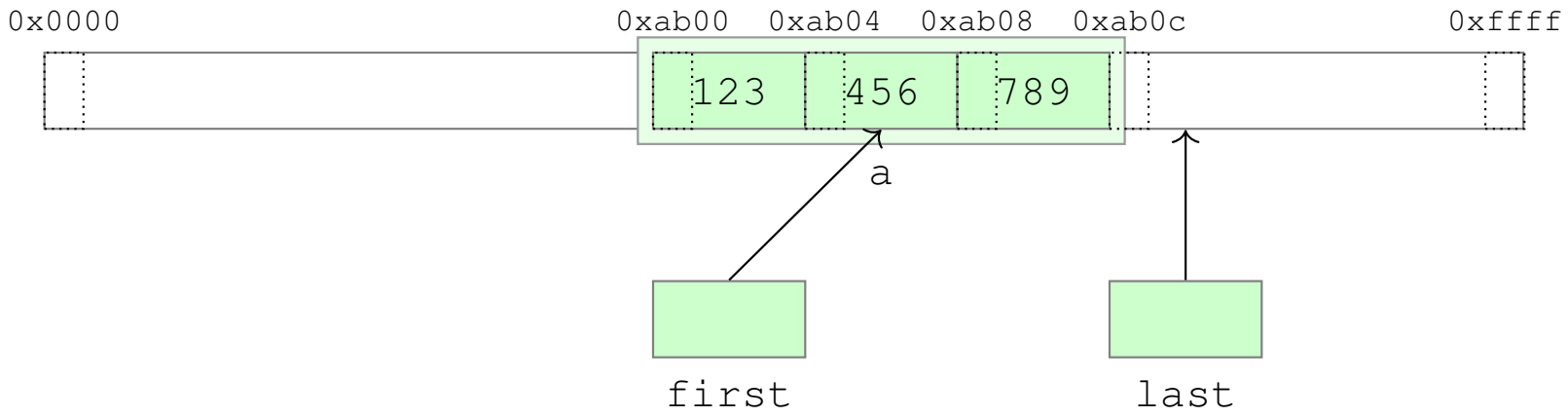

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



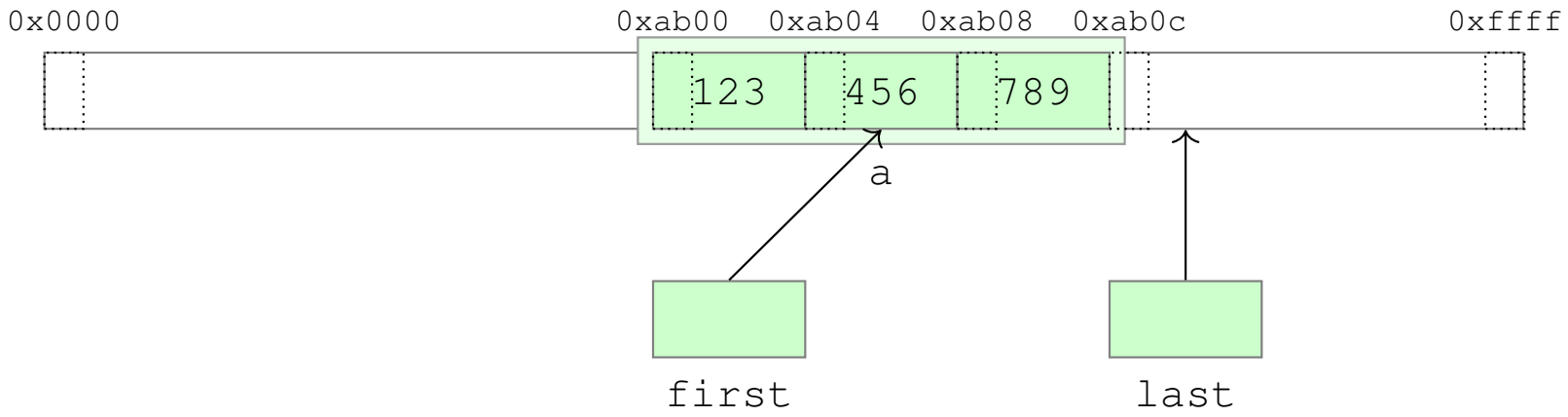

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



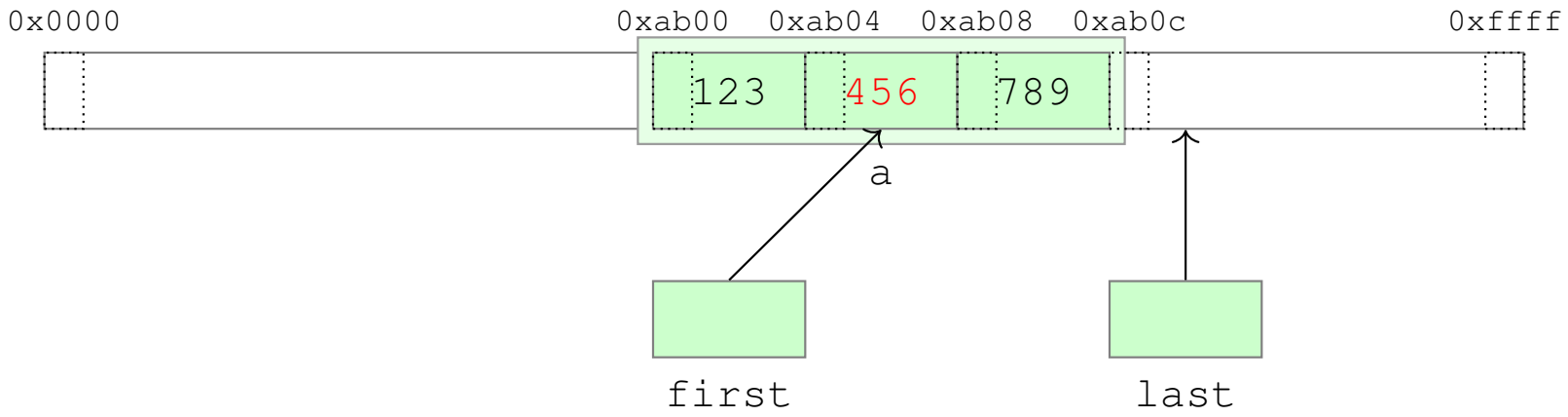

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



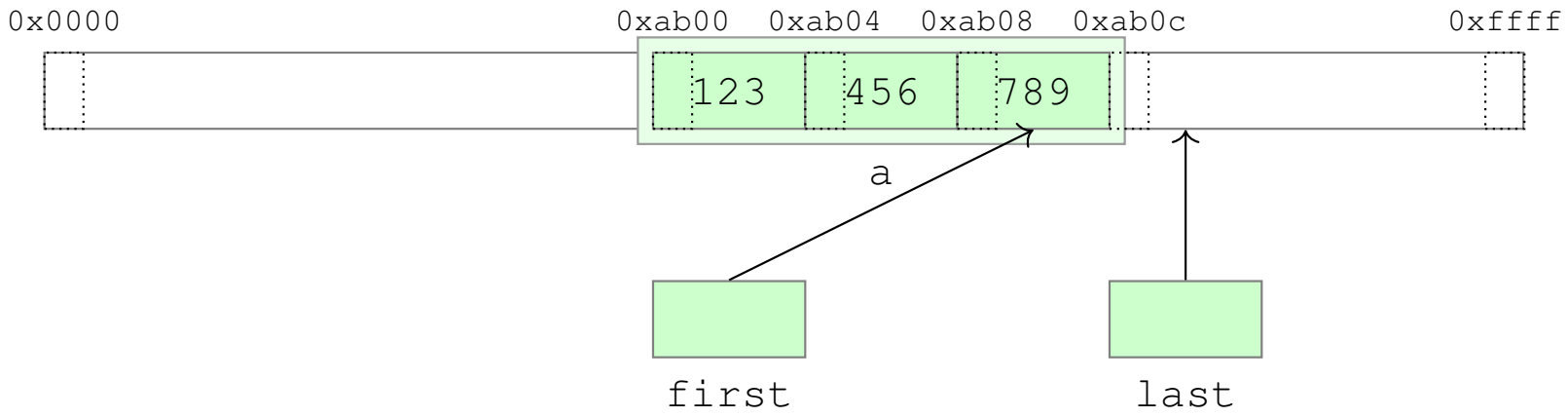

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



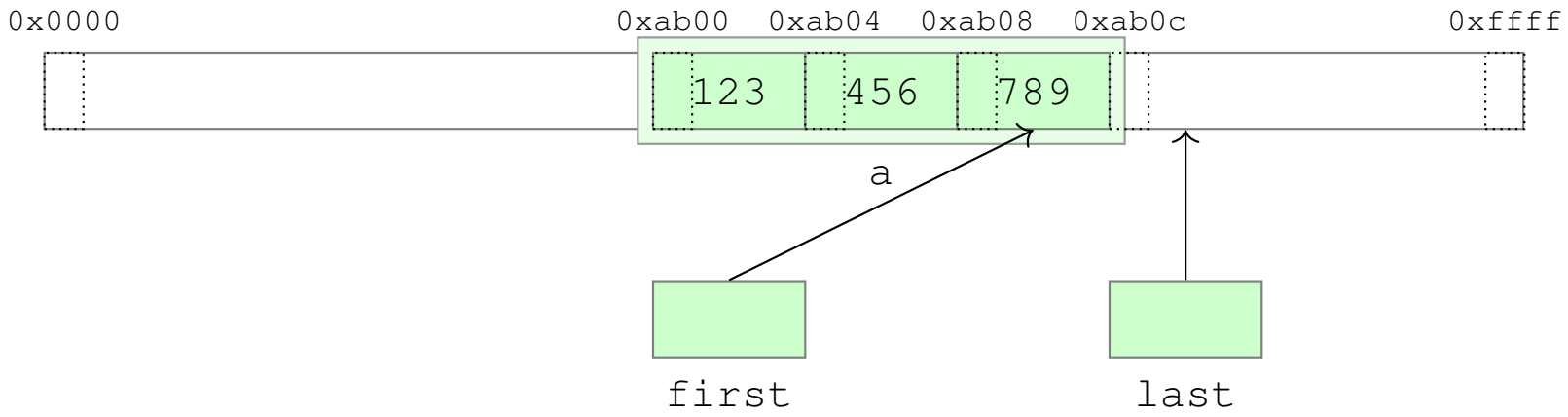

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



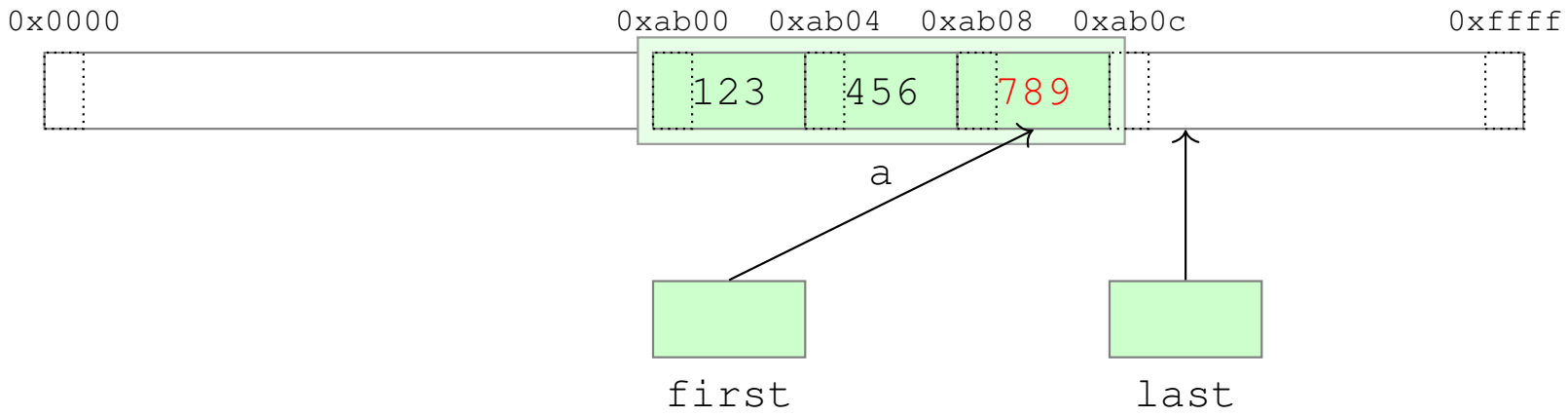

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



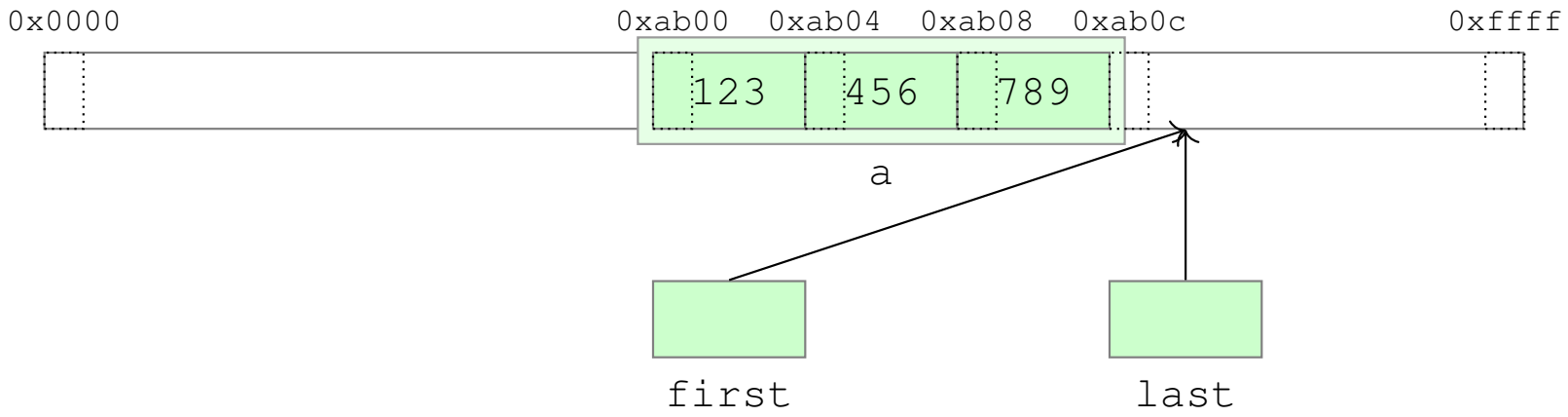

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();          // or std::begin(a)
auto last = a.end();             // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



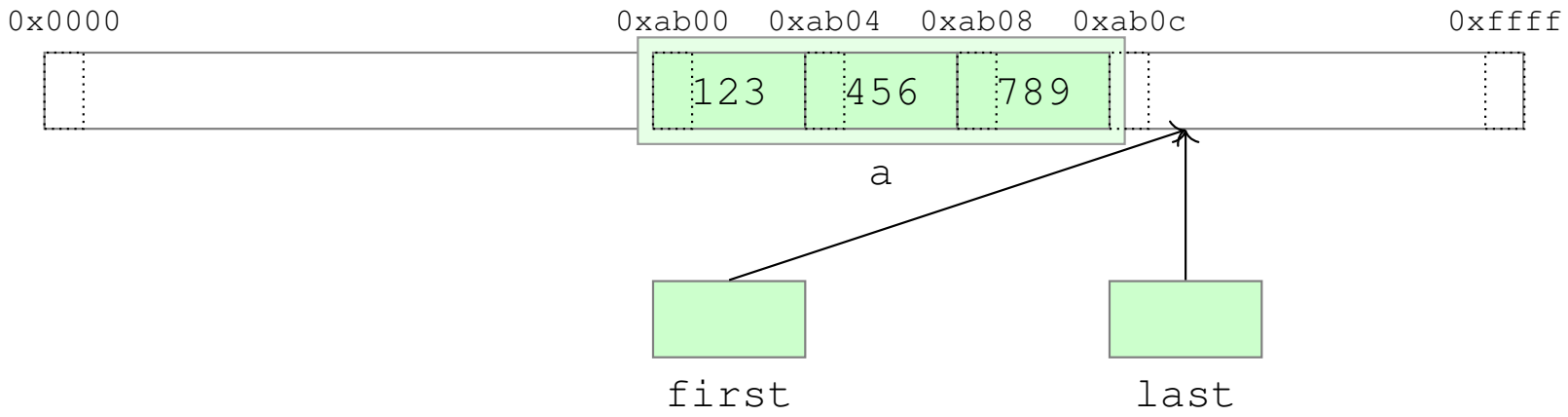

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



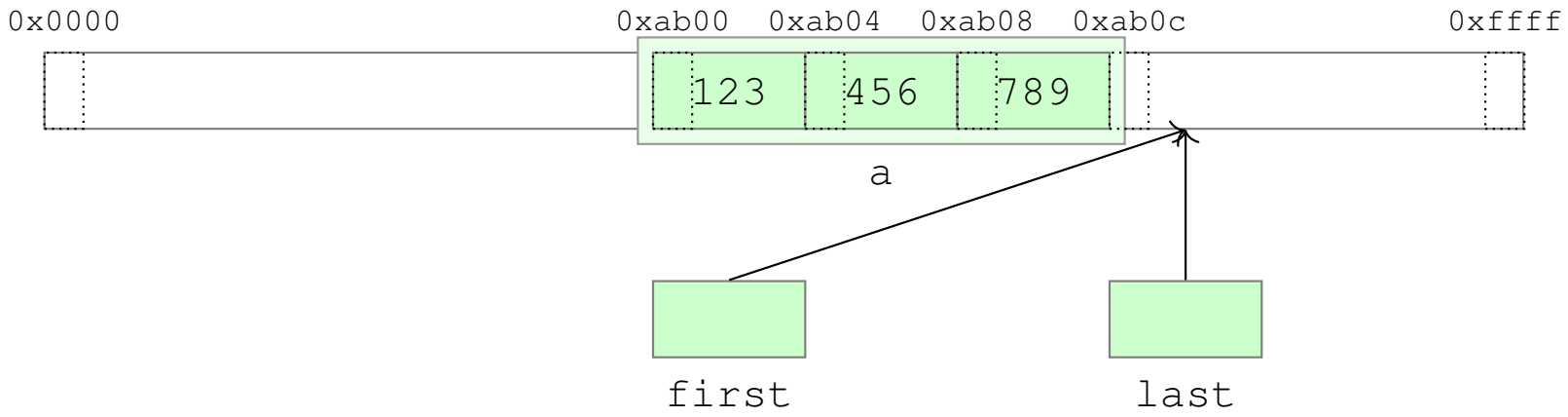

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



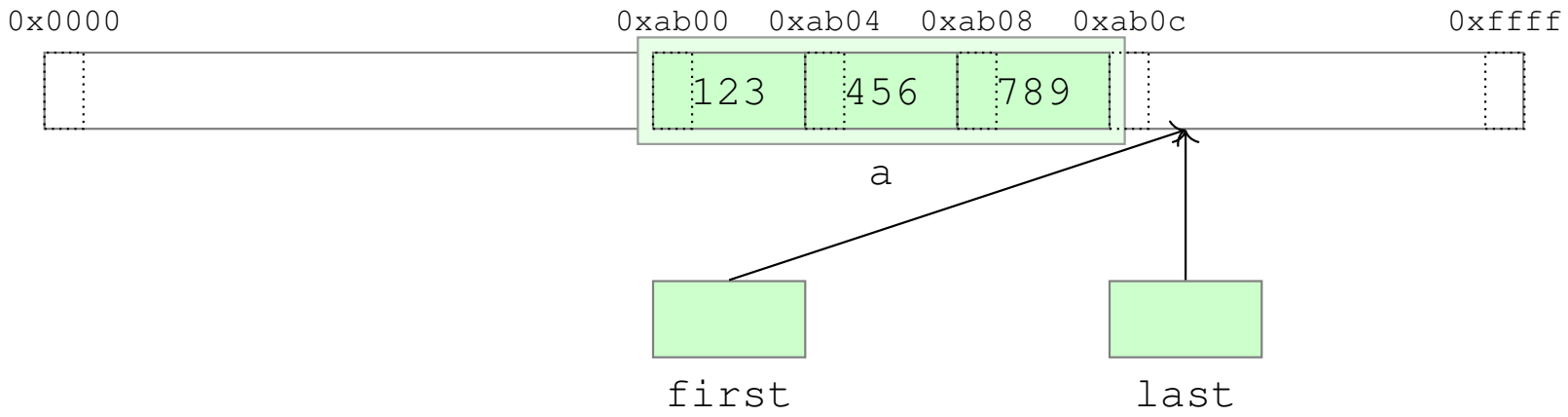

```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto last = a.end();            // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto const last = a.end();      // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::array<int,3> a = {123, 456, 789};
auto first = a.begin();         // or std::begin(a)
auto const last = a.end();      // or std::end(a)
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::array<T>::iterator` models the *RandomAccessIterator* concept

Generic programming

- A style of programming in which algorithms are written in terms of concepts

Generic programming

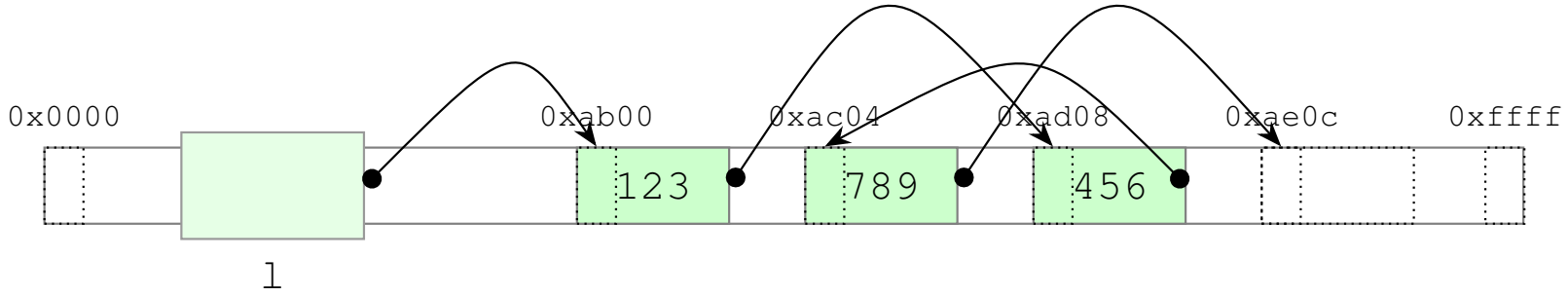
- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy
  - e.g. supported expressions, nested typedefs, memory layout, ...

Generic programming

- A style of programming in which algorithms are written in terms of concepts
- A concept is a set of requirements that a type needs to satisfy
  - e.g. supported expressions, nested typedefs, memory layout, . . .

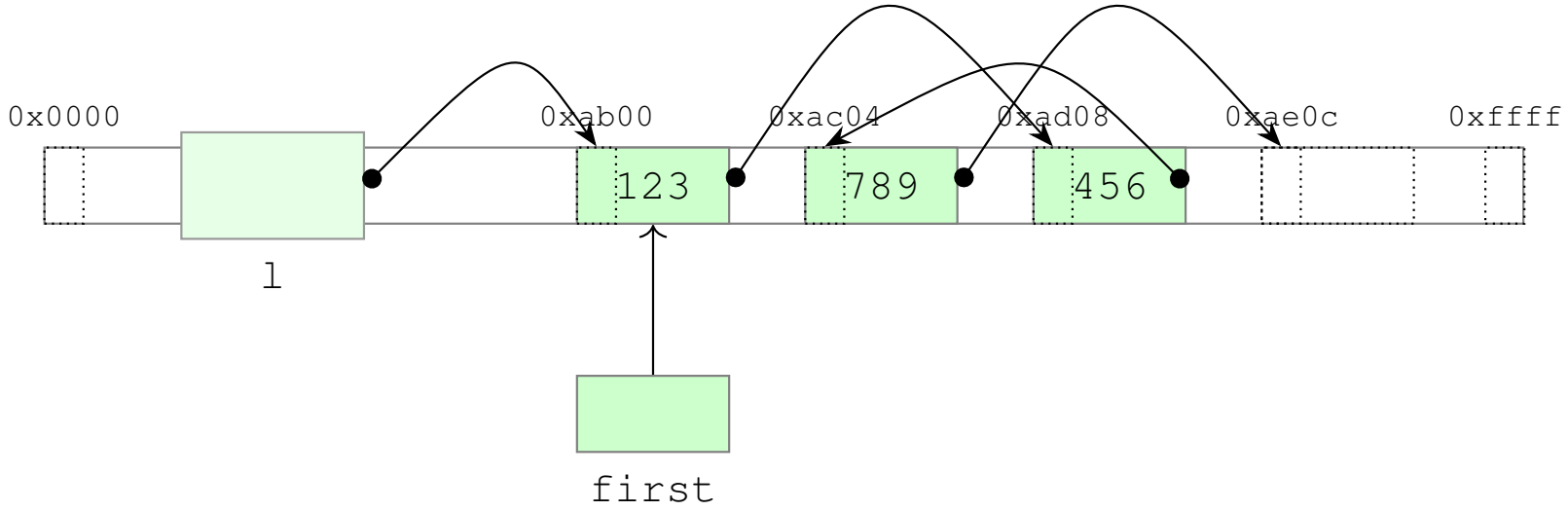
```
template <class Iterator, class T>
Iterator
find(Iterator first, Iterator last, const T& value)
{
  for (; first != last; ++first)
    if (*first == value)
      break;
  return first;
}
```

Range (cont.)



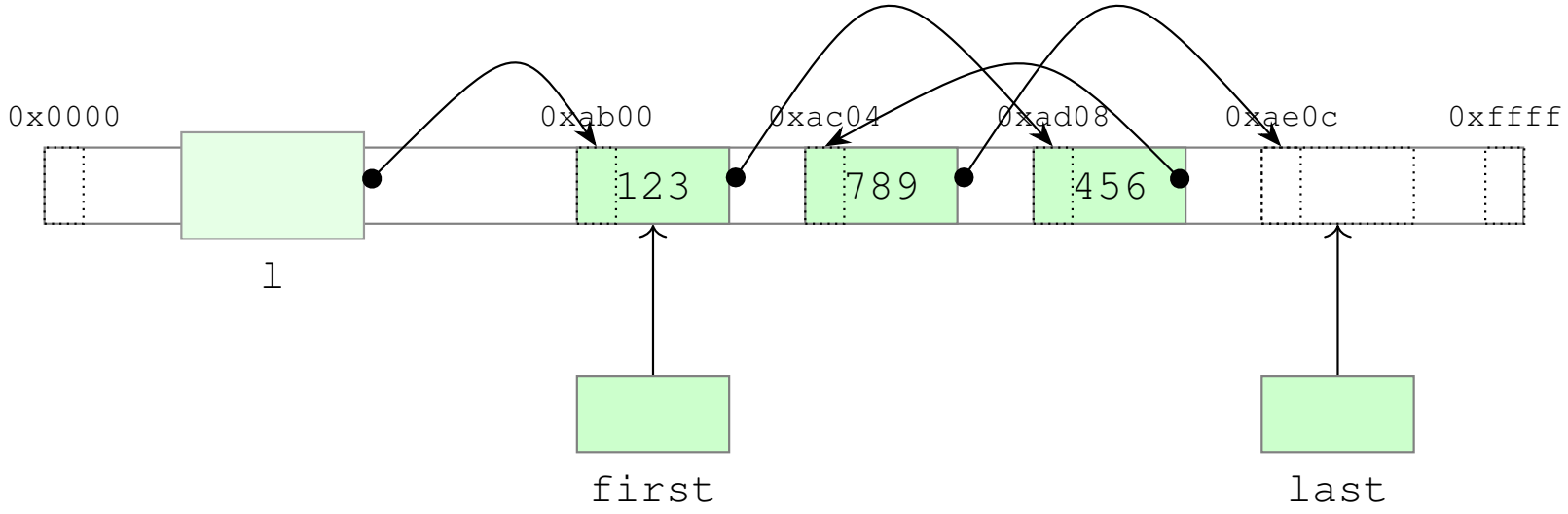

```
std::forward_list<int> l = {123, 456, 789};
```

Range (cont.)



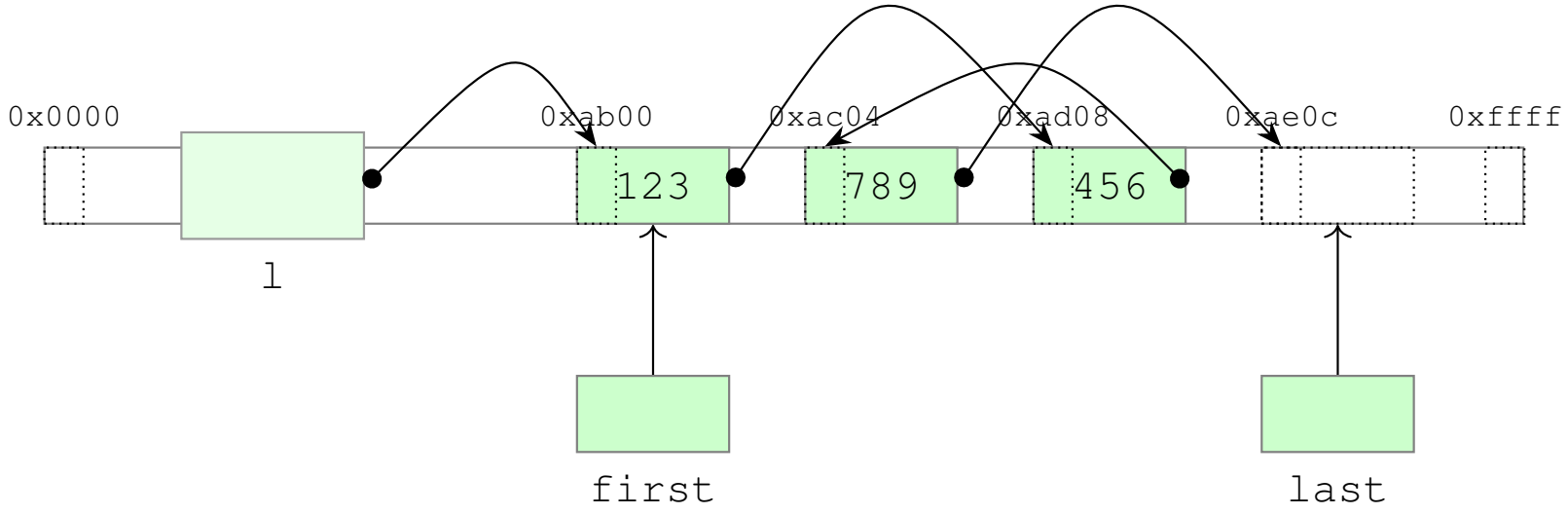

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
```

Range (cont.)



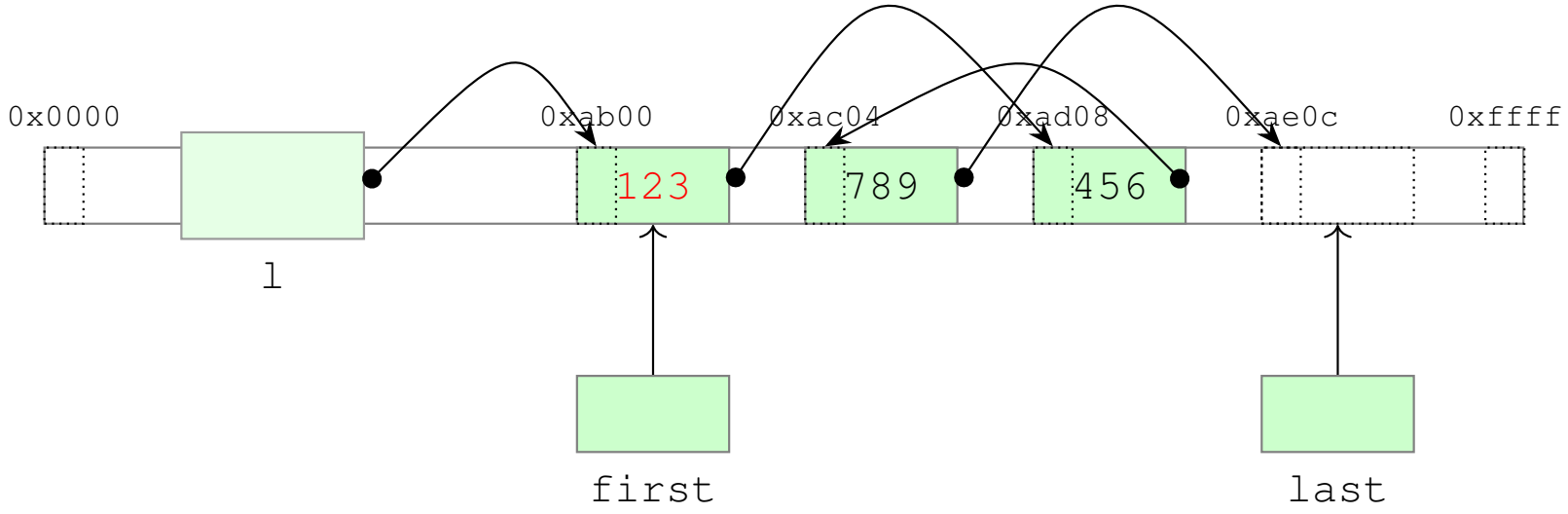

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
```

Range (cont.)



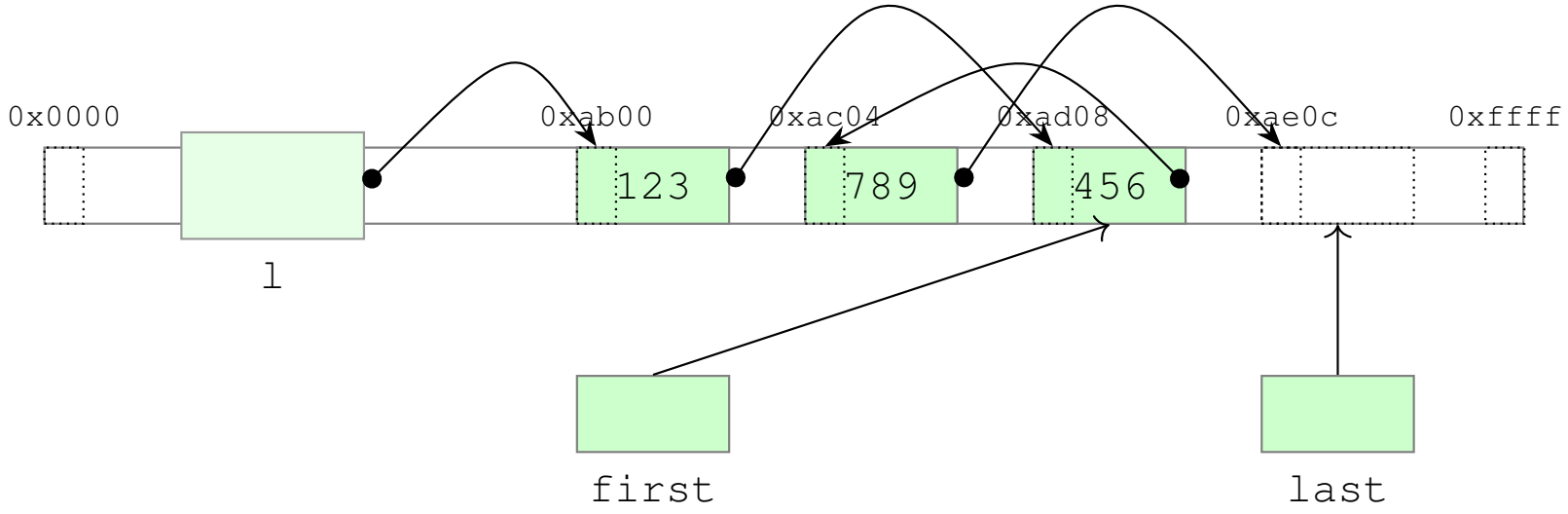

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



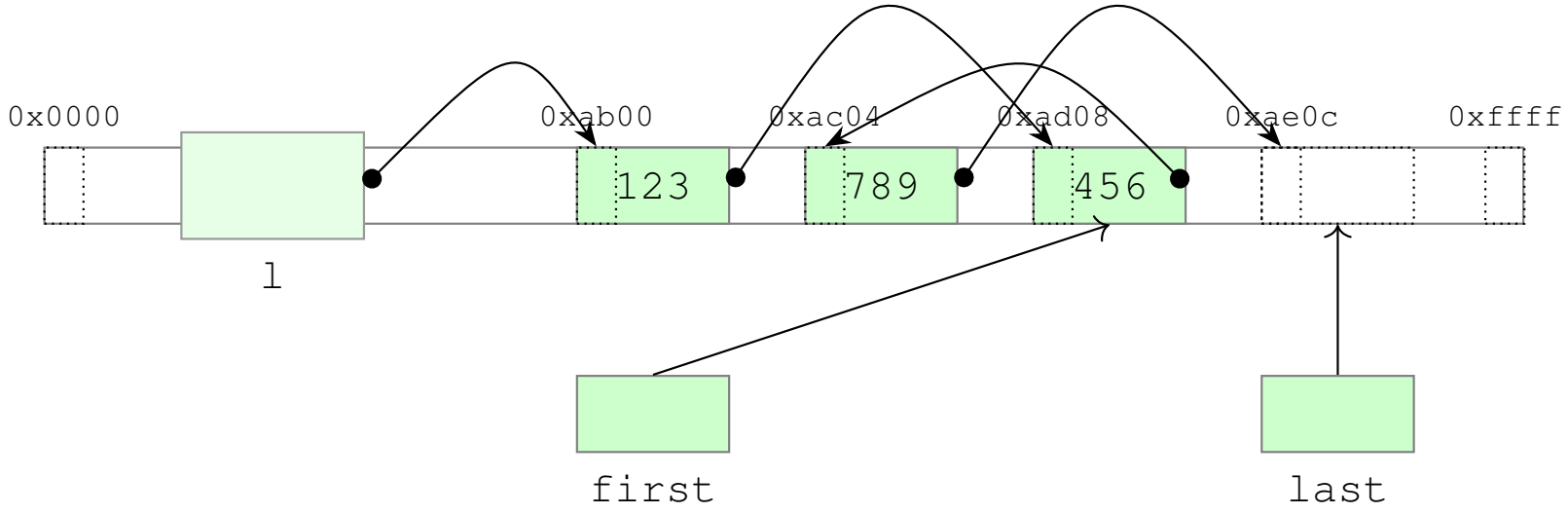

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



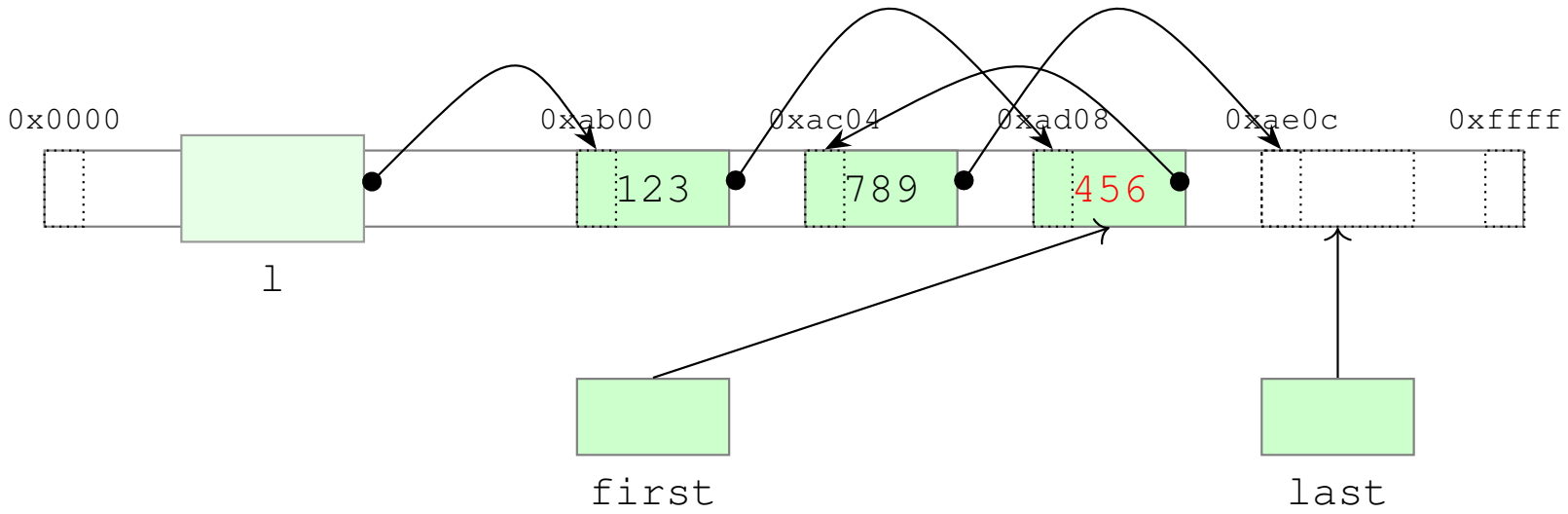

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



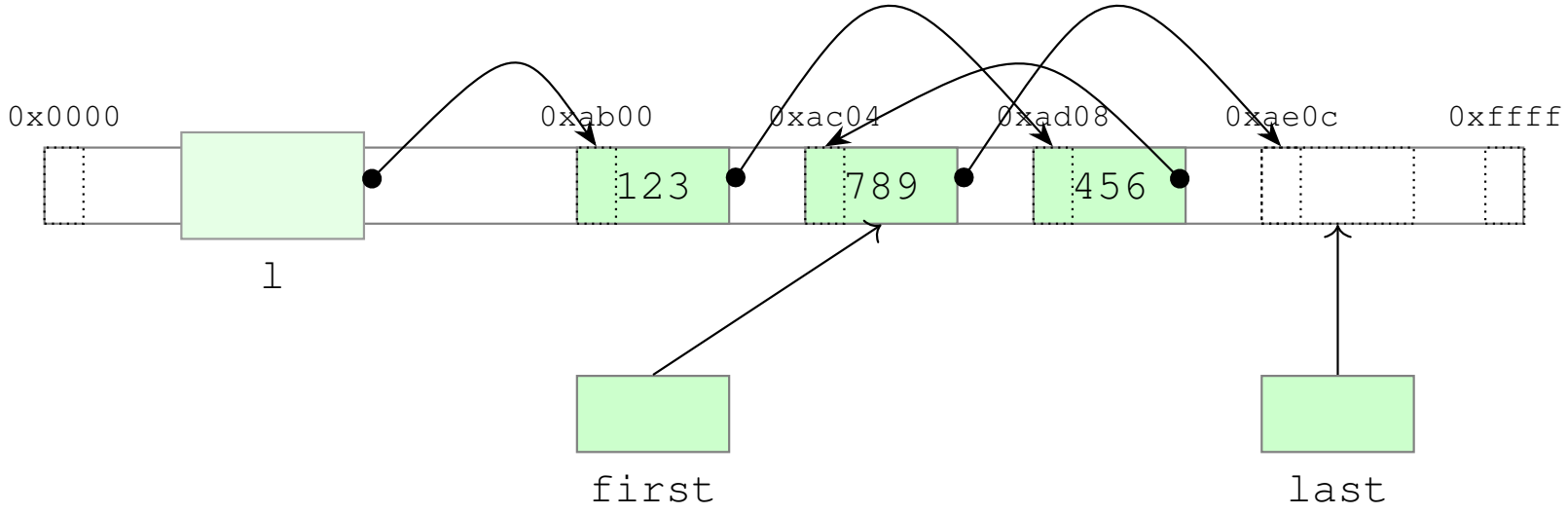

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



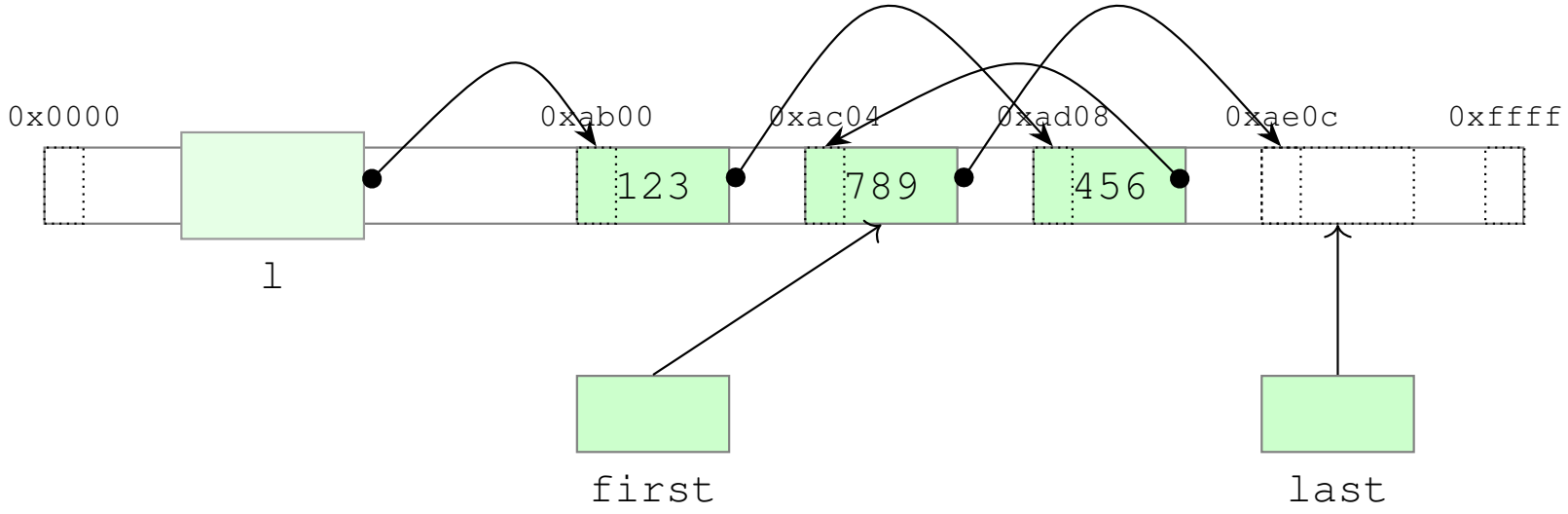

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



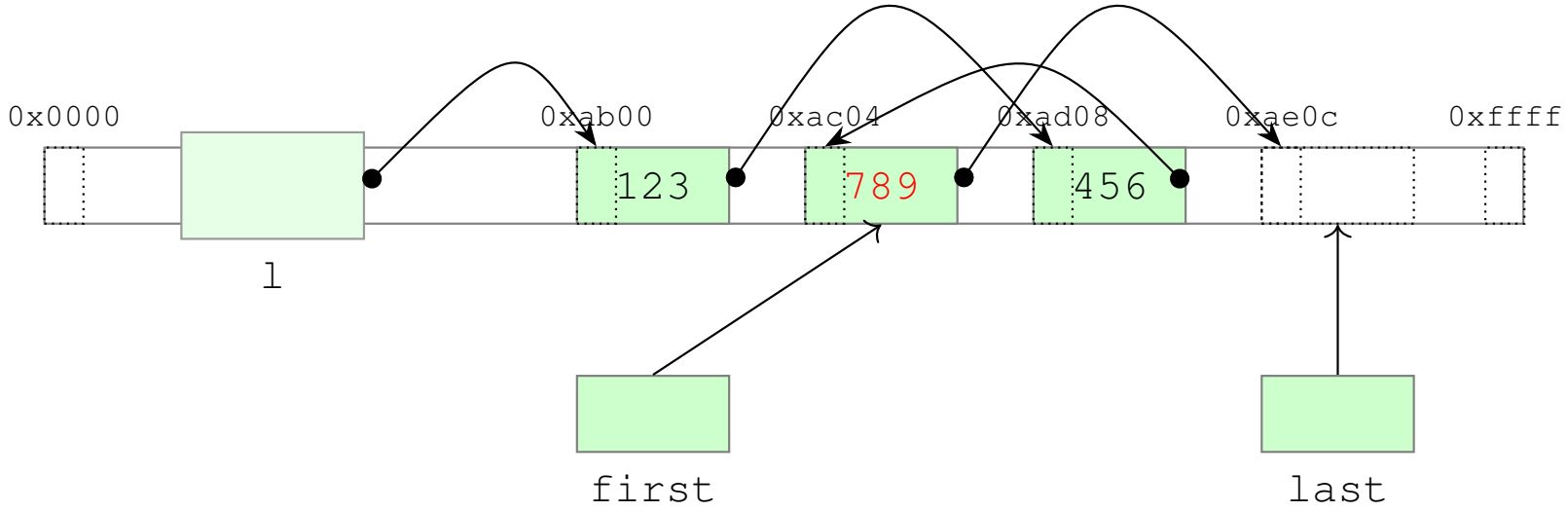

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



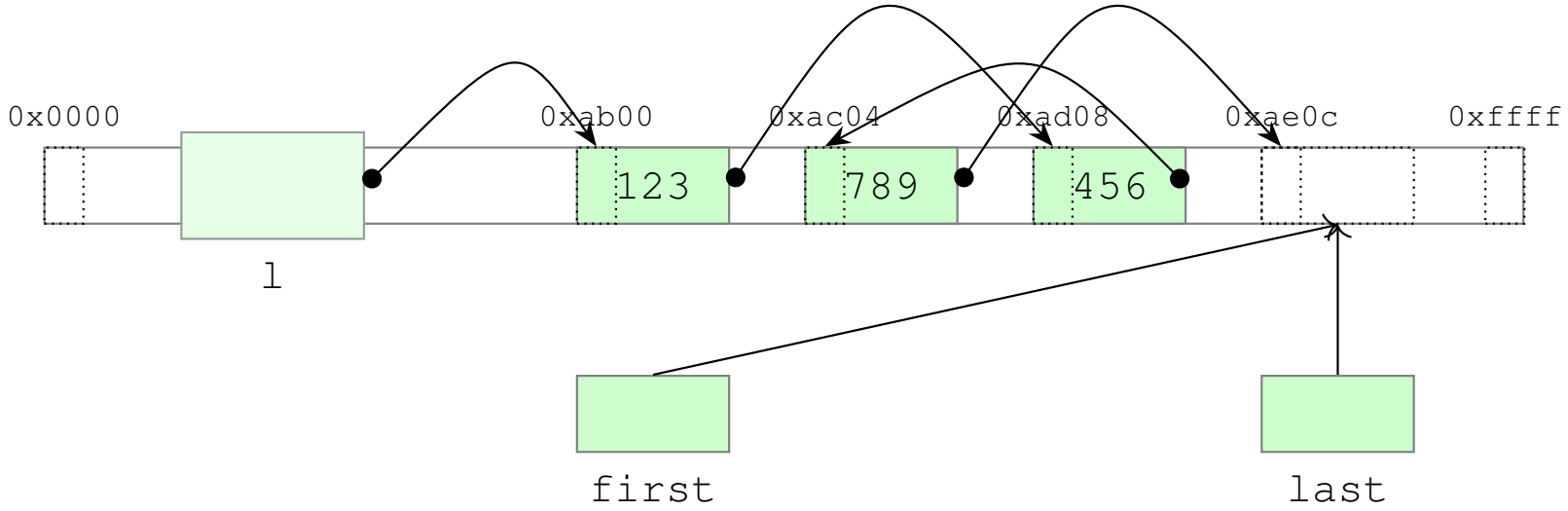

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



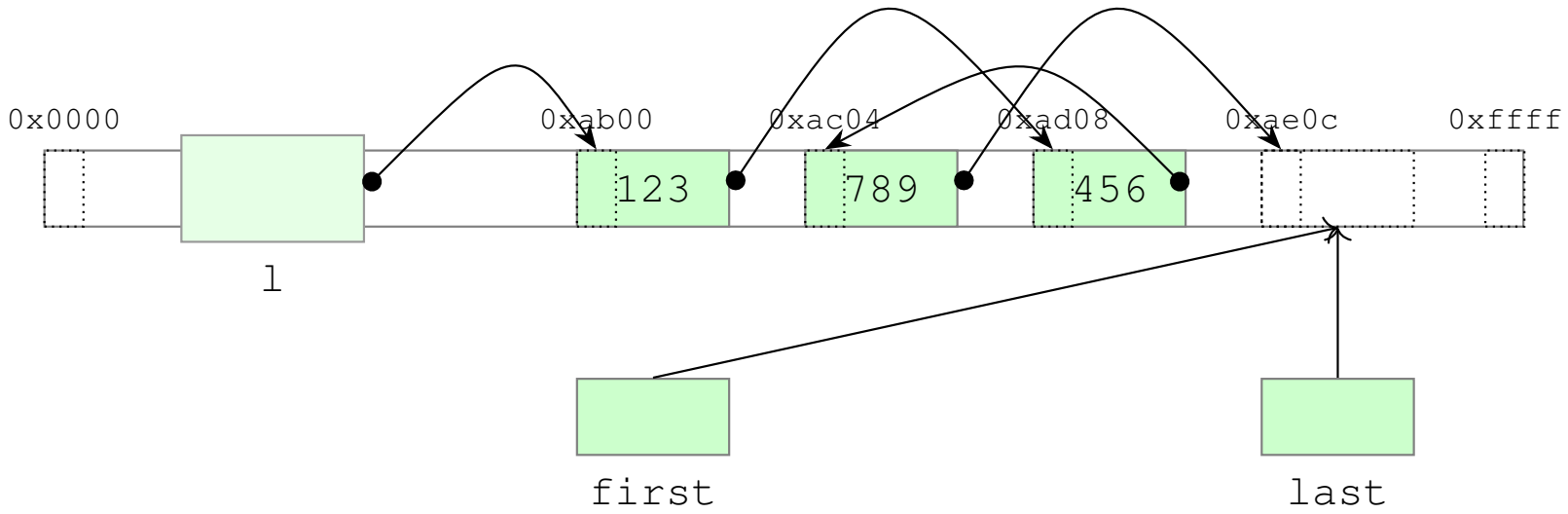

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)



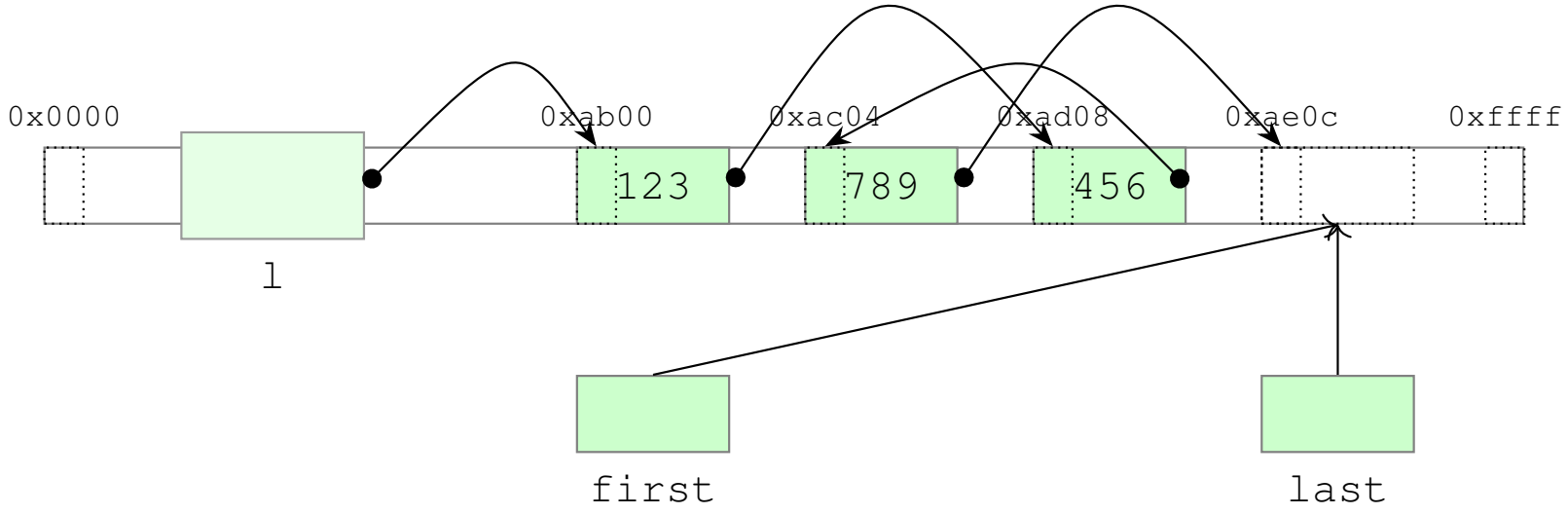

```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

Range (cont.)




```
std::forward_list<int> l = {123, 456, 789};
auto first = l.begin();
auto const last = l.end();
while (first != last) {
  ... *first ...;
  ++first;
}
```

- `std::forward_list<T>::iterators` models the *ForwardIterator* concept

Algorithms and ranges

- Examples

```
std::vector<int> v = { 2, 5, 4, 0, 1 };

// sort the first half of the vector in ascending order
std::sort(std::begin(v), std::begin(v) + v.size() / 2);

// sum up the vector elements, initializing the sum to 0
auto s = std::accumulate(std::begin(v), std::end(v), 0);

// append the partial sums of the vector elements into a list
std::list<int> l;
std::partial_sum(std::begin(v), std::end(v), std::back_inserter(l));

// find the first element with value 4
auto it = std::find(std::begin(v), std::end(v), 4);
```

- Some algorithms are customizable passing a function

```
auto s = accumulate(v.begin(), v.end(), std::string{"X"}, cat);
assert(s == "X-2-5-4-0-1");
```

Function objects

A mechanism to define *something-callable-like-a-function*

Function objects

A mechanism to define *something-callable-like-a-function*

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}


auto s = cat("XY", 5); // XY-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat{};
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = cat("XY", 5); // XY-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = Cat{}("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an `operator()`

```
auto cat(
  string const& s, int i
) {
  return s + '-' + to_string(i);
}



auto s = cat("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, cat
); // XY-2-3-5
```

```
struct Cat {
 auto operator() (
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

Cat cat;
auto s = Cat{}("XY", 5); // XY-5

vector<int> v{2,3,5};
auto s = accumulate(
    begin(v), end(v),
    string{"XY"}, Cat{}
); // XY-2-3-5
```

Function objects

A function object, being the instance of a class, can have state

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = cat2("XY", 5); // XY+5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = cat1("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = cat2("XY", 5); // XY+5

vector<int> v{2,3,5};
auto s3 = accumulate(..., cat1); // XY-2-3-5
auto s4 = accumulate(..., cat2); // XY+2+3+5
```

Function objects

A function object, being the instance of a class, can have state

```
class CatWithState {
  char c_;
 public:
  explicit CatWithState(char c) : c_{c} {}
  auto operator()(string const& s, int i) const {
    return s + c_ + to_string(i);
  }
};

CatWithState cat1{'-'};
auto s1 = CatWithState{'-'}("XY", 5); // XY-5

CatWithState cat2{'+'};
auto s2 = CatWithState{'+'}("XY", 5); // XY+5

vector<int> v{2,3,5};
auto s3 = accumulate(..., CatWithState{'-'}); // XY-2-3-5
auto s4 = accumulate(..., CatWithState{'+'}); // XY+2+3+5
```

Function objects (cont.)

An example from the standard library

```
#include <random>

// random bit generator (mersenne twister)
std::mt19937 gen;

// generate N 32-bit unsigned integer numbers
for (int n = 0; n != N; ++n) {
  std::cout << gen() << '\n';
}

// generate N floats distributed normally (mean: 0., stddev: 1.)
std::normal_distribution<float> dist;
for (int n = 0; n != N; ++n) {
  std::cout << dist(gen) << '\n';
}

// generate N ints distributed uniformly between 1 and 6 included
std::uniform_int_distribution<> roll_dice(1, 6);
for (int n = 0; n != N; ++n) {
  std::cout << roll_dice(gen) << '\n';
}
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});
```

```
accumulate(...,CatWithState{'-'});
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});

accumulate(...,
 [](string const& s, int i) {
   return s + '-' + to_string(i);
 }
);

accumulate(...,CatWithState{'-'});
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});

accumulate(...,
 [](string const& s, int i) {
   return s + '-' + to_string(i);
 }
);

accumulate(...,CatWithState{'-'});

char c{'-'};
accumulate(...,
 [=](string const& s, int i) {
  return s + c + to_string(i);
 }
);
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct Cat {
 auto operator()(
  string const& s, int i
 ) const {
  return s + '-' + to_string(i);
 }
};

class CatWithState {
 char c_;
public:
 explicit CatWithState(char c)
   : c_{c} {}
 auto operator()
 (string const& s, int i) const
 {return s + c_ + to_string(i);}
};
```

```
accumulate(..., Cat{});

accumulate(...,
 [](string const& s, int i) {
   return s + '-' + to_string(i);
 }
);

accumulate(...,CatWithState{'-'});


accumulate(...,
 [c = '-'](string const& s, int i) {
  return s + c + to_string(i);
 }
);
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = []
```

```
class SomeUniqueName {

 public:


  auto operator()

};


auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [](int  i)
{ return i + v; }
```

```
class SomeUniqueName {

 public:


  auto operator()(int i)
  { return i + v ; }
};


auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [](int  i)
{ return i + v; }
```

```
class SomeUniqueName {

 public:



  auto operator()(int i)
  { return i + v ; }
};

int v = 3;
auto l = SomeUniqueName{ };
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [=](int  i)
{ return i + v; }
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}

  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
int v = 3;

auto l = [=](int  i)
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}

  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [v = 3](int  i)
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}

  auto operator()(int i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [v = 3](auto i)
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  template<typename T>
  auto operator()(T   i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda expression (cont.)

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The `operator()` corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto l = [v = 3](auto i) -> int
{ return i + v; }

auto r = l(5); // 8
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  template<typename T>
  int  operator()(T   i)
  { return i + v_; }
};

int v = 3;
auto l = SomeUniqueName{v};
auto r = l(5); // 8
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [v] {};
```

```
class SomeUniqueName {
  int v_;
 public:
  explicit SomeUniqueName(int v)
    : v_{v} {}
  ...
};

auto l = SomeUniqueName{v};
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [&v] {};
```

```
class SomeUniqueName {
  int& v_;
 public:
  explicit SomeUniqueName(int& v)
    : v_{v} {}
  ...
};

auto l = SomeUniqueName{v};
```

Lambda: capturing

- Automatic variables used in the body need to be captured
  - `[]` capture nothing
  - `[=]` capture all by value
  - `[k]` capture `k` by value
  - `[&]` capture all by reference
  - `[&k]` capture `k` by reference
  - `[=, &k]` capture all by value but `k` by reference
  - `[&, k]` capture all by reference but `k` by value

```
int v = 3;
auto l = [&v] {};
```

```
class SomeUniqueName {
  int& v_;
 public:
  explicit SomeUniqueName(int& v)
    : v_{v} {}
  ...
};

auto l = SomeUniqueName{v};
```

- Global variables are available without being captured

Hands-on

- Try the code snippets
  - for a quick check of the syntax use `http://gcc.godbolt.org/`
- C++ and Memory $\rightarrow$ Algorithms
- Starting from `algo.cpp`, write code to
  - sum all the elements of the vector
  - multiply all the elements of the vector
  - sort the vector in ascending and descending order
  - move the even numbers to the beginning
  - move the three central numbers to the beginning
  - erase from the vector the elements that satisfy a predicate
  - ...
- How much is the overhead of a lambda?

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variables captured by value are not modifiable

```
[] {};
```

```
struct SomeUniqueName {
  auto operator()() const {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variables captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable {};
```

```
struct SomeUniqueName {
  auto operator()() {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variables captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable -> void {};
```

```
struct SomeUniqueName {
  void operator()() {}
};
```

Lambda: `const` and `mutable`

- By default the call to a lambda is `const`
  - Variables captured by value are not modifiable
- A lambda can be declared `mutable`

```
[]() mutable -> void {};
```

```
struct SomeUniqueName {
  void operator()() {}
};
```

- Variables captured by reference can be modified
  - There is no way to capture by `const&`

```
int v = 3;
[&v] { ++v; }();
assert(v == 4);
```

Lambda: dangling reference

Be careful not to have dangling references in a closure

- It's similar to a function returning a reference to a local variable

```
auto make_lambda()
{
  int v = 3;
  return [&] { return v; }; // return a closure
}

auto l = make_lambda();
auto d = l(); // the captured variable is dangling here
```

```
auto start_in_thread()
{
  int v = 3;
  return std::async([&] { return v; });
}
```

std::function

- Type-erased wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function
- Some space and time overhead, so use only if a template parameter is not satisfactory

std::function

- Type-erased wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function
- Some space and time overhead, so use only if a template parameter is not satisfactory

```
#include <functional>

int sum_squares(int x, int y) { return x * x + y * y; }

int main() {
  std::vector<std::function<int(int, int)>> v {
    std::plus<>{},        // has a compatible operator()
    std::multiplies<>{}, // idem
    &sum_squares)
  };
  for (int k = 10; k <= 1000; k *= 10) {
    v.push_back([k](int x, int y) -> int { return k * x + y; });
  }

  for (auto const& f : v) { std::cout << f(4, 5) << '\n'; }
}
```

Outline

Weaknesses of a `T*`

- Critical information is not encoded in the type
  - Am I the owner of the pointee? Should I delete it?
  - Is the pointee an object or an array of objects? of what size?
  - Was it allocated with `new`, `malloc` or even something else (e.g. `fopen` returns a `FILE*`)?

```
T* p = create_something();
```

Weaknesses of a `T*`

- Critical information is not encoded in the type
- Owning pointers are prone to leaks and double deletes

```
{
  T* p = new T{};
  ...
  // ops, forgot to delete p
}
{
  T* p = new T;
  ...
  delete p;
  ...
  delete p; // ops, delete again
}
```

Weaknesses of a `T*`

- Critical information is not encoded in the type
- Owning pointers are prone to leaks and double deletes
- Owning pointers are unsafe in presence of exceptions

```
{
  T* p = new T;
  ... // potentially throwing code
  delete p;
}
```

Weaknesses of a `T*`

- Critical information is not encoded in the type
- Owning pointers are prone to leaks and double deletes
- Owning pointers are unsafe in presence of exceptions
- Runtime overhead
  - dynamic allocation/deallocation
  - indirection

Debugging memory problems

- Valgrind is a suite of debugging and profiling tools for memory management, threading, caching, etc.
- Valgrind Memcheck can detect
  - invalid memory accesses
  - use of uninitialized values
  - memory leaks
  - bad frees
- It's precise, but slow

Debugging memory problems

- Valgrind is a suite of debugging and profiling tools for memory management, threading, caching, etc.
- Valgrind Memcheck can detect
  - invalid memory accesses
  - use of uninitialized values
  - memory leaks
  - bad frees
- It's precise, but slow

```
$ g++ leak.cpp
$ valgrind ./a.out
==18331== Memcheck, a memory error detector
...
```

Debugging memory problems (cont.)

- *Address Sanitizer* (ASan)
- The compiler instruments the executable so that at runtime ASan can catch problems similar, but not identical, to valgrind
- Faster than valgrind

Debugging memory problems (cont.)

- *Address Sanitizer* (ASan)
- The compiler instruments the executable so that at runtime ASan can catch problems similar, but not identical, to valgrind
- Faster than valgrind

```
$ g++ -fsanitize=address leak.cpp
$ ./a.out

=================================================================
==18338==ERROR: LeakSanitizer: detected memory leaks
...
```

Hands-on

- C++ and Memory $\rightarrow$ Memory issues
- Compile, run directly and run through valgrind and/or ASan
  - `non_owning_pointer.cpp`
  - `array_too_small.cpp`
  - `leak.cpp`
  - `double_delete.cpp`
  - `missed_delete.cpp`
- Try and fix the problems

When to use a `T*`

- To represent a *link* to an object when
  - the object is not owned, and
  - the link may be null or the link can be re-bound
- Mutable and immutable scenarios
  - `T*` vs `T const*`

When not to use a `T*`

- To represent a link to an object when
  - the object is owned, or
  - the link can never be null, and the link cannot be re-bound
- Alternatives
  - use a copy
  - use a (const) reference

```
T& tr = t1;  // tr is an alias for t1
tr = t2;     // doesn't re-bind tr, assigns t2 to t1

T* tp = &t1; // tp points to t1
tp = &t2;    // re-binds tp, it now points to t2
```

  - use a resource-managing object
    - `std::array`, `std::vector`, `std::string`, *smart pointers*, ...

Resource management

- Dynamic memory is just one of the many types of resources manipulated by a program:
  - thread, mutex, socket, file, ...
- C++ offers powerful tools to manage resources
  - *"C++ is my favorite garbage collected language because it generates so little garbage"*

Smart pointers

- Objects that behave like pointers, but also manage the lifetime of the pointee
- Leverage the RAII idiom
  - Resource Acquisition Is Initialization
  - Resource (e.g. memory) is acquired in the constructor
  - Resource (e.g. memory) is released in the destructor
- Importance of how the destructor is designed in C++
  - deterministic: guaranteed execution at the end of the scope
  - order of execution opposite to order of construction
- Guaranteed no leak nor double release, even in presence of exceptions

Smart pointers (cont.)

```
template<typename Pointee>
class SmartPointer {
  Pointee* m_p;
 public:
  explicit SmartPointer(Pointee* p): m_p{p} {}
  ~SmartPointer() { delete m_p; }



};

class Complex { ... };

{
  SmartPointer<Complex> sp{new Complex{1., 2.}};


}
```

At the end of the scope (i.e. at the closing `}`) `sp` is destroyed and its destructor `delete`s the pointee

Smart pointers (cont.)

```
template<typename Pointee>
class SmartPointer {
  Pointee* m_p;
 public:
  explicit SmartPointer(Pointee* p): m_p{p} {}
  ~SmartPointer() { delete m_p; }



};

class Complex { ... };

{
  SmartPointer<Complex> sp{new Complex{1., 2.}};
  sp->real();
  (*sp).real();
}
```

At the end of the scope (i.e. at the closing `}`) `sp` is destroyed and its destructor `delete`s the pointee

Smart pointers (cont.)

```
template<typename Pointee>
class SmartPointer {
  Pointee* m_p;
 public:
  explicit SmartPointer(Pointee* p): m_p{p} {}
  ~SmartPointer() { delete m_p; }
  Pointee* operator->() { return m_p; }
  Pointee& operator*() { return *m_p; }

};

class Complex { ... };

{
  SmartPointer<Complex> sp{new Complex{1., 2.}};
  sp->real();
  (*sp).real();
}
```

At the end of the scope (i.e. at the closing `}`) `sp` is destroyed and its destructor `delete`s the pointee

Smart pointers (cont.)

```
template<typename Pointee>
class SmartPointer {
  Pointee* m_p;
 public:
  explicit SmartPointer(Pointee* p): m_p{p} {}
  ~SmartPointer() { delete m_p; }
  Pointee* operator->() { return m_p; }
  Pointee& operator*() { return *m_p; }
  ...
};

class Complex { ... };

{
  SmartPointer<Complex> sp{new Complex{1., 2.}};
  sp->real();
  (*sp).real();
}
```

At the end of the scope (i.e. at the closing `}`) `sp` is destroyed and its destructor `delete`s the pointee

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_unique<X>(t1, t2)}; // better
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_unique<X>(t1, t2)}; // better
use(x);
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);      // by value

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_unique<X>(t1, t2)}; // better
use(x);                               // error, non-copyable
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_unique<X>(t1, t2)}; // better
use(x);                              // error, non-copyable
use(std::move(x));
```

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- No space nor time overhead
- Non-copyable, movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::unique_ptr<X> px);

std::unique_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_unique<X>(t1, t2)}; // better
use(x);                              // error, non-copyable
use(std::move(x));                   // ok, movable
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_shared<X>(t1, t2)}; // better
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_shared<X>(t1, t2)}; // better
use(x);
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_shared<X>(t1, t2)}; // better
use(x);                              // ok, copyable
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_shared<X>(t1, t2)}; // better
use(x);                              // ok, copyable
use(std::move(x));
```

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
  - for the management, not for access
- Copyable and movable

```
struct X {
  ...
  X(T1, T2);
};

void use(std::shared_ptr<X> px);

std::shared_ptr<X> x{new X{t1, t2}}; // explicit new
auto x{std::make_shared<X>(t1, t2)}; // better
use(x);                              // ok, copyable
use(std::move(x));                   // ok, movable
```

On *smart*_ptr<T>

- Prefer `unique_ptr` unless you need `shared_ptr`
  - You can always move a `unique_ptr` into a `shared_ptr`

On *smart*_ptr<T>

- Prefer unique_ptr unless you need shared_ptr
  - You can always move a unique_ptr into a shared_ptr
- unique_ptr supports also arrays, shared_ptr in part

```
std::unique_ptr<int[]> p{new int[n]};
auto p = std::make_unique<int[]>(n);
```

On *smart*_ptr<T>

- Prefer `unique_ptr` unless you need `shared_ptr`
  - You can always move a `unique_ptr` into a `shared_ptr`
- `unique_ptr` supports also arrays, `shared_ptr` in part

```
std::unique_ptr<int[]> p{new int[n]};
auto p = std::make_unique<int[]>(n);
```

- Access to the raw pointer is available
  - e.g. to pass to legacy APIs
  - *smart*_ptr<T>::get()
    - returns a **non-owning** `T*`
  - `unique_ptr<T>::release()`
    - returns an **owning** `T*`
    - must be explicitly managed

smart_ptr and functions

Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself

*smart*_ptr and functions

Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself

- by value of a unique_ptr, to transfer ownership

```
void take(std::unique_ptr<Thing> u);
auto u = std::make_unique<Thing>();
take(u);              // error
take(std::move(u)); // ok
```

*smart*_ptr and functions

Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself

- by value of a `unique_ptr`, to transfer ownership

```
void take(std::unique_ptr<Thing> u);
auto u = std::make_unique<Thing>();
take(u);              // error
take(std::move(u)); // ok
```

- by value of a `shared_ptr`, to keep the resource alive

```
auto s = std::make_shared<Thing>();
std::thread t{[=] { do_something_with(s); }};
```

smart_ptr and functions

Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself

- by value of a `unique_ptr`, to transfer ownership

```
void take(std::unique_ptr<Thing> u);
auto u = std::make_unique<Thing>();
take(u);             // error
take(std::move(u)); // ok
```

- by value of a `shared_ptr`, to keep the resource alive

```
auto s = std::make_shared<Thing>();
std::thread t{[=] { do_something_with(s); }};
```

- by reference, to interact with the smart pointer itself

```
void print_count(std::shared_ptr<Thing> const& s) {
  std::cout << s.use_count() << '\n';
};
auto s = std::make_shared<Thing>();
print_count(s);
```

smart_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

smart_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }



auto s = make_shared<Thing>();
```

smart_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }



auto s = make_shared<Thing>();
f(s);
```

smart_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }
void f(Thing* t)                 { if (t) t->g(); } // better


auto s = make_shared<Thing>();
f(s);
f(s->get());
```

smart_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }
void f(Thing* t)                 { if (t) t->g(); } // better
void f(Thing& t)                 { t.g(); }         // better

auto s = make_shared<Thing>();
f(s);
f(s->get());
if (s) f(*s);
```

*smart*_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }
void f(Thing* t)                 { if (t) t->g(); } // better
void f(Thing& t)                 { t.g(); }         // better

auto s = make_shared<Thing>();
f(s);
f(s->get());
if (s) f(*s);
```

- Return a *smart*_ptr from a function if the function has dynamically allocated a resource that is passed to the caller

```
auto factory() { return std::make_unique<Thing>(); }
```

*smart*_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }
void f(Thing* t)                 { if (t) t->g(); } // better
void f(Thing& t)                 { t.g(); }         // better

auto s = make_shared<Thing>();
f(s);
f(s->get());
if (s) f(*s);
```

- Return a *smart*_ptr from a function if the function has dynamically allocated a resource that is passed to the caller

```
auto factory() { return std::make_unique<Thing>(); }

auto u = factory();
std::shared_ptr<Thing> s = std::move(u);
```

*smart*_ptr and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
struct Thing { void g(); }
void f(std::shared_ptr<Thing> s) { if (s) s->g(); }
void f(Thing* t)                 { if (t) t->g(); } // better
void f(Thing& t)                 { t.g(); }         // better

auto s = make_shared<Thing>();
f(s);
f(s->get());
if (s) f(*s);
```

- Return a *smart*_ptr from a function if the function has dynamically allocated a resource that is passed to the caller

```
auto factory() { return std::make_unique<Thing>(); }

auto u = factory();
std::shared_ptr<Thing> s = std::move(u);

std::shared_ptr<Thing> s = factory();
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    int(*)(FILE*)
  >{
  std::fopen(...),
  &std::fclose
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    decltype(&std::fclose)
  >{
  std::fopen(...),
  &std::fclose
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<int(FILE*)>
  >{
  std::fopen(...),
  std::fclose
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  std::fclose
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  [](FILE* f){ std::fclose(f); }
};
```

*smart*_ptr custom deleter

- *smart*_ptr is a general-purpose resource handler
- The resource release is not necessarily done with delete
- unique_ptr and shared_ptr support a *custom deleter*

```
FILE* f = std::fopen(...);
...
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
auto s = std::shared_ptr<FILE>{
  std::fopen(...),
  std::fclose
};

auto u = std::unique_ptr<
    FILE,
    std::function<void(FILE*)>
  >{
  std::fopen(...),
  [](auto f){ std::fclose(f); }
};
```

Hands-on

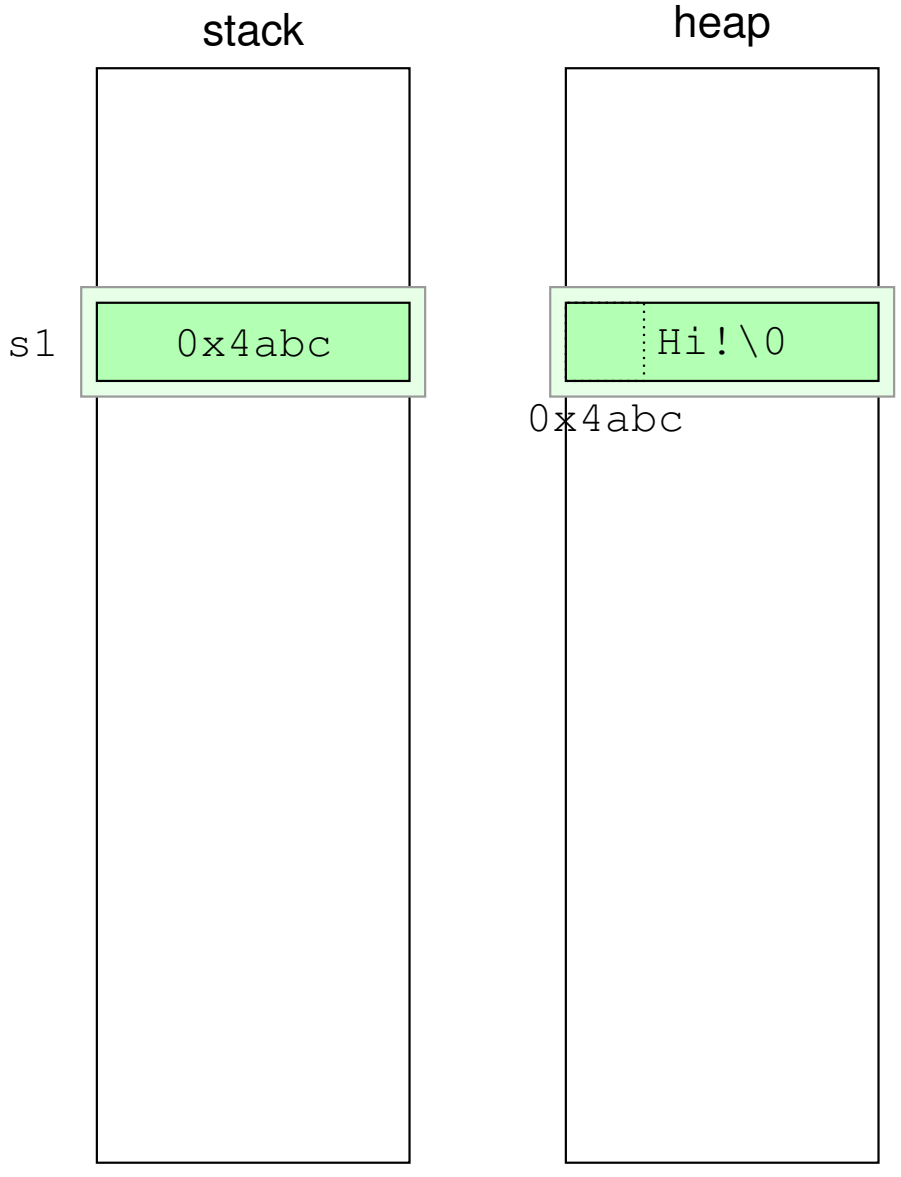
- Try the code snippets
- C++ and Memory $\rightarrow$ Memory issues
  - Adapt the exercises to use smart pointers
- C++ and Memory $\rightarrow$ Smart pointers
- Starting from `dir.cpp`, write code to:
  - create a smart pointer managing a `DIR` resource obtained with the `opendir` function call
  - associate a deleter to that smart pointer
  - implement a function to read the names of the files in that directory
  - check if the deleter is called at the right moment
  - hide the creation of the smart pointer behind a factory function
  - populate a vector of FILEs, properly wrapped in a smart pointer, obtained opening the regular files in that directory
  - ...

Outline

We can do better than copying

```
class String {
  char* s_;
  ...
};
```

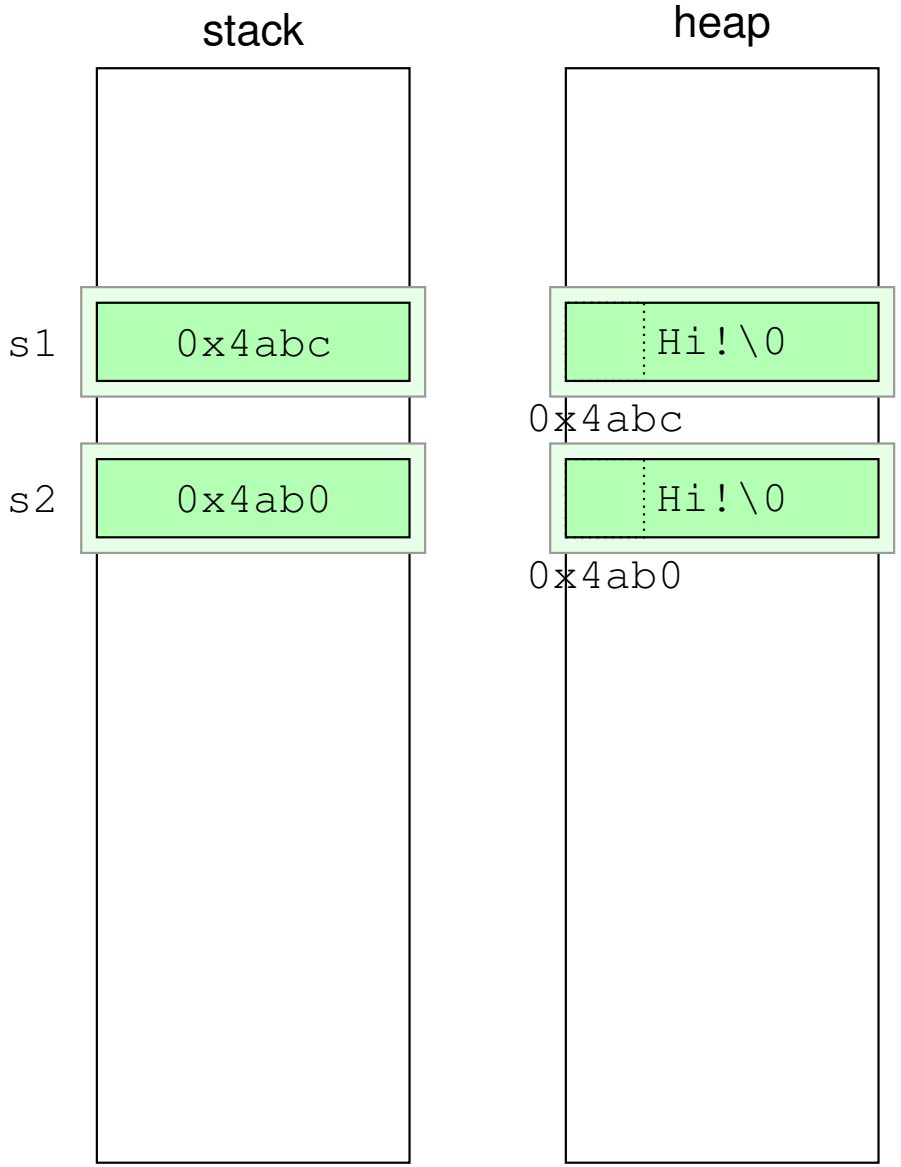
```
String s1{"Hi!"};
```



We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

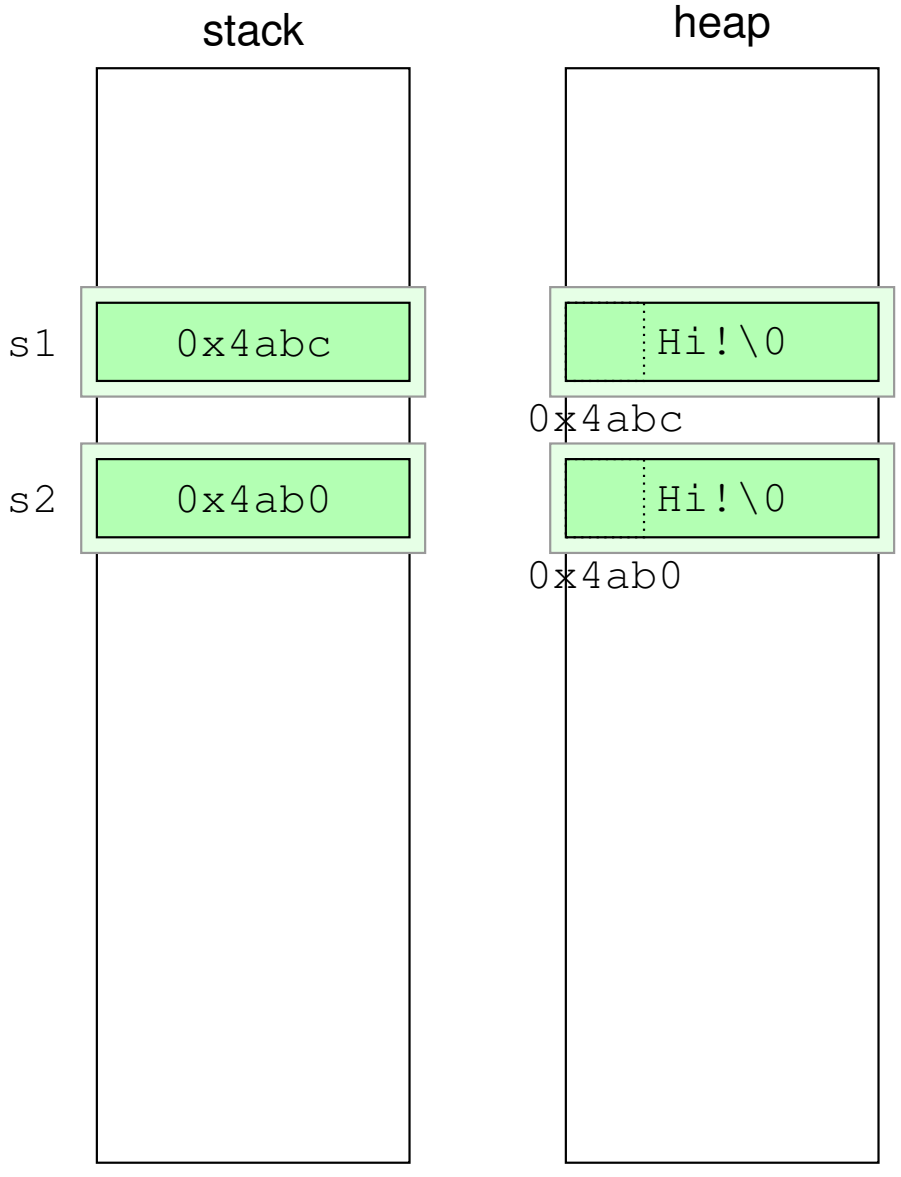


We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed



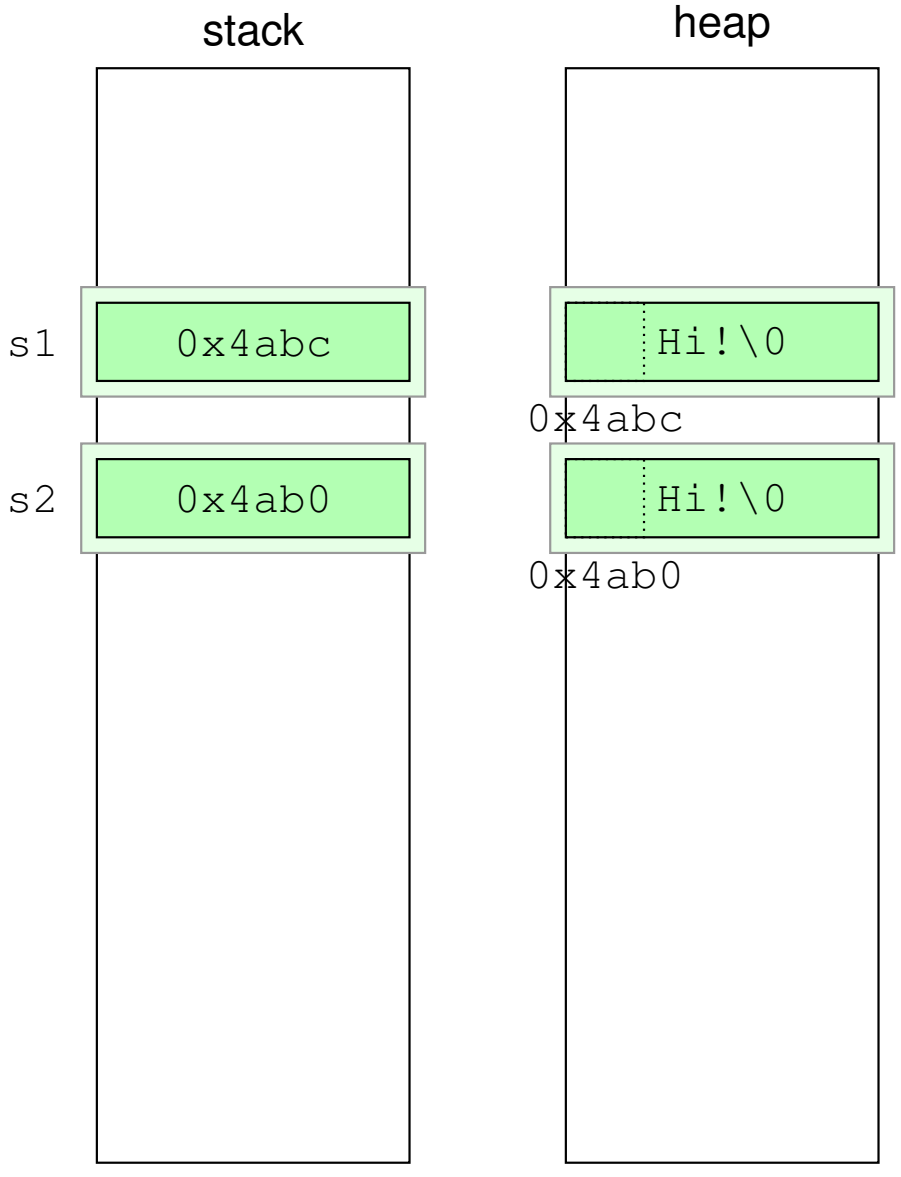
We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed

```
String get_string() { return "Hey"; }
String s3{get_string()};
```



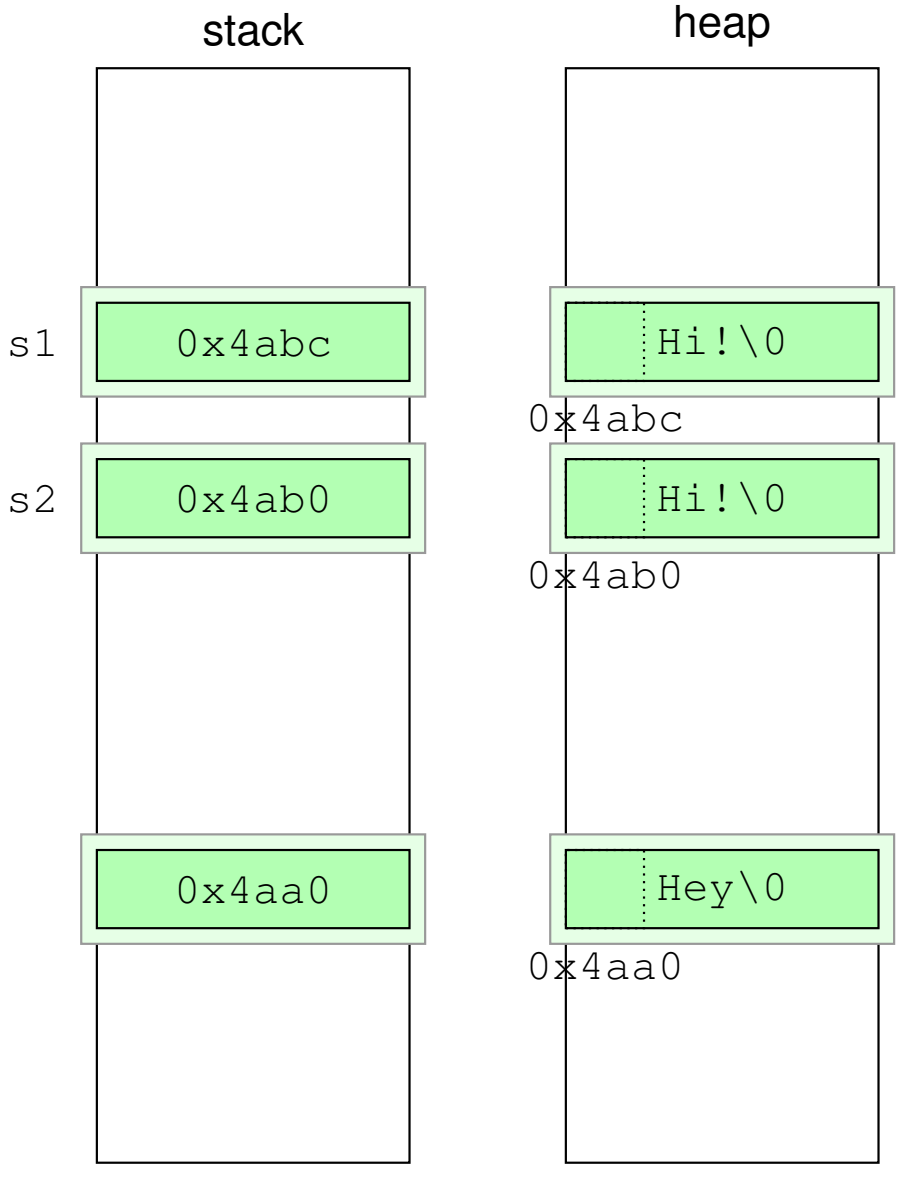
We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed

```
String get_string() { return "Hey"; }
String s3{get_string()};
```



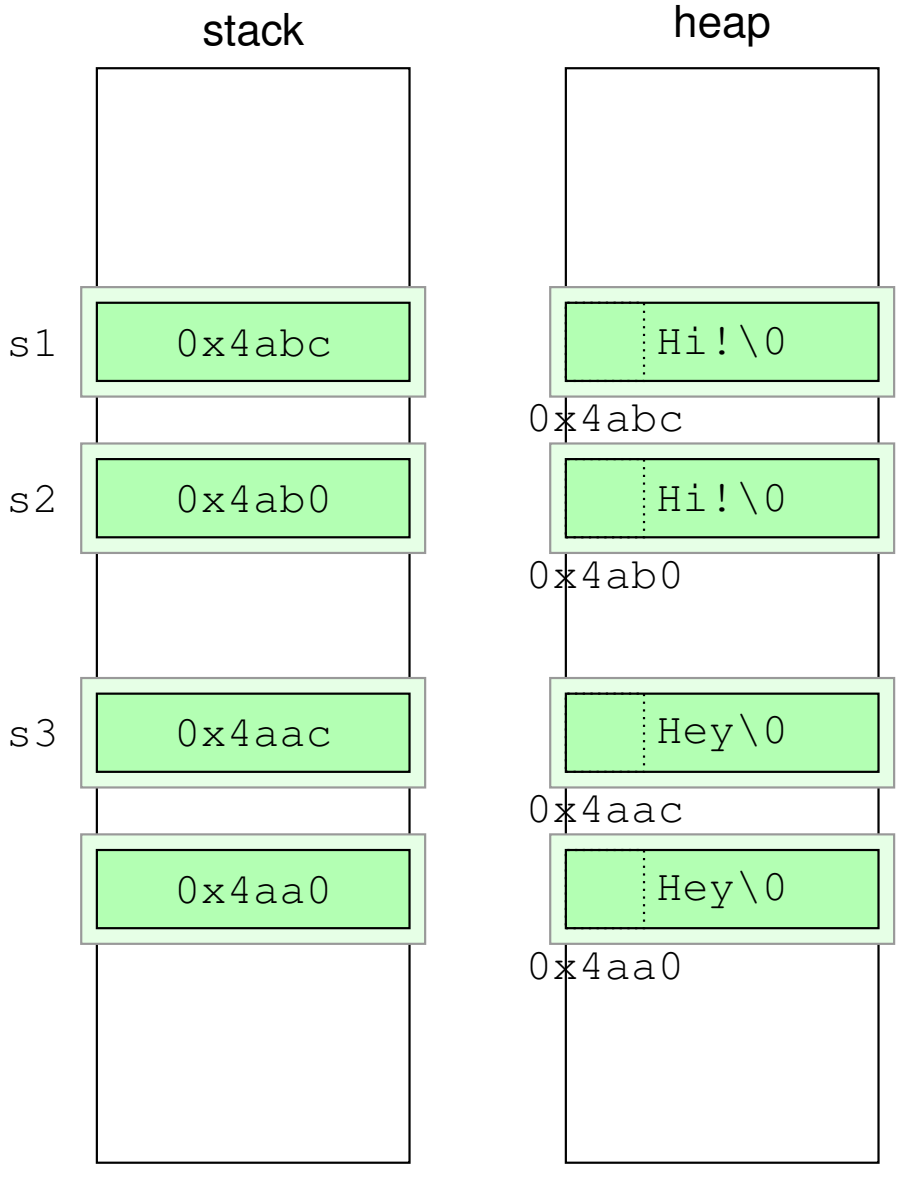
We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed

```
String get_string() { return "Hey"; }
String s3{get_string()};
```



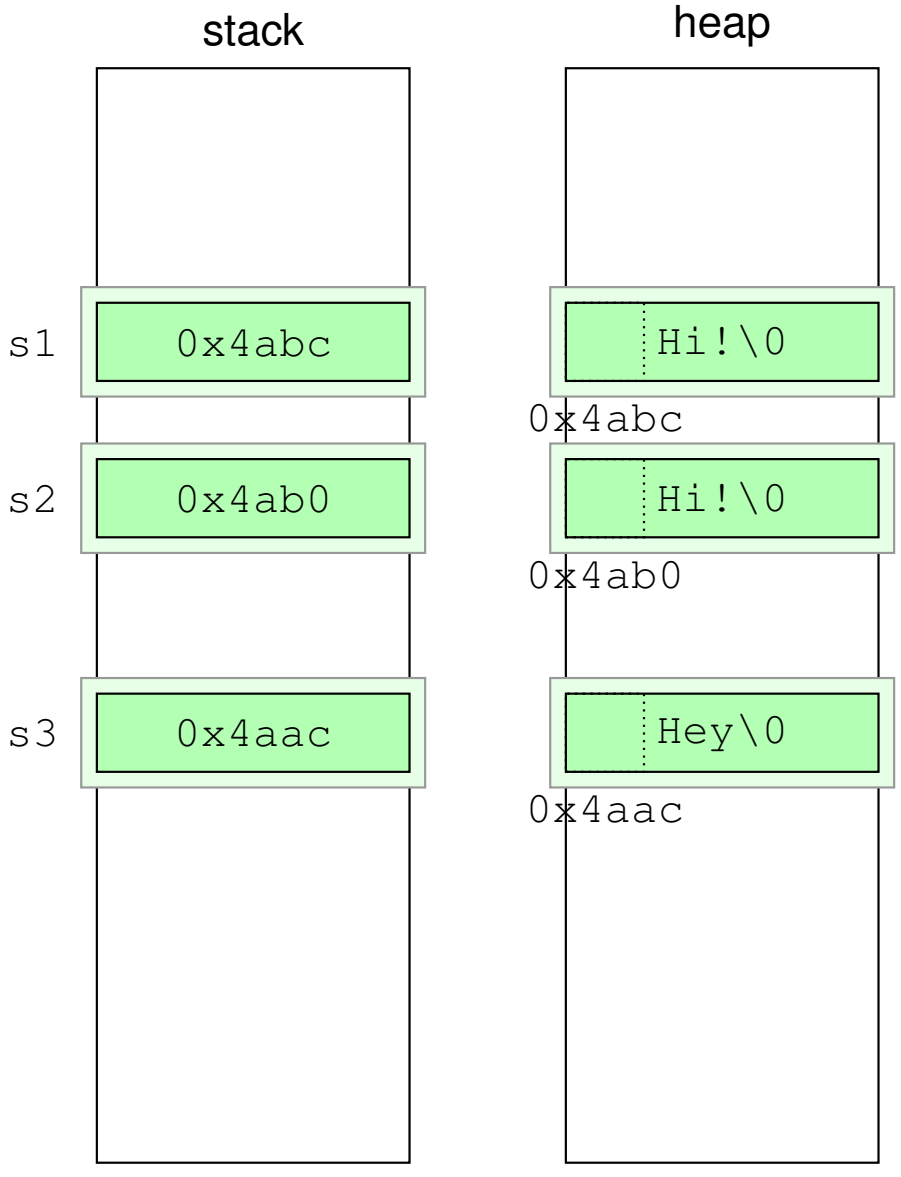
We can do better than copying

```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed

```
String get_string() { return "Hey"; }
String s3{get_string()};
```



We can do better than copying

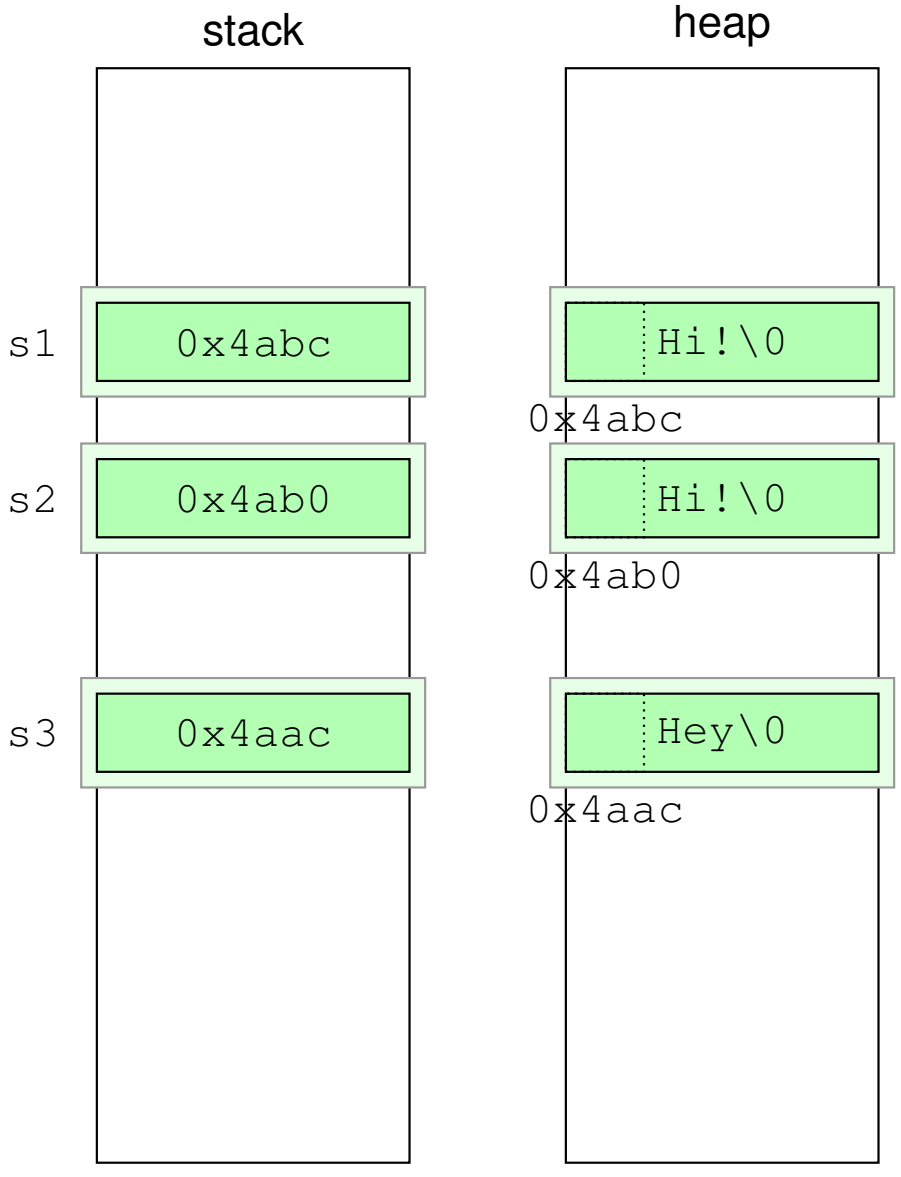
```
class String {
  char* s_;
  ...
};
```

```
String s1{"Hi!"};
String s2{s1};
```

- Both `s1` and `s2` exist at the end
- The “deep” copy is needed

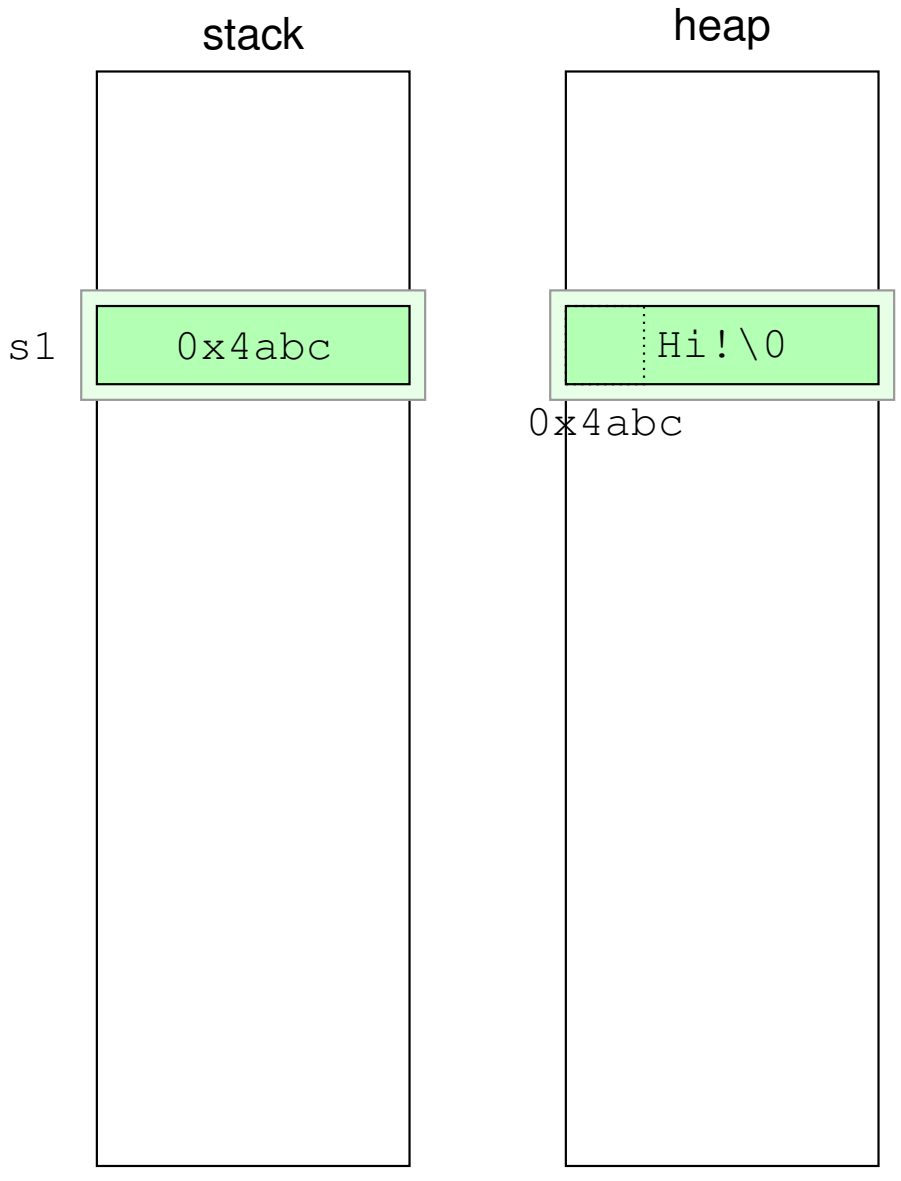
```
String get_string() { return "Hey"; }
String s3{get_string()};
```

- Only `s3` exists at the end
- The “deep” copy is a waste



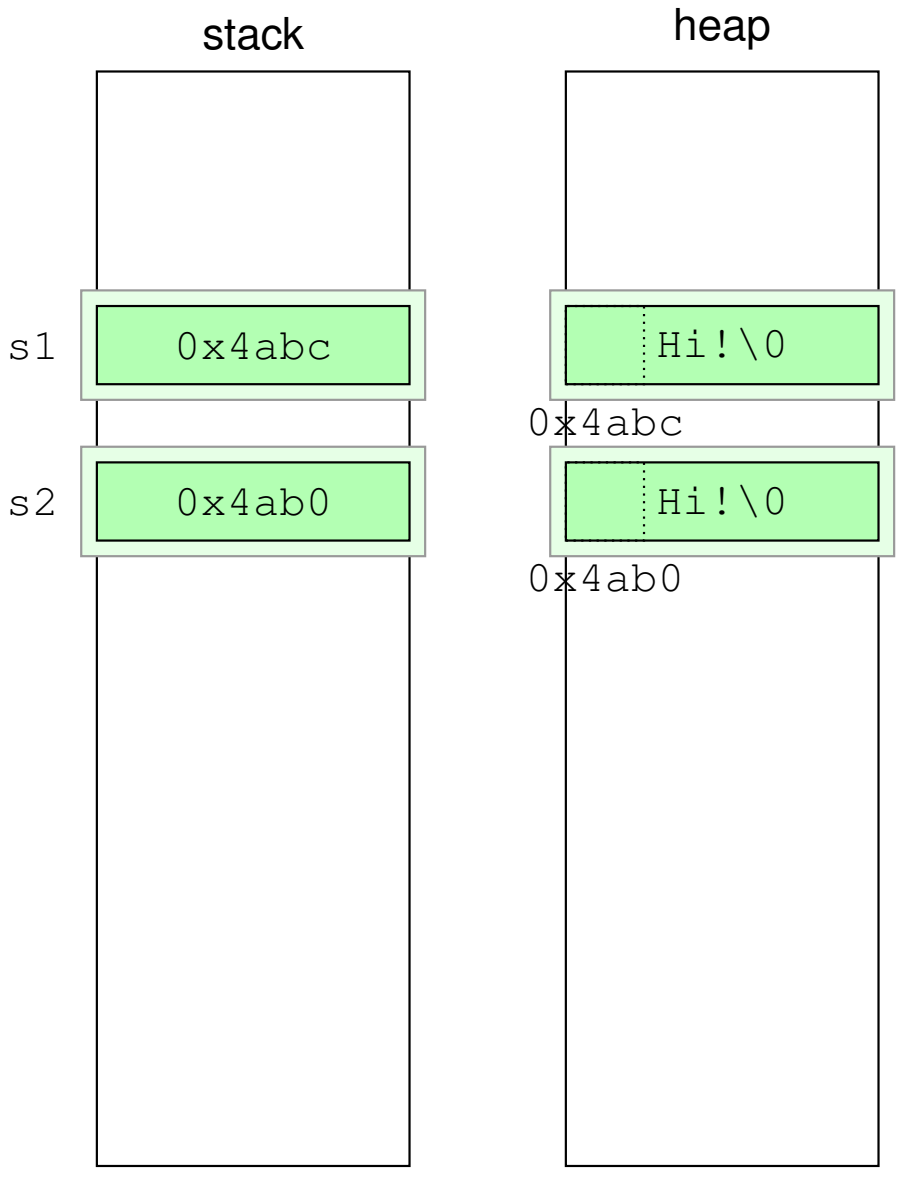
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }

  ...
};

String s1{"Hi!"};
```



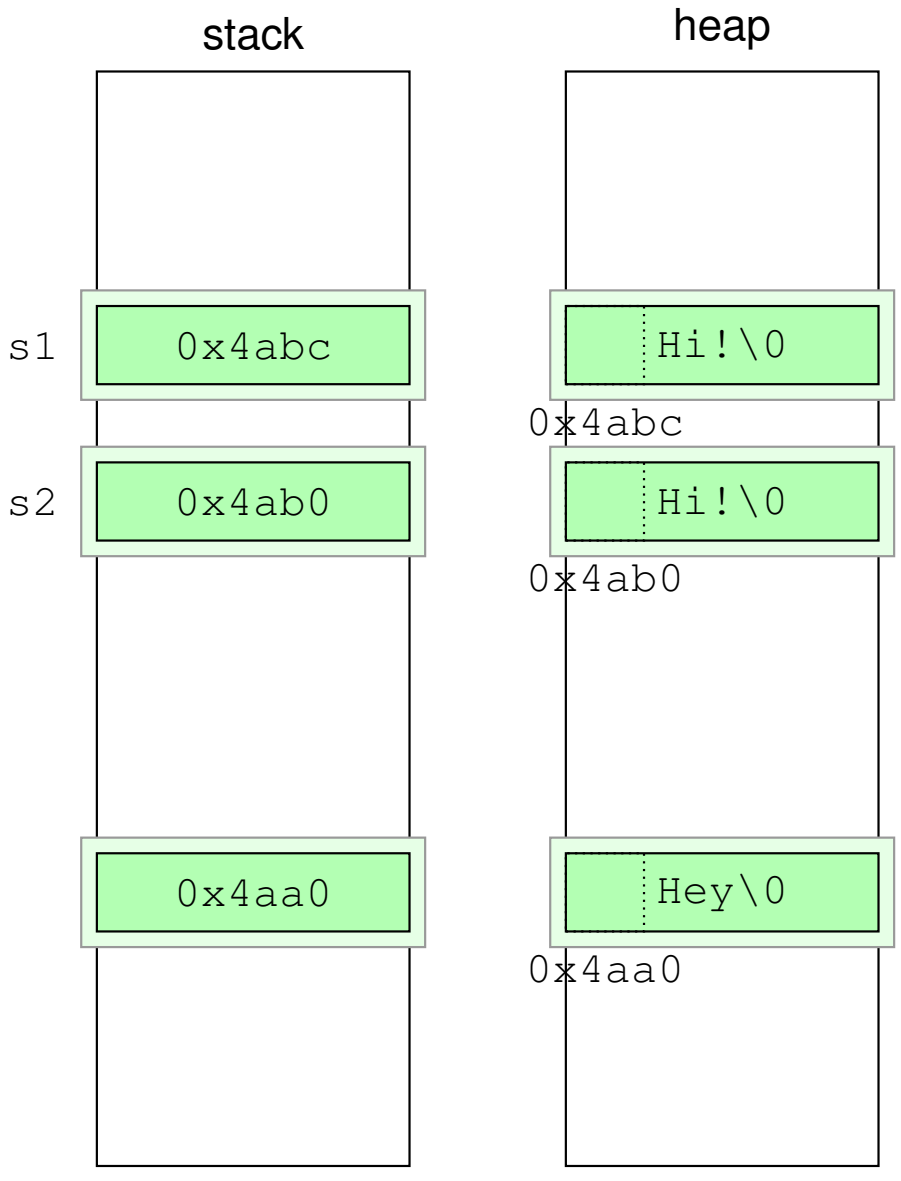
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }



  ...
};

String s1{"Hi!"};
String s2{s1};
```



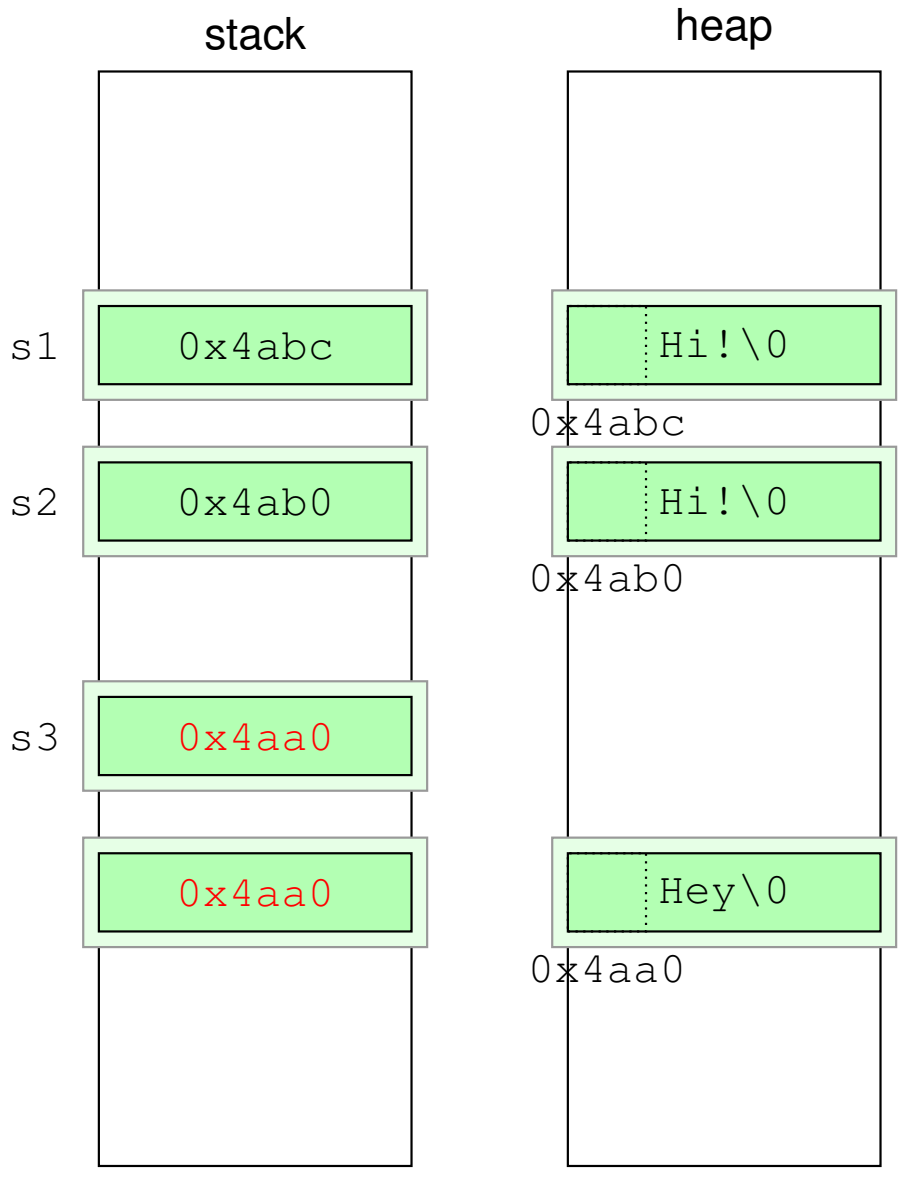
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }



  ...
};

String s1{"Hi!"};
String s2{s1};
String s3{get_string()};
```



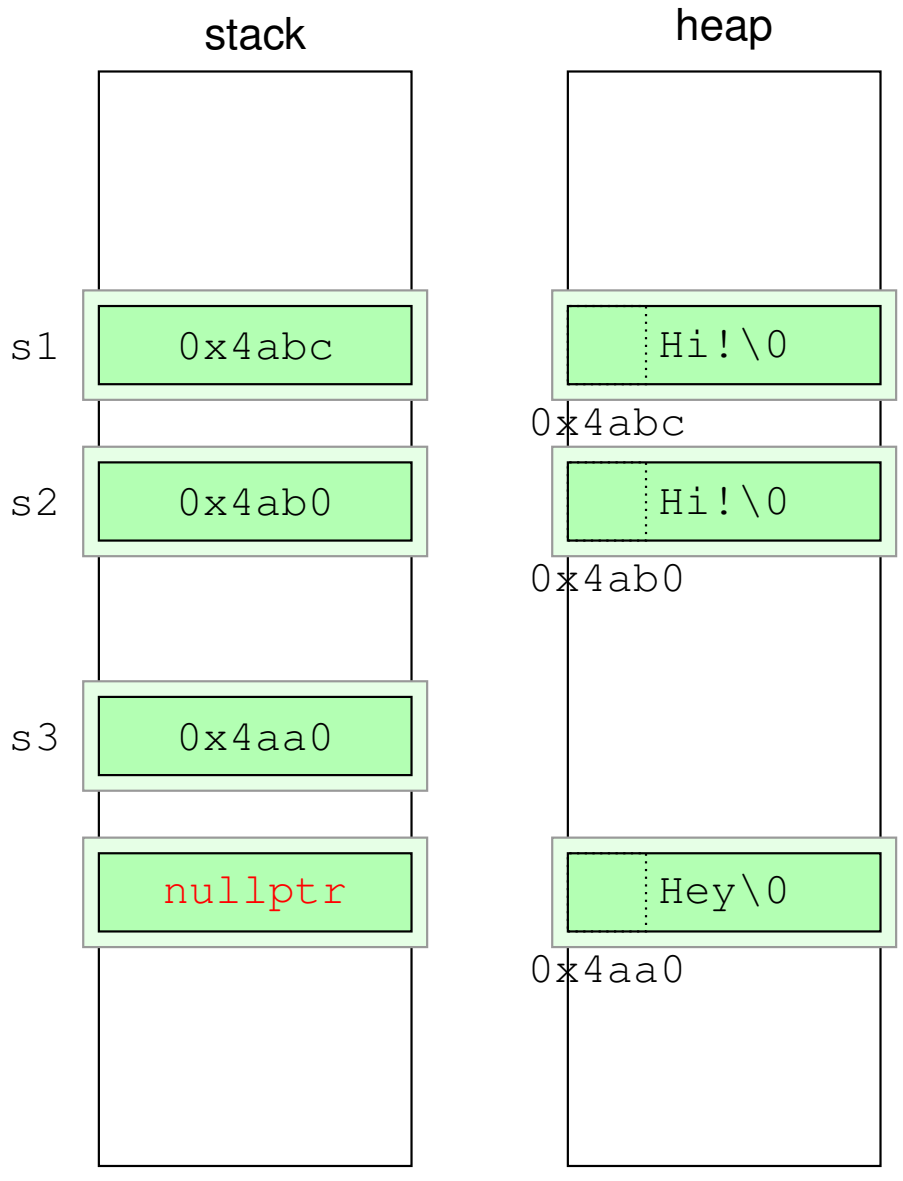
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {

  }
  ...
};

String s1{"Hi!"};
String s2{s1};
String s3{get_string()};
```



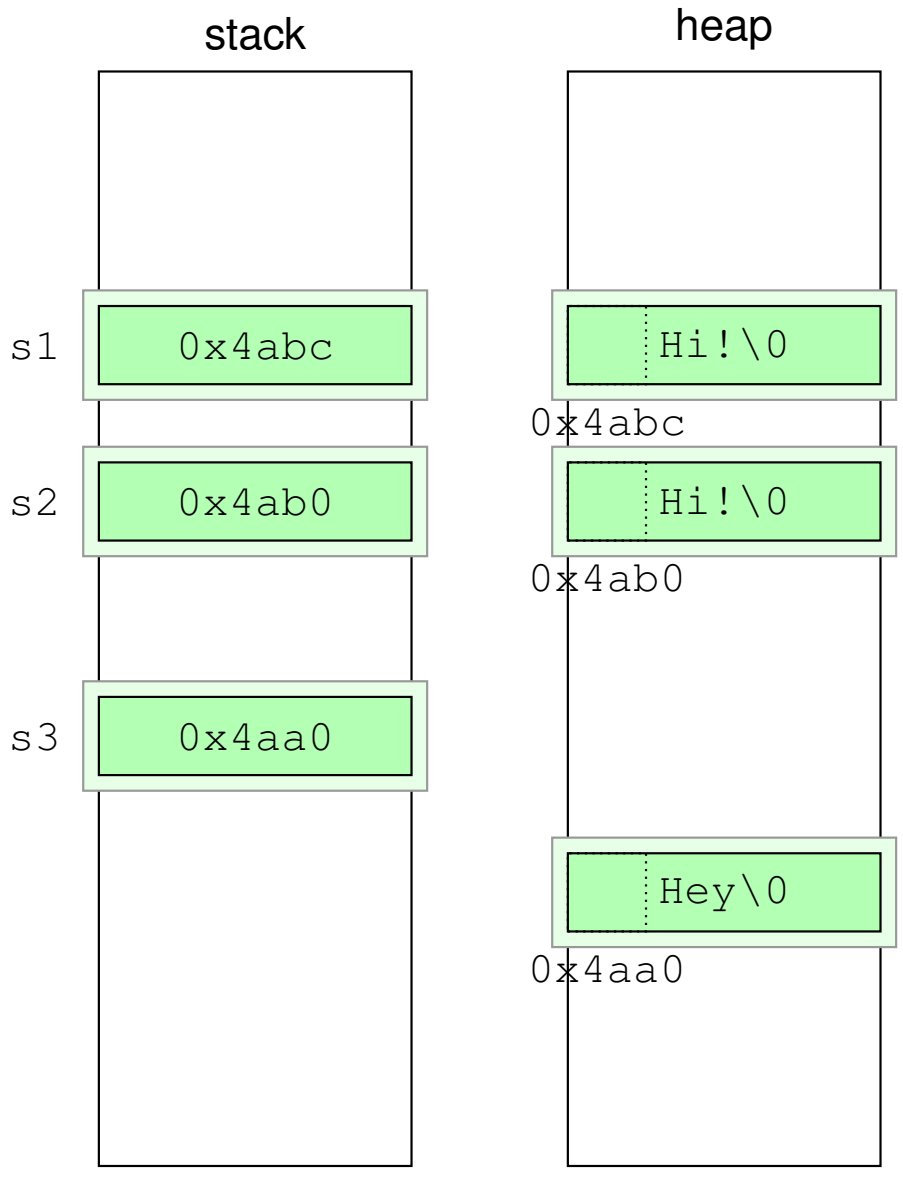
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  ...
};

String s1{"Hi!"};
String s2{s1};
String s3{get_string()};
```



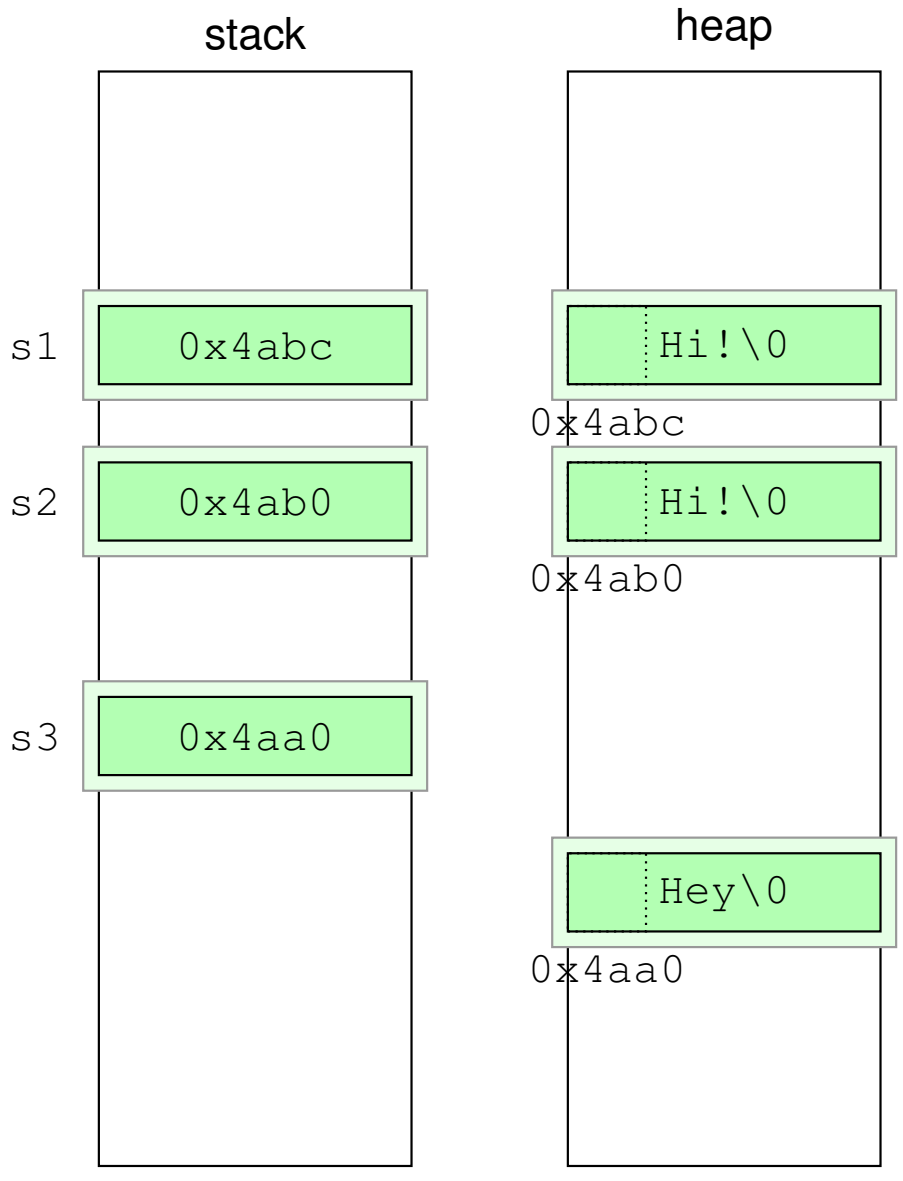
Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  ...
};

String s1{"Hi!"};
String s2{s1};
String s3{get_string()};
```



Copy vs move

```
class String {
  char* s_;
 public:
  String(char const* s) {
    size_t size = strlen(s) + 1;
    s_ = new char[size];
    memcpy(s_, s, size);
  }
  ~String() { delete [] s_; }
  // copy
  String(String const& other) {
    size_t size = strlen(other.s_) + 1;
    s_ = new char[size];
    memcpy(s_, other.s_, size);
  }
  // move
  String(??? tmp): s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
  ...
};

String s1{"Hi!"};
String s2{s1};
String s3{get_string()};
```



lvalues vs rvalues

- The taxonomy of values in C++ is complex
  - glvalue, prvalue, xvalue, lvalue, rvalue
- We can assume

  lvalue A named object
  - for which you can take the address
  - l stands for “left” because it used to represent the left-hand side of an assignment

  rvalue An unnamed (temporary) object
  - for which you can’t take the address
  - r stands for “right” because it used to represent the right-hand side of an assignment

Rvalue reference

- A `T&&` is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

Rvalue reference

- A `T&&` is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;
```

Rvalue reference

- A `T&&` is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &   r = t;
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
Thing      &  r = make_thing();
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
Thing      &  r = make_thing(); // error
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
Thing      &  r = make_thing(); // error
Thing      && r = make_thing();
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
Thing      &  r = make_thing(); // error
Thing      && r = make_thing(); // ok
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing       &  r = t;            // ok
Thing       && r = t;            // error
Thing       &  r = make_thing(); // error
Thing       && r = make_thing(); // ok
Thing const&  r = make_thing();
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing       &  r = t;            // ok
Thing       && r = t;            // error
Thing       &  r = make_thing(); // error
Thing       && r = make_thing(); // ok
Thing const&  r = make_thing(); // ok (!)
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing       &  r = t;            // ok
Thing       && r = t;            // error
Thing       &  r = make_thing(); // error
Thing       && r = make_thing(); // ok
Thing const&  r = make_thing(); // ok (!)
Thing const&& r = make_thing();
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing       &  r = t;            // ok
Thing       && r = t;            // error
Thing       &  r = make_thing(); // error
Thing       && r = make_thing(); // ok
Thing const&  r = make_thing(); // ok (!)
Thing const&& r = make_thing(); // ok, but what for?
```

Rvalue reference

- A **`T&&`** is an rvalue reference
  - introduced in C++11
- It binds to rvalues but not to lvalues

```
class Thing;
Thing make_thing();

Thing t;

Thing      &  r = t;            // ok
Thing      && r = t;            // error
Thing      &  r = make_thing(); // error
Thing      && r = make_thing(); // ok
Thing const&  r = make_thing(); // ok (!)
Thing const&& r = make_thing(); // ok, but what for?
```

```
class String {
  // move constructor
  String(String&& tmp) : s_(tmp.s_) {
    tmp.s_ = nullptr;
  }
};

String s2{s1};           // call String::String(String const&)
String s3{get_string()}; // call String::String(String&&)
```

Rvalue reference (cont.)

- Any function can accept rvalue references

```
void foo(String&&);

foo(get_string());
foo(String{"hello"});
```

- lvalues can be explicitly transformed into rvalues

```
String s;
foo(s);              // error
foo(std::move(s)); // ok, I don't care any more about s
s.size();            // dangerous
```

Overloading on &&

- A function can be overloaded for temporaries
  - useful if there are significant opportunities of optimization

```
void foo(Widget const&) {...}
void foo(Widget&&) {...}

Widget w{...};
foo(w);            // calls foo(Widget const&)
foo(Widget{...}); // calls foo(Widget&&)
```

Overloading on &&

- A function can be overloaded for temporaries
  - useful if there are significant opportunities of optimization

```
void foo(Widget const&) {...}
void foo(Widget&&) {...}

Widget w{...};
foo(w);            // calls foo(Widget const&)
foo(Widget{...}); // calls foo(Widget&&)
```

- For more than one parameter it becomes less desirable
  - consider pass by value, if move is cheap
  - especially useful for "sinks", e.g. in constructors

```
struct S {
  T1 t1_; T2 t2_;
  S(T1 t1, T2 t2) : t1_(std::move(t1)), t2_(std::move(t2)) {...}
};

T1 t1; T2 t2;
S s{t1, make_t2()};
S s{make_t1(), t2};
```

Special member functions

- A class has five special member functions
  - Plus the default constructor

```
class Widget {
  Widget(Widget const&);             // copy constructor
  Widget& operator=(Widget const&); // copy assignment
  Widget(Widget&&);                  // move constructor
  Widget& operator=(Widget&&);       // move assignment
  ~Widget();                         // destructor
};
```

Special member functions

- A class has five special member functions
  - Plus the default constructor

```
class Widget {
  Widget(Widget const&);             // copy constructor
  Widget& operator=(Widget const&); // copy assignment
  Widget(Widget&&);                  // move constructor
  Widget& operator=(Widget&&);       // move assignment
  ~Widget();                         // destructor
};
```

- The compiler can generate them automatically according to some convoluted rules
  - The behavior depends on the behavior of data members

Special member functions

- A class has five special member functions
  - Plus the default constructor

```
class Widget {
  Widget(Widget const&);              // copy constructor
  Widget& operator=(Widget const&); // copy assignment
  Widget(Widget&&);                   // move constructor
  Widget& operator=(Widget&&);        // move assignment
  ~Widget();                          // destructor
};
```

- The compiler can generate them automatically according to some convoluted rules
  - The behavior depends on the behavior of data members
- Rules of thumb

  Rule of zero Don't declare them and rely on the compiler

  Rule of five If you need to declare one, declare them all

  - Consider `= default` and `= delete`
  - But don't explicitly `= delete` move operations

Copy operations

```
class Widget {
  ...
  Widget(Widget const& other);
  Widget& operator=(Widget const& other);
};
```

Copy operations

```
class Widget {
  ...
  Widget(Widget const& other);
  Widget& operator=(Widget const& other);
};
```

copy constructor Allows the construction of an object as a copy of another object

```
Widget w1;
Widget w2{w1};
```

copy assignment Allows to change the value of an existing object as a copy of another object

```
Widget w1, w2;
w2 = w1;
```

Copy operations

```
class Widget {
  ...
  Widget(Widget const& other);
  Widget& operator=(Widget const& other);
};
```

copy constructor Allows the construction of an object as a copy of another object

```
Widget w1;
Widget w2{w1};
```

copy assignment Allows to change the value of an existing object as a copy of another object

```
Widget w1, w2;
w2 = w1;
```

- The two objects are/remain distinct
- The copied-from object is not changed
- After the copy the two objects should compare equal

Move operations

```
class Widget {
  ...
  Widget(Widget&& other);
  Widget& operator=(Widget&& other);
};
```

Move operations

```
class Widget {
  ...
  Widget(Widget&& other);
  Widget& operator=(Widget&& other);
};
```

move constructor Allows the construction of an object stealing the internals of another object

```
Widget w{make_widget()};
```

move assignment Allows to change the value of an existing object stealing the internals of another object

```
Widget w;
w = make_widget();
```

Move operations

```
class Widget {
  ...
  Widget(Widget&& other);
  Widget& operator=(Widget&& other);
};
```

move constructor Allows the construction of an object stealing the internals of another object

```
Widget w{make_widget()};
```

move assignment Allows to change the value of an existing object stealing the internals of another object

```
Widget w;
w = make_widget();
```

- The two objects are/remain distinct
- The moved-from object is usually changed
  - to a *valid but unspecified* state
  - it must be at least destructible and possibly reassignable

On move

- A move is typically cheaper than a copy, but it can be as expensive

On move

- A move is typically cheaper than a copy, but it can be as expensive
- If the *Return Value Optimization* is not applied, the return value of a function is moved, not copied, into destination

On move

- A move is typically cheaper than a copy, but it can be as expensive
- If the *Return Value Optimization* is not applied, the return value of a function is moved, not copied, into destination
- `operator=(T&&)` can assume that the argument is a temporary, hence different from `this`
  - There is no need to check for self-assignment
  - But be sure that in such event there is no crash
  - Rule of thumb: `std::swap` must work

```
template<typename T>
void swap(T& a, T& b) {
  T t{std::move(a)};
  a = std::move(b);
  b = std::move(t);
}
```

= default

- Explicitly tell the compiler to generate a special member function according to the default implementation

= default

- Explicitly tell the compiler to generate a special member function according to the default implementation

```
class Widget {
  int i = 0;
 public:
  Widget(Widget const&);

};

static_assert(std::is_copy_constructible<Widget>::value);
static_assert(!std::is_default_constructible<Widget>::value);
```

= default

- Explicitly tell the compiler to generate a special member function according to the default implementation

```
class Widget {
  int i = 0;
 public:
  Widget(Widget const&);
  Widget() = default;
};

static_assert(std::is_copy_constructible<Widget>::value);
static_assert(std::is_default_constructible<Widget>::value);
```

= delete

- A function can be declared as *deleted*, marking it with = delete

```
template<typename P>
class SmartPointer {
  ...
  SmartPointer(SmartPointer const&) = delete;
  SmartPointer& operator=(SmartPointer const&) = delete;
};
```

= delete

- A function can be declared as *deleted*, marking it with = delete
- For example, a class can be made non copyable deleting its copy operations

```
template<typename P>
class SmartPointer {
  ...
  SmartPointer(SmartPointer const&) = delete;
  SmartPointer& operator=(SmartPointer const&) = delete;
};

using SPI = SmartPointer<int>;

static_assert(!std::is_copy_constructible<SPI>::value);
static_assert(!std::is_copy_assignable<SPI>::value);
```

= delete

- A function can be declared as *deleted*, marking it with = delete
- For example, a class can be made non copyable deleting its copy operations
- Calling a deleted functions causes a compilation error

```
template<typename P>
class SmartPointer {
  ...
  SmartPointer(SmartPointer const&) = delete;
  SmartPointer& operator=(SmartPointer const&) = delete;
};

using SPI = SmartPointer<int>;

static_assert(!std::is_copy_constructible<SPI>::value);
static_assert(!std::is_copy_assignable<SPI>::value);

SPI sp1, sp2;
SPI sp3{sp1}; // error
```

= delete

- A function can be declared as *deleted*, marking it with = delete
- For example, a class can be made non copyable deleting its copy operations
- Calling a deleted functions causes a compilation error

```
template<typename P>
class SmartPointer {
  ...
  SmartPointer(SmartPointer const&) = delete;
  SmartPointer& operator=(SmartPointer const&) = delete;
};

using SPI = SmartPointer<int>;

static_assert(!std::is_copy_constructible<SPI>::value);
static_assert(!std::is_copy_assignable<SPI>::value);

SPI sp1, sp2;
SPI sp3{sp1}; // error
sp2 = sp1;    // error
```

= delete

- A function can be declared as *deleted*, marking it with = delete
- For example, a class can be made non copyable deleting its copy operations
- Calling a deleted functions causes a compilation error
- Any function can be deleted

```
template<typename P>
class SmartPointer {
  ...
  SmartPointer(SmartPointer const&) = delete;
  SmartPointer& operator=(SmartPointer const&) = delete;
};

using SPI = SmartPointer<int>;

static_assert(!std::is_copy_constructible<SPI>::value);
static_assert(!std::is_copy_assignable<SPI>::value);

SPI sp1, sp2;
SPI sp3{sp1}; // error
sp2 = sp1;    // error
```

Return a value from a function

- Returning a large value from a function is often perceived as slow

Return a value from a function

- Returning a large value from a function is often perceived as slow
  - Return “by pointer”

```
std::unique_ptr<LargeObject> make_large_object() {
  return std::make_unique<LargeObject>();
}

auto lo = make_large_object();
lo->...; // use the object, via a pointer
```

Return a value from a function

- Returning a large value from a function is often perceived as slow
  - Return “by pointer”

```
std::unique_ptr<LargeObject> make_large_object() {
  return std::make_unique<LargeObject>();
}

auto lo = make_large_object();
lo->...; // use the object, via a pointer
```

  - Use “out” arguments

```
void make_large_object(LargeObject& o) {
  o = LargeObject{};   // requires copy assignment
}

LargeObject lo;        // requires default constructor
make_large_object(lo);
lo....                 // use the object
```

Return a value from a function (cont.)

- There are very few reasons for not doing the obvious

```
LargeObject make_large_object() {
  return LargeObject{};
}

auto lo = make_large_object(); // possibly auto const
lo....                        // use the object
```

- In fact the compiler is allowed to elide the copy of the returned value into the final destination
  - (N)RVO – (Named) Return Value Optimization
  - If (N)RVO is not applied, C++11 mandates a move, if available
  - If the move is not available, copy; this may be really expensive

Return value optimization

Unnamed

```
Widget make_widget()
{
  if (...) {
    return Widget{};
  }
  return Widget{};
}

auto w = make_widget();
```

Named

```
Widget make_widget()
{
  Widget result;
  if (...) {
    result = Widget{};
  }
  return result;
}

auto w = make_widget();
```

Return value optimization

Unnamed

```
Widget make_widget()
{
  if (...) {
    return Widget{};
  }
  return Widget{};
}

auto w = make_widget();
```

Named

```
Widget make_widget()
{
  Widget result;
  if (...) {
    result = Widget{};
  }
  return result;
}

auto w = make_widget();
```

- Try not to mix named and unamed return statements in the same function
- Don't `return std::move(result)`

Hands-on

- C++ and Memory $\rightarrow$ Move
- Starting from the existing `string.cpp`, write code to
  - Complete the set of the special member functions so that `String` is copyable and movable
  - Add `operator[]` (const and non-const) to access a character at a given position
  - Add a `c_str()` member function to access the underlying C-style string
  - Use a smart pointer instead of a raw pointer (in the array form)
  - ...

Outline

Outline

Mechanisms for error management

The sooner the errors are identified, the better

- `static_assert`
  - Logical assertion that must be valid at compile time
- `assert`
  - Logical assertion that must be valid at run time
- Exceptions
  - To express an error condition happening at run time, typically related to a lack of resource
- C-style error codes
  - They can be ignored (but they should not!)
- ...

static_assert

Check that a certain constant boolean expression is satisfied during compilation

- If not, fail compilation with the specified message

static_assert

Check that a certain constant boolean expression is satisfied during compilation

- If not, fail compilation with the specified message

```
#include <type_traits>

struct C {
  C(C const&) = default;
  C& operator=(C const&) = delete;
};

static_assert(!std::is_default_constructible<C>::value, "");
static_assert( std::is_copy_constructible_v<C>);
static_assert(!std::is_copy_assignable_v<C>);
static_assert( std::is_move_constructible_v<C>);
static_assert(!std::is_move_assignable_v<C>);
static_assert( std::is_destructible_v<C>);
static_assert(sizeof(C) == 1);
```

static_assert

Check that a certain constant boolean expression is satisfied during compilation

- If not, fail compilation with the specified message

```
#include <type_traits>

struct C {
  C(C const&) = default;
  C& operator=(C const&) = delete;
};

static_assert(!std::is_default_constructible<C>::value, "");
static_assert( std::is_copy_constructible_v<C>);
static_assert(!std::is_copy_assignable_v<C>);
static_assert( std::is_nothrow_move_constructible_v<C>);
static_assert(!std::is_move_assignable_v<C>);
static_assert( std::is_destructible_v<C>);
static_assert(sizeof(C) == 1);
```

static_assert

Check that a certain constant boolean expression is satisfied during compilation

- If not, fail compilation with the specified message

```
#include <type_traits>

struct C {
  C(C const&) = default;
  C& operator=(C const&) = delete;
};

static_assert(!std::is_default_constructible<C>::value, "");
static_assert( std::is_copy_constructible_v<C>);
static_assert(!std::is_copy_assignable_v<C>);
static_assert( std::is_nothrow_move_constructible_v<C>);
static_assert(!std::is_move_assignable_v<C>);
static_assert( std::is_destructible_v<C>);
static_assert(sizeof(C) == 1);
```

A static assertion declaration can appear practically anywhere

- There is no effect, hence no overhead, at run time

`assert`

Check that a certain boolean expression is satisfied at run time

- If not satisfied, it means that the state of the program is corrupted $\rightarrow$ better to `terminate` as soon as possible

assert

Check that a certain boolean expression is satisfied at run time

- If not satisfied, it means that the state of the program is corrupted → better to `terminate` as soon as possible

```
template<class T> class Vector {
  T* p;
  ...
  T& operator[](int n) {


    return p[n];
  }
};
```

assert

Check that a certain boolean expression is satisfied at run time

- If not satisfied, it means that the state of the program is corrupted → better to `terminate` as soon as possible

```
template<class T> class Vector {
  T* p;
  ...
  T& operator[](int n) {
    assert(p != nullptr);          // class invariant (sort of)

    return p[n];
  }
};
```

assert

Check that a certain boolean expression is satisfied at run time

- If not satisfied, it means that the state of the program is corrupted → better to `terminate` as soon as possible

```
template<class T> class Vector {
  T* p;
  ...
  T& operator[](int n) {
    assert(p != nullptr);          // class invariant (sort of)
    assert(n >= 0 && n < size()); // function pre-condition
    return p[n];
  }
};
```

assert

Check that a certain boolean expression is satisfied at run time

- If not satisfied, it means that the state of the program is corrupted → better to `terminate` as soon as possible

```
template<class T> class Vector {
  T* p;
  ...
  T& operator[](int n) {
    assert(p != nullptr);          // class invariant (sort of)
    assert(n >= 0 && n < size()); // function pre-condition
    return p[n];
  }
};
```

Useful during testing/debugging

- Can be disabled for performance reasons (`-DNDEBUG`)
- Avoid side effects in `asserts`

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};


auto make_thing() {




  return Thing{    };
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};


auto make_thing() {
  auto res = acquire_resources_to_build_thing();




  return Thing{res};
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};


auto make_thing() {
  auto res = acquire_resources_to_build_thing();
  if (!success(res)) {


  }
  return Thing{res};
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};
class Exception {...};

auto make_thing() {
  auto res = acquire_resources_to_build_thing();
  if (!success(res)) {
    Exception e{...};
    throw e;
  }
  return Thing{res};
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};
class Exception {...};

auto make_thing() {
  auto res = acquire_resources_to_build_thing();
  if (!success(res)) {
    Exception e{...};
    throw e;
  }
  return Thing{res}; // not executed in case of exception
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};
class Exception {...};

auto make_thing() {
  auto res = acquire_resources_to_build_thing();
  if (!success(res)) {

    throw Exception{...};
  }
  return Thing{res}; // not executed in case of exception
}
```

Exceptions

- Mechanism to report errors out of a function, stopping its execution
- Useful to express *post-conditions*
- Help separate application logic from error management

```
class Thing {...};
class Exception {...};

auto make_thing() {
  auto res = acquire_resources_to_build_thing();
  if (!success(res)) {

    throw Exception{...};
  }
  return Thing{res}; // not executed in case of exception
}
```

Note that all local variables (e.g. `res`) are properly destroyed when exiting the function, be it via `return` or via `throw`

Exception propagation

```
auto high() {

    mid();



}
auto mid() {
  low();

}
auto low() {



}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {

  low();


}

auto low() {



}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();


}

auto low() {



}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();


}

auto low() {
  // this part is executed


}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();


}

auto low() {
  // this part is executed
  throw E{};

}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();


}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed

}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {

    // this part is executed
    mid();




}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {
  try {
    // this part is executed
    mid();

  } catch (E& e) {

  }
}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {
  try {
    // this part is executed
    mid();
    // this part is not executed
  } catch (E& e) {

  }
}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {
  try {
    // this part is executed
    mid();
    // this part is not executed
  } catch (E& e) {
    // use e
  }
}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {
  try {
    // this part is executed
    mid();
    // this part is not executed
  } catch (E& e) { // by reference
    // use e
  }
}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

Exception propagation

```
auto high() {
  try {
    // this part is executed
    mid();
    // this part is not executed
  } catch (E& e) { // by reference
    // use e
  }
}

auto mid() {
  T t; // this part is executed
  low();
  // this part is not executed
  // T is properly destroyed
}

auto low() {
  // this part is executed
  throw E{};
  // this part is not executed
}
```

- An exception is propagated up the stack of function calls until a suitable catch clause is found
- If no suitable catch clause is found the program is `terminate`d
- During stack unwinding all automatic objects are properly destroyed
  - Remember smart pointers!

Exception safety

Different levels of safety guarantees (for member functions):

basic If an exception is thrown, no resource is leaked and the object is left in a *valid but unspecified* state

- the object should be at least safely assignable and destroyable
- every class should provide at least the basic guarantee

strong Transaction semantics: if an exception is thrown, the object's state is as it was before the function was called

no-throw The operation is always successful and no exception leaves the function

`noexcept`

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or

`noexcept`

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage exceptions

`noexcept`

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage exceptions $\rightarrow$ better `terminate`

noexcept

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage exceptions → better `terminate`

```
class Handle {
  Handle(Handle&& o) noexcept : ... { ... }
  ...
};
```

noexcept

- A function can be declared `noexcept`, telling the compiler that the function
  - doesn't throw, or
  - is not able to manage exceptions → better `terminate`

```
class Handle {
  Handle(Handle&& o) noexcept : ... { ... }
  ...
};
```

- Declaring functions (not only member functions) `noexcept` helps the compiler to optimize the code
- If move operations, especially the constructor, are `noexcept` the compiler/library can apply **significant** optimizations
  - E.g. in order to provide the strong guarantee `std::vector::push_back` must copy, not move, objects, if the move can throw

Move and `noexcept`

- `T& T::operator=(T&& tmp)` is typically easy to make `noexcept`

Move and `noexcept`

- `T& T::operator=(T&& tmp)` is typically easy to make `noexcept`
  - Rely on the `noexcept`-ness of data members' move-assignments

Move and `noexcept`

- `T& T::operator=(T&& tmp)` is typically easy to make `noexcept`
  - Rely on the `noexcept`-ness of data members' move-assignments
- `T::T(T&& tmp)` may be more difficult
  - Start with one object (`tmp`), end up with two (`*this` and `tmp`)

Move and `noexcept`

- `T& T::operator=(T&& tmp)` is typically easy to make `noexcept`
  - Rely on the `noexcept`-ness of data members' move-assignments
- `T::T(T&& tmp)` may be more difficult
  - Start with one object (`tmp`), end up with two (`*this` and `tmp`)
  - Can rely on `T::T()` being `noexcept` as well
  - Which is not obvious if a resource has to be acquired

Destructor and `noexcept`

- The destructor is by default `noexcept`
  - i.e. releasing a resource should not fail

Destructor and `noexcept`

- The destructor is by default `noexcept`
  - i.e. releasing a resource should not fail
- Don't do anything overly complicated in the destructor or swallow all exceptions locally

Destructor and `noexcept`

- The destructor is by default `noexcept`
  - i.e. releasing a resource should not fail
- Don't do anything overly complicated in the destructor or swallow all exceptions locally

```
class Thing {
  ~Thing()
  {
    try {
      :
    } catch (...) { // catch all exceptions
      // e.g. log something, provided logging doesn't throw
    }
  }
};
```

Destructor and `noexcept`

- The destructor is by default `noexcept`
  - i.e. releasing a resource should not fail
- Don't do anything overly complicated in the destructor or swallow all exceptions locally

```
class Thing {
  ~Thing()
  {
    try {
      :
    } catch (...) { // catch all exceptions
      // e.g. log something, provided logging doesn't throw
    }
  }
};
```

- It's always possible to declare a destructor, like any other function, `noexcept(false)`

Hands-on

- C++ and Memory $\rightarrow$ Contracts
- Identify
  - the invariant of the String class
  - the pre-conditions and the post-condition of its member functions (when applicable)
  - other logical checkpoints both at compile time and run time
- Express them with `static_assert`s, `assert`s and exceptions, including `noexcept`
- Hint: imagine to remove the default constructor of `String` and see what the consequences are

Hands-on

- C++ and Memory $\rightarrow$ Static analysis