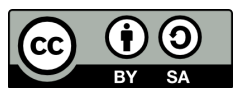
Efficient C++ Programming

F. Giacomini

INFN-CNAF

ESC@SESAME 2026 — SESAME, 1–6 February 2026



Outline

Outline

C++ question #1

Consider the following function:

```
auto Sum(std::vector<int> const& v)
{
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != v.size(); ++i) {
    sum += v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` is invoked passing a vector containing the values `{1, 2, 3}`?

C++ question #1

Consider the following function:

```
auto Sum(std::vector<int> const& v)
{
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != v.size(); ++i) {
    sum += v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` *is invoked passing a vector containing the values* `{1, 2, 3}`*?*

A: It doesn't compile because we are trying to modify `v`, which is a const (reference to a) vector

C++ question #2

Consider the following function:

```
auto Sum(std::vector<int>& v)
{
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != v.size(); ++i) {
    sum += v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` is invoked passing a vector containing the values `{1, 2, 3}`?

C++ question #2

Consider the following function:

```
auto Sum(std::vector<int>& v)
{
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != v.size(); ++i) {
    sum += v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` *is invoked passing a vector containing the values* `{1, 2, 3}`*?*

A: The code compiles and the function returns 10. `v` is a non-const (reference to a) vector, so it can be changed.

C++ question #3

Consider the following function:

```
auto Sum(std::vector<int>& v)
{
  auto another_v = v;
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != another_v.size(); ++i) {
    sum += another_v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` is invoked passing a vector containing the values `{1, 2, 3}`?

C++ question #3

Consider the following function:

```
auto Sum(std::vector<int>& v)
{
  auto another_v = v;
  v.push_back(4);
  auto sum = 0;
  for (int i = 0; i != another_v.size(); ++i) {
    sum += another_v[i];
  }
  return sum;
}
```

Does it compile? If it compiles, what is the returned value if `Sum` is invoked passing a vector containing the values `{1, 2, 3}`?

A: The code compiles and the function returns 6. `another_v` is a **copy** of `v`; changing `v` doesn't affect `another_v`.

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

Learn more

- We'll just touch a few subjects about the C++ language and its standard library

Learn more

- We'll just touch a few subjects about the C++ language and its standard library
- There are many high-quality resources to go deeper
  - Learn C++
  - News, Status & Discussion about Standard C++
  - C++ reference
  - C++ Core Guidelines
  - C++ Conference (presentations, videos)
  - C++ Now Conference (presentations, videos)
  - Meeting C++ Conference (presentations, videos)
  - Existing source code (e.g. boost libraries)

Standards

- A new standard is published every three years.
- *Working drafts*, almost the same as the final published document

  C++03 `https://wg21.link/n1905`
  C++11 `https://wg21.link/std11`
  C++14 `https://wg21.link/std14`
  C++17 `https://wg21.link/std17`
  C++20 `https://wg21.link/std20`
  C++23 `https://wg21.link/std23`
  C++26 `https://wg21.link/std` (current draft)

  LaTeXsources at `https://github.com/cplusplus/draft`
- *Working papers* at

  `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`

Compilers

- The ESC machines provide at least two C++ compilers: use gcc 14.2.1 (see instructions on how to enable it), but the default gcc 11.5 is also available
- You can also edit and try your code online with multiple compilers at
  - `https://godbolt.org/`
  - `https://coliru.stacked-crooked.com/`
  - `https://wandbox.org/`

Outline

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - mathematical functions
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...

The C++ standard library

- The standard library contains components of general use
  - containers (data structures)
  - algorithms
  - strings
  - input/output
  - mathematical functions
  - random numbers
  - regular expressions
  - concurrency and parallelism
  - filesystem
  - ...
- The subset containing containers and algorithms is known as STL (Standard **Template** Library)
  - But templates are everywhere

Containers of objects

- A program often needs to manage collections of objects
  - e.g. a string of characters, a dictionary of words, a list of particles, a matrix, ...

Containers of objects

- A program often needs to manage collections of objects
  - e.g. a string of characters, a dictionary of words, a list of particles, a matrix, ...
- A *container* is an object that contains other objects
- The C++ Standard Library provides a few container classes
  - implemented as class templates
  - different characteristics and operations, some common traits

Dynamic memory allocation

- Objects are preferably created on the *stack*
- However, it's not always possible to know at compile time which type of object is needed or how many of them

Dynamic memory allocation

- Objects are preferably created on the *stack*
- However, it's not always possible to know at compile time which type of object is needed or how many of them
  - run-time polymorphism

```
struct Shape { ··· };
struct Rectangle : Shape { ··· };
struct Circle : Shape { ··· };

Shape* s{nullptr};
char c; std::cin >> c;
switch (c) {
 case 'r': s = new Rectangle{···}; break;
 case 'c': s = new Circle{···}; break;
}
```

Dynamic memory allocation

- Objects are preferably created on the *stack*
- However, it's not always possible to know at compile time which type of object is needed or how many of them
  - run-time polymorphism

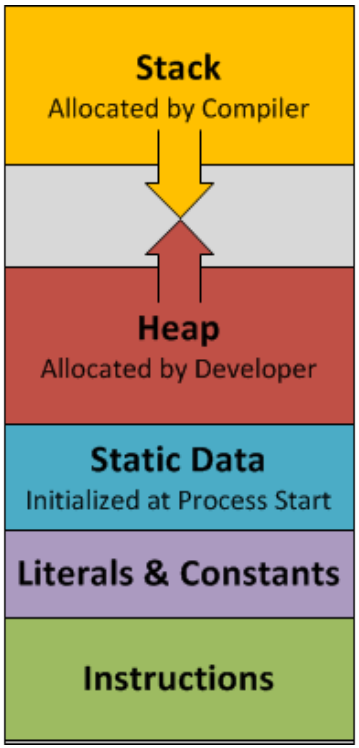
```
struct Shape { ··· };
struct Rectangle : Shape { ··· };
struct Circle : Shape { ··· };

Shape* s{nullptr};
char c; std::cin >> c;
switch (c) {
 case 'r': s = new Rectangle{···}; break;
 case 'c': s = new Circle{···}; break;
}
```

  - dynamic collections of objects

```
int n; std::cin >> n;
std::vector<Particle> v;
for (int i = 0; i != n; ++i) {
  v.push_back(Particle{···});
}
```

Memory layout of a process

- A process is a running program
- When a program is started the operating system brings the contents of the corresponding file into memory according to well-defined conventions
  - Stack
    - function local variables
    - function call bookkeeping
  - Heap
    - dynamic allocation
  - Global data
    - literals and variables
    - initialized and uninitialized (set to 0)
  - Program instructions



Stack vs Heap

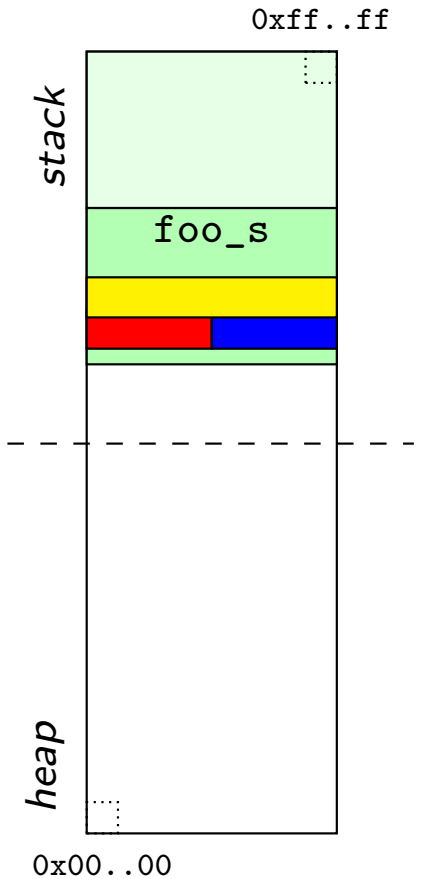
Dynamic memory is allocated on the *free store*, also known as *heap*

```
struct S {
  int    n;
  float  f;
  double d;
};
```

Stack vs Heap

Dynamic memory is allocated on the *free store*, also known as *heap*

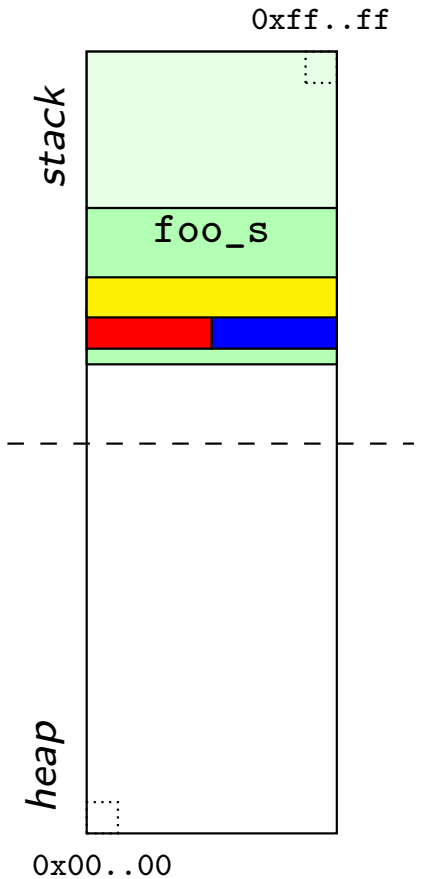
```
struct S {
  int    n;
  float  f;
  double d;
};

auto foo_s() {
  S s;
  ...
}
```

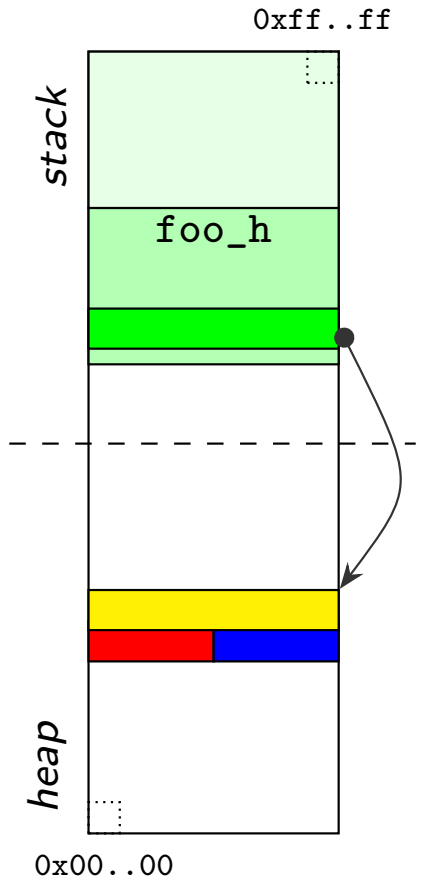


Stack vs Heap

Dynamic memory is allocated on the *free store*, also known as *heap*

```
struct S {
  int    n;
  float  f;
  double d;
};

auto foo_s() {
  S s;
  ...
}

auto foo_h() {
  S* s = new S;
  ...
}
```

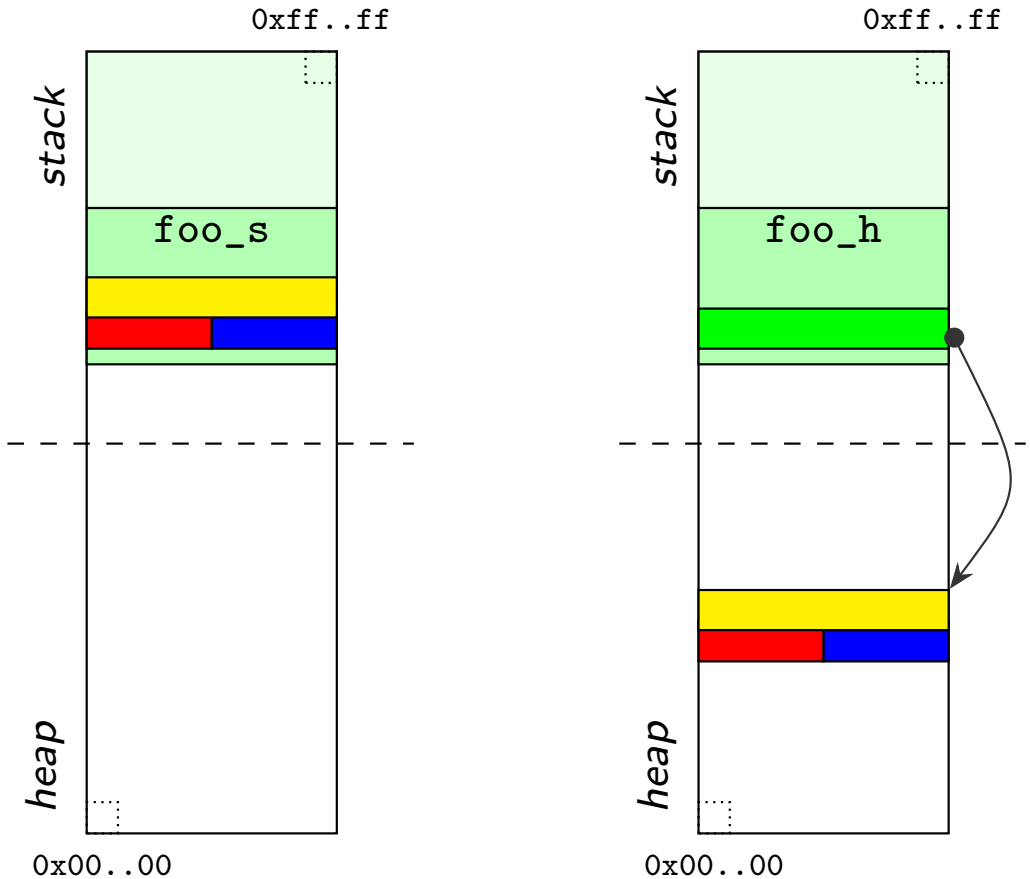




Stack vs Heap

Dynamic memory is allocated on the *free store*, also known as *heap*

```
struct S {
  int    n;
  float  f;
  double d;
};

auto foo_s() {
  S s;
  ...
}

auto foo_h() {
  S* s = new S;
  ...
}
```



Allocating on the heap has both space and time overhead

STL Containers

- Objects that contain and own other objects
- Different characteristics and operations, some common traits
- Implemented as class templates

Sequence The client decides where an element gets inserted

- `array`, `deque`, `forward_list`, `list`, `vector`

Associative The container decides where an element gets inserted

Ordered The elements are sorted

- `map`, `multimap`, `set`, `multiset`

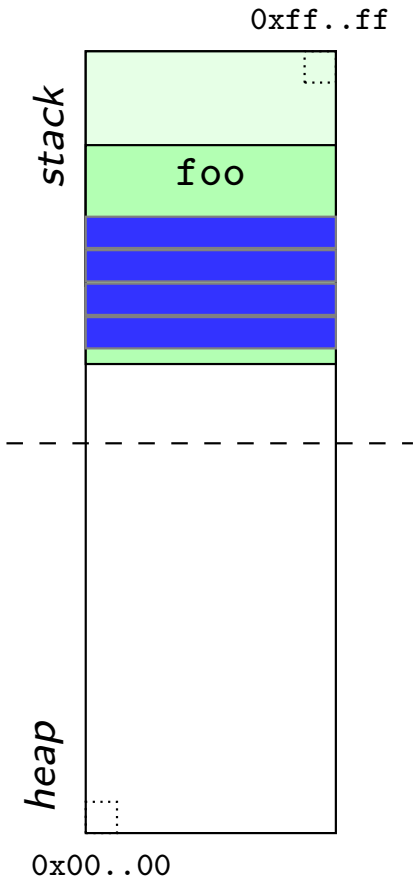
Unordered The elements are *hashed* (*)

- `unordered_*`

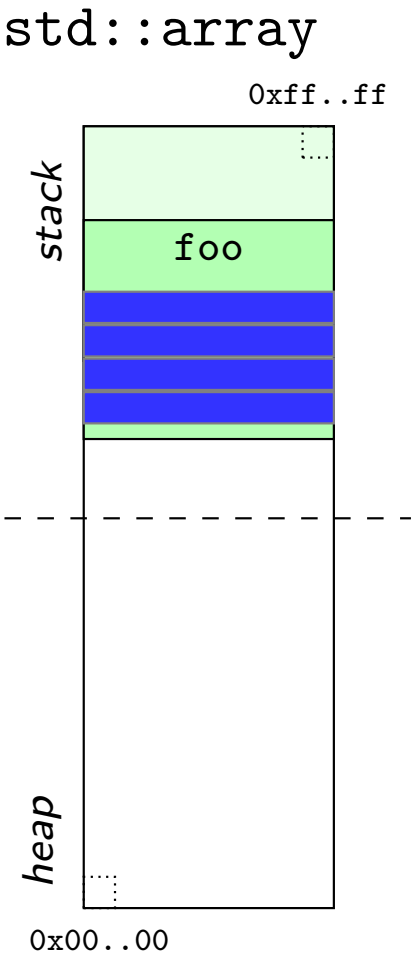
(*) The *hash* function associated to the container, when applied to the key, returns the position inside the container

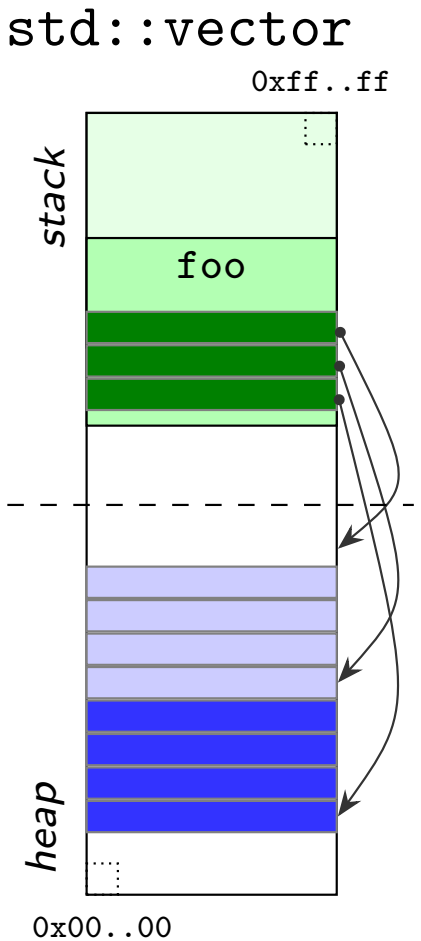
Sequence containers

`std::array`



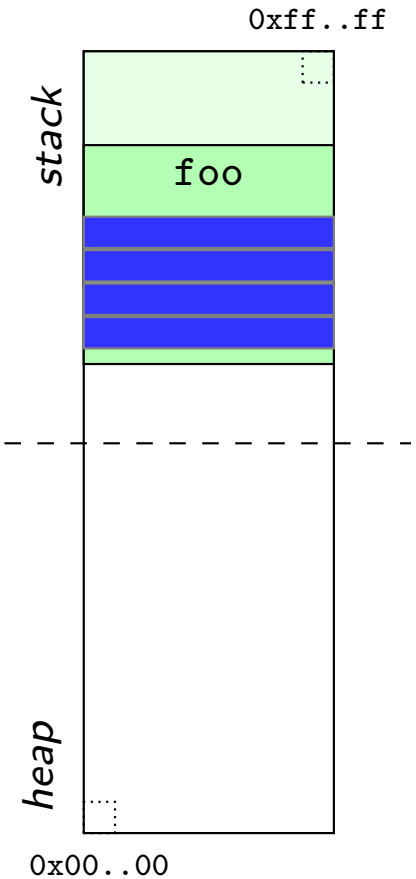
Sequence containers



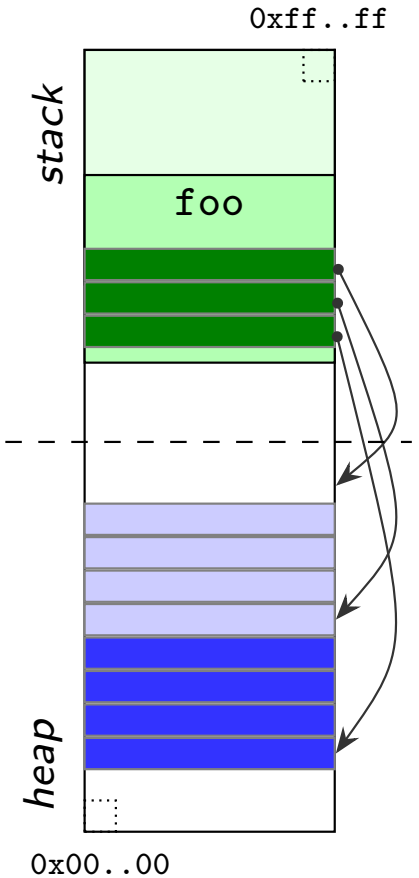


Sequence containers

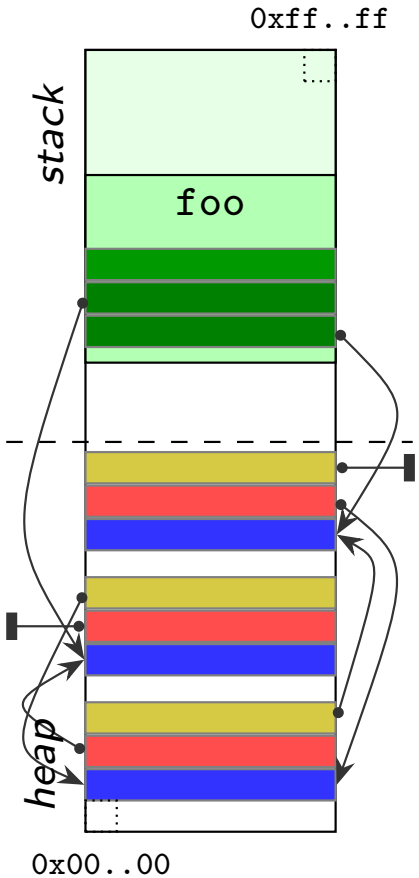
`std::array`



`std::vector`



`std::list`

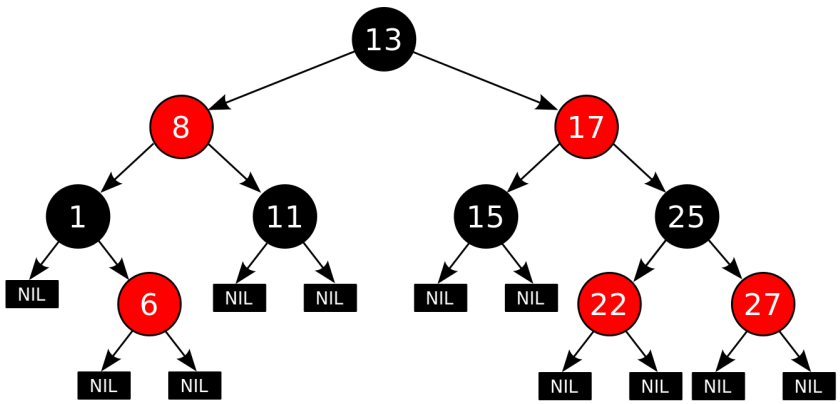


Associative ordered containers

- They contain ordered values (`set` and `multiset`) or key-value pairs (`map` and `multimap`)
- Search, removal and insertion have logarithmic complexity

Associative ordered containers

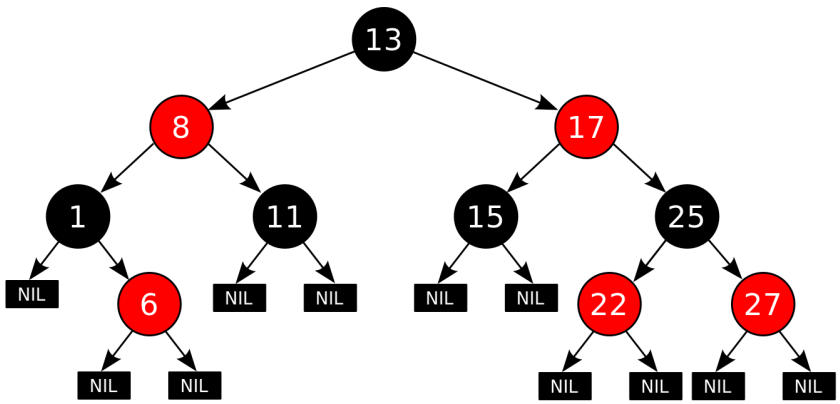
- They contain ordered values (`set` and `multiset`) or key-value pairs (`map` and `multimap`)
- Search, removal and insertion have logarithmic complexity
- Typically implemented as balanced (red-black) trees



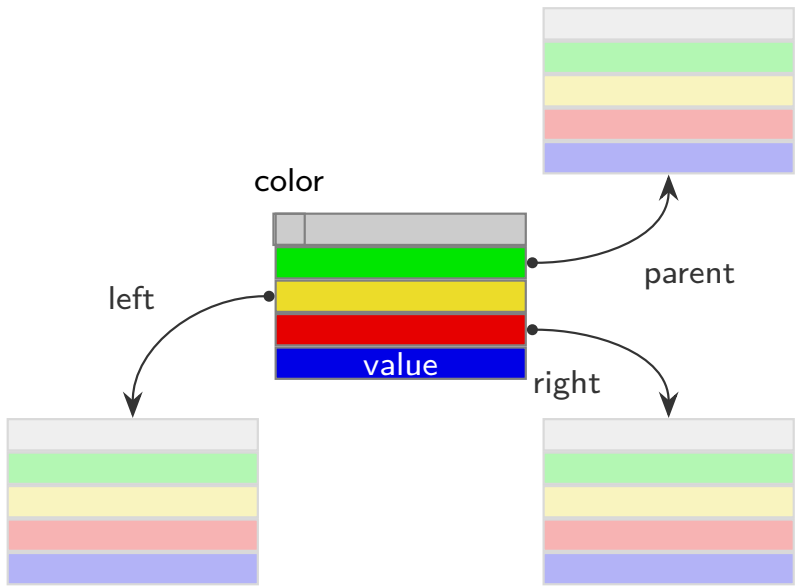
By Cburnett – Own work, CC BY-SA 3.0
https://commons.wikimedia.org/w/index.php?curid=1508398

Associative ordered containers

- They contain ordered values (`set` and `multiset`) or key-value pairs (`map` and `multimap`)
- Search, removal and insertion have logarithmic complexity
- Typically implemented as balanced (red-black) trees



By Cburnett – Own work, CC BY-SA 3.0
https://commons.wikimedia.org/w/index.php?curid=1508398



std::vector<T>

Dynamic container of elements of type `T`

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default

```
#include <vector>

std::vector<int> a;       // empty vector of ints
```

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default

```
#include <vector>

std::vector<int> a;       // empty vector of ints
std::vector<int> b{2};    // one element, initialized to 2
```

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default
- be careful with initialization: {} vs ()
  - check if there is a constructor that takes an std::initializer_list

```
#include <vector>

std::vector<int> a;      // empty vector of ints
std::vector<int> b{2};   // one element, initialized to 2
std::vector<int> c(2);   // two elements (!), value-initialized (0 for int)
```

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default
- be careful with initialization: {} vs ()
  - check if there is a constructor that takes an std::initializer_list

```
#include <vector>

std::vector<int> a;       // empty vector of ints
std::vector<int> b{2};    // one element, initialized to 2
std::vector<int> c(2);    // two elements (!), value-initialized (0 for int)
std::vector<int> d{2,1}; // two elements, initialized to 2 and 1
std::vector<int> e(2,1); // two elements, both initialized to 1
```

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default
- be careful with initialization: {} vs ()
  - check if there is a constructor that takes an std::initializer_list

```
#include <vector>

std::vector<int> a;       // empty vector of ints
std::vector<int> b{2};    // one element, initialized to 2
std::vector<int> c(2);    // two elements (!), value-initialized (0 for int)
std::vector<int> d{2,1}; // two elements, initialized to 2 and 1
std::vector<int> e(2,1); // two elements, both initialized to 1


auto f = b; // make a copy, f and b are two distinct objects
f == b;     // true
```

std::vector<T> (cont.)

- The `size` method gives the number of elements in the vector
- The `empty` method tells if the vector is empty
- `operator[]` gives access to the i^{th} element

```
std::vector<int> vec{4,2,7};

assert(!vec.empty());
std::cout << vec.size(); // print 3
vec[1] = 5;              // vec is now {4,5,7}
std::cout << vec[1];     // print 5
```

std::vector<T> (cont.)

- The `size` method gives the number of elements in the vector
- The `empty` method tells if the vector is empty
- `operator[]` gives access to the i^{th} element

```
std::vector<int> vec{4,2,7};

assert(!vec.empty());
std::cout << vec.size(); // print 3
vec[1] = 5;              // vec is now {4,5,7}
std::cout << vec[1];     // print 5
```

- Remember that counting starts from 0!

```
vec[0];             // first element
vec[1];             // second element
vec[vec.size()-1]; // vec[2], third and last element
vec[vec.size()];   // vec[3], element doesn't exist, undefined behaviour!
```

std::vector<T> (cont.)

- The `push_back` method adds an element at the end of the vector

std::vector<T> (cont.)

- The `push_back` method adds an element at the end of the vector

```
vec.push_back(-2);        // vec is now {4,5,7,-2}
vec.push_back(0);         // vec is now {4,5,7,-2,0}
std::cout << vec.size(); // print 5
```

std::vector<T> (cont.)

- The push_back method adds an element at the end of the vector

```
vec.push_back(-2);        // vec is now {4,5,7,-2}
vec.push_back(0);         // vec is now {4,5,7,-2,0}
std::cout << vec.size(); // print 5
```

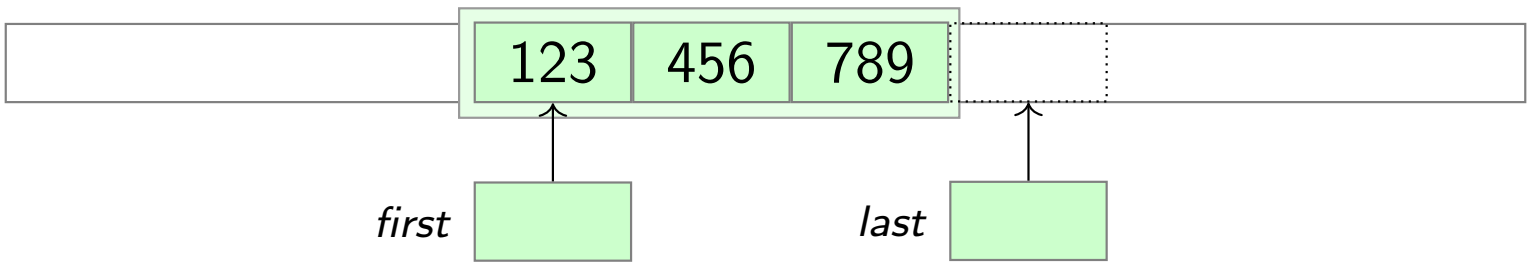
```
// fill a vector with numbers read from standard input
std::vector<double> v;
for (double d; std::cin >> d; ) {
  v.push_back(d);
}
```

Iterators and ranges

- An iterator is an object that indicates a position within a range
  - A container, such as a vector, is a range

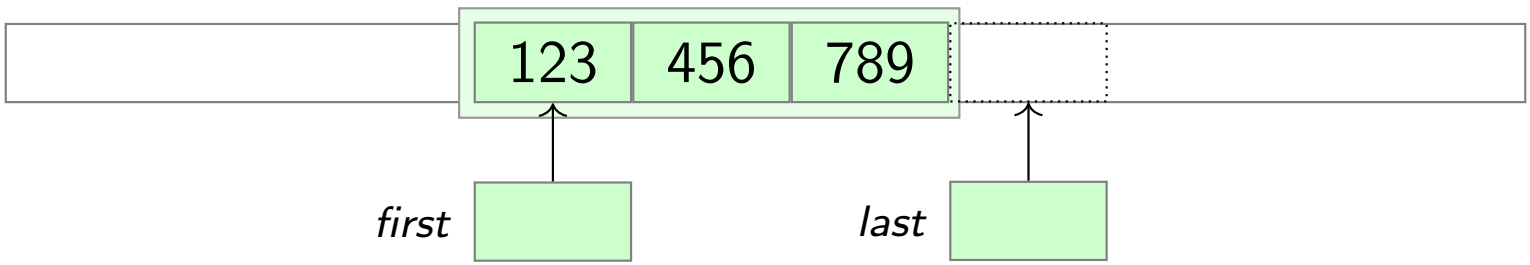
Iterators and ranges

- An iterator is an object that indicates a position within a range
  - A container, such as a vector, is a range
- In fact, a pair of iterators [*first*, *last*) represents a range
  - the range is *half-open*
    - *first* points to the first element of the range
    - *last* points to **one past** the last element of the range
  - *first* == *last* means the range is empty



Iterators and ranges

- An iterator is an object that indicates a position within a range
  - A container, such as a vector, is a range
- In fact, a pair of iterators [*first*, *last*) represents a range
  - the range is *half-open*
    - *first* points to the first element of the range
    - *last* points to **one past** the last element of the range
  - *first* == *last* means the range is empty



- Ranges are typically obtained from containers calling methods `begin` and `end` (or the more generic functions `std::begin` and `std::end`)

```
std::vector<int> v {···};
auto first = v.begin(); // std::vector<int>::iterator
auto last  = v.end();   // std::vector<int>::iterator
```

Operations on iterators

- Syntactically, operations on iterators are inspired by pointers
- There is a minimal set of operations supported by an iterator `it`
- `*it` gives access to the element pointed to by `it`

```
std::vector<int> v {1,2,3};
auto it = v.begin();
std::cout << *it; // print 1
*it = 4;          // v is now {4,2,3}
```

```
auto it = v.end();

*it; // undefined behaviour, it doesn't point inside the range
```

Operations on iterators (cont.)

- `it->member` gives access to a member (data or function) of the element pointed to by `it`
  - equivalent to `(*it).member`, note the parenthesis

```
std::vector<Point> v {Point{1,2}, Point{3,4}};
auto it = v.begin();
std::cout << (*it).x; // print 1
std::cout << it->x;   // equivalent
```

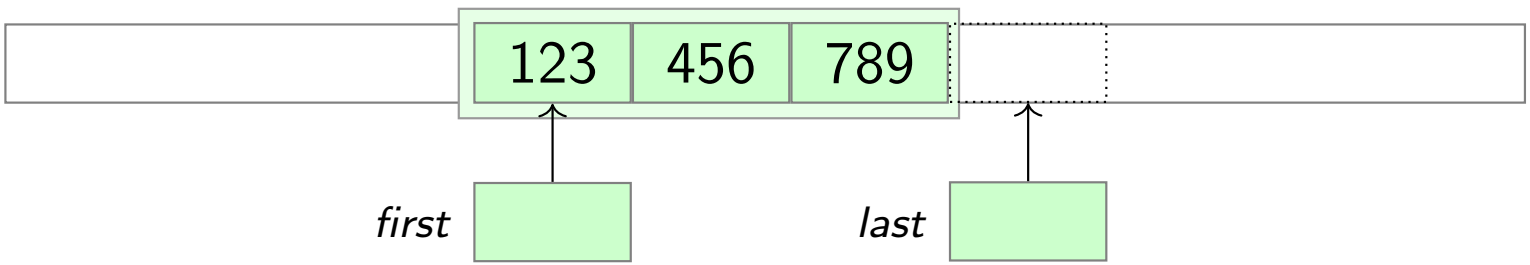
```
struct Point {
  double x;
  double y;
};
```

```
std::vector<std::string> v {"hello", "world"};
auto itv = v.begin();    // itv points to the first string in the vector
auto its = itv->begin(); // its points to the first character
                         //   of the first string ('h');
                         //   a string is a container of characters
```

Operations on iterators (cont.)

- `++it` advances `it` so that it points to the next element in the range

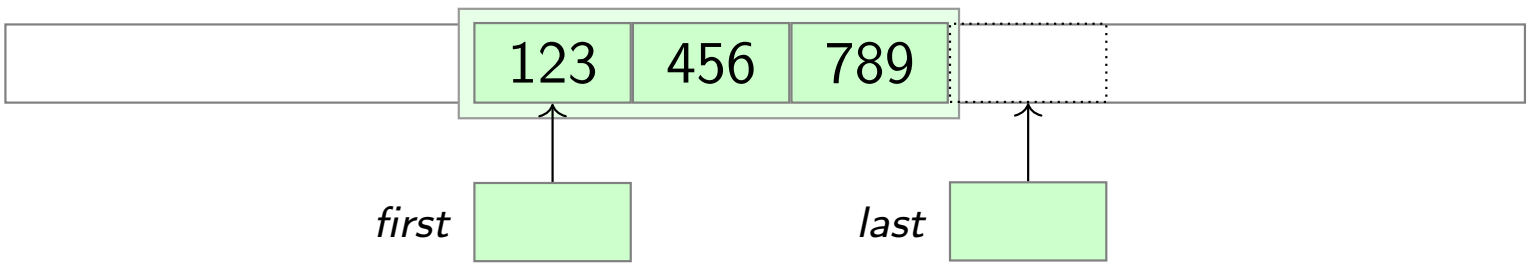
```
std::vector<int> v {123, 456, 789};
auto first = v.begin();
auto last  = v.end();
std::cout << *first; // print 123
```



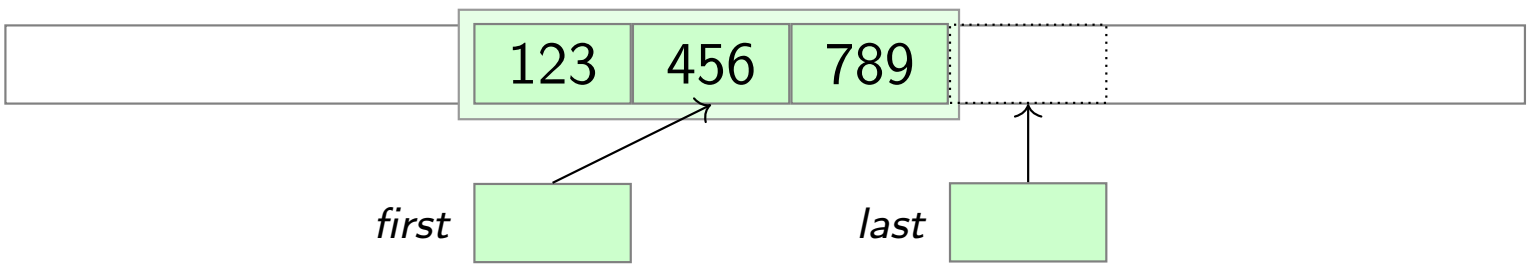
Operations on iterators (cont.)

- `++it` advances `it` so that it points to the next element in the range

```
std::vector<int> v {123, 456, 789};
auto first = v.begin();
auto last  = v.end();
std::cout << *first; // print 123
```



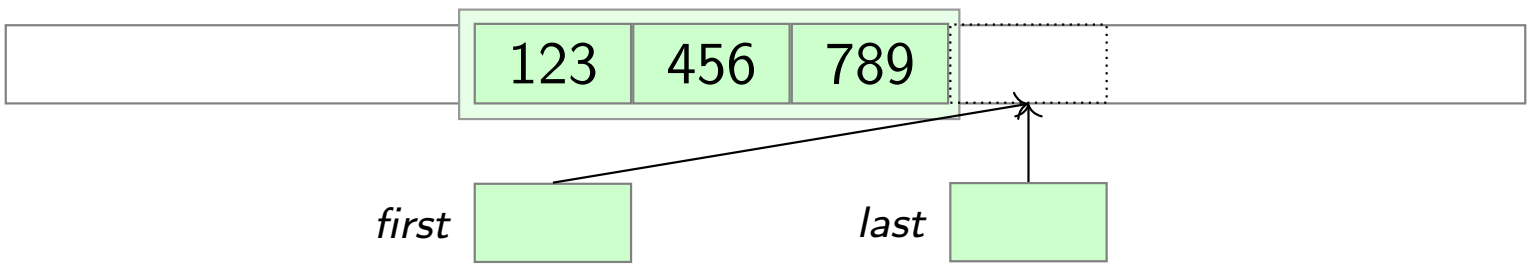

```
++first;
std::cout << *first; // print 456
```



Operations on iterators (cont.)

- `it1 == it2` (`it1 != it2`) tells if `it1` and `it2` point to the same element (different elements) of the range

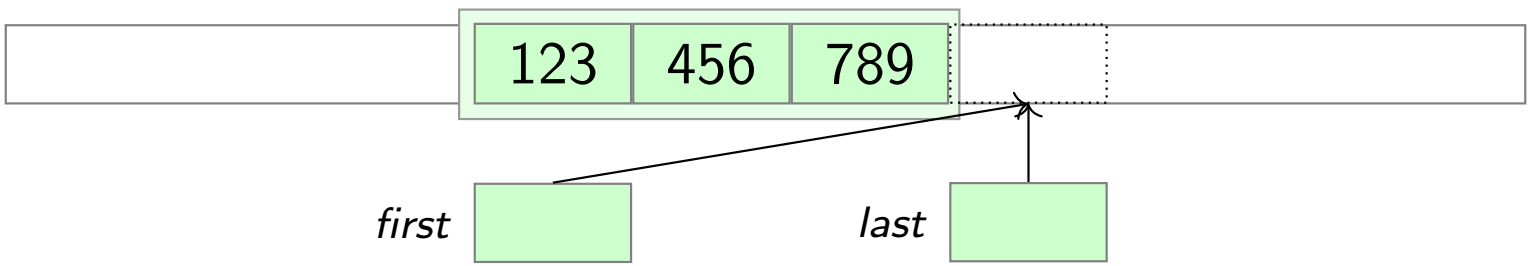
```
++first; ++first;
first == last;  // true
```



Operations on iterators (cont.)

- `it1 == it2` (`it1 != it2`) tells if `it1` and `it2` point to the same element (different elements) of the range

```
++first; ++first;
first == last;  // true
```



- Other operations (`--it`, `it + n`, `it += n`, `it < it2`, ...) may be supported, depending on the underlying range
  - vector iterators support them all

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ··· };
```

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ··· };
```

```
auto sum = 0;
for (int i {0}, n = c.size(); i != n; ++i) {
  auto const& v = c[i];
  sum += v;
}
```

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ··· };
```

```
auto sum = 0;
for (int i {0}, n = c.size(); i != n; ++i) {
  auto const& v = c[i];
  sum += v;
}
```

```
auto sum = 0;
for (auto it = c.begin(), last = c.end(); it != last; ++it) {
  auto const& v = *it;
  sum += v;
}
```

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ··· };
```

```
auto sum = 0;
for (int i {0}, n = c.size(); i != n; ++i) {
  auto const& v = c[i];
  sum += v;
}
```

```
auto sum = 0;
for (auto it = c.begin(), last = c.end(); it != last; ++it) {
  auto const& v = *it;
  sum += v;
}
```

```
auto sum = 0;
for (auto const& v : c) { // translates to the loop with iterators
  sum += v;
}
```

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ··· };
```

```
auto sum = 0;
for (int i {0}, n = c.size(); i != n; ++i) {
  auto const& v = c[i];
  sum += v;
}
```

```
auto sum = 0;
for (auto it = c.begin(), last = c.end(); it != last; ++it) {
  auto const& v = *it;
  sum += v;
}
```

```
auto sum = 0;
for (auto const& v : c) { // translates to the loop with iterators
  sum += v;
}
```

```
auto sum = std::accumulate(c.begin(), c.end(), 0); // algorithm
```

std::vector<T>: removing elements

- The `erase` method removes the element pointed to by the iterator passed as argument (must not be `end()`)

```
// remove the central element
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it);                        // size decreased by 1
```

std::vector<T>: removing elements

- The erase method removes the element pointed to by the iterator passed as argument (must not be end())

```
// remove the central element
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it);                        // size decreased by 1
```

- The erase method can remove a range, passing two iterators

```
// erase the 2nd half of the vector
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it, v.end());               // size ~halved here
```

std::vector<T>: removing elements

- The `erase` method removes the element pointed to by the iterator passed as argument (must not be `end()`)

```
// remove the central element
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it);                        // size decreased by 1
```

- The `erase` method can remove a range, passing two iterators

```
// erase the 2nd half of the vector
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it, v.end());               // size ~halved here
```

- In general, iterators pointing to erased elements are not valid anymore
  - For a vector, iterators pointing to elements following the erased one are also invalidated, including the end iterator

```
v.erase(it);
*it;       // undefined behaviour (UB)
++it;      // UB, be careful in loops, rather use it = v.erase(it)
it = ···;  // ok
```

std::vector<T>: inserting elements

- The `insert` method inserts an element before the position indicated by an iterator
  - subsequent elements are shifted one position towards the end of the vector

```
// insert the value 42 in the middle of the vector
auto it = v.begin() + v.size() / 2;
v.insert(it, 42);                    // size increased by 1
```

std::vector<T>: inserting elements

- The `insert` method inserts an element before the position indicated by an iterator
  - subsequent elements are shifted one position towards the end of the vector

```
// insert the value 42 in the middle of the vector
auto it = v.begin() + v.size() / 2;
v.insert(it, 42);                    // size increased by 1
```

- Other overloads of `insert` are available, see reference

std::vector<T>: inserting elements

- The `insert` method inserts an element before the position indicated by an iterator
  - subsequent elements are shifted one position towards the end of the vector

```
// insert the value 42 in the middle of the vector
auto it = v.begin() + v.size() / 2;
v.insert(it, 42);                    // size increased by 1
```

- Other overloads of `insert` are available, see reference
- Existing iterators pointing to elements after the insertion position are invalidated
  - If a memory reallocation occurs (see later), all existing iterators pointing into the vector are invalidated

`std::vector<T>`: controlling capacity

- For efficiency reasons, `std::vector` keeps extra capacity available for subsequent insertions
- The method `capacity` returns the current capacity allocated for a vector
  - $0 \leq$ `v.size()` $\leq$ `v.capacity()`

std::vector<T>: controlling capacity

- For efficiency reasons, `std::vector` keeps extra capacity available for subsequent insertions
- The method `capacity` returns the current capacity allocated for a vector
  - $0 \leq$ `v.size()` $\leq$ `v.capacity()`
- The method `reserve` allows to pre-allocate capacity in view of subsequent insertions
  - It's a good practice to do it whenever possible

```
std::vector<int> v;
v.reserve(N); // only one memory allocation
for (int i{0}; i != N; ++i) {
  v.push_back(···);
}
```

`std::array<T, N>`

Container of `N` elements of type `T`

- `N` known and fixed at compile time
- layout is contiguous in memory

std::array<T, N>

Container of N elements of type T

- N known and fixed at compile time
- layout is contiguous in memory

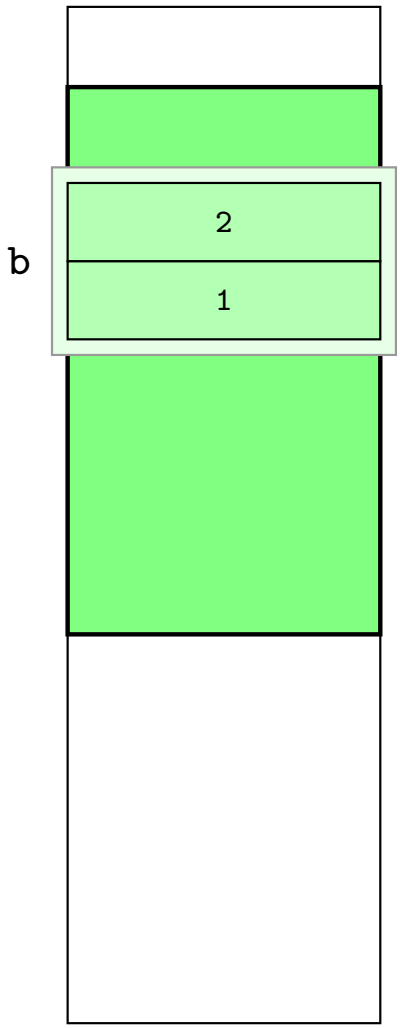
```
#include <array>

// 2 ints, uninitialized
std::array<int,2> a;

// 2 ints, initialized to 1 and 2
std::array<int,2> b{1, 2};

// 2 ints, value-initialized (0 for int)
std::array<int,2> c{};

// 2 ints, initialized to 1 and 0
std::array<int,2> d{1};
```

std::array<T, N>

Container of N elements of type T

- N known and fixed at compile time
- layout is contiguous in memory

```
#include <array>

// 2 ints, uninitialized
std::array<int,2> a;

// 2 ints, initialized to 1 and 2
std::array<int,2> b{1, 2};

// 2 ints, value-initialized (0 for int)
std::array<int,2> c{};

// 2 ints, initialized to 1 and 0
std::array<int,2> d{1};
```

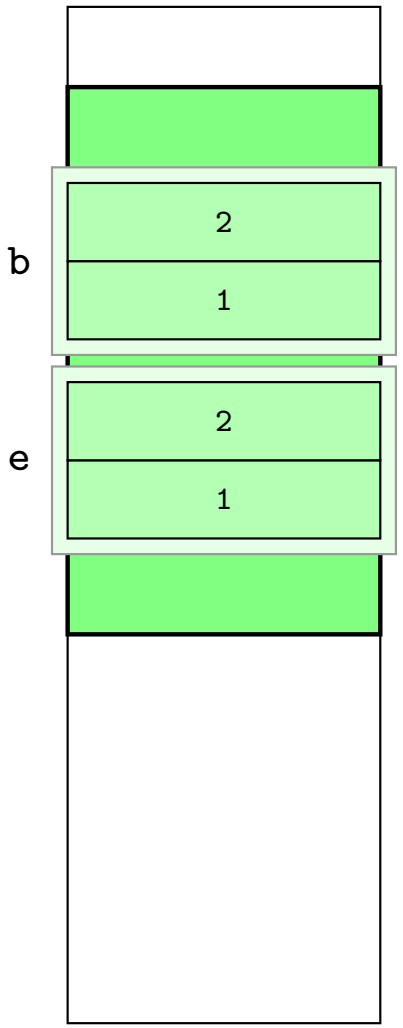


std::array<T, N>

Container of N elements of type T

- N known and fixed at compile time
- layout is contiguous in memory

```
#include <array>

// 2 ints, uninitialized
std::array<int,2> a;

// 2 ints, initialized to 1 and 2
std::array<int,2> b{1, 2};

// 2 ints, value-initialized (0 for int)
std::array<int,2> c{};

// 2 ints, initialized to 1 and 0
std::array<int,2> d{1};

// make a copy
auto e = b;  // std::array<int,2>
assert(e == b);
```



std::array<T, N> (cont.)

- The `size` method gives the number of elements in the array (which corresponds to `N`)
- `operator[]` gives access to the i^{th} element

```
int const M{4};
std::array<int, M> arr{1, 2}; // {1, 2, 0, 0}

std::cout << arr.size(); // print 4
arr[1] = 3;              // arr is now {1, 3, 0, 0}
std::cout << arr[1];     // print 3
```

- `begin`, `end`, `empty`, `front`, `back` methods
- Since the size is fixed, there is no `push_back`, `insert`, `erase`

Hands-on

- C++ $\rightarrow$ Containers
- Starting from `containers.cpp`, fill a `std::array`, a `std::vector` and a `std::list` with 1'000'0000'000 numbers, measuring the time it takes for each
- For `std::vector` try with and without calling `reserve`
- For `std::vector` and `std::list` try with a container already constructed with the right size
- For `std::array` the operation can fail badly. Why?

Outline

Algorithms

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

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates
- Sum all the elements of a container `cont` of `int`s

```
int sum {0};
for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {
  sum += *it;
}
```

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates
- Sum all the elements of a container `cont` of `int`s

```
int sum {0};
for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {
  sum += *it;
}
```

```
auto sum = std::accumulate(cont.begin(), cont.end(), 0); // better
```

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates
- Sum all the elements of a container `cont` of `int`s

```
int sum {0};
for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {
  sum += *it;
}
```

```
auto sum = std::accumulate(cont.begin(), cont.end(), 0); // better
```

- Find an element equal to `val` in a container `cont`

```
auto it = cont.begin();
auto const last = cont.end();
for (; it != last; ++it) {
  if (*it == val) {
    break;
  }
}
```

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates
- Sum all the elements of a container `cont` of `int`s

```
int sum {0};
for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {
  sum += *it;
}
```

```
auto sum = std::accumulate(cont.begin(), cont.end(), 0); // better
```

- Find an element equal to `val` in a container `cont`

```
auto it = cont.begin();
auto const last = cont.end();
for (; it != last; ++it) {
  if (*it == val) {
    break;
  }
}
```

```
auto it = std::find(cont.begin(), cont.end(), val); // better
```

Generic programming

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

```
template<class InputIterator, class Tp>
Tp accumulate(InputIterator first, InputIterator last, Tp init)
{
    for (; first != last; ++first)
        init = init + *first;
    return init;
}
```

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

Concepts

- A *concept* is a set of requirements that a type needs to satisfy at compile time
  - e.g. supported expressions, nested typedefs, memory layout, ...

Concepts

- A *concept* is a set of requirements that a type needs to satisfy at compile time
  - e.g. supported expressions, nested typedefs, memory layout, ...
- Concepts constrain the types that can be used as template arguments
  - Implicit until C++20, based on how those types are used in a class/function template definition

Concepts

- A *concept* is a set of requirements that a type needs to satisfy at compile time
  - e.g. supported expressions, nested typedefs, memory layout, ...
- Concepts constrain the types that can be used as template arguments
  - Implicit until C++20, based on how those types are used in a class/function template definition
  - C++20 introduces a specific syntax and a set of generally useful concepts (not further discussed)

```
template<class T>
concept Incrementable = requires(T t) { ++t; };

template<Incrementable T>
auto advance(T& t) { ++t; }

int i {42};
advance(i); // ok, int is a model of Incrementable

struct S {};
S s;
advance(s); // error, S is not a model of Incrementable
```

Examples of algorithms

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

Algorithms in action

```
std::array a {23, 54, 41, 0, 18};

// sort the array in ascending order
std::sort(std::begin(a), std::end(a));

// sum up the array elements, initializing the sum to 0
auto s = std::accumulate(std::begin(a), std::end(a), 0);
// similar
auto r = std::reduce(std::begin(a), std::end(a));

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

auto p = std::inner_product(std::begin(a), std::end(a), std::begin(v), 0);

// find the first element with value 42, if existing
auto it = std::find(std::begin(v), std::end(v), 42);
```

Hands-on

- C++ $\rightarrow$ Algorithms
- Take again `containers.cpp` and replace the existing loops with appropriate algorithms, e.g. `std::fill` and `std::fill_n`, possibly using a `std::back_insert_iterator`

Why using standard algorithms

- They are correct

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop
- They are efficient

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop
- They are efficient
  - They give computational complexity guarantees

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop
- They are efficient
  - They give computational complexity guarantees
  - How fast do they run? how much additional memory do they need?

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop
- They are efficient
  - They give computational complexity guarantees
  - How fast do they run? how much additional memory do they need?
- They enable simple access to *parallelism*

```
#include <execution>

std::vector<int> v {···};

std::sort(std::execution::par, v.begin(), v.end());

auto it = std::find(std::execution::par, v.begin(), v.end(), 123);
```

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for`/`while` loop
- They are efficient
  - They give computational complexity guarantees
  - How fast do they run? how much additional memory do they need?
- They enable simple access to *parallelism*

```
#include <execution>

std::vector<int> v {···};

std::sort(std::execution::par, v.begin(), v.end());

auto it = std::find(std::execution::par, v.begin(), v.end(), 123);
```

- They don't play well with applications based on the design pattern called *Structure-of-Arrays*

Computational complexity

- A measure of how many resources a computation will need for a given input size
  - Typically the resource is time but can be space (memory)
  - For example: how many comparisons does the sort algorithm do for a range of one million elements?
- Of typical interest are the average case and the worst case

Computational complexity

- A measure of how many resources a computation will need for a given input size
  - Typically the resource is time but can be space (memory)
  - For example: how many comparisons does the sort algorithm do for a range of one million elements?
- Of typical interest are the average case and the worst case
- The complexity is a function f of the input size n, but usually only the asymptotic behaviour is given
  - Big-O notation
  - $\mathcal{O}(g(n))$ means that, for a large n, $f(n) \leq cg(n)$, for some constant c
  - Note how constant factors don't matter in big-O notation

Computational complexity

- A measure of how many resources a computation will need for a given input size
  - Typically the resource is time but can be space (memory)
  - For example: how many comparisons does the sort algorithm do for a range of one million elements?
- Of typical interest are the average case and the worst case
- The complexity is a function f of the input size n, but usually only the asymptotic behaviour is given
  - Big-O notation
  - $\mathcal{O}(g(n))$ means that, for a large n, $f(n) \leq cg(n)$, for some constant c
  - Note how constant factors don't matter in big-O notation
- For example
  - `std::vector<T>::push_back` is (amortized) $\mathcal{O}(1)$, i.e. constant
  - `std::binary_search` is $\mathcal{O}(\log n)$, i.e. logarithmic
  - `std::find` is $\mathcal{O}(n)$, i.e. linear
  - `std::sort` is $\mathcal{O}(n \log n)$

Functions

- A function associates a sequence of statements (the function *body*) with a name and a list of zero or more parameters
- A function may return a value
- Multiple functions can have the same name $\rightarrow$ *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
- A function may return a value
- Multiple functions can have the same name $\rightarrow$ *overloading*
  - different parameter lists
- A function returning a `bool` is called a *predicate*

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

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
  for (; first != last; ++first)
    if (pred(*first))          // unary predicate
      break;
  return first;
}
```

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
  for (; first != last; ++first)
    if (pred(*first))          // unary predicate
      break;
  return first;
}

bool lt42(int n) { return n < 42; }
```

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
  for (; first != last; ++first)
    if (pred(*first))          // unary predicate
      break;
  return first;
}

bool lt42(int n) { return n < 42; }

auto it = find_if(v.begin(), v.end(), lt42);
```

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
  for (; first != last; ++first)
    if (pred(*first))          // unary predicate
      break;
  return first;
}

bool lt42(int n) { return n < 42; }

auto it = find_if(v.begin(), v.end(), lt42);
```

Some algorithms are customizable passing a function

Algorithms and functions

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

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
  for (; first != last; ++first)
    if (pred(*first))          // unary predicate
      break;
  return first;
}

bool lt42(int n) { return n < 42; }

auto it = find_if(v.begin(), v.end(), lt42);
auto it = find_if(v.begin(), v.end(), [](int n) { return n < 42; } );
```

Some algorithms are customizable passing a function

Function objects

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

Function objects

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

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true
```

Function objects

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

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

Function objects

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

- A class with an overloaded `operator()`

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};
```

Function objects

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

- A class with an overloaded `operator()`

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
```

Function objects

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

- A class with an overloaded `operator()`

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
```

Function objects

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

- A class with an overloaded `operator()`

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
auto b = lt42(32); // true
```

Function objects

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

- A class with an overloaded `operator()`

```
auto lt42(int n)
{
  return n < 42;
}



auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42 // or directly: LessThan42{}
); // *it == 32
```

Function objects (cont.)

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

Function objects (cont.)

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

```
class LessThan {
  int m_;
 public:
  explicit LessThan(int m) : m_{m} {}
  auto operator()(int n) const {
    return n < m_;
  }
};
```

Function objects (cont.)

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

```
class LessThan {
  int m_;
 public:
  explicit LessThan(int m) : m_{m} {}
  auto operator()(int n) const {
    return n < m_;
  }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true
```

Function objects (cont.)

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

```
class LessThan {
  int m_;
 public:
  explicit LessThan(int m) : m_{m} {}
  auto operator()(int n) const {
    return n < m_;
  }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true

LessThan lt24 {24};
auto b2 = lt24(32); // false
```

Function objects (cont.)

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

```
class LessThan {
  int m_;
 public:
  explicit LessThan(int m) : m_{m} {}
  auto operator()(int n) const {
    return n < m_;
  }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true

LessThan lt24 {24};
auto b2 = lt24(32); // false


std::vector v {61,32,51};
auto i1 = std::find_if(..., lt42); // *i1 == 32

auto i2 = std::find_if(..., lt24); // i2 == v.end(), i.e. not found
```

Function objects (cont.)

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

```
class LessThan {
  int m_;
 public:
  explicit LessThan(int m) : m_{m} {}
  auto operator()(int n) const {
    return n < m_;
  }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true
// or: auto b1 = LessThan{42}(32);
LessThan lt24 {24};
auto b2 = lt24(32); // false
// or: auto b2 = LessThan{24}(32);

std::vector v {61,32,51};
auto i1 = std::find_if(..., lt42); // *i1 == 32
// or: auto i1 = std::find_if(..., LessThan{42});
auto i2 = std::find_if(..., lt24); // i2 == v.end(), i.e. not found
// or: auto i2 = std::find_if(..., LessThan{24});
```

Function objects (cont.)

An example from the standard library

```
#include <random>

// random bit generator
std::default_random_engine eng;

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

Hands-on

- C++ $\rightarrow$ Algorithms
- Let's implement `std::default_random_engine`, which is usually an alias for a *linear congruential* generator. Let's consider *minstd_rand0*, which produces a sequence according to

$$x_{n+1} = 16807 x_n \mod (2^{31} - 1)$$

Write a class `LinearCongruential` whose constructor initializes the sequence with a seed (with a default value of 1) and an `operator()` that updates the internal value (the x_n) and returns it. The type of the numbers involved in the computations is `unsigned long int`.
Print a few numbers and check that they correspond to what is produced by `std::default_random_engine`.

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 LessThan42 {
  auto operator()(int n)
  {
    return n < 42;
  }
};

class LessThan {
  int m_;
 public:
  explicit LessThan(int m)
    : m_{m} {}
  auto operator()(int n) const
  {
    return n < m_;
  }
};
```

```
std::find_if(..., LessThan42{});
```

```
std::find_if(..., LessThan{m});
```

Lambda expression

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

```
struct LessThan42 {
  auto operator()(int n)
  {
    return n < 42;
  }
};

class LessThan {
  int m_;
 public:
  explicit LessThan(int m)
    : m_{m} {}
  auto operator()(int n) const
  {
    return n < m_;
  }
};
```

```
std::find_if(..., LessThan42{});

std::find_if(..., [](int n) {
                    return n < 42;
                  }
);

std::find_if(..., LessThan{m});
```

Lambda expression

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

```
struct LessThan42 {
  auto operator()(int n)
  {
    return n < 42;
  }
};

class LessThan {
  int m_;
 public:
  explicit LessThan(int m)
    : m_{m} {}
  auto operator()(int n) const
  {
    return n < m_;
  }
};
```

```
std::find_if(..., LessThan42{});

std::find_if(..., [](int n) {
                    return n < 42;
                  }
);

std::find_if(..., LessThan{m});

auto m = ···;
std::find_if(..., [=](int n) {
                    return n < m;
                  }
);
```

Lambda expression

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

```
struct LessThan42 {
  auto operator()(int n)
  {
    return n < 42;
  }
};

class LessThan {
  int m_;
 public:
  explicit LessThan(int m)
    : m_{m} {}
  auto operator()(int n) const
  {
    return n < m_;
  }
};
```

```
std::find_if(..., LessThan42{});

std::find_if(..., [](int n) {
                    return n < 42;
                  }
);

std::find_if(..., LessThan{m});

auto m = ···;
std::find_if(..., [=](int n) {
                    return n < m;
                  }
);
std::find_if(..., [m = ···](int n) {
                    return n < m;
                  }
);
```

Lambda closure

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 closure

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 lt = [ ]
```

```
class SomeUniqueName {

 public:


   auto operator()         const

};


auto lt = SomeUniqueName{ };
```

Lambda closure

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 lt = [ ](int n)
          { return n < v; }
```

```
class SomeUniqueName {

 public:


  auto operator()(int n) const
  { return n < v; }
};


auto lt = SomeUniqueName{ };
```

Lambda closure

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 v = 42;

auto lt = [ ](int n)
           { return n < v; }
```

```
class SomeUniqueName {

 public:


  auto operator()(int n) const
  { return n < v; }
};

auto v = 42;
auto lt = SomeUniqueName{ };
```

Lambda closure

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 v = 42;

auto lt = [v](int n)
          { return n < v; }
```

```
class SomeUniqueName {
  int v;
 public:
  explicit SomeUniqueName(int v)
    : v{v} {}
  auto operator()(int n) const
  { return n < v; }
};

auto v = 42;
auto lt = SomeUniqueName{v};
```

Lambda closure

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 v = 42;

auto lt = [v](int n)
          { return n < v; }

auto r = lt(5); // true
```

```
class SomeUniqueName {
  int v;
 public:
  explicit SomeUniqueName(int v)
    : v{v} {}
  auto operator()(int n) const
  { return n < v; }
};

auto v = 42;
auto lt = SomeUniqueName{v};
auto r = lt(5); // true
```

Lambda closure

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 v = 42;

auto lt = [v](int n)
          { return n < v; }

auto r = lt(5); // true
```

```
class SomeUniqueName {
  int v;
 public:
  explicit SomeUniqueName(int v)
    : v{v} {}
  auto operator()(int n) const
  { return n < v; }
};

auto v = 42;
auto lt = SomeUniqueName{v};
auto r = lt(5); // true
```

- Two lambda expressions produce objects of different types, even if they are identical

Lambda capture

- Automatic variables used in the body of the lambda need to be captured

Lambda capture

- Automatic variables used in the body of the lambda need to be captured
  - `[]` capture nothing
  - `[=]` capture all (what is needed) by value
  - `[k]` capture `k` by value

Lambda capture

- Automatic variables used in the body of the lambda need to be captured
  - [] capture nothing
  - [=] capture all (what is needed) by value
  - [k] capture k by value
  - [&] capture all (what is needed) by reference
  - [&k] capture k by reference

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

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

auto l = SomeUniqueName{v};
```

Lambda capture

- Automatic variables used in the body of the lambda need to be captured
  - `[]` capture nothing
  - `[=]` capture all (what is needed) by value
  - `[k]` capture `k` by value
  - `[&]` capture all (what is needed) 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

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

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

auto l = SomeUniqueName{v};
```

Lambda capture

- Automatic variables used in the body of the lambda need to be captured
  - `[]` capture nothing
  - `[=]` capture all (what is needed) by value
  - `[k]` capture `k` by value
  - `[&]` capture all (what is needed) 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

```
auto 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

Lambda explicit return type

- The return type of the call operator can be explicity specified

```
[=](int n) -> bool { return n < v; }
```

becomes

```
class SomeUniqueName {
  ...
  bool operator()(int n) const
  { return n < v; }
};
```

Generic lambda

- If a parameter of the lambda expression is `auto`, the lambda expression is *generic*
- The call operator is a template

```
[](auto n) { ··· }
```

becomes

```
class SomeUniqueName {
  ...
  template<typename T>
  auto operator()(T n) const { ··· }
};
```

Lambda: const and mutable

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

```
[i]                          {··· ++i ···}
```

```
class SomeUniqueName {
  int i;
  ...
  auto operator()() const {··· ++i ···}
};
```

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
  - The parameter list is mandatory

```
[i]() mutable          {··· ++i ···}
```

```
class SomeUniqueName {
  int i;
  ...
  auto operator()() {··· ++i ···}
};
```

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
  - The parameter list is mandatory
- If present, the explicit return type goes after mutable

```
[i]() mutable -> bool {··· ++i ···}
```

```
class SomeUniqueName {
  int i;
  ...
  bool operator()() {··· ++i ···}
};
```

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
  - The parameter list is mandatory
- If present, the explicit return type goes after mutable

```
[i]() mutable -> bool {··· ++i ···}
```

```
class SomeUniqueName {
  int i;
  ...
  bool operator()() {··· ++i ···}
};
```

- Variables captured by reference can be modified

```
int v = 3;
[&v] { ++v; } (); // NB the lamdba is immediately invoked
assert(v == 4);
```

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
  - The parameter list is mandatory
- If present, the explicit return type goes after mutable

```
[i]() mutable -> bool {··· ++i ···}
```

```
class SomeUniqueName {
  int i;
  ...
  bool operator()() {··· ++i ···}
};
```

- Variables captured by reference can be modified

```
int v = 3;
[&v] { ++v; } (); // NB the lamdba is immediately invoked
assert(v == 4);
```

- There is no direct way to capture by const&, but one can use std::as_const

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() // auto here is unavoidable
{
  int v = 3;
  return [&] { return v; }; // return a closure
}

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

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() // auto here is unavoidable
{
  int v = 3;
  return [&] { return v; }; // return a closure
}

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

- Capture by reference only if the lambda closure doesn't survive the current scope

Hands-on

- C++ $\rightarrow$ Algorithms
- Given a short vector of small `int`s
  - compute the product of the numbers (Hint: use `std::accumulate`). Be careful with the overflow
  - sort it in decreasing order (Hint: use `std::sort` with an appropriate comparison function)
  - compute the mean and the standard deviation in one pass (Hint: use `std::accumulate`, accumulating the sum and the sum of squares in a `struct`)
- Generate `N` numbers, using one of the available random number distributions, and insert them in a `std::vector` (Hint: use `std::generate_n`)

std::function

- *Type-erased* wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function

std::function

- *Type-erased* wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function

```
#include <functional>

using Function = std::function<int(int,int)>; // signature

Function f1 { std::plus<int>{} };
Function f2 { [](int a, int b) { return a * b; } };
Function f3 { [](auto a, auto b) { return std::gcd(a,b); } };
```

std::function

- *Type-erased* wrapper that can store and invoke any callable entity with a certain signature
  - function, function object, lambda, member function

```
#include <functional>

using Function = std::function<int(int,int)>; // signature

Function f1 { std::plus<int>{} };
Function f2 { [](int a, int b) { return a * b; } };
Function f3 { [](auto a, auto b) { return std::gcd(a,b); } };
```

- 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

```
#include <functional>

using Function = std::function<int(int,int)>; // signature

Function f1 { std::plus<int>{} };
Function f2 { [](int a, int b) { return a * b; } };
Function f3 { [](auto a, auto b) { return std::gcd(a,b); } };
```

- Some space and time overhead, so use only if a template parameter is not satisfactory

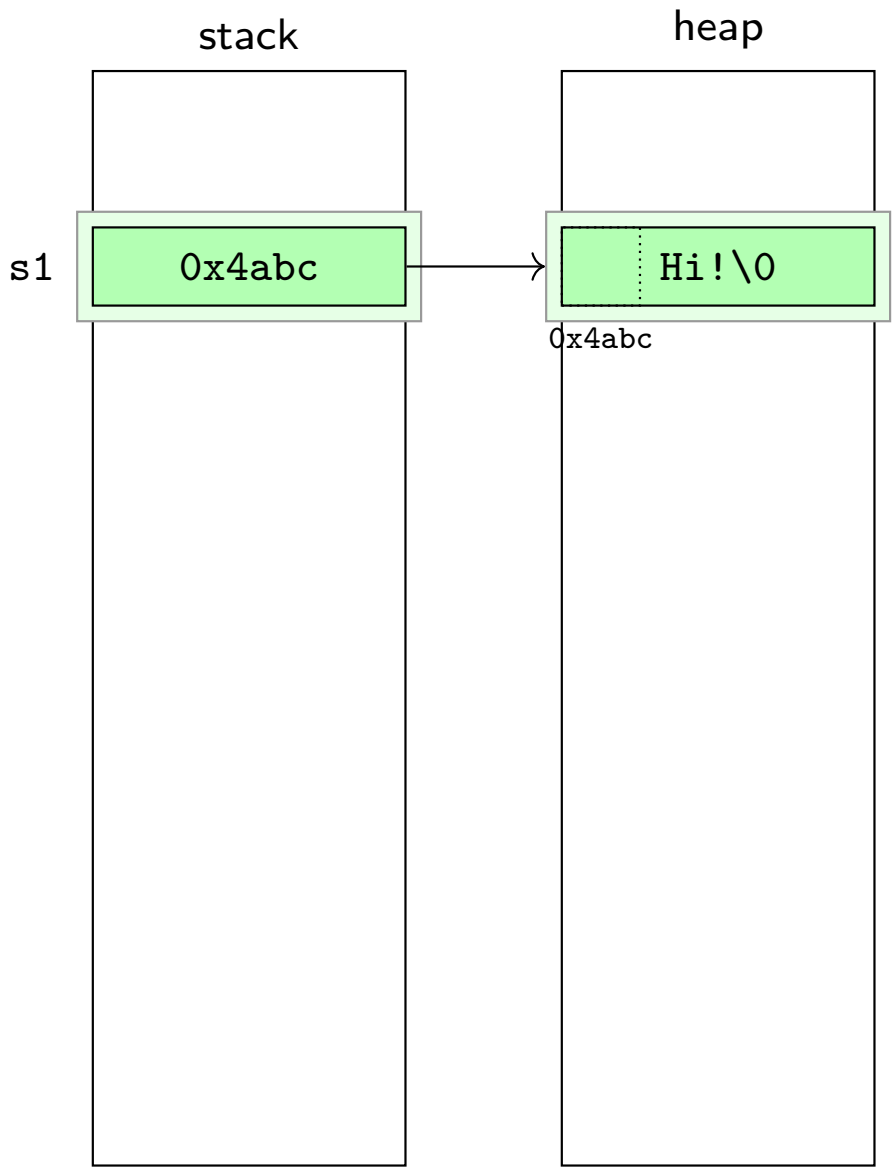
```
std::vector<Function> functions { f1, f2, f3 };

for (auto& f : functions) {
  std::cout << f(121, 42) << '\n'; // 163 5082 1
}
```

Outline

We can do better than copying

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

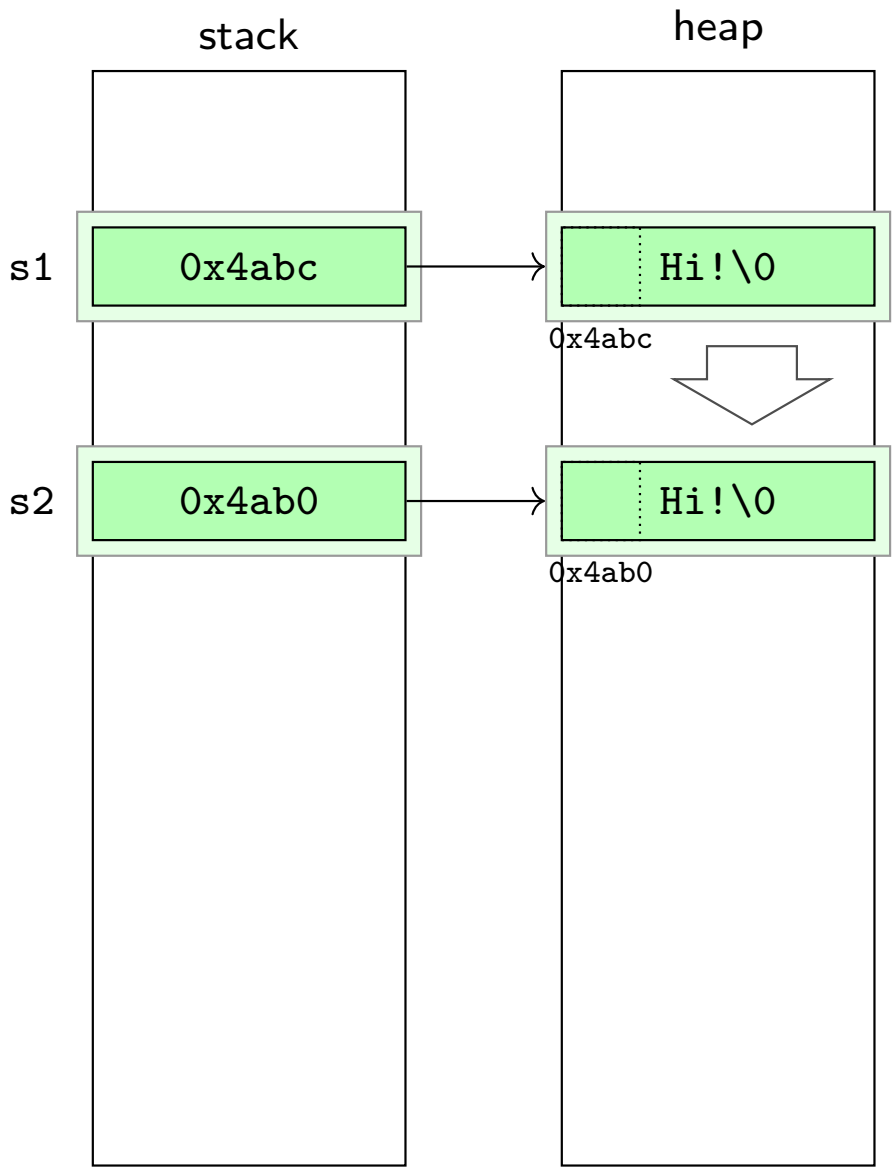
```
String s1{"Hi!"};
```



We can do better than copying

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

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

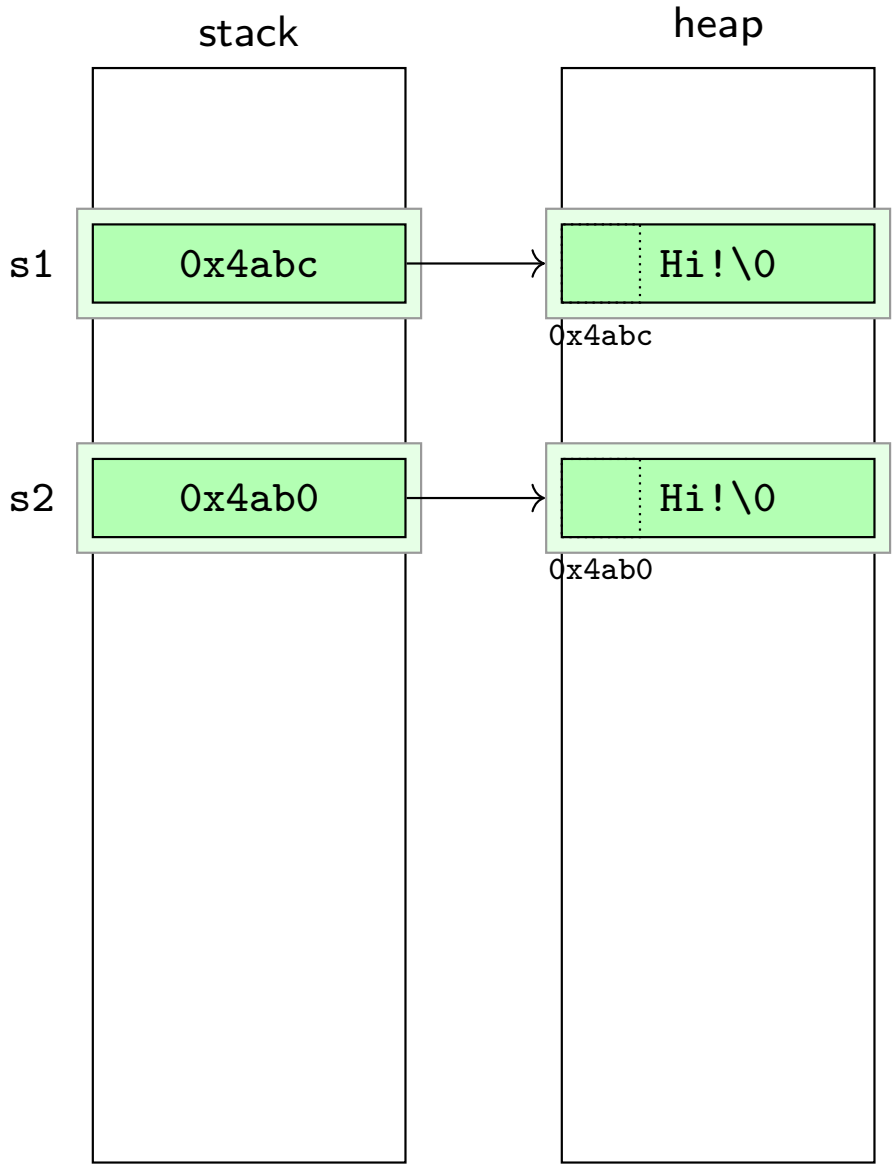


We can do better than copying

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

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

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



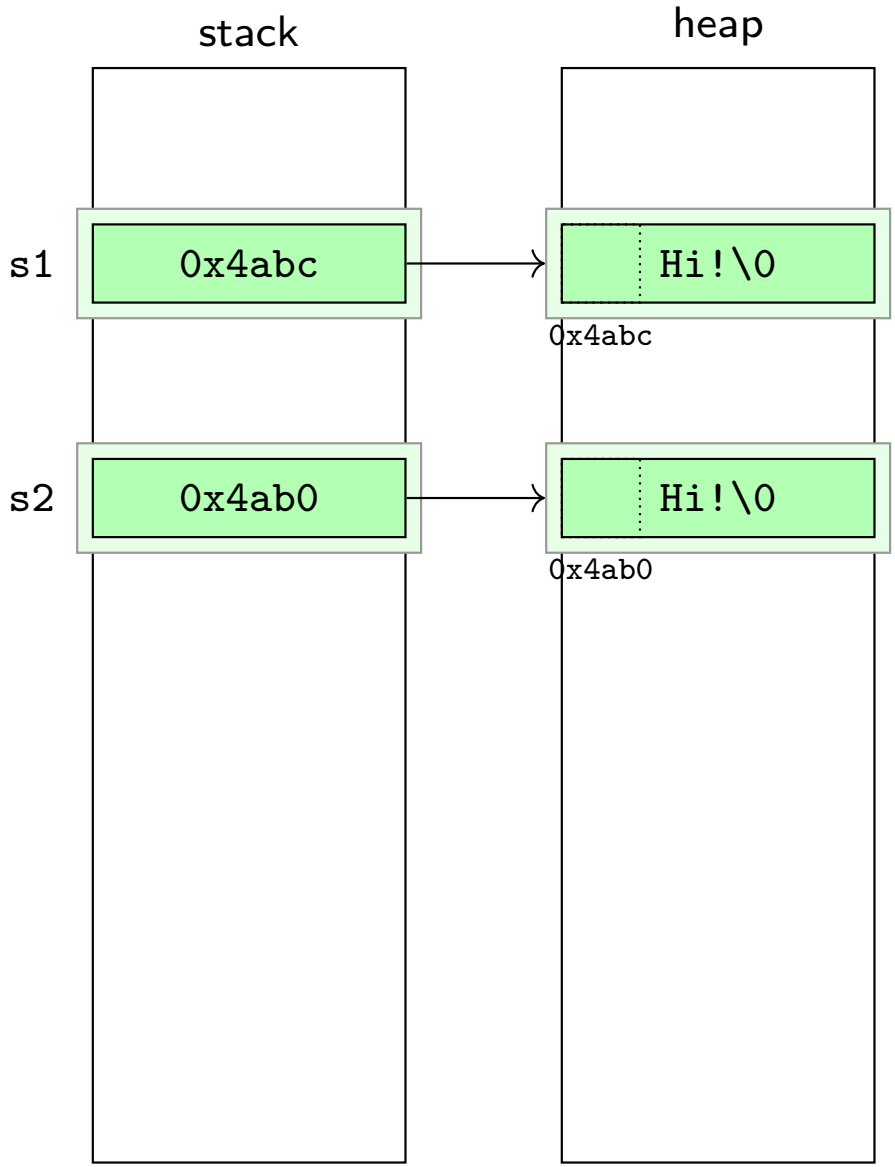
We can do better than copying

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



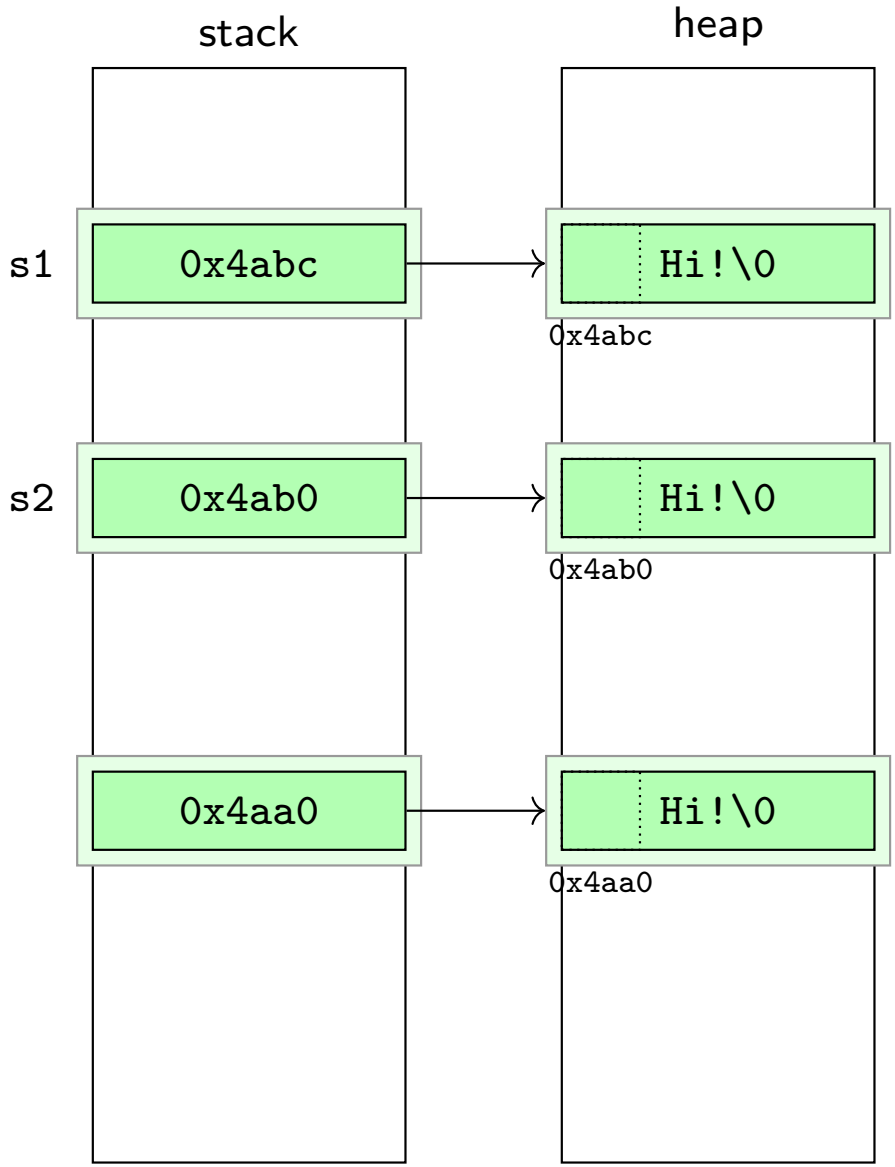
We can do better than copying

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



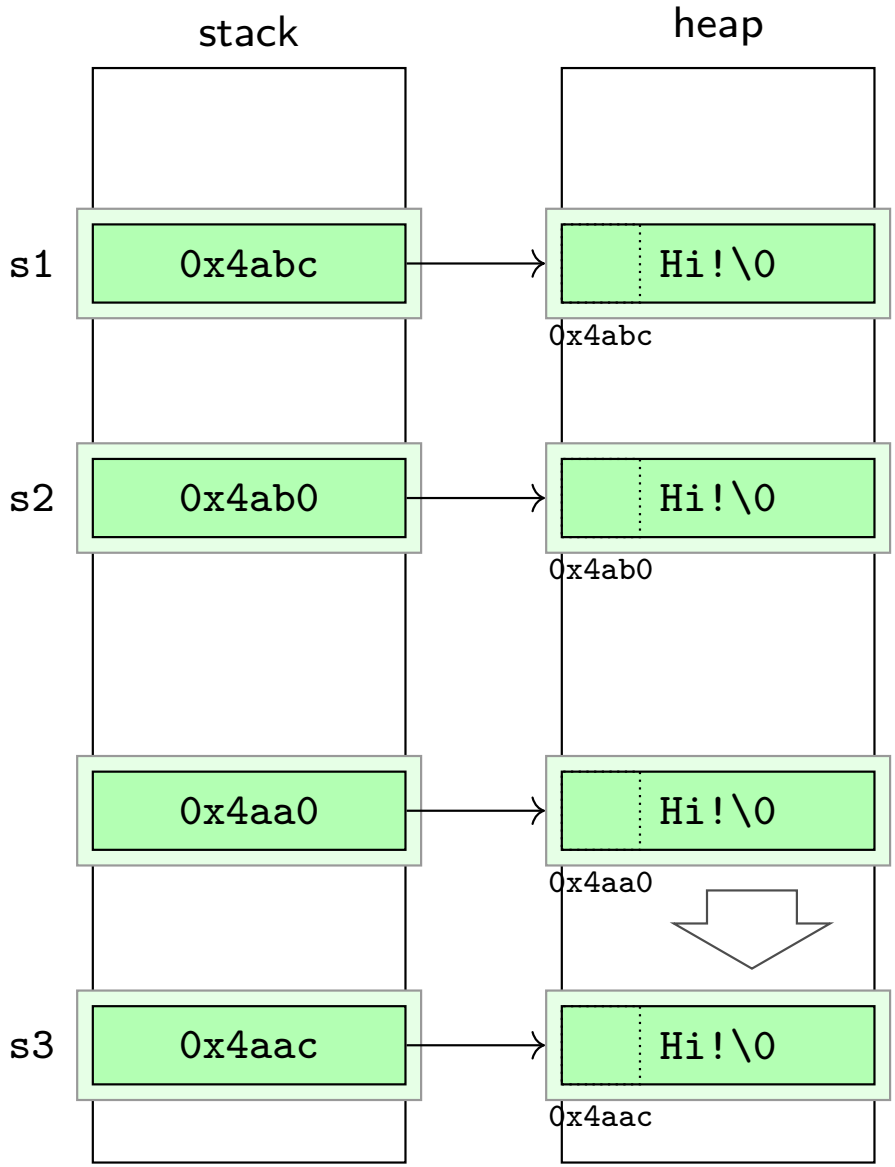
We can do better than copying

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



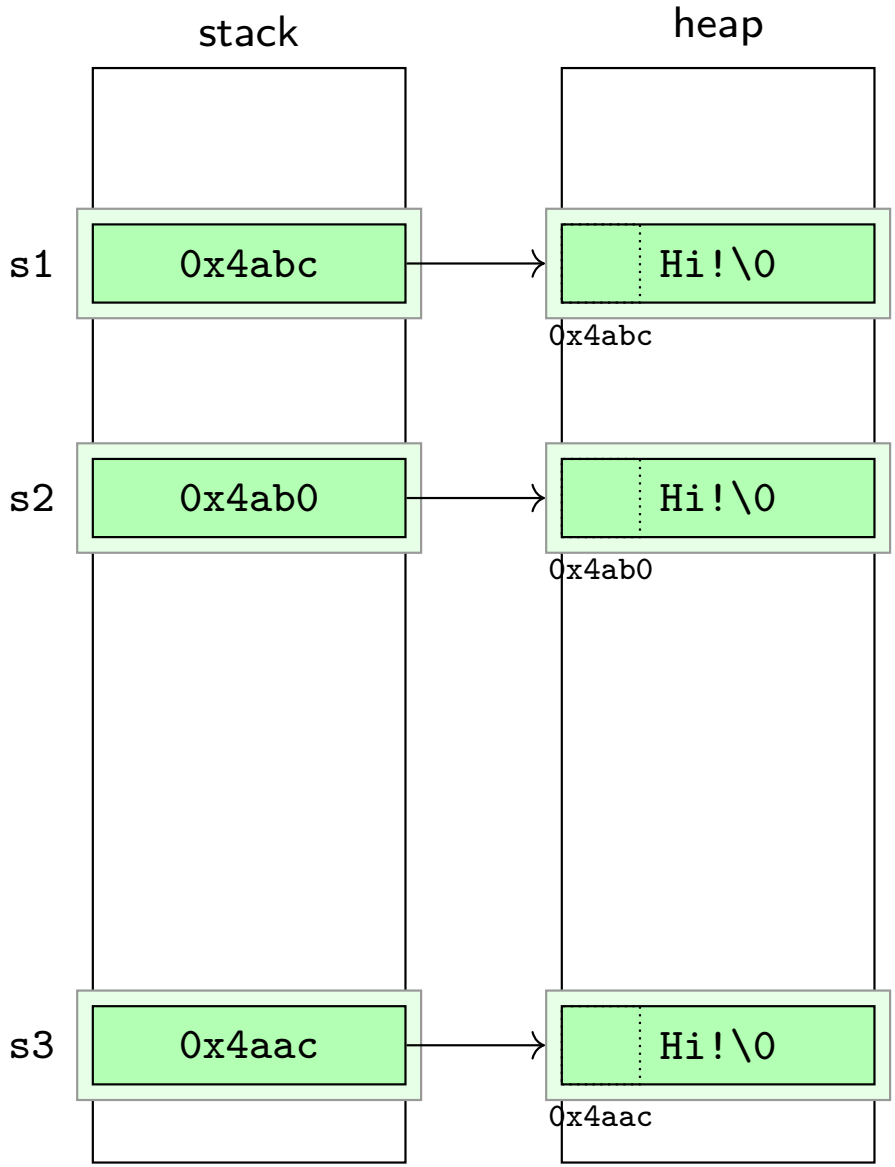
We can do better than copying

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



We can do better than copying

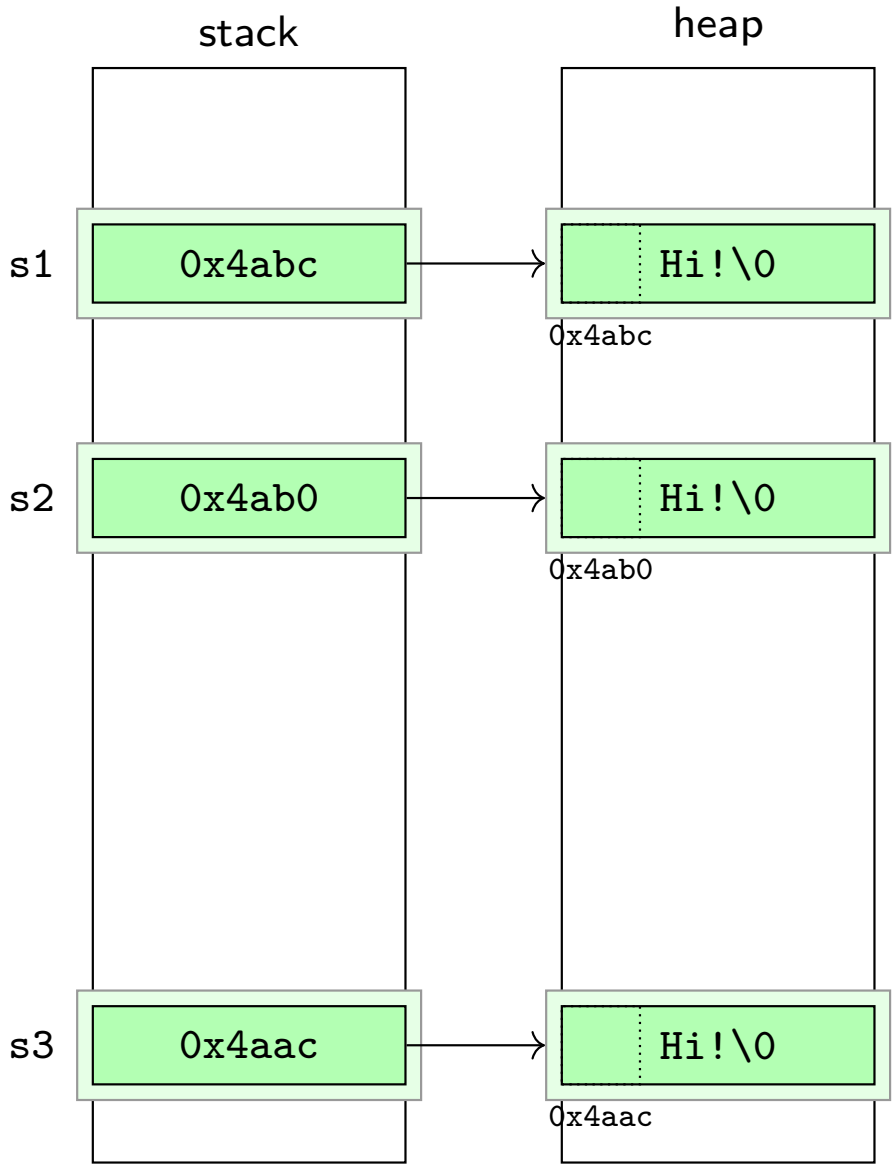
```
class String {
  char* s_;
  ...
};
```

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

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

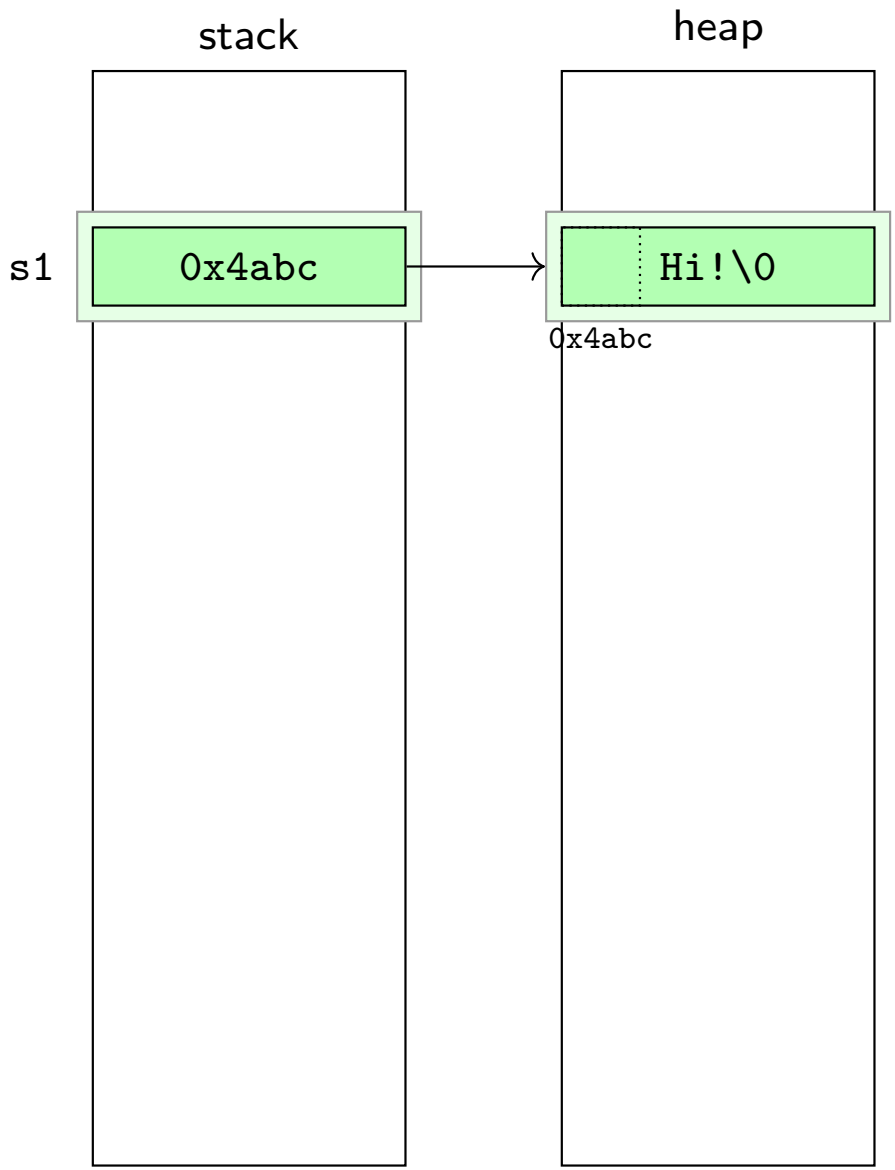
```
String get_string() { return "Hi!"; }
String s3{get_string()};
```

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



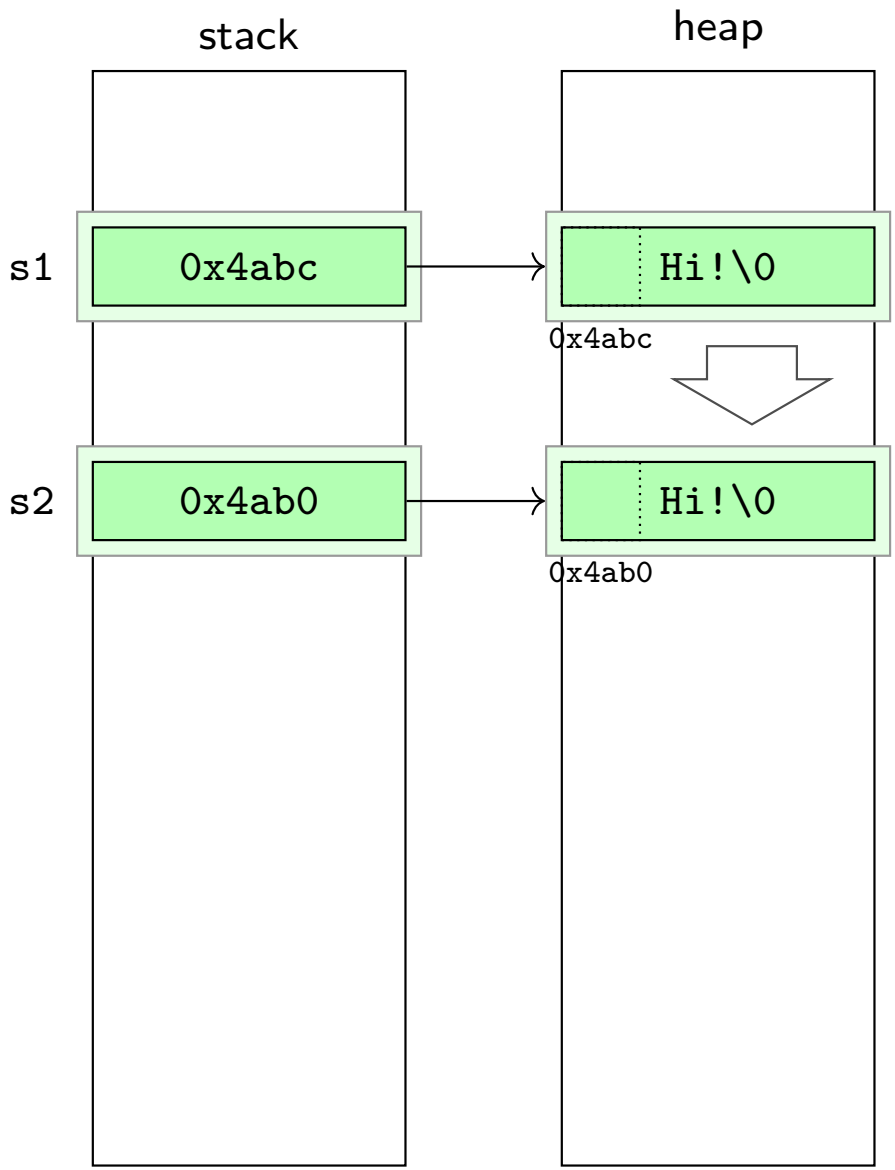
Copy vs move

```
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!"};
```



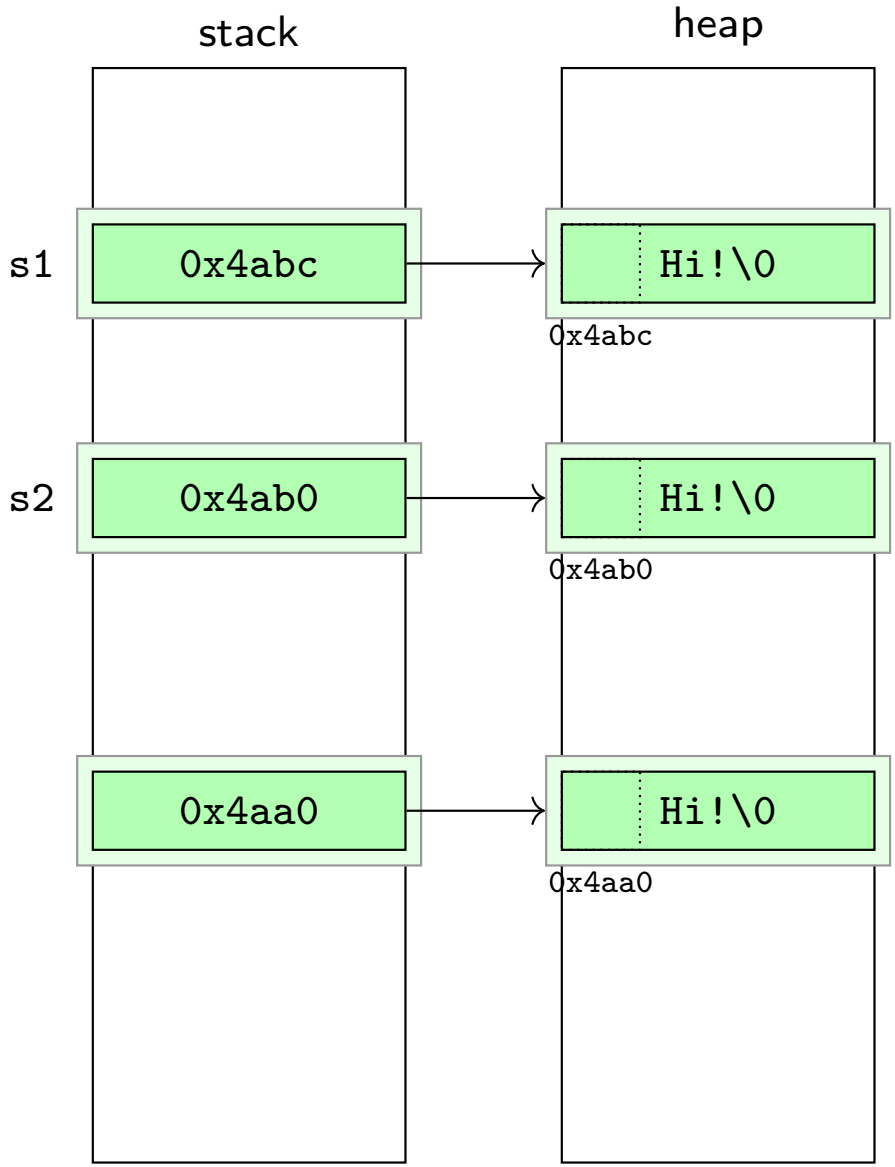
Copy vs move

```
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};
```



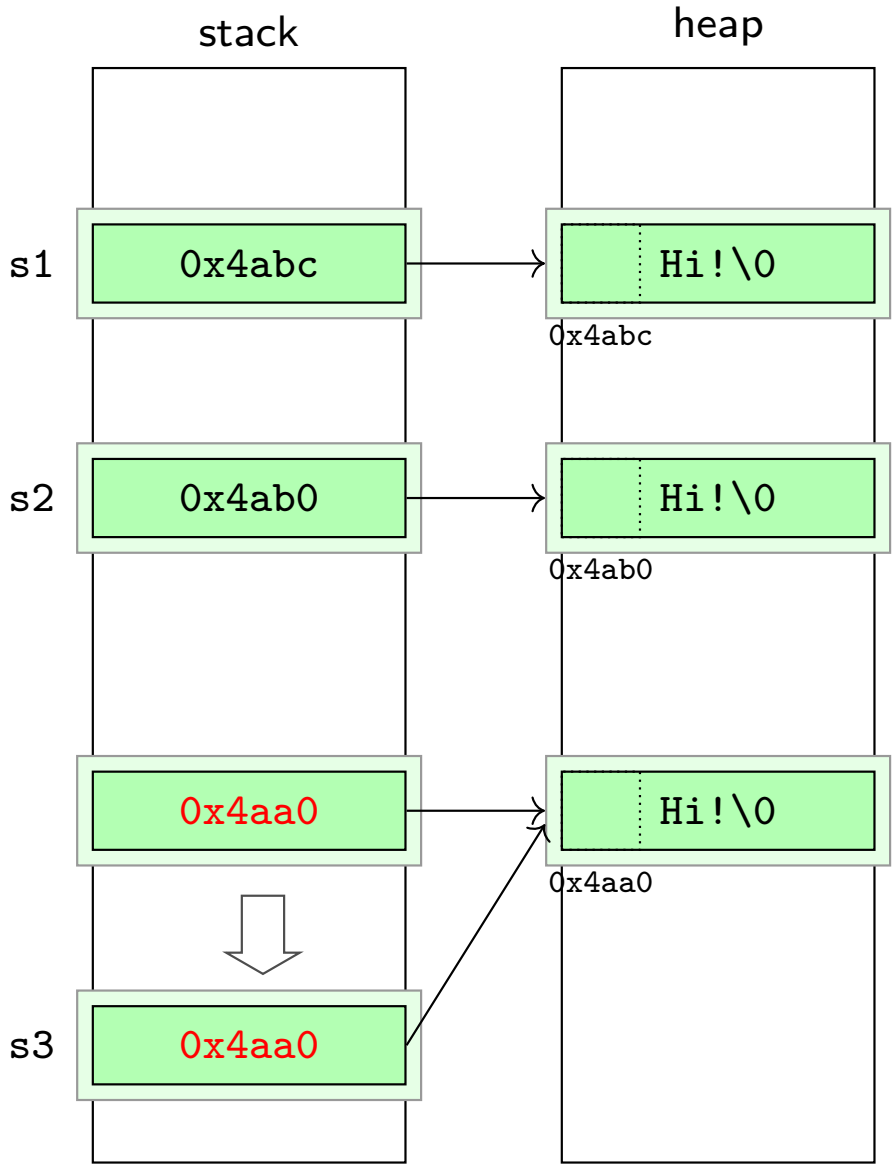
Copy vs move

```
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()};
```



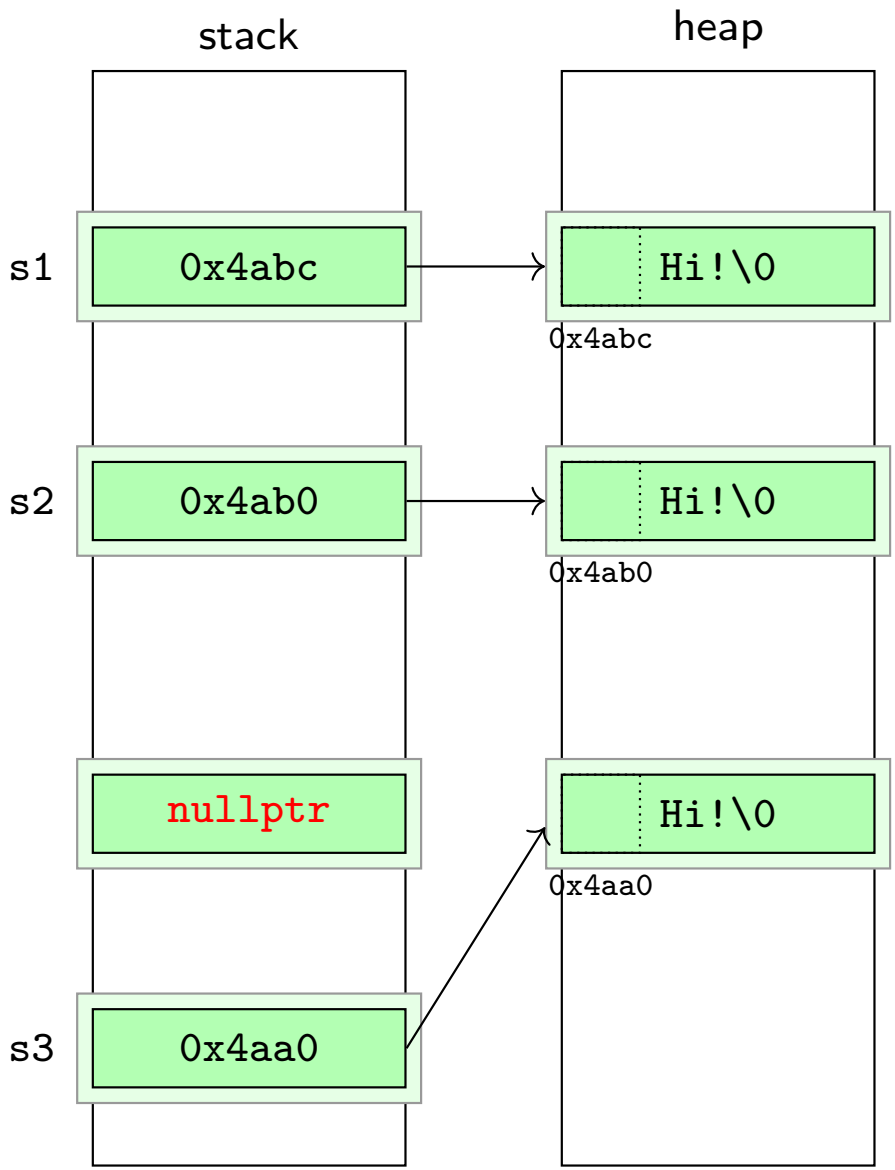
Copy vs move

```
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()};
```



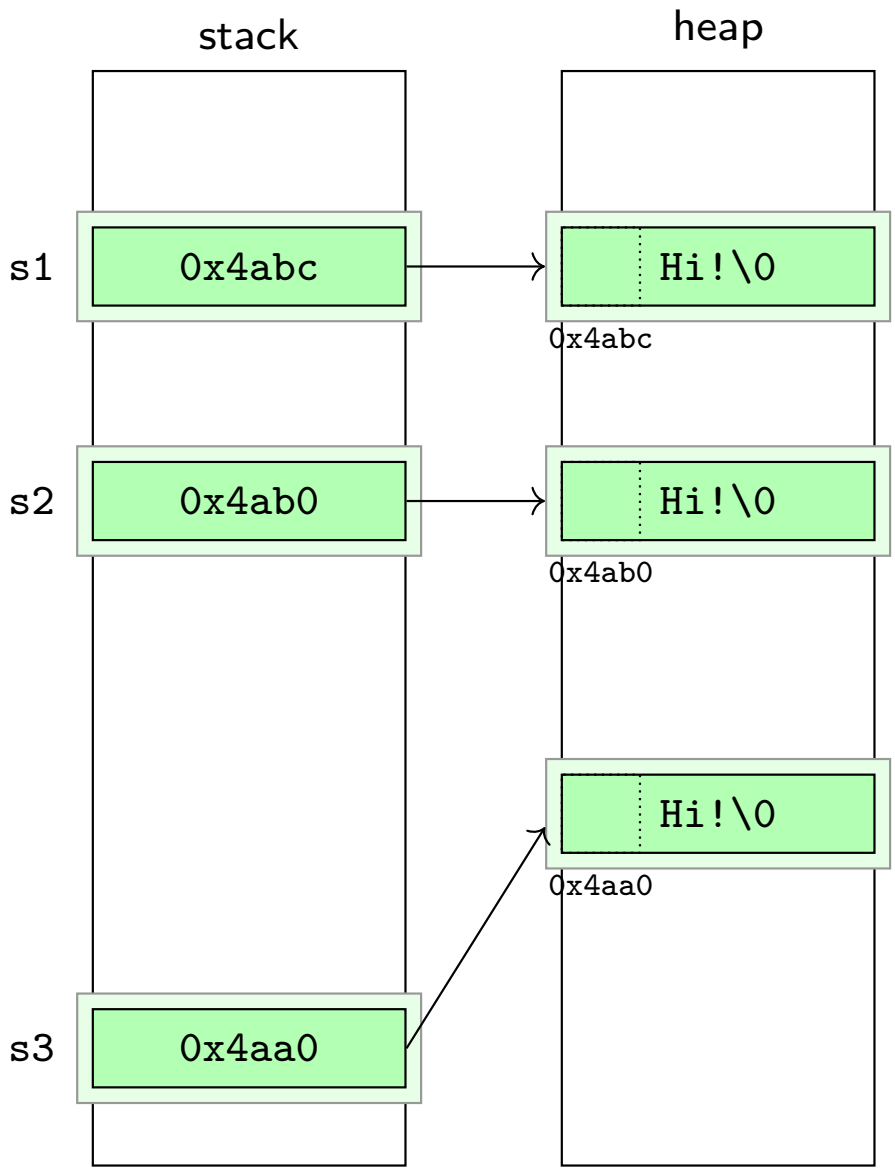
Copy vs move

```
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()};
```



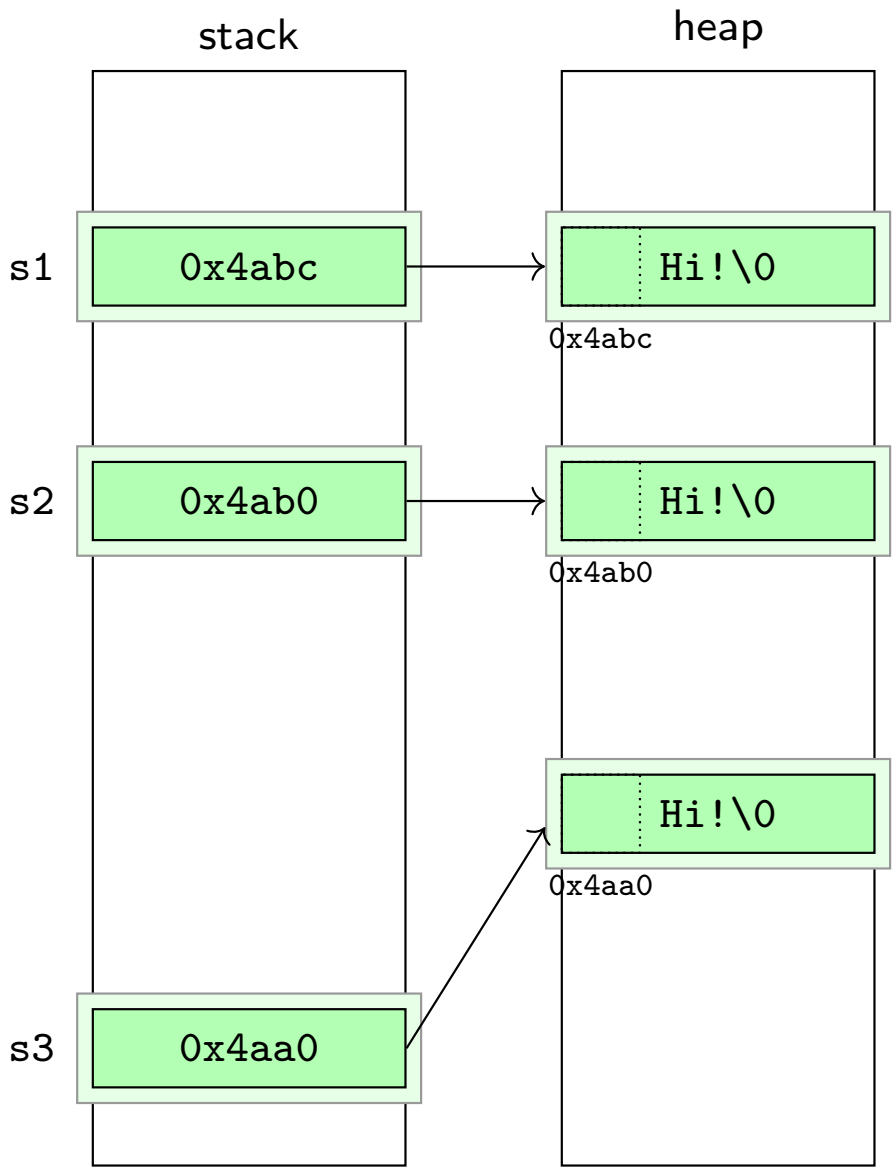
Copy vs move

```
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 String {
  // copy constructor
  String(String const& other) { ··· }
  // move constructor
  String(String&& tmp) { ··· }
};
```

Rvalue reference

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

```
class String {
  // copy constructor
  String(String const& other) { ··· }
  // move constructor
  String(String&& tmp) { ··· }
};

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

Rvalue reference

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

```
class String {
  // copy constructor
  String(String const& other) { ··· }
  // move constructor
  String(String&& tmp) { ··· }
};

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

Rvalue reference

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

```
class String {
  // copy constructor
  String(String const& other) { ··· }
  // move constructor
  String(String&& tmp) { ··· }
};

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

- Any function can accept rvalue references, not only *special member functions*

Move semantics

- Move semantics allows to properly move (i.e. transfer) the responsibility of a resource from one managing object to another
- Mainly driven by optimization considerations but also a solution to a proper management of resources

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&&) noexcept;            // move constructor
  Widget& operator=(Widget&&) noexcept; // 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&&) noexcept;            // move constructor
  Widget& operator=(Widget&&) noexcept; // 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&&) noexcept;            // move constructor
  Widget& operator=(Widget&&) noexcept; // move assignment
  ~Widget();                            // destructor
};
```

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

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

  Rule of five If you need to declare one, declare them all
  - Consider `= default` and `= delete`

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&&) noexcept;            // move constructor
  Widget& operator=(Widget&&) noexcept; // move assignment
  ~Widget();                            // destructor
};
```

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

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

  Rule of five If you need to declare one, declare them all
  - Consider `= default` and `= delete`
- For efficiency reasons, it's important that move operations are `noexcept`, which usually implies that the default constructor is `noexcept`

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}; // or, auto 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}; // or, auto 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 transferring the internals of another object

```
Widget w{make_widget()}; // or, auto w = make_widget()
```

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

```
Widget w{make_widget()}; // or, auto w = make_widget()
```

move assignment Allows to change the value of an existing object transferring 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

Move operations (cont.)

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

Move operations (cont.)

- 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

Move operations (cont.)

- 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
- `std::move` enables the use of a move operation instead of the corresponding copy operation
  - In practice it's a `static_cast` of an lvalue reference to an rvalue reference

Destruction

- At the end of a block statement (i.e. when the closing brace } is encountered) all objects of automatic storage duration are automatically *destroyed*

Destruction

- At the end of a block statement (i.e. when the closing brace } is encountered) all objects of automatic storage duration are automatically *destroyed*
- Destroying an object means
  - for a class type, a special member function, called *destructor*, if present, is run
  - all the sub-objects (e.g. data members) are destroyed, recursively
  - the storage is freed

Destruction

- At the end of a block statement (i.e. when the closing brace } is encountered) all objects of automatic storage duration are automatically *destroyed*
- Destroying an object means
  - for a class type, a special member function, called *destructor*, if present, is run
  - all the sub-objects (e.g. data members) are destroyed, recursively
  - the storage is freed
- The order of destruction is the opposite of the order of construction/definition
  - a later-defined object can use a previously-defined one

```
{
  S s{···};
  ···
  T t{s};
  ···
} // t is destroyed first, then s
```

The destructor

- The *destructor* is a special class member function that is called when an object of that class goes out of scope and has to be destroyed

The destructor

- The *destructor* is a special class member function that is called when an object of that class goes out of scope and has to be destroyed
- The purpose of the destructor is to leave no garbage behind

The destructor

- The *destructor* is a special class member function that is called when an object of that class goes out of scope and has to be destroyed
- The purpose of the destructor is to leave no garbage behind
- A class can have only one destructor, declared as `~classname()`

The destructor

- The *destructor* is a special class member function that is called when an object of that class goes out of scope and has to be destroyed
- The purpose of the destructor is to leave no garbage behind
- A class can have only one destructor, declared as ~*classname*()

```
class Widget {
  // handle to resources
 public:
  Widget(···)
  {
    // acquire resources
  }
  ~Widget()
  {
    // release resources
  }
  ···
};
```

RAII idiom

- Resource Acquisition Is Initialization

RAII idiom

- Resource Acquisition Is Initialization
  - The constructor accepts/acquires a resource

RAII idiom

- Resource Acquisition Is Initialization
  - The constructor accepts/acquires a resource
  - The destructor releases it

RAII idiom

- Resource Acquisition Is Initialization
  - The constructor accepts/acquires a resource
  - The destructor releases it
- The object is responsible for the correct lifetime management of that resource

RAII idiom

- Resource Acquisition Is Initialization
  - The constructor accepts/acquires a resource
  - The destructor releases it
- The object is responsible for the correct lifetime management of that resource
- Guaranteed no garbage left behind, even in presence of exceptions

= 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_v<Widget>);
static_assert(!std::is_default_constructible_v<Widget>);
```

= 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_v<Widget>);
static_assert(std::is_default_constructible_v<Widget>);
```

= delete

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

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

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


static_assert(!std::is_copy_constructible_v<Widget>);
static_assert(!std::is_copy_assignable_v<Widget>);
```

= 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

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


static_assert(!std::is_copy_constructible_v<Widget>);
static_assert(!std::is_copy_assignable_v<Widget>);

Widget w1{···}, w2{···};
auto w3 = w1; // 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

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


static_assert(!std::is_copy_constructible_v<Widget>);
static_assert(!std::is_copy_assignable_v<Widget>);

Widget w1{···}, w2{···};
auto w3 = w1; // error
w2 = w1;      // 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

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


static_assert(!std::is_copy_constructible_v<Widget>);
static_assert(!std::is_copy_assignable_v<Widget>);

Widget w1{···}, w2{···};
auto w3 = w1; // error
w2 = w1;      // error
```

Implementing copying

- The copy constructor typically takes an object of the same class by const reference

```
Widget(Widget const& other) : ··· {···}
```

- The copy assignment operator typically:
  - takes an object of the same class by const reference
  - returns the object itself by reference
  - checks for auto-assignment

```
Widget& operator=(Widget const& other)
{
  if (this != &other) {
    ···
  }
  return *this;
}
```

Implementing moving

- If there are no resources involved, a move operation in fact becomes a copy operation

Implementing moving

- If there are no resources involved, a move operation in fact becomes a copy operation
- Move operations typically modify the source object, to transfer its internal resources
- After a move the two objects are typically different

Implementing moving

- If there are no resources involved, a move operation in fact becomes a copy operation
- Move operations typically modify the source object, to transfer its internal resources
- After a move the two objects are typically different
- A move should leave the source object in a *"valid but unspecified state"*
  - i.e. **the class invariant is preserved**
  - not always easy, especially for the move constructor

Implementing moving

- If there are no resources involved, a move operation in fact becomes a copy operation
- Move operations typically modify the source object, to transfer its internal resources
- After a move the two objects are typically different
- A move should leave the source object in a *"valid but unspecified state"*
  - i.e. **the class invariant is preserved**
  - not always easy, especially for the move constructor
- Move operations should be declared `noexcept`, which means "this function/operation doesn't fail" (more or less)

Implementing moving

- If there are no resources involved, a move operation in fact becomes a copy operation
- Move operations typically modify the source object, to transfer its internal resources
- After a move the two objects are typically different
- A move should leave the source object in a *"valid but unspecified state"*
  - i.e. **the class invariant is preserved**
  - not always easy, especially for the move constructor
- Move operations should be declared `noexcept`, which means "this function/operation doesn't fail" (more or less)
- The automatically generated operations may not be correct
- The move operations can be explicitly provided or suppressed

Implementing moving (cont.)

- The move constructor typically takes an object of the same class by (non-const!) rvalue reference

```
Widget(Widgets&& other) : ··· {···}
```

- The move assignment operator typically:
  - takes an object of the same class by (non-const) rvalue reference
  - returns the object itself by reference
  - can assume that the argument is a temporary, hence not itself, so it need **not** check for auto-assignment

```
Widget& operator=(Widget&& other)
{
  ...
  return *this;
}
```

- Rule of thumb: `std::swap` must work

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

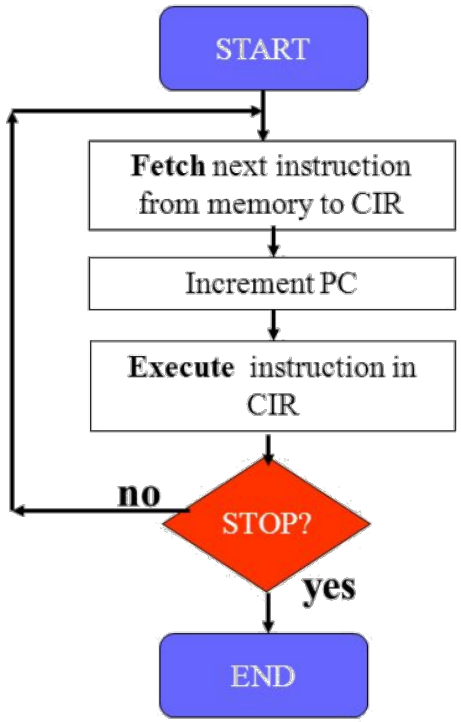
Hands-on

- C++ $\rightarrow$ Move operations
- Open the program `string.cpp` and complete the existing code to:
  - Implement the set of the special member functions so that `String` is copyable and movable
  - Once you have completed the lecture on Memory, you can come back to this code and use a `unique_ptr` instead of a raw pointer in the private part of `String`
  - ...
- Use the class `Tracking` in `tracking.hpp` in various situations, such as copying and moving `Tracking` objects or passing/returning them to/from functions, and see which special member functions are called in each situation.

Outline

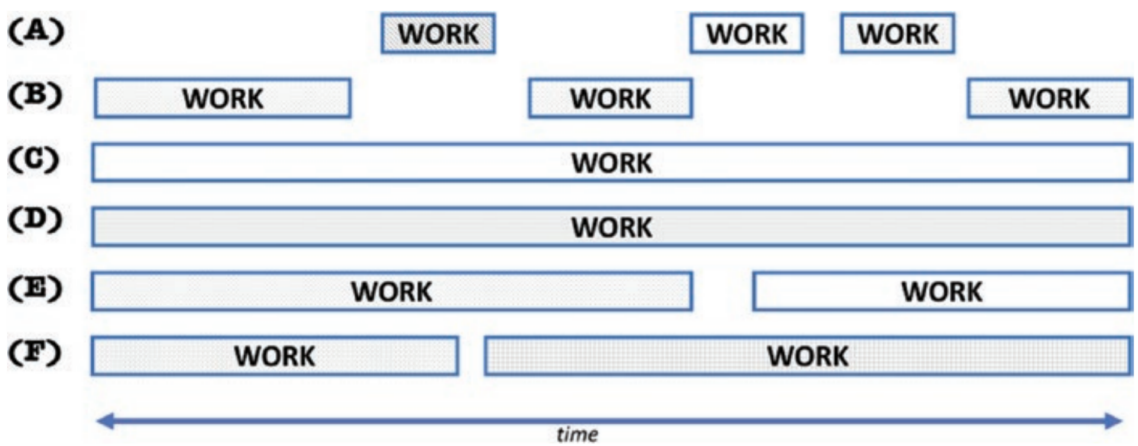
What is a Thread?

- A *thread of execution* is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread
  - ⇒ Each thread has its own Program Counter, which is the register that keeps the address of the next instruction to execute
  - ⇒ Each thread starts from the invocation of a function
  - ⇒ Each thread has its own Stack memory



Concurrency and parallelism

- A C++ program can have more than one thread running *concurrently*. The execution can be viewed as an interleaving of all its threads
  - Usually it's the operating system that schedules the execution of a thread on an available processor
- On a multi-core processor, these threads can also run in *parallel*, i.e. at the same time



Threads share memory

- Every thread in a program can potentially use every object and function in a program
  - All threads can access the memory of any other thread
  - This simplifies collaboration among threads, but it requires care

Let's start a thread

```
#include <iostream>
#include <thread>

auto f(int i)
{
    std::cout << "Hello from thread " << i << '\n';
}

int main()
{
  std::thread t{f, 0};
  std::cout << "Hello from the main thread\n";
  t.join();
}
```

- We need a *top-level invocation function*
- We need to *join* the thread before the corresponding object is destroyed
  - Joining is a blocking operation
- The main thread can do other things while the spawned thread is running

Let's start a thread (cont.)

```
#include <iostream>
#include <thread>

int main()
{
  std::jthread t{[](int i) {
      std::cout << "Hello from thread " << i << '\n';
    },
    0
  };
  std::cout << "Hello from the main thread\n";
}
```

- The top-level invocation function can be a lambda expression
- In C++20 we can use `std::jthread`, which joins automatically in its destructor
  - Compile with `g++ -std=c++20 threads.cpp`
  - Note that in this case the destructor is a blocking operation

Let's start many threads

```
#include <iostream>
#include <thread>
#include <vector>

int main()
{
  auto f = [](int i) {
    std::cout << "Hello from thread " << i << '\n';
  };
  std::vector<std::jthread> v;
  for (int i0; i != 5; ++i) {
    v.emplace_back(f, i);
  }
}
```

A typical run of the program can give:

```
Hello from thread 0
Hello from thread Hello from thread 3
Hello from thread 5
Hello from thread 1
Hello from thread 4
2
```

Data race

- A race condition occurs when multiple threads read from and write to the same memory without proper *synchronization*
- The race may sometimes finish correctly and therefore complete without apparent errors, while at other times it may finish incorrectly
- If a data race occurs, the behavior of the program is **undefined**

Mutual exclusion

- The main synchronization mechanism is a *mutex*, short for *mutual exclusion*
- It is used to protect a *critical section* of code, i.e. a piece of code that only one thread at a time can execute
- C++ offers the `std::mutex` type, typically used together with a `std::scoped_lock`, which applies the RAII idiom
- In our case, the critical section is the use of the shared object `std::cout`, whose access needs to be synchronized
  - Note that access to data private to a thread or to shared read-only data do not need synchronization

Let's start many threads correctly

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

int main()
{
  std::mutex cout_mx;
  auto f = [&](int i) {
    std::scoped_lock l(cout_mx);
    std::cout << "Hello from thread " << i << '\n';
  };
  std::vector<std::jthread> v;
  for (int i0; i != 5; ++i) { v.emplace_back(f, i); }
}
```

A typical run of the program now gives:

```
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 0
Hello from thread 4
```