EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Applied Bioinformatics

Programming in C/C++

- Smart Pointers & Move -

Lifetime & Scope

- repetition
- automatic storage duration
- static storage duration
- dynamic storage duration

Global and Local Variables

```cpp
#include <iostream>

char c = 'C';                                              // variable at global scope

int main() {

  char d = 'D';                                            // local variable, function body scope

  cout << c << ' ' << d << '\n';

  { // introduces new local scope
    char e = 'E';
    cout << c << ' ' << d << ' ' << e << '\n';
  } // variable e goes "out-of-scope" here, lifetime ends

  // std::cout << e << '\n'; // error: e not defined
}
```

- How long does data remain in memory?

Storage Duration (how)

- **automatic storage duration:**
  - allocated at beginning of the enclosing code block and deallocated at end

- **static storage duration:**
  - allocated when the program begins and deallocated when the program ends
  - only one instance of the object exists

- **thread storage duration:**
  - allocated when the thread begins and deallocated when the thread ends
  - one instance per thread

- **dynamic storage duration:**
  - allocated "manually" by **`new`** and deleted by **`delete`**

Storage Duration (what)

- **automatic storage duration:**
  - all local objects (except those declared `static`, `extern` or `thread_local`)

- **static storage duration:**
  - those declared with `static` or `extern`
  - all objects declared at namespace scope (including global namespace),

- **thread storage duration:**
  - objects declared `thread_local`; `thread_local` can appear together with
  - `static` or `extern`

- **dynamic storage duration:**
  - objects allocated "manually" by **`new`**

Storage Duration (what)

- **automatic storage duration:**
  - usually allocated on the **stack**
- **static storage duration:**
  - usually has separate memory region
- **thread storage duration:**
  - usually has separate memory region
- **dynamic storage duration:**
  - usually allocated on the **heap**

For **automatic and static storage duration,** allocation and de-allocation happens automatically (taken care by the compiler)

For **dynamic storage duration (at run-time)**, allocation and de-allocation can happen during any time of program execution based on **run-time** decisions.

`static` variable

```
void foobar() {
  static size_t i = 0; // static variable in local scope
  cout << ++i << endl;
}

foobar();   // 1
foobar();   // 2
```

- The **static "local" variable** `i` **exists only once** in the whole program
- `i` is **allocated** on program start and exists until the end of the program
- Each invocation of `foobar()` increments and prints `i`
- The name `i` is **only visible locally within the function**, not globally

`static` member variable

```
struct MyStruct {
    static size_t i = 23;        // static data member
    size_t j = 42;               // non-static data member
};

std::cout << MyStruct::i;        // access class member - no instance

MyStruct s;
std::cout << s.i;                // works too, but not recommended
```

- The **static member variable** `i` **exists also only once** in the whole program
- Allocated on program start, exists until the end of the program
- If **public**, it is **globally visible** under the name `MyStruct::i`
- (Detail: if an object `s` of type `MyStruct` exists, `i` is also accessible as `s.i` but using `MyStruct::i` is clearer/less ambiguous.)

Revisit: Stack vs. Heap

- **stack**: for all local variables, call-stack, …
- **heap**: for everything being explicitly allocated by `new (malloc)`
- **data segment:** global non-constant objects, e.g., static
- **code segment:** global constant objects

Remember:

- Slower to allocate/access memory from the heap, i.e. dynamic storage is slower

- Heap can be much larger: grow to be the entire system memory, while the stack is often limited to a few MBs, i.e. more dynamic storage is possible

Being able to allocate user defined types on the stack is one reason C and C++ can be faster than other languages (e.g., Java allocates Java objects on the heap).

Pointers Revisited

Recall: Dynamic Array

Example:

```
std::cin >> s;                    // ask the user for the size

int64_t *arr = new int64_t[s]{}; // allocate and initialize elements to zero

for (size_t j = 0; j < s; ++j)   // user enters numbers
  std::cin >> arr[j];

for (size_t j = 0; j < s; ++j)   // print it
  std::cout << arr[j] << ' ';

delete[] arr;                     // free memory
```

Recall:

- Dynamically allocate a block of memory (here of type `int64_t`)
- Pointer stores address of the first element
- Pointer does not know that it represents an array (thus, doesn't know its size).
- Memory needs to be freed at the end of scope

Recall: Working with Raw Pointers

```cpp
int64_t *arr = new int64_t[7]{}; // pointer that owns the array

arr[0] = 42;                     // assign to 0th value
std::cout << arr[0];             // prints 0th element: 42

int64_t *ptr = arr;              // create pointer to begin
*ptr = 23;                       // assign through pointer
std::cout << *it;                // prints 0th element, now: 23

++it;                            // incrementing pointer moves to next element
std::cout << *it;                // prints 1st element ("0")

it += 3;                         // moves to 4th...

delete[] arr;
```

- Pointers into arrays allow arithmetic operations and work similar to **iterators**

Recall: Working with Raw Pointers – Danger Zone

```
int64_t *arr = new int64_t[7]{}; // pointer to array
int64_t *it = new int64_t{7};    // pointer to a single int64_t, initialized to 7
it = arr;


*it = 23;
std::cout << *it;
++it;
std::cout << *it;
it += 3;
delete it;

delete[] arr;
```

Can you spot two errors?

Recall: Working with Raw Pointers – Danger Zone

```
int64_t *arr = new int64_t[7]{}; // pointer to array
int64_t *it = new int64_t{7};    // pointer to a single int64_t, initialized to 7
it = arr;                        // address overwritten
                                 // access to single int64_t no longer possible
                                 // -> memory leak
*it = 23;                        // assign through pointer
std::cout << *it;                // 23
++it;                            // incrementing pointer moves to next
std::cout << *it;                // prints 1st element: 0
it += 3;                         // moves to 4th...
delete it;                       // calling delete on single element in the array
                                 // free(): invalid pointer. Aborted
delete[] arr;                    // freeing up the array
```

- Overwriting addresses
- Using address to invalid pointer
- Pointer arithmetic on pointers to single element

Recall: Working with Raw Pointers – Danger Zone

```
class MyVec {
  int64_t *arr;
public:
  MyVec(int s) { arr = new int64_t[s]{}; }

  ~MyVec() { delete[] arr; }

 *getArray() { return arr; }
};
```

```
int main() {
  MyVec vec(5);

  int64_t *arr = vec.getArray();
  // …
  delete[] arr; // free the array

  return 0;
} // <- crashes with double-free
```

- Overwriting addresses
- Using address to invalid pointer
- Pointer arithmetic on pointers to single element
- Double free on the same address

Working with Raw Pointers

- can represent **single values OR arrays**

- hold **new values** (ownership) or **existing values** (like references)

- But: ownership "not implemented" → need to **`delete / delete[]`** yourself!

- Working with raw pointers gets particularly complex if **exceptions** occur as there is not anymore only a single code location where pointers need to be freed. (later more…)

"**With great power comes great responsibility**"

ancient adage

MyVector v0.1

Classes help us to hide some of the complexity of pointers but also difficult to get right:

```
template <typename T> class MyVector {
private:
  T *data{}; // = nullptr
  size_t size{};
public:
  MyVector(size_t const s) {      // allocate on construction
    size = s;
    data = new T[s]{};
  }
  ~MyVector() { delete[] data; }  // delete on construction
  MyVector() = default;           // too lazy… let's just add a few default ones
  MyVector(MyVector const &) = default;
  MyVector &operator=(MyVector const &) = default;
  T *begin() { return data; }     // too lazy, let's just use pointers as iterators
  T *end() { return data + size; }
};
```

- We allocate on construction and delete on destruction! Problem solved right?

MyVector v0.1

```
MyVector<size_t> v1{7};
MyVector<size_t> v2{9};
v1 = v2; // wrong! =default does not what we want

*(v1.begin()) = 3; // wrong! alters v2!
```

```
template <typename T
private:
  T *data{}; // = nu
  size_t size{};
public:
  MyVector(size_t const s) {      // allocate on construction
    size = s;
    data = new T[s]{};
  }
  ~MyVector() { delete[] data; }  // delete on construction
  MyVector() = default;           // too lazy… let's just add a few default co
  MyVector(MyVector const &) = default;
  MyVector &operator=(MyVector const &) = default;
  T *begin() { return data; }     // too lazy, let's j                  tors
  T *end() { return data + size; }
};
```

Address-sanitizers or valgrind might help to detect these errors

- Any change to **`v1`** will alter **`v2`**
- **`v1`**’s array no longer accessible, will not be freed - **`v2`**’s array will be deleted twice
- Same problems as before!

MyVector v0.2

```
public:
  MyVector() = default;

  MyVector &operator=(MyVector const &rhs) {
    size = rhs.size;      // copy size
    delete[] data;
    data = new T[size];  // deallocate and reallocate
    for (size_t i = 0; i < size; ++i)
      data[i] = rhs.data[i]; // copy every element
    return *this;
  }

  MyVector(MyVector const &rhs) { operator=(rhs); } // copy constructor
```

- `=default` did the wrong thing here
- On the right track, still far way to `std::vector` :
  - **user-defined copy assignment operator** to handle deallocation and reallocation, as well as copying of the elements
  - **copy construction** is defined as delegating to copy assignment

Pitfalls of Raw Pointers

Summary:

- Using pointers is a frequent source of bugs, e.g.:
  - Memory leaks (forgetting to delete)
  - Double delete (deleting something twice)
  - Use-after-free (using an object after it was deleted)

- Pointers are a legacy from C, even more problematic in C++ because of exceptions

- Many uses of pointers can be easily replaced with references in C++

- C++ has introduced **smart pointers** to avoid typical problems

Smart Pointers `<memory>`

- Modern C++ classes to simplify dynamic memory management
- Expression of ownership (who owns what?)
- Resource Acquisition is Initialization (RAII)

Smart Pointers

```
std::unique_ptr<T> / std::unique_ptr<T[]>:
```

- cannot be copied
- deletes its pointee when its own lifetime ends
- for "unique" access to some resource

```
std::shared_ptr<T> / std::shared_ptr<T[]>:
```

- can be copied; copies share data
- when lifetime of **last copy** ends, it deletes its pointee
- for *shared* access to some resource

```
std::weak_ptr<T> / std::weak_ptr<T[]>:
```

- initialized by **`shared_ptr`**, but could have longer lifetime
- in order to access the value, explicit conversion to **`make_shared`** is necessary
- If it fails **`shared_ptr`** already expired
- for *non-owning* access. Sometimes required to break cycles between **`shared_ptr`**.

A smart pointer is a **vocabulary type** – it's domain: clearly express **ownership** and **resource management**

MyVector v0.3 – `std::unique_ptr`

Ok let's try to improve our class with these new types:

```
template <typename T> class MyVector {
private:
  size_t size{};
  std::unique_ptr<T[]> data{};     // we have unique ownership -> unique_ptr

public:
  MyVector() = default;
  ~MyVector() = default;           // success! can now be defaulted
  MyVector(size_t const s) : size(s), data(new T[s]{}) {}
```

- We solved the problem of the automatic free at end of life/scope.
- But how to copy? **`unique_ptr`** cannot be copied, they are… unique!

MyVector v0.3 – `std::unique_ptr`

We need a custom `operator=`:

```
MyVector &operator=(MyVector const &rhs) {
  size = rhs.size;           // copy size
  data.reset(new T[size]); // deallocate and reallocate

  for (size_t i = 0; i < size; ++i)
    data[i] = rhs.data[i]; // copy every element
  return *this;
}

MyVector(MyVector const &rhs) { operator=(rhs); } // as before…
```

- Same as the raw pointer version, except we replaced:

```
delete[] data;
data = new T[size]; // deallocate and reallocate
```

- Our own `std::vector` class is taking shape. It now also supports assignment and copy construction.

Resource Acquisition is Initialization (RAII)

Important concept:

- An idiom that describes **tying resource management to the lifetime of an object**:
  - **resource acquisition** (memory allocation) during construction
  - **resource release** (memory deallocation) upon destruction

- Introduce **class invariant**: *“Resource is accessible as long as object exists.”*

- Clear **ownership** makes it easier to reason about resource management and is **safer** in most situations.

- **Smart pointers** can tie the lifetime and management of a heap allocated object – the pointee – to the lifetime of a stack-allocated (and automatically managed) object, the pointer

- **RAII** is a general concept, it appears outside the context of memory management: e.g., database or network connections, file access, …

Summary – Pointers / Smart Pointers

- Objects in C++ can have different lifetimes
- Objects or arrays **dynamically allocated** with `new` are usually allocated on the **heap** and **referenced by a pointer**
- This is slower, but more flexible

Recommendation:

- Avoid pointers and dynamic storage if possible. (This is C++, not C!)
  → Prefer **references** to refer to existing other variables
- If you dynamically allocate, always use **smart pointers** (instead of raw pointers), because they clean-up after themselves
- Only use raw pointers if you refer to **pre-existing memory/objects** and you need to be able to change where you point to
- In general, follow the RAII principle

Move Semantics

- r-value references
- Moving objects instead of copying them

Variables and Reference Binding

```
size_t i1{1};
size_t const i2{1};

// all valid assignments - we copy so we don't care about const:
size_t s1a = i1;
size_t s1b = i2;
size_t s1c = size_t{1};       // assign from temporary object with value 1

size_t const s2a = i1;
size_t const s2b = i2;
size_t const s2c = size_t{1};
```

Quiz:

```
// Now assignment to references. Which ones are valid?
size_t &s3a = i1;
size_t &s3b = i2;
size_t &s3c = size_t{1};

size_t const &s4a = i1;
size_t const &s4b = i2;
size_t const &s4c = size_t{1};
```

Variables and Reference Binding

```
size_t i1{1};
size_t const i2{1};

// all valid assignments - we copy so we don't care about const:
size_t s1a = i1;
size_t s1b = i2;
size_t s1c = size_t{1};       // assign from temporary object with value 1

size_t const s2a = i1;
size_t const s2b = i2;
size_t const s2c = size_t{1};
```

Quiz:

```
// Now assignment to references. Which ones are valid?
size_t &s3a = i1;
size_t &s3b = i2;                 // error! Can't bind non-const ref to const (i2)
size_t &s3c = size_t{1};          // error! Can't bind to temporary from non-const ref

size_t const &s4a = i1;
size_t const &s4b = i2;
size_t const &s4c = size_t{1};  // also works. We can bind to temporary from const ref
```

Variables and Reference Binding

- If we copy, we don't care about const-ness.
- We can only bind **`&`** to **`non-const`**-objects
- **`&`** cannot bind to a temporary value, but we saw that **`const &`** can

```
void print(string const &s)
{
  cout << s;
}

string sf{"foo"};
print(sf);    // pass by ref
print("bax"); // <- temporary
```

- The **`const &`** binds to the temporary and extends its lifetime to match its own lifetime
- This **special feature** of **`const &`** allows us to write interfaces that handle references and (temporary) values

- Q: Why can't non-**`const &`** do this?
- A: **`void print(std::string & s)`** would suggest that the function's purpose is to permanently change an outside parameter. Since the temporary does not exist outside of the function, this would not make sense and easily lead to errors.

MyVector v0.3

```
...
public:
  MyVector(MyVector const &rhs);
  MyVector &operator=(MyVector const &rhs);
...
```

- This behavior of `const &` is the reason why the **copy constructor** / **copy assignment operator** also accept temporaries.
- But do copy constructor and assignment operator handle them optimally?

MyVector v0.4alpha – Move Assignment

- For temporaries, we would like to just **transfer ownership** instead of (re)allocating/copying.
- There exists a way in C++ to indicate that we want to bind references to temporaries. To mark these special references C++ uses `&&`.

```
MyVector &operator=(MyVector && rhs) {
    size = rhs.size;    // copy size
    delete[] data;      // free old data
    data = rhs.data;    // point data to rhs' data
    rhs.data = nullptr; // prevent rhs from deleting
    return *this;
  }
```

- This looks correct, but we have two problems:
  - we still need the old assignment implementation for copying from references
  - we can't take `rhs` by something that is `const`, because we are changing it

Variables and Reference Binding

```
size_t i1{1};
size_t const i2{1};
// assignment to copy -> all valid assignments …
```

```
// assignment to references
size_t &s3a = i1;
size_t &s3b = i2;                  // error! Can't bind non-const ref to const (i2)
size_t &s3c = size_t{1};          // error! Can't bind to temporary from non-const ref

size_t const &s4a = i1;
size_t const &s4b = i2;
size_t const &s4c = size_t{1};  // works

// New: assignment to r-value references
size_t && s4a = i1;               // error! Only binds to r-value
size_t && s4b = i2;               // error! Only binds to r-value
size_t && s4c = size_t{1};       // works! Binds to r-value
```

- `size_t &&` is an **r-value** reference.
- It only binds to **r-values** i.e. "temporary objects", string literals, etc.
- r as in "exists only on the right side of assignments" and don't have a **name**

Explicit Moving – `std::move()`

```
std::vector v0{1, 2, 3, 5};
std::vector v1{v0};              // v1 is copy of v0
std::vector v2{std::move(v0)}; // contents of v0 moved into v2
```

- It is possible to mark a non-temporary object as temporary using **`std::move`**
- Note that **`std::move`** **`does not move!`**
- It merely casts its argument to **`&&`** so that special functions (e.g. the **move constructor**) are chosen and can provide a better implementation (e.g., that "steal" the contents)
- **Attention**: this leaves the moved-from object in "valid, but unspecified state" → you cannot use them afterwards, potential for errors!
- So, when is this useful?

Explicit Moving – `std::move()`

```
void foo(MyVector<size_t> && s) {
  /*…*/
}

void bar(MyVector<size_t> && s) {
  /* … do something with s and forward to foo …*/
  foo(std::move(s));
}

bar(MyVector<size_t>{77777777}); // call with large temporary
```

Frequent use-case – **ownership transfer**, **preventing expensive copies**.

- When you want to **pass a temporary to the next function**, you need to **explicitly** call **`std::move`** again because it loses it's "temporary-ness" in every scope again.

- One way to reason about it:
  - `MyVector<size_t>{77777777}` is a temporary: it doesn't have a name (r-value)
  - In `void bar(MyVector<size_t> && s)` it suddenly has a name "s" (l-value) - so we need to use `std::move` to mark it again as temporary (r-value)

Move Constructor

Move constructor - motivating example

- **`std::swap`** is the STL function to swaps two objects
- In sorting algorithms, the speed of swapping can make a **huge difference**!
- Two objects of a **move-constructible** and **move-assignable** type can be efficiently swapped via **`std::swap`**
- Missing in our `MyVector` class: the move constructor

```
// Move constructor just uses the move assignment operator=
MyVector(MyVector&& rhs) noexcept // promise that this does not throw an exception.
{
  *this = std::move(other);
}

MyVector<size_t> mv1{7};
MyVector<size_t> mv2{9};
std::swap(mv1, mv2); // contents now swapped efficiently
```

```
  // STL container can be swapped
  std::vector<std::string> v1{123456};
  std::vector<std::string> v2{789012};
  std::swap(v1, v2); // big but swapped in (nearly) no-time
```

Not Using the Move Constructor

```
class NoMove {
  int *data; // raw data pointer
public:
  // Constructor
  NoMove(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  NoMove(const NoMove &source) : NoMove{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Destructor
  ~NoMove() {
    if (data != nullptr)
      cout << "Destructor " << *data << endl;
    else
      cout << "Destructor - nullptr" << endl;
    delete data; // Free memory
  }
};
```

```
int main() {
  // Create vector of NoMove Class
  vector<NoMove> vec;
  // Inserting object of NoMove class
  vec.push_back(NoMove{1});
  vec.push_back(NoMove{2});
  return 0;
}
```

Toy Example:

- How many constructors are called?
- Which constructors are called (e.g., how many copies are made by the copy constructor)
- If we simply add two temporaries to a `std::vector`
- With/without move constructor

[based on www.geeksforgeeks.org]

Not Using the Move Constructor

```cpp
class NoMove {
  int *data; // raw data pointer
public:
  // Constructor
  NoMove(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  NoMove(const NoMove &source) : NoMove{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Destructor
  ~NoMove() {
    if (data != nullptr)
      cout << "Destructor " << *data << endl;
    else
      cout << "Destructor - nullptr" << endl;
    delete data; // Free memory
  }
};
```

```cpp
int main() {
  // Create vector of NoMove Class
  vector<NoMove> vec;
  // Inserting object of NoMove class
  vec.push_back(NoMove{1});
  vec.push_back(NoMove{2});
  return 0;
}
```

```
Constructor 1
Constructor 1
CC - Deep Copy 1
Destructor 1

Constructor 2
Constructor 2
CC - Deep Copy 2
Constructor 1
CC - Deep Copy 1
Destructor 1
Destructor 2

Destructor 1
Destructor 2
```

!?!

[based on www.geeksforgeeks.org]

Not Using the Move Constructor

```
class NoMove {
  int *data; // raw data pointer
public:
  // Constructor
  NoMove(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  NoMove(const NoMove &source) : NoMove{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Destructor
  ~NoMove() {
    if (data != nullptr)
      cout << "Destructor " << *data << endl;
    else
      cout << "Destructor - nullptr" << endl;
    delete data; // Free memory
  }
};
```

```
int main() {
  // Create vector of NoMove Class
  vector<NoMove> vec;
  // Inserting object of NoMove class
  vec.push_back(NoMove{1});
  vec.push_back(NoMove{2});
  return 0;
}
```

```
Constructor 1              construct temporary
Constructor 1              construct and copy
CC - Deep Copy 1             in vec
Destructor 1               destroy temporary
after first push_back
Constructor 2              construct temporary
Constructor 2              construct and copy
CC - Deep Copy 2             to grown vec
Constructor 1              construct and copy
CC - Deep Copy 1             to grown vec
Destructor 1               destroy before grow
Destructor 2               destroy temporary
after second push_back
Destructor 1               ~vec: call ~NoMove
Destructor 2               … on both elements
```

Using the Move Constructor

```cpp
class Move {
  int *data; // raw data pointer
public:
  // Constructor
  Move(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  Move(const Move &source) : Move{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Move Constructor
  Move(Move &&source) : data{source.data} {
    // directly take over the allocated memory
    cout << "Move Constructor " << *source.data << endl;
    source.data = nullptr;
  }
  // Destructor
  ~Move() { /* as before */ }
};
```

```cpp
int main() {
  // Create vector of Move Class
  vector<Move> vec;

  // Inserting object of Move class
  vec.push_back(Move{10});
  vec.push_back(Move{20});
  return 0;
}
```

Using the Move Constructor

```
class Move {
  int *data; // raw data pointer
public:
  // Constructor
  Move(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  Move(const Move &source) : Move{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Move Constructor
  Move(Move &&source) : data{source.data} {
    // directly take over the allocated memory
    cout << "Move Constructor " << *source.data << endl;
    source.data = nullptr;
  }
  // Destructor
  ~Move() { /* as before */ }
};
```

```
int main() {
  // Create vector of Move Class
  vector<Move> vec;

  // Inserting object of Move class
  vec.push_back(Move{1});
  vec.push_back(Move{2});
  return 0;
}
```

```
Output:
Constructor 1
Move Constructor 1
Destructor - nullptr

Constructor 2
Move Constructor 2
Constructor 1
CC - Deep Copy 1
Destructor 1
Destructor - nullptr

Destructor 1
Destructor 2
```

- Much better but still one copy ???
- Any idea what we might have missed?

Using the Move Constructor

```
class Move {
  int *data; // raw data pointer
public:
  // Constructor
  Move(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  Move(const Move &source) : Move{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Move Constructor
  Move(Move &&source) : data{source.data} {
    // directly take over the allocated memory
    cout << "Move Constructor " << *source.data << endl;
    source.data = nullptr;
  }
  // Destructor
  ~Move() { /* as before */ }
};
```

```
int main() {
  // Create vector of Move Class
  vector<Move> vec;

  // Inserting object of Move class
  vec.push_back(Move{1});
  vec.push_back(Move{2});
  return 0;
}
```

```
Output:
Constructor 1               temporary
Move Constructor 1          moved to vec
Destructor - nullptr        destroy temporary
after first push_back
Constructor 2               temporary
Move Constructor 2          moved to grown vec
Constructor 1               ??? cp to grown vec
CC - Deep Copy 1            ???
Destructor 1                destroy before grow
Destructor - nullptr        destroy temp
after second push_back
Destructor 1                ~vec: call ~Move
Destructor 2                … on both elements
```

- Much better but still one copy ???
- Any idea what we might have missed?

Using the Move Constructor

```
class Move {
  int *data; // raw data pointer
public:
  // Constructor
  Move(int d) {
    data = new int; // allocate on heap
    *data = d;
    cout << "Constructor " << d << endl;
  };
  // Copy constructor - explicit
  Move(const Move &source) : Move{*source.data} {
    // copying the data by making deep copy
    cout << "CC - Deep Copy " << *source.data << endl;
  }
  // Move Constructor
  Move(Move &&source) noexcept : data{source.data} {
    // directly take over the allocated memory
    cout << "Move Constructor " << *source.data << endl;
    source.data = nullptr;
  }
  // Destructor
  ~Move() { /* as before */ }
};
```

```
int main() {
  // Create vector of Move Class
  vector<Move> vec;

  // Inserting object of Move class
  vec.push_back(Move{1});
  vec.push_back(Move{2});
  return 0;
}
```

```
Output:
Constructor 1               temporary
Move Constructor 1          moved to vec
Destructor - nullptr        destroy temporary
after first push_back
Constructor 2               temporary
Move Constructor 2          moved to grown vec
Move Constructor 1          cp to grown vec
Destructor 1                destroy before grow
Destructor - nullptr        destroy temp
after second push_back
Destructor 1                ~vec: call ~Move
Destructor 2                … on both elements
```

- `std::vector` needs our guarantee that our class does not error during a move!
- Otherwise, it uses a copy.

Using the Move Constructor

- Another improvement: if possible, reserve space to prevent vector growth

```
int main() {
  // Create vector of Move Class
  vector<NoMove> vec;
  vec.reserve(2);

  // Inserting object of Move class
  vec.push_back(Move{1});
  vec.push_back(Move{2});
  return 0;
}
```

```
Constructor 1
Constructor 1
CC - Deep Copy 1
Destructor
Constructor 2
Constructor 2
CC - Deep Copy 2
Destructor
Destructor
Destructor
```

```
int main() {
  // Create vector of Move Class
  vector<Move> vec;
  vec.reserve(2);

  // Inserting object of Move class
  vec.push_back(Move{1});
  vec.push_back(Move{2});
  return 0;
}
```

```
Constructor 1
Move Constructor 1
Destructor
Constructor 2
Move Constructor 2
Destructor
Destructor
Destructor
```

Using in-place construction with emplace

- If our goal is only fast construction but we don't care about fast moves, we can use in-place construction. Works also with our NoMove type.
- In-place construction gets rid of the move or copy and constructs the object directly in the vector. Even without a temporary.

```
int main() {
  // Create vector of Move Class
  vector<NoMove> vec;
  vec.reserve(2);

  // construction object in vector
  vec.emplace_back(1);
  vec.emplace_back(2);
  return 0;
}
```

```
Constructor 1
Constructor 2
Destructor
Destructor
```

```
int main() {
  // Create vector of Move Class
  vector<Move> vec;
  vec.reserve(2);

  // construction object in vector
  vec.emplace_back(1);
  vec.emplace_back(2);
  return 0;
}
```

```
Constructor 1
Constructor 2
Destructor
Destructor
```

- But this doesn't give us benefits for the NoMove class e.g., swap elements or vector grow still slow.

Move constructor

- In our toy example, we have seen how the move constructor reduced the number of copies to zero. **This can have an enormous speed impact in practice.**
- If we only care about the performance impact of copies made during construction, we can sometimes get away with **in-place construction** on pre-reserved vectors.
- What many do: Implement move constructor/assignment only for classes that are frequently stored in container and in practice (benchmark!) expensive to copy.

Rule of Six:

Whenever you write a specific copy constructor implement all the following six functions:

- a default constructor: `X()`
- a copy constructor: `X(const X&)`
- a copy assignment: `operator=(const X&)`
- a move constructor: `X(X&&)`
- a move assignment: `operator=(X&&)`
- a destructor: `~X()`

=default on Constructors / Destructors

```
// Default constructor
Birthday() : month{}, day{} { /* ... */ }

// Copy constructor
Birthday(Birthday const & rhs)
{
  set_day(rhs.get_day());
  set_month(rhs.get_month());
}

// Move constructor
Birthday(Birthday && rhs) { /* ... */ }

// Custom constructor
Birthday(uint16_t const m,
         uint16_t const d)
{
  set_month(m);
  set_day(d);
}

// more custom constructors …

// Destructor
~Birthday() {}
```

```
// default Default constructor
Birthday() : month{}, day{} = default;

// default Copy constructor
Birthday(Birthday const & rhs) = default;


// default Move constructor
Birthday(Birthday && rhs) = default;

// Custom constructor
Birthday(uint16_t const m, uint16_t const d)
{
    set_month(m);
    set_day(d);
}

// more custom constructors ...

// default Destructor
~Birthday() = default;
```

`=default` forcing the compiler to generate a `default`ed constructor of that signature

[from "An introduction to C++“, Hauswedell et al.]

=delete on Constructors / Destructors

```
// Default constructor
Birthday() : month{}, day{} { /* ... */ }

// Copy constructor
Birthday(Birthday const & rhs)
{
  set_day(rhs.get_day());
  set_month(rhs.get_month());
}

// Move constructor
Birthday(Birthday && rhs) { /* ... */ }

// Custom constructor
Birthday(uint16_t const m,
         uint16_t const d)
{
  set_month(m);
  set_day(d);
}

// more custom constructors …

// Destructor
~Birthday() {}
```

```
// deleted Default constructor
Birthday() : month{}, day{} = delete;

// deleted Copy constructor
Birthday(Birthday const & rhs) = delete;


// deleted move constructor
Birthday(Birthday && rhs) = delete;

// Custom constructor
Birthday(uint16_t const m, uint16_t const d)
{
    set_month(m);
    set_day(d);
}


// more custom constructors ...
// default Destructor
~Birthday() = delete;
```

- `=delete:` Compiler is expliclty prevented from generating a constructor of that signature
- Any call to that function will result in an error at compile-time
  e.g., you cannot have a local variable of type `Birthday`.

MyVector – Summary Move Construction/Assignment

```
public:
  MyVector() = default;
  MyVector(MyVector const &rhs) { /*…*/ }
  MyVector(MyVector &&rhs) { /*…*/ }
  MyVector &operator=(MyVector const &rhs) { /*…*/ }
  MyVector &operator=(MyVector &&rhs) { /*…*/ }
  ~MyVector() = default; // if using smart pointer
```

- The **move-constructor** and the **move-assignment operator** allow us to avoid unnecessary copies when the source is a *temporary*.
- By default they move-assign / move-construct all members.
- If you provide your own **copy-constructor** / **copy-assignment operator**, move-* will **never** be declared implicitly, you should define or explicitly `=default` them ("Rule-of-6").

Summary

- **r-value** references `&&` only bind to *temporary objects*.

- Used to define **move-constructors** and **move-assignment operators**, preferred overloads for temporaries.

- These can be used to "steal" dynamically allocated memory instead of copying which is more efficient (memory on the stack is always copied!)
- **r-value** references are seldomly used outside of move construction/assignment.

- If you "pass on" a temporary, you need to explicitly `std::move` it.

Remember: `std::move` is one of the biggest misnomer in the STL. **It doesn't move** (!) but performs a casts to an r-value reference (which then can lead to a move).

Flexible Types

`union` / `std::variant`

union

```
union S {
  std::int32_t n;     // occupies 4 bytes
  std::uint16_t s[2]; // occupies 4 bytes
  std::uint8_t c;     // occupies 1 byte
};                    // the whole union occupies 4 bytes

  S s = {0x12345678}; // initializes the first member, s.n is now the active member
  // at this point, reading from s.s or s.c is undefined behavior

  std::cout << std::hex << "s.n = " << s.n << '\n';
  s.s[0] = 0x0011; // s.s is now the active member

  // at this point, reading from n or c is UB but most compilers define it
  std::cout << "s.c is now " << +s.c << '\n' // 11 or 00, depending on platform
            << "s.n is now " << s.n << '\n'; // 12340011 or 00115678
```

- Representation of different type in the same memory location
  - Access to different types by ‘member’
  - Reading out the different types be reinterpreting the bit pattern (your own risk)
- **union** is old C++-style
- Extra variable needed if we need to store what the current active member is

std::variant

```
std::variant<int, float> v, w;
v = 42;             // variant v contains int
w = 3.1415927f;     // variant w contains float

int i = std::get<int>(v);
assert(42 == i); // succeeds
w = std::get<int>(v);       // w now contains int
w = std::get<0>(v);         // same effect as the previous line
w = v;                      // same effect as the previous line
//  std::get<double>(v);    // error: no double in [int, float]
//  std::get<3>(v);         // error: valid index values are 0 and 1

try {
  std::get<float>(w); // w contains int, not float: will throw an exception
}
catch (const std::bad_variant_access& ex) {
    std::cout << ex.what() << '\n';
}
```

- A modern `union` type
- Strongly typed access
  - Access by requesting type (might fail – but can be caught by exception)
- Represented type can change during run-time

Associative Containers
set & map

Associative Containers – Overview

Ordered	Unordered	Description
`std::set`	`std::unordered_set`	collection of unique keys
`std::multiset`	`std::unordered_multiset`	collection of keys
`std::map`	`std::unordered_map`	collection of key-value pairs; keys unique
`std::multimap`	`std::unordered_multimap`	collection of key-value pairs

- Ordered associative containers are sorted by key, operations are in O(log(n)).
- Unordered associative container are not sorted, operations are in O(1) – using hashes
- Maps are the most popular associative containers. ("dictionaries" in other languages)

Associative Containers – `std::map`

```cpp
//        key             value
std::map<std::string, size_t> name_frequency{};

//              key          value
name_frequency["Horst"] = 7000; // [] creates element if not found
name_frequency["Angela"] = 5439;
name_frequency["Horst"] = 6999; // overwrites previous value
name_frequency.at("TEST");      // runtime error: doesn’t exists
name_frequency["TEST"];         // creates element with key Test and value 0

for (auto const &[name, freq] : name_frequency)
  std::cout << name << " appears " << freq << " times!\n";
```

- The element type of the map is a `pair` so we can use **structured bindings** to decompose the `pair` into individual variables.
- The first line printed will be `"Angela..."`, because elements are sorted
- Each `[]` takes O(log(n)), ordered map usually implemented as tree.
- Access with [] creates element by default-construction or zero-initialization (primitive types) if entry does not exist.

Associative Containers – `std::unordered map`

```
//                          key            value
std::unordered_map<std::string, size_t> name_frequency{};

//                 key            value
name_frequency["Horst"] = 7000; // [] creates element if not found
name_frequency["Angela"] = 5439;
name_frequency["Horst"] = 6999; // overwrites previous value
name_frequency.at("TEST");      // runtime error: doesn't exists
name_frequency["TEST"];         // creates element with key Test and value 0

for (auto const &[name, freq] : name_frequency)
  std::cout << name << " appears " << freq << " times!\n";
```

- The element type of the map is a **`pair`** so we can use **structured bindings** to decompose the **`pair`** into individual variables.
- It is **undefined which element is printed first**!
- Each **`[]`** takes O(1); unordered map usually implemented as hash table.
- Access with [] creates element by default-construction or zero-initialization (primitive types) if entry does not exist.

Summary

- Smart Pointer

- Move semantics
  - r-value reference
  - move constructor
  - move assignment

- union / variant

- **`std::(unordered_)(multi)set`**
- **`std::(unordered_)(multi)map`**