Modern C++

Jakub Zalewski

Adva Optical Networking

December 5, 2017

Motivation

Motivation

- Adoption of modern C++ in ADVA

Motivation

- Adoption of modern C++ in ADVA
- Show gains of using the C++11/C++14/C++17 standards

Motivation

- Adoption of modern C++ in ADVA
- Show gains of using the C++11/C++14/C++17 standards
- Prevent misusing the modern C++ features

Motivation

- Adoption of modern C++ in ADVA
- Show gains of using the C++11/C++14/C++17 standards
- Prevent misusing the modern C++ features
- Write idiomatic code which is easy to understand

Outline

Memory Management

Motivation

- Correct and efficient usage of "smart" pointers in various scopes
  - Local objects
  - Return values
  - Function parameters

Memory Management

Motivation

- Correct and efficient usage of ”smart” pointers in various scopes
  - Local objects
  - Return values
  - Function parameters
- Correct ownership transfer

Memory Management
Motivation

- Correct and efficient usage of "smart" pointers in various scopes
  - Local objects
  - Return values
  - Function parameters
- Correct ownership transfer
- Safe usage of raw pointers

Local variable

Heap allocation

```
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   int ret = myObj->doSomething();
   if (ret < 0 )
   {

      return;
   }
   myObj->doSomethingElse();
   delete myObj;
}
```

Local variable

Heap allocation

```
void func(int i)
{
    MyClass* myObj = new MyClass(i);
    int ret = myObj->doSomething();
    if (ret < 0 )
    {

        return;
    }
    myObj->doSomethingElse();
    delete myObj;
}
```

Pretty obvious bug

Early return from function forgetting to free the memory. One of the most common sources of memory leaks.

Local variable

Heap allocation

```cpp
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   int ret = myObj->doSomething();
   if (ret < 0 )
   {
      delete myObj;
      return;
   }
   myObj->doSomethingElse();
   delete myObj;
}
```

Pretty obvious bug

Early return from function forgetting to free the memory. One of the most common sources of memory leaks.

Local variable

Heap allocation

```
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   int ret = myObj->doSomething();
   if (ret < 0 )
   {
      delete myObj;
      return;
   }
   myObj->doSomethingElse();
   delete myObj;
}
```

Pretty obvious bug

Early return from function forgetting to free the memory. One of the most common sources of memory leaks.

Not so obvious bug

What if doSomething or doSomethingElse throws an exception?

Local variable

Heap allocation

```
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   int ret = myObj->doSomething();
   if (ret < 0 )
   {
      delete myObj;
      return;
   }
   myObj->doSomethingElse();
   delete myObj;
}
```

Pretty obvious bug

Early return from function forgetting to free the memory. One of the most common sources of memory leaks.

Not so obvious bug

What if doSomething or doSomethingElse throws an exception?

Exceptions may cause leaks too

If an exception is thrown inside this function the memory allocated prior to that exception will leak.

Local variable

Heap allocation

```
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   try
   {
      int ret = myObj->doSomething();
      if (ret < 0 )
      {
         delete myObj;
         return;
      }
      myObj->doSomethingElse();
   }
   catch (...)
   {
      delete myObj;
      throw;
   }
   delete myObj;
}
```

Pretty obvious bug

Early return from function forgetting to free the memory. One of the most common sources of memory leaks.

Not so obvious bug

What if doSomething or doSomethingElse throws an exception?

Exceptions may cause leaks too

If an exception is thrown inside this function the memory allocated prior to that exception will leak.

Exception safety

The function must catch all the exceptions to make sure it frees the memory.

Local variable

Heap allocation

```
void func(int i)
{
   MyClass* myObj = new MyClass(i);
   try
   {
      int ret = myObj->doSomething();
      if (ret < 0 )
      {
         delete myObj;
         return;
      }
      myObj->doSomethingElse();
   }
   catch (...)
   {
      delete myObj;
      throw;
   }
   delete myObj;
}
```

...but in fact we don't need to allocate myObj on the heap

Local variable

Stack allocation

```
void func(int i)
{
    MyClass myObj;
    auto ret = myObj.doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj.doSomethingElse();
}
```

- Stack variable insures automatic cleanup
- Stack allocation is much faster than heap allocation
- The code is more concise

Stack should be the default choice for local variable.

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    MyClass* myObj = Factory::createObj(i);
    try
    {
        int ret = myObj->doSomething();
        if (ret < 0 )
        {
            delete myObj;
            return;
        }
        myObj->doSomethingElse();
    }
    catch (...)
    {
        delete myObj;
        throw;
    }
    delete myObj;
}
```

- An object is returned by a function which allocates it on the heap

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    MyClass* myObj = Factory::createObj(i);
    try
    {
        int ret = myObj->doSomething();
        if (ret < 0 )
        {
            delete myObj;
            return;
        }
        myObj->doSomethingElse();
    }
    catch (...)
    {
        delete myObj;
        throw;
    }
    delete myObj;
}
```

- An object is returned by a function which allocates it on the heap
- All the cleanup code is needed to make sure the function is exception safe and won't leak

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    auto myObj = std::unique_ptr<MyClass>(
        Factory::createObj(i)});
    auto ret = myObj->doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj->doSomethingElse();
}
```

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    auto myObj = std::unique_ptr<MyClass>(
        Factory::createObj(i)});
    auto ret = myObj->doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj->doSomethingElse();
}
```

- Heap allocated memory is encapsulated inside the `unique_ptr`

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    auto myObj = std::unique_ptr<MyClass>(
        Factory::createObj(i)});
    auto ret = myObj->doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj->doSomethingElse();
}
```

- Heap allocated memory is encapsulated inside the `unique_ptr`
- Remember: Stack should be the default choice for local variable.

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    auto myObj = std::unique_ptr<MyClass>(
        Factory::createObj(i)});
    auto ret = myObj->doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj->doSomethingElse();
}
```

- Heap allocated memory is encapsulated inside the `unique_ptr`
- Remember: Stack should be the default choice for local variable.
- `unique_ptr` is indeed a local variable on the stack

Local variable

Object is allocated by a factory function

```
void func(int i)
{
    auto myObj = std::unique_ptr<MyClass>(
        Factory::createObj(i)});
    auto ret = myObj->doSomething();
    if (ret < 0 )
    {
        return;
    }
    myObj->doSomethingElse();
}
```

- Heap allocated memory is encapsulated inside the `unique_ptr`
- Remember: Stack should be the default choice for local variable.
- `unique_ptr` is indeed a local variable on the stack
- The heap cleanup happens when the `unique_ptr` goes out of scope

Heap allocation

```
void func(int i)
{
   int* bigArray = new int[i];
   // Do stuff with the array
   delete[] myObj;
}
```

Heap allocation

```
void func(int i)
{
    int* bigArray = new int[i];
    // Do stuff with the array
    delete[] myObj;
}
```

Must remember to use `delete[]` instead of `delete`

Heap allocation

```
void func(int i)
{
   auto myObj = std::make_unique<int[]>(i);
   // Do stuff with the array
   //...
}
```

- Use of `make_unique` convenience function - no explicit `new`
- `unique_ptr` takes care of de-allocating the array - no explicit `delete`

Local object

Stack should be the first choice for local variable

Example

```
MyClass p1(100);
```

Local object

When heap is in the only option use `unique_ptr`

Example

```
std::unique_ptr<MyClass> p1(new MyClass(100));
```

Local object

When heap is in the only option use `unique_ptr`

Example

```
std::unique_ptr<MyClass> p1(new MyClass(100));
```

Prefer using `make_unique` over `unique_ptr` constructor directly

Example

```
auto p2 = std::make_unique<MyClass>(100);
```

Local object

When heap is in the only option use `unique_ptr`

Example

```
std::unique_ptr<MyClass> p1(new MyClass(100));
```

Prefer using `make_unique` over `unique_ptr` constructor directly

Example

```
auto p2 = std::make_unique<MyClass>(100);
```

Avoid using `new` outside the `unique_ptr` constructor

Example

```
auto p1 = new MyClass(100);
// Do sthg
std::unique_ptr<MyClass> pu(p1);
```

Local object

Don't use raw pointer to represent ownership!

Example

```
auto p1 = new MyClass(100);
// Do sthg
delete p2;
```

Local object

Don't use raw pointer to represent ownership!

Example

```
auto p1 = new MyClass(100);
// Do sthg
delete p2;
```

Don't use `delete`! unless in very specific use cases.

Factory function

```
MyClass*
makeObj(const std::string& name)
{
   if (name == "A")
      return new MyDerivedClassA();
   else if(name == "B")
      return new MyDerivedClassB();
   return 0;
}
```

Factory function

```
MyClass*
makeObj(const std::string& name)
{
   if (name == "A")
      return new MyDerivedClassA();
   else if(name == "B")
      return new MyDerivedClassB();
   return 0;
}
```

- Author of `makeObj` API has to document that the raw pointer returned by the function allocates memory which has to be freed by the caller

Factory function

```cpp
MyClass*
makeObj(const std::string& name)
{
   if (name == "A")
      return new MyDerivedClassA();
   else if(name == "B")
      return new MyDerivedClassB();
   return 0;
}
```

```cpp
MyClass* obj = makeObj("A");
// Do sthg with obj
delete obj;
```

- Author of `makeObj` API has to document that the raw pointer returned by the function allocates memory which has to be freed by the caller
- User of the API must remember to free this memory

Factory function

```
MyClass*
makeObj(const std::string& name)
{
   if (name == "A")
      return new MyDerivedClassA();
   else if(name == "B")
      return new MyDerivedClassB();
   return 0;
}
```

```
std::unique_ptr<MyClass> obj(
         makeObj("A"));
```

- Author of `makeObj` API has to document that the raw pointer returned by the function allocates memory which has to be freed by the caller
- User of the API must remember to free this memory
- User of the API may assign the result to a `unique_ptr`

Factory function

```
std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
   if (name == "A")
      return std::make_unique<MyDerivedClassA>();
   else if (name == "B")
      return std::make_unique<MyDerivedClassB>();
   return nullptr;
}
```

Factory function

```
std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
   if (name == "A")
      return std::make_unique<MyDerivedClassA>();
   else if (name == "B")
      return std::make_unique<MyDerivedClassB>();
   return nullptr;
}
```

- Returning `unique_ptr` explicitly reflects the intention of transferring the ownership to the caller

Factory function

```cpp
std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
    if (name == "A")
        return std::make_unique<MyDerivedClassA>();
    else if (name == "B")
        return std::make_unique<MyDerivedClassB>();
    return nullptr;
}
```

```cpp
auto obj = makeObj("A");
```

- Returning `unique_ptr` explicitly reflects the intention of transferring the ownership to the caller
- The caller automatically and seamlessly becomes the owner
- What the caller gets is the `unique_ptr` so they don't need to worry about the de-allocation

Factory function

```cpp
std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
    if (name == "A")
        return std::make_unique<MyDerivedClassA>();
    else if (name == "B")
        return std::make_unique<MyDerivedClassB>();
    return nullptr;
}
```

```cpp
std::shared_ptr<MyClass> shObj = makeObj("B");
```

- Returning `unique_ptr` explicitly reflects the intention of transferring the ownership to the caller
- The caller automatically and seamlessly becomes the owner
- What the caller gets is the `unique_ptr` so they don't need to worry about the de-allocation
- `unique_ptr` can be assigned to `shared_ptr`

Factory function

Don't return a raw pointer when transferring ownership from the callee to the caller

Example

```
MyClass*
makeObj()
{
   return new MyDerivedClassA();
}
```

Factory function

Prefer returning `unique_ptr` to transfer ownership from the callee to the caller

Example

```
std::unique_ptr<MyClass>
makeObj()
{
    return std::make_unique<MyDerivedClassA>();
}
```

Factory function

Prefer returning `unique_ptr` to transfer ownership from the callee to the caller

Example

```
std::unique_ptr<MyClass>
makeObj()
{
    return std::make_unique<MyDerivedClassA>();
}
```

Returning `shared_ptr` makes sense only if all the callers want to consume `shared_ptr`.

Example

```
std::shared_ptr<MyClass>
makeObj()
{
    return std::make_shared<MyClass>();
}
```

Factory function

Prefer returning non-polymorphic types by value

Example

```
MyClass
makeObj()
{
   MyClass obj;
   // do something with obj
   return obj;
}
auto obj = makeObj();
```

Factory function

Prefer returning non-polymorphic types by value

Example

```
MyClass
makeObj()
{
    MyClass obj;
    // do something with obj
    return obj;
}
auto obj = makeObj();
```

Don't try to move it.

Example

```
MyClass
makeObj()
{
    MyClass obj;
    return std::move(obj);
}
```

Return by value

```cpp
struct A
{
    std::string s;
};

A makeA()
{
    A a; //default ctor
    a.s = "Some looooooong string";
    return a;
}
```

```cpp
auto a = makeA(); // move ctor
```

Since C++11 return by value is implemented employing move constructor - the return value from func is moved to the value it is assigned to.

Return by value

```
struct A
{
    std::string s;
};

A makeA()
{
    A a; //default ctor
    a.s = "Some looooooong string";
    return a;
}
```

```
auto a = makeA(); // move ctor optimized away
```

However it many cases the compiler will apply Return Value Optimization and elide the move constructor.

Return by value

```
struct A
{
    std::string s;
};

A makeA()
{
    A a; //default ctor
    a.s = "Some looooooong string";
    return std::move(a);
}
```

```
auto a = makeA(); // move ctor
```

Don't `move`

Explicit moving will inhibit the RVO and force compiler to move.

Return by value

```
struct A
{
    std::string s;
};

struct B : A
{};


A makeA()
{
    B b; //default ctor
    b.s = "Some looooooong string";
    return b;
}


auto a = makeA(); //copy ctor
```

Return by value

```
struct A
{
    std::string s;
};

struct B : A
{};


A makeA()
{
    B b; //default ctor
    b.s = "Some looooooong string";
    return b;
}


auto a = makeA(); //copy ctor
```

Slicing

Implicit conversion from B to A requires copy.

Transferring ownership to a function

```
void func(MyClass* obj)
{
    obj->doSomethingElse();
    delete obj;
}
```

```
MyClass* obj = new MyClass();
obj->doSomething();
func(obj);
```

- Author of the `func` API has to document that this function frees the memory
- The pointer after calling `func` is invalid

Transferring ownership to a function

```cpp
void func(std::unique_ptr<MyClass> obj)
{
    obj->doSomethingElse();
}
```

```cpp
auto obj = std::make_unique<MyClass>();
obj->doSomething();
func(std::move(obj));
```

- The `unique_ptr` parameter enforces that this function takes over the ownership of `obj`
- It gets clear from the code the the ownership transfer is happening
- The obj is automatically cleaned up upon `func` return

Transferring ownership to a function

```
void func(MyClass* obj)
{
    obj->doSomethingElse();
    delete obj;
}

MyClass* obj = new MyClass();
obj->doSomething();
func(obj);
```

```
void func(std::unique_ptr<MyClass> obj)
{
    obj->doSomethingElse();
}

auto obj = std::make_unique<MyClass>();
obj->doSomething();
func(std::move(obj));
```

Transferring ownership to a function

```cpp
void func(MyClass* obj)
{
    obj->doSomethingElse();
    delete obj;
}

MyClass* obj = new MyClass();
obj->doSomething();
func(obj);
delete obj; // Oooohps....
```

Invalid pointer

obj points to deallocated memory

```cpp
void func(std::unique_ptr<MyClass> obj)
{
    obj->doSomethingElse();
}

auto obj = std::make_unique<MyClass>();
obj->doSomething();
func(std::move(obj));
```

Transferring ownership to a function

```cpp
void func(MyClass* obj)
{
    obj->doSomethingElse();
    delete obj;
}

MyClass* obj = new MyClass();
obj->doSomething();
func(obj);
delete obj; // Oooohps....
```

```cpp
void func(std::unique_ptr<MyClass> obj)
{
    obj->doSomethingElse();
}

auto obj = std::make_unique<MyClass>();
obj->doSomething();
func(std::move(obj));
```

Invalid pointer

obj points to deallocated memory

Valid pointer

obj has `nullptr` value

Transferring ownership to a function

Use `unique_ptr` as function parameter when transferring ownership to that function

Example

```
void func(std::unique_ptr<MyClass> obj);
```

Transferring ownership to a function

Use `unique_ptr` as function parameter when transferring ownership to that function

Example

```
void func(std::unique_ptr<MyClass> obj);
```

Don't use raw pointer!

Example

```
void func(MyClass* obj);
```

Class member

Use direct variable for non-polymorphic members

Example

```
class Foo
{
   Bar mBar;
};
```

Use `unique_ptr` for polymorphic types

Example

```
class Foo
{
   std::unique_ptr<Bar> mBar;
};
```

Class member

Use `shared_ptr` **ONLY** if the variable is **shared** with someone else.

Example

```
class Foo
{
   std::shared_ptr<Bar> mBar;
public:
   Foo(std::shared_ptr<Bar> val) : mBar(val) {}
};
```

Class member

Prefer using `unique_ptr` from raw pointers as class members if the class is the owner of the memory.

Example

```
class Foo
{
   Bar* mBar;
public:
   Foo() : mBar(new Bar) {}
   ~Foo() { delete mBar;}
};
```

Example

```
class Foo
{
   std::unique_ptr<Bar> mBar;
public:
   Foo() : mBar(new Bar) {}
   ~Foo() = default;
};
```

Class member

Use `unique_ptr` to transfer ownership to a class

Example

```
class Foo
{
   Bar* mBar;
public:
   Foo(Bar* bar) : mBar(bar) {}
   ~Foo() { delete mBar;}
};
```

Example

```
class Foo
{
   std::unique_ptr<Bar> mBar;
public:
   Foo(std::unique_ptr<Bar> bar) : mBar(std::move(bar)) {}
   ~Foo() = default;
};
```

Class member

Pimpl idiom

```cpp
//Foo.hpp
class Foo
{
   struct Impl;
   Impl* mImpl;
public:
   Foo();
   ~Foo();
   void doSomething();
};
```

```cpp
//Foo.cpp
struct Foo::Impl
{
   void doSomething() {/*...*/}
};

Foo::Foo() : mImpl(new Impl()){}
Foo::~Foo() {delete mImpl;}
void Foo::doSomething()
{
   mImpl->doSomething();
}
```

Pimpl

A common idiom to break include dependencies and hide implementation details.

Class member

Pimpl idiom

```cpp
//Foo.hpp
class Foo
{
    struct Impl;
    const std::unique_ptr<Impl> mImpl;
public:
    Foo();
    ~Foo();
    void doSomething();
};
```

```cpp
//Foo.cpp
struct Foo::Impl
{
    void doSomething() {/*...*/}
};

Foo::Foo() : mImpl(std::make_unique<Impl>()){}
Foo::~Foo() = default;
void Foo::doSomething()
{
    mImpl->doSomething();
}
```

Pimpl

A common idiom to break include dependencies and hide implementation details.

`const unique_ptr` to represent Pimpl

Class member

Forward declaration

```cpp
//Bar.hpp
class Bar {/*...*/};
```

```cpp
//Foo.hpp
class Bar;
class Foo
{
   Bar* mBar;
public:
   Foo();
   ~Foo() = default;
   void doSomething();
};
```

```cpp
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo() : mBar(new Bar) {}
void Foo::doSomething)(){/*...*/}
```

Forward declaration

If a member is a pointer or a reference there's a good practice to forward declare the class instead of including its whole definition. The definition is included into the cpp file.

Class member

Forward declaration

```cpp
//Bar.hpp
class Bar {/*...*/};
```

```cpp
//Foo.hpp
class Bar;
class Foo
{
   Bar* mBar;
public:
   Foo();
   ~Foo() = default;
   void doSomething();
};
```

```cpp
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo() : mBar(new Bar) {}
void Foo::doSomething)(){/*...*/}
```

Forward declaration

If a member is a pointer or a reference there's a good practice to forward declare the class instead of including its whole definition. The definition is included into the cpp file.

Incomplete type

Type which is declared but not defined.

Class member

Forward declaration with `unique_ptr`

```
//Bar.hpp
class Bar {/*...*/};
```

```
//Foo.hpp
class Bar;
class Foo
{
    std::unique_ptr<Bar> mBar;
public:
    Foo();
    ~Foo() = default;
    void doSomething();
};
```

```
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo() : mBar(std::make_unique<Bar>()){}
void Foo::doSomething)(){/*...*/}
```

Class member

Forward declaration with unique_ptr

```cpp
//Bar.hpp
class Bar {/*...*/};
```

```cpp
//Foo.hpp
class Bar;
class Foo
{
   std::unique_ptr<Bar> mBar;
public:
   Foo();
   ~Foo() = default;
   void doSomething();
};
```

```cpp
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo() : mBar(std::make_unique<Bar>()){}
void Foo::doSomething)(){/*...*/}
```

Compiler error

`Bar` is an `Incomplete type`. The type T held in `unique_ptr` must be complete where the destructor is defined.

Class member

Forward declaration with unique_ptr

```
//Bar.hpp
class Bar {/*...*/};
```

```
//Foo.hpp
class Bar;

class Foo
{
   std::unique_ptr<Bar> mBar;
public:
   Foo();
   ~Foo() {}
   void doSomething();
};
```

```
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo() : mBar(std::make_unique<Bar>()){}
void Foo::doSomething)(){/*...*/}
```

Compiler error

Bar is an Incomplete type. The type T held in unique_ptr must be complete where the destructor is defined.

Compiler error

Same issue with empty destructor **defined** in the header file.

Class member

Forward declaration with `unique_ptr`

```
//Bar.hpp
class Bar {/*...*/};
```

```
//Foo.hpp
class Bar;

class Foo
{
   std::unique_ptr<Bar> mBar;
public:
   Foo();
   ~Foo();
   void doSomething();
};
```

```
//Foo.cpp
#include "Foo.hpp"
#include "Bar.hpp"

Foo::Foo()
   : mBar(std::make_unique<Bar>()){}
Foo::~Foo() = default;
void Foo::doSomething)(){/*...*/}
```

Correct class with forward declared `unique_ptr`

The desctrutor should only be **declared** in the header file. The definition must be in the cpp file because this is where the `Bar` type is complete.

Prefer using forward declarations for pointer and reference class members over including their definitions into the header.

Mind that `unique_ptr` destructor definition requires pointer type to be complete.

unique_ptr<T> cannot leak

```
struct S
{
   std::unique_ptr<MyClass> mObj;
};

std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
   if (name == "A")
      return std::make_unique<MyDerivedClassA >();
   else if (name == "B")
      return std::make_unique<MyDerivedClassB >();
   return nullptr;
}

void func(std::unique_ptr<MyClass> obj)
{
   obj->doSomething();
   S s{std::move(obj}};
   s->mObj->doSomething()
}
```

```
auto obj = makeObj("A");
obj->doSomething();
func(obj);
```

- Object allocated and transferred to a local variable
- Ownership of object transferred to a function
- Ownership transferred to a class

unique_ptr<T> cannot leak

```
struct S
{
   std::unique_ptr<MyClass> mObj;
};

std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
   if (name == "A")
      return std::make_unique<MyDerivedClassA>();
   else if (name == "B")
      return std::make_unique<MyDerivedClassB>();
   return nullptr;
}

void func(std::unique_ptr<MyClass> obj)
{
   obj->doSomething();
   S s{std::move(obj)};
   s->mObj->doSomething()
}
```

```
auto obj = makeObj("A");
obj->doSomething();
func(obj);
```

- Object allocated and transferred to a local variable
- Ownership of object transferred to a function
- Ownership transferred to a class

unique_ptr<T> cannot leak

```
struct S
{
   std::unique_ptr<MyClass> mObj;
};

std::unique_ptr<MyClass>
makeObj(const std::string& name)
{
   if (name == "A")
      return std::make_unique<MyDerivedClassA>();
   else if (name == "B")
      return std::make_unique<MyDerivedClassB>();
   return nullptr;
}

void func(std::unique_ptr<MyClass> obj)
{
   obj->doSomething();
   S s{std::move(obj)};
   s->mObj->doSomething()
}
```

```
auto obj = makeObj("A");
obj->doSomething();
func(obj);
```

- Object allocated and transferred to a local variable
- Ownership of object transferred to a function
- Ownership transferred to a class

unique_ptr<T> cannot leak

- There is no explicit memory allocation/deallocation
- The ownership is seamlessly transferred from one function to another
- Only automatic variable created and passed - no leaks by default

Avoid using `new` and `delete` explicitly!

unique_ptr<T> simplified implementation

```
template <typename T>
class unique_ptr
{
    T* _ptr = nullptr;
public:
    unique_ptr() = default;
    unique_ptr(T* p) : _ptr{p} {}
    ~unique_ptr() { delete _ptr; }
    unique_ptr(unique_ptr&& u)
       {
           _ptr = u.release();
       }
    unique_ptr& operator=(unique_ptr&& u)
    {
        reset(u.release()); return *this;
    }

   T* get() const { return _ptr; }
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    void reset(T* p) {delete _ptr; _ptr = p;}
    T* release()
   {
      auto p = get(); _ptr = nullptr;
      return p;
   }
private:
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

unique_ptr<T> simplified implementation

```
template <typename T>
class unique_ptr
{
    T* _ptr = nullptr;
public:
    unique_ptr() = default;
    unique_ptr(T* p) : _ptr{p} {}
    ~unique_ptr() { delete _ptr; }
    unique_ptr(unique_ptr&& u)
       {
           _ptr = u.release();
       }
    unique_ptr& operator=(unique_ptr&& u)
    {
        reset(u.release()); return *this;
    }

   T* get() const { return _ptr; }
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    void reset(T* p) {delete _ptr; _ptr = p;}
    T* release()
   {
      auto p = get(); _ptr = nullptr;
      return p;
   }
private:
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

Resource Acquisition Is Initialization (RAII)

The lifetime of the allocated memory is bound to the constructor and destructor

unique_ptr<T> simplified implementation

```
template <typename T>
class unique_ptr
{
   T* _ptr = nullptr;
public:
    unique_ptr() = default;
    unique_ptr(T* p) : _ptr{p} {}
    ~unique_ptr() { delete _ptr; }
    unique_ptr(unique_ptr&& u)
       {
           _ptr = u.release();
       }
    unique_ptr& operator=(unique_ptr&& u)
    {
        reset(u.release()); return *this;
    }

   T* get() const { return _ptr; }
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    void reset(T* p) {delete _ptr; _ptr = p;}
    T* release()
   {
      auto p = get(); _ptr = nullptr;
      return p;
   }
private:
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

Resource Acquisition Is Initialization (RAII)

The lifetime of the allocated memory is bound to the constructor and destructor

Move construction & assignment

Allows transferring ownership from one `unique_ptr` to another

unique_ptr<T> simplified implementation

```
template <typename T>
class unique_ptr
{
    T* _ptr = nullptr;
public:
    unique_ptr() = default;
    unique_ptr(T* p) : _ptr{p} {}
    ~unique_ptr() { delete _ptr; }
    unique_ptr(unique_ptr&& u)
       {
           _ptr = u.release();
       }
    unique_ptr& operator=(unique_ptr&& u)
    {
        reset(u.release()); return *this;
    }

   T* get() const { return _ptr; }
    T* operator->() const {return get();}
    T& operator*() const {return *get();}
    void reset(T* p) {delete _ptr; _ptr = p;}
    T* release()
   {
      auto p = get(); _ptr = nullptr;
      return p;
   }
private:
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

Resource Acquisition Is Initialization (RAII)

The lifetime of the allocated memory is bound to the constructor and destructor

Move construction & assignment

Allows transferring ownership from one `unique_ptr` to another

Inhibit copy construction & assignment

Insures uniqueness - one and only one `unique_ptr` can own the allocate memory

unique_ptr<T> overhead?

```cpp
struct A {int i = 5;};

int main()
{
    auto* a = new A();
    delete a;
}
```

```
sub     rsp, 8
mov     edi, 4
call    operator new(unsigned long)
mov     esi, 4
mov     DWORD PTR [rax], 5
mov     rdi, rax
call
operator delete(void*, unsigned long)
xor     eax, eax
add     rsp, 8
ret
```

unique_ptr<T> overhead?

```cpp
struct A {int i = 5;};

int main()
{
   auto* a = new A();
   delete a;
}
```

```asm
sub     rsp, 8
mov     edi, 4
call    operator new(unsigned long)
mov     esi, 4
mov     DWORD PTR [rax], 5
mov     rdi, rax
call
operator delete(void*, unsigned long)
xor     eax, eax
add     rsp, 8
ret
```

```cpp
struct A {int i = 5;};

int main()
{
   auto a = std::make_unique<A>();

}
```

```asm
sub     rsp, 8
mov     edi, 4
call    operator new(unsigned long)
mov     esi, 4
mov     DWORD PTR [rax], 5
mov     rdi, rax
call
operator delete(void*, unsigned long)
xor     eax, eax
add     rsp, 8
ret
```

unique_ptr<T> overhead?

```cpp
struct A {int i = 5;};

int main()
{
    auto* a = new A();
    delete a;
}
```

```cpp
struct A {int i = 5;};

int main()
{
    auto a = std::make_unique<A>();

}
```

```
sub     rsp, 8
mov     edi, 4
call    operator new(unsigned long)
mov     esi, 4
mov     DWORD PTR [rax], 5
mov     rdi, rax
call
operator delete(void*, unsigned long)
xor     eax, eax
add     rsp, 8
ret
```

```
sub     rsp, 8
mov     edi, 4
call    operator new(unsigned long)
mov     esi, 4
mov     DWORD PTR [rax], 5
mov     rdi, rax
call
operator delete(void*, unsigned long)
xor     eax, eax
add     rsp, 8
ret
```

Zero-overhead abstraction.

A few words on `shared_ptr<T>`

It is one of the most overused facilities in the STL library.

- `shared_ptr<T>` is disparately more complex feature than `unique_ptr<T>`
- It is heavy and expensive
  - atomic reference counting & thread safety
  - virtual functions calls
- May impede performance

If you are tempted to use `shared_ptr` think twice (or thrice) if you really need it and be aware of the cost being paid.
`unique_ptr` is always the default choice for smart pointer. `shared_ptr` is the last resort.

Can smart pointers leak?

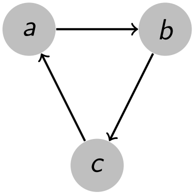
```
struct Node
{
    std::shared_ptr<Node> mNext;
};
```

```
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```

Can smart pointers leak?

```
struct Node
{
    std::shared_ptr<Node> mNext;
};
```

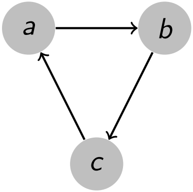
```
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```



Can smart pointers leak?

```
struct Node
{
    std::shared_ptr<Node> mNext;
};
```

```
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```

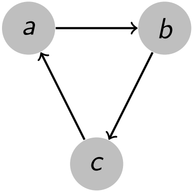


```
==17575== 120 (40 direct, 80 indirect) bytes in 1 blocks are definitely lost
in loss record 3 of 3
```

Can smart pointers leak?

```
struct Node
{
    std::shared_ptr<Node> mNext;
};
```

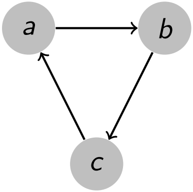
```
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```



Can smart pointers leak?

```cpp
struct Node
{
    std::shared_ptr<Node> mNext;
};
```

```cpp
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```



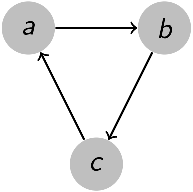
Ownership cycle

None of the 3 nodes ever gets deleted.

Can smart pointers leak?

```
struct Node
{
    std::weak_ptr<Node> mNext;
};
```

```
{
 auto a = std::make_shared<Node>();
 auto b = std::make_shared<Node>();
 auto c = std::make_shared<Node>();
 a->mNext = b;
 b->mNext = c;
 c->mNext = a;
}
```



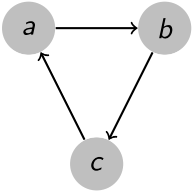
`weak_ptr` edges

`weak_ptr` holds a shared pointer but does not hold shared ownership. Allows to break the ownership dependency and avoid leaks.

Can smart pointers leak?

```
struct Node
{
    Node* mNext = nullptr;
};
```

```
{
    auto a = std::make_unique<Node>();
    auto b = std::make_unique<Node>();
    auto c = std::make_unique<Node>();
    a->mNext = b.get();
    b->mNext = c.get();
    c->mNext = a.get();
}
```

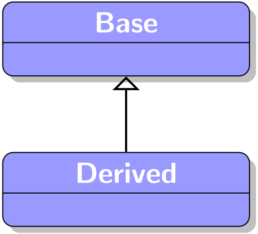


Raw pointer edges

No leaks are guaranteed by default by using only `unique_ptr`

Polymorphic types

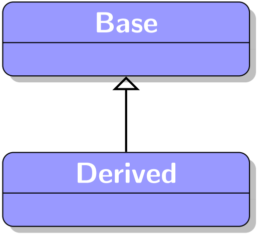
unique_ptr<T>



Polymorphic types

`unique_ptr<T>`

- `Base` and `Derived` are related types.

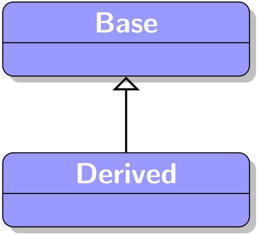


Polymorphic types

unique_ptr<T>

- `Base` and `Derived` are related types.
- `Derived*` is convertible to `Base*`.

```
Base* b = new Derived(); // OK
```



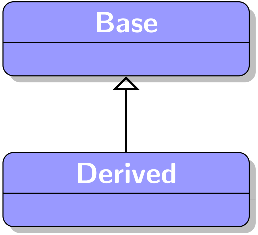
Polymorphic types

unique_ptr<T>

- Base and Derived are related types.
- Derived* is convertible to Base*.

```
Base* b = new Derived(); // OK
```

- unique_ptr<Base> and unique_ptr<Derived> are **NOT** related types



Polymorphic types

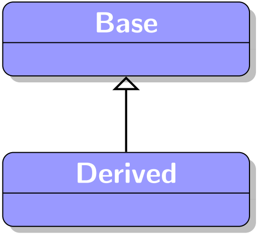
unique_ptr<T>

- Base and Derived are related types.
- Derived* is convertible to Base*.

```
Base* b = new Derived(); // OK
```

- unique_ptr<Base> and unique_ptr<Derived> are **NOT** related types
- Is unique_ptr<Derived> convertible to unique_ptr<Base>?

```
std::unique_ptr<Base> b =
    std::make_unique<Derived>() // OK???;
```



unique_ptr (less) simplified implementation

Conversion construction & assignment

```
template <typename T>
class unique_ptr
{
    T* _ptr = nullptr;
public:
   /*..*/

   template <typename U>
   unique_ptr(unique_ptr<U>&& u)
   {
      static_assert(is_covertible<U*,T*>::value,
         "U not convertible to T");
      _ptr = u.release();
   }

   template <typename U>
   unique_ptr&
    operator=(unique_ptr<U>&& u)
    {
      static_assert(is_covertible<U*,T*>::value,
         "U not convertible to T");
        reset(u.release());
        return *this;
    }

   /*..*/
};
```

Constructor & assigment operator

Take rvalue reference to a `unique_ptr` which stores a different type

If U* is convertible to T* then unique_ptr<U> is convertible to unique_ptr<T>

Polymorphic types

unique_ptr<T>

```
struct Base
{
    virtual ~Base() = default;
};

struct Derived : Base
{
};
```

```
std::unique_ptr<Base> b = std::make_unique<Derived>();
```

Will work correctly **only** if the `Base` has virtual destructor.

Polymorphic types

unique_ptr<T>

```
struct Base
{
    virtual ~Base() = default;
};

struct Derived : Base
{
};
```

```
std::unique_ptr<Base> b = std::make_unique<Derived>();
```

Will work correctly **only** if the `Base` has virtual destructor.

```
template<class T,
       class Deleter = std::default_delete<T>
> class unique_ptr;
```

The deleter is bound to type T. If destructor is not virtual then only Base class destructor will be called!

Non-virtual destructor case

```cpp
struct Base
{
   ~Base() = default;
};


struct Derived : Base
{
};

template <typename T>
struct MyDeleter
{
    template <typename U>
    void operator()(U* p) const
    {
        delete static_cast<T*>(p);
    }
};
```

```cpp
std::unique_ptr<Base, MyDeleter<Derived>> b = std::make_unique<Derived>();
```

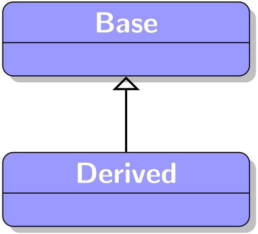
The deleter is still bound to type T but the `operator()` will downcast the Base class to Derived insuring the Derived destructor to be called.

Assign a pointer of the Derived class to a pointer to the Base class only when the Base class has virtual destructor.

More on polymorphic types

unique_ptr<T>

- Base and Derived are related types.
- Derived* is converitble to Base*.



- unique_ptr<Base> and unique_ptr<Derived> are **NOT** related types
- unique_ptr<Derived> is convertible to unique_ptr<Base>

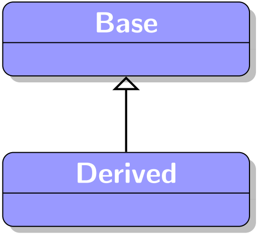
More on polymorphic types

unique_ptr<T>

- Base and Derived are related types.
- Derived* is converitble to Base*.
- Base* can be casted to Derived*

```
Base* b = new Derived(); // OK
auto* d = static_cast<Derived*>(b); // OK
```

- unique_ptr<Base> and unique_ptr<Derived> are **NOT** related types
- unique_ptr<Derived> is convertible to unique_ptr<Base>



More on polymorphic types

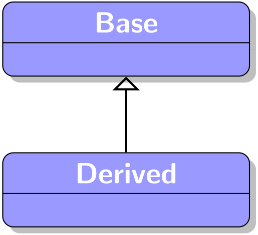
unique_ptr<T>

- Base and Derived are related types.
- Derived* is converitble to Base*.
- Base* can be casted to Derived*

```
Base* b = new Derived(); // OK
auto* d = static_cast<Derived*>(b); // OK
```

- unique_ptr<Base> and unique_ptr<Derived> are **NOT** related types
- unique_ptr<Derived> is convertible to unique_ptr<Base>
- unique_ptr<Base> **CANNOT** be casted to unique_ptr<Derived>

```
std::unique_ptr<Base> b =
   std::make_unique<Derived>() // OK;
auto d = static_cast<
      std::unique_ptr<Derived>>(b); // error
```



A `unique_ptr<U>` can only be moved to `unique_ptr<T>` if U* is convertible to T*.

Example

```
auto d = std::make_unique<Derived>();
std::unique_ptr<Base> b = std::move(d);
auto d_ptr = static_cast<Derived*>(b.get());
```

A `unique_ptr<U>` can only be moved to `unique_ptr<T>` if U* is convertible to T*.

Example

```
auto d = std::make_unique<Derived>();
std::unique_ptr<Base> b = std::move(d);
auto d_ptr = static_cast<Derived*>(b.get());
```

There is no cast semantics for `unique_ptr`!

Example

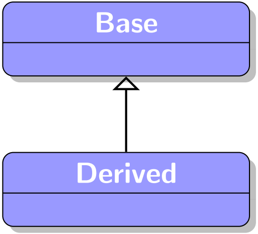
```
std::unique_ptr<Base> b = std::make_unique<Derived>();
auto d = static_cast<Derived>(b); // error
```

(The less casting in the code the better)

Polymorphic types

shared_ptr<T>

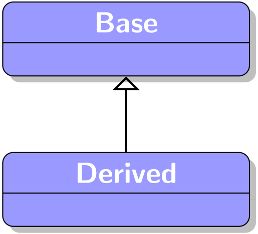
- `shared_ptr<Base>` and `shared_ptr<Derived>` are **NOT** related types



Polymorphic types

shared_ptr<T>

- `shared_ptr<Base>` and `shared_ptr<Derived>` are **NOT** related types
- `shared_ptr<Derived>` is convertible to `shared_ptr<Base>`

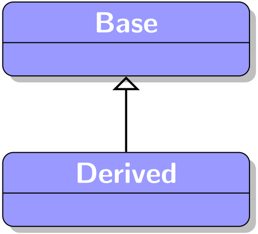


Polymorphic types

shared_ptr<T>

- shared_ptr<Base> and shared_ptr<Derived> are **NOT** related types
- shared_ptr<Derived> is convertible to shared_ptr<Base>
- shared_ptr<Base> **CAN** be casted to shared_ptr<Derived>

```
std::shared_ptr<Base> b =
   std::make_shared<Derived>() // OK;
auto d = std::static_pointer_cast<
      std::shared_ptr<Derived>>(b); // OK
```



A `shared_ptr<U>` defines casting semantics for all 4 cast types:

- `static_pointer_cast`
- `dynamic_pointer_cast`
- `const_pointer_cast`
- `reinterpret_pointer_cast`

Example

```
std::shared_ptr<Base> b = std::make_shared<Derived>()
auto d = std::static_pointer_cast<std::shared_ptr<Derived>>(b);
```

A `shared_ptr<U>` defines casting semantics for all 4 cast types:

- `static_pointer_cast`
- `dynamic_pointer_cast`
- `const_pointer_cast`
- `reinterpret_pointer_cast`

Example

```
std::shared_ptr<Base> b = std::make_shared<Derived>()
auto d = std::static_pointer_cast<std::shared_ptr<Derived>>(b);
```

but ...

... this is expensive

Remember: `shared_ptr<Base>` and `shared_ptr<Derived>` are **unrelated** types

```
template< class T, class U >
std::shared_ptr<T> static_pointer_cast(const std::shared_ptr<U>& r)
{
    auto p = static_cast<T*>(r.get());
    return std::shared_ptr<T>(r, p);
}
```

Remember: `shared_ptr<Base>` and `shared_ptr<Derived>` are **unrelated** types

```
template< class T, class U >
std::shared_ptr<T> static_pointer_cast(const std::shared_ptr<U>& r)
{
    auto p = static_cast<T*>(r.get());
    return std::shared_ptr<T>(r, p);
}
```

Here casting actually means creating another `shared_ptr` instance.

Remember: `shared_ptr<Base>` and `shared_ptr<Derived>` are **unrelated** types

```
template< class T, class U >
std::shared_ptr<T> static_pointer_cast(const std::shared_ptr<U>& r)
{
    auto p = static_cast<T*>(r.get());
    return std::shared_ptr<T>(r, p);
}
```

Here casting actually means creating another `shared_ptr` instance.

Aliasing constructor

An *aliasing constructor* - `shared_ptr` shares ownership with `r` but stores `p`

What about `raw pointers`?

Raw pointers are still OK

Function parameters where the function performs read or write operations on the object but **does not become the owner**

Example

```
void func(MyClass* obj);
void func(const MyClass* obj);
```

Raw pointers are still OK

Return types from a function where the caller **does not become the owner**

Example

```
std::map<int, std::unique_ptr<MyClass>> cache;

MyClass* makeAndSaveObj(int index)
{
   auto obj = std::make_unique<MyClass>();
   auto ptr = obj.get();
   // Do something with obj
   cache.insert({index, std::move(obj)});
   return ptr;
}
```

Objects in containers

```
struct Foo
{
    int i;
    double d;
    char ch[10];
    long long long_int;
};
```

Objects in containers

```
struct Foo
{
    int i;
    double d;
    char ch[10];
    long long long_int;
};
```

Store Foo in a container as value or pointer?

`std::vector<Foo>`

or

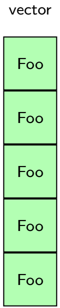
`std::vector<std::unique_ptr<Foo>>`

Objects in containers

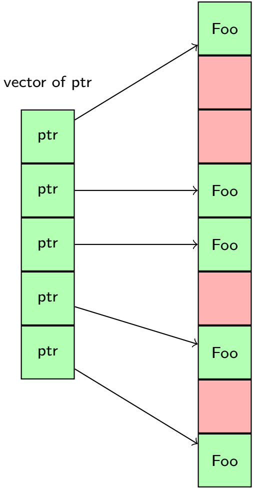
```
struct Foo
{
    int i;
    double d;
    char ch[10];
    long long long_int;
};
```

Operation	`vector<Foo>`	`vector<unique_ptr<Foo>>`
push_back	0.0324s	0.0382s
push_back (*reserve*)	0.0093s	0.0331s
traverse	0.0021s	0.0032s
sort	0.2009s	0.2167s

Table: 1M elements



Traversing data in contiguous memory.



Traversing data in sparse memory.

Objects in containers

```
struct BigFoo
{
    int i;
    double d;
    char ch[1000];
    long long long_int;
};
```

Objects in containers

```
struct BigFoo
{
    int i;
    double d;
    char ch[1000];
    long long long_int;
};
```

Operation	`vector<BigFoo>`	`vector<unique_ptr<BigFoo>>`
push_back	0.7142	0.3237s
push_back (*reserve*)	0.150	0.262s
traverse	0.0088s	0.0095s
sort	1.415s	0.2605s

Table: 1M elements

Store non-polymorphic objects as pointers in STL containers if:

- Their size is big
- Mutating algorithms will be run on the container.

In other case prefer storing object as values.

Moving objects

```
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const BigObject& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(BigObject&& other)
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};
```

Moving objects

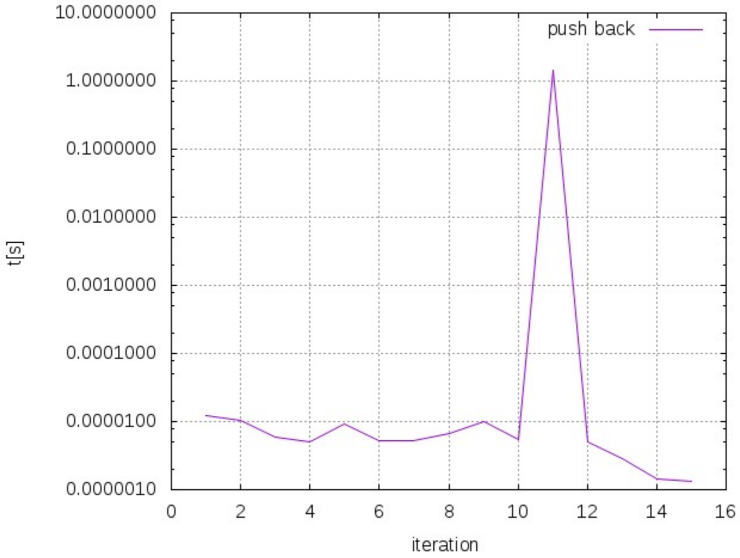
```
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const BigObject& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(BigObject&& other)
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};
```

Moving objects

```
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const BigObject& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(BigObject&& other)
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};
```

Moving objects

```
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const BigObject& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(BigObject&& other)
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};

std::vector<Buffer> vec;
vec.reserve(10);
for (auto i = 1; i <= 15; i++)
{
    vec.push_back(Buffer{100000000});
}
```

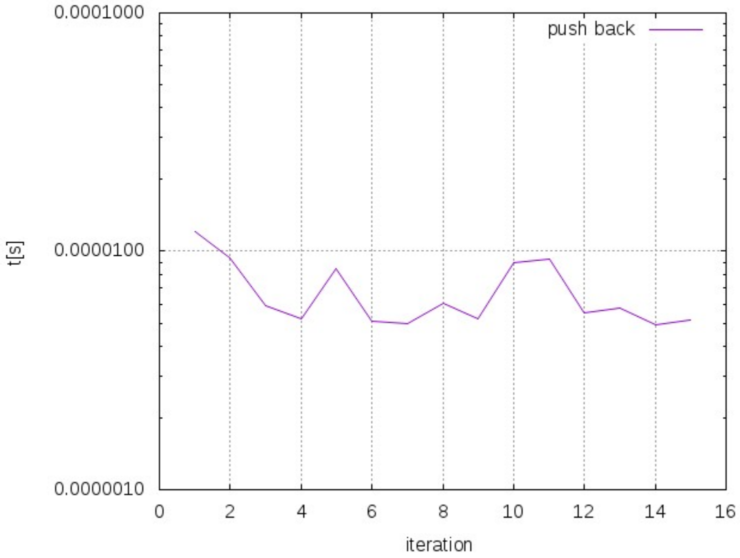

push back
10.0000000
1.0000000
0.1000000
0.0100000
0.0010000
0.0001000
0.0000100
0.0000010
t[s]
0
2
4
6
8
10
12
14
16
iteration

```cpp
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const Buffer& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(Buffer&& other)
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};
```

Strong exception guarantee

During re-allocation `std::vector` uses `move_if_no_except` function to move elements from the old to the new buffer. If the move constructor has no noexcept qualifier it will resort to copy.

```cpp
struct Buffer
{
    Buffer(size_t s) : data{new int[s]}, size{s} {}
    Buffer(const Buffer& other)
    {
        data.reset(new int[other.size]);
        size = other.size;
        std::copy(other.data.get(), other.data.get() + other.size, data.get());
    }
    Buffer(Buffer&& other) noexcept
    {
        data.swap(other.data);
        size = other.size;
    }

    std::unique_ptr<int[]> data;
    size_t size;
};
```



push back
t[s]
0.0001000
0.0000100
0.0000010
0 2 4 6 8 10 12 14 16
iteration

move_if_noexcept

move_if_noexcept

```
template <class T>
typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
                            std::is_copy_constructible<T>::value,
                        const T&,
                        T&&>::type
move_if_noexcept(T& x) noexcept
{
    using type =
        typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
                                    std::is_copy_constructible<T>::value,
                                const T&, T &&>::type;

    return static_cast<type>(x);
}
```

move_if_noexcept

move_if_noexcept

```
template <class T>
typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
                          std::is_copy_constructible<T>::value,
                      const T&,
                      T&&>::type
move_if_noexcept(T& x) noexcept
{
    using type =
        typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
                                  std::is_copy_constructible<T>::value,
                              const T&, T&&>::type;

    return static_cast<type>(x);
}
```

conditional

```
template<bool B, typename T, typename U>
struct conditional { using type  = T; };

template<typename T, typename U>
struct conditional<false, T, U> { using type = U; };
```

move_if_noexcept

move_if_noexcept

```
template <class T>
typename std::conditional<!std::is_nothrow_move_constructible<T>::value &&
                          std::is_copy_constructible<T>::value,
                          const T&,
                          T&&>::type
move_if_noexcept(T& x) noexcept;
```

```
struct A
{
    A() = default;
    A(const A&) {/**/}
    A(A&&) {/**/}
};
struct B
{
    B() = default;
    B(const B&) {/**/}
    B(B&&) noexcept {/**/}
};
```

```
A a;
B b;
auto a2 = std::move_if_noexcept(a); // copy
auto b2 = std::move_if_noexcept(b); // move
```

Prefer storing non-polymorphic objects as values in STL containers.

If `move` operation is expensive reserve the memory first (`vector`).

Prefer storing non-polymorphic objects as values in STL containers.

If `move` operation is expensive reserve the memory first (`vector`).

Make sure the move constructor is `noexcept`.

Resource Acquisition Is Initialization (RAII)

Resource Acquisition Is Initialization (RAII)

One the the most fundamental techniques in C++.

Facilitates automatic resource cleanup to prevent leaks.

Resource Acquisition Is Initialization (RAII)

- memory
- files
- directories
- file descriptors
- mutexes,locks

Resource Acquisition Is Initialization (RAII)

Resource Acquisition Is Initialization (RAII)

Memory

```
{
    std::unique_ptr<Foo> f{new Foo{}};
}
// f is deleted, memory is free
```

Resource Acquisition Is Initialization (RAII)

Memory

```
{
    std::unique_ptr<Foo> f{new Foo{}};
}
// f is deleted, memory is free
```

File handle

```
{
    std::ofstream f("file.txt");
}
// file handle resource is encapsulated in the ofstream class
```

Resource Acquisition Is Initialization (RAII)

Memory

```
{
    std::unique_ptr<Foo> f{new Foo{}};
}
// f is deleted, memory is free
```

File handle

```
{
    std::ofstream f("file.txt");
}
// file handle resource is encapsulated in the ofstream class
```

Mutex

```
std::mutex gmutex;
{
    std::lock_guard<std::mutex> lock{gmutex};
}
// mutex is unlocked
```

Resource Acquisition Is Initialization (RAII)

Directory handle

```
{
    std::filesystem::path p{"dirname"};
    for (auto& x : std::filesystem::directory_iterator(p))
    {
        // ...
    }
    // file handle resource is encapsulated in the directory_entry class
}
```

Resource Acquisition Is Initialization (RAII)

Directory handle

```
{
    std::filesystem::path p{"dirname"};
    for (auto& x : std::filesystem::directory_iterator(p))
    {
        // ...
    }
    // file handle resource is encapsulated in the directory_entry class
}
```

New STL feature

Standard `filesystem` library is only available since C++17.

Resource Acquisition Is Initialization (RAII)

Directory handle

```
{
    boost::filesystem::path p{"dirname"};
    for (auto& x : boost::filesystem::directory_iterator(p))
    {
        // ...
    }
    // file handle resource is encapsulated in the directory_entry class
}
```

Also available in boost

When using compiler not supporting C++17, the `filesystem` library is available in boost.

Resource Acquisition Is Initialization (RAII)

Directory handle

```
{
    DIR* dir = opendir("dirname");
    if (dir)
    {
        struct dirent* dent;
        while(dent=readdir(dir))
        {
            // ...
        }
    }
    close(dir);
}
```

C API

Prefer using the `std` or `boost` `filesystem` library over the C API.

Resource Acquisition Is Initialization (RAII)

If a standard library (or boost) does not provide RAII wrapper for a given resource, write your own.

Resource Acquisition Is Initialization (RAII)

File descriptor

```
{
    int fd = socket(/*...*/)
    if (fd != -1)
    {
        //...
    }

    close(fd);
}
{
    int fd = open("/dev/random");
    if (fd != -1)
    {
        //...
    }

    close(fd);
}
```

Resource Acquisition Is Initialization (RAII)

File descriptor

```cpp
class FileDescriptor
{
    int mFd = -1;
public:
    FileDescriptor(int fd) : mFd{fd}
    {
        if (mFd < 0)
        {
            throw std::invalid_argument { "Invalid descriptor." };
        }
    }

    ~FileDescriptor()
    {
        if ( -1 == ::close(mFd) )
        {
            // perhaps log some error
        }
    }

    int get() const { return mFd; }

    operator int() const { return get(); }
};
```

Resource Acquisition Is Initialization (RAII)

File descriptor

```
class FileDescriptor
{
    int mFd = -1;
public:
    /*
      ...
    */

    FileDescriptor(const FileDescriptor&) = delete;
    FileDescriptor& operator=(const FileDescriptor&) = delete;

    FileDescriptor(FileDescriptor&& other) noexcept
    {
        mFd = other.mFd;
        other.mFd = -1;
    }
    FileDescriptor& operator=(FileDescriptor&& other) noexcept
    {
        mFd = other.mFd;
        other.mFd = -1;
        return *this;
    }
};
```

Resource Acquisition Is Initialization (RAII)

File descriptor

```
{
    int fd = socket(/*...*/)
    if (fd != -1)
    {
        //...
    }

    close(fd);
}
{
    int fd = open("/dev/random");
    if (fd != -1)
    {
        //...
    }

    close(fd);
}
```

Resource Acquisition Is Initialization (RAII)

File descriptor

```
{
    try
    {
        FileDescriptor fd{socket(/*...*/)};
    }
    catch(const std::invalid_argument& e) {}
}
{
    try
    {
        FileDescriptor fd{open("/dev/random")};
    }
    catch(const std::invalid_argument& e) {}
}
```

Function parameters.

Motivation

- C++ offers a variaty of options to pass parameters to functions

Motivation

- C++ offers a variaty of options to pass parameters to functions
- With C++11 came r-value references and smart pointers adding more options

Motivation

- C++ offers a variaty of options to pass parameters to functions
- With C++11 came r-value references and smart pointers adding more options
- In many cases more than 1 option will work but usually only 1 makes the most sense

- `fun(int p)`
- `fun(MyClass* p)`
- `fun(const MyClass* p)`
- `fun(MyClass& p)`
- `fun(const MyClass& p)`
- `fun(MyClass** p)`
- `fun(MyClass*& p)`
- `fun(MyClass*** p)`
- ...

- fun(int p)
- fun(MyClass* p)
- fun(const MyClass* p)
- fun(MyClass& p)
- fun(const MyClass& p)
- fun(MyClass** p)
- fun(MyClass*& p)
- fun(MyClass*** p)
- …

- fun(MyClass&& p)
- fun(unique_ptr<MyClass> p)
- fun(shared_ptr<MyClass> p)
- fun(const shared_ptr<MyClass>& p)
- template<class T> fun(T&& p)
- …

Sinking value

C++03 approach

```
class MyClass
{
    std::string mName;
    int mCount = 0;
public:
    void setName(const std::string& name)
    {
        mName = name;
    }

    void setCount(int count)
    {
        mCount = count;
    }
};
```

Pass by reference

Copying object is expensive.

Sinking value

C++03 approach

```
class MyClass
{
    std::string mName;
    int mCount = 0;
public:
    void setName(const std::string& name)
    {
        mName = name;
    }

    void setCount(int count)
    {
        mCount = count;
    }
};
```

Pass by value

Copying object is cheap.

Sinking value

With C++11 standard the common way of sinking value has changed.

Sinking value

```cpp
class MyClass
{
    std::string mName;
public:
    void setName(std::string name)
    {
        mName = std::move(name);
    }
};
```

```cpp
MyClass m;
std::string s1 = "Yoda";
m.setName(s1); // copy + move
m.setName("Luke"); // move
m.setName(std::move(s1)); // move + move
```

Sinking value

```cpp
class MyClass
{
    std::string mName;
public:
    void setName(std::string name)
    {
        mName = std::move(name);
    }
};
```

```cpp
MyClass m;
std::string s1 = "Yoda";
m.setName(s1); // copy + move
m.setName("Luke"); // move
m.setName(std::move(s1)); // move + move
```

Pass by value

Moving object is cheap.

Sinking value

```cpp
class MyClass
{
    std::string mName;
public:
    void setName(std::string name)
    {
        mName = std::move(name);
    }
};
```

```cpp
MyClass m;
std::string s1 = "Yoda";
m.setName(s1); // copy + move
m.setName("Luke"); // move
m.setName(std::move(s1)); // move + move
```

```cpp
class MyClass
{
    std::string mName;
public:
    void setName(const std::string& name)
    {
        mName = name;
    }
};
```

```cpp
MyClass m;
std::string s1 = "Yoda";
m.setName(s1); // copy
m.setName("Luke"); // copy
m.setName(std::move(s1)); // copy
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

```
class MyClass
{
    Data mData;
public:
    void setData(Data data)
    {
        mData = std::move(data);
    }
};
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};


class MyClass
{
    Data mData;
public:
    void setData(Data data)
    {
        mData = std::move(data);
    }
};
```

```
MyClass m;
Data d;
m.setData(std::move(d)); // move + move
```

Sinking value

```cpp
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

```cpp
class MyClass
{
    Data mData;
public:
    void setData(Data data)
    {
        mData = std::move(data);
    }
};
```

Not efficient

The object is moved twice.

```cpp
MyClass m;
Data d;
m.setData(std::move(d)); // move + move
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

```
class MyClass
{
    Data mData;
public:
    void setData(Data&& data)
    {
        mData = std::move(data);
    }
};
```

```
MyClass m;
Data d;
m.setData(std::move(d)); // move
```

Sinking value

```
struct Data
{
    Data() = default;
    Data(const Data& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    Data(Data&& d)
    {
        std::copy_n(d.buff, 1000, buff);
    }

    char buff[1000];
};
```

```
class MyClass
{
    Data mData;
public:
    void setData(Data&& data)
    {
        mData = std::move(data);
    }
};
```

Efficient

The object is moved only once.

```
MyClass m;
Data d;
m.setData(std::move(d)); // move
```

Sinking value

```
class MyClass
{
    Data mData;
public:
    void setData(Data&& data)
    {
        mData = std::move(data);
    }
};
```

```
MyClass m;
Data d;
m.setData(d);
```

Sinking value

```cpp
class MyClass
{
    Data mData;
public:
    void setData(Data&& data)
    {
        mData = std::move(data);
    }
};
```

```cpp
MyClass m;
Data d;
m.setData(d);
```

Compiler error

Passing l-value reference to a function which takes r-value reference.

Sinking value

```
class MyClass
{
    Data mData;
public:
    void setData(Data&& data)
    {
        mData = std::move(data);
    }
    void setData(const Data& data)
    {
        mData = data;
    }
};
```

```
MyClass m;
Data d;
m.setData(d);
```

OK

Another overload which takes l-value reference.

Sinking value

Prefer passing by value when `move` operation for the passed object is cheap.

Example

```
class MyClass
{
   std::string mName;
public:
   MyClass(std::string name) : mName{std::move(name)}{}
   void setName(std::string name) {mName = std::move(name);}
};
```

Sinking value

Prefer passing by r-value when `move` operation for the passed object is expensive.

Example

```
class MyClass
{
   BigData mData;
public:
   MyClass(const BigData& data) : mData{data}{}
   MyClass(BigData&& data) : mData{std::move(data)}{}
   void setData(const BigData& data) {mData = data;}
   void setData(BigData&& data) {mData = std::move(data);}
};
```

Ownership transfer

```
void f(std::unique_ptr<A> a)
{
}
```

```
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

```
void f(std::unique_ptr<A>&& a)
{
}
```

```
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

Ownership transfer

```
void f(std::unique_ptr<A> a)
{
}
```

```
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

```
void f(std::unique_ptr<A>&& a)
{
}
```

```
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

What's the difference?

Ownership transfer

```cpp
void f(std::unique_ptr<A> a)
{
   //ownership has already been transferred
   //from caller

}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

```cpp
void f(std::unique_ptr<A>&& a)
{
   //ownership has not yet been transferred
   //from caller.
   //Only the reference has been passed.
}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
```

Ownership transfer

```cpp
void f(std::unique_ptr<A> a)
{
   //ownership has already been transferred
   //from caller

}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
// ptr has nullptr value
```

Guaranteed ownership transfer

Passing `unique_ptr` by value so the ptr is moved to function argument.

```cpp
void f(std::unique_ptr<A>&& a)
{
   //ownership has not yet been transferred
   //from caller.
   //Only the reference has been passed.
}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
//ptr may or may not have nullptr value
```

Potential ownership transfer

Passing `unique_ptr` by reference so the ptr is **NOT** moved function argument. It may or may not be moved inside the function.

Ownership transfer

```cpp
void f(std::unique_ptr<A> a)
{
   //ownership has already been transferred
   //from caller

}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
// ptr has nullptr value
```

Guaranteed ownership transfer

Passing `unique_ptr` by value so the ptr is moved to function argument.

```cpp
void f(std::unique_ptr<A>&& a)
{
   //ownership has not yet been transferred
   //from caller.
   //Only the reference has been passed.
}
```

```cpp
auto ptr = std::make_unique<A>();
f(std::move(ptr));
//ptr may or may not have nullptr value
```

Potential ownership transfer

Passing `unique_ptr` by reference so the ptr is **NOT** moved function argument. It may or may not be moved inside the function.

Calling `std::move` does not imply that anything is being moved.

std::move

Standard implementation

```
/**
 *  @brief  Convert a value to an rvalue.
 *  @param  __t  A thing of arbitrary type.
 *  @return The parameter cast to an rvalue-reference to allow moving it.
 */
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept
{ return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
```

std::move

Standard implementation

```
/**
 *  @brief  Convert a value to an rvalue.
 *  @param  __t  A thing of arbitrary type.
 *  @return The parameter cast to an rvalue-reference to allow moving it.
 */
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept
{ return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
```

`std::move` does not move anything. Just casts to r-value reference.

std::move

Standard implementation

```
/**
 *  @brief  Convert a value to an rvalue.
 *  @param  __t  A thing of arbitrary type.
 *  @return The parameter cast to an rvalue-reference to allow moving it.
 */
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept
{ return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
```

`std::move` does not move anything. Just casts to r-value reference.

The actual moving happens in `move constructor` or `move assigment operator`.

shared_ptr<T>

Where could we use it?

shared_ptr<T>

Event loop example...

```
struct TransactionContext
{
   ...
   std::vector<Object> objectsCreated;
   std::vector<Object> objectsDeleted;
   std::vector<Message> messagesToSend;
   ...
};


void processEvent(const Event& e, TransactionContext& context){...}
void commitTransaction(const TransactionContext& context){...}
void sendMessages(const TransactionContext& context){...}


void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      TransactionContext context;
      processEvent(event,  context);
      commitTransaction(context);
      sendMessages(context);
   }
}
```

```cpp
struct TransactionContext
{
   ...
   std::vector<Object> objectsCreated;
   std::vector<Object> objectsDeleted;
   std::vector<Message> messagesToSend;
   ...
};
```

```cpp
void processEvent(const Event& e, TransactionContext& context){...}
void commitTransaction(const TransactionContext& context){...}
void sendMessages(const TransactionContext& context){...}
```

```cpp
void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      TransactionContext context;
      processEvent(event, context);
      commitTransaction(context);
      sendMessages(context);
   }
}
```

```cpp
struct TransactionContext
{
   ...
   std::vector<Object> objectsCreated;
   std::vector<Object> objectsDeleted;
   std::vector<Message> messagesToSend;
   ...
};



void processEvent(const Event& e, TransactionContext& context){...}
void commitTransaction(const TransactionContext& context){...}
void sendMessages(const TransactionContext& context){...}



void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      TransactionContext context;
      processEvent(event,  context);
      commitTransaction(context);
      sendMessages(context);
   }
}
```

```cpp
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}


void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      TransactionContext context;
      processEvent(event,  context);
      commitTransaction(context);
      sendMessages(context);
   }
}
```

```cpp
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}


void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      TransactionContext context;
      processEvent(event,  context);
      commitTransaction(context);
      sendMessages(context);
   }
}
```

Blocks the event loop.

```cpp
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}


void evtLoop()
{
   while (true)
    {
        auto event = receiveEvent();
        TransactionContext context;
        processEvent(event,  context);
        commitTransaction(context);
        std::thread(sendMessages, context)).detach();
    }
}
```

```cpp
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}

void evtLoop()
{
   while (true)
    {
        auto event = receiveEvent();
        TransactionContext context;
        processEvent(event,  context);
        commitTransaction(context);
        std::thread(sendMessages, context)).detach();
    }
}
```

Makes a copy of `context` object.

```
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}


void evtLoop()
{
    while (true)
 {
    auto event = receiveEvent();
    TransactionContext context;
    processEvent(event,  context);
    commitTransaction(context);
    std::thread(sendMessages, std::cref(context))).detach();
 }
}
```

```cpp
void sendMessages(const TransactionContext& context)
{
   for (const auto& m : context.messagesToSend)
   {
      auto sent = send(m);
      if (!sent) {...}
   }
}


void evtLoop()
{
    while (true)
 {
    auto event = receiveEvent();
    TransactionContext context;
    processEvent(event,  context);
    commitTransaction(context);
    std::thread(sendMessages, std::cref(context))).detach();
 }
}
```

Race condition - `context` may get destroyed before `sendMessages` completes.

```cpp
void sendMessages(std::shared_ptr<TransactionContext> context)
{
   for (const auto& m : context->messagesToSend)
   {
      auto sent = send(m); //synchronous send
      if (!sent) {...}
   }
}


void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      auto context = std::make_shared<TransactionContext>();
      processEvent(event, *context);
      commitTransaction(*context);
      std::thread(sendMessages, context).detach();
   }
}
```

Both threads share ownership of `context`.

```cpp
void evtLoop()
{
   while (true)
   {
      auto event = receiveEvent();
      auto context = std::make_shared<TransactionContext>();
      processEvent(event, *context);
      commitTransaction(*context);
      std::thread(sendMessages, context).detach();
   }
}
```

Both threads share ownership of `context`.

Use `shared_ptr` when a function is called in a different thread and shares ownership of the paramater with another thread.

Returning allocated memory from a function via output parameter

```
size_t allocateBuffer(char** buf)
{
    *buf = new char[100];
    ...
    return 100;
}
```

```
char* b;
size_t s = allocateBuffer(&b);
```

C/C++03 style

Pass pointer by pointer.

Returning allocated memory from a function via output parameter

```
size_t allocateBuffer(char** buf)
{
    *buf = new char[100];
    ...
    return 100;
}
```

```
char* b;
size_t s = allocateBuffer(&b);
```

C/C++03 style

Pass pointer by pointer.

```
size_t allocateBuffer(char*& buf)
{
    buf = new char[100];
    ...
    return 100;
}
```

```
char* b;
size_t s = allocateBuffer(b);
```

C++03 style

Pass pointer by reference.

Returning allocated memory from a function via output parameter

```
size_t allocateBuffer(char** buf)
{
   *buf = new char[100];
   ...
   return 100;
}
```

```
char* b;
size_t s = allocateBuffer(&b);
```

C/C++03 style

Pass pointer by pointer.

```
size_t allocateBuffer(char*& buf)
{
   buf = new char[100];
   ...
   return 100;
}
```

```
char* b;
size_t s = allocateBuffer(b);
```

C++03 style

Pass pointer by reference.

Both methods imply raw pointer memory ownership.

```
size_t allocateBuffer(char*& buf)
{
   buf = new char[100];
   ...
   return 100;
}

char* b;
size_t s = allocateBuffer(b);
```

```cpp
size_t allocateBuffer(std::unique_ptr<char[]>& buf)
{
   buf = std::make_unique<char[]>(100);
   ...
   return 100;
}

std::unique_ptr<char[]> b;
auto s = allocateBuffer(b);
```

```cpp
size_t allocateBuffer(std::unique_ptr<char[]>& buf)
{
    buf = std::make_unique<char[]>(100);
    ...
    return 100;
}

std::unique_ptr<char[]> b;
auto s = allocateBuffer(b);
```

"Empty" pointer needs to be declared prior to calling the method.

```cpp
size_t allocateBuffer(std::unique_ptr<char[]>& buf)
{
    buf = std::make_unique<char[]>(100);
    ...
    return 100;
}

std::unique_ptr<char[]> b;
auto s = allocateBuffer(b);
```

Two constructor calls and one move assignment operator call to create a single object.

```cpp
size_t allocateBuffer(std::unique_ptr<char[]>& buf)
{
    buf.reset(new char[100]);
    ...
    return 100;
}

std::unique_ptr<char[]> b;
auto s = allocateBuffer(b);
```

Just one default constructor call but now using explicit new.

```
size_t allocateBuffer(std::unique_ptr<char[]>& buf)
{
    buf.reset(new char[100]);
    ...
    return 100;
}

std::unique_ptr<char[]> b;
auto s = allocateBuffer(b);
```

Isn't it just better to return the `unique_ptr`...?

```cpp
std::pair<size_t, std::unique_ptr<char[]>> allocateBuffer()
{
   auto buf = std::make_unique<char[]>(100);
   ...
   return {100, std::move(buf)};
}

auto b = allocateBuffer();
```

```cpp
std::pair<size_t, std::unique_ptr<char[]>> allocateBuffer()
{
    auto buf = std::make_unique<char[]>(100);
    ...
    return {100, std::move(buf)};
}

auto b = allocateBuffer();
```

No "empty" value. The pointer is declared and initialized with the desired value at the same time.

```cpp
std::pair<size_t, std::unique_ptr<char[]> allocateBuffer()
{
    auto buf = std::make_unique<char[]>(100);
    ...
    return {100, std::move(buf)};
}

auto b = allocateBuffer();
```

No "empty" value. The pointer is declared and initialized with the desired value at the same time.

More concise code.

```cpp
std::pair<size_t, std::unique_ptr<char[]> allocateBuffer()
{
    auto buf = std::make_unique<char[]>(100);
    ...
    return {100, std::move(buf)};
}

auto b = allocateBuffer();
```

No "empty" value. The pointer is declared and initialized with the desired value at the same time.

More concise code.

Return Value Optimization insures only 1 constructor call.

```cpp
std::pair<size_t, std::unique_ptr<char[]> allocateBuffer()
{
    auto buf = std::make_unique<char[]>(100);
    ...
    return {100, std::move(buf)};
}

auto b = allocateBuffer();
```

No "empty" value. The pointer is declared and initialized with the desired value at the same time.

More concise code.

Return Value Optimization insures only 1 constructor call.

Logically dependent values of buffer pointer and size are tied into a single `pair` object.

```cpp
bool createA(std::unique_ptr<A>& a)
{
    if (success)
    {
        a = std::make_unique<A>();
        return true;
    }
    else
    {
        a = nullptr;
        return false;
    }
}
```

```cpp
std::unique_ptr<A> a;
if (createA(a))
{
}
```

Indication of failure

Return `false` if function fails.

```cpp
std::unique_ptr<A> createA()
{
    if (success)
    {
        return std::make_unique<A>();
    }
    else
    {
        return nullptr;
    }
}
```

```cpp
auto a = createA();
if (a)
{
}
```

If the function fails then the nullptr is enough of the evidence of the failure...

```cpp
ErrorCode createA(std::unique_ptr<A>& a)
{
    if (success)
    {
        a = std::make_unique<A>();
        return ErrorCode::SUCCESS;
    }
    else if (resource_busy)
    {
        a = nullptr;
        return ErrorCode::RESOURCE_BUSY;
    }
    else if(resource_no_exist)
    {
        a = nullptr;
        return ErrorCode::NO_RESOURCE;
    }
}
```

Sometimes more concrete error code is needed...

```cpp
std::uniue_ptr<A> a;
auto err = createA(a);
if (err != ErrorCode::SUCCESS)
{
}
```

```cpp
ErrorCode createA(std::unique_ptr<A>& a)
{
    if (success)
    {
        a = std::make_unique<A>();
        return ErrorCode::SUCCESS;
    }
    else if (resource_busy)
    {
        a = nullptr;
        return ErrorCode::RESOURCE_BUSY;
    }
    else if(resource_no_exist)
    {
        a = nullptr;
        return ErrorCode::NO_RESOURCE;
    }
}
```

Sometimes more concrete error code is needed...

Again, we need to declare a variable before we know what to initialize it with.

```cpp
std::uniue_ptr<A> a;
auto err = createA(a);
if (err != ErrorCode::SUCCESS)
{
}
```

```cpp
std::pair<ErrorCode, std::unique_ptr<A>>
 createA()
{
   if (success)
   {
      return {ErrorCode::SUCCESS, std::make_unique<A>()};
   }
   else if (resource_busy)
   {
      return {ErrorCode::RESOURCE_BUSY, nullptr};
   }
   else if(resource_no_exist)
   {
      return {ErrorCode::NO_RESOURCE, nullptr};
   }
}
```

```cpp
auto p = createA();
if (p.first == ErrorCode::SUCCESS)
{
   auto a = std::move(p.second);
}
```

Return both error code and value as pair.

```cpp
std::unique_ptr<A> createA(ErrorCode& error)
{
    if (success)
    {
        error = ErrorCode::SUCCESS;
        return std::make_unique<A>();
    }
    else if (resource_busy)
    {
        error = ErrorCode::RESOURCE_BUSY;
        return nullptr;
    }
    else if(resource_no_exist)
    {
        error = ErrorCode::NO_RESOURCE;
        return nullptr;
    }
}
```

ErrorCode as ouput parameter?

```cpp
ErrorCode error;

auto p = createA(error);
if (error == ErrorCode::SUCCESS)
{
}
```

```cpp
std::unique_ptr<A> createA(ErrorCode& error)
{
    if (success)
    {
        error = ErrorCode::SUCCESS;
        return std::make_unique<A>();
    }
    else if (resource_busy)
    {
        error = ErrorCode::RESOURCE_BUSY;
        return nullptr;
    }
    else if(resource_no_exist)
    {
        error = ErrorCode::NO_RESOURCE;
        return nullptr;
    }
}
```

```cpp
ErrorCode error;

auto p = createA(error);
if (error == ErrorCode::SUCCESS)
{
}
```

`ErrorCode` as ouput parameter?

Not uncommon pattern

E.g. boost filesystem API

```cpp
std::unique_ptr<A> createA(ErrorCode& error)
{
    if (success)
    {
        error = ErrorCode::SUCCESS;
        return std::make_unique<A>();
    }
    else if (resource_busy)
    {
        error = ErrorCode::RESOURCE_BUSY;
        return nullptr;
    }
    else if(resource_no_exist)
    {
        error = ErrorCode::NO_RESOURCE;
        return nullptr;
    }
}
```

```cpp
ErrorCode error;

auto p = createA(error);
if (error == ErrorCode::SUCCESS)
{
}
```

`ErrorCode` as ouput parameter?

Not uncommon pattern

E.g. boost filesystem API

Uninitialized variable

```cpp
std::unique_ptr<A> createA(ErrorCode& error)
{
    if (success)
    {
        error = ErrorCode::SUCCESS;
        return std::make_unique<A>();
    }
    else if (resource_busy)
    {
        error = ErrorCode::RESOURCE_BUSY;
        return nullptr;
    }
    else if(resource_no_exist)
    {
        error = ErrorCode::NO_RESOURCE;
        return nullptr;
    }
}
```

```cpp
ErrorCode error = ErrorCode::SUCCESS;


auto p = createA(error);
if (error == ErrorCode::SUCCESS)
{
}
```

`ErrorCode` as ouput parameter?

Not uncommon pattern

E.g. boost filesystem API

```cpp
std::unique_ptr<A> createA()
{
   if (success)
   {
      return std::make_unique<A>();
   }
   else if (resource_busy)
   {
      throw Exception{ErrorCode::RESOURCE_BUSY};
   }
   else if(resource_no_exist)
   {
      throw Exception{ErrorCode::NO_RESOURCE};
   }
}
```

Throwing exception may be a good solution when cause of the failure is internal system malfunction, not user error.

Avoid passing pointer to a pointer or reference to a pointer to a function to allocate memory for that pointer.

Example

```
size_t allocateBuffer(char** buf);
size_t allocateBuffer(char*& buf);
```

Consider not passing `non-const` `unique_ptr` l-value reference to allocate memory.

Example

```cpp
size_t allocateBuffer(std::unique_ptr<char[]>& buf);
bool createA(std::unique_ptr<A>& a);
ErrorCode createA(std::unique_ptr<A>& a);
```

Instead return `unique_ptr` from the function. Return `std::pair` if an additional variable (e.g. size, error code) needs to be returned.

Example

```cpp
std::unique_ptr<A> createA();
std::pair<size_t, std::unique_ptr<char[]> allocateBuffer();
std::pair<ErrorCode, std::unique_ptr<A>> createA();
```

Consider not returning values using an output parameter.

Example

```
std::unique_ptr<A> createA(ErrorCode& error);
```

Consider not returning values using an output parameter.

Example

```
std::unique_ptr<A> createA(ErrorCode& error);
```

Consider throwing exceptions rather than returning error codes.

Raw pointers and references as function parameters

Raw pointers and references as function parameters

Pass complex types to avoid copying or moving

Example

```
void f(const A& a, const B& b);
void g(const A* a, const B* b);
```

Raw pointers and references as function parameters

Pass complex types to avoid copying or moving

Example

```
void f(const A& a, const B& b);
void g(const A* a, const B* b);
```

Pass objects to be modified by the callee

Example

```
void f(A& a, B& b);
void g(A* a, B* b);
```

Raw pointer or reference?

Raw pointer or reference?

Approach 1

- Use (`const`) reference if a parameter is read-only.
- Use (non-`const`) pointer if a parameter is read-write.

Example

```
void f(const A& a);
void g(A* a);

A a;
f(a); // doesn't modify a
g(&a); // modifies a
```

Raw pointer or reference?

Approach 2

- Use reference if a parameter is mandatory
- Use pointer if a parameter is optional (`nullptr` is a valid value)

Example

```
void f1(const A& a);
void f2(A& a);
void g1(const A* a);
void g2(A* a);

A a;
f1(a); // a is mandatory and not modified
f2(a); // a is mandatory and may be modified
g1(&a); // a is optional and not modified
g2(&a); // a is optional and may be modified
```

Reference to unique_ptr

```
void f(const std::uniue_ptr<A>& a);

auto a = std::make_unique<A>();
f(a);
```

Reference to unique_ptr

```
void f(const std::uniue_ptr<A>& a);

auto a = std::make_unique<A>();
f(a);
```

Not very practical.

Reference to unique_ptr

```
void f(const A& a);

auto a = std::make_unique<A>();
f(*a);
```

Just use const r-value refernce.

Reference to unique_ptr

```
std::vector<std::unique_ptr<A>> vec;
std::find_if(vec.begin(), vec.begin(),
             [](const std::unique_ptr<A>& v){ /*return*/ });
```

Used as predicate for STL algorithm where unique_ptr is the value.

Universal reference

```
template <typename T>
void foo(T&& t);
```

Universal reference

```
template <typename T>
void foo(T&& t);
```

`T&&` is NOT an r-value reference.

Universal reference

```
template <typename T>
void foo(T&& t);
```

`T&&` is NOT an r-value reference.

`T&&` is a 'universal' reference.

Universal reference

```
template <typename T>
void foo(T&& t);
```

`T&&` is NOT an r-value reference.

`T&&` is a 'universal' reference.

`T&&` can be either an r-value or l-value reference depending on what was passed.

Universal reference

```
template <typename T>
void foo(T&& t);
```

`T&&` is NOT an r-value reference.

`T&&` is a 'universal' reference.

`T&&` can be either an r-value or l-value reference depending on what was passed.

Reference collapsing rules

- A& & collapses to A&
- A& && collapses to A&
- A&& & collapses to A&
- A&& && collapses to A&&

Universal reference

```
template <typename T>
void f(T&& t);

Foo foo;

f(foo); // t is l-value ref

Foo& lref = foo;
f(lref); // t is l-value ref

f(std::move(foo)); // t is r-value ref

f(Foo{}); // t is r-value ref
```

T is Foo&.

Universal reference

```
template <typename T>
void f(T&& t);

Foo foo;

f(foo); // t is l-value ref

Foo& lref = foo;
f(lref); // t is l-value ref

f(std::move(foo)); // t is r-value ref

f(Foo{}); // t is r-value ref
```

T is Foo&.

Universal reference

```
template <typename T>
void f(T&& t);

Foo foo;

f(foo); // t is l-value ref

Foo& lref = foo;
f(lref); // t is l-value ref

f(std::move(foo)); // t is r-value ref

f(Foo{}); // t is r-value ref
```

T is Foo.

Universal reference

```
template <typename T>
void f(T&& t);

Foo foo;

f(foo); // t is l-value ref

Foo& lref = foo;
f(lref); // t is l-value ref

f(std::move(foo)); // t is r-value ref

f(Foo{}); // t is r-value ref
```

T is Foo.

Universal reference

```
template <typename T>
void g(T&& t)
{
}

template <typename T>
void f(T&& t)
{
    g(t);
}
```

Universal reference

```
template <typename T>
void g(T&& t)
{
    // type of t is l-value ref
}

template <typename T>
void f(T&& t)
{
    // type of t is l-value ref
    g(t);
}
```

```
Foo foo;
f(foo);
```

Universal reference

```
class Foo
{
    Bar bar;
public:
    Foo (Bar&& b) : bar{b} {} // This is a copy, not move
}
```

```
Bar bar;
Foo foo{std::move(bar)};
```

Universal reference

```
class Foo
{
    Bar bar;
public:
    Foo (Bar&& b) : bar{std::move(b)} {} // This is a copy, not move
}
```

```
Bar bar;
Foo foo{std::move(bar)};
```

Universal reference

```
template <typename T>
void g(T&& t)
{
    // t is l-value ref

}

template <typename T>
void f(T&& t)
{
    // t is r-value ref
    g(t);
}
```

```
Foo foo;
f(std::move(foo));
```

Universal reference

```
template <typename T>
void g(T&& t)
{
    // t is l-value ref

}

template <typename T>
void f(T&& t)
{
    // t is r-value ref
    g(t);
}
```

```
Foo foo;
f(std::move(foo));
```

Universal reference

```
template <typename T>
void g(T&& t)
{
    // t is r-value ref
    T t1 = std::move(t);
}

template <typename T>
void f(T&& t)
{
    // t is r-value ref
    g(std::forward<T>(t));
}
```

```
Foo foo;
f(std::move(foo));
```

OK

T is deduced to be Foo.

Always use `std::forward` when passing universal refernce to function which takes universal reference.

Universal reference

Universal references are often used to delagate functions calls:

Universal reference

Universal references are often used to delagate functions calls:

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

Universal reference

Universal references are often used to delagate functions calls:

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

```
template< class Function, class... Args>
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
    async( Function&& f, Args&&... args );
```

Universal reference

```
int sum(const std::vector<int>& v)
{
    return std::accumulate(v.begin(), v.end(), 0);
}

std::vector<int> v = {3,4,5,5,6,8,7};
auto s = std::async(sum, std::cref(v));
```

Constructor parameters

```
struct Foo
{
    Foo() { std::cout << "Foo ctor\n"; }
};

struct Bar
{
    Bar(Foo f) : foo{f} { std::cout << "Bar ctor\n"; }

    Foo foo;
};
```

```
int
main()
{
    Bar b(Foo());
}
```

What's the output of that this program?

Constructor parameters

```
struct Foo
{
    Foo() { std::cout << "Foo ctor\n"; }
};

struct Bar
{
    Bar(Foo f) : foo{f} { std::cout << "Bar ctor\n"; }

    Foo foo;
};
```

```
int
main()
{
    Bar b(Foo());
}
```

Output

Constructor parameters

```
struct Foo
{
    Foo() { std::cout << "Foo ctor\n"; }
};

struct Bar
{
    Bar(Foo f) : foo{f} { std::cout << "Bar ctor\n"; }

    Foo foo;
};
```

```
int
main()
{
    Bar b(Foo());
}
```

Output

Most vexing parse

Compiler will interpret this as a function declaraion not as a constructor call!

Constructor parameters

```
struct Foo
{
    Foo() { std::cout << "Foo ctor\n"; }
};

struct Bar
{
    Bar(Foo f) : foo{f} { std::cout << "Bar ctor\n"; }

    Foo foo;
};
```

```
int
main()
{
    Bar b((Foo()));
}
```

Output

```
Foo ctor
Bar ctor
```

C++03

Additional pair of parenthesis.

Constructor parameters

```
struct Foo
{
    Foo() { std::cout << "Foo ctor\n"; }
};

struct Bar
{
    Bar(Foo f) : foo{f} { std::cout << "Bar ctor\n"; }

    Foo foo;
};
```

```
int
main()
{
    Bar b{Foo{}};
}
```

C++11

Curly braces instead of parenthesis.

Constructor parameters

By default use curly braces in constructor calls.

copy & move constructors

If the **move constructor** is explicitly declared then the **copy constructor** is implicitly deleted (if not declared).

```cpp
class Foo
{
public:
    Foo() = default;
private:
    int val = 0;
}
```

```cpp
Foo foo;
Foo foo2{foo};
```

OK

Both copy & move constructors are implicitly defined.

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(Foo&&) = default;
private:
    int val = 0;
}
```

```cpp
Foo foo;
Foo foo2{foo};
```

Compilation failure

Copy constructor is implicitly deleted because of explicit move constructor declaration.

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(Foo&&) = default;
    Foo(const Foo&) = default;
private:
    int val = 0;
}
```

```cpp
Foo foo;
Foo foo2{foo};
```

OK

Copy constructor is explicitly declared.

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = default;

    template <typename T>
    Foo(T&& t)
    {
        val = t;
    }
private:
    int val = 0;
}
```

```cpp
Foo foo{5};
Foo foo2{std::move(foo)};
```

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = default;

    template <typename T>
    Foo(T&& t)
    {
        val = t;
    }
private:
    int val = 0;
}
```

```cpp
Foo foo{5};
Foo foo2{std::move(foo)};
```

Compilation failure

error: cannot convert Foo to int in assignment

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = default;
    Foo(Foo&&) = default;

    template <typename T>
    Foo(T&& t)
    {
        val = t;
    }
private:
    int val = 0;
}
```

```cpp
Foo foo{5};
Foo foo2{std::move(foo)};
```

OK

Default move constructor overload used as per funcion overloading rules.

```cpp
class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = default;
    Foo(Foo&&) = default;

    template <typename T>
    Foo(T&& t)
    {
        val = t;
    }
private:
    int val = 0;
}
```

```cpp
Foo foo{5};
Foo foo2{std::move(foo)};
```

OK

Default move constructor overload used as per funcion overloading rules.

Function overloading precedence

```cpp
int sum(int a, int b) {return a+b;}
template <typename T> sum(T a, T b) {return a+b;}

sum(1, 5); //non-template overload is chosen
```

```cpp
class Foo
{
public:
    Foo() = default;
    //Foo(const Foo&) = default;
    //Foo(Foo&&) = default;

    template <typename T>
    Foo(T&& t)
    {
        val = t;
    }
private:
    int val = 0;
}
```

```cpp
Foo foo{5};
Foo foo2{std::move(foo)};
```

OK

Default (implicit) move constructor overload used.

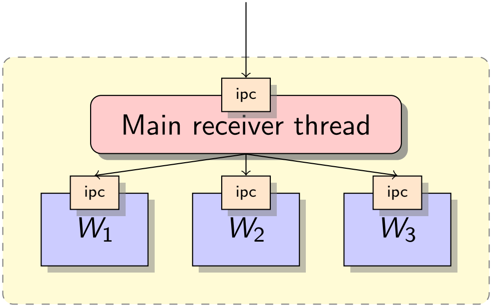
Asynchronous function calls

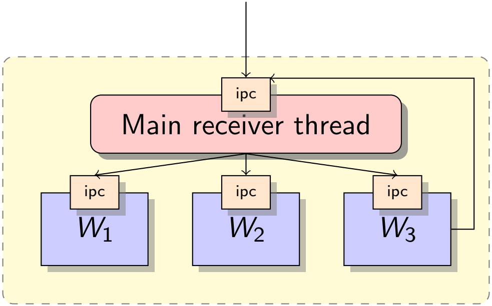
Motivation

- C++11 brings new facilities for asynchronous processing

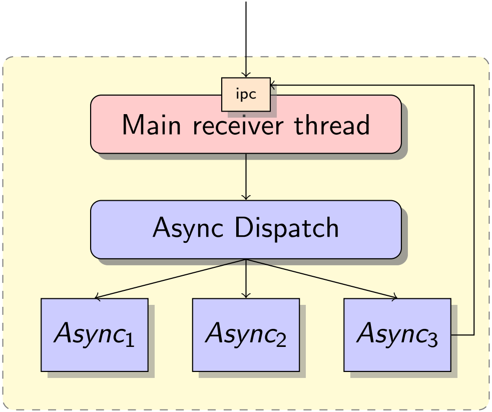
Motivation

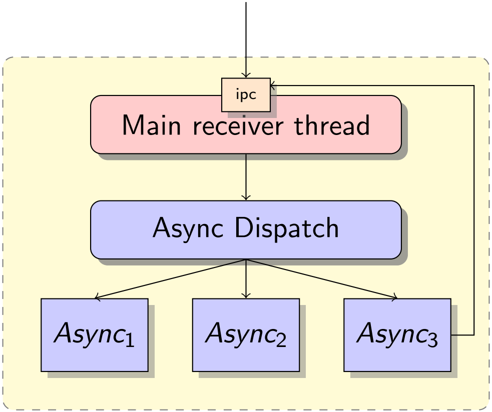
- C++11 brings new facilities for asynchronous processing
- Being able to store any function in a callable object allows creating generic tasks
- With this together we can write a quite generic task dispatcher without polymorphism


ipc
Main receiver thread
ipc
ipc
ipc
W_1
W_2
W_3



Worker thread sends (internal) IPC message to receiver thread.


ipc
Main receiver thread
Async Dispatch
$Async_1$
$Async_2$
$Async_3$


ipc
Main receiver thread
Async Dispatch
$Async_1$
$Async_2$
$Async_3$

Asynchronous task dispatcher

Polymorphic task

```
class ITask;

class AsyncDispatch
{
public:
    //...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

Polymorphic task

```
class ITask;

class AsyncDispatch
{
public:
    // ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

Polymorphic task

```
class ITask;

class AsyncDispatch
{
public:
    // ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

Polymorphic task

```
class ITask;

class AsyncDispatch
{
public:
    //...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

Polymorphic task

```cpp
class ITask;

class AsyncDispatch
{
public:
    // ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```cpp
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

Polymorphic task

```
class ITask;

class AsyncDispatch
{
public:
    // ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

```
class ITask
{
public:
    ITask()          = default;
    virtual ~ITask() = default;

    virtual void operator()() = 0;
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    AsyncDispatch() : mWorker{[this]() { working(); }} {}

    ~AsyncDispatch() {...}


    void
    push(std::unique_ptr<ITask> task)
    {
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mQueue.push_front(std::move(task));
        }
        mCond.notify_one();
    }

    void
    wait()
    {
        std::unique_lock<std::mutex> lock(mMutex);
        mCond.wait(lock, [this]() { return mQueue.empty(); });
    }
private:
    void working() {//..//}
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    AsyncDispatch() : mWorker{[this]() { working(); }} {}

    ~AsyncDispatch() {...}


    void
    push(std::unique_ptr<ITask> task)
    {
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mQueue.push_front(std::move(task));
        }
        mCond.notify_one();
    }

    void
    wait()
    {
        std::unique_lock<std::mutex> lock(mMutex);
        mCond.wait(lock, [this]() { return mQueue.empty(); });
    }
private:
    void working() {//..//}
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    AsyncDispatch() : mWorker{[this]() { working(); }} {}

    ~AsyncDispatch() {...}


    void
    push(std::unique_ptr<ITask> task)
    {
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mQueue.push_front(std::move(task));
        }
        mCond.notify_one();
    }

    void
    wait()
    {
        std::unique_lock<std::mutex> lock(mMutex);
        mCond.wait(lock, [this]() { return mQueue.empty(); });
    }
private:
    void working() {//..//}
};
```

AsyncDispatch

```
class AsyncDispatch
{
    ...
private:
    void
    working()
    {
        while (!mTerminate)
        {
            std::unique_ptr<ITask> task;
            std::unique_lock<std::mutex> lock(mMutex);
            mCond.wait(lock, [this]() { return !mQueue.empty()   mTerminate; });
            if (!mQueue.empty())
            {
                task = std::move(mQueue.back());
                mQueue.pop_back();
            }

            lock.unlock();

            if (task)
            {
                (*task)();
            }

            lock.lock();
            if (mQueue.empty())
            {
                mCond.notify_one();
            }
        }
    }
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    ...

    ~AsyncDispatch()
    {
        wait();
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mTerminate = true;
        }
        mCond.notify_one();
        mWorker.join();
    }

    ...
};
```

AsyncDispatch

```cpp
void
doSomething(const std::string s)
{
    std::cout << "Doing " << s << '\n';
}


int
main()
{
    class TaskDoSomething : public ITask
    {
        std::string param;

    public:
        TaskDoSomething(std::string s) : param{std::move(s)} {}
        void
        operator()() override
        {
            doSomething(param);
        }
    };

    AsyncDispatch dispatch;
    dispatch.push(std::make_unique<TaskDoSomething>("Task 1"));
    dispatch.push(std::make_unique<TaskDoSomething>("Task 2"));
    dispatch.push(std::make_unique<TaskDoSomething>("Task 3"));
}
```

AsyncDispatch

This works but is not very convenient or elegant because:

- In order to exectute a function asychrounsly one has to create a wrapper class

AsyncDispatch

This works but is not very convenient or elegant because:

- In order to exectute a function asychrounsly one has to create a wrapper class
- It's intrusive because that wrapper class has to inherit from a interface

AsyncDispatch

This works but is not very convenient or elegant because:

- In order to exectute a function asychrounsly one has to create a wrapper class
- It's intrusive because that wrapper class has to inherit from a interface
- Passing parameters is non natural - they become members of the wrapper class

AsyncDispatch

```
int
main()
{
    class TaskDoSomething : public ITask
    {
        std::string param;

    public:
        TaskDoSomething(std::string s) : param{std::move(s)} {}
        void
        operator()() override
        {
            doSomething(param);
        }
    };

    AsyncDispatch dispatch;
    dispatch.push(std::make_unique<TaskDoSomething>("Task 1"));
    dispatch.push(std::make_unique<TaskDoSomething>("Task 2"));
    dispatch.push(std::make_unique<TaskDoSomething>("Task 3"));
}
```

AsyncDispatch

```cpp
int
main()
{
    AsyncDispatch dispatch;
    dispatch.push(doSomething, "Task 1");
    dispatch.push(doSomething, "Task 2");
    dispatch.push(doSomething, "Task 3");
}
```

Can we do this?

lambda expression

```
[captures](params)->ret{body}
```

lambda expression

```
int mod = 3;
std::vector<int> v = {4,1,7,3,9,1,8};
auto it = std::find_if(v.begin(), v.end(), [mod](int i){return (i % mod == 0);});
```

lambda expression

```
int mod = 3;
std::vector<int> v = {4,1,7,3,9,1,8};
auto it = std::find_if(v.begin(), v.end(), [&mod](int i){return (i % mod == 0);});
```

lambda expression

```
int mod = 3;
std::vector<int> v = {4,1,7,3,9,1,8};
auto f = [&mod](int i){return (i % mod == 0);};
auto it = std::find_if(v.begin(), v.end(), f);
```

lambda expression

```
std::vector<int> v = {4,1,7,3,9,1,8};
std::thread th{[&v](){std::sort(v.begin(), v.end());}};
```

lambda expression

```
std::vector<int> v = {4,1,7,3,9,1,8};
std::thread th{[&v](){std::sort(v.begin(), v.end(),
                                  [](int a, int b){return a >= b;});}};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    // ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

AsyncDispatch

```cpp
class AsyncDispatch
{
public:
    // ...
private:
    using Task = std::function<void()>;
    std::deque<Task> mQueue;
    std::thread mWorker;
    std::mutex mMutex;
    std::condition_variable mCond;
    bool mTerminate = false;
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    ...

    void
    push(std::unique_ptr<ITask> task)
    {
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mQueue.push_front(std::move(task));
        }
        mCond.notify_one();
    }

    ...
private:
    std::deque<std::unique_ptr<ITask>> mQueue;
    ...
};
```

AsyncDispatch

```
class AsyncDispatch
{
public:
    ...

    template <typename Callable, typename... Args>
    void
    push(Callable&& callable, Args&&... args)
    {
        {
            std::unique_lock<std::mutex> lock(mMutex);
            mQueue.push_front([&]() { callable(std::forward<Args>(args)...); });
        }
        mCond.notify_one();
    }
    ...
private:
    using Task = std::function<void()>;
    std::deque<Task> mQueue;
    ...
};
```

AsyncDispatch

```
class AsyncDispatch
{
    ...
private:
    void
    working()
    {
        while (!mTerminate)
        {
            std::unique_ptr<ITask> task;
            std::unique_lock<std::mutex> lock(mMutex);
            mCond.wait(lock, [this]() { return !mQueue.empty()  mTerminate; });
            if (!mQueue.empty())
            {
                task = std::move(mQueue.back());
                mQueue.pop_back();
            }

            lock.unlock();

            if (task)
            {
                (*task)();
            }

            lock.lock();
            if (mQueue.empty())
            {
                mCond.notify_one();
            }
        }
    }
};
```

AsyncDispatch

```
class AsyncDispatch
{
    ...
private:
    void
    working()
    {
        while (!mTerminate)
        {
            Task task;
            std::unique_lock<std::mutex> lock(mMutex);
            mCond.wait(lock, [this]() { return !mQueue.empty()   mTerminate; });
            if (!mQueue.empty())
            {
                task = std::move(mQueue.back());
                mQueue.pop_back();
            }

            lock.unlock();

            if (task)
            {
                task();
            }

            lock.lock();
            if (mQueue.empty())
            {
                mCond.notify_one();
            }
        }
    }
};
```

AsyncDispatch

```
void
doSomething(const std::string s)
{
    std::cout << "Doing␣" << s << '\n';
}

void
doSomethingElse(const std::string s, int i)
{
    std::cout << "Doing␣else␣" << s << "␣" << i << '\n';
}

int
main()
{
    AsyncDispatch dispatch;
    dispatch.push(doSomething, "Task␣1");
    dispatch.push(doSomethingElse, "Task␣1", 5);
}
```

AsyncDispatch

```
int sum(const std::vector<int>Y vec)
{
    return std::accumulate(vec.begin(), vec.end(), 0);
}
```

std::packaged_task and std::future

```
int plus(int a, int b) {return a + b;}

std::packaged_task<int(int,int)> task(plus);
std::future<int> result = task.get_future();

std::thread th(std::move(task), 2, 10);

// do other stuff here
...

// fetch return value whenever needed
auto s = result.get();
```

AsyncDispatch

```
int sum(const std::vector<int>& vec)
{
    return std::accumulate(vec.begin(), vec.end(), 0);
}
```

AsyncDispatch

```
int sum(const std::vector<int>& vec)
{
    return std::accumulate(vec.begin(), vec.end(), 0);
}

AsyncDispatch dispatch;
std::packaged_task<int(const std::vector<int>&)> task{sum};
auto future_sum = task.get_future();
std::vector<int> vec = {1, 2, 3, 4};
dispatch.push(std::move(task), std::cref{vec});

// do something

auto sum = future_sum.get();
```

std::thread and two-phase construction

```
class A
{
    std::thread mThread;
    std::atomic<bool> mRunning{false};
public:
    A() = default;

    ~A()
    {
        mRunning.store(false);
        mThread.join();
    }

    void
    init()
    {
        auto func = [this]() {
            while (mRunning) { /*do stuff*/ }};
        mRunning.store(true);
        mThread = std::thread{func};
    }
};
```

std::thread and two-phase construction

```
class A
{
    std::thread mThread;
    std::atomic<bool> mRunning{false};
public:
    A() = default;

    ~A()
    {
        mRunning.store(false);
        mThread.join();
    }

    void
    init()
    {
        auto func = [this]() {
            while (mRunning) { /*do stuff*/ }};
        mRunning.store(true);
        mThread = std::thread{func};
    }
};


int main()
{
    A a;
    a.init();
    a.init();
    //..
}
```

std::thread and two-phase construction

```cpp
class A
{
    std::thread mThread;
    std::atomic<bool> mRunning{false};
public:
    A() = default;

    ~A()
    {
        mRunning.store(false);
        mThread.join();
    }

    void
    init()
    {
        auto func = [this]() {
            while (mRunning) { /*do stuff*/ }};
        mRunning.store(true);
        mThread = std::thread{func};
    }
};
```

```cpp
int main()
{
    A a;
    a.init();
    a.init();
    //..
}
```

Program abnormally terminates

`thread` has not been joined prior to being assigned new thread of execution.

std::thread and two-phase construction

```
class A
{
    std::thread mThread;
    std::atomic<bool> mRunning{false};
public:
    /*...*/

    void
    init()
    {
        auto func = [this]() {
            while (mRunning) { /*do stuff*/ }};

        if (mThread.joinable())
        {
            mRunning.store(false);
            mThread.join();
        }
        mRunning.store(true);
        mThread = std::thread{func};
    }
};
```

```
int main()
{
   A a;
   a.init();
   a.init();
   //..
}
```

Program runs

thread is joined prior to restarting.

std::thread and two-phase construction

```cpp
class A
{
    std::thread mThread;
    std::atomic<bool> mRunning{true};
public:
    A() : mThread{[this]() {
            while (mRunning) { /*do stuff*/ }}}
    {
    }

    ~A()
    {
        mRunning.store(false);
        mThread.join();
    }
};
```

```cpp
int main()
{
    A a;
}
```

One-phase construction

`thread` is started during object construction.

C API

Motivation

- It is not uncommon to find C library functions being used in C++ in spite of C++ library equivalents being available

Motivation

- It is not uncommon to find C library functions being used in C++ in spite of C++ library equivalents being available
- While it's OK in some cases, in others C functions are less safe and more prone to errors

Motivation

- It is not uncommon to find C library functions being used in C++ in spite of C++ library equivalents being available
- While it's OK in some cases, in others C functions are less safe and more prone to errors
- Specifically C string manipulation functions are often sources of buffer overflows
    - `strcpy`
    - `strcat`
    - `sscanf`
    - ...

C-string vs `std::string`

C-string vs `std::string`

The general recommendation is: use `std::string` and avoind using C raw string.

C-string vs `std::string`

The general recommendation is: use `std::string` and avoind using C raw string.

but...

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strcpy(ifr.ifr_name, ifname.c_str());
    // ioctl(fd, ..., &ifr);
    //...
}
```

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strcpy(ifr.ifr_name, ifname.c_str());
    // ioctl(fd, ..., &ifr);
    //...
}
```

strcpy

Who guarantees that `ifname` fits into the fixed-size `ifr_name` buffer?

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strncpy(ifr.ifr_name, ifname.c_str(), sizeof(ifr.irf_name));
    // ioctl(fd, ..., &ifr);
    //...
}
```

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strncpy(ifr.ifr_name, ifname.c_str(), sizeof(ifr.irf_name));
    // ioctl(fd, ..., &ifr);
    //...
}
```

strncpy

Less bad but still wrong - buffer is not null-terminated.

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strncpy(ifr.ifr_name, ifname.c_str(), sizeof(ifr.irf_name) - 1);
    ifr.ifr_name[sizeof(ifr.irf_name) - 1] = '\0';
    // ioctl(fd, ..., &ifr);
    //...
}
```

Interfacing with C API

```
void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    strncpy(ifr.ifr_name, ifname.c_str(), sizeof(ifr.irf_name) - 1);
    ifr.ifr_name[sizeof(ifr.irf_name) - 1] = '\0';
    // ioctl(fd, ..., &ifr);
    //...
}
```

strncpy

Correct but could be inefficient.

Safe string copy

```cpp
size_t
safeStringCopy(const std::string& src, char* dest, size_t destLen)
{
    if (destLen == 0)
    {
        return 0;
    }
    auto len = std::min(src.length(), destLen - 1);
    std::copy_n(src.begin(), len, dest);
    dest[len] = '\0';
    return len;
}
```

Safe string copy

```
size_t
safeStringCopy(const std::string& src, char* dest, size_t destLen)
{
    if (destLen == 0)
    {
        return 0;
    }
    auto len = std::min(src.length(), destLen - 1);
    std::copy_n(src.begin(), len, dest);
    dest[len] = '\0';
    return len;
}

void configureNetworkInterface(const std::string& ifname)
{
    ifreq ifr;
    safeStringCopy(ifname, ifr_ifr_name, sizeof(ifr.ifr_name));
    // ioctl(fd, ..., &ifr);
    //...
}
```

Safe string copy

```
safeStringCopy(...)
        push    rbx
        xor     ebx, ebx
        test    rdx, rdx
        je      .L1
        mov     rbx, QWORD PTR [rdi+8]
        sub     rdx, 1
        mov     rcx, rsi
        cmp     rdx, rbx
        cmovbe  rbx, rdx
        test    rbx, rbx
        jne     .L12
.L3:
        mov     BYTE PTR [rcx+rbx], 0
.L1:
        mov     rax, rbx
        pop     rbx
        ret
.L12:
        mov     rsi, QWORD PTR [rdi]
        mov     rdx, rbx
        mov     rdi, rcx
        call    memmove
        mov     rcx, rax
        jmp     .L3
```

Never use `strcpy` to copy a string which lengh is not known at compile time into a fixed size array;

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
strcpy(s, "Sometimes seems OK");
```

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
strcpy(s, "Sometimes seems OK");
```

strcpy

String fits into the buffer.

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
strcpy(s, "Sometimes seems NOT OK");
```

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
strcpy(s, "Sometimes seems NOT OK");
```

strcpy

String does not fit into the buffer.

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
safeStringCopy("Sometimes seems OK", s, sizeof(s));
```

Fixed size buffer copy

```
const int SIZE = 20;

char s[SIZE];
safeStringCopy("Sometimes seems OK", s, sizeof(s));
```

Safe but not efficient

`safeStringCopy(const std::string& src, char* dest, size_t destLen)`

Fixed size buffer copy

```
size_t
safeStringCopy(const char* src, char* dest, size_t destLen)
{
    if (destLen == 0)
    {
        return 0;
    }
    auto len = std::min(strlen(src), destLen - 1);
    std::copy_n(src, len, dest);
    dest[len] = '\0';
    return len;
}

const int SIZE = 20;

char s[SIZE];
safeStringCopy("Sometimes seems OK", s, sizeof(s));
```

C-string vs `std::string`

```cpp
class WithString
{
    std::string mName;
};

std::vector<WithString> vec;
```

```cpp
class WithCString
{
    char mName[20];
};

std::vector<WithCString> vec;
```

C-string vs `std::string`

```cpp
class WithString
{
    std::string mName;
};

std::vector<WithString> vec;
```

```cpp
class WithCString
{
    char mName[20];
};

std::vector<WithCString> vec;
```

What is more efficient?

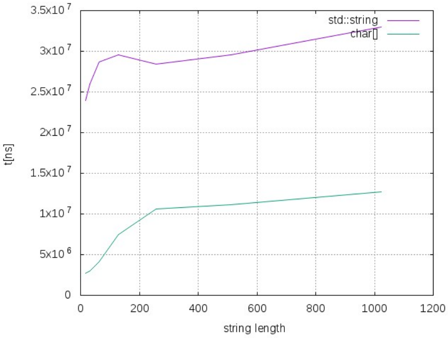


Figure: find

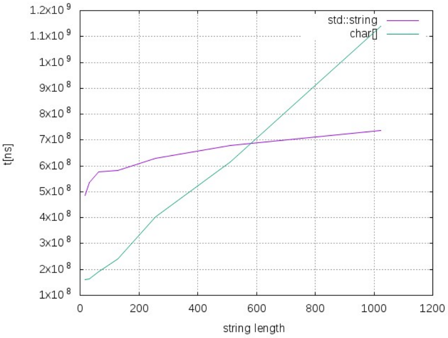


Figure: sort

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    const char* name() const
    {
        return mName;
    }
};
```

C-string vs `std::string`

```cpp
class WithCString
{
    char mName[20];
public:
    const char* name() const
    {
        return mName;
    }
};
```

Returns raw string forcing user to use raw string functions.

C-string vs std::string

```cpp
class WithCString
{
    char mName[20];
public:
    std::string name() const
    {
        return std::string{mName, 20};
    }
};
```

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    std::string name() const
    {
        return std::string{mName, 20};
    }
};
```

Returns std::string but requires a new copy of the original string.

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    std::string name() const
    {
        return std::string{mName, 20};
    }
};
```

```
WithCString str {"Johnny"};
auto n = str.name().c_str();
printf(%s, n);
```

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    std::string name() const
    {
        return std::string{mName, 20};
    }
};
```

```
WithCString str {"Johnny"};
const char* n = str.name().c_str();
printf(%s, n);
```

Return from `str.name()` is a temporary object

Pointer passed to printf is invalid.

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    std::string_view name() const
    {
        return std::string_view{mName, 20};
    }
};
```

C-string vs std::string

```
class WithCString
{
    char mName[20];
public:
    std::string_view name() const
    {
        return std::string_view{mName, 20};
    }
};
```

C++17 introduces `string_view` class which can wrap an existing buffer into `string`-like object.

STL algorithms

Motivation

- Standard Library provides a wide variety of algorithms on containers which don't get as much popularity as they deserve

Motivation

- Standard Library provides a wide variety of algorithms on containers which don't get as much popularity as they deserve
- Programmers tend to write their own raw loops to solve the same problems

Motivation

- Standard Library provides a wide variety of algorithms on containers which don't get as much popularity as they deserve
- Programmers tend to write their own raw loops to solve the same problems
- Usually these are clumsy loops (often nested) with hard to understand control flow
- The loops often are specific to only one container type

Motivation

- Standard Library provides a wide variety of algorithms on containers which don't get as much popularity as they deserve
- Programmers tend to write their own raw loops to solve the same problems
- Usually these are clumsy loops (often nested) with hard to understand control flow
- The loops often are specific to only one container type
- Sometimes these loops are not as efficient as they could be

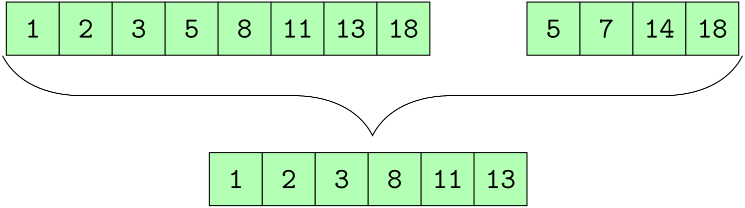
What does this code do?

```
std::vector<int> v1 = {1, 2, 3, 5, 8, 11, 13, 18};
std::vector<int> v2 = {5, 7, 14, 18};
std::vector<int> v3;

for (auto it1 = v1.begin(), it2 = v2.begin(); it1 != v1.end();)
 {
     if (*it1 < *it2)
     {
         v3.push_back(*it1++);
     }
     else if (*it1 > *it2)
     {
         ++it2;
     }
     else
     {
         ++it1;
         ++it2;
     }
 }

// What's in v3?
```

What does this code do?

```
std::vector<int> v1 = {1, 2, 3, 5, 8, 11, 13, 18};
std::vector<int> v2 = {5, 7, 14, 18};
std::vector<int> v3;

std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                    std::back_inserter(v3));
```

std::set_difference



What if vector is not sorted?

```
std::vector<int> v1 = {3, 2, 1, 18, 14, 8, 13, 5};
std::vector<int> v2 = {5, 7, 14, 18};
std::vector<int> v3;
```

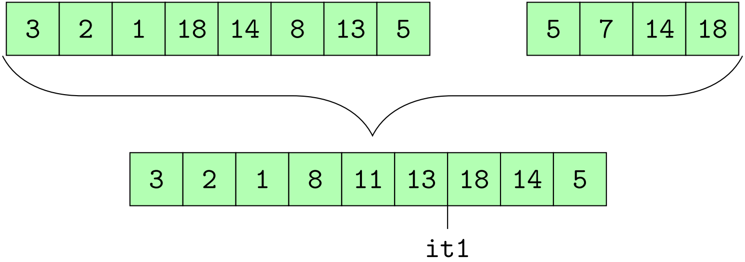
What if vector is not sorted?

```
std::vector<int> v1 = {3, 2, 1, 18, 14, 8, 13, 5};
std::vector<int> v2 = {5, 7, 14, 18};
std::vector<int> v3;


v3 = v1;
auto it = std::remove_if(v3.begin(), v3.end(), [&v2](int x)
        {
            return std::find(v2.begin(), v2.end(), x) != v2.end();
        });
```

What if vector is not sorted?

```
std::vector<int> v1 = {3, 2, 1, 18, 14, 8, 13, 5};
std::vector<int> v2 = {5, 7, 14, 18};
std::vector<int> v3;


auto v3 = v1;
auto it = std::remove_if(v3.begin(), v3.end(), [&v2](int x)
        {
            return std::find(v2.begin(), v2.end(), x) != v2.end();
        });
v3.erase(it, v3.end());
```

std::remove_if



Motivation

Avoid writing raw loops.
Learn STL algorithms and use them where applicable.
Combine multiple STL algorithms into more complex procedures.

Generate random string

```
std::string
randomString(size_t n)
{
    static std::random_device rd;
    std::default_random_engine re(rd());
    std::uniform_int_distribution<char> dist{33, 122};
    std::string s(n, '\0');
    std::generate(s.begin(), s.end(), [&re, &dist]() { return dist(re); });
    return s;
}
```

Check if two containers have same elements

```cpp
template <class Container>
haveSameElements(Container c1, Container c2)
{
    if (c1.size() != c2.size())
    {
        return false;
    }

    std::sort(c1.begin(), c1.end());
    std::sort(c2.begin(), c2.end());
    return c1 == c2;
}
```

Check if two containers have same elements

```
template <class Container>
haveSameElements(const Container c1&, const Container& c2)
{
    return std::is_permutation(c1.begin(), c1.end(), c2.begin());
}
```

Remove duplicates

```
template <typename T> void removeDuplicates (std::vector<T>& v)
{
    if (v.size() > 1)
    {
        typename std::vector<T>::iterator it;
        std::set<T> seen;

        for (it = v.begin(); it != v.end();)
        {
            if (seen.find(*it) == seen.end())
            {
                seen.insert(*it);
                it++;
            }
            else
            {
                it = v.erase(it);
            }
        }
    }
}
```

Remove duplicates

```
template <typename T> void removeDuplicates (std::vector<T>& v)
{
    v.sort(v.begin(), v.end());
    v.erase(std::unique(v.begin(), v.end()), end());
}
```

Find index

```
template<typename T>
int find(const std::vector<T>& v, const T& val)
{
    int idx = 0;
    int found = false;
    for (const auto& x : v)
    {
        if (x == val)
        {
            found = true;
            break;
        }
        idx++;
    }
    return found ? idx : -1;
}
```

Find index

```
template<typename T>
int find(const std::vector<T>& v, const T& val)
{
    auto it = std::find(v.begin(), v.end(), val);
    return it != v.end() ? std::distance(v.begin(), it) : -1;
}
```

Employee database

- Database storing basic information on employees
- Supports basic operations
  - Insertion/Removal
  - Look up
  - Traversal
- Optimized for look-ups and traversals at the cost of insertion and removal
- Goals
  - Implement statistical algorithms with minimal or no usage of raw loops

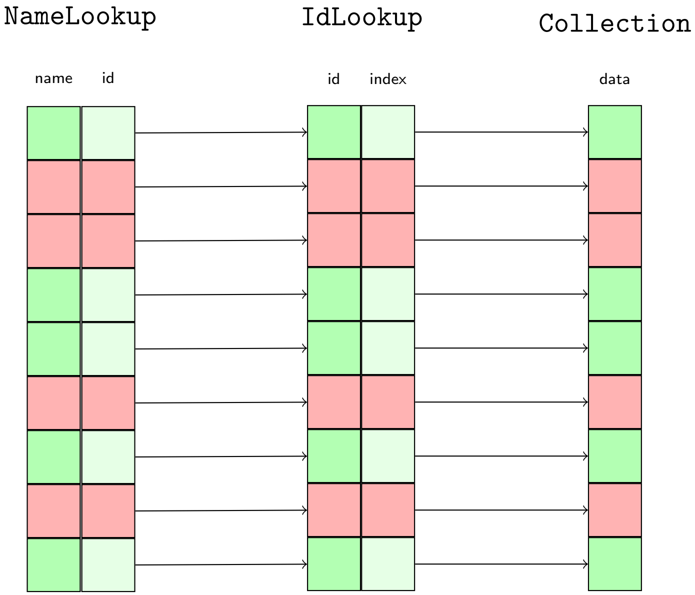
Employee database

```
enum class Profession
{
    ENGINEER,
    DOCTOR,
    LAWYER
};

struct EmployeeRecord
{
    std::string name;
    Profession profession = Profession::ENGINEER;
    int age                 = 0;
    int salary              = 0;
    uint64_t id             = 0;

    EmployeeRecord() = default;
    EmployeeRecord(std::string name, Profession profession, int age, int salary)
        : name(std::move(name)), profession(profession), age(age), salary(salary)
    {
    }
};
```

Employee database

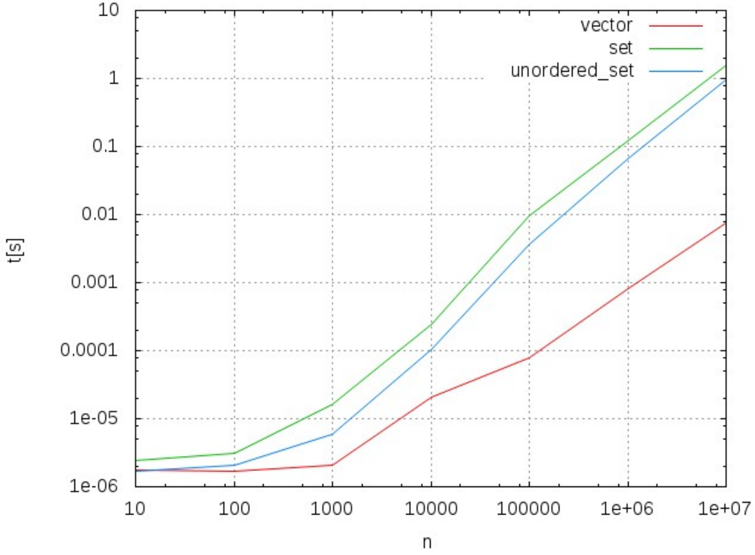
```
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employee database



Iteration efficiency

Cache locality matters



Employee database

```cpp
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employee database

```cpp
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employee database

```cpp
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employee database

```
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employee database

```
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

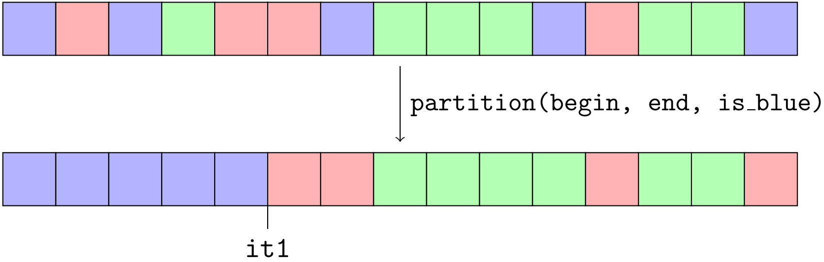
Employee database

```
class EmployeesDb
{
    using Collection = std::vector<EmployeeRecord>;
    using NameLookup = std::unordered_map<std::string, uint64_t>;
    using IdLookup   = std::unordered_map<uint64_t, size_t>;

    Collection mEmployees;
    NameLookup mNameLookup;
    IdLookup mIdLookup;

public:
    using Iterator = Collection::const_iterator;

    EmployeesDb() = default;
    EmployeesDb(std::vector<EmployeeRecord> employees);
    ~EmployeesDb() = default;

    uint64_t insert(EmployeeRecord data);
    void remove(const std::string& name);
    void remove(uint64_t id);
    Iterator find(uint64_t id) const;
    Iterator find(const std::string& name) const;
    size_t size() const;
    Iterator begin() const;
    Iterator end() const;

private:
    void generateIdLookup();
    void generateNameLookup();
};
```

Employees database

```
EmployeesDb::EmployeesDb(std::vector<EmployeeRecord> employees)
    : mEmployees(std::move(employees))
{
    // Collection internally partitioned by Profession
    auto it =
        std::partition(mEmployees.begin(), mEmployees.end(), [](const EmployeeRecord& e) {
            return e.profession == Profession::ENGINEER;
        });
    std::partition(it, mEmployees.end(), [](const EmployeeRecord& e) {
        return e.profession == Profession::DOCTOR;

    });

    generateNameLookup();
    generateIdLookup();
}
```

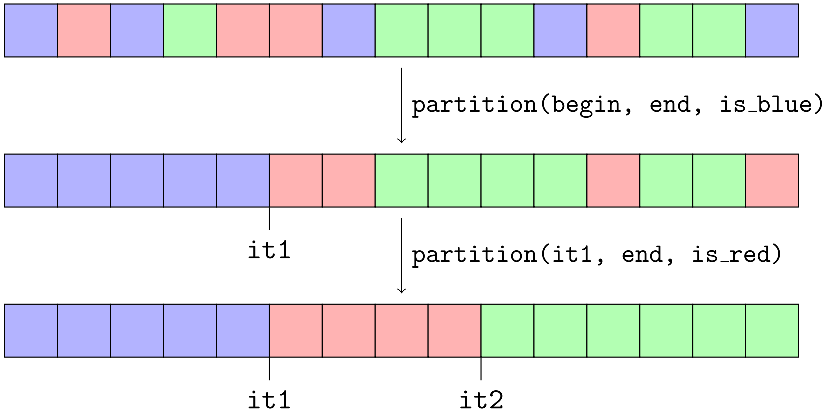
std::partition



Employees database

```
EmployeesDb::EmployeesDb(std::vector<EmployeeRecord> employees)
    : mEmployees(std::move(employees))
{
    // Collection internally partitioned by Profession
    auto it =
        std::partition(mEmployees.begin(), mEmployees.end(), [](const EmployeeRecord& e) {
            return e.profession == Profession::ENGINEER;
        });
    std::partition(it, mEmployees.end(), [](const EmployeeRecord& e) {
        return e.profession == Profession::DOCTOR;

    });

    generateNameLookup();
    generateIdLookup();
}
```

std::partition



Employees database

range

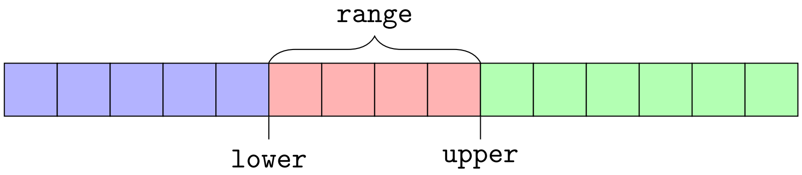
```
std::pair<EmployeesDb::Iterator, EmployeesDb::Iterator>
range(const EmployeesDb& db, Profession profession)
{
    auto compPos1 = [](const EmployeeRecord& e, Profession pos) {
        return e.profession < pos;
    };

    auto compPos2 = [](Profession pos, const EmployeeRecord& e) {
        return pos < e.profession;
    };

    auto begin = std::lower_bound(db.begin(), db.end(), profession, compPos1);
    auto end   = std::upper_bound(db.begin(), db.end(), profession, compPos2);
    return {begin, end};
}
```

Employees database

range

```
std::pair<EmployeesDb::Iterator, EmployeesDb::Iterator>
range(const EmployeesDb& db, Profession profession)
{
    auto compPos1 = [](const EmployeeRecord& e, Profession pos) {
        return e.profession < pos;
    };

    auto compPos2 = [](Profession pos, const EmployeeRecord& e) {
        return pos < e.profession;
    };

    auto begin = std::lower_bound(db.begin(), db.end(), profession, compPos1);
    auto end   = std::upper_bound(db.begin(), db.end(), profession, compPos2);
    return {begin, end};
}
```

std::lower_bound and std::upper_bound



Employees database

Min and Max

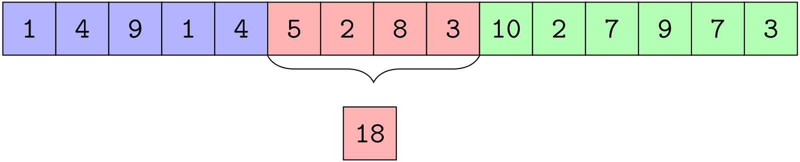
```
std::pair<EmployeesRecord, EmployeesRecord>
minMaxSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r = range(db, profession);
    auto minMax = std::minmax_element(r.first, r.second,
                                [](const EmployeeRecord& e1, const EmployeeRecord& e2) {
                                    return e1.salary < e2.salary;
                                });
      return {*minMax.first, *minMax.second};
}
```

Employees database

Average

```
int
avgSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r = range(db, profession);
    auto totalSalary =
        std::accumulate(r.first, r.second, uint64_t{0},
                        [](uint64_t s, const EmployeeRecord& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(r.first, r.second);
    return totalSalary / noOfEmployees;
}
```

std::accumulate



Employees database
Average

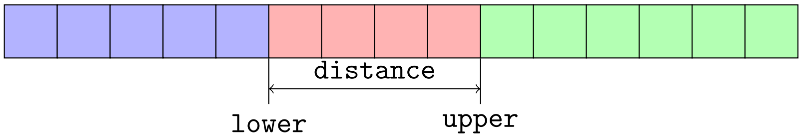
```
int
avgSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r = range(db, profession);
    auto totalSalary =
        std::accumulate(r.first, r.second, uint64_t{0},
                        [](uint64_t s, const EmployeeRecord& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(r.first, r.second);
    return totalSalary / noOfEmployees;
}
```

Employees database

Average

```
int
avgSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r = range(db, profession);
    auto totalSalary =
        std::accumulate(r.first, r.second, uint64_t{0},
                        [](uint64_t s, const EmployeeRecord& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(r.first, r.second);
    return totalSalary / noOfEmployees;
}
```

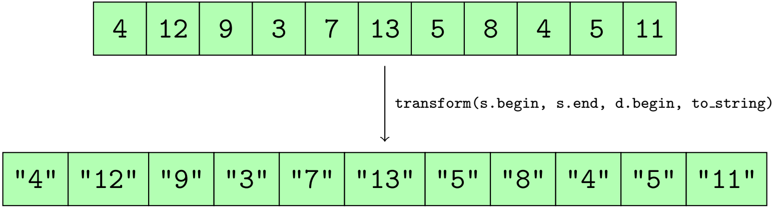
std::distance



Employees database
Median

```
int
medianSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);
    std::vector<int> salaries(noOfEmployees);
    std::transform(r.first, r.second, salaries.begin(),
                   [](const EmployeeRecord& e) { return e.salary; });

    std::nth_element(salaries.begin(), salaries.begin() + salaries.size() / 2,
                     salaries.end());
    return salaries[salaries.size() / 2];
}
```

std::transform

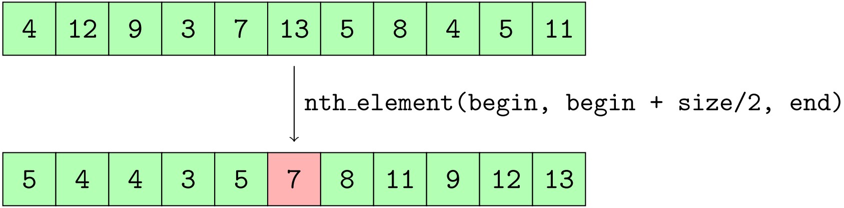


Employees database

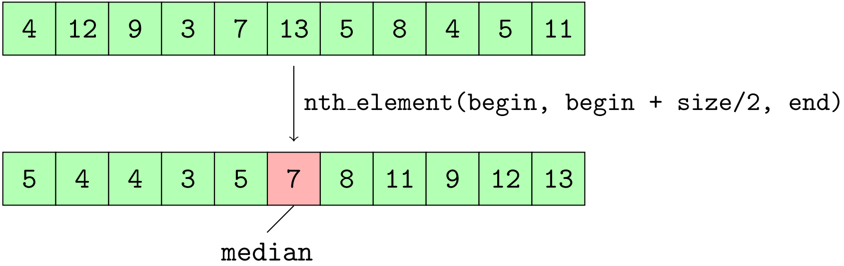
Median

```cpp
int
medianSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);
    std::vector<int> salaries(noOfEmployees);
    std::transform(r.first, r.second, salaries.begin(),
                   [](const EmployeeRecord& e) { return e.salary; });

    std::nth_element(salaries.begin(), salaries.begin() + salaries.size() / 2,
                     salaries.end());
    return salaries[salaries.size() / 2];
}
```

std::nth_element



std::nth_element



Employees database
Median

```cpp
int
medianSalaryPerPosition(const EmployeesDb& db, Profession profession)
{
    auto r              = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);
    std::vector<int> salaries(noOfEmployees);
    std::transform(r.first, r.second, salaries.begin(),
                  [](const EmployeeRecord& e) { return e.salary; });

    std::nth_element(salaries.begin(), salaries.begin() + salaries.size() / 2,
                    salaries.end());
    return salaries[salaries.size() / 2];
}
```

Employees database

Top N

```
std::vector<EmployeesDb::Iterator>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    std::nth_element(
        data.begin(), data.end() - n, data.end(),
        [](const Helper& e1, const Helper& e2) { return e1.salary < e2.salary; });

    std::sort(data.end() - n, data.end(),
              [](const Helper& e1, const Helper& e2) { return e1.salary > e2.salary; });

    std::vector<EmployeesDb::Iterator> employeesTopN(n);
    std::transform(data.end() - n, data.end(), employeesTopN.begin(),
                   [&db](const Helper& e) { return db.find(e.id); });
    return employeesTopN;
}
```

Employees database

Top N

```
std::vector<EmployeesDb::Iterator>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    std::nth_element(
        data.begin(), data.end() - n, data.end(),
        [](const Helper& e1, const Helper& e2) { return e1.salary < e2.salary; });

    std::sort(data.end() - n, data.end(),
              [](const Helper& e1, const Helper& e2) { return e1.salary > e2.salary; });

    std::vector<EmployeesDb::Iterator> employeesTopN(n);
    std::transform(data.end() - n, data.end(), employeesTopN.begin(),
                   [&db](const Helper& e) { return db.find(e.id); });
    return employeesTopN;
}
```

Employees database

Top N

```
std::vector<EmployeesDb::Iterator>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    std::nth_element(
        data.begin(), data.end() - n, data.end(),
        [](const Helper& e1, const Helper& e2) { return e1.salary < e2.salary; });

    std::sort(data.end() - n, data.end(),
              [](const Helper& e1, const Helper& e2) { return e1.salary > e2.salary; });

    std::vector<EmployeesDb::Iterator> employeesTopN(n);
    std::transform(data.end() - n, data.end(), employeesTopN.begin(),
                   [&db](const Helper& e) { return db.find(e.id); });
    return employeesTopN;
}
```

std::nth_element

4	12	9	3	7	13	5	8	4	5	11

`nth_element(begin, end - 5, end)`

5	4	4	3	5	7	8	11	9	12	13

Employees database

Top N

```
std::vector<EmployeesDb::Iterator>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r           = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    auto fSalaryComp = [](const Helper& e1, const Helper& e2) {
      return e1.salary < e2.salary;
    };

   std::nth_element(data.begin(), data.end() - n, data.end(), fSalaryComp);

    std::sort(data.end() - n, data.end(), fSalaryComp);

    std::vector<EmployeesDb::Iterator> employeesTopN(n);
    std::transform(data.rbegin(), data.rbegin() + n, employeesTopN.begin(),
            [&db](const Helper& e) { return db.find(e.id); });
    return employeesTopN;
}
```

Employees database

Top N

```
std::vector<EmployeesDb::Iterator>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    auto fSalaryComp = [](const Helper& e1, const Helper& e2) {
      return e1.salary < e2.salary;
    };

   std::nth_element(data.begin(), data.end() - n, data.end(), fSalaryComp);

    std::sort(data.end() - n, data.end(), fSalaryComp);

    std::vector<EmployeesDb::Iterator> employeesTopN(n);
    std::transform(data.rbegin(), data.rbegin() + n, employeesTopN.begin(),
             [&db](const Helper& e) { return db.find(e.id); });
    return employeesTopN;
}
```

std::nth_element

4	12	9	3	7	13	5	8	4	5	11

`nth_element(begin, end - 5, end)`

5	4	4	3	5	7	8	11	9	12	13

`sort(end - 5, end)`

5	4	4	3	5	7	8	9	11	12	13

Employees database

Top N

```
std::vector<EmployeesRecord>
topNSalariesPerPosition(const EmployeesDb& db, Profession profession, int n)
{
    auto r             = range(db, profession);
    auto noOfEmployees = std::distance(r.first, r.second);

    struct Helper
    {
        uint64_t id;
        int salary;
    };

    std::vector<Helper> data(noOfEmployees);
    std::transform(r.first, r.second, data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.id, e.salary};
    });

    auto fSalaryComp = [](const Helper& e1, const Helper& e2) {
      return e1.salary < e2.salary;
    };

   std::nth_element(data.begin(), data.end() - n, data.end(), fSalaryComp);

    std::sort(data.end() - n, data.end(), fSalaryComp);

    std::vector<EmployeesRecord> employeesTopN(n);
    std::transform(data.rbegin(), data.rbegin() + n, employeesTopN.begin(),
            [&db](const Helper& e) { return *db.find(e.id); });
    return employeesTopN;
}
```

Employees database

Average

```
int
avgSalaryPerAgeRange(const EmployeesDb& db, std::pair<int, int> ageRange)
{
    struct Helper
    {
        int age;
        int salary;
    };

    std::vector<Helper> data(db.size());
    std::transform(db.begin(), db.end(), data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.age, e.salary};
    });

    auto it = std::partition(data.begin(), data.end(), [&ageRange](const Helper& e) {
        return e.age >= ageRange.first && e.age <= ageRange.second;
    });

    auto totalSalary =
        std::accumulate(data.begin(), it, uint64_t{0},
                        [](uint64_t s, const Helper& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(data.begin(), it);
    return totalSalary / noOfEmployees;
}
```

Employees database

Average

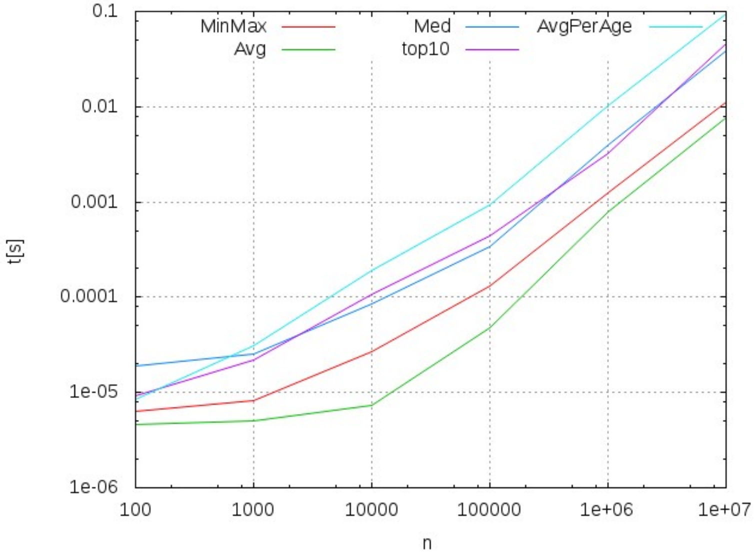
```
int
avgSalaryPerAgeRange(const EmployeesDb& db, std::pair<int, int> ageRange)
{
    struct Helper
    {
        int age;
        int salary;
    };

    std::vector<Helper> data(db.size());
    std::transform(db.begin(), db.end(), data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.age, e.salary};
    });

    auto it = std::partition(data.begin(), data.end(), [&ageRange](const Helper& e) {
        return e.age >= ageRange.first && e.age <= ageRange.second;
    });

    auto totalSalary =
        std::accumulate(data.begin(), it, uint64_t{0},
                        [](uint64_t s, const Helper& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(data.begin(), it);
    return totalSalary / noOfEmployees;
}
```

Employees database

Average

```
int
avgSalaryPerAgeRange(const EmployeesDb& db, std::pair<int, int> ageRange)
{
    struct Helper
    {
        int age;
        int salary;
    };

    std::vector<Helper> data(db.size());
    std::transform(db.begin(), db.end(), data.begin(), [](const EmployeeRecord& e) {
        return Helper{e.age, e.salary};
    });

    auto it = std::partition(data.begin(), data.end(), [&ageRange](const Helper& e) {
        return e.age >= ageRange.first && e.age <= ageRange.second;
    });

    auto totalSalary =
        std::accumulate(data.begin(), it, uint64_t{0},
                        [](uint64_t s, const Helper& e) { return s + e.salary; });
    auto noOfEmployees = std::distance(data.begin(), it);
    return totalSalary / noOfEmployees;
}
```

Algorithm efficiency



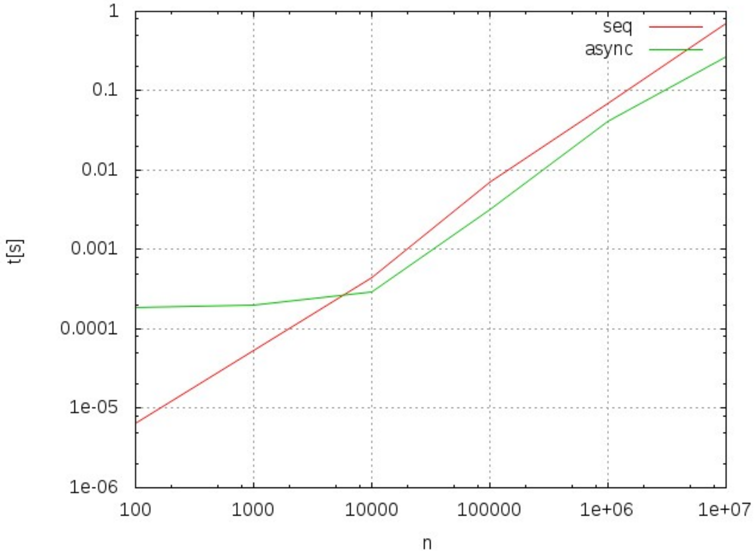
Parallel computing

```
EmployeesDb db(emps);

auto s1 = avgSalaryPerAgeRange(db, {31, 35});
auto s2 = avgSalaryPerAgeRange(db, {36, 40});
auto s3 = avgSalaryPerAgeRange(db, {41, 45});
auto s4 = avgSalaryPerAgeRange(db, {46, 50});
auto s5 = avgSalaryPerAgeRange(db, {51, 55});
auto s6 = avgSalaryPerAgeRange(db, {56, 60});
```

Parallel computing

```
EmployeesDb db(emps);

auto s1 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(31, 35));
auto s2 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(36, 40));
  auto s3 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(41, 45));
  auto s4 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(46, 50));
  auto s5 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(51, 55));
  auto s6 = std::async(std::launch::async, avgSalaryPerAgeRange, std::cref(db),
                     std::make_pair(56, 60));

  s1.wait();
  s2.wait();
  s3.wait();
  s4.wait();
  s5.wait();
  s6.wait();
```

Parallel computing



Conditional compilation.

Motivation

- Quite often implementation has to consider platform specific details or idiosyncrasies
- The implementation then combines code which is platform agnostic and platform specific
- It is still common practice to pollute common implementation with platform specific details by using C-style `#ifdef` macros
- That leads to so called "spaghetti code" which becomes hard to maintain

```
void f(const X* x, const Y* y)
{
    //common code

#if defined(PRODUCT_A)
    // Product A code
#elif defined(PRODUCT_B)
    // Product B code
#elif defined(PRODUCT_C)
    // Product C code
#endif

    // more common code
}
```

```
void g(const X* x)
{
#if defined(PRODUCT_A)
    // Product A code
#elif defined(PRODUCT_B)
    // Product B code
#elif defined(PRODUCT_C)
    // Product C code
#endif
}

void f(const X* x, const Y* y)
{
    //common code
    g(x);
    //more common code
}
```

```
void g_a(const X* x){//Product A code}
void g_b(const X* x){//Product B code}
void g_c(const X* x){//Product C code}

void g(const X* x)
{
#if defined(PRODUCT_A)
    g_a(x);
#elif defined(PRODUCT_B)
    g_b(x);
#elif defined(PRODUCT_C)
    g_c(x);
#endif
}

void f(const X* x, const Y* y)
{
    //common code
    g(x);
    //more common code
}
```

```cpp
struct Product_A {};
struct Product_B {};
struct Product_C {};

void g(const X* x, Product_A){//Product A code}
void g(const X* x, Product_B){//Product B code}
void g(const X* x, Product_C){//Product C code}

void g(const X* x)
{
#if defined(PRODUCT_A)
    g(x, Product_A{});
#elif defined(PRODUCT_B)
    g(x, Product_B{});
#elif defined(PRODUCT_C)
    g(x, Product_C{});
#endif
}

void f(const X* x, const Y* y)
{ ...  }
```

```cpp
struct Product_A {};
struct Product_B {};
struct Product_C {};

#if defined(PRODUCT_A)
    using Product = Product_A;
#elif defined(PRODUCT_B)
    using Product = Product_B;
#elif defined(PRODUCT_C)
    using Product = Product_C;
#endif

void g(const X* x, Product_A){//Product A code}
void g(const X* x, Product_B){//Product B code}
void g(const X* x, Product_C){//Product C code}

void g(const X* x)
{
    g(x, Product{});
}

void f(const X* x, const Y* y) { ... }
```

std::enable_if

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T;}

template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

std::enable_if

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T;}

template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

Allows to write specialized overloads of the same function for different types based on the type's traits. Very useful tool for generic programming for e.g. when optimizing functions based on types.

Utilizes the *SFINAE* rule - *Substitution Failure Is Not An Error*

```
template<class InputIt, class OutputIt>
void copy_n(InputIt first, size_t n, OutputIt dest_first)
{
    copy_n_impl(first, last, dest_first);
}
```

```cpp
template<class InputIt, class OutputIt>
void copy_n(InputIt first, size_t n, OutputIt dest_first)
{
    copy_n_impl(first, last, dest_first);
}
```

```cpp
struct DataWithString { std::string data; };
struct DataWithRawString { char data[20]; };

std::vector<DataWithString> src, dst;
copy_n(src, src.size(), dst);

std::vector<DataWithRawString> src, dst;
copy_n(src, src.size(), dst);
```

```cpp
template<class InputIt, class OutputIt>
void copy_n(InputIt first, size_t n, OutputIt dest_first)
{
    copy_n_impl(first, last, dest_first);
}


template<class InputIt, class OutputIt>
void copy_n_impl(InputIt first, size_t n, OutputIt dest_first,
          std::enable_if_t<std::is_pod<
            std::iterator_traits<InputIt>::value_type>::value>* = nullptr)
{
    memcpy(dest_first, first, n);
}

template<class InputIt, class OutputIt>
void copy_n_impl(InputIt first, size_t n, OutputIt dest_first,
          std::enable_if_t<!std::is_pod<
            std::iterator_traits<InputIt>::value_type>::value>* = nullptr)
{
    for (size_t i = 0; i < n; i++)
        *dest_first = *first;
}
```

```cpp
struct DataWithString { std::string data; };
struct DataWithRawString { char data[20]; };

std::vector<DataWithString> src, dst;
copy_n(src, src.size(), dst); // non-POD

std::vector<DataWithRawString> src, dst;
copy_n(src, src.size(), dst); // POD - memcpy will be used
```

```cpp
struct Product_A {};
struct Product_B {};
struct Product_C {};

#if defined(PRODUCT_A)
    using Product = Product_A;
#elif defined(PRODUCT_B)
    using Product = Product_B;
#elif defined(PRODUCT_C)
    using Product = Product_C;
#endif

template <class T>
void g(const X* x, std::enable_if_t<std::is_same<T, Product_A>::value>* = nullptr)
{//Product A code}
template <class T>
void g(const X* x, std::enable_if_t<std::is_same<T, Product_B>::value>* = nullptr)
{//Product B code}
template <class T>
void g(const X* x, std::enable_if_t<std::is_same<T, Product_C>::value>* = nullptr)
{//Product C code}

void g(const X* x)
{
    g<Product>(x);
}
```

```
struct Product_A {}; // arm
struct Product_B {}; // ppc
struct Product_C {}; // ppc

#if defined(PRODUCT_A)
    using Product = Product_A;
#elif defined(PRODUCT_B)
    using Product = Product_B;
#elif defined(PRODUCT_C)
    using Product = Product_C;
#endif

template <typename T>
struct is_arm
{
    static constexpr bool value = std::is_same<T, Product_A>::value;
};

template <typename T>
struct is_ppc
{
    static constexpr bool value = std::is_same<T, Product_B>::value
                                  std::is_same<T, Product_C>::value;
};

template <class T>
void g(const X* x, std::enable_if_t<is_arm<T>::value>* = nullptr)
{//Product A code}
template <class T>
void g(const X* x, std::enable_if_t<is_ppc<T>::value>* = nullptr)
{//Product B & C code}

void g(const X* x)
{
    g<Product>(x);
}
```

```cpp
struct ArmBased {};
struct PpcBased {};
struct Product_A : ArmBased {}; // arm
struct Product_B : PpcBased {}; // ppc
struct Product_C : PpcBased {}; // ppc

#if defined(PRODUCT_A)
    using Product = Product_A;
#elif defined(PRODUCT_B)
    using Product = Product_B;
#elif defined(PRODUCT_C)
    using Product = Product_C;
#endif

template <typename T>
struct is_arm : std::is_base<ArmBased, T> {};

template <typename T>
struct is_ppc : std::is_base<PpcBased, T> {};

template <class T>
void g(const X* x, std::enable_if_t<is_arm<T>::value>* = nullptr)
{//Product A code}
template <class T>
void g(const X* x, std::enable_if_t<is_ppc<T>::value>* = nullptr)
{//Product B & C code}

void g(const X* x)
{
    g<Product>(x);
}
```

Thank you.