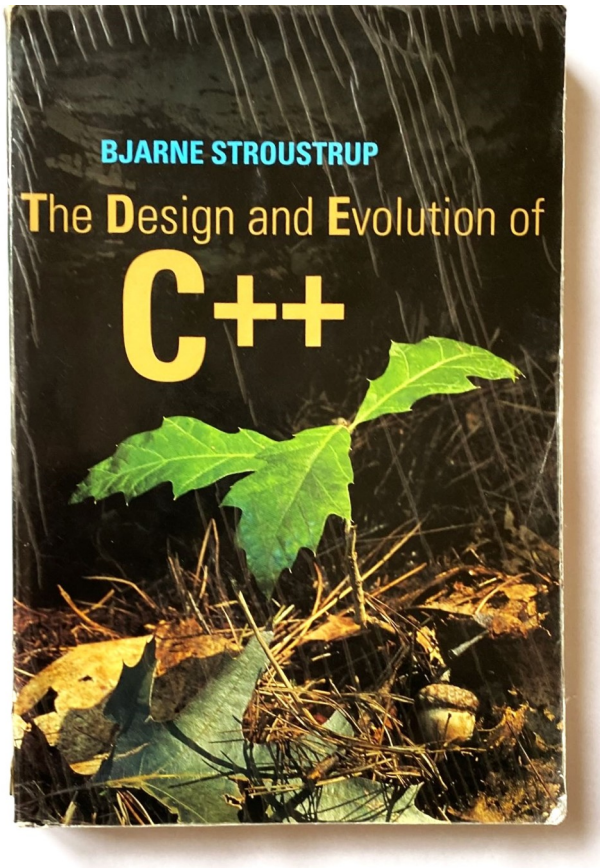
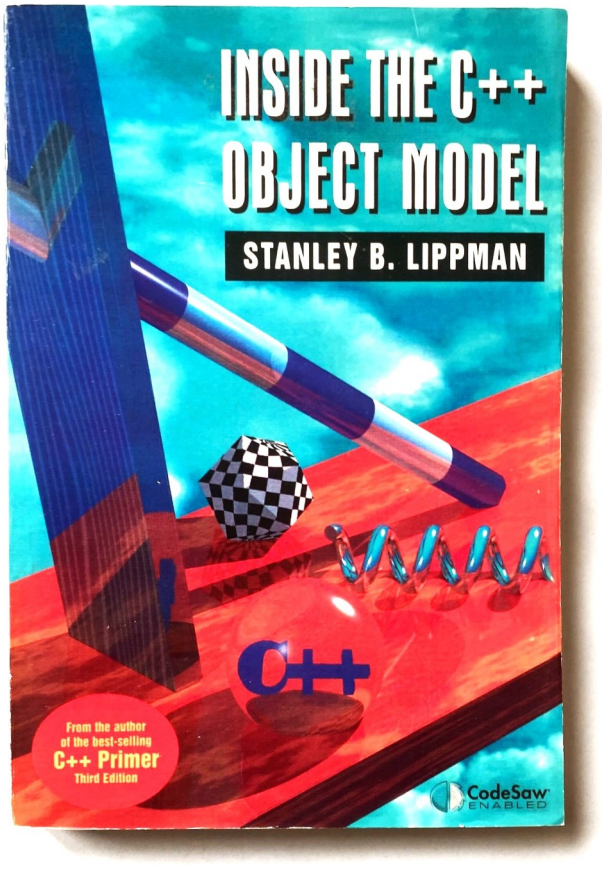
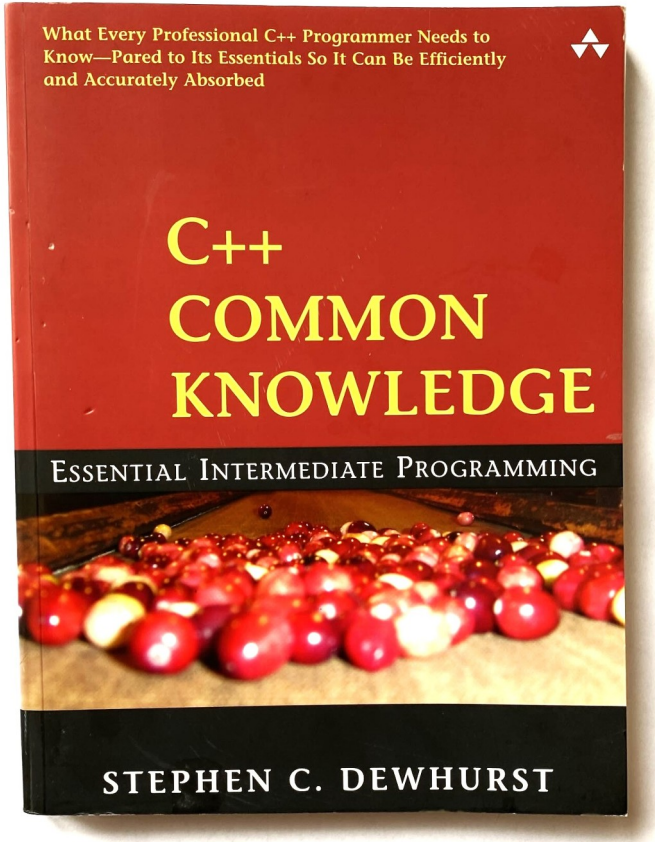
C++ Common Knowledge

Dawid Zalewski

zaldawid@gmail.com
github.com/zaldawid
linkedin.com/in/dawidzalewski


BJARNE STROUSTRUP
The Design and Evolution of
C++


INSIDE THE C++
OBJECT MODEL
STANLEY B. LIPPMAN
C++
From the author of the best-selling C++ Primer Third Edition
CodeSaw ENABLED


What Every Professional C++ Programmer Needs to Know—Pared to Its Essentials So It Can Be Efficiently and Accurately Absorbed
C++
COMMON
KNOWLEDGE
ESSENTIAL INTERMEDIATE PROGRAMMING
STEPHEN C. DEWHURST

Spot the errors (1/3)

```cpp
struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
  operator named_object() const { return { name }; }
};

struct named_object {
  std::string name;
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};

bool operator==(const named_object& a, const named_object& b) {
  return !(a != b);
}
```

Spot the errors (2/3)

```
auto kCap = 16zu;

template <typename T> struct named_heap : named_collection_base {

  named_heap(std::string name) :
    capacity{ kCap },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] } {
      named_collection_base::name = std::move(name);
    }

  ~named_heap() noexcept {
    delete[] data_min;
    delete[] data_max;
  }

  T *data_max, *data_min;
  std::size_t capacity;
};
```

+ copy ctor
+ copy assignment
+ move ctor
+ move assignment

Spot the errors (3/3)

```cpp
void needs_a_collection(std::unique_ptr<named_collection_base> ptr);

/* ~~~ */

std::unique_ptr<named_collection_base> heap{new named_heap<double>{"heap"}};

auto copy = std::make_unique<named_collection_base>(*heap);

if (*copy == *heap) {
  needs_a_collection(std::move(copy));
}
```

https://bit.ly/mcpp23

Spot the errors

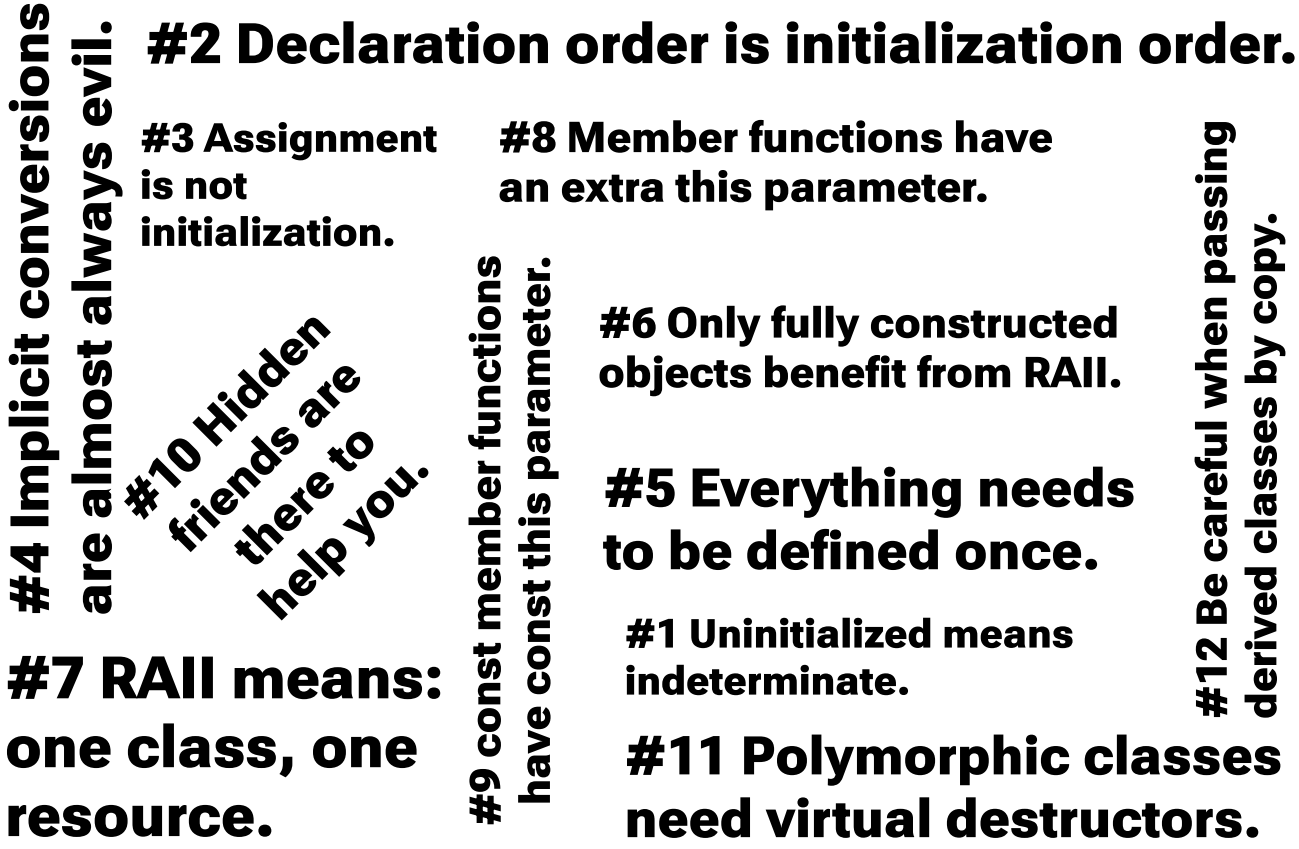
- `g++` & `clang++` (`-Wall -Wextra -pedantic -std=c++20`)
  - 2 warnings
- `msvc` (`/W4 /permissive- /std:c++20`)
  - 1 warning
- Common knowledge:
  - **10+ violations that will cause problems**

https://bit.ly/mcpp23

Spot the errors

- Not initializing data members
- Reading uninitialized data (UB)
- Allowing implicit conversions
- Confusing assignment with initialization
- Violating *one definition rule*
- Neglecting RAII & exception safety
- Leaking memory
- Ignoring const rules
- Triggering infinite recursion
- Slicing objects
- Making *name lookup* cry

C++ common knowledge



Item #1

Uninitialized means indeterminate

Uninitialized means indeterminate

```
struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
};

named_collection_base ncb{};

std::cout << ncb.name;  // OK

std::cout << ncb.size;  // UB
```

size has an indeterminate value here

Uninitialized means indeterminate

```
struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
};

named_collection_base ncb{};

std::cout << ncb.name;  // OK

std::cout << ncb.size;  // OK
```

The braces!

Uninitialized means indeterminate

```
struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
};

named_collection_base ncb;

std::cout << ncb.name;  // OK

std::cout << ncb.size;  // UB again :(
```

No braces after ncb

Uninitialized means indeterminate

```
struct named_collection_base {
  std::string name{};
  std::size_t size{};
  named_collection_base() {}
};

named_collection_base ncb;

std::cout << ncb.name;  // OK

std::cout << ncb.size;  // OK
```

} default, in-class member initializers

Item #2

Declaration order is initialization order

Declaration order == initialization order

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap() :
    capacity{ 16zu },                    ③
    data_min{ new T[capacity] },         ①
    data_max{ new T[capacity] }          ②
  {  }


  T *data_min;                           ①
  T *data_max;                           ②
  std::size_t capacity;                  ③
};
```

Declaration order == initialization order

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap() :
    capacity{ 16zu },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
  {  }

  T *data_min;
  T *data_max;
  std::size_t capacity;
};
```

capacity has an indeterminate value here

Declaration order == initialization order

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap() :
    capacity{ 16zu },                  ①
    data_min{ new T[capacity] },       ②
    data_max{ new T[capacity] }        ③
  {  }


  std::size_t capacity;                ①
  T *data_min;                         ②
  T *data_max;                         ③
};
```

Declaration order == initialization order

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap() :

    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
  {  }


  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

Item #3

Assignment is not initialization

Assignment is not initialization

```
template <typename T>
struct named_heap : named_collection_base {

 named_heap(std::string name) :

    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   {
     named_collection_base::name = std::move(name);
   }

  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

This is not the place to initialize base classes (or data members)

Assignment is not initialization

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap(std::string name) :

    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   {
     named_collection_base::name = std::move(name);
   }

  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

All the initialization happens here already

This is not the place to initialize base classes (or data members)

Assignment is not initialization

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap(std::string name) :
    named_collection_base{},
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   {
     named_collection_base::name = std::move(name);
   }

  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

All the initialization happens here already

Assignment is not initialization

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap(std::string name) :
    named_collection_base{ std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   {

   }

  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

Item #4

Implicit conversions are almost always evil

Implicit conversion are almost always evil

```
template <typename T>
struct named_heap : named_collection_base {

  named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  /* ~~~ */
};
```

```
named_heap<double> heap = "The adventures of Alice in the Wonderland"s;
```

Implicit conversion are almost always evil

```cpp
template <typename T>
struct named_heap : named_collection_base {

  named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  /* ~~~ */
};

void crunch_the_numbers_intensively(const named_heap<double>& heap);

crunch_the_numbers_intensively("One litte, one medium and one big number"s);
```

Implicit conversion are almost always evil

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  /* ~~~ */
};

void crunch_the_numbers_intensively(const named_heap<double>& heap);

crunch_the_numbers_intensively("One litte, one medium and one big number"s);
```

Implicit conversion are almost always evil

```
std::unique_ptr<named_collection_base> heap{new named_heap<double>{"heap"}};

auto copy = std::make_unique<named_collection_base>(*heap);

if (*copy == *heap) {
  needs_a_collection(std::move(copy));
}
```

Implicit conversion are almost always evil

```
template <typename T>
struct named_heap : named_collection_base;

struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
  operator named_object() const { return { name }; }
};


bool operator==(const named_object& a, const named_object& b);
```

Implicit conversion are almost always evil

```
template <typename T>
struct named_heap : named_collection_base;

struct named_collection_base {
  std::string name;
  std::size_t size;
  named_collection_base() {}
  explicit operator named_object() const { return { name }; }
};


bool operator==(const named_object& a, const named_object& b);
```

Item #5

Everything needs to be defined once

Everything needs to be defined (just) once

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ 16zu };
  T *data_min{};
  T *data_max{};
};
```

A magic number :(

Everything needs to be defined (just) once

```cpp
auto kCap = 16zu;

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

Everything needs to be defined (just) once

```cpp
auto kCap = 16zu;

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

named_heap.h

Everything needs to be defined (just) once

named_heap.h

```
#ifndef NAMED_HEAP_H
#define NAMED_HEAP_H

auto kCap = 16zu;

#endif
```

Everything needs to be defined (just) once

number_cruncher.cpp

```
...

#include "named_heap.h"

...
...
```

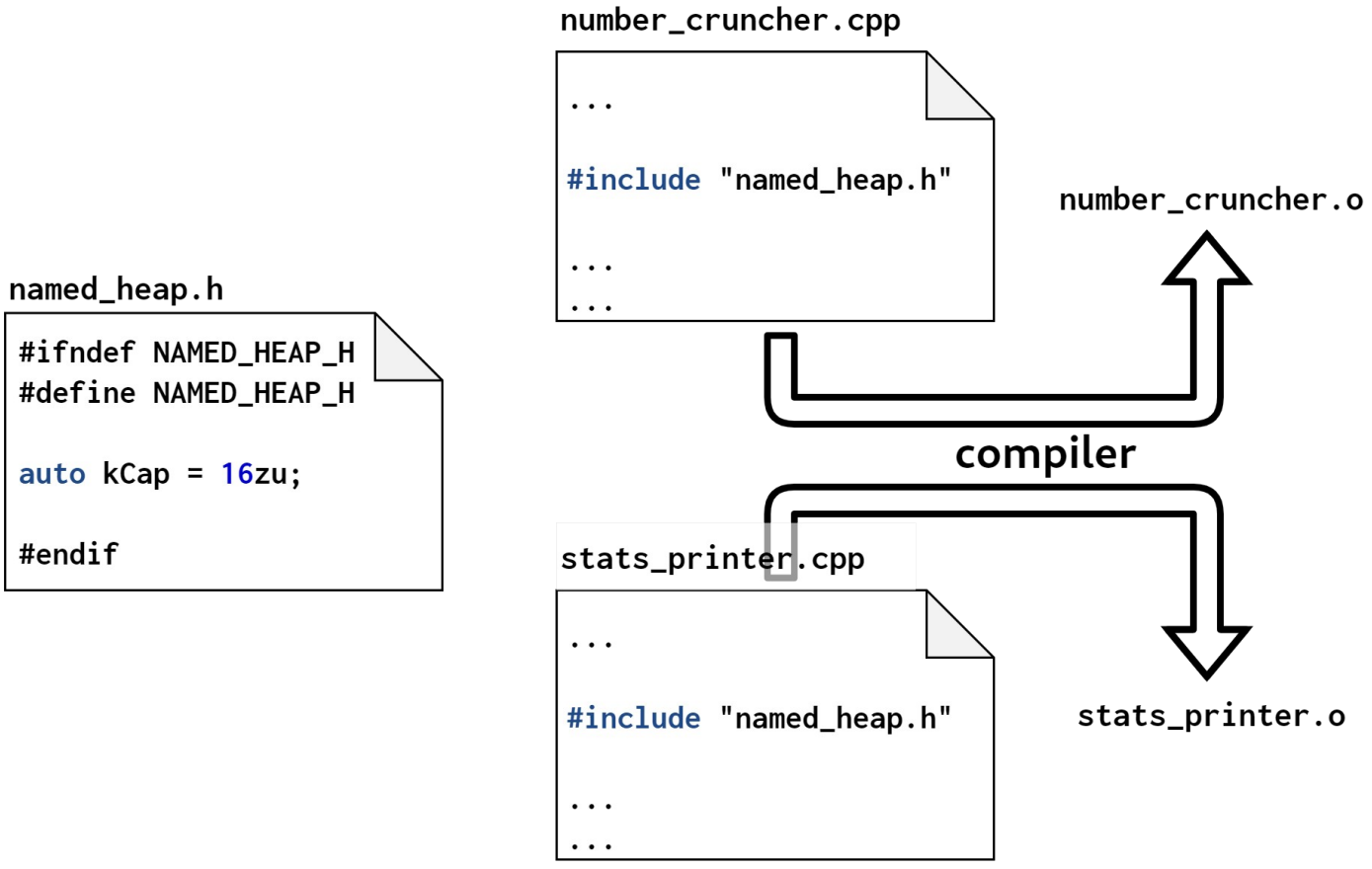
named_heap.h

```
#ifndef NAMED_HEAP_H
#define NAMED_HEAP_H

auto kCap = 16zu;

#endif
```

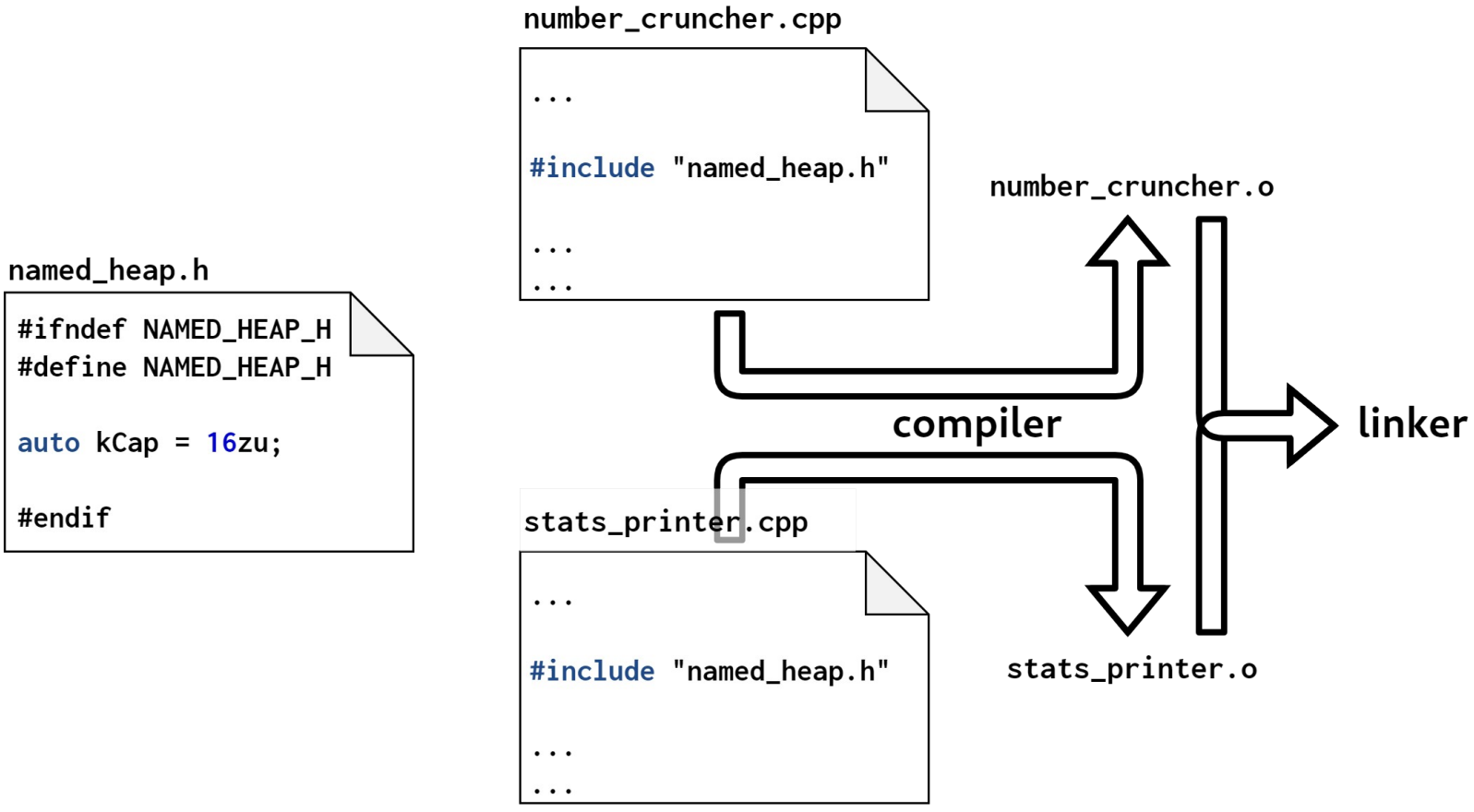
stats_printer.cpp

```
...

#include "named_heap.h"

...
...
```

Everything needs to be defined (just) once



Everything needs to be defined (just) once



One definition rule

- No translation unit shall contain more than one definition of a *definable item*.
- Every program shall contain at least one definition of every function or variable that is *odr-used* in that program [...].
- For any definable item with definitions in multiple translation units [...] if the item is a non-inline non-templated function or variable [...] the program is ill-formed.

Everything needs to be defined (just) once

```
auto kCap = 16zu;

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

named_heap.h

Everything needs to be defined (just) once

```
constexpr auto kCap = 16zu;

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

named_heap.h

Everything needs to be defined (just) once

```cpp
inline auto kCap = 16zu;

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

named_heap.h

Everything needs to be defined (just) once

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[capacity] },
    data_max{ new T[capacity] }
   { }

  constexpr static auto kCap = 16zu;

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

named_heap.h

Item #6

Only fully constructed objects benefit from RAII

Only fully constructed objects benefit from RAII

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ new T[kCap] },
    data_max{ new T[kCap] }
   { }

  ~named_heap() noexcept {
    delete[] data_min;
    delete[] data_max;
  }

  std::size_t capacity{ kCap };
  T *data_min{};
  T *data_max{};
};
```

Only fully constructed objects benefit from RAII

```
template <typename T>
struct named_heap {

  explicit named_heap(std::string) :
    data_min{ new T[kCap] },
    data_max{ new T[kCap] }
  { }

  ~named_heap() noexcept {
    delete[] data_min;
    delete[] data_max;
  }

  T *data_min{};
  T *data_max{};
};
```

The heap

43	00	c2	a1	1e	32	2a	2f
70	e3	f5	ff	45	7f	2c	b4
00	42	42	ge	7a	3c	14	1d
2a	fe	65	6a	00	00	71	2c

std:alloc_error

Does this delete[] ever get called?

Only fully constructed objects benefit from RAII

A destructor is guaranteed to be called for every fully constructed object.

Only fully constructed objects benefit from RAII

```
template <typename T>
struct named_heap {

  explicit named_heap(std::string) :
    data_min{ new T[kCap] },
    data_max{ new T[kCap] }
  { }

  ~named_heap() noexcept {
    delete[] data_min;
    delete[] data_max;
  }

  T *data_min{};
  T *data_max{};
};
```

The heap

43	00	c2	a1	1e	32	2a	2f
70	e3	f5	ff	45	7f	2c	b4
00	42	42	ge	7a	3c	14	1d
2a	fe	65	6a	00	00	71	2c

std:alloc_error

Does this delete[] ever get called?

Only fully constructed objects benefit from RAII

Function-try-block

```
template <typename T>
struct named_heap {

  explicit named_heap(std::string) try :
    data_min{ new T[kCap] },
    data_max{ new T[kCap] }
  { }
  catch(std::bad_alloc const& ex){
    delete[] data_min;
    delete[] data_max;
  }

  ~named_heap() noexcept { /*~~~*/ }

  T *data_min{};
  T *data_max{};
};
```

Does this delete[] ever get called?

Only fully constructed objects benefit from RAII

```cpp
template <typename T>
struct named_heap {

  named_heap() = default;

  explicit named_heap(std::string) :
    named_heap{}
    {
      data_min = new T[ kCap ]{};
      data_max = new T[ kCap ]{};
    }

  ~named_heap() noexcept { /* ~~~ */ }

  T *data_min{};
  T *data_max{};
};
```

Only fully constructed objects benefit from RAII

```
template <typename T>
struct named_heap {

  named_heap() = default;

  explicit named_heap(std::string) :
    named_heap{}
    {
      data_min = new T[ kCap ]{};
      data_max = new T[ kCap ]{};
    }

  ~named_heap() noexcept { /* ~~~ */ }

  T *data_min{};
  T *data_max{};
};
```

Item #7

RAII means:
one class, one resource

RAII means: one class one resource

```
template <typename T>
struct named_heap {



  explicit named_heap(std::string) :
     data_min{ new T[ kCap ]{} }
  {


  }

  ~named_heap() noexcept { /* ~~~ */ }

  T *data_min{};

};
```

RAII means: one class one resource

```
template <typename T>
struct dynamic_memory{

  dynamic_memory(std::size_t sz) :
    size{sz},
    data{new T[sz]{}}
  {};

  ~dynamic_memory() noexcept {
    delete[] data;
  }

  size_t size{};
  T *data{};
};
```

RAII means: one class one resource

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ kCap },
    data_max{ kCap }
   { }

  ~named_heap() noexcept {
  }

  std::size_t capacity{ kCap };
  dynamic_memory<T> data_min;
  dynamic_memory<T> data_max;
};
```

RAII means: one class one resource

```
template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name) :
    named_collection_base{ .name=std::move(name) },
    data_min{ std::make_unique<T[]>(capacity) },
    data_max{ std::make_unique<T[]>(capacity) }
   { }




  std::size_t capacity{ kCap };
  std::unique_ptr<T[]> data_min;
  std::unique_ptr<T[]> data_max;
};
```

Item #8

Member functions have an extra `this` parameter

Member functions have an extra this parameter

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Member functions have an extra this parameter

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

```
bool named_object::operator!=(const named_object& other) {
  return name != other.name;
}
```

```
bool named_object::operator!=(named_object * this, const named_object& other) {
  return this->name != other.name;
}
```

Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}
```

Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}
```

Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}
```

```
/* ~~~ */

named_object alice{"alice"};
named_object bob{"bob"};

if (alice == bob) { /* ~~~ */ }
```

Member functions have an extra this parameter

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};

inline bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}

/* ~~~ */

named_object alice{"alice"};
named_object bob{"bob"};

if (alice == bob) { /* ~~~ */ }
```

Member function

Free function

③

②

①

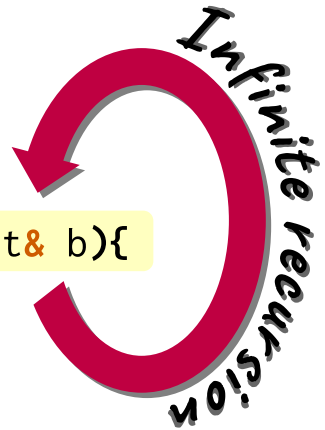
Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}
```



Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !a.operator!=( b );
}
```

Member functions have an extra this parameter

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};

inline bool operator==(const named_object& a, const named_object& b){
  return !named_object::operator!=(&a, b);
}

bool named_object::operator!=(named_object * this, const named_object& other) {
  return this->name != other.name;
}
```

Member function

Free function

const named_object* != const named_object&

Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !(a != b);
}
```

Member functions have an extra this parameter

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};
```

Free function

```
inline bool operator==(const named_object& a, const named_object& b){
  return !!(a == b);
}
```

`const named_object&` `const named_object&`

Item #9

`const` member functions have `const this` parameter

Const member functions have const this

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};

bool named_object::operator!=(named_object * this, const named_object& other) {
  return this->name != other.name;
}
```

Const member functions have const this

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) {
    return name != other.name;
  }
};

bool named_object::operator!=(const named_object * this, const named_object& other) {
  return this->name != other.name;
}
```

Const member functions have const this

Member function

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) const {
    return name != other.name;
  }
};

bool named_object::operator!=(const named_object * this, const named_object& other) {
  return this->name != other.name;
}
```

Const member functions have const this

```
struct named_object {
  std::string name{};
  bool operator!=(const named_object& other) const {   ③
    return name != other.name;
  }
};

inline bool operator==(const named_object& a, const named_object& b){   ②
  return !(a != b);
}

/* ~~~ */

named_object alice{"alice"};
named_object bob{"bob"};

if (alice == bob) { /* ~~~ */ }   ①
```

Member function

Free function

Item #10

Hidden friends are there to help you

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};
};

inline bool operator==(const named_object& a, const named_object& b){
  return a.name == b.name;
}

named_object alice{"alice"};

named_object bob{"bob"};

if (alice == bob) {
  /* ~~~ */
}
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};
};

inline bool operator==(const named_object& a, const named_object& b){
  return a.name == b.name;
}

std::unique_ptr<named_collection_base> heap{new named_heap<double>{"heap"}};

auto copy = std::make_unique<named_collection_base>(*heap);

if (*copy == *heap) {
  std::cout << "IT WORKS!!!";
}
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};
};

inline bool operator==(const named_object& a, const named_object& b){
  return a.name == b.name;
}

struct named_collection_base {
  std::string name;
  std::size_t size;
  operator named_object() const { return { name }; }
};
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};
};

inline bool operator==(const named_object& a, const named_object& b){
  return a.name == b.name;
}

struct named_collection_base {
  std::string name;
  std::size_t size;
  operator named_object() const { return { name }; }
};
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};

  friend bool operator==(const named_object& a, const named_object& b){
    return a.name == b.name;
  }
};

struct named_collection_base {
  std::string name;
  std::size_t size;
  operator named_object() const { return { name }; }
};
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};

  friend bool operator==(const named_object& a, const named_object& b){
    return a.name == b.name;
  }
};

std::unique_ptr<named_collection_base> heap{new named_heap<double>{"heap"}};

auto copy = std::make_unique<named_collection_base>(*heap);

if (*copy == *heap) {
  std::cout << "IT WONT'T EVEN COMPILE:(";
}
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};

  friend bool operator==(const named_object& a, const named_object& b){
    return a.name == b.name;
  }
};

struct named_collection_base {
  std::string name;
  std::size_t size;
  operator named_object() const { return { name }; }
};
```

Hidden friends are there to help you

```cpp
struct named_object {
  std::string name{};

  friend auto operator<=>(named_object const& a, named_object const& b) = default;


};

struct named_collection_base {
  std::string name;
  std::size_t size;
  operator named_object() const { return { name }; }
};
```

Item #11

Polymorphic classes need virtual destructors

Polymorphic classes need virtual destructors

```
std::unique_ptr<named_collection_base> ptr{new named_heap<double>{"heap"}};
/* scope ends here */
```

```
named_collection_base * ptr = new named_heap<double>{"heap"};
/* ... */
delete heap;
```

```
void needs_a_collection(std::unique_ptr<named_collection_base> ptr);
/* ... */
auto ptr = std::make_unique<named_heap<double>>("heap");

needs_a_collection(std::move(ptr));
```

Polymorphic classes need virtual destructors

```
std::unique_ptr<named_collection_base> ptr{new named_heap<double>{"heap"}};

named_collection_base * ptr = new named_heap<double>{"heap"};

void needs_a_collection(std::unique_ptr<named_collection_base> ptr);
```

```
delete ptr;   {   ptr->~named_heap();
                  ::operator delete (ptr);
```

Polymorphic classes need virtual destructors

```
struct named_collection_base {
  std::string name{};
  std::size_t size{};
};

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name);

  std::unique_ptr<T[]> data_min;
  std::unique_ptr<T[]> data_max;
};
```

Polymorphic classes need virtual destructors

```
struct named_collection_base {
  std::string name{};
  std::size_t size{};
};

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name);

  std::unique_ptr<T[]> data_min;
  std::unique_ptr<T[]> data_max;
};
```

No virtual destructor in the base class!

Polymorphic classes need virtual destructors

```cpp
std::unique_ptr<named_collection_base> ptr{new named_heap<double>{"heap"}};

named_collection_base * ptr = new named_heap<double>{"heap"};

void needs_a_collection(std::unique_ptr<named_collection_base> ptr);
```

```cpp
delete ptr;
```

```cpp
ptr->~named_collection_base();
::operator delete (ptr);
```

Polymorphic classes need virtual destructors

```cpp
struct named_collection_base {
  std::string name{};
  std::size_t size{};
  virtual ~named_collection_base() noexcept = default;
};

template <typename T>
struct named_heap : named_collection_base {

  explicit named_heap(std::string name);

  std::unique_ptr<T[]> data_min;
  std::unique_ptr<T[]> data_max;
};
```

Yep, that's all...

Polymorphic classes need virtual destructors

```
std::unique_ptr<named_collection_base> ptr{new named_heap<double>{"heap"}};

named_collection_base * ptr = new named_heap<double>{"heap"};

void needs_a_collection(std::unique_ptr<named_collection_base> ptr);

delete ptr;   {   ptr->~named_heap();
                  ::operator delete (ptr);
```

Item #12

Be careful when passing derived classes by copy (beware of slicing)

Be careful when passing derived classes by copy

```cpp
void needs_a_collection(std::unique_ptr<named_collection_base> ptr);

/* ~~~ */

auto heap = std::make_unique<named_heap<double>>("heap");

auto copy = std::make_unique<named_collection_base>(*heap);

needs_a_collection(std::move(copy));
```

```cpp
new named_collection_base( *heap );
```

Be careful when passing derived classes by copy

```
void needs_a_collection(named_collection_base copy);

/* ~~~ */

named_heap<double> heap{"heap"};

needs_a_collection(heap);
```

Be careful when passing derived classes by copy

heap

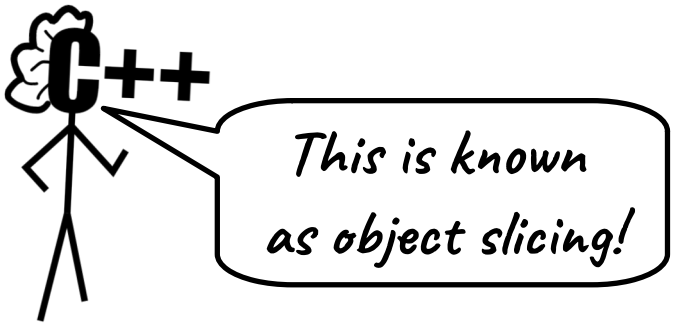
std::string name	"heap"
std::size_t size	0
std::size_t capacity	16
std::unique_ptr<T> data_min	0x559847422f20
std::unique_ptr<T> data_max	0x559847422fb0

Be careful when passing derived classes by copy

heap → copy constructor of named_collection_base → copy

std::string name	"heap"
std::size_t size	0
std::size_t capacity	16
std::unique_ptr<T> data_min	0x559847422f20
std::unique_ptr<T> data_max	0x559847422fb0

std::string name	"heap"
std::size_t size	0



Be careful when passing derived classes by copy

```cpp
void needs_a_collection(std::unique_ptr<named_collection_base> ptr);

/* ~~~ */

auto heap = std::make_unique<named_heap<double>>("heap");

auto copy = std::make_unique<named_collection_base>(*heap);

needs_a_collection(std::move(copy));
```

Be careful when passing derived classes by copy

```cpp
void needs_a_collection(std::unique_ptr<named_collection_base> ptr);

/* ~~~ */

auto heap = std::make_unique<named_heap<double>>("heap");

auto copy = std::make_unique<decltype(heap)::element_type>(*heap);

needs_a_collection(std::move(copy));
```

Be careful when passing derived classes by copy

```cpp
void needs_a_collection(const named_collection_base& coll);

/* ~~~ */

named_heap<double> heap{"heap"};

needs_a_collection(heap);
```

C++ common knowledge

#1 Uninitialized means indeterminate.

#2 Declaration order is initialization order.

#3 Assignment is not initialization.

#4 Implicit conversions are almost always evil.

#5 Everything needs to be defined once.

#6 Only fully constructed objects benefit from RAII.

#7 RAII means: one class, one resource.

#8 Member functions have an extra this parameter.

#9 const member functions have const this parameter.

#10 Hidden friends are there to help you.

#11 Polymorphic classes need virtual destructors.

#12 Be careful when passing derived classes by copy.

Thank you

zaldawid@gmail.com
github.com/zaldawid
linkedin.com/in/dawidzalewski