# C++26
# Летняя встреча ISO WG21

Полухин Антон
Эксперт разработчик C++

# Содержание

01;

# static_assert

# static_assert

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final;
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...







 private:









  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```cpp
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...

  ~FastPimpl() noexcept { // Used in `*cpp` only
    Validate<sizeof(T), alignof(T)>();
    reinterpret_cast<T*>(&storage_)->~T();
  }

 private:









  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...

  ~FastPimpl() noexcept { // Used in `*cpp` only
    Validate<sizeof(T), alignof(T)>();
    reinterpret_cast<T*>(&storage_)->~T();
  }

 private:









  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...

  ~FastPimpl() noexcept { // Used in `*cpp` only
    Validate<sizeof(T), alignof(T)>();
    reinterpret_cast<T*>(&storage_)->~T();
  }

 private:

  template <std::size_t ActualSize, std::size_t ActualAlignment>
  static void Validate() noexcept {
    static_assert(Size == ActualSize, "invalid Size: Size == sizeof(T) failed");
    static_assert(Alignment == ActualAlignment,
                  "invalid Alignment: Alignment == alignof(T) failed");
  }

  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...

  ~FastPimpl() noexcept { // Used in `*cpp` only
    Validate<sizeof(T), alignof(T)>();
    reinterpret_cast<T*>(&storage_)->~T();
  }

 private:

  template <std::size_t ActualSize, std::size_t ActualAlignment>
  static void Validate() noexcept {
    static_assert(Size == ActualSize, "invalid Size: Size == sizeof(T) failed");
    static_assert(Alignment == ActualAlignment,
                  "invalid Alignment: Alignment == alignof(T) failed");
  }

  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
 public:
  // ...

  ~FastPimpl() noexcept { // Used in `*cpp` only
    Validate<sizeof(T), alignof(T)>();
    reinterpret_cast<T*>(&storage_)->~T();
  }

 private:

  template <std::size_t ActualSize, std::size_t ActualAlignment>
  static void Validate() noexcept {
    static_assert(Size == ActualSize, "invalid Size: Size == sizeof(T) failed");
    static_assert(Alignment == ActualAlignment,
                  "invalid Alignment: Alignment == alignof(T) failed");
  }

  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
<source>: error: static assertion failed: invalid Size: Size == sizeof(T) failed

<source>: In instantiation of 'void FastPimpl<T, Size, Alignment>::validate() [with int
ActualSize = 32; int ActualAlignment = 8; T = std::string; int Size = 8; int Alignment = 8]'
```

# static_assert

```
<source>: error: static assertion failed: invalid Size: Size == sizeof(T) failed

<source>: In instantiation of 'void FastPimpl<T, Size, Alignment>::validate() [with int
ActualSize = 32; int ActualAlignment = 8; T = std::string; int Size = 8; int Alignment = 8]'
```

# static_assert

<source>: error: static assertion failed: invalid Size: Size == sizeof(T) failed

<source>: In instantiation of 'void FastPimpl<T, Size, Alignment>::validate() [with int **ActualSize = 32**; int ActualAlignment = 8; T = std::string; int Size = 8; int Alignment = 8]'

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
  // ...

 private:
  template <std::size_t ActualSize, std::size_t ActualAlignment>
  static void Validate() noexcept {
    static_assert(
        Size == ActualSize,
        fmt::format("Template argument 'Size' should be {}", ActualSize).c_str()
    );
    static_assert(
        Alignment == ActualAlignment,
        fmt::format("Template argument 'Alignment' should be {}", ActualAlignment).c_str()
    );
  }

  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
template <class T, std::size_t Size, std::size_t Alignment>
class FastPimpl final {
  // ...

 private:
  template <std::size_t ActualSize, std::size_t ActualAlignment>
  static void Validate() noexcept {
    static_assert(
        Size == ActualSize,
        fmt::format("Template argument 'Size' should be {}", ActualSize).c_str()
    );
    static_assert(
        Alignment == ActualAlignment,
        fmt::format("Template argument 'Alignment' should be {}", ActualAlignment).c_str()
    );
  }

  alignas(Alignment) std::byte storage_[Size];
};
```

# static_assert

```
<source>: error: static assertion failed: Template argument 'Size' should be 32
```

# static_assert

```
<source>: error: static assertion failed: Template argument 'Size' should be 32
```

# static_assert

```
<source>: error: static assertion failed: Template argument 'Size' should be 32
```

# 02;

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for (const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for (const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for (const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```
<source>:12:19: warning: unused variable 'x' [-Wunused-variable]
 for (const auto& x: list) {
                  ^
```

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for ([[maybe_unused]] const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for ([[maybe_unused]] const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```cpp
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for ([[maybe_unused]] const auto& x: list) {
     ++ count;
 }

 return count;
}
```

```
template <class T>
std::size_t list_count(const T& list) {
 std::size_t count = 0;

 for (const auto& _: list) {    // OK в C++26
     ++ count;
 }

 return count;
}
```

```
template <class T>
std::size_t list_count_and_insert(T& list) {
 std::size_t count = 0;

 for (const auto& _: list) {   // OK в C++26
     ++ count;
 }

 auto _ = list.insert(count);  // OK в C++26
 auto _ = list.insert(count);  // OK в C++26
 auto _ = list.insert(count);  // OK в C++26

 return count;
}
```

03;

# to_string

# to_string(floating_point)

# to_string(floating_point)

```
auto s = std::to_string(1e-7);
```

# to_string(floating_point)

```cpp
auto s = std::to_string(1e-7); // C++20: "0.000000"
```

# to_string(floating_point)

```
auto s = std::to_string(1e-7); // C++20: "0.000000" или "0,000000"
```

# to_string(floating_point)

```
auto s = std::to_string(1e-7); // C++20: "0.000000" или "0,000000"
                               // C++26: "1e-7"
```

04;

# Hazard Pointer

# Hazard Pointer

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}
```

# Hazard Pointer

```cpp
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# Hazard Pointer

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

05;

# RCU

# RCU

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it's safe
    std::rcu_barrier();     // delete the remaining
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it's safe
    std::rcu_barrier();     // delete the remaining
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it's safe
    std::rcu_barrier();     // delete the remaining
}
```

```cpp
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

```cpp
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

```
struct Data : std::hazard_pointer_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::hazard_pointer h = std::make_hazard_pointer();
    Data* p = h.protect(pdata_);
    userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    old->retire();
}
```

# RCU 2

# RCU 2

```
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}


void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}


void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}

void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2

```cpp
struct Data
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);

    std::rcu_synchronize(); // wait until it’s safe
    delete old;
}

void shutdown() {
    writer(nullptr);
}
```

```cpp
struct Data : std::rcu_obj_base<Data>
{ /* members */ };

std::atomic<Data*> pdata_;

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    Data* p = pdata_;
    if (p) userFn(p);
}

void writer(Data* newdata) {
    Data* old = pdata_.exchange(newdata);
    if (old) old->retire();
}


void shutdown() {
    writer(nullptr);
    std::rcu_synchronize(); // wait until it’s safe
    std::rcu_barrier();     // delete the remaining
}
```

# RCU 2.5

# RCU 2.5

```
struct Data { /* members */ };
struct Data2 { /* members */ };

std::atomic<Data*> pdata_;
std::atomic<Data2*> pdata2_{getData2()};
```

# RCU 2.5

```
struct Data { /* members */ };
struct Data2 { /* members */ };

std::atomic<Data*> pdata_;
std::atomic<Data2*> pdata2_{getData2()};

template <typename Func>
void reader_op(Func userFn) {
    std::scoped_lock l(std::rcu_default_domain());
    userFn(pdata1_.load(), pdata2_.load());
}
```

# 06;

# native_handle()

# native_handle()

# native_handle()

- std::basic_filebuf

# native_handle()

- std::basic_filebuf
- std::basic_ifstream

# native_handle()

- std::basic_filebuf
- std::basic_ifstream
- std::basic_ofstream

# native_handle()

- std::basic_filebuf
- std::basic_ifstream
- std::basic_ofstream
- std::basic_fstream

07;

# *function*

# *function*

# *function*

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function<bool(int)> pred);
    // ...
};
```

# *function*

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function<bool(int)> pred);
    // ...
};
```

# *function*

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function<bool(int)> pred);
    // ...
};
```

# Проблемы std::function

# Проблемы std::function

- Требует копируемости объекта

# Проблемы std::function

- Требует копируемости объекта
- Не работает с noexcept

# Проблемы std::function

- Требует копируемости объекта
- Не работает с noexcept
- Не передаётся в регистрах

# Проблемы std::function

- Требует копируемости объекта
- Не работает с noexcept
- Не передаётся в регистрах
- Сломанный const

# std::function_ref

# std::function_ref

```cpp
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function_ref<bool(int) const noexcept> pred);
    // ...
};
```

# std::function_ref

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function_ref<bool(int) const noexcept> pred);
    // ...
};
```

# std::function_ref

```cpp
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::function_ref<bool(int) const noexcept> pred);
    // ...
};
```

# std::move_only_function C++23

# std::move_only_function

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    void AsyncReadUntil(std::move_only_function<bool(int) noexcept> pred);
    // ...
};
```

# std::move_only_function

```cpp
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    void AsyncReadUntil(std::move_only_function<bool(int) noexcept> pred);
    // ...
};
```

# std::move_only_function

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    void AsyncReadUntil(std::move_only_function<bool(int) noexcept> pred);
    // ...
};
```

# std::copyable_function

# std::copyable_function

```
struct Socket {
    /// @brief Reads the stream until the predicate returns `true`.
    /// @param pred predicate that will be called for each byte read and EOF.
    std::string ReadUntil(std::copyable_function<bool(int) noexcept> pred);
    // ...
};
```

# 08;

# constexpr

# constexpr

# constexpr

- <cmath> and <complex>

# constexpr

- <cmath> and <complex>
- static_cast для void*

# constexpr

- <cmath> and <complex>
- static_cast для void*
- stable_sort, stable_partition, inplace_merge

# 09;

# std::submdspan

# std::submdspan

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                           R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                           R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {



}
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)


}
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';

}
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}
```

```
Greens by row:
2 5 8
11 14 17
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';
```

```
Greens by row:
2 5 8
11 14 17
```

```
Greens of row 1:
11 14 17
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                          R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
for(size_t i = 0; i != all_greens.extent(0); i++)
  std::cout <<  all_greens[i] << ' ';
```

# std::submdspan

```
std::vector<short> image = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
//                            R G B,R G B,R G B, R  G  B, R  G  B, R  G  B
enum Colors: unsigned { kRed, kGreen, kBlue, kTotalColors};

auto int_2d_rgb = std::mdspan(image.data(), 2, 3, (int)kTotalColors);
std::cout << "\nGreens by row:\n";
for(size_t row = 0; row != int_2d_rgb.extent(0); row++) {
  for(size_t column = 0; column != int_2d_rgb.extent(1); column++)
    std::cout << int_2d_rgb[row, column, (int)kGreen] << ' ';
  std::cout << "\n";
}

std::cout << "Greens of row 1:\n";
auto greens_of_row0 = std::submdspan(int_2d_rgb, 1, std::full_extent, (int)kGreen);
for(size_t column = 0; column != greens_of_row0.extent(0); column++)
  std::cout << greens_of_row0[column] << ' ';

std::cout << "\nAll greens:\n";
auto pixels = std::mdspan(
   int_2d_rgb.data_handle(), int_2d_rgb.extent(0) * int_2d_rgb.extent(1), (int)kTotalColors);
auto all_greens = std::submdspan(
   pixels, std::full_extent, std::integral_constant<int, (int)kGreen>{});
for(size_t i = 0; i != all_greens.extent(0); i++)
  std::cout <<  all_greens[i] << ' ';
```

```
Greens by row:
2 5 8
11 14 17
```

```
Greens of row 1:
11 14 17
```

```
All greens:
2 5 8 11 14 17
```

10;

# И ещё...

# И ещё…

# И ещё…

- Testing for success or failure of <charconv> functions

# И ещё…

- Testing for success or failure of <charconv> functions
- Hashing support for std::chrono value classes

# И ещё…

- Testing for success or failure of <charconv> functions
- Hashing support for std::chrono value classes
- Formatting pointers

# И ещё…

- Testing for success or failure of <charconv> functions
- Hashing support for std::chrono value classes
- Formatting pointers
- Heterogeneous overloads

# И ещё…

- Testing for success or failure of <charconv> functions
- Hashing support for std::chrono value classes
- Formatting pointers
- Heterogeneous overloads
- Checking if a union alternative is active

# И ещё…

- Testing for success or failure of <charconv> functions
- Hashing support for std::chrono value classes
- Formatting pointers
- Heterogeneous overloads
- Checking if a union alternative is active
- Bind front and back to NTTP callables

# Спасибо за внимание

Полухин Антон
Эксперт разработчик C++