C++11/14 Bootstrap

CppCon 2015 Pre-conference Training

Michael Caisse

michael.caisse@ciere.com

Part I

Object Enhancements

Outline

- Inheriting Constructors
- Delegating Constructors
- Explicit Conversion Operators
- Explicit Overrides and Final
- POD Reworked
- Defaulted and Deleted Member Functions

The Problem

```
struct bar
{
   bar(int i) : i_(i) {}

   int i_;
};


struct foo : bar
{
   foo(int i) : bar(i) {}
};
```

The Solution

Inheriting Constructor

```
struct bar
{
   bar(int i) : i_(i) {}

   int i_;
};


struct foo : bar
{
   using bar::bar;
};
```

ciere.com

Outline

- Inheriting Constructors
- Delegating Constructors
- Explicit Conversion Operators
- Explicit Overrides and Final
- POD Reworked
- Defaulted and Deleted Member Functions

The Problem

```
struct bar
{
   bar()         : i_(42)  {}
   bar(int i)    : i_(i)   {}

private:
   int i_;
};
```

The Problem

```
struct bar
{
   bar()
   {
     init(42);
   }

   bar(int i)
   {
     init(i);
   }

private:
   void init(int i){ i_ = i; }
   int i_;
};
```

The Problem

References must be initialized.

```
struct client
{
  client(ioservice & io)
      : io_(io)
  {}

  client(ioservice & io, std::string delim)
      : io_(io)
      , delim_sequence_(delim)
  {}


private:
   ioservice & io_;
   std::string delim_sequence_;
};
```

The Solution

Delegating Constructor

```
struct bar
{
   bar()        : bar(42)  {}
   bar(int i)   : i_(i)    {}

private:
   int i_;
};
```

The Solution

C++11 Member Initialization

```
struct bar
{
   bar()                   {}
   bar(int i) : i_(i)      {}

private:
   int i_ = 42;
};
```

The Solution

```
struct client
{
  client(ioservice & io)
      : io_(io)
  {}

  client(ioservice & io, std::string delim)
      : client(io)
      , delim_sequence_(delim)
  {}


private:
   ioservice & io_;
   std::string delim_sequence_;
};
```

Error

```
error: an initializer for a delegating constructor must appear alone
```

The Solution

```
struct client
{
  client(ioservice & io)
      : io_(io)
  {}

  client(ioservice & io, std::string delim)
      : client(io)
      , delim_sequence_(delim)
  {}


private:
   ioservice & io_;
   std::string delim_sequence_;
};
```

Error

```
error: an initializer for a delegating constructor must appear alone
```

The Solution

General rule: delegate to the constructor that takes the most arguments

```
struct client
{
  client(ioservice & io)
     : client(io, "dx078")
  {}

  client(ioservice & io, std::string delim)
     : io_(io)
     , delim_sequence_(delim)
  {}

private:
   ioservice & io_;
   std::string delim_sequence_;
};
```

Delegating Constructor Thoughts

- Solves compile-time defaults
- Initialization in constructor
- Delegation of reference initialization
- C++11 defines an object being constructed when ***any*** constructor finishes
- We now have simplified member initialization

Outline

The Problem

```
struct foo
{
    operator bool()
    {
        return true;
    }
};

int main()
{
    foo f;
    int i = f + 42;
}
```

The Problem

```
struct foo
{
    operator bool()
    {
        return true;
    }

    operator int()
    {
        return 42;
    }
};

int main()
{
    foo f;
    int i = f + 42;
}
```

Output - clang

```
error: use of overloaded operator '+' is ambiguous (with operand types 'foo' and 'int')
    int v = f + 42;
            ~ ^ ~~
```

The Problem

```
struct foo
{
    operator bool()
    {
        return true;
    }

    operator int()
    {
        return 42;
    }
};

int main()
{
    foo f;
    int i = f + 42;
}
```

Output - clang

```
error: use of overloaded operator '+' is ambiguous (with operand types 'foo' and 'int')
   int v = f + 42;
           ~ ^ ~~
```

The Solution

Explicit Conversion Operator

```
struct foo
{
    explicit operator bool()
    {
        return true;
    }

    operator int()
    {
        return 42;
    }
};

int main()
{
    foo f;
    if(f)
    {
      int i = f + 42;
    }
}
```

Outline

Overrides - The Problem

```
struct bar
{
   virtual void do_stuff(int i)
   {}
};

struct foo : bar
{
   virtual void do_stuff(float i)
   {}
};
```

Overrides - The Solution

For **`virtual`** functions.

```
struct bar
{
    virtual void do_stuff(int i)
    {}
};

struct foo : bar
{
    virtual void do_stuff(float i) override
    {}
};
```

Output - clang

```
error: 'do_stuff' marked 'override' but does not override any member functions
   virtual void do_stuff(float i) override
                ^
```

Overrides - The Solution

For **`virtual`** functions.

```
struct bar
{
   virtual void do_stuff(int i)
   {}
};

struct foo : bar
{
   virtual void do_stuff(float i) override
   {}
};
```

Output - clang

```
error: 'do_stuff' marked 'override' but does not override any member functions
   virtual void do_stuff(float i) override
                ^
```

Final - Classes

```
struct bar final
{
    virtual void do_stuff(int i)
    {}
};

struct foo : bar
{
    virtual void do_stuff(int i)
    {}
};
```

Output - clang

```
error: base 'bar' is marked 'final'
struct foo : bar
             ^
```

Final - Classes

```
struct bar final
{
   virtual void do_stuff(int i)
   {}
};

struct foo : bar
{
   virtual void do_stuff(int i)
   {}
};
```

Output - clang

```
error: base 'bar' is marked 'final'
struct foo : bar
             ^
```

Final - Methods

```
struct bar
{
   virtual void do_stuff(int i) final
   {}
};

struct foo : bar
{
   virtual void do_stuff(int i)
   {}
};
```

Output - clang

```
error: declaration of 'do_stuff' overrides a 'final' function
   virtual void do_stuff(int i)
                ^
```

Final - Methods

```
struct bar
{
   virtual void do_stuff(int i) final
   {}
};

struct foo : bar
{
   virtual void do_stuff(int i)
   {}
};
```

Output - clang

```
error: declaration of 'do_stuff' overrides a 'final' function
   virtual void do_stuff(int i)
                ^
```

Outline

- Inheriting Constructors
- Delegating Constructors
- Explicit Conversion Operators
- Explicit Overrides and Final
- POD Reworked
- Defaulted and Deleted Member Functions

What is a POD?

What is a POD?

C++03 Definition:

- Plain Old Data type
- Can be statically initialized
- Layouts are compatible with C

What is a POD?

C++03 Definition:

- Plain Old Data type
- Can be statically initialized
- Layouts are compatible with C

What is a POD?

C++03 Definition:

- Plain Old Data type
- Can be statically initialized
- Layouts are compatible with C

What is a POD?

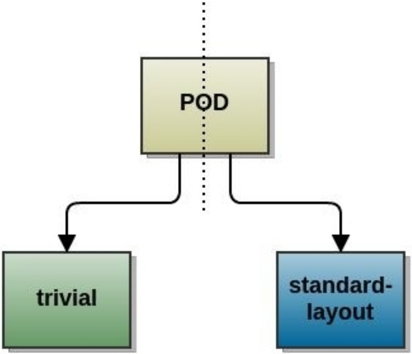
C++03 POD:

```
struct foo
{
  int i;
};
```

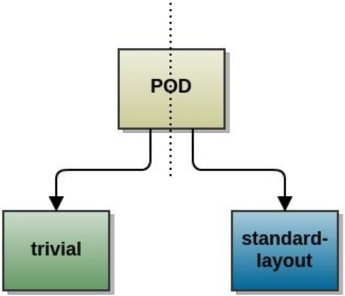
C++03 Not POD:

```
struct bar
{
  bar(int j) : i(j) {}
  int i;
};
```

What is a POD?



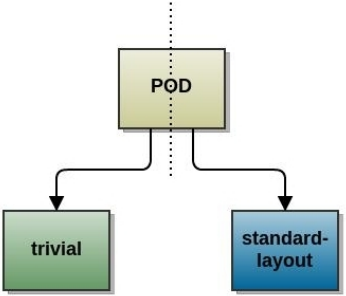
What is a POD?



- can be statically initialized
- legal to copy via `memcpy`
- lifetime starts with storage
- trivial default/copy/move ctr
- trivial copy/move assign
- trivial non-virtual destructor

- no virtual functions/bases
- non-static data have the same access control, are in the same single class
- true for bases/non-statics
- no bases of same type as first defined non-static data

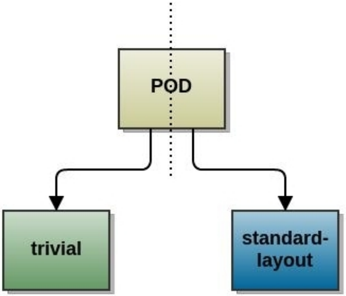
What is a POD?



- can be statically initialized
- legal to copy via **`memcpy`**
- lifetime starts with storage
- trivial default/copy/move ctr
- trivial copy/move assign
- trivial non-virtual destructor

- no virtual functions/bases
- non-static data have the same access control, are in the same single class
- true for bases/non-statics
- no bases of same type as first defined non-static data

What is a POD?



- can be statically initialized
- legal to copy via **`memcpy`**
- lifetime starts with storage
- trivial default/copy/move ctr
- trivial copy/move assign
- trivial non-virtual destructor

- no virtual functions/bases
- non-static data have the same access control, are in the same single class
- true for bases/non-statics
- no bases of same type as first defined non-static data

Outline

- Inheriting Constructors
- Delegating Constructors
- Explicit Conversion Operators
- Explicit Overrides and Final
- POD Reworked
- Defaulted and Deleted Member Functions

Special Functions

Special Member Functions:

- default constructor
- copy constructor
- copy assignment operator
- destructor

Special Functions

Global operators:

- sequence **`operator ,`**
- address-of **`operator &`**
- indirection **`operator *`**
- member access **`operator ->`**
- member indirection **`operator ->*`**
- free-store allocation **`operator new`**
- free-store deallocation **`operator delete`**

ciere.com

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

A Few Problems with Defaults

- Constructor definitions are coupled
- Default destructor is inappropriate to polymorphic classes
- Once a default is suppressed, there is no way to get it back
- Default implementation are often more efficient
- Non-default implementations are non-trivial affecting type semantics (non-POD)
- Cannot prohibit special member/global functions without declaring non-trivial substitute
- free-store deallocation `operator delete`

Compiler Diagnostics

```
struct foo
{
private:
   foo(){}
};

int main()
{
   foo f;
}
```

Output - clang

```
error: `foo::foo()` is private
```

ciere.com

Compiler Diagnostics

```
struct foo
{
private:
    foo(){}
};

int main()
{
    foo f;
}
```

Output - clang

```
error: 'foo::foo()' is private
```

ciere.com

Compiler Diagnostics

```
struct foo
{
   foo() = delete;
};

int main()
{
   foo f;
}
```

Output - clang

```
error: use of deleted function 'foo::foo()'
```

Compiler Diagnostics

```
struct foo
{
   foo() = delete;
};

int main()
{
   foo f;
}
```

Output - clang

```
error: use of deleted function 'foo::foo()'
```

ciere.com

Default Example

```
struct foo
{
    foo(){}
    foo(int i) : i_(i)
    {}

private:
    int i_;
};

int main()
{
    foo f;
}
```

Default Example

```
struct foo
{
    foo() = default;
    foo(int i) : i_(i)
    {}

private:
    int i_;
};

int main()
{
    foo f;
}
```

Another Example

```
struct gorp
{
   gorp() = default;
   virtual ~gorp();
   gorp( const gorp & );
};

inline gorp::gorp(const gorp &) = default;

gorp::~gorp() = default;
```

What does this do?

```
struct foo
{
    foo() = default;
    foo & operator=(foo const &) = delete;
    foo(foo const &) = delete;
};
```

What does this do?

```
struct foo
{
   void * operator new(std::size_t) = delete;
};
```

What does this do?

```
struct foo
{
   foo(long long);
   foo(long) = delete;
};

extern void bar(foo, long long);
void bar(foo, long) = delete;
```

ciere.com

Default Standard Notes

- indicates that the function's default definition should be used
- an inline and explicitly defaulted definition is trivial if and only if the implicit definition would have been trivial
- the explicit default has normal exception specification semantics

Delete Standard Notes

- all lookup and overload resolution occurs before the deleted definition is noted
- definition is deleted, not the symbol
- deleted definition of a function must be its first declaration
- deleted definition mechanism is orthogonal to access specifiers
- deleted functions are trivial

Part II

rvalue and move

Outline

- Move Basics

Special Member Functions

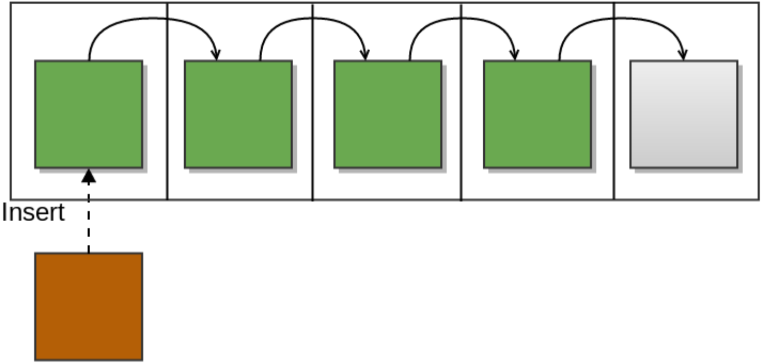
Moving on with:

- Move Constructor
- Move Assignment

Motivation for Move

```
std::string s(' ', 1000);
std::vector<std::string> v(1000,s);
v.insert(v.begin(),s);
```

Motivation for Move



Motivation for Move

This is ugly:

```
void make_circus(circular & cirque)
{
   // create a circus
}

circular cirque;
make_circus(cirque);
```

Motivation for Move

This makes sense in our value semantic language.

```
cirular make_circus()
{
   circular cirque;
   // create a circus
   return cirque;
}

circular cirque = make_circus();
```

Motivation for Move

```
my_special_type o;

// manipulate and do things with o
// ..

// store for later use
storage.push_back(o);
```

Motivation for Move

What is it about copying in the previous examples that we don't like?

We want to reuse the guts from the source object to populate the destination object.

Motivation for Move

What is it about copying in the previous examples that we don't like?

We want to reuse the guts from the source object to populate the destination object.

Motivation for Move

Optimization:

- ability to recognize the object is a temporary
- ability to indicate that the object is no longer needed ... it is expiring

Move only types:

- name some

Motivation for Move

Bind to a rvalue:

```
foo(bar && b);          // rvalue reference
```

Motivation for Move

```
foo(bar && b);          // rvalue reference
foo(bar const & b);     // lvalue reference
```

```
bar z;
foo(z);
```

Motivation for Move

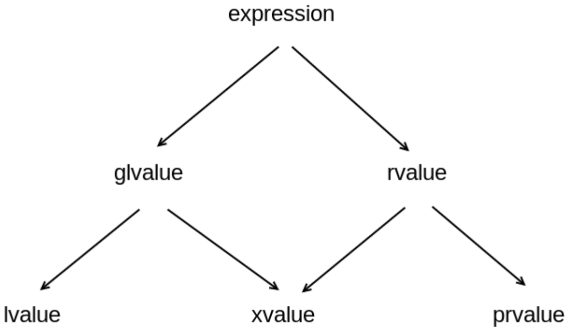
```
foo(bar && b);          // rvalue reference
foo(bar const & b);     // lvalue reference


bar make_bar()
{
   bar b;
   // ...
   return b;
}

foo(make_bar());
```

Motivation for Move

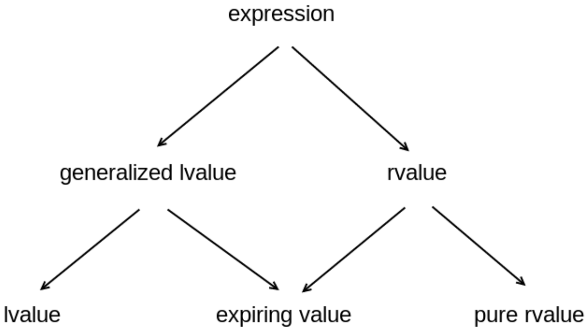
What about this?

```
my_special_type o;

// manipulate and do things with o
// ..

// store for later use
storage.push_back(o);
```

Motivation for Move



Motivation for Move



Motivation for Move

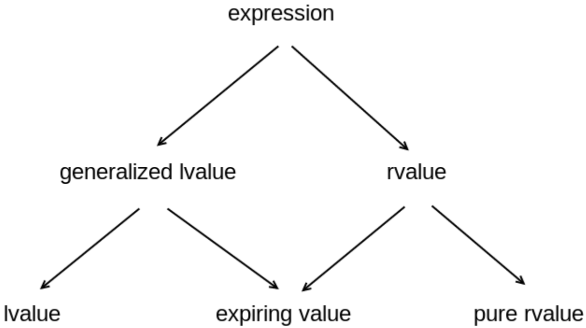
```
my_special_type o;

// manipulate and do things with o
// ..

// store for later use
storage.push_back(std::move(o));
```

What is **std::move**

```
template <class T>
inline
T&& move(T&& x)
{
    return static_cast<T&&>(x);
}
```

Motivation for Move

The **rvalue** from an **lvalue** is an **xvalue.**



ciere.com

Move Special Member functions

What do you notice about the copy declarations versus the move declarations?

```
class circular
{
public:
    circular(std::size_t i=20);
    ~circular();

    circular(circular const & rhs);
    circular & operator=(circular const & rhs);

    circular(circular && rhs);
    circular & operator=(circular && rhs);
};
```

Semantics of a Move

Ask yourself:

What does it mean for your class to be moved?

What are the post-move requirements on your class?

Semantics of a Move

Ask yourself:

What does it mean for your class to be moved?
What are the post-move requirements on your class?

circular

```
class circular
{
public:

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Constructor

```
class circular
{
public:

   circular(circular && rhs)

      : buffer(rhs.buffer)
      , head(rhs.head), tail(rhs.tail)
      , size(rhs.size)
   {
      rhs.buffer = nullptr;
   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Constructor

```
class circular
{
public:

    circular(circular && rhs)

        : buffer(rhs.buffer)
        , head(rhs.head), tail(rhs.tail)
        , size(rhs.size)
    {
        rhs.buffer = nullptr;
    }

private:
    uint8_t *buffer, *head, *tail;
    std::size_t size;
};
```

Move Assignment

```
class circular
{
public:
    circular & operator=(circular && rhs)
    {
        if(&rhs != this)
        {
            delete [] buffer;
            size = rhs.size;
            buffer = rhs.buffer;
            head = rhs.head;
            tail = rhs.tail;

            rhs.buffer = nullptr;
            rhs.head = nullptr;
            rhs.tail = nullptr;
            rhs.size = 0;
        }
        return *this;
    }
private:
    uint8_t *buffer, *head, *tail;
    std::size_t size;
};
```

Move Assignment

```
class circular
{
public:
    circular & operator=(circular && rhs)
    {
        if(&rhs != this)
        {
            delete [] buffer;
            size = rhs.size;
            buffer = rhs.buffer;
            head = rhs.head;
            tail = rhs.tail;

            rhs.buffer = nullptr;
            rhs.head = nullptr;
            rhs.tail = nullptr;
            rhs.size = 0;
        }
        return *this;
    }
private:
    uint8_t *buffer, *head, *tail;
    std::size_t size;
};
```

Move Assignment

```
class circular
{
public:

   circular & operator=(circular && rhs)
   {

      if(&rhs != this)
      {
         using std::swap;
         swap(buffer,rhs.buffer);
         swap(head,rhs.head);
         swap(tail,rhs.tail);
         swap(size,rhs.size);
      }
      return *this;

   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Assignment

```
class circular
{
public:
   circular & operator=(circular && rhs)
   {
      using std::swap;
      swap(buffer,rhs.buffer);
      swap(head,rhs.head);
      swap(tail,rhs.tail);
      swap(size,rhs.size);

      return *this;
   }
private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Instrumented

```
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}
```

```
{
   std::cout << "-> start 1" << std::endl;
   circular a;
   a = amazing_stuff();
   std::cout << "<- end 1" << std::endl;
}
```

Move Instrumented

```
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}
```

```
{
   std::cout << "-> start 1" << std::endl;
   circular a;
   a = amazing_stuff();
   std::cout << "<- end 1" << std::endl;
}
```

Move Intrumented

```
-> start 1
circular default constructor
circular default constructor
circular move assign
circular destructor
<- end 1
circular destructor
```

```
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}

{
   std::cout << "-> start 1" << std::endl;
   circular a;
   a = amazing_stuff();
   std::cout << "<- end 1" << std::endl;
}
```

Move Instrumented

```
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}
```

```
{
   std::cout << "-> start 2" << std::endl;
   circular a = amazing_stuff();
   std::cout << "<- end 2" << std::endl;
}
```

Move Instrumented

```
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}
```

```
{
   std::cout << "-> start 2" << std::endl;
   circular a = amazing_stuff();
   std::cout << "<- end 2" << std::endl;
}
```

Move Instrumented

```
-> start 2
circular default constructor
<- end 2
circular destructor
```

```cpp
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}


{
   std::cout << "-> start 2" << std::endl;
   circular a = amazing_stuff();
   std::cout << "<- end 2" << std::endl;
}
```

Standard 12.8 [31]

"When certain criteria are met, an **implementation** is **allowed** to **omit** the **copy/move construction** of a class object, **even** if the **copy/move constructor and/or destructor** for the object have **side effects.**

In such cases, the **implementation treats** the **source** and **target** of the omitted copy/move operation as simply **two** different **ways** of **referring to the same object ...**"

Standard 12.8 [31]

"When certain criteria are met, an **implementation** is **allowed** to **omit** the **copy/move construction** of a class object, **even** if the **copy/move constructor and/or destructor** for the object have **side effects.**

In such cases, the **implementation treats** the **source** and **target** of the omitted copy/move operation as simply **two** different **ways** of **referring to the same object ...**"

RVO / NRVO

What costs less than a move?

RVO / NRVO

Don’t break
Return **V**alue **O**ptimization
or
Named **R**eturn **V**alue **O**ptimization.

Breaking RVO

```
circular amazing_broken_stuff()
{
   circular circus;
   // ...
   return std::move(circus);
}

{
   std::cout << "-> start 3" << std::endl;
   circular a = amazing_broken_stuff();
   std::cout << "<- end 3" << std::endl;
}
```

Breaking RVO

```
circular amazing_broken_stuff()
{
   circular circus;
   // ...
   return std::move(circus);
}

{
   std::cout << "-> start 3" << std::endl;
   circular a = amazing_broken_stuff();
   std::cout << "<- end 3" << std::endl;
}
```

Breaking RVO

```
-> start 3
circular default constructor
circular move constructor
circular destructor
<- end 3
circular destructor
```

```cpp
circular amazing_broken_stuff()
{
   circular circus;
   // ...
   return std::move(circus);
}

{
   std::cout << "-> start 3" << std::endl;
   circular a = amazing_broken_stuff();
   std::cout << "<- end 3" << std::endl;
}
```

Breaking RVO

```
circular broken_choice_stuff()
{
   circular soleil;
   circular ringling;
   bool wants_animals = false;
   // ...
   return wants_animals ? ringling : soleil;
}
```

Breaking RVO

```
circular amazing_conversion_stuff()
{
    // ...
    return 42;
}
```

Move is Assign?

Move our `circular` type with the array.

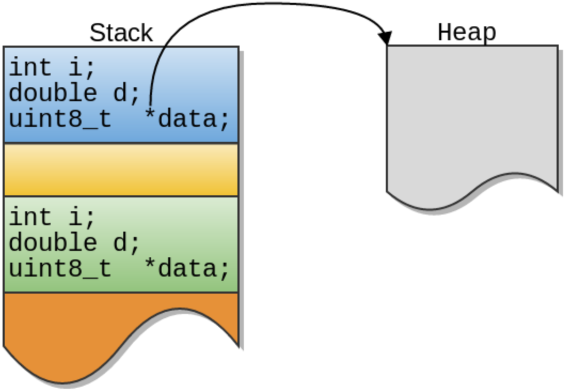
```
class circular
{
private:
   uint8_t buffer[10];
   uint8_t *head, *tail;
};
```

Move is not Magic

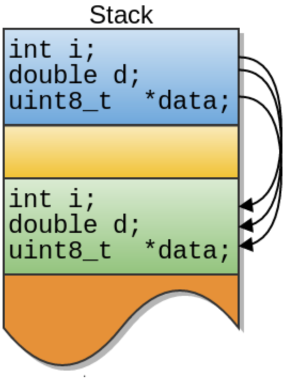
Move our `circular` type with the array.

```
class circular
{

private:
    uint8_t buffer[10];
    uint8_t *head, *tail;
};
```

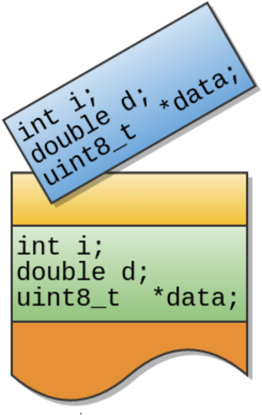
Move is not Magic



Move is not Magic



Move is not Magic



Move is not Magic

Move is copy if there are no externally managed resources.

`noexcept`

Some containers have a *strong exception guarantee* for certain operations.

For example: `std::vector::insert`

`noexcept`

Some containers have a *strong exception guarantee* for certain operations.

For example: **`std::vector::insert`**

noexcept

```
circular(circular && rhs)  noexcept
   : buffer(rhs.buffer)
   , head(rhs.head), tail(rhs.tail)
   , size(rhs.size)
{
   rhs.buffer = nullptr;
}
```

noexcept

```
circular & operator=(circular && rhs) noexcept
{
   using std::swap;
   swap(buffer,rhs.buffer);
   swap(head,rhs.head);
   swap(tail,rhs.tail);
   swap(size,rhs.size);

   return *this;
}
```

noexcept

```
template <class T>
void swap(T & a, T & b)
noexcept( std::is_nothrow_move_constructible<T>::value &&
          std::is_nothrow_move_assignable<T>::value)
{
   T tmp(std::move(a));
   a = std::move(b);
   b = std::move(tmp);
}
```

swap

```
friend void swap(circular & a, circular & b)
{
   using std::swap;
   //...
}
```

Part III

Uniform Initializers

Outline

- Initializer List
- Uniform Initialization

Initializing Arrays

C++03 and C for PODs

```
struct point
{
  int x;
  int y;
};

point my_point = {8,-4};
point path[]   = { {8,-4}, {1,2}, {42,19}, {7,12} };
```

Initializing std Containers

C++11 introduces the `std::initializer_list<T>`

```
struct point
{
  int x;
  int y;
};

std::vector<point> path = { {8,-4}, {1,2}, {42,19}, {7,12} };

std::vector<std::string> names = {"Sam", "Mary",
                                   "Rick", "Ella" };
```

ciere.com

Initializer List

`std::initializer_list<T>:`

Method	Description
`size`	number of elements
`begin`	iterator to first element
`end`	end iterator

Initializer List Exercise

Write a data type `storage` that uses a `std::vector` internally and provides the following:

- constructed with an initializer_list
- has an `add_some` method that takes an initializer_list argument and appends new values to the back of the vector.

Initializer List Exercise

Write a data type `storage` that uses a `std::vector` internally and provides the following:

- constructed with an initializer_list
- has an `add_some` method that takes an initializer_list argument and appends new values to the back of the vector.

```
#include <vector>

class storage
{
public:
    storage(???)
    {}

    void add_some(????)
    {}

private:
    std::vector vec_
};
```

Initializer List Exercise - Solution

```
#include <vector>

template<typename T>
class storage
{
public:
    storage(std::initializer_list<T> i)
        : vec_(i)
    {}

    void add_some(std::initializer_list<T> i)
    {
        vec_.insert(vec_.end(), i.begin(), i.end());
    }

private:
    std::vector<T> vec_
};
```

ciere.com

Outline

- Initializer List
- Uniform Initialization

Uniform Initializing

```
struct point
{
  int x, y;
};

struct private_point
{
  private_point(int x, int y) : x_(x), y_(y)
  {}

  point as_point() const  { return{ x_, y_ }; }

private:
  int x_, y_;
};

point my_point{8,-4};
private_point your_point{8,-4};
std::vector<point> path{ {8,-4}, {1,2}, {42,19}, {7,12} };
```

Uniform Initializing

```
struct point
{
  int x = 0;
  int y = 0;
};

point p = {42,8};
```

Produces an error in C++11.

Valid in C++14.

Uniform Initializing

Eliminates the most vexing parse.

```
class time;

schedule sched(time());
```

Uniform Initializing

Eliminates the most vexing parse.

```
class time;

schedule sched(time{});
```

ciere.com

Part IV

Variety Time

Outline

- Integer Types
- Right Angle Brackets
- nullptr
- constexpr

Integer types

C99 Adoptions

- `long long int` is supported. At least as large as `long int` with a minumum of 64-bits.

Integer types

include <cstdint>

Exact widths.

- `int8_t`, `int16_t`, `int32_t`, `int64_t`
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

ciere.com

Integer types

include <cstdint>

Smallest integer types with at least the specified width.

- `int_least8_t`, `int_least16_t`, `int_least32_t`, `int_least64_t`
- `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t`

Integer types

include <cstdint>

Fastest integer types with at least the specified width.

- `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `int_fast64_t`
- `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t`

ciere.com

Integer types

include <cstdint>

- `intmax_t`, `uintmax_t`
- `intptr_t`, `uintptr_t`

Outline

- Integer Types
- Right Angle Brackets
- nullptr
- constexpr

Right Angle Parsing Woes

```
std::map<std::string,std::vector<int> > foo;
```

Right Angle Parsing Woes

```
std::map<std::string,std::vector<int>> foo;
```

ciere.com

Outline

- Integer Types
- Right Angle Brackets
- nullptr
- constexpr

ciere.com

nullptr

C++ defines 0 as being both an integer constant and a null pointer constant.

```
foo(int)    {...}
foo(char*)  {...}
...
foo(0);
foo((char*)0);
```

nullptr

```
std::string foo(false);    // calls char* version w/ null
std::string foo(true);     // error
```

nullptr

`nullptr` is an added keyword.

- `nullptr_t` defines the type, it is a typedef
- `nullptr_t` is a POD type that is convertible to both a pointer type and a pointer-to-member type.
- All objects of type `nullptr_t` are equivalent
- An object of type `nullptr_t` can be converted to any pointer or pointer-to-member type.

ciere.com

nullptr

`nullptr` is an added keyword.

- `nullptr_t` defines the type, it is a typedef
- `nullptr_t` is a POD type that is convertible to both a pointer type and a pointer-to-member type.
- All objects of type `nullptr_t` are equivalent
- An object of type `nullptr_t` can be converted to any pointer or pointer-to-member type.

nullptr

- It cannot be converted to any other type (including any integral or bool type)
- It cannot be used in an arithmetic expression
- It cannot be assigned to an integral value
- It cannot be compared to an integral value

nullptr

```
foo(int)   {...}
foo(char*) {...}
...
foo(0);
foo(nullptr);
```

nullptr

```
template<typename T>
void bar( T * t );

bar( nullptr );     // error, no deduction to a pointer type
bar( (float*) nullptr );  // deduces T = float
```

Outline

- Integer Types
- Right Angle Brackets
- nullptr
- constexpr

constexp support

gcc	Clang	MSVC	Intel C++
4.6	3.1	VS 2015 RTM	13

constexp motivation

```
int special_number(){ return 42; }

float some_array[ special_number() * 2 ];
```

constexp motivation

```
constexpr int special_number(){ return 42; }

float some_array[ special_number() * 2 ];
```

What can be a constant expressions

Definition of a:

- object
- function
- function template

or Declaration of

- static data member of literal type

constexp examples

```
constexpr int square(int x) // OK
{
  return x * x;
}

constexpr int bufsz = 1024; // OK

constexpr struct pixel {    // error: pixel is a type
  int x;
  int y;
};

int next(constexpr int x)   // error, function parameter
{ return x + 1; }

extern constexpr int memsz; // error: not a definition
```

ciere.com

constexpr functions

- its return type shall be a literal type
- its parameter types shall be literal types
- its function-body shall be a compound-statement of the form `{ return expression; }` where expression is a potential constant expression – *C++14 removes restriction*
- every implicit conversion used in converting expression to the function return type shall be one of those allowed in a constant expression

ciere.com

constexpr function examples

```
constexpr int square(int x)
{ return x * x; }                 // OK

constexpr long long_max()
{ return 2147483647; }            // OK

constexpr int abs(int x)
{ return x < 0 ? -x : x; }        // OK

constexpr void f(int x)           // error: return type is void
{ /* ... */ }

constexpr int prev(int x)
{ return --x; }                   // error: use of decrement

constexpr int g(int x, int n) // error in c++11: body not
{                                 // just ''return expr''
  int r = 1;
  while (--n > 0) r *= x;
  return r;
}
```

more constexpr examples

```
struct length
{
  explicit constexpr length(int i = 0) : val(i) { }
private:
  int val;
};


struct pixel
{
  int x;
  int y;
};
constexpr pixel ur = { 1294, 1024 };

constexpr double g = 9.8;
```

`constexpr` Exercise

Using **`constexpr`** write a method that calculates the factorial of 8 and 42 and prints the result.

constexpr function examples

```
#include <iostream>

constexpr long long fact(unsigned j)
{
   return j == 0 ? 1 : j*fact(j-1);
}

const long long fact_8  = fact(8);
const long long fact_42 = fact(42);

int main()
{
   std::cout << "fact 8  : " << fact_8 << std::endl;
   std::cout << "fact 42 : " << fact_42 << std::endl;
}
```

Part V

auto and decltype

Outline

- auto
- decltype
- Alternative Function Syntax
- Function Return Type Deduction - C++14

Motivation

```
std::map<int,string>::mapped_type v = my_map[key];

std::vector<int>::const_iterator iter = vec_.cbegin();

??? f = std::bind( &foo, 42 );
```

Motivation

```
auto v = my_map[key];

auto iter = vec_.cbegin();

auto f = std::bind( &foo, 42 );
```

Value Semantics

The default is value semantics for type deduction:

```
bar & foo();

auto   v = foo();      // v is a bar
auto & n = foo();      // n is a bar reference
```

ciere.com

Where can it be used?

Anywhere the type can be deduced and without error...

```
auto x = 1, *y = &x;

auto * x = new auto(1);

auto g(9.8);

int foo();
auto x1 = foo();
const auto & x2 = foo();
auto & x3 = foo();      // error : cannot bind a ref to a temp
```

Outline

- auto
- decltype
- Alternative Function Syntax
- Function Return Type Deduction - C++14

Some examples

```
float b = 42.1234;
decltype(b) a = 12.34;

const std::vector<int> v(1);
auto a = v[0];          // a has type int
decltype(v[1]) b = 1; //b has type const int&
//
//   std::vector<int>::operator[](size_type) const
```

Some examples

```
int foo(double);
decltype(foo(42.1)) j;

template <typename T>
struct bar
{
  T stuff;
};

auto * p = new bar<float>;
decltype(p->stuff) g;
```

decltype(auto)

In C++14 use `decltype(auto)` to use the `decltype` rules for an `auto` declaration.

```
const std::vector<int> v(1);
auto a = v[0];            // a has type int
decltype(auto) b = v[0]; // b has type const int&
```

Outline

- auto
- decltype
- Alternative Function Syntax
- Function Return Type Deduction - C++14

Motivation

```
int         foo(int i)    { return 42; }
std::string foo(double d) { return "wow"; }
```

```
template<typename T>
decltype(foo(t)) bar(T t)
{
  return foo(t);
}
```

```
std::cout << "bar(5): " << bar(5) << std::endl;
std::cout << "bar(5.1): " << bar(5.1) << std::endl;
```

Output - clang

```
          decl_example.cpp:10:14: error: use of undeclared identifier 't'
decltype(foo(t)) bar(T t) return foo(t);
             ^
```

Motivation

```
int         foo(int i)    { return 42; }
std::string foo(double d) { return "wow"; }


template<typename T>
decltype(foo(t)) bar(T t)
{
   return foo(t);
}


 std::cout << "bar(5): " << bar(5) << std::endl;
 std::cout << "bar(5.1): " << bar(5.1) << std::endl;
```

Output - clang

```
          decl_example.cpp:10:14: error: use of undeclared identifier 't'
decltype(foo(t)) bar(T t) return foo(t);
             ^
```

Motivation

```
int         foo(int i)    { return 42; }
std::string foo(double d) { return "wow"; }


template<typename T>
decltype(foo(T())) bar(T t)
{
   return foo(t);
}


 std::cout << "bar(5): " << bar(5) << std::endl;
 std::cout << "bar(5.1): " << bar(5.1) << std::endl;
```

Motivation

```
int          foo(int i)    { return 42; }
std::string foo(double d) { return "wow"; }


template<typename T>
auto bar(T t) -> decltype(foo(t))
{
   return foo(t);
}


std::cout << "bar(5): " << bar(5) << std::endl;
std::cout << "bar(5.1): " << bar(5.1) << std::endl;
```

Example

```
auto amazing_func() -> int
{
  return 42;
}

constexpr auto more_amazing_func() -> int
{
  return 42;
}
```

Outline

- auto
- decltype
- Alternative Function Syntax
- Function Return Type Deduction - C++14

Example

```
auto amazing_func()
{
  return 42;
}
```

Example

Problem. Return type can't be deduced before the recursive call.

```
auto factorial(long n)
{
   if(n > 1)
      return factorial(n-1)*n;
   else
      return 1;
}
```

Example

This works:

```
auto factorial(long n)
{
    if(n == 1)
        return 1;
    else
        return factorial(n-1)*n;
}
```

ciere.com

Part VI

Range-based `for` loop

Outline

- Examples
- Exercise

Standard Example

```
int array[5] = { 1, 2, 3, 4, 5 };
for(int & x : array)
    x *= 2;
```

```
std::vector<float> vec = { 1.1, 2.2, 3.3, 4.4, 5.5 };
for(auto & x : vec)
    x *= 2;
```

```
for(auto x : { 1.1, 2.2, 3.3, 4.4, 5.5 } )
    std::cout << x*2 << ",";
```

Standard Example

```
int array[5] = { 1, 2, 3, 4, 5 };
for(int & x : array)
    x *= 2;

std::vector<float> vec = { 1.1, 2.2, 3.3, 4.4, 5.5 };
for(auto & x : vec)
    x *= 2;

for(auto x : { 1.1, 2.2, 3.3, 4.4, 5.5 } )
    std::cout << x*2 << ",";
```

ciere.com

Standard Example

```
int array[5] = { 1, 2, 3, 4, 5 };
for(int & x : array)
    x *= 2;


std::vector<float> vec = { 1.1, 2.2, 3.3, 4.4, 5.5 };
for(auto & x : vec)
    x *= 2;


for(auto x : { 1.1, 2.2, 3.3, 4.4, 5.5 } )
    std::cout << x*2 << ",";
```

Compatible with...

- C-style arrays
- Initializer List
- Any type that has a `begin()` and `end()` returning iterators

Compatible with...

- C-style arrays
- Initializer List
- Any type that has a `begin()` and `end()` returning iterators

Compatible with...

- C-style arrays
- Initializer List
- Any type that has a `begin()` and `end()` returning iterators

Outline

- Examples
- Exercise

Exercise

Make `bar::amazing_type` compatible with range-based for.

`http://ciere.com/cpp11/range_exercise1.cpp`

Exercise - Solution

```
template <typename T>
class amazing_type
{
public:
   typedef typename std::vector<T>::iterator iterator;
   typedef typename std::vector<T>::const_iterator const_iterator;

   amazing_type() = default;
   amazing_type(amazing_type const &) = delete;

   amazing_type(std::initializer_list<T> init)
      : vec_(init)
   {}

   const_iterator first_one() const   { return vec_.begin(); }
   const_iterator ending()    const   { return vec_.end();   }

   iterator       first_one()         { return vec_.begin(); }
   iterator       ending()            { return vec_.end();   }

private:
   std::vector<T> vec_;
};
```

Exercise - Solution

```
template <typename T>
auto begin(amazing_type<T> & a) -> decltype(a.first_one())
{
   return a.first_one();
}

template <typename T>
auto end(amazing_type<T> & a) -> decltype(a.ending())
{
   return a.ending();
}
```

Part VII

Function

Outline

- Overview
- function
- Exercise
- Summary

Overview

`include <functional>`

- Function object wrappers for deferred calls
- Can store and invoke functions or function objects
- Uses Small Buffer Optimization (SBO)

Overview

```
include <functional>
```

- Function object wrappers for deferred calls
- Can store and invoke functions or function objects
- Uses Small Buffer Optimization (SBO)

Overview

```
include <functional>
```

- Function object wrappers for deferred calls
- Can store and invoke functions or function objects
- Uses Small Buffer Optimization (SBO)

Outline

- Overview
- function
- Exercise
- Summary

function - old school - free function

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

void old_school()
{
    int(*callback)(int,int) = &sum;

    int result = callback(5,3);
    cout << "sum is: " << result << endl;
}
```

ciere.com

function - old school - functor

```
struct adder
{
    int operator()(int a, int b){ return a+b; }
};


void functor()
{
    adder callback = adder{};

    int result = callback(5,3);
    cout << "sum is: " << result << endl;
}
```

function - old school - member function

```
struct alu
{
   int add(int a, int b){ return a+b; }
};


void member_function()
{
   int (alu::*callback)(int,int) = &alu::add;

   alu my_alu;
   int result = (my_alu.*callback)(5,3);
   cout << "sum is: " << result << endl;

   alu *alu_ptr = new alu;
   result = (alu_ptr->*callback)(5,3);
   cout << "sum is: " << result << endl;
}
```

function - function declarator-based syntax

```
std::function< R(A1,A2,A3...)  > callback;
```

function - function declarator-based syntax

```
std::function< R(A1,A2,A3...)  > callback;
```

function declarator syntax

function - function declarator-based syntax

```
std::function< R(A1,A2,A3...)  > callback;
```

return value

function - function declarator-based syntax

```
std::function< R(A1,A2,A3...)  > callback;
```

arguments

function - function declarator-based syntax

```
std::function< int(int,int) > callback;
```

function - with std::function

```
int result;
std::function< int(int,int) > callback;

// free function
callback = &sum;
result = callback(5,3);

// functor
callback = adder();
result = callback(5,3);

// member function
std::function< int(alu*,int,int) > callback2;
callback2 = &alu::add;

alu my_alu;
result = callback2(&my_alu,5,3);

alu *alu_ptr = new alu;
result = callback2(alu_ptr,5,3);
```

storage and calling conventions

Type	Old School Define	std::function
Free	`int(*callback)(int,int)`	`function< int(int,int) >`
Functor	`object_t callback`	`function< int(int,int) >`
Member	`int (object_t::*callback)(int,int)`	`function< int(object_t*,int,int) >`

Type	Old School Calling	std::function
Free	`callback(5,3)`	`callback(5,3)`
Functor	`callback(5,3)`	`callback(5,3)`
Member	`(object.*callback)(5,3)`	`callback(&object,5,3)`
Member	`(object_ptr->*(5,3)`	`callback(object_ptr,5,3)`

ciere.com

function - stateful functors

```
struct sum
{
    sum() : value_(0) {}
    int operator()(int a){ return value_ += a; }
    int value_;
};


sum my_sum;
std::function< int(int) > sum1 = my_sum;
std::function< int(int) > sum2 = my_sum;

int s1 = sum1(10);
int s2 = sum2(20);

cout << "after 1: " << s1 << endl;
cout << "after 2: " << s2 << endl;
```

function - stateful functors

```
after 1:  10
after 2:  20
```

```
struct sum
{
    sum() : value_(0) {}
    int operator()(int a){ return value_ += a; }
    int value_;
};


sum my_sum;
std::function< int(int) > sum1 = my_sum;
std::function< int(int) > sum2 = my_sum;

int s1 = sum1(10);
int s2 = sum2(20);

cout << "after 1: " << s1 << endl;
cout << "after 2: " << s2 << endl;
```

function - stateful functors

```
struct sum
{
    sum() : value_(0) {}
    int operator()(int a){ return value_ += a; }
    int value_;
};


sum my_sum;
std::function< int(int) > sum1 = std::ref(my_sum);
std::function< int(int) > sum2 = std::ref(my_sum);

int s1 = sum1(10);
int s2 = sum2(20);

cout << "after 1: " << s1 << endl;
cout << "after 2: " << s2 << endl;
```

function - stateful functors

```
after 1:  10
after 2:  30
```

```
struct sum
{
    sum() : value_(0) {}
    int operator()(int a){ return value_ += a; }
    int value_;
};


sum my_sum;
std::function< int(int) > sum1 = std::ref(my_sum);
std::function< int(int) > sum2 = std::ref(my_sum);

int s1 = sum1(10);
int s2 = sum2(20);

cout << "after 1: " << s1 << endl;
cout << "after 2: " << s2 << endl;
```

function - test for validity

```
std::function< void(std::string) > paint;
paint("green");
```

Throws `bad_function_call`

function - test for validity

```
std::function< void(std::string) > paint;

if(paint)
{
    paint("green");
}
```

Outline

- Overview
- function
- Exercise
- Summary

function - Exercise!

Exercise 1

`http://ciere.com/cpp11/function.cpp`

Exercise - Solution

```
struct worker
{
   template< typename T >
   void queue(T func, int value)
   {
      queue_.push_back(std::make_pair(func,value));
   }

   int go(int initial)
   {
      int value = initial;

      for(auto & v : queue_)
      {
         value = (v.first)(value, v.second);
      }

      return value;
   }

   using callback_t = std::function<int(int,int)>;
   using queue_t = std::vector< std::pair<callback_t, int> >;
   queue_t queue_;
};
```

Exercise - Solution

```
int main()
{
    worker my_worker;

    // queue the work here
    my_worker.queue( &sum        , 5 );
    my_worker.queue( &sub        , 15 );
    my_worker.queue( divide{}    , 3 );
    my_worker.queue( &sub        , 10 );
    my_worker.queue( multiply{}, 4 );

    int result = my_worker.go(100);
    std::cout << "result: " << result << std::endl;

    return 1;
}
```

Outline

Summary

- Allows arbitrary compatible function objects to be targets (instead of requiring an exact function signature)
- May be used with argument-binding and other function object construction libraries
- It has predictible behavior when an empty function object is called.

Part VIII

Bind

Outline

- Overview
- bind
- Exercise
- Summary

Overview

`include <functional>`

- Generalization of `std::bind1st` and `std::bind2nd`
- Supports function objects, functions, function pointers, and member function pointers
- Bind arguments to values or placeholders

ciere.com

Overview

`include <functional>`

- Generalization of `std::bind1st` and `std::bind2nd`
- Supports function objects, functions, function pointers, and member function pointers
- Bind arguments to values or placeholders

Overview

`include <functional>`

- Generalization of `std::bind1st` and `std::bind2nd`
- Supports function objects, functions, function pointers, and member function pointers
- Bind arguments to values or placeholders

Outline

- Overview
- bind
- Exercise
- Summary

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )
```

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )
```

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )
```

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )
```

Creating Functors

Bind creates **functors** that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )
```

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )();
```

Creating Functors

Bind creates functors that *bind* the callee and arguments.

```
int sub( int a, int b ){return a-b; }

bind( sub, 7, 2 )();

sub( 7, 2 );
```

Storing Bound Objects

```
int sub( int a, int b ){ return a-b; }

std::function<int()> f = bind( sub, 7, 2 );

f();
```

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int)> f = bind( sub, _1, 2 );

f(7);
```

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int)> f = bind( sub, 7, _1 );

f(2);
```

ciere.com

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int,int)> f = bind( sub, _2, _1 );

f(7,2);
```

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int,int)> f = bind( sub, _2, _1 );

f(7,2);
```

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int,int)> f = bind( sub, _2, _1 );

f(7,2);
```

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int,int)> f = bind( sub, 29, 9 );

f(7,2);
```

ciere.com

bind - Placeholders

```
int sub( int a, int b ){ return a-b; }

std::function<int(int,int)> f = bind( sub, 29, 9 );

f(7,2);
```

Result : 20

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, _1, _2 )( 10, 5 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }


int result = bind( divide, _1, _2 )( 10, 5 );
cout << "result: " << result << endl;


result:  2
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }


std::function<int(int,int)> func = bind( divide, _1, _2 );
int result = func( 10, 5 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

std::function<int(int,int)> func = bind( divide, _1, _2 );
int result = func( 10, 5 );
cout << "result: " << result << endl;
```

```
result:  2
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, _2, _1 )( 10, 5 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, _2, _1 )( 10, 5 );
cout << "result: " << result << endl;

result:  0
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, _1, 5 )( 10 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }


int result = bind( divide, _1, 5 )( 10 );
cout << "result: " << result << endl;


result:  2
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }


int result = bind( divide, 10, _1 )( 5 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, 10, _1 )( 5 );
cout << "result: " << result << endl;

result:  2
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, 20, _5 )( 1, 5, 9, 8, 2, 10 );
cout << "result: " << result << endl;
```

bind - quiz

```
int divide( int x, int y ){ return x / y; }

int result = bind( divide, 20, _5 )( 1, 5, 9, 8, 2, 10 );
cout << "result: " << result << endl;

result:  10
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }


int x = 5;
std::function<int()> func = bind( add, 20, x );
int result = func();

cout << "result: " << result << endl;
```

```
result:  25
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }

int x = 5;
std::function<int()> func = bind( add, 20, x );
int result = func();

cout << "result: " << result << endl;
```

```
result:  25
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }


int x = 5;
std::function<int()> func = bind( add, 20, x );

x = 10;
int result = func();

cout << "result: " << result << endl;


result:  25
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }


int x = 5;
std::function<int()> func = bind( add, 20, x );

x = 10;
int result = func();

cout << "result: " << result << endl;
```

```
result:  25
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }


int x = 5;
function<int()> func = bind( add, 20, std::ref(x) );

x = 10;
int result = func();

cout << "result: " << result << endl;
```

bind quiz - Continued

```
int add( int x, int y ){ return x + y; }


int x = 5;
function<int()> func = bind( add, 20, std::ref(x) );

x = 10;
int result = func();

cout << "result: " << result << endl;


result:  30
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


adder my_adder;
function<int(int,int)> func = bind( &adder::add
                                  , my_adder
                                  , _1, _2 );

int result = func( 16, 8 );
cout << "result: " << result << endl;
cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  0
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


adder my_adder;
function<int(int,int)> func = bind( &adder::add
                                  , my_adder
                                  , _1, _2 );

int result = func( 16, 8 );
cout << "result: " << result << endl;
cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  0
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


adder my_adder;
function<int(int,int)> func = bind( &adder::add
                                  , std::ref(my_adder)
                                  , _1, _2 );

int result = func( 16, 8 );
cout << "result: " << result << endl;
cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  24
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


adder my_adder;
function<int(int,int)> func = bind( &adder::add
                                  , std::ref(my_adder)
                                  , _1, _2 );

int result = func( 16, 8 );
cout << "result: " << result << endl;
cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  24
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


 adder my_adder;
 function<int(int,int)> func = bind( &adder::add
                                   , &my_adder
                                   , _1, _2 );

 int result = func( 16, 8 );
 cout << "result: " << result << endl;
 cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  24
```

bind quiz - Continued

```
struct adder
{
   adder() : last_(0) {}

   int add(int x, int y){ last_ = x + y; return last_; }

   int last_;
};


 adder my_adder;
 function<int(int,int)> func = bind( &adder::add
                                   , &my_adder
                                   , _1, _2 );

 int result = func( 16, 8 );
 cout << "result: " << result << endl;
 cout << "last: " << my_adder.last_ << endl;
```

```
result:  24
my_adder.last:  24
```

Binding Member Functions

```
struct alu
{
   int add( int a, int b ){ return a+b; }
   int sub( int a, int b ){ return a-b; }
   int multiple( int a, int b ){ return a*b; }
   int divide( int a, int b ){ return a/b; }
};
```

```
alu alu_;

bind( &alu::add, alu_, 7, 5 )();
bind( &alu::sub, &alu_, _1, 5 )(29);
bind( &alu::divide, ref(alu_), 42, _4 )(21, 7, 3, 6, 9);
```

Binding Functors

```
struct add
{
   int operator()( int a, int b ){ return a+b; }
};
```

```
bind( add(), 7, 5 )();
bind<int>( add(), 7, 5 )();
bind<double>( add(), 7, 5 )();
```

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

ciere.com

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

ciere.com

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

ciere.com

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

bind - composition

We can next `bind` calls to compose higher level functionality.

```
int sub( int a, int b ){ return a-b; }

void compose()
{
  int values[] = {5, 3, 8, 1, 9};
  std::for_each( values, values+5,
                 bind<int>(printf,"%d ", bind(sub,_1,3) ));
}
```

```
---------
    2 0 5 -2 6
```

ciere.com

bind with shared_ptr - the blender

```
struct blender
{
   blender(){ cout << "create blender" << endl; }
   ~blender(){ cout << "destroy blender" << endl; }

   void power( bool on )
   {
      cout << "turn on : " << on << endl;
   }

   void speed( int percent )
   {
      cout << "set speed : " << percent << endl;
   }
};
```

bind with shared_ptr - the worker

```
struct worker
{
   template< typename T >
   void add( T work )
   {
      work_queue.push_back( work );
   }

   void do_work()
   {
      while( !work_queue.empty() )
      {
         work_queue.front()();
         work_queue.pop_front();
      }
   }

   std::deque< std::function<void()> > work_queue;
};
```

ciere.com

bind with shared_ptr - put it together

```
worker worker_;

cout << "<---- enter scope -----" << endl;
{
   worker_.add( bind<int>(printf,"start blending\n") );

   shared_ptr<blender> blender_( new blender );

   worker_.add( bind( &blender::power, blender_, true ) );
   worker_.add( bind( &blender::speed, blender_, 25 )   );
   worker_.add( bind( &blender::speed, blender_, 80 )   );
   worker_.add( bind( &blender::speed, blender_, 35 )   );
   worker_.add( bind( &blender::power, blender_, false ));

   worker_.add( bind<int>(printf,"end blending\n") );
}
cout << "----- exit scope ---->" << endl;

worker_.do_work();
```

bind with `shared_ptr` - put it together

```
worker worker_;

cout << "<---- enter scope -----" << endl;
{
   worker_.add( bind<int>(printf,"start blending\n") );

   shared_ptr<blender> blender_( new blender );

   worker_.add( bind( &blender::power, blender_, true ) );
   worker_.add( bind( &blender::speed, blender_, 25 )   );
   worker_.add( bind( &blender::speed, blender_, 80 )   );
   worker_.add( bind( &blender::speed, blender_, 35 )   );
   worker_.add( bind( &blender::power, blender_, false ));

   worker_.add( bind<int>(printf,"end blending\n") );
}
cout << "----- exit scope ---->" << endl;

worker_.do_work();
```

```
<--- enter scope ---
create blender
--- exit scope --->
start blending
turn on : 1
set speed : 25
set speed : 80
set speed : 35
turn on : 0
destroy blender
end blending
```

Outline

- Overview
- bind
- Exercise
- Summary

bind - Exercise 1

Exercise 1

Convert `std::function` Exercise 1 so it doesn't require a seperate storage mechanism for the second argument.

bind - Exercise 3

Exercise 3

Using the alu, compose with std::bind a functor that will take the sequence of integers and

accumulate (value * 2) - 3

```
x = 0;
x += (value * 2) - 3;
```

Print the answer.

Outline

- Overview
- bind
- Exercise
- Summary

ciere.com

Summary

- Employ `std::bind` and `std::function` for powerful/flexible callbacks
- Use bind to ....

ciere.com

Part IX

Lambda Expressions

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Some Motivation - Old School

```
vector<int>::const_iterator iter     = cardinal.begin();
vector<int>::const_iterator iter_end = cardinal.end();

int total_elements = 1;
while( iter != iter_end )
{
   total_elements *= *iter;
   ++iter;
}
```

ciere.com

Some Motivation - Functor

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          product<int>(total_elements) );
```

```
template <typename T>
struct product
{
   product( T & storage ) : value(storage) {}

   template< typename V>
   void operator()( V & v )
   {
      value *= v;
   }

   T & value;
};
```

Some Motivation - Functor

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          product<int>(total_elements) );


template <typename T>
struct product
{
   product( T & storage ) : value(storage) {}

   template< typename V>
   void operator()( V & v )
   {
      value *= v;
   }

   T & value;
};
```

ciere.com

Some Motivation - Phoenix

```
// Boost.Phoenix
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          phx::ref(total_elements) *= _1 );
```

Some Motivation - Lambda Expression

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&total_elements](int i){total_elements *= i;} );
```

Some Motivation - Lambda Expression

Before:

```
vector<int>::const_iterator iter     = cardinal.begin();
vector<int>::const_iterator iter_end = cardinal.end();

int total_elements = 1;
while( iter != iter_end )
{
   total_elements *= *iter;
   ++iter;
}
```

After:

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&total_elements](int i){total_elements *= i;} );
```

Functors / Lambda comparison

```
struct mod
{
    mod(int m) : modulus(m) {}
    int operator()(int v){ return v % modulus; }
    int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textit{introducer}\\\textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

ciere.com

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textbf{introducer}\\ \textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

ciere.com

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textit{introducer}\\ \textbf{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer capture}}\ \underbrace{\texttt{(int v\_)}}_{\textbf{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

ciere.com

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textit{introducer}\\ \textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\color{red}\texttt{->int}}_{\textbf{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textit{introducer}\\\textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textbf{statement}}$$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer}}\ \underbrace{\texttt{(int v\_) ->int}}_{\textbf{declarator}}\underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

ciere.com

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod](int v\_)->int\{return v\_ \% my\_mod;\}}}_{\textit{lambda expression}}$$

$$\Downarrow$$

closure object

- Evaluation of the expression results in a temporary called a closure object
- A closure object is unnamed
- A closure object behaves like a function object

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod](int v\_)->int\{return v\_ \% my\_mod;\}}}_{\textit{lambda expression}}$$

$$\Downarrow$$

closure object

- Evaluation of the expression results in a temporary called a closure object
- A closure object is unnamed
- A closure object behaves like a function object

ciere.com

[&]()->rt{...} – introducer

```
[](){ cout << "foo" << endl; }
```

ciere.com

`[&]()->rt{...}` – introducer

$$\underbrace{\texttt{[]}}_{\textbf{introducer}}\ \underbrace{\texttt{()}}_{\textit{parameters}}\ \underbrace{\texttt{\{cout << "foo" <<endl;\}}}_{\textit{statement}}$$

We start off a lambda expression with the introducer

`[&]()->rt{...}` – introducer

$$\underbrace{\texttt{[](){cout << "foo" <<endl;}}}_{}$$

⇓

closure object

`[&]()->rt{...}` – introducer

How can we call this nullary *function object*-like temporary?

```
[](){ cout << "foo" << endl; } ();
```

Output

```
foo
```

ciere.com

`[&]()->rt{...}` – introducer

How can we call this nullary *function object*-like temporary?

```
[](){ cout << "foo" << endl; } ();
```

Output

```
foo
```

ciere.com

[&]()->rt{...} – parameter

```
[](int v){cout << v << "*6=" << v*6 << endl;} (7);
```

Output

```
7*6=42
```

[&]()->rt{...} – parameter

```
[](int v){cout << v << "*6=" << v*6 << endl;} (7);
```

Output

```
7*6=42
```

[&]()->rt{...} – parameter

```
int i = 7;

[](int & v){ v *= 6; } (i);

cout << "the correct value is: " << i << endl;
```

Output

```
the correct value is: 42
```


ciere.com

`[&]()->rt{...}` – parameter

```
int i = 7;

[](int & v){ v *= 6; } (i);

cout << "the correct value is: " << i << endl;
```

Output

```
the correct value is:  42
```

[&]()->rt{...} – parameter

```
int j = 7;

[](int const & v){ v *= 6; } (j);

cout << "the correct value is: " << j << endl;
```

Compile error

```
error: assignment of read-only reference 'v'
```

`[&]()->rt{...}` – parameter

```
int j = 7;

[](int const & v){ v *= 6; } (j);

cout << "the correct value is: " << j << endl;
```

Compile error

```
error:  assignment of read-only reference 'v'
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int v)
{v *= 6; cout << "v: " << v << endl;} (j);
```

Output

```
v: 42
```

[&]()->rt{...} – parameter

```
int j = 7;

[](int v)
{v *= 6; cout << "v: " << v << endl;} (j);
```

Output

```
v: 42
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int & v, int j){ v *= j; } (j,6);

cout << "j: " << j << endl;
```

Notice that the lambda's parameters do not affect the namespace.

Output

```
j: 42
```

ciere.com

`[&]()->rt{...}` – parameter

```
int j = 7;

[](int & v, int j){ v *= j; } (j,6);

cout << "j: " << j << endl;
```

Notice that the lambda's parameters do not affect the namespace.

Output

```
j: 42
```

ciere.com

`[&]()->rt{...}` – parameter

```
[](){ cout << "foo" << endl; } ();
```

same as

```
[]{ cout << "foo" << endl; } ();
```

Lambda expression without a declarator acts as if it were ()

ciere.com

`[&]()->rt{...}` – parameter

```
[](){ cout << "foo" << endl; } ();
```

same as

```
[]{ cout << "foo" << endl; } ();
```

Lambda expression without a declarator acts as if it were **()**

`[&]()->rt{...}` – capture

We commonly want to capture state or access values outside our *function objects*.

With a function object we use the constructor to populate state.

```
struct mod
{
   mod(int m_ ) : modulus( m_ ) {}
   int operator()(int v_){ return v_ % modulus; }
   int modulus;
};
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

[&]()->rt{...} – capture

Lambda expressions provide an optional capture.

```
[my_mod](int v_) ->int { return v_ % my_mod; }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[&](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[=]() { ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[&identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[foo,&bar,gorp](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[&,identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[=,&identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Capture default all by reference

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&](int i){ total_elements *= i; } );
```

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
   int i = 0;
   while( !done() )
   {
      v.push_back( i++ );
   }
}

vector<int> stuff;
fill( stuff,
      [&]{ return stuff.size() >= 8; } );
```

Output

0 1 2 3 4 5 6 7

ciere.com

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
      [&]{ return stuff.size() >= 8; } );
```

Output

```
0 1 2 3 4 5 6 7
```

`[&]()->rt{...}` – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
        [&]{ int sum=0;
            for_each( stuff.begin(), stuff.end(),
                      [&](int i){ sum += i; } );
            return sum >= 10;
          }
    );
```

Output

0 1 2 3 4

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
        [&]{ int sum=0;
             for_each( stuff.begin(), stuff.end(),
                       [&](int i){ sum += i; } );
             return sum >= 10;
           }
    );
```

Output

```
0 1 2 3 4
```

[=]()->rt{...} – capture

Capture default all by value

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [=](int v){ return v % my_mod; } );
```

`[=]()->rt{...}` – capture

Where is the value captured?

```
int v = 42;
auto func = [=]{ cout << v << endl; };
v = 8;
func();
```

[=]()->rt{...} – capture

Where is the value captured?

```
int v = 42;
auto func = [=]{ cout << v << endl; };
v = 8;
func();
```

At the time of evaluation

Output

```
42
```

ciere.com

[=]()->rt{...} – capture

```
int i = 10;
auto two_i = [=]{ i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Compile error

```
error: assignment of member
'capture_test()::<lambda()>::i' in read-only
object
```

[=]()->rt{...} – capture

```
int i = 10;
auto two_i = [=]{ i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Compile error

error: assignment of member 'capture_test()::<lambda()>::i' in read-only object

ciere.com

[=]()->rt{...} – capture

Lambda closure objects have a public `inline` function call operator that:

- Matches the parameters of the lambda expression
- Matches the return type of the lambda expression
- Is declared **`const`**

Make mutable:

```
int i = 10;
auto two_i = [=]() mutable { i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Output

```
2i:20 i:10
```

`[=]()->rt{...}` – capture

Lambda closure objects have a public `inline` function call operator that:

- Matches the parameters of the lambda expression
- Matches the return type of the lambda expression
- Is declared **`const`**

Make mutable:

```
int i = 10;
auto two_i = [=]() mutable { i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Output

```
2i:20 i:10
```

[=,&identifer]()->rt{...} – capture

```
class gorp
{
    vector<int> values;
    int m_;

public:

    gorp(int mod) : m_(mod) {}
    gorp& put(int v){ values.push_back(v); return *this; }


    int extras()
    {
        int count = 0;
        for_each( values.begin(), values.end(),
                  [=,&count](int v){ count += v % m_; } );

        return count;
    }
};

gorp g(4);
g.put(3).put(7).put(8);
cout << "extras: " << g.extras();
```

`[=,&identifer]()->rt{...}` – capture

Capture default by value and count by reference

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

[=,&identifer]()->rt{...} – capture

Capture count by reference, accumulate, return

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

ciere.com

`[=,&identifer]()->rt{...}` – capture

How did we get **m_**?

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

`[=,&identifer]()->rt{...}` – capture

Implicit capture of **`this`** by value

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

ciere.com

[=,&identifer]()->rt{...} – capture

```
class gorp
{
   vector<int> values;
   int m_;

public:

   int extras()
   {
      int count = 0;
      for_each( values.begin(), values.end(),
                [=,&count](int v){ count += v % m_; } );

      return count;
   }
};

gorp g(4);
g.put(3).put(7).put(8);
cout << "extras: " << g.extras();
```

```
extras:  6
```

[=]()->rt{...} – capture

Will this compile? If so, what is the result?

```
struct foo
{
   foo() : i(0) {}
   void amazing(){ [=]{ i=8; }(); }

   int i;
};

foo f;
f.amazing();
cout << "f.i : " << f.i;
```

[=] ()->rt{...} – capture

this implicity captured. **mutable** not required.

```
struct foo
{
    foo() : i(0) {}
    void amazing(){ [=]{ i=8; }(); }

    int i;
};

foo f;
f.amazing();
cout << "f.i : " << f.i;
```

Output

```
f.i : 8
```

`[=,&identifer]()->rt{...}` – capture

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[i,j,&z](){...} // ok
[&a,b](){...}   // ok
[z,&i,z](){...} // bad, z listed twice
```

`[=,&identifer]()->rt{...}` – capture

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[=,&j,&z](){...}   // ok
[=,this](){...}    // bad, no this with default =
[=,&i,z](){...}    // bad, z by value
```

`[=,&identifer]()->rt{...}` – capture

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[&,j,z](){...}     // ok
[&,this](){...}    // ok
[&,i,&z](){...}    // bad, z by reference
```

`[=]()->rt{...}` – capture

Scope of capture:

- Captured entity must be defined or captured in the immediate enclosing lambda expression or function

[=]()->rt{...} – capture

```
int i = 8;
{
    int j = 2;
    auto f = [=]{ cout << i/j; };
    f();
}
```

Output

4

[=]()->rt{...} – capture

```
int i = 8;
{
    int j = 2;
    auto f = [=]{ cout << i/j; };
    f();
}
```

Output

4

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Output

4

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Output

4

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [i]()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Output

4

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [i]()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Output

4

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [] ()
    {
       int j = 2;
       auto m = [=]{ cout << i/j; };
       m();
    };

f();
```

Compile error

```
error: 'i' is not captured
```

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    []()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Compile error

`error: 'i' is not captured`

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [&]{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Compile error

```
error: assignment of read-only location
'...()::<lambda()>::<lambda()>::i'
```

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [&]{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Compile error

```
error: assignment of read-only location
'...()::<lambda()>::<lambda()>::i'
```

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [i]() mutable
    {
        int j = 2;
        auto m = [&i,j]()mutable{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Output

```
inner: 4 outer: 8
```

ciere.com

[=] ()->rt{...} – capture

```
int i = 8;
auto f =
    [i]() mutable
    {
      int j = 2;
      auto m = [&i,j]()mutable{ i /= j; };
      m();
      cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Output

```
inner: 4 outer: 8
```

ciere.com

[=]()->rt{...} – capture

```
int i=1,  j=2, k=3;
auto f =
    [i,&j,&k]() mutable
    {
        auto m =
            [&i,j,&k]() mutable
            {
                i=4; j=5; k=6;
            };

        m();
        cout << i << j << k;
    };

f();
cout << " : " << i << j << k;
```

Output

```
426 :  126
```

[=]()->rt{...} – capture

```
int i=1,  j=2, k=3;
auto f =
    [i,&j,&k]() mutable
    {
        auto m =
            [&i,j,&k]() mutable
            {
                i=4; j=5; k=6;
            };

        m();
        cout << i << j << k;
    };

f();
cout << " : " << i << j << k;
```

Output

```
426 :  126
```

[=]()->rt{...} – capture

- Closure object has implicity-declared copy constructor / destructor.

```
uct trace

ace()  : i(0)                { cout << "construct\n"; }
ace(trace const &)           { cout << "copy construct\n"; }
race()                       { cout << "destroy\n"; }
ace& operator=(trace&)       { cout << "assign\n"; return *this;}

t i;
```

ciere.com

[=]()->rt{...} – capture

- Closure object has implicity-declared copy constructor / destructor.

```
uct trace

ace() : i(0)                { cout << "construct\n"; }
ace(trace const &)          { cout << "copy construct\n"; }
race()                      { cout << "destroy\n"; }
ace& operator=(trace&)      { cout << "assign\n"; return *this;}

t i;
```

ciere.com

[=]()->rt{...} – capture

Closure object has implicity-declared copy constructor / destructor.

```
{
    trace t;
    int i = 8;

    // t not used so not captured
    auto m1 = [=](){ return i/2; };
}
```

Output

```
construct
destroy
```

[=]()->rt{...} – capture

Closure object has implicity-declared copy constructor / destructor.

```
{
   trace t;
   int i = 8;

   // t not used so not captured
   auto m1 = [=](){ return i/2; };
}
```

Output

```
construct
destroy
```

[=]()->rt{...} – capture

```
{
    trace t;

    // capture t by value
    auto m1 = [=](){ int i=t.i; };

    cout << "-- make copy --" << endl;
    auto m2 = m1;
}
```

Output

```
construct
copy construct
- make copy -
copy construct
destroy
destroy
destroy
```

[=]()->rt{...} – capture

```
{
    trace t;

    // capture t by value
    auto m1 = [=](){ int i=t.i; };

    cout << "-- make copy --" << endl;
    auto m2 = m1;
}
```

Output

```
construct
copy construct
- make copy -
copy construct
destroy
destroy
destroy
```

`[=]()->rt{...}` – capture

C++14 adds Capture Expressions:

```
auto f = [foo = 1](){ cout << foo << endl; }
f();
```

`[=]()->rt{...}` – capture

Capture Expressions allow move-only types to be *captured*

```
auto ptr = std::make_unique<some_type>();
auto f = [p = std::move(ptr)](){ ... }
```

`ptr is destroyed when f leaves scope.`

[=]()->rt{...} – capture

```
auto f = [foo = 0]() mutable
         { cout << ++foo << endl; }
f();
f();
```

`[=]()->rt{...}` – capture

C++14 adds Polymorphic Lambda Expressions:

```
auto f = [](auto x, auto y){ return x+y; }
f(7.6,42);
f(std::string{"hi "},std::string{"mom"});
```

`auto` uses template argument type deduction rules here.

ciere.com

`[&]()->rt{...}` – return type

If the return type is omited from the lambda expression

and the statement has a return such as:

`{ ... return expression; }` *C++11 constraints*

then it is the type of the returned expression after:

- lvalue-to-rvalue conversion
- array-to-pointer conversion
- function-to-pointer conversion

Otherwise, the type is **`void`**

ciere.com

`[&]()->rt{...}` – return type

C++14 Return Deduction

If the return type is omited from the lambda expression

then it is the type of the returned expression after:

- lvalue-to-rvalue conversion
- array-to-pointer conversion
- function-to-pointer conversion

Otherwise, the type is **`void`**

Note: All returns expressions must deduce to the same type

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Storing / Passing Lambda Objects

Seen two ways so far:

- `tempalte<typename T> void foo(T f)`
- `auto f = []{};`

Function pointer

If the lambda expression has no capture it can be converted to a function pointer with the same signature.

```
typedef int(*f_type)(int);

f_type f = [](int i){ return i+20; };

cout << f(8);
```

Output

```
28
```

Function pointer

If the lambda expression has no capture it can be converted to a function pointer with the same signature.

```
typedef int(*f_type)(int);

f_type f = [](int i){ return i+20; };

cout << f(8);
```

Output

```
28
```

ciere.com

`function<R(Args...)>`

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

```
std::function< R ( A1, A2, A3...) > f;
```

ciere.com

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

ciere.com

`function<R(Args...)>`

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

ciere.com

function<R(Args...)>

Type	Old School Define	std::function
Free	`int(*callback)(int,int)`	`function< int(int,int) >`
Member	`int (object_t::*callback)(int,int)`	`function< int(int,int) >`
Functor	`object_t callback`	`function< int(int,int) >`

Function pointers

```
int my_free_function(std::string s)
{
   return s.size();
}
```

```
std::function< int(std::string) > f;
f = my_free_function;

int size = f("cierelabs.net");
```

Member function pointers

```
struct my_struct
{
   my_struct( std::string const & s) : s_(s) {}
   int size() const { return s_.size(); }
   std::string s_;
};
```

```
my_struct mine("cierelabs.net");
std::function< int() > f;

f = std::bind( &my_struct::size, std::ref(mine) );
int size = f();
```

function<R(Args...)>

Functors

```
struct my_functor
{
   my_functor( std::string const & s) : s_(s) {}
   int operator()() const
   {
      return s_.size();
   }

   std::string s_;
};


my_functor mine("cierelabs.net");
std::function< int() > f;

f = std::ref(mine);
int size = f();
```

function<R(Args...)>

Closure Objects

```
std::function< int(std::string const &) > f;

f = [](std::string const & s){ return s.size(); };
int size = f("cierelabs.net");
```

ciere.com

Fun with function

```
std::function<int(int)> f1;
std::function<int(int)> f2 =
    [&](int i)
    {
        cout << i << " ";
        if(i>5) { return f1(i-2); }
    };

f1 =
    [&](int i)
    {
        cout << i << " ";
        return f2(++i);
    };

f1(10);
```

Output

10 11 9 10 8 9 7 8 6 7 5 6 4 5

Fun with function

```
std::function<int(int)> f1;
std::function<int(int)> f2 =
    [&](int i)
    {
        cout << i << " ";
        if(i>5) { return f1(i-2); }
    };

f1 =
    [&](int i)
    {
        cout << i << " ";
        return f2(++i);
    };

f1(10);
```

Output

10 11 9 10 8 9 7 8 6 7 5 6 4 5

More fun with function

```cpp
std::function<int(int)> fact;

fact =
    [&fact](int n)->int
        {
            if(n==0){ return 1; }
            else
            {
                return (n * fact(n-1));
            }
        } ;

cout << "factorial(4) : " << fact(4) << endl;
```

Output

```
factorial(4) :  24
```

More fun with function

```
std::function<int(int)> fact;

fact =
    [&fact](int n)->int
        {
            if(n==0){ return 1; }
            else
            {
                return (n * fact(n-1));
            }
        } ;

cout << "factorial(4) : " << fact(4) << endl;
```

Output

```
factorial(4) :  24
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

ciere.com

Exercise

- Class that can queue callable "things"
- The callable "thing" takes an `int` argument
- The callable "thing" returns an `int`
- The class will have a method:

```
int run( int init )
```

- When `run` is called:
  - Call each of the queued items
  - `init` will be the initial state of the first call
  - The result of each call feeds the input of the next call
  - The result of final call will be the return value of `run`

Exercise

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

struct machine
{
   template< typename T >
   void add( T f )
   {
      to_do.push_back(f);
   }

   int run( int v )
   {
      std::for_each( to_do.begin(), to_do.end(),
                     [&v]( std::function<int(int)> f )
                     { v = f(v); } );
      return v;
   }

   std::vector< std::function<int(int)> > to_do;
};

int foo(int i){ return i+4; }

int main()
{
   machine m;
   m.add( [](int i){ return i*3; } );
   m.add( foo );
   m.add( [](int i){ return i/5; } );

   std::cout << "run(7) : " << m.run(7) << std::endl;
   return 1;
}
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

ciere.com

Where can we use them?

Lambda expression cannot appear in an unevaluated operand.

- typeid
- sizeof
- noexcept
- decltype

Some thoughts

Make Stepanov happy, revisit standard algorithms.

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Your Thoughts

How are you going to use lambdas?

Part X

Smart Pointers

Overview

- RAII for managing dynamically allocated objects
- `std::auto_ptr` has been deprecated
- `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`

Overview

- RAII for managing dynamically allocated objects
- `std::auto_ptr` has been deprecated
- `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`

Overview

- RAII for managing dynamically allocated objects
- `std::auto_ptr` has been deprecated
- `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`

ciere.com

Outline

- unique_ptr
- shared_ptr
- weak_ptr
- shared_from_this

unique_ptr

- Not Copyable
- One Owner

unique_ptr - the slushie

```
struct slushie
{
   slushie( const std::string & flavor ) : flavor_(flavor)
   {
      std::cout << "make " << flavor_  << " slushie\n";
   }

   ~slushie()
   {
      std::cout << "clean-up " << flavor_ << " slushie\n";
   }

   void consume()
   {
      std::cout << "too fast!\n";
      throw brain_freeze();
   }

   std::string flavor_;
};
```

unique_ptr - plain ol' new

```
void wild_slushie_shack()
{
   std::cout << "open slushie shack" << std::endl;

   slushie* biggie = new slushie("purple");
   biggie->consume();
   delete biggie;

   std::cout << "slushie shack closed" << std::endl;
}



try{ wild_slushie_shack();  }
catch(std::exception const& e)
{
   std::cout << "oh no! " << e.what() << std::endl;
}
```

```
open slushie shack
make purple slushie
too fast!
oh no! brain freeze
```

unique_ptr - plain ol' new

```
void wild_slushie_shack()
{
   std::cout << "open slushie shack" << std::endl;

   slushie* biggie = new slushie("purple");
   biggie->consume();
   delete biggie;

   std::cout << "slushie shack closed" << std::endl;
}



try{ wild_slushie_shack();  }
catch(std::exception const& e)
{
   std::cout << "oh no! " << e.what() << std::endl;
}
```

```
open slushie shack
make purple slushie
too fast!
oh no! brain freeze
```

unique_ptr - RAII to the rescue

```
void safe_slushie_shack()
{
   std::cout << "open slushie shack" << std::endl;

   unique_ptr<slushie> biggie = std::make_unique<slushie>("purple");
   biggie->consume();

   std::cout << "slushie shack closed" << std::endl;
}



try{ safe_slushie_shack();  }
catch(std::exception const& e)
{
   std::cout << "oh no! " << e.what() << std::endl;
}
```

```
open slushie shack
make purple slushie
too fast!
clean-up purple slushie
oh no! brain freeze
```

unique_ptr - RAII to the rescue

```
void safe_slushie_shack()
{
   std::cout << "open slushie shack" << std::endl;

   unique_ptr<slushie> biggie = std::make_unique<slushie>("purple");
   biggie->consume();

   std::cout << "slushie shack closed" << std::endl;
}



try{ safe_slushie_shack();  }
catch(std::exception const& e)
{
   std::cout << "oh no! " << e.what() << std::endl;
}
```

```
open slushie shack
make purple slushie
too fast!
clean-up purple slushie
oh no! brain freeze
```

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

```
unique_ptr<cup> my_cup = make_unique<cup>();
unique_ptr<cup> your_cup = make_unique<cup>();

your_cup = my_cup;

-------
unique_synopsis.cpp:16:13: error:
overload resolution selected deleted operator '
   your_cup = my_cup;
   ~~~~~~~~ ^ ~~~~~~
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

```
unique_ptr<cup> my_cup = make_unique<cup>();
unique_ptr<cup  your_cup( my_cup );

-------
unique_synopsis.cpp:22:20: error:
error: call to deleted constructor of 'unique_p
   unique_ptr<cup> your_cup( my_cup );
                  ^         ~~~~~~
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr - Synopsis

`std::auto_ptr` - a *bad* idea

```
auto_ptr<cup> my_cup( new cup );
auto_ptr<cup> your_cup( new cup );


your_cup = my_cup;


your_cup->drink();
my_cup->drink();
  **boom**
```

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();
```

unique_ptr - Synopsis

`std::auto_ptr` - a *really bad* idea - implicit lvalue move

```
int clean( auto_ptr<cup> some_cup );
...

auto_ptr<cup> my_cup( new cup );
clean( my_cup );
my_cup->fill();   **boom**
my_cup->drink();
```

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
```

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr - Synopsis

```
template<class T>
class unique_ptr {
  public:
    using element_type = T;

    unique_ptr(unique_ptr const &) = delete;
    void operator=(unique_ptr const &) = delete;

    explicit unique_ptr(T * p = nullptr);
    ~unique_ptr();

    void reset(T * p = nullptr);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    explicit operator bool() const;

    void swap(unique_ptr & b);
};

template<class T> void swap(unique_ptr<T> & a, unique_ptr<T> & b);
```

- Methods **never** throw
- Strict ownership
- Acts like a pointer
- Supports swap and move

unique_ptr Summary

- Provides RAII semantics for dynamically allocated objects
- Use when there is a single owner
- *Don't* use `auto_ptr`. Favor `unique_ptr` or `unique_ptr`

Outline

- unique_ptr
- shared_ptr
- weak_ptr
- shared_from_this

shared_ptr

- Copyable
- Shared Ownership
- Last owner deletes
- Two flavors: `shared_ptr` and `shared_array`

shared_ptr - Copyable

```
shared_ptr<cup> my_cup = make_shared<cup>("blue");
shared_ptr<cup> your_cup = make_shared<cup>("green");

your_cup = my_cup;

your_cup->fill();
my_cup->drink();
```

```
create blue cup
create green cup
destroy green cup
fill blue cup
drink blue cup
destroy blue cup
```

shared_ptr - Copyable

```
shared_ptr<cup> my_cup = make_shared<cup>("blue");
shared_ptr<cup> your_cup = make_shared<cup>("green");

your_cup = my_cup;

your_cup->fill();
my_cup->drink();
```

create blue cup
create green cup
destroy green cup
fill blue cup
drink blue cup
destroy blue cup

shared_ptr - reference counted

```
shared_ptr<cup> my_cup;
{
   std::cout << "<-- enter ---" << std::endl;
   shared_ptr<cup> your_cup = make_shred<cup>("green");
   your_cup->fill();
   your_cup->drink();
   my_cup = your_cup;
   std::cout << "--- exit --->" << std::endl;
}
clean(my_cup);
```

```
<- enter --
create green cup
fill green cup
drink green cup
-- exit -->
cleaning green cup
destroy green cup
```

shared_ptr - reference counted

```
shared_ptr<cup> my_cup;
{
   std::cout << "<-- enter ---" << std::endl;
   shared_ptr<cup> your_cup = make_shred<cup>("green");
   your_cup->fill();
   your_cup->drink();
   my_cup = your_cup;
   std::cout << "--- exit --->" << std::endl;
}
clean(my_cup);
```

```
<- enter --
create green cup
fill green cup
drink green cup
-- exit -->
cleaning green cup
destroy green cup
```

shared_ptr - arbitrary object handling

```
shared_ptr<void> generic;
{
   std::cout << "<-- enter ---" << std::endl;
   shared_ptr<cup> my_cup = make_shared<cup>("black");
   generic = my_cup;
   std::cout << "--- exit --->" << std::endl;
}

shared_ptr<void> generic = make_shared<cup>("white");
```

```
<- enter --
create black cup
-- exit -->
destroy black cup
```

```
create white cup
destroy white cup
```

shared_ptr - arbitrary object handling

```
shared_ptr<void> generic;
{
   std::cout << "<-- enter ---" << std::endl;
   shared_ptr<cup> my_cup = make_shared<cup>("black");
   generic = my_cup;
   std::cout << "--- exit --->" << std::endl;
}

shared_ptr<void> generic = make_shared<cup>("white");
```

```
<- enter --
create black cup
-- exit -->
destroy black cup


create white cup
destroy white cup
```

shared_ptr - unnamed temporaries

Avoid unnamed `shared_ptr` temporaries.

```
void paint( shared_ptr<cup> new_cup, color_t color );
```

Bad! The evaluation order of function arguments is unspecified.

```
paint( shared_ptr<cup>( new cup ), get_color() );
```

Good!

```
shared_ptr<cup> new_cup( new cup );
paint( new_cup, get_color() );

// or
paint( make_shared<cup>(), get_color() );
```

shared_ptr - custom deleter

```
template<class T> class shared_ptr {

  public:
    shared_ptr(); // never throws
    template<class Y> explicit shared_ptr(Y * p);
    template<class Y, class D> shared_ptr(Y * p, D d);
```

Requirements:

- **p** must be convertible to **T***
- **D** must be **CopyConstructible**
- The copy constructor and destructor of **D** must not throw
- The expression `d(p)` must be well-formed and not throw

shared_ptr - specialized FILE

Work out how you might use `shared_ptr` to automatically close a FILE that is no longer in use.

Hint: Sketch out interface for

- open
- read
- write

shared_ptr - thread safety

- Thread safety rules apply to the `shared_ptr` instances and not on their pointees
- Same level of thread safety as built-in types
- Instance can be *read* simultneously by multiple threads
- Different instances can be *written to* simultneously by multiple threads
- All other simultaneous access is undefined

shared_ptr - thread safety

```
shared_ptr<int> p(new int(42));

// thread A
shared_ptr<int> p2(p); // reads p

// thread B
shared_ptr<int> p3(p); // OK, multiple reads are safe
```

shared_ptr - thread safety

```
shared_ptr<int> p(new int(42));

// thread A
p.reset(new int(1912)); // writes p

// thread B
p2.reset(); // OK, writes p2
```

ciere.com

shared_ptr - thread safety

```
shared_ptr<int> p(new int(42));

// thread A
p = p3; // reads p3, writes p

// thread B
p3.reset(); // writes p3; undefined, simultaneous read/write
```

ciere.com

shared_ptr - thread safety

```
// thread A
p3 = p2; // reads p2, writes p3

// thread B
// p2 goes out of scope: undefined, the destructor
// is considered a "write access"
```

shared_ptr - thread safety

```
// thread A
p3.reset(new int(1));

// thread B
p3.reset(new int(2)); // undefined, multiple writes
```

shared_ptr

- Shared ownership
- Reference counted, watch for cyclic ownership
- Most implementations are lock free

Outline

- unique_ptr
- shared_ptr
- weak_ptr
- shared_from_this

weak_ptr

- Copyable
- Stores a *weak* reference to a `shared_ptr` managed object
- Doesn't own anything
- Use to break reference counting cycles
- Obtain `shared_ptr` from a `weak_ptr`

ciere.com

weak_ptr - Synopsis

Methods never throw.

```
template<class T> class weak_ptr {

  public:
    typedef T element_type;

    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);

    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;

    void reset();
    void swap(weak_ptr<T> & b);
};

template<class T, class U>
  bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
  void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

weak_ptr - Synopsis

Construct from a `shared_ptr`.

```
template<class T> class weak_ptr {

  public:
    typedef T element_type;

    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);

    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;

    void reset();
    void swap(weak_ptr<T> & b);
};

template<class T, class U>
  bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
  void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

weak_ptr - Synopsis

Construct from another `weak_ptr`.

```
template<class T> class weak_ptr {

  public:
    typedef T element_type;

    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);

    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;

    void reset();
    void swap(weak_ptr<T> & b);
};

template<class T, class U>
  bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
  void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

weak_ptr - Synopsis

Construct `shared_ptr` from a `weak_ptr` or use `lock`

```
template<class T> class weak_ptr {

  public:
    typedef T element_type;

    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);

    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;

    void reset();
    void swap(weak_ptr<T> & b);
};

template<class T, class U>
  bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
  void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

weak_ptr - Synopsis

These are primarily for diagnostics.

```
template<class T> class weak_ptr {

  public:
    typedef T element_type;

    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);

    ~weak_ptr();

    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(shared_ptr<Y> const & r);

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;

    void reset();
    void swap(weak_ptr<T> & b);
};

template<class T, class U>
  bool operator<(weak_ptr<T> const & a, weak_ptr<U> const & b);

template<class T>
  void swap(weak_ptr<T> & a, weak_ptr<T> & b);
```

weak_ptr - Cup Cleaner

```
using registered_container_t = std::vector< weak_ptr<cup> >;
registered_container_t registered_cups;

shared_ptr<cup> blue = make_shared<cup>("blue");
registered_cups.push_back( blue );

{
   shared_ptr<cup> green = make_shared<cup>("green") );
   registered_cups.push_back( green );

   {
      shared_ptr<cup> red = make_shared<cup>("red");
      registered_cups.push_back( red );
   }

   shared_ptr<cup> orange = make_shared<cup>("orange");
   registered_cups.push_back( orange );

   cout << "<--- clean all cups ----" << endl;
   clean_all_cups();
   cout << "---- clean all cups --->" << endl;
}
```

weak_ptr - Cup Cleaner - the race

```
void clean_all_cups_race()
{
    for(auto & c : registered_cups)
    {
        if( !c.expired() )
        {
            shared_ptr<cup> some_cup( c );
            clean( some_cup );
        }
        else
        {
            cout << "----- missing cup -----\n";
        }
    }
}
```

weak_ptr - Cup Cleaner - using lock

```
void clean_all_cups()
{
    for(auto & c : registered_cups)
    {
        if( shared_ptr<cup> some_cup = c.lock() )
        {
            clean( some_cup );
        }
        else
        {
            cout << "<missing cup>\n";
        }
    }
}
```

ciere.com

weak_ptr - Cup Cleaner - using lock

```
create blue cup
create green cup
create red cup
destroy red cup
create orange cup
<-- clean all cups ---
cleaning blue cup
cleaning green cup
<missing cup>
cleaning orange cup
--- clean all cups -->
destroy orange cup
destroy green cup
destroy blue cup
```

weak_ptr - Cup Cleaner - constructing from weak

```
void clean_all_cups_throw()
{
    for(auto & c : registered_cups)
    {
        try
        {
            clean( c );
        }
        catch( std::bad_weak_ptr& e )
        {
            cout << "< " << e.what() << " >\n";
        }
    }
}
```

weak_ptr - Cup Cleaner - constructing from weak

```
create blue cup
create green cup
create red cup
destroy red cup
create orange cup
<-- clean all cups ---
cleaning blue cup
cleaning green cup
< tr1::bad_weak_ptr >
cleaning orange cup
--- clean all cups -->
destroy orange cup
destroy green cup
destroy blue cup
```

Outline

- unique_ptr
- shared_ptr
- weak_ptr
- shared_from_this

shared_from_this

- Allows us to get a `shared_ptr` from within an object
- Uses CRTP base `enable_shared_from_this` to *inject* the required support

shared_from_this

```
class asio_client_handler
   : public std::enable_shared_from_this<asio_client_handler>
{
private:
   void read_packet()
   {
      asio::async_read_until( socket_, in_packet, '\0',
                              std::bind( &type::read_packet_done
                                       , shared_from_this()
                                       , _1, _2) );
   }

   void read_packet_done( boost::system::error_code const & error
                        , int bytes_transferred)
   {
      if(error){ return; }

      try
      {
         std::istream stream(&in_packet);
         packet_t packet = ciere::json::construct(stream);
         update_handler(shared_from_this(), packet);
      }
      catch( ... )
      {}

      read_packet();
   }
};
```

Part XI

Tuple

Outline

- Overview
- tuple
- Summary

ciere.com

Overview

- Fixed sized collection
- `std::pair` ... but more
- Multiple return values from functions
- Provides ties

ciere.com

Overview

- Fixed sized collection
- `std::pair` ... but more
- Multiple return values from functions
- Provides ties

Overview

- Fixed sized collection
- `std::pair` ... but more
- Multiple return values from functions
- Provides ties

Overview

- Fixed sized collection
- `std::pair` ... but more
- Multiple return values from functions
- Provides ties

Outline

- Overview
- tuple
- Summary

Making Tuples

```
struct slushie
{
   slushie(const char* flavor ) : flavor_( flavor )
   {}
   ~slushie(){}
   std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;   // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```
struct slushie
{
   slushie(const char* flavor ) : flavor_( flavor )
   {}
   ~slushie(){}
   std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;   // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```
struct slushie
{
   slushie(const char* flavor ) : flavor_( flavor )
   {}
   ~slushie(){}
   std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;    // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```cpp
struct slushie
{
   slushie(const char* flavor ) : flavor_( flavor )
   {}
   ~slushie(){}
   std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;    // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```
struct slushie
{
   slushie(const char* flavor ) : flavor_( flavor )
   {}
   ~slushie(){}
   std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;   // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```
struct slushie
{
    slushie(const char* flavor ) : flavor_( flavor )
    {}
    ~slushie(){}
    std::string flavor_;
};

tuple< double, std::string > a;

tuple< double, std::string > b(9.8, "cat");

tuple< double, slushie, int > c(12.3, slushie("purple"), 42);

tuple< double, slushie, int > d;   // Compile ERROR

double x = 17.5;
tuple< double&, int, int > e(x, 42, 8);

tuple< double&, int, int > f;  // Compile ERROR
```

Making Tuples

```
tuple<std::string,int> make_it()
{
   std::string flavor("purple");
   int cup_count = 42;

   return std::make_tuple(flavor,cup_count);
}
```

Accessing Tuples

```
tuple< double, std::string, int > foo( 12.3
                                     , "purple"
                                     ,  42 );
```

```
cout << "element 1: " << std::get<1>(foo) << endl;
cout << "element 2: " << std::get<2>(foo) << endl;
```

```
foo: (12.3 purple 42)
element 1: purple
element 2: 42
```

Accessing Tuples

```
tuple< double, std::string, int > foo( 12.3
                                     , "purple"
                                     ,  42 );
```

```
cout << "element 1: " << std::get<1>(foo) << endl;
cout << "element 2: " << std::get<2>(foo) << endl;
```

```
foo: (12.3 purple 42)
element 1: purple
element 2: 42
```

ciere.com

Comparing Tuples

```
tuple<double, std::string, int> foo(12.3, "purple", 42);
tuple<double, std::string, int> bar(11.3, "purple", 42);
tuple<double, std::string, int> gorp(12.3, "purple", 42);

cout << "foo==bar  : " << (foo==bar) << endl;
cout << "foo==gorp : " << (foo==gorp) << endl;
```

```
foo==bar  : 0
foo==gorp : 1
```

ciere.com

Comparing Tuples

```
tuple<double, std::string, int> foo(12.3, "purple", 42);
tuple<double, std::string, int> bar(11.3, "purple", 42);
tuple<double, std::string, int> gorp(12.3, "purple", 42);

cout << "foo==bar  : " << (foo==bar) << endl;
cout << "foo==gorp : " << (foo==gorp) << endl;
```

```
foo==bar  : 0
foo==gorp : 1
```

ciere.com

Tie

```
tuple<int,int,int,int> get_version()
{
   return std::make_tuple( 1, 4, 2, 9 );
}
```

1.4.2.9

```
int major, minor, patch, build;

std::tie( major,
          minor,
          patch,
          build ) =  get_version();

cout << major << "."
     << minor << "."
     << patch << "."
     << build << endl;
```

Tie

```
tuple<int,int,int,int> get_version()
{
   return std::make_tuple( 1, 4, 2, 9 );
}


int major, minor, patch, build;

std::tie( major,
          minor,
          patch,
          build ) =  get_version();

cout << major << "."
     << minor << "."
     << patch << "."
     << build << endl;
```

1.4.2.9

Tie Some

```
tuple<int,int,int,int> get_version()
{
   return std::make_tuple( 1, 4, 2, 9 );
}


int patch;

std::tie( std::ignore,
          std::ignore,
          patch,
          std::ignore ) =  get_version();

cout << "patch: " << patch << endl;
```

```
patch: 2
```

Tie Some

```
tuple<int,int,int,int> get_version()
{
   return std::make_tuple( 1, 4, 2, 9 );
}


int patch;

std::tie( std::ignore,
          std::ignore,
          patch,
          std::ignore ) =  get_version();

cout << "patch: " << patch << endl;
```

```
patch: 2
```

Outline

- Overview
- tuple
- Summary

Summary

- Provides tuple facilities in C++
- As efficient as struct
- Can produce cleaner code

Summary

- Provides tuple facilities in C++
- As efficient as struct
- Can produce cleaner code

Summary

- Provides tuple facilities in C++
- As efficient as struct
- Can produce cleaner code

Part XII

New Tricks with Templates

Outline

- extern template
- template aliases
- variadic templates

Instantiation Rules

Compiler must instantiate a template when it finds a fully specified version in a translation unit.

foo.cpp

```
void some_method()
{
   my_amazing_type<std::string::iterator> s;
   ...
}
```

bar.cpp

```
void another_method()
{
   my_amazing_type<std::string::iterator> s;
   ...
}
```

Exporting and Forcing Instantiation

We can force the compiler to instantiate a type with this syntax:

my_types.cpp

```
template class my_amazing_type<std::string::iterator>;
```

We can supress instantiation with this syntax:

my_types.hpp

```
extern template class my_amazing_type<std::string::iterator>;
```

Outline

- extern template
- template aliases
- variadic templates

ciere.com

Motivation

```
template<typename T1, typename T2, typename T3>
struct amazing_thing
{
  ...
};

template<typename T>
typedef amazing_thing<int,T,float> amazing_other;
```

The Solution

Template Alias

```
template<typename T1, typename T2, typename T3>
struct amazing_thing
{
  ...
};

template<typename T>
using amazing_other = amazing_thing<int,T,float>;
```

Other uses for **using**

Type aliasing

```
using funct_t = void (*) (int,int);
using int_t = long long;
```

Outline

- extern template
- template aliases
- variadic templates

Motivation

```
template<typename T1>
void my_amazing_thing(T1 v1);

template<typename T1, typename T2>
void my_amazing_thing(T1 v1, T2 v2);

template<typename T1, typename T2, typename T3>
void my_amazing_thing(T1 v1, T2 v2, T3 v3);

template<typename T1, typename T2, typename T3, typename T4>
void my_amazing_thing(T1 v1, T2 v2, T3 v3, T4 v4);
```

Motivation

```
template<typename... T>
void my_amazing_thing(T... v);
```

Variadic Templates

```
template<typename ...T>
void foo(T... v)
{
   bar(v...);
}
```

- Declares parameter pack - elipse left of the parameter name
- Unpacks a parameter pack - elipse right of the template or function call parameter

Variadic Templates

```
template<typename ...T>
void foo(T... v)
{
   bar(v...);
}
```

- Declares parameter pack - elipse left of the parameter name
- Unpacks a parameter pack - elipse right of the template or function call parameter

Variadic Templates

```
template<typename ...T>
void foo(T... v)
{
   bar(v...);
}
```

- Declares parameter pack - elipse left of the parameter name
- Unpacks a parameter pack - elipse right of the template or function call parameter

Variadic Templates

```
template<typename T0, typename... Tn>
void foo(T0 v0, Tn... v)
{
   bar(v0, v...);
}
```

Variadic Templates

How do we iterate the parameter pack?

Recursion

```
void bar()
{}

template<typename T0, typename... Tn>
void bar(T0 v0, Tn... v)
{
   std::cout << v0 << ' ';
   bar(v...);
}
```

Expand Trick

```
template<typename... T>
void expand(T... t)
{}

template<typename... T>
void bar(T... t)
{
    expand( [&]{ std::cout << t << ' '; }()... );
}
```

Output - clang

```
error: no matching function for call to 'expand'
   expand( [&] std::cout « t « ' '; ()... );
   ^~~~~~
...
note: candidate template ignored: substitution failure [with T = <void, void, void>]:
 argument may not have 'void' type
void expand(T... t)
     ^          ~
```

Expand Trick

```
template<typename... T>
void expand(T... t)
{}

template<typename... T>
void bar(T... t)
{
   expand( [&]{ std::cout << t << ' '; }()... );
}
```

Output - clang

```
error: no matching function for call to 'expand'
   expand( [&] std::cout « t « ' '; ()... );
   ^~~~~~
...
note: candidate template ignored: substitution failure [with T = <void, void, void>]:
 argument may not have 'void' type
void expand(T... t)
     ^           ~
```

Expand Trick

```
template<typename... T>
void expand(T... t)
{}

template<typename... T>
void bar(T... t)
{
   expand( ([&]{ std::cout << t << ' '; }(), 1)... );
}


int main()
{
   bar(12,42.5,'x');
}
```

Output

```
12 42.5 x
```

Lucky?

Expand Trick

```
template<typename... T>
void expand(T... t)
{}

template<typename... T>
void bar(T... t)
{
   expand( ([&]{ std::cout << t << ' '; }(), 1)... );
}


int main()
{
   bar(12,42.5,'x');
}
```

Output

```
12 42.5 x
```

Lucky?

Expand Trick

```
struct expand
{
   template<typename... T>
   expand(T... t)
   {}
};

template<typename... T>
void bar(T... t)
{
   expand{ ([&]{ std::cout << t << ' '; }(), 1)... };
}


int main()
{
   bar(12,42.5,'x');
}
```

Exercise 1

Write a **`weak_bind`** implementation.

- the first parameter will be a pointer to a class method
- the second argument will be a **`std::shared_ptr`** to an object
- the following N arguments will be the bind args
- store internally as a **`std::weak_ptr`**
- only invoke the method if the pointer is not expired
- utilize variadic template parameters
- use **`std::bind`** under the hood

Files - Build

```
exercise/weak_bind.cpp
bjam weak_bind
```

Exercise 1

Write a **`weak_bind`** implementation.

- the first parameter will be a pointer to a class method
- the second argument will be a **`std::shared_ptr`** to an object
- the following N arguments will be the bind args
- store internally as a **`std::weak_ptr`**
- only invoke the method if the pointer is not expired
- utilize variadic template parameters
- use **`std::bind`** under the hood

Files - Build

```
exercise/weak_bind.cpp
bjam weak_bind
```

Exercise 2

Write your very own version of tuple called ... toople.
Bonus points if you write an implementation of get.

Files - Build

```
exercise/toople.cpp
bjam toople
```

Exercise 2

Write your very own version of tuple called ... toople.
Bonus points if you write an implementation of get.

Files - Build

```
exercise/toople.cpp
bjam toople
```

Exercise 2 - Solution

```
template<typename... T>
struct toople;


template<>
struct toople<>
{};


template<typename H, typename... T>
struct toople<H,T...>
{
   using Head = H;
   using Tail = toople<T...>;

   Head     head;
   Tail     tail;
};
```

Exercise 2 - Solution

```
int main()
{
   toople<int,float,char,double> t;
   get<2>(t) = 'z';
   std::cout << "have : " << get<2>(t) << std::endl;

   return 1;
}
```

Exercise 2 - Solution

```
template<typename T>
struct pack_size;

template<typename... T>
struct pack_size< toople<T...> >
{
   static const int value = sizeof...(T);
};



template<int N, typename T>
auto get(T & t) -> typename get_type<N,T>::type &
{
   static_assert( N < pack_size<T>::value
                , "invalid offset" );


   return get_value<N,T>::apply(t);
}
```

Exercise 2 - Solution

```
template<typename T>
struct pack_size;

template<typename... T>
struct pack_size< toople<T...> >
{
   static const int value = sizeof...(T);
};



template<int N, typename T>
auto get(T & t) -> typename get_type<N,T>::type &
{
   static_assert( N < pack_size<T>::value
                , "invalid offset" );

   return get_value<N,T>::apply(t);
}
```

Exercise 2 - Solution

```
template<typename H, typename... T>
struct toople<H,T...>
{
   using Head = H;
   using Tail = toople<T...>;

   Head     head;
   Tail     tail;
};


template<int N,typename T>
struct get_type
{
   using type = typename get_type<N-1,typename T::Tail>::type;
};

template<typename T>
struct get_type<0,T>
{
   using type = typename T::Head;
};
```

Exercise 2 - Solution

```
template<typename H, typename... T>
struct toople<H,T...>
{
   using Head = H;
   using Tail = toople<T...>;

   Head     head;
   Tail     tail;
};


template<int N,typename T>
struct get_value
{
   static typename get_type<N,T>::type & apply(T & t)
   {
      return get_value<N-1,typename T::Tail>::apply(t.tail);
   }
};
```

Exercise 2 - Solution

```
template<int N,typename T>
struct get_value
{
   static typename get_type<N,T>::type & apply(T & t)
   {
      return get_value<N-1,typename T::Tail>::apply(t.tail);
   }
};

template<typename T>
struct get_value<0,T>
{
   static typename T::Head & apply(T & t)
   {
      return t.head;
   }
};
```

Part XIII

Thread

Outline

- Asynchronous Overview
- `std::async`
- Synchronization

What is Asynchronous Activity

Daughter #1

me:	"Please make me a coffee."
daughter:	"Sure Dad"

time passes ... I work. She makes a cappuccino.

daughter:	"Here is your coffee."
me:	"Thanks"

What is Asynchronous Activity

Daughter #3

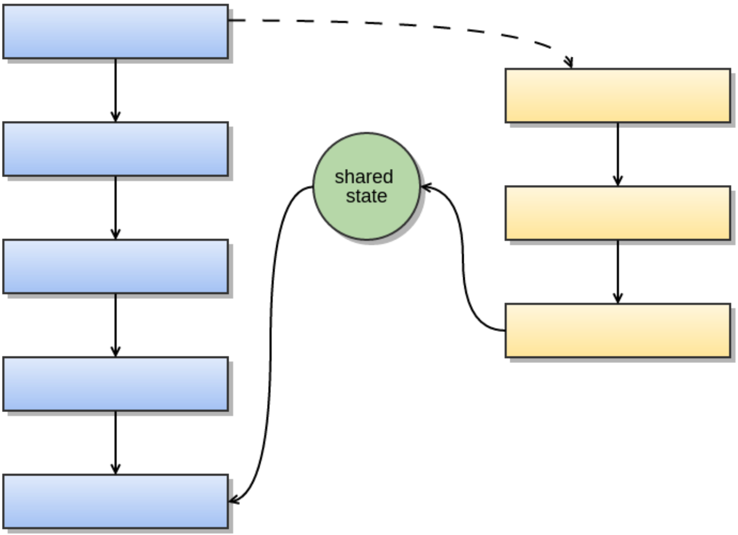
me:	“Please make me a coffee.”
daughter:	“I would love to!”

we both walk to the machine. I supervise (watch). She makes a cappuccino.

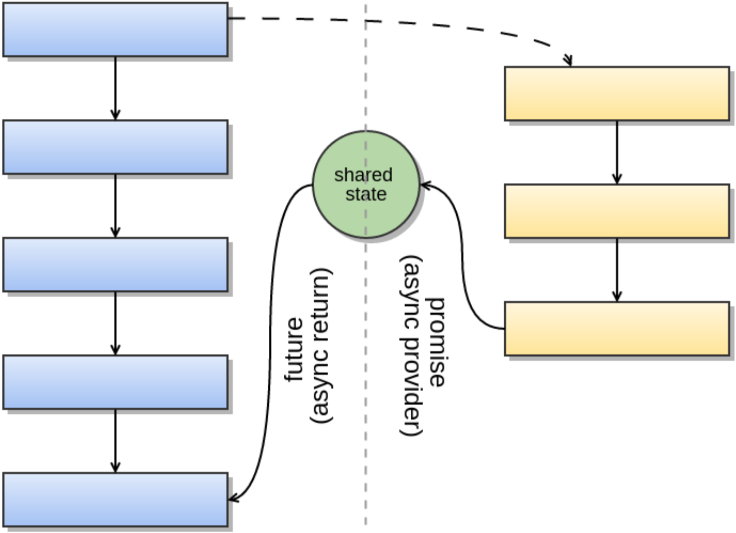
daughter:	“Here is your coffee.”
me:	“Thanks”

ciere.com

Asynchronous



Asynchronous



Outline

- Asynchronous Overview
- `std::async`
- Synchronization

std::async

```
template< class Function, class... Args>
std::future<typename std::result_of<Function(Args...)>::type>
    async( Function&& f, Args&&... args );
```

```
template< class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
    async( std::launch policy, Function&& f, Args&&... args );
```

`std::async`

- Takes a callable and arguments
- Returns the **`std::future`**, the async return object
- The *promise* is satisfied with the return value of the callable

`std::async`

What is the **`std::launch`** policy?

- **`std::launch::async`** - execute the function on a separate thread
- **`std::launch::deferred`** - lazily execute the function when the *future* is accessed. Non-timed access.
- Not specified, same as **`std::launch::async | std::launch::deferred`**

std::async

```
auto f = std::async(  [](id_t id) -> std::string
                      {
                        return lookup_name(id)
                      }
                    , my_id
                   );
```

std::async

```
auto f = std::async(  std::launch::async
                    , [](id_t id) -> std::string
                      {
                        return lookup_name(id)
                      }
                    , my_id
                   );
```

std::future

What can be done with a **future**?

- Wait for the result
- Wait for a relative/absolute time for the result
- Check if the future has a shared state
- Get the result

std::future

```
template <typename T>
struct future
{
  future();

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

  future(future &&);
  future & operator=(future &&);

  // ...
};
```

std::future

```
template <typename T>
struct future
{
  // ...

  T get();
  T & get();
  void get();

  bool valid() const;

  std::shared_future<T> share();

  // ...
};
```

std::future

```
template <typename T>
struct future
{
  // ...

  void wait() const;

  template <class Repr, class Period>
  std::future_status wait_for(
            std::chrono::duration<Repr,Period> const &) const;

  template <class Clock, class Duration>
  std::future_status wait_until(
            std::chrono::time_point<Clock,Duration> const &) co
};
```

`std::future_status`

`std::future_status` :

- **`deferred`** - Function has not started
- **`ready`** - Result is ready
- **`timeout`** - Timeout has expired

std::future_status

```
auto f = std::async( std::launch::deferred
                   , [](){ return 42; } );

std::future_status status;
do{
   status = f.wait_for(std::chrono::seconds(1));

   if (status == std::future_status::deferred)
   {
      std::cout << "deferred\n";
   }
   else if (status == std::future_status::timeout)
   {
      std::cout << "timeout\n";
   }
   else if (status == std::future_status::ready)
   {
      std::cout << "ready!\n";
   }
} while(status != std::future_status::ready);
```

std::async

```
auto f = std::async(  std::launch::async
                    , [](id_t id) -> std::string
                      {
                        return lookup_name(id)
                      }
                    , my_id
                   );

...

auto name = f.get();
```

std::async

```
std::string lookup_name(id_t id)
{
  auto connection = open_db_connection();
  if(!connection)
  {
    throw no_connection;
  }

  ...
}



auto f = std::async( ... );

auto name = f.get();  // gack ???
```

Exercise

Convert the previous work queue exercise so that **`go`** uses **`async`** to perform each operation.

It will wait on the future and then launch the next operation.

Bonus points if you make **`go`** asynchronous with a signature of:

```
std::future<int> go( int initial );
```

```
std::future<int> result = my_worker.go(100);
std::cout << "result: " << result.get() << std::endl;
```

Solution

```
std::future<int> go(int initial)
{
   return std::async( std::launch::async
                    , [this](int value)
                      {
                         for( auto func : queue_ )
                         {
                            std::future<int> f = std::async( func
                                                           , value);
                            value = f.get();
                         }
                         return value;
                      }
                    , initial );
}
```

Solution

```
std::future<int> go(int initial)
{
   return std::async( std::launch::async
                    , [this](int value)
                      {
                         for( auto func : queue_ )
                         {
                            auto f = std::async(func, value);
                            value = f.get();
                         }
                         return value;
                      }
                    , initial );
}
```

Continuations

When async task A is done, run task B with the result.

boost::future

```
auto f = boost::async( [](id_t id) -> std::string
                       {
                         return lookup_name(id)
                       }
                     , my_id
                     );

auto price = f.then( [](future<std::string> name) -> double
                     {
                       return get_price(name.get());
                     }
                   );

std::cout << "stock price: " << price.get() << std::endl;
```

boost::future

```
auto price = boost::async( [](id_t id) -> std::string
                           {
                             return lookup_name(id)
                           }
                         , my_id
                         )
                    .then( [](future<std::string> name)
                           {
                             return get_price(name.get());
                           }
                         );

std::cout << "stock price: " << price.get() << std::endl;
```

std::packaged_task

std::packaged_task:

- Packages a callable for later execution
- Associates shared state
- Get the **future** before invoking

std::packaged_task

```
template <class ResultType, class... ArgTypes>
struct std::packaged_task<ResultType(ArgTypes...)>
{
  packaged_task();

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

  packaged_task(packaged_task &&);
  packaged_task & operator=(packaged_task &&);

  // useful constructors .... (not shown)

  std::future<ResultType> get_future();

  void operator()(ArgTypes...);

  void make_ready_at_thread_exit(ArgTypes...);

  void reset();
};
```

std::packaged_task

```
auto task = std::packaged_task(std::bind( get_stock_price
                                        , "klac" ) );

// ...

auto f = task.get_future();

// launch somehow .....

// do some work

std::cout << "klac price: " << f.get() << std::endl;
```

std::thread

```
auto task = std::packaged_task(std::bind( get_stock_price
                                        , "klac" ) );

// ...

auto f = task.get_future();

auto t = std::thread(task);

// do some work

std::cout << "klac price: " << f.get() << std::endl;
```

`std::thread`

- Represents thread of execution (unless default constructed)
- Has a unique handle
- Can be joined to block on completion
- Can be detached from thread handle
- Can get the native handle
- Low level concept

`std::thread`

A *joinable* thread object that is destroyed (exits scope) will call **`std::terminate()`** !!!

std::thread

```
auto task = std::packaged_task(std::bind( get_stock_price
                                        , "klac" ) );

// ...

auto f = task.get_future();

auto t = std::thread(task);

// do some work

std::cout << "klac price: " << f.get() << std::endl;
t.join();
```

std::thread

```
auto task = std::packaged_task(std::bind( get_stock_price
                                        , "klac" ) );

// ...

auto f = task.get_future();

std::thread(task).detach();

// do some work

std::cout << "klac price: " << f.get() << std::endl;
```

std::thread

```
struct worker
{
  worker(){};
  ~worker()
  {
    if(work_thread.joinable())
    {
      work_thread.detach();
      //work_thread.join();
    }
  }

  // ....

  std::thread work_thread_;
};
```

std::thread

At construction, the execution begins.
You can pass additional arguments for the callable.

```
auto t = std::thread( [](int i)
                      {
                        std::cout << i << " is the number!\n";
                      }
                    , 42 );
```

Exceptions

If the function/functor that was passed to the `std::thread` constructor propagates an exception, `std::terminate()` will be called!

our::async

```
template <class F, class ...Args>
std::future<typename std::result_of<F(Args...)>::type
async(F f, Args... args) async
{
  std::promise<std::result_of<F(Args...)>::type> promise;
  auto future{promise.get_future()};

  std::thread thread( /* amazing stuff */ );

  thread.detach();
  return std::move(future);
}
```

our::async

```
std::thread thread(
    []( decltype(promise) && promise_
      , F f_, Arg &&... args_)
    {
      try
      {
        promise_.set_value(f_(std::forward<Args>(args_)...));
      }
      catch (...)
      {
        promise_.set_exception(std::current_exception());
      }
    }
    , std::move(promise)
    , f, std::forward<Args>(args)... );
```

Outline

- Asynchronous Overview
- `std::async`
- Synchronization

Mutual Exclusion

`include <mutex>` to gain mutex support.

- **`std::mutex`**
- **`std::timed_mutex`**
- **`std::recursive_mutex`**
- **`std::recursive_timed_mutex`**
- **`std::shared_mutex`**

std::mutex

```
void lock();
bool try_lock();
void unlock();
```

ciere.com

std::timed_mutex

```
void lock();
bool try_lock();
bool try_lock_for(const chrono::duration<R,P>& d);
bool try_lock_until(const chrono::time_point<C,D>& t);
void unlock();
```

std::recursive_mutex

```
void lock();
bool try_lock();
void unlock();
```

std::recursive_timed_mutex

```
void lock();
bool try_lock();
bool try_lock_for(const chrono::duration<R,P>& d);
bool try_lock_until(const chrono::time_point<C,D>& t);
void unlock();
```


ciere.com

std::shared_mutex

Note: **C++14**

Exclusive locking:

```
void lock();
bool try_lock();
bool try_lock_for(const chrono::duration<R,P>& d);
bool try_lock_until(const chrono::time_point<C,D>& t);
void unlock();
```

Shared locking:

```
void lock_shared();
bool try_lock_shared();
bool try_lock_shared_for(const chrono::duration<R,P>& d);
bool try_lock_shared_until(const chrono::time_point<C,D>& t);
void unlock_shared();
```

Lock Types

`include <mutex>` for the locks

- **`std::lock_guard`**
- **`std::unique_lock`**
- **`std::shared_lock`** – C++14

Using Locks

```
struct foo
{
  void add_one(int v)
  {
    std::lock_guard<std::mutex> lock(vec_mutex_);

    vec_.push_back(v);
  }

private:
  std::vector<int> vec_;
  std::mutex vec_mutex_;
};
```

Lock functions

Avoids deadlock despite argument order.

- `void lock(Lockable1&, Lockable2&, ...)`
- `void try_lock(Lockable1&, Lockable2&, ...)`

condition_variable

Allows one thread to wait for notification from another thread that a condition has become true.

condition_variable

```
std::condition_variable cond;
std::mutex mut;
bool data_ready;

void wait_for_data_to_process()
{
    std::unique_lock<std::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}
```

```
void prepare_data_for_processing()
{
    retrieve_data(); prepare_data();
    {
        std::lock_guard<std::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}
```

condition_variable

```
std::condition_variable cond;
std::mutex mut;
bool data_ready;
void wait_for_data_to_process()
{
    std::unique_lock<std::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}
```

```
void prepare_data_for_processing()
{
    retrieve_data(); prepare_data();
    {
        std::lock_guard<std::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}
```

condition_variable

```
std::condition_variable cond;
std::mutex mut;
bool data_ready;
void wait_for_data_to_process()
{
    std::unique_lock<std::mutex> lock(mut);
    while(!data_ready)
    {
       cond.wait(lock);
    }
    process_data();
}
```

```
void prepare_data_for_processing()
{
    retrieve_data(); prepare_data();
    {
        std::lock_guard<std::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}
```