BERKELEY LAB

Lawrence Berkeley National Laboratory

U.S. DEPARTMENT OF ENERGY

UNIVERSITY OF CALIFORNIA

2016 Berkeley C++ Summit

Bryce Adelstein Lelbach
Computer Architecture Group, Computing Research Division
balelbach@lbl.gov, /u/blelbach, @blelbach

WiFi:

SSID: `lbnl-visitor`

No password

Monday, October 17th

	Building 59, Room 3101	Building 59, Room 4102
9:00 to 9:55	Modern C++ for HPC *Bryce Adelstein Lelbach, LBNL*	
10:00 to 10:55	C++ @ Intel *Robert Geva, Intel*	
11:00 to 11:55	Programming Massively Parallel Hardware with Agency *Jared Hoberock, NVIDIA*	The Chombo AMR Framework: Refactoring using AMRStencil as Defensive Programming *Brian Van Straalen, LBNL*
12:00 to 1:00	Lunch	
1:00 to 1:35	Clang/LLVM: A Modern C++ Compiler with an HPC Twist *Hal Finkel, ANL*	Charm++: Motivation and Basic Concepts *Jonathan Lifflander, SNL*
1:40 to 2:15	UPC++: Asynchrony and Active Messages *John Bachan, LBNL*	CAF: C++ Actor Framework *Matthias Vallentin, UC Berkeley*
2:20 to 2:55	The RAJA Encapsulation for Architecture Portability *David Beckingsale, LLNL*	OpenCL, SYCL and Codeplay *Michael Wong, Codeplay*
	Break	
3:15 to 4:10	HPX: A C++ Standard Library for Parallelism & Concurrency *Hartmut Kaiser, LSU*	
4:15 to 5:10	Kokkos: Performance Portability & Productivity for C++ Applications *Carter Edwards, SNL*	
5:15 to 6:00	Grill the Committee Panel *ISO C++ Commitee Members*	
6:00 to 7:00		Reception

Tuesday, October 18th

	Building 59, Room 3101	Building 59, Room 3104
8:45 to 12:45	Kokkos Tutorial *Carter Edwards, Christian Trott*	ISO C++ SG14 Meeting on HPC
12:45 to 1:45	Lunch	
1:45 to 5:45	HPX Tutorial *STE||AR Group*	Charm++ Tutorial *Jonathan Lifflander*

STE||AR GROUP

BERKELEY LAB

In the event of a fire:

- Horn/strobes will activate.
- Evacuate immediately through the northeast or southeast exits. Main (3rd) floor occupants can use the center exit.
- If possible without delaying your evacuation, take personal belongings such as your wallet, keys and coat. You may not be able to return.
- Follow instructions of Building Emergency Team members who will direct you to the Assembly Area behind Building 70A.
- An additional Assembly Area is located outside Building 88, below Wang Hall and across Cyclotron Road.

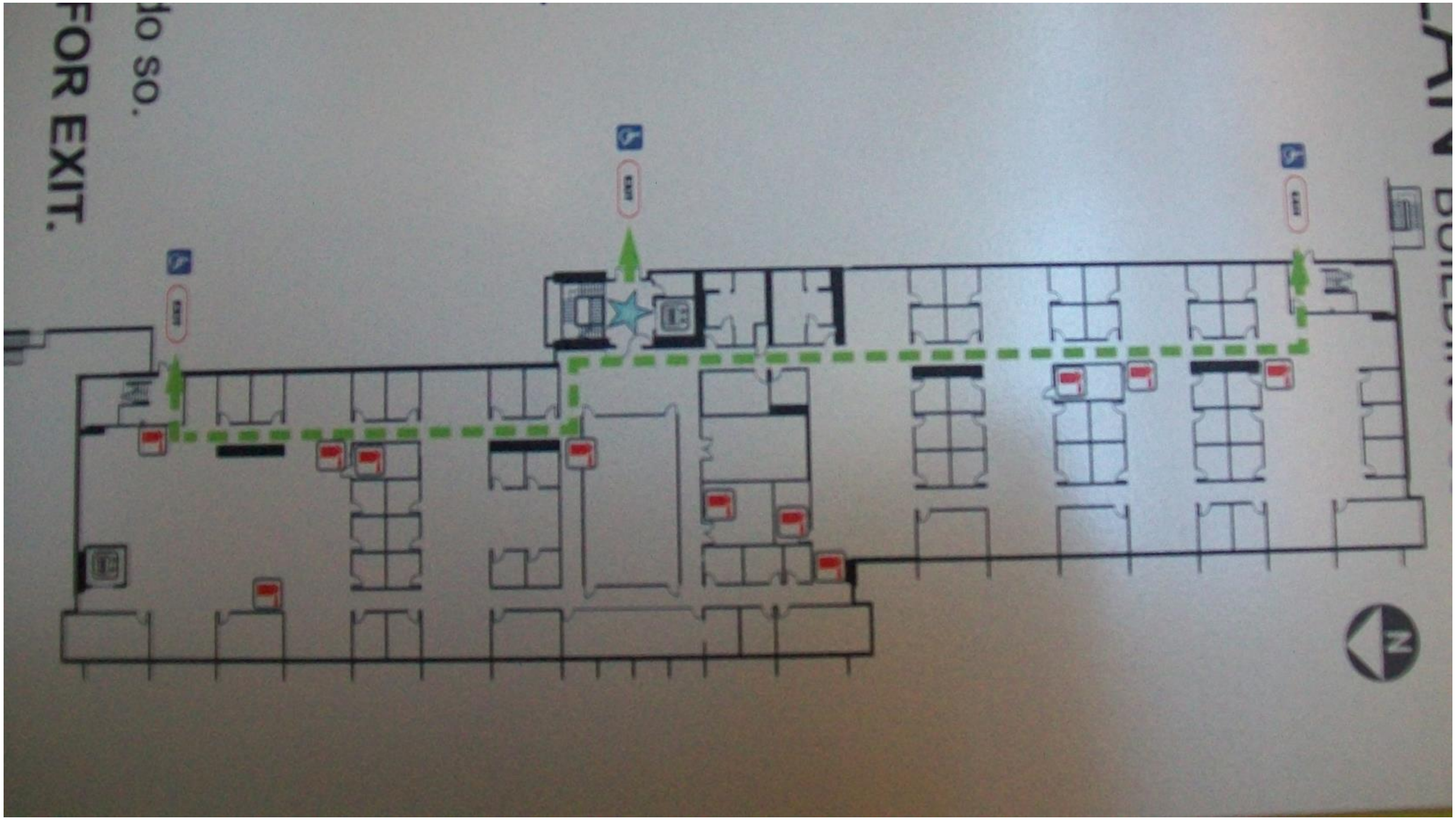
In the event of an earthquake, please:

- Duck, Cover, and Hold On to something solid until the shaking stops.
- If there is nothing to duck under, take shelter next to a solid wall, or squat down in place, and cover your head with your arms.
- In the machine room, move away from the computer racks, but stay within the blue “moat”.
- Once shaking has stopped, evacuate as just described.

Anyone with limited mobility:

Should evacuate to one of the Areas of Refuge located in the northeast and southeast stairwells (4^{th} floor) or outside the building (3^{rd} floor and compute levels).

Communicate with others exiting the building that you will remain in the Area of Refuge, and use the communication devices (4^{th} floor) if necessary.



do so.
FOR EXIT.

Restrooms are located on each floor near the center of the building, on the east wall. This is to your left as you enter the building.

Breakrooms are located on each floor, also near the center of the building, on the west wall. This is directly opposite the main entrance on the third floor. And directly above that, on the fourth floor.

Sustainable Berkeley Lab (sbl.lbl.gov) requests we sort trash into:

- Compostable
- Paper
- Landfill
- Other recycling

Please help us minimize waste going to the landfill by correctly disposing of food, beverage, and other waste.

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Modern C++ for HPC

2016 Berkeley C++ Summit

Bryce Adelstein Lelbach
Computer Architecture Group, Computing Research Division
balelbach@lbl.gov, /u/blelbach, @blelbach

STE||AR GROUP

Modern C++

==

Modern C++

!=

C++11

Modern C++

==

Use abstractions without fear

Only pay for what you use

Write for clarity and correctness first

Combine multiple programming paradigms

```c
typedef struct {
  int x, y;
} int_point2d;

void scale(int_point2d* op, const int_point2d* ip, int a) {
  op->x = p->x * a;
  op->y = p->y * a;
}

double magnitude(const int_point2d* p) {
  return sqrt(p->x * p->x + p->y * p->y);
}
```

STE||AR GROUP

BERKELEY LAB

```cpp
class int_point2d {
private:
  int x, y;

public:
  int_point2d();
  int_point2d(int, int);
  int_point2d(int_point2d const&);
  int_point2d& operator=(int_point2d const&);

  virtual ~int_point2d();

  void set_x(int);
  int const& get_x() const;

  void set_y(int);
  int const& get_y() const;

  // ...
```

STE||AR GROUP

BERKELEY LAB

```
class int_point2d {
private:
  int x, y;

public:
  // ...

  int_point2d scale(int a) const {
    int_point2d tmp(this->x * a, this->y * a);
    return tmp;
  }

  double magnitude() const {
    return std::sqrt(this->x * this->x + this->y * this->y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

C++

is a

multi-paradigm

programming

language

Imperative

Procedural

Object-Oriented

Generic

Functional

```
template <typename T>
struct point2d {


  T x, y;

  point2d scale(T a) const {
    return point2d{x * a, y * a};
  }

  double magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  point2d scale(T a) const {
    return point2d{x * a, y * a};
  }

  double magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  point2d scale(T a) const {
    return point2d{x * a, y * a};
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;







  point2d scale(T a) const {
    return point2d{x * a, y * a};
  }


  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    return point2d{x * a, y * a};
  }


  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [a] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [a] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```cpp
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [a] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [a] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [=] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    auto mulA = [&a] (T e) { return a * e; };
    return transform(mulA);
  }

  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    return transform([&a] (T e) { return a * e; });
  }


  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

```cpp
template <typename T>
struct point2d {
  static_assert(std::is_arithmetic_v<T>);

  T x, y;

  template <typename F>
  point2d transform(F f) const {
    return point2d{f(x),  f(y) };
  }

  point2d scale(T a) const {
    return transform([&a] (T e) { return a * e; });
  }


  auto magnitude() const {
    return is_zero(x, y) ? 0 : std::sqrt(x * x + y * y);
  }
};
```

STE||AR GROUP

BERKELEY LAB

The Modern C++ Prime Directive: Avoid C-style Abstractions

STE||AR GROUP

BERKELEY LAB

C-Style Abstraction	Replace With

`T* p = new T;`
`/* ... */`
`delete p;`
 	
`T* p = (T*)malloc(sizeof(T));`
`/* ... */`
`delete p;`
	`std::unique_ptr<T> p(new T);`

`T* da = new T[N];`
`/* ... */`
`delete[] da;`
 	
`T* da = (T*)malloc(sizeof(T)*N);`
`/* ... */`
`free(da);`
	`std::unique_ptr<T[N]> da(new T[N]);`
	`// OR`
	`std::vector<T> da(N);`

`T sa[n];`	`std::array<T, N> sa;`

STE||AR GROUP

BERKELEY LAB

```cpp
{
  std::unique_ptr<T> p(new T);

  // ...

} // p goes out of scope and is
  // destructed; its destructor
  // calls delete.
```

Resource

Acquisition

Is

Initialization

STE||AR GROUP

BERKELEY LAB

```
{
  std::unique_ptr<T> p(new T);

  // ...

} // p goes out of scope and is
  // destructed; its destructor
  // calls delete.


std::mutex mtx;
{
  std::lock_guard l(mtx);
  // mtx is now locked.

  // ...

} // l goes out of scope and is
  // destructed; its destructor
  // unlocks mtx.
```

Resource

Acquisition

Is

Initialization

```
template <typename G>
double* generate_vector(int N, G g);
```

```cpp
template <typename G>
double* generate_vector(int N, G g) {
  double* result = new double[N];
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}
```

STE||AR GROUP

BERKELEY LAB

```
template <typename G>
std::unique_ptr<double[]> generate_vector(int N, G g) {
  std::unique_ptr<double[]> result(new double[N]);
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}
```

```cpp
template <typename G>
std::vector<double> generate_vector(int N, G g) {
  std::vector<double> result;
  result.reserve(N);
  for (int i = 0; i < N; ++i)
    result.push_back(g(i));
  return result;
}
```

But Bryce…

What about the copies?

```
struct T {
  T();                  // Default constructor.
  T(const T& other); // Copy constructor.
};
```

```
struct T {
  T();                    // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T();
```

```cpp
struct T {
  T();                  // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T() {
  T local;
  // ...
  return local;
}
```

```
struct T {
  T();                  // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T();




T o(make_T());
```

```
struct T {
  T();                    // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.



T o(make_T()); // o is copied from the temporary.
```

```
struct T {
  T();                   // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.



T o(make_T()); // o is copied from the temporary.
               // The copy might be elided.
```

```
struct T {
  T();                   // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is copied from the temporary.
               // The copy might be elided.
```

```cpp
struct T {
  T();                    // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is copied from the temporary.
               // The copy might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied (the copy may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.

STE||AR GROUP

BERKELEY LAB

```cpp
struct T {
  T();                    // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is copied from the temporary.
               // The copy might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied (the copy may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.

STE||AR GROUP

BERKELEY LAB

```
struct T {
  T();                   // Default constructor.
  T(const T& other); // Copy constructor.
};


T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is copied from the temporary.
               // The copy might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied (the copy may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

```cpp
struct T {
  T();                    // Default constructor.
  T(const T& other); // Copy constructor.
  T(T&& other);       // Move constructor.
};

T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is moved from the temporary.
               // The move might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied or moved (the copy/move may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

STE||AR GROUP

BERKELEY LAB

```
struct T {
  T();                  // Default constructor.
  T(const T& other); // Copy constructor.
  T(T&& other);      // Move constructor.
};

T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is moved from the temporary.
               // The move might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied or moved (the copy/move may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

STE||AR GROUP

BERKELEY LAB

```cpp
template <typename G>
std::unique_ptr<double[]> generate_vector(int N, G g) {
  std::unique_ptr<double[]> result(new double[N]);
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}
```

```cpp
template <typename G>
std::unique_ptr<double[]> generate_vector(int N, G g) {
  std::unique_ptr<double[]> result(new double[N]);
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}
```

STE||AR GROUP

BERKELEY LAB

```
struct T {
  T();                   // Default constructor.
  T(const T& other); // Copy constructor.
  T(T&& other);      // Move constructor.
};

T make_T(); // Returns a temporary value.

T m;
T n(m); // n is copied from m.


T o(make_T()); // o is moved from the temporary.
               // The move might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied or moved (the copy/move may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

STE||AR GROUP

BERKELEY LAB

```
struct T {
  T();                   // Default constructor.
  T(const T& other);     // Copy constructor.
  T(T&& other);          // Move constructor.
};

T make_T(); // Returns a temporary value.

T m = // ...
T n(std::move(m)); // m is moved into n; m is in a valid
                   // but unspecified state.

T o(make_T()); // o is moved from the temporary.
               // The move might be elided.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied or moved (the copy/move may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

But Bryce…

What about the copies?

```cpp
template <typename G>
std::vector<double> generate_vector(int N, G g) {
  std::vector<double> result;
  result.reserve(N);
  for (int i = 0; i < N; ++i)
    result.push_back(g(i));
  return result;
}
```

But Bryce…

What about the copies?

What copies? Just moves!

Moves are cheap

```
struct T {
  T();                   // Default constructor.
  T(const T& other);     // Copy constructor.
  T(T&& other);          // Move constructor.
};

T make_T(); // Returns a temporary value.

T p;

f(p);                // Pass-by-value: the temporary is copied into v.
f(std::move(p));     // Pass-by-value: the temporary is moved into v.
f(make_T());         // Pass-by-value: temporary is move elided into v.
```

Signature	Parameter Passing Style
`f(T v);`	Pass by value. v is copied or moved (the copy/move may be elided).
`f(T& v);`	Pass by reference.
`f(const T& v);`	Pass by constant reference.
`f(T&& v);`	Pass by reference to temporary (AKA rvalue reference). Ownership of v can be taken.

STE||AR GROUP

BERKELEY LAB

```
auto f(const std::vector<double>& v) {
  std::vector<double> local(v); // Always copies
  local.push_back(3.14);
  return local; // May copy before C++17 if no elision happens.
                // Guaranteed to elide in C++17.
}

auto g(std::vector<double> v) { // May copy if passed a
                                // non-temporary or no
                                // elision happens.
    v.push_back(3.14);
    return v; // May copy before C++17 if no elision happens.
              // Guaranteed to elide in C++17.
}
```

STE||AR GROUP

BERKELEY LAB

Modern C++ moves computations to compile time

```cpp
template <typename T, int N, typename G>
std::array<T, N> generate_array(G g) {
  std::array<T, N> result{};
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}

double sine(double x);

const auto v = generate_array<double, 4096>(sine);
```

STE||AR GROUP

BERKELEY LAB

```
template <typename T, int N, typename G>
constexpr std::array<T, N> generate_array(G g) {
  std::array<T, N> result{};
  for (int i = 0; i < N; ++i)
    result[i] = g(i);
  return result;
}

constexpr double sine(double x);

constexpr auto v = generate_array<double, 4096>(sine);
```

STE||AR GROUP

BERKELEY LAB

```
template <typ
constexpr std
  std::array<
  for (int i
    result[i]
  return resu
}

constexpr dou

constexpr aut
```

```
.file             "wandbox.constexpr_sine_array.cc"
.section          .rodata
.type             _ZStL19piecewise_construct, @object
.size             _ZStL19piecewise_construct, 1
_ZStL19piecewise_construct:
.zero             1
.align 32
.type             _ZL1v, @object
.size             _ZL1v, 32768
_ZL1v:
.long             0
.long             0
.long             2401828618
.long             1072360788
.long             3938178374
.long             1072503030

          ... 8000 more lines of data section ...

.long             2252342392
.long             -1075907706
.long             2583472312
.long             -1074794969
.ident            "GCC: (GNU) 7.0.0 20161015 (experimental)"
.section          .note.GNU-stack,"",@progbits
```

STE||AR GROUP

BERKELEY LAB

C++11/14/17/2x is a parallel programming language:

- Parallel shared-memory abstract machine (C++11)
- Shared-memory memory model (C++11)
- Concurrency library (C++11, Concurrency TS v1)
- Parallel algorithms library (C++17)

C++17

Parallelism TS v1

Local Parallel Algorithms
<algorithm>, <numeric>

Basic Execution Policies
<execution_policy>

Concurrency TS v1

Local Dataflow Programming
<future>

Local Sync Primitives
<barrier>, <latch>

C++14

Local Sync Primitives
<shared_mutex>

C++11

Threading
<thread>

Local Async Programming
<future>

Multithreaded Execution Model
1.10 [intro.multithread]

Thread-Local Storage
3.7.2 [basic.stc.thread]

Local Sync Primitives
<mutex>, <shared_ptr>, …

Atomic Operations
<atomic>

```
namespace std {

template <typename Function, typename... Args>
auto async(Function f, Args&&... args);

}
```

```cpp
namespace std {

template <typename Function, typename... Args>
auto async(Function f, Args&&... args );

}

void   f();



auto ff = std::async(f);
```

```cpp
namespace std {

template <typename Function, typename... Args>
auto async(Function f, Args&&... args );

}

void   f();
int    g(int a);
double h(int a, double b, std::string const& c);

auto ff = std::async(f);
auto fg = std::async(g, 32);
auto fh = std::async(h, 32, 1.0, "hello");
```

STE||AR GROUP

BERKELEY LAB

```cpp
namespace std {

template <typename Function, typename... Args>
auto async(Function f, Args&&... args );

}

void   f();
int    g(int a);
double h(int a, double b, std::string const& c);

std::future<void>   ff = std::async(f);
std::future<int>    fg = std::async(g, 32);
std::future<double> fh = std::async(h, 32, 1.0, "hello");
```

```cpp
unsigned fibonacci(unsigned n)
{
    // If we know the answer, we're done.
    if (n < 2) return n;

    // Asynchronously calculate one of the sub-terms.
    future<unsigned> f = async(fibonacci, n - 1);

    // Synchronously calculate the other sub-term.
    unsigned r = fibonacci(n - 2);

    // Wait for the future and calculate the result.
    return f.get() + r;
}
```

```
R                  f(/* args */);
// Call the function f synchronously.

std::future<Q> g = // ...

std::future<R> h = g.then(f);
// Attach a continuation function, f, to future g.
```

STE||AR GROUP

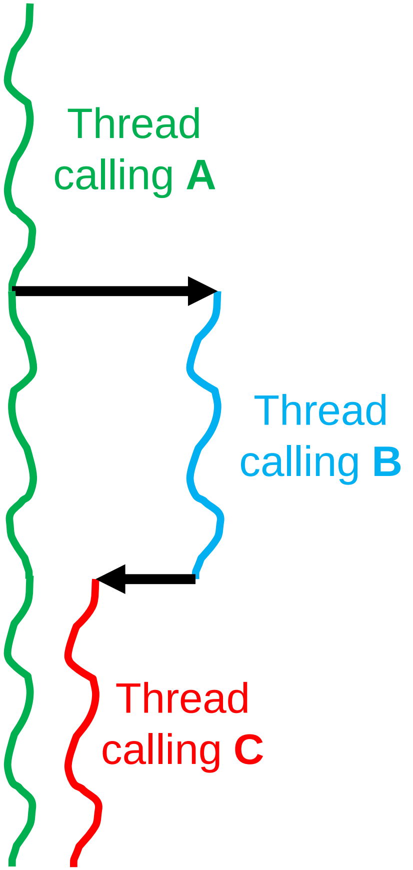
BERKELEY LAB

```
int B();

// Depends on B.
double C(std::future<int> b);


std::future<double> A() {
  std::future<int> b = std::async(B);

  return b.then(C);
}
```



```cpp
std::future<T> fa;
std::future<U> fb;
std::future<V> fc;

auto fabc = std::when_all(fa, fb, fc);
// fabc is ready when fa, fb and fc are all ready.

auto fabc = std::when_any(fa, fb, fc);
// fabc is ready when at least one of fa, fb and fc is ready.
```

STE||AR GROUP

BERKELEY LAB

```
std::vector<T> x = // ...

std::for_each(x.begin(), x.end(), process);
```

```
std::vector<T> x = // ...

std::for_each(std::par_unseq, x.begin(), x.end(), process);
```

```
std::vector<T> x = // ...

#pragma omp parallel for simd
for (std::size_t i = 0; i < x.size(); ++i)
    process(x[i]);
```

```
std::vector<T> x = // ...

std::for_each(std::par_unseq, x.begin(), x.end(), process);
```

Standard execution policies:

- `std::seq` – operations are indeterminately sequenced in the **calling thread**.
- `std::par` – operations are indeterminately sequenced with respect to each other within the **same thread**.
- `std::par_unseq` – operations are unsequenced with respect to each other and possibly **interleaved**.

Standard execution policies:

- `std::seq` – serial.
- `std::par` – task-parallel.
- `std::par_unseq` – task-parallel and vectorized.

```
std::vector<double> x = // ...

double norm =
 std::sqrt(std::transform_reduce(std::par, x.begin(), x.end(), 0.0));

std::vector<double> y = // ...
std::vector<double> z = // ...

double dot_product =
 std::transform_reduce(std::par, y.begin(), y.end(), z.begin(), 0.0);
```

STE||AR GROUP

BERKELEY LAB

C++17

Concurrency TS v1

Local Dataflow Programming
<future>

Local Sync Primitives
<barrier>, <latch>

Parallelism TS v1

Local Parallel Algorithms
<algorithm>, <numeric>

Basic Execution Policies
<execution>

C++14

Local Sync Primitives
<shared_mutex>

C++11

Threading
<thread>

Local Async Programming
<future>

Multithreaded Execution Model
1.10 [intro.multithread]

Thread-Local Storage
3.7.2 [basic.stc.thread]

Local Sync Primitives
<mutex>, <shared_ptr>, …

Atomic Operations
<atomic>

Reflection TS

Static Type Reflection

Networking TS

Asynchronous Networking
<net>

Concurrency TS v2

Local Shared Memory
<shmem>

Local Concurrent Data Structures
<atomic_queue>, …

Executors
<execution_policy>

Parallelism TS v2

Local Async Parallel Algorithms
<algorithm>, <numeric>

New Algorithms
<algorithm>, <numeric>

Library Funds v3

Multi-Dimensional Arrays
<mdspan>, <mdarray>

Coroutines TS

Coroutines

Concurrency TS v1

Local Dataflow Programming
<future>

Local Sync Primitives
<barrier>, <latch>

Distributed TS

Memory Spaces
<accessor>, ...

Distributed Dataflow Programming
<future>

Distributed Parallel Algorithms
<algorithm>, <numeric>

Distributed Execution and Memory Model

Distributed Executors
<execution_policy>

Distributed Data Structures
<global_vector>, …

Distributed Sync Primitives
<global_mutex>, …

Distributed Atomic Operations
<global_atomic>

Reflection TS

Static Type Reflection

Networking TS

Asynchronous Networking
<net>

Concurrency TS v2

Local Shared Memory
<shmem>

Local Concurrent Data Structures
<atomic_queue>, …

Executors
<execution_policy>

Parallelism TS v2

Local Async Parallel Algorithms
<algorithm>, <numeric>

New Algorithms
<algorithm>, <numeric>

Library Funds v3

Multi-Dimensional Arrays
<mdspan>, <mdarray>

Coroutines TS

Coroutines

Concurrency TS v1

Local Dataflow Programming
<future>

Local Sync Primitives
<barrier>, <latch>

STE||AR GROUP

BERKELEY LAB