Thinking Functionally in C++

BRIAN RUTH

Cppcon
The C++ Conference

2023

October 01 - 06

Thinking Functionally in C++

BRIAN RUTH
Senior Software Engineer - Garmin
he/him

Cppcon
The C++ Conference

2023
October 01 - 06

Prelude:

Setting Expectations

Setting Expectations

The goal of this talk is to show you different ways of thinking about a problem

- You do not need to be familiar with functional programming
- We will not cover any advanced FP or mathematical concepts
- Some examples may not be best practices
- Functional concepts will be interleaved with OO and imperative code

If you want a deeply functional topic:

Ben Deane
Applicative: The Forgotten Functional Pattern
Wednesday, October 4 • 14:00 - 15:00

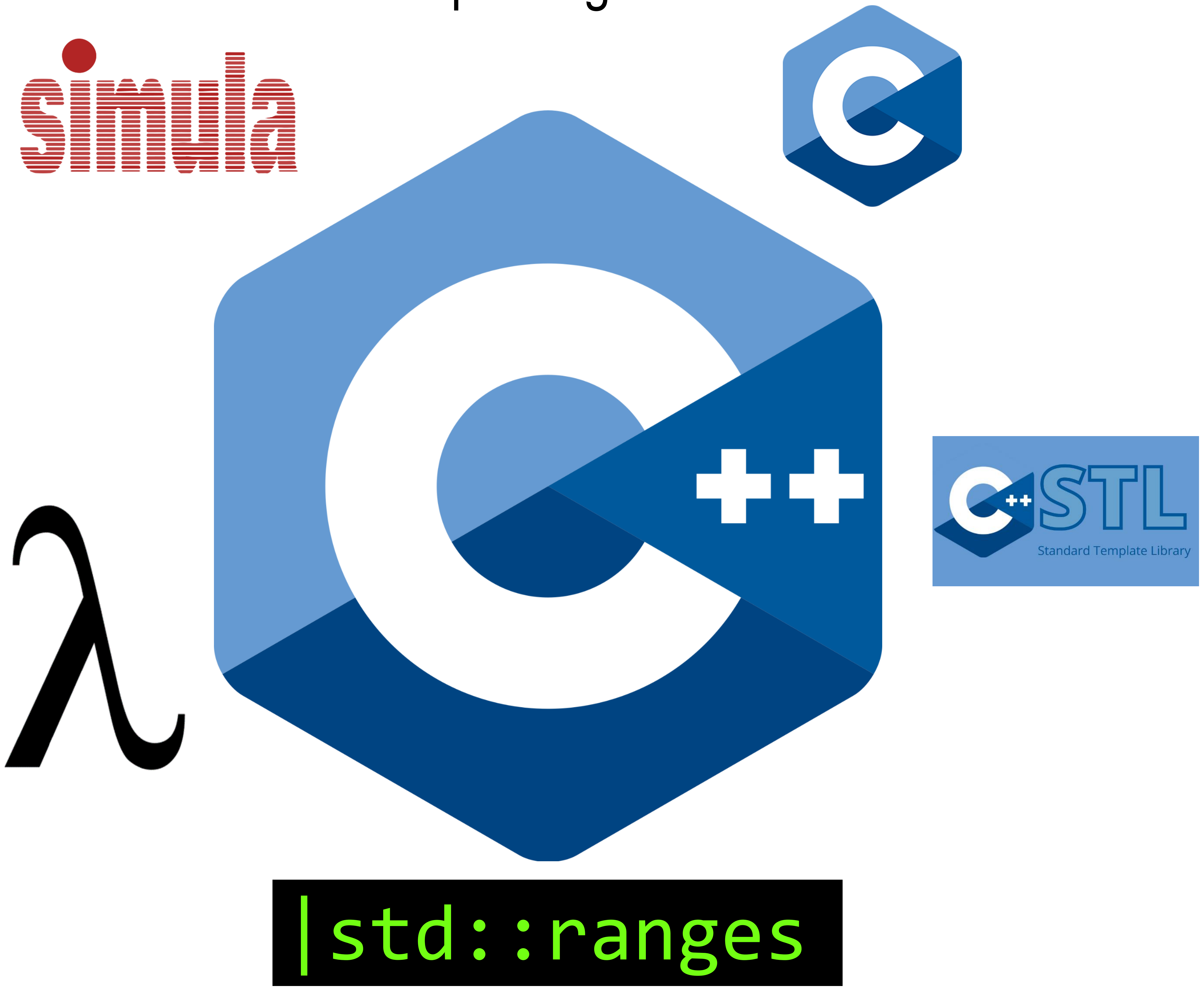
Introduction:

C++: A Multi-paradigm Language

C++ has something for everyone

For the purposes of this talk, we will limit ourselves to 3 different paradigms

- Imperative or procedural
  - Roots with C
- Object Oriented
  - Roots with Simula
- Functional
  - Standard Template Library (STL)
  - Lambdas
  - Ranges



C++ has something for everyone: Imperative

```
int main() {
 auto fh = fopen("script.txt", "r");
 char line[255];
 const char* ForbiddenWord[] = {"it", "It", "IT"};
 int numForbiddenWords = 0;
 while(fgets(line, sizeof line, fh) != nullptr){
  const char* delims = " \n\r,;!-?\"";
  auto* nextWord = strtok(&line[0], delims);
  while(nextWord != nullptr) {
   for(int i = 0; i < 3; ++i) {
    if(strcmp(ForbiddenWord[i], nextWord) == 0) {
     ++numForbiddenWords;
     break;
    }
   }
   nextWord = strtok(nullptr, delims);
  }
 }
 fclose(fh);
 printf("Number of forbidden words: %d\n", numForbiddenWords);
}
```

C++ has something for everyone: Object Oriented

```
int main() {
 auto fh = std::ifstream("script.txt");
 std::stringstream text;
 text << fh.rdbuf();

 int numForbiddenWords = 0;
 std::string nextWord;
 CaseIgnoreComparer comp;
 while (text >> nextWord) {
   if(comp.Equal(nextWord, "it")) {
     ++numForbiddenWords;
   }
 }

 Console c;
 c.Print("Number of forbidden words: ");
 c.Print(numForbiddenWords);
 c.Print("\n");
}
```

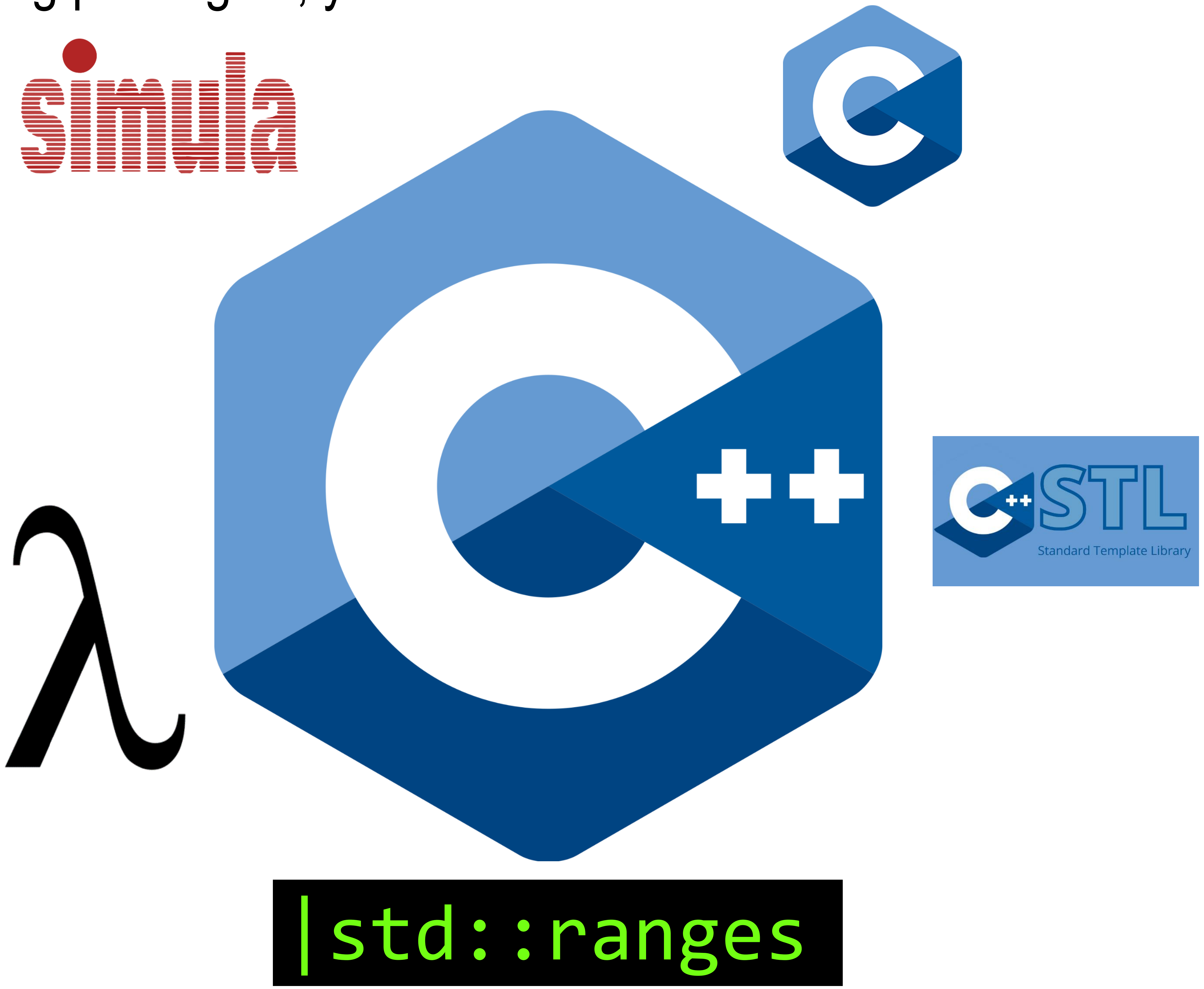
C++ has something for everyone: Functional

```
int main() {
 const auto words = ParseWordsFromFile("script.txt");
 const auto noPunctuationWords = RemovePunctuationFromWords(words);
 const auto upperCaseWords = MakeUpperCase(noPunctuationWords);
 const auto numForbiddenWords = std::ranges::count_if(upperCaseWords,
  [](const auto& word) {
   return word == "IT";
 });

 std::cout << "Number of forbidden words: " << numForbiddenWords << "\n";
}
```

C++ has something for everyone

Because C++ supports all these programming paradigms, you can mix and match them in the same program

```
int main() {
 auto fh = fopen("script.txt", "r");
 const auto words = ParseWordsFromFile(fh);
 const auto numForbiddenWords =
  std::ranges::count_if(words,
   [](const auto& word) {
    CaseIgnoreComparer comp;
    return comp.Equal(word,"it");
   });

 fclose(fh);
 printf("Number of forbidden words: %d\n",
  numForbiddenWords);
}
```



Part 1:

Identifying code functionally

Functional Code Categories

- **Actions**
  - Depend on when or how many times they are called. Observable changes occur.
    - `puts("hello world");`
    - `LaunchRocket();`
    - `x = 4;`
- **Calculations**
  - Depend only on their inputs and not when or how often they are called. Calling them with the same inputs always results in the same output. No observable changes occur.
    - `std::plus(2,4)`
    - `IsEven(integer);`
    - `std::all_of(begin(integers), end(integers), IsEven)`
- **Data**
  - Unchanging records of events. Used as inputs to calculations and actions. Record the results of calculations and actions.
    - `{2, 4, 6, 8}`
    - `struct Name { std::string First; std::string Last; };`
    - `enum struct Color { Green,Orange, Purple };`

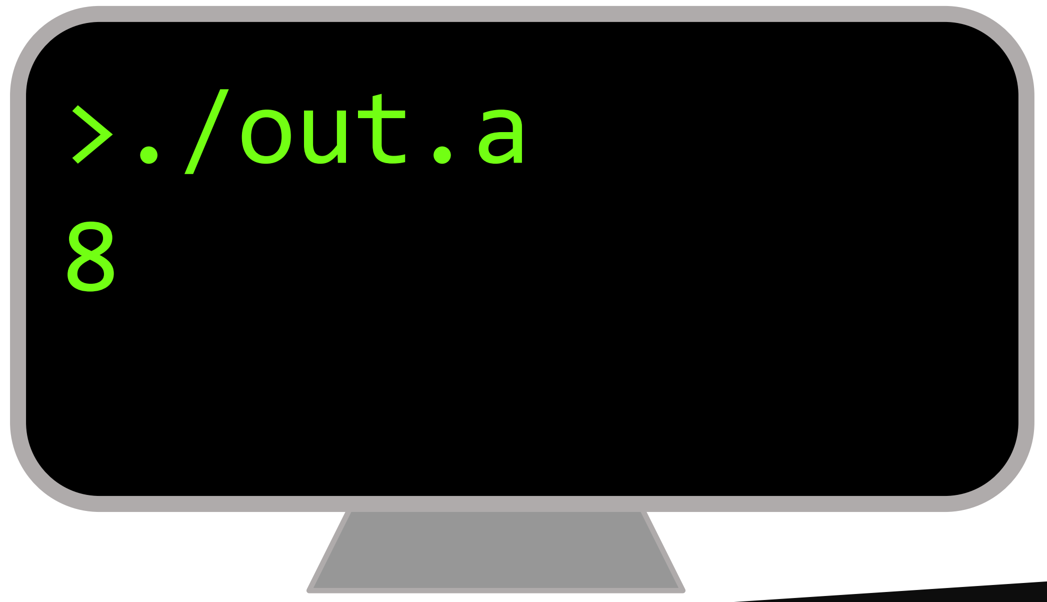
Why are these categories important?

- **Actions**
  - Allow input to programs that is unknown when the program was written
  - Performing an action has consequences
  - Affect how a program executes
- **Calculations**
  - Reliable, a calculation always produces the same resulting data when given the same input data.
  - Encapsulated, has no effect outside of itself
  - Thread safe, since it is entirely self contained, no ordering or locking is necessary
- **Data**
  - Fundamental building block
  - Immutable, data does not change
  - Transparent, you can look at data and see what it is
  - Open to interpretation, data can mean different things to different components without changing value
  - Used by calculations and actions to communicate with other calculations and actions

Is a variable Data?

A variable is shorthand for referencing the result of a Calculation or Action.

```
                2 + 2;                4
          int X = 2 + 2;           X  4
             X = X + X;
 X = (2 + 2) + (2 + 2);
             X = 4 + 4;               8
      printf("%d", X);
printf("%d", (4 + 4) );
 printf("%d", 8 );
```



```
>./out.a
8
```

Data: It's all about context

Baker
Recipe

Clerk
Inventory Item

Accountant
Profit Margin

Nutritionist
Ingredients

Customer
Expense

Monster
Food

Breaking down a problem

Barb's bakery wants to give their employees a fun gift on their birthday. They found a company that offers discounted gift cards to local restaurants that rotate on a quarterly basis. Because of the awesome discount, the gift card value is going to be $10 for each year of service. To make it even more personal, they will print out a birthday card with the gift card options that will be put on their desk at the beginning of the day!

Breaking down a problem

Get a list of all employees whose birthday is this week

Barb's Bakery

Get the current gift card options

Determine the amount for the gift card

Print out birthday card

Barb's bakery wants to give their employees a fun gift on their birthday. They found a company that offers discounted gift cards to local restaurants that rotate on a quarterly basis. Because of the awesome discount, the gift card value is going to be $10 for each year of service. To make it even more personal, they will print out a birthday card with the gift card options that will be put on their desk at the beginning of the day!

Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Get the current gift card options

Determine the amount for the gift card

Print out birthday card

Barb's bakery wants to give their employees a fun gift on their birthday. They found a company that offers discounted gift cards to local restaurants that rotate on a quarterly basis. Because of the awesome discount, the gift card value is going to be $10 for each year of service. To make it even more personal, they will print out a birthday card with the gift card options that will be put on their desk at the beginning of the day!

Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Get the current gift card options

Determine the amount for the gift card

Print out birthday card

Action: The list of employees changes as people leave or are hired.

Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Get the current gift card options

Determine the amount for the gift card

Print out birthday card

Action: The list of employees changes as people leave or are hired.

Action: Depending on what deals they can get, the gift card options may change

Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Action: The list of employees changes as people leave or are hired.

Get the current gift card options

Action: Depending on what deals they can get, the gift card options may change

Determine the amount for the gift card

Calculation: Given the hire date and today's date, return a dollar amount.

Print out birthday card

Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Action: The list of employees changes as people leave or are hired.

Get the current gift card options

Action: Depending on what deals they can get, the gift card options may change

Determine the amount for the gift card

Calculation: Given the hire date and today's date, return a dollar amount.

Print out birthday card

Action: You don't want to print the card twice or print it after their birthday.

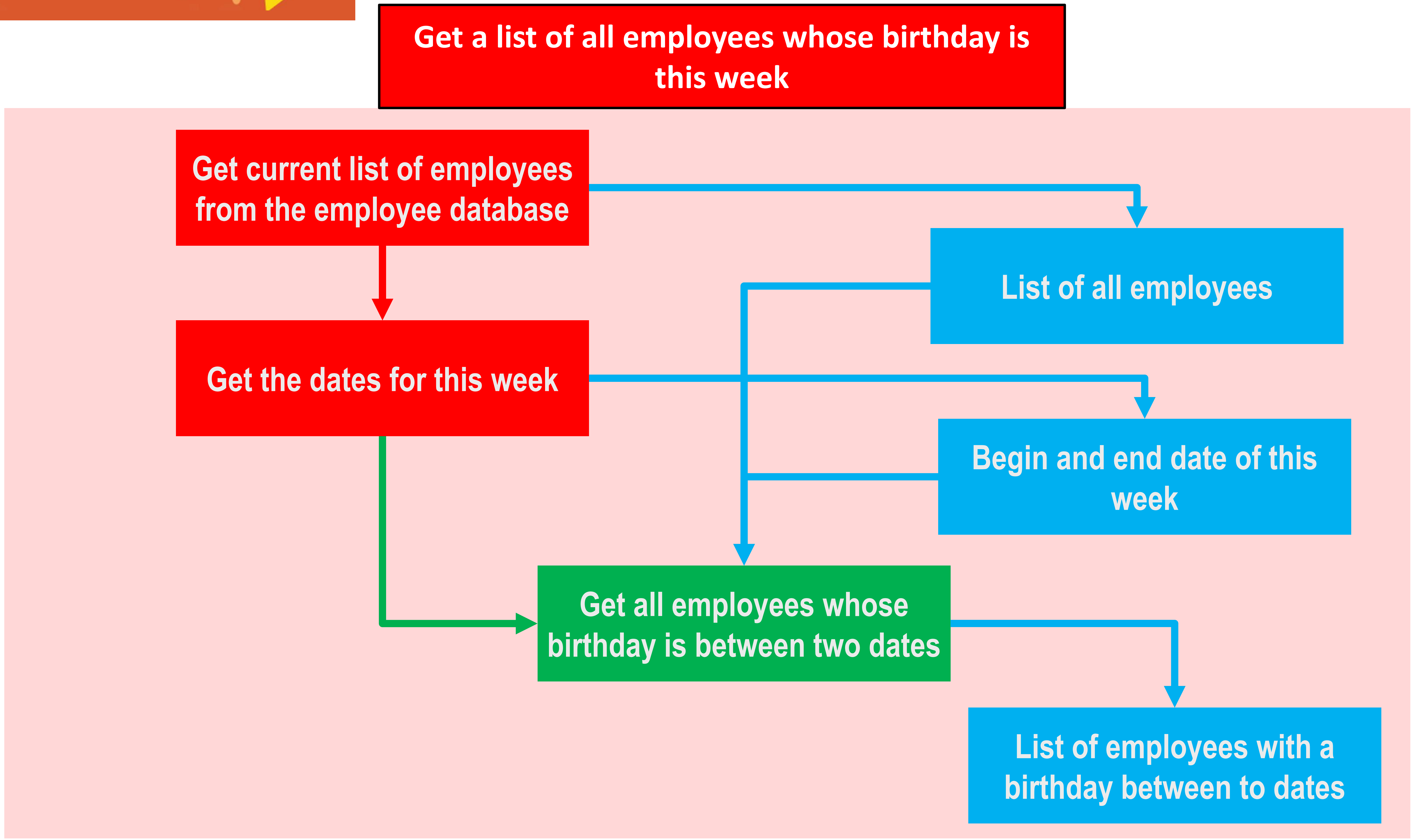
Breaking down a problem

Barb's Bakery

Get a list of all employees whose birthday is this week

Breaking down a problem

Barb's Bakery



Get a list of all employees whose birthday is this week

Get current list of employees from the employee database

List of all employees

Get the dates for this week

Begin and end date of this week

Get all employees whose birthday is between two dates

List of employees with a birthday between to dates

A possible implementation

```
std::vector<Employee> GetBirthdayEmployeesThisWeek() {
 const auto allEmployees = GetCurrentEmployees();
 const auto dateRange = GetThisWeekDateRange();
std::vector<Employee> birthdayEmployees;
 std::copy_if(begin(allEmployees), end(allEmployees),
   std::back_inserter(birthdayEmployees), [dateRange](auto& employee) {
     return employee.birthday >= range.firstDay &&
            employee.birthday <= range.lastDay;
   }
 );
 return birthdayEmployees;
}

int main() {
 const auto birthdayPeeps = GetBirthdayEmployeesThisWeek();
 ...
}
```

A possible implementation

```
std::vector<Employee> GetBirthdayEmployeesThisWeek() {
 const auto allEmployees = GetCurrentEmployees();
 const auto dateRange = GetThisWeekDateRange();
std::vector<Employee> birthdayEmployees;
 std::copy_if(begin(allEmployees), end(allEmployees),
   std::back_inserter(birthdayEmployees), [dateRange](auto& employee) {
     return employee.birthday >= range.firstDay &&
            employee.birthday <= range.lastDay;
   }
 );
 return birthdayEmployees;
}

int main() {
 const auto birthdayPeeps = GetBirthdayEmployeesThisWeek();
 ...
}
```

A possible implementation

```
std::vector<Employee> GetBirthdayEmployeesThisWeek() {
 const auto allEmployees = GetCurrentEmployees();
 const auto dateRange = GetThisWeekDateRange();
std::vector<Employee> birthdayEmployees;
 std::copy_if(begin(allEmployees), end(allEmployees),
   std::back_inserter(birthdayEmployees), [dateRange](auto& employee) {
     return employee.birthday >= range.firstDay &&
            employee.birthday <= range.lastDay;
   }
 );
 return birthdayEmployees;
}

int main() {
 const auto birthdayPeeps = GetBirthdayEmployeesThisWeek();
 ...
}
```

A possible implementation

```
std::vector<Employee> GetBirthdayEmployeesThisWeek() {
 const auto allEmployees = GetCurrentEmployees();
 const auto dateRange = GetThisWeekDateRange();
std::vector<Employee> birthdayEmployees;
 std::copy_if(begin(allEmployees), end(allEmployees),
   std::back_inserter(birthdayEmployees), [dateRange](auto& employee) {
     return employee.birthday >= range.firstDay &&
            employee.birthday <= range.lastDay;
   }
 );
 return birthdayEmployees;
}

int main() {
 const auto birthdayPeeps = GetBirthdayEmployeesThisWeek();
 ...
}
```

A possible implementation

```
std::vector<Employee> GetBirthdayEmployeesThisWeek() {
 const auto allEmployees = GetCurrentEmployees();
 const auto dateRange = GetThisWeekDateRange();
std::vector<Employee> birthdayEmployees;
 std::copy_if(begin(allEmployees), end(allEmployees),
   std::back_inserter(birthdayEmployees), [dateRange](auto& employee) {
     return employee.birthday >= range.firstDay &&
            employee.birthday <= range.lastDay;
   }
 );
 return birthdayEmployees;
}

int main() {
 const auto birthdayPeeps = GetBirthdayEmployeesThisWeek();
 ...
}
```

Isolate the Actions

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Isolate the Actions

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Isolate the Actions

```cpp
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Isolate the Actions

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Isolate the Actions

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Isolate the Actions

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Create reusable calculations

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
const auto longTenureEmployees = FilterEmployees(allEmployees, longTenureFilter);
const auto longTenureBirthday = FilterEmployees(longTenureEmployees, birthdayFilter);
```

Part 2:

Functions as Data

Passing functions to functions

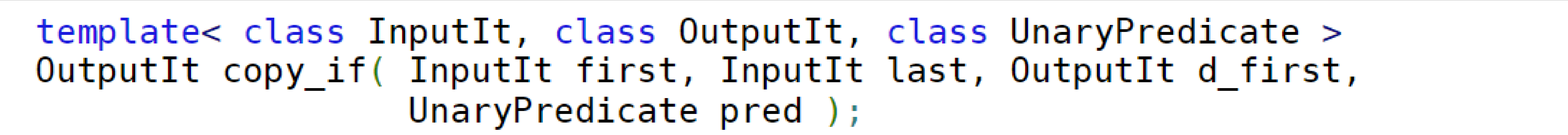
Like any other data, functions can be stored in a variable

```
bool LessThan5(int a) { return a < 5; }

bool(* lt5)(int) = LessThan5;
```

That variable can then be used as the input to another function

```
int IncrementIf(int value, bool(*condition)(int)) {
 return condition(value) ? ++value : value;
}

int result = IncrementIf(7, lt5);
```

The Standard Template Library algorithms are built on the fact that you can treat functions as data.



```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last, OutputIt d_first,
                  UnaryPredicate pred );
```

```
std::copy_if(begin(Numbers), end(Numbers), begin(SmallNumbers), lt5);
```

Returning functions from functions

Like any other data, functions can be returned from a function and stored in a variable

```
auto BuildLessThanCheck(int maxValue) {
 return [maxValue](int value) { return value < maxValue; };
}

auto LessThan7 = BuildLessThanCheck(7);
```

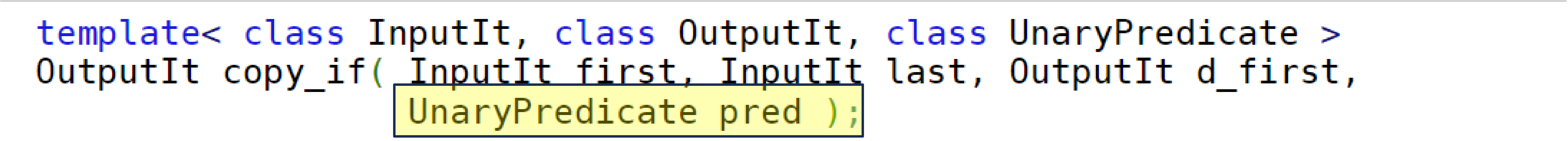
That variable can then be used as the input to another function, or called directly

```
if(LessThan7(Numbers[0])) {
 auto firstBigNumber = std::find_if_not(begin(Numbers), end(Numbers), LessThan7);
 ...
}
```

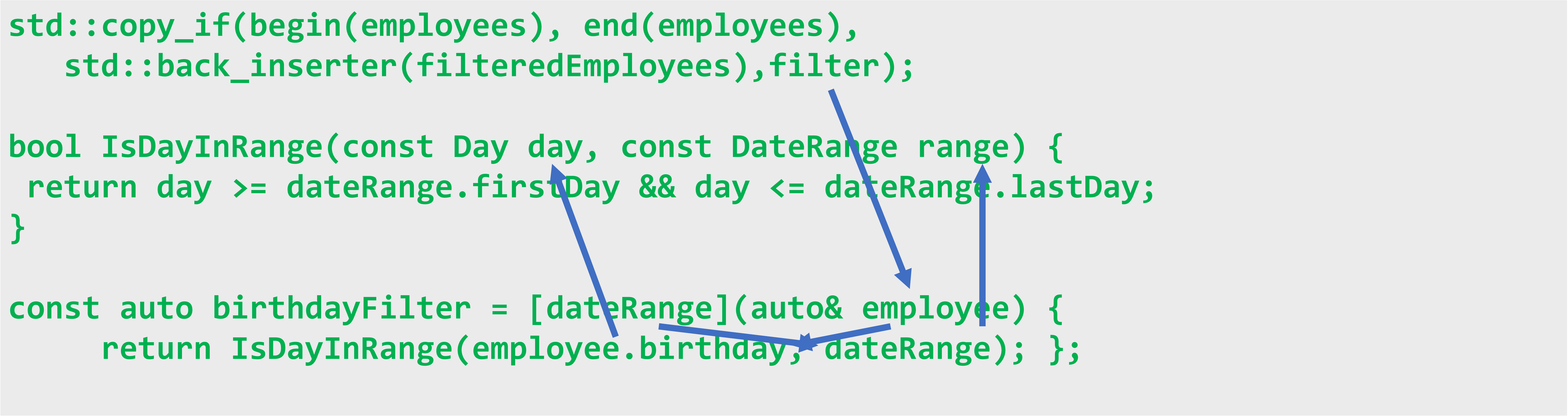
Adapt functions to algorithms

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

int main() {
 const auto dateRange = GetThisWeekDateRange();
 const auto allEmployees = GetCurrentEmployees();
 const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };

 const auto birthdayEmployees = FilterEmployees(allEmployees, birthdayFilter);
 ...
}
```

Adapt functions to algorithms



```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last, OutputIt d_first,
                  UnaryPredicate pred );
```



```
std::copy_if(begin(employees), end(employees),
   std::back_inserter(filteredEmployees),filter);

bool IsDayInRange(const Day day, const DateRange range) {
 return day >= dateRange.firstDay && day <= dateRange.lastDay;
}

const auto birthdayFilter = [dateRange](auto& employee) {
     return IsDayInRange(employee.birthday, dateRange); };
```

This process is known as currying

Bakery Automation

Barb's Bakery

A few years ago, Barb's bakery invested in some ovens that can automate parts of the baking process. The API to control the ovens is written in C, so they hired a contractor to write a controller that works with their recipes. Now that contractor has retired, and they've run into some issues. Sometimes, the oven didn't turn off at the end of a recipe. The manufacturer is also constantly coming out with new features that they want to incorporate into their process.

The existing code...

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);
```

The existing code...

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);
```

```
int main() {
    auto oven1 = oven_reserve_next_available();
    oven_turn_on(oven1);
    oven_set_temperature(oven1, 375);
    int temperature = 0;
    if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
      return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
       return -1;
      }
    }


    ....
}
```

Not turning off? RAII to the rescue!

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);
```

```
int main() {
    auto oven1 = oven_reserve_next_available();
    oven_turn_on(oven1);
    oven_set_temperature(oven1, 375);
    int temperature = 0;
    if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
      return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
       return -1;
      }
    }


    ....
}
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { TurnOff(); oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

Not turning off? RAII to the rescue!

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);
```

```
int main() {
    auto oven1 = oven_reserve_next_available();
    oven_turn_on(oven1);
    oven_set_temperature(oven1, 375);
    int temperature = 0;
    if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
      return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
       return -1;
      }
    }


    ....
}
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { TurnOff(); oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

```
auto oven1 = Oven(oven_reserve_next_available());
oven1.TurnOn();
oven1.SetTemperature(375);
```

Not turning off? RAII to the rescue!

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { TurnOff(); oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

```
int main() {
    auto oven1 = oven_reserve_next_available();
    oven_turn_on(oven1);
    oven_set_temperature(oven1, 375);
    int temperature = 0;
    if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
      return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if( oven_get_temperature(oven1, &temperature) != OVEN_OK) {
       return -1;
      }
    }


    ....
}
```

```
int main() {
    auto oven1 = Oven(oven_reserve_next_available());
    oven1.TurnOn();
    oven1.SetTemperature(375);

    int temperature = 0;
    if(oven1.GetTemperature(temperature) != OVEN_OK) {
     return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if(oven1.GetTemperature(temperature) != OVEN_OK) {
       return -1;
      }
    }


    ....
}
```

Supporting new functions

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes
```

Supporting new functions

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 OVEN_ERR SetTime(int numMinutes) {
   return oven_set_time(mHandle, numMinutes);
  }

 OVEN_ERR GetTimeRemaining(int& minutesRemaining) {
   return oven_get_temperature(mHandle, *minutesRemaining);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

Supporting new functions

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes
```

```
if(oven1.SetTime(60) != OVEN_OK) {
 return -1;
}

int minutes_left = 0;
if(oven1.GetTimeRemaining(minutes_left) != OVEN_OK) {
  return -1;
}
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 OVEN_ERR SetTime(int numMinutes) {
   return oven_set_time(mHandle, numMinutes);
  }

 OVEN_ERR GetTimeRemaining(int& minutesRemaining) {
   return oven_get_temperature(mHandle, *minutesRemaining);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

Supporting new functions

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes
```

```
int main() {
    auto oven1 = Oven(oven_reserve_next_available());
    oven1.TurnOn();
    oven1.SetTemperature(375);

    int temperature = 0;
    if(oven1.GetTemperature(temperature) != OVEN_OK) {
     return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if(oven1.GetTemperature(temperature) != OVEN_OK) {
       return -1;
      }
    }

    if(oven1.SetTime(60) != OVEN_OK) {
     return -1;
    }

    int minutes_left = 0;
    if(oven1.GetTimeRemaining(minutes_left) != OVEN_OK) {
      return -1;
    }

    ....
}
```

```
class Oven {
 public:
  Oven(OVEN_HANDLE handle): mHandle(handle){}
  ~Oven() { oven_release(mHandle); }

  OVEN_ERR TurnOn() { return oven_turn_on(mHandle); }
  OVEN_ERR TurnOff() { return oven_turn_off(mHandle); }
  OVEN_ERR SetTemperature(int temperature) {
   return oven_set_temperature(mHandle, temperature);
  }
  OVEN_ERR GetTemperature(int& temperature) {
   return oven_get_temperature(mHandle, *temperature);
  }

 OVEN_ERR SetTime(int numMinutes) {
   return oven_set_time(mHandle, numMinutes);
  }

 OVEN_ERR GetTimeRemaining(int& minutesRemaining) {
   return oven_get_temperature(mHandle, *minutesRemaining);
  }

 private:
  OVEN_HANDLE mHandle;
};
```

Lambdas are objects...

```
    auto Lambda = [value](int other){
      return value < other;
    };
```

Lambdas are objects… Objects are created from classes

```
    auto Lambda = [value](int other){
      return value < other;
    };
```

```
class UnspeakableLambda {
 public:
  UnspeakableLambda(int aValue) : value(aValue){};
  ~UnspeakableLambda() = default;

  bool operator()(int other) const { return value < other; }
 private:
  const int value;
}

UnspeakableLambda l(value);
l(4);
```

Use a lambda to store the handle

```
auto Oven =[](auto handle) {



           };
```

Use a lambda to store the handle

```
auto Oven =[](auto handle) {
             return [h = handle]


           };
```

Use a lambda to store the handle

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args)


           };
```

Use a lambda to store the handle

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
                    };
           };
```

Call C API through the lambda

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes_remaining);
```

Call C API through the lambda

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes_remaining);
```

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
                    };
           };
```

Call C API through the lambda

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes_remaining);
```

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
                    };
           };
```

```
auto oven1 = Oven(oven_reserve_next_available());
oven1(oven_turn_on);
oven1(oven_set_temperature, 375);
```

Call C API through the lambda

```
int main() {
    auto oven1 = Oven(oven_reserve_next_available());
    oven1(oven_turn_on);
    oven1(oven_set_temperature, 375);

    int temperature = 0;
    if(oven1(oven_get_temperature, &temperature) != OVEN_OK) {
     return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if(oven1(oven_get_temperature, &temperature) != OVEN_OK) {
        return -1;
      }
    }

    if(oven1(oven_set_time, 60) != OVEN_OK) {
     return -1;
    }

    int minutes_left = 0;
    if(oven1(oven_get_remaining_time, &minutes_left) != OVEN_OK) {
      return -1;
    }

    ....
}
```

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
                    };
           };
```

Lambdas are classes...

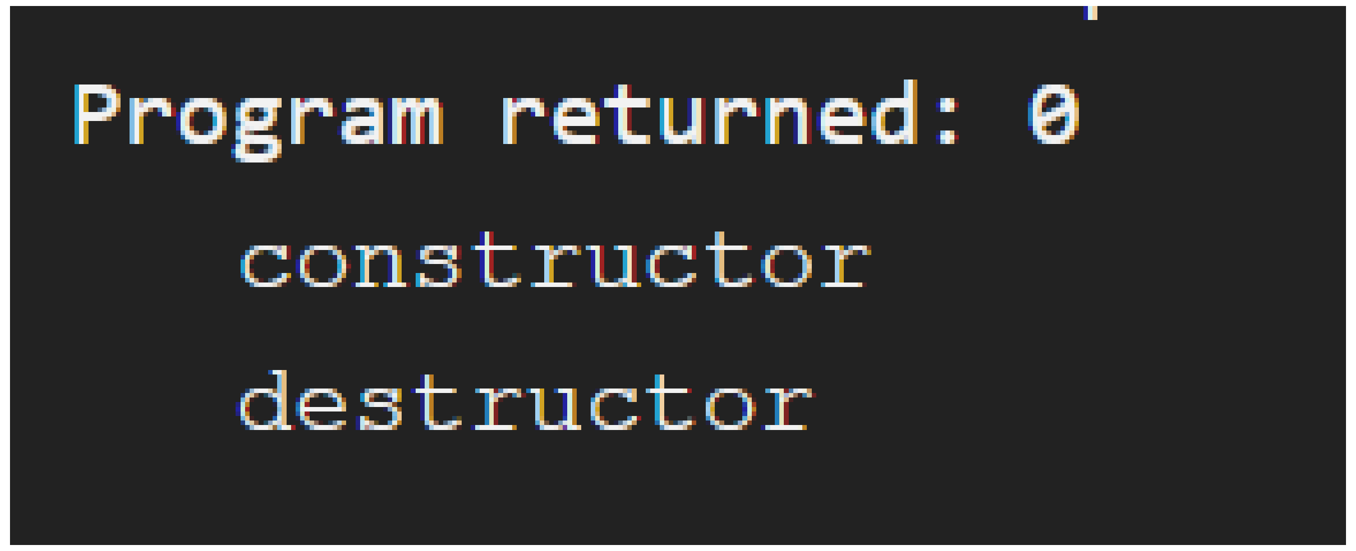
```
class UnspeakableLambda {
 public:
  UnspeakableLambda(int aValue) : value(aValue){};


  bool operator()(int other) const { return value < other; }
 private:
  const int value;
}

UnspeakableLambda l(value);
l(4);
```

Lambdas are classes... Classes define constructors and destructors

```
class UnspeakableLambda {
 public:
  UnspeakableLambda(int aValue) : value(aValue){};
  ~UnspeakableLambda() = default;

  bool operator()(int other) const { return value < other; }
 private:
  const int value;
}

UnspeakableLambda l(value);
l(4);
```

RAII with lambdas

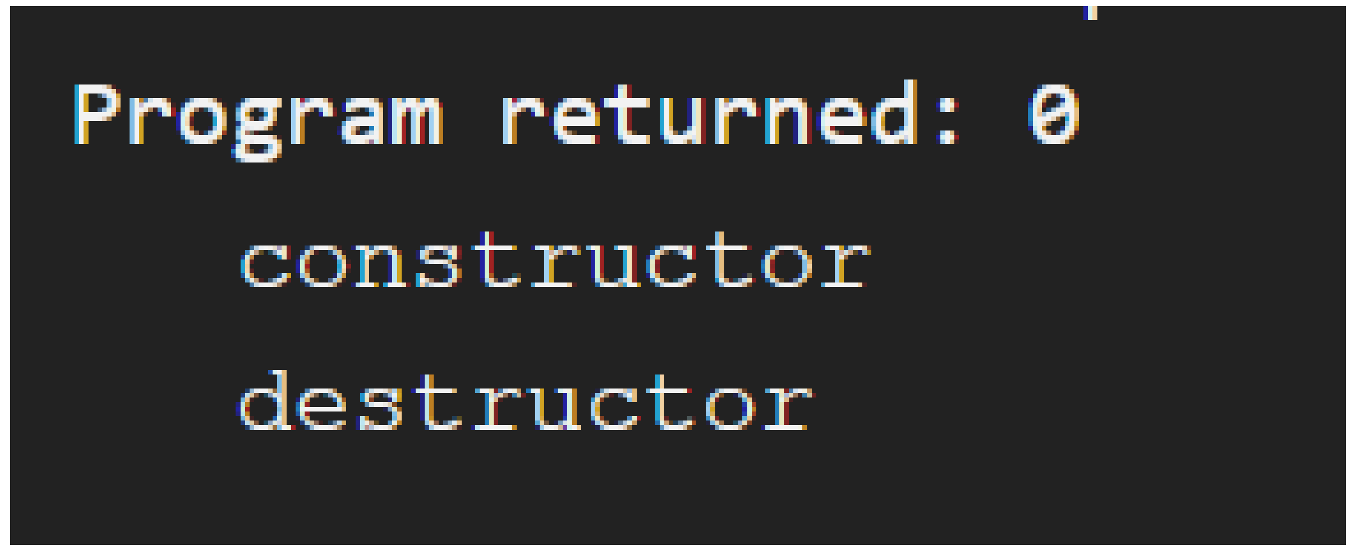
```
auto object = []() {
 struct S {
  S(){ puts("constructor"); }
  ~S() { puts("destructor"); }
 };
 return S{};
};

int main() {
 auto obj = object();
}
```



```
Program returned: 0
   constructor
   destructor
```

RAII with lambdas

```
int main() {
 auto obj = []() {
  struct S {
   S(){ puts("constructor"); }
   ~S() { puts("destructor"); }
  };
  return S{};
 }();
}
```



```
Program returned: 0
    constructor
    destructor
```

RAII with lambdas

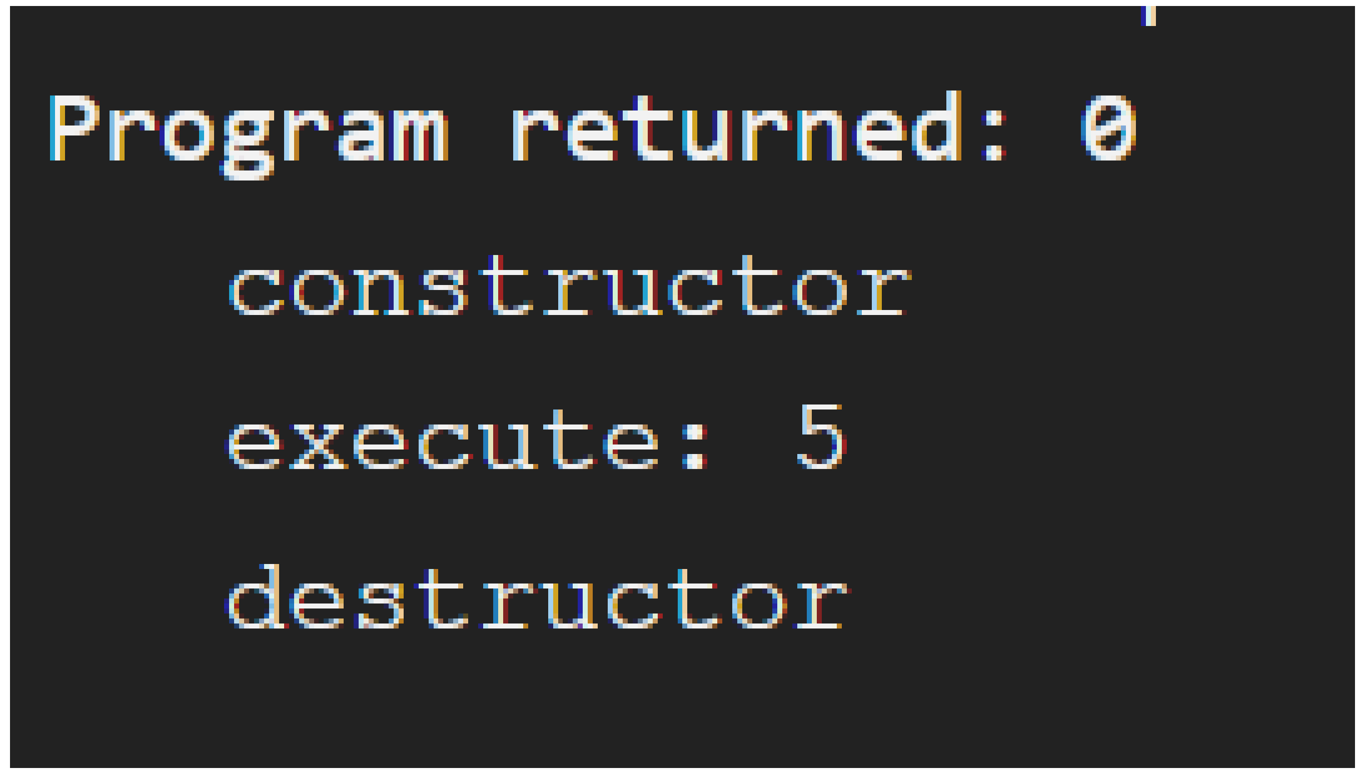
```
auto RAII =[ obj = []() {
     struct S {
      S(){ puts("constructor"); }
      ~S() { puts("destructor"); }
     };
     return S{};
    }()]
```

RAII with lambdas

```
auto RAII =[ obj = []() {
     struct S {
      S(){ puts("constructor"); }
      ~S() { puts("destructor"); }
     };
     return S{};
    }()] (int value){
 printf("execute: %d\n", value);
};
```

RAII with lambdas

```
auto RAII =[ obj = []() {
     struct S {
      S(){ puts("constructor"); }
      ~S() { puts("destructor"); }
     };
     return S{};
    }()] (int value){
 printf("execute: %d\n", value);
};

int main() {
 RAII(5);
}
```



```
Program returned: 0
   constructor
   execute: 5
   destructor
```

Creating an object wrapper using a lambda

```
auto Oven =[](auto handle) {
             return [h = handle](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
                    };
           };
```

Creating an object wrapper using a lambda

```
auto Oven =[](auto handle) {
             return [h = handle, obj = [handle](){
              struct S {
               OVEN_HANDLE h;
               S(int h_) : h(h_) {}
               ~S() { oven_turn_off(h); oven_release(h); }
              };
              return S(handle);
           }()](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
              };
           };
```

Creating an object wrapper using a lambda

```
auto Oven =[](auto handle) {
             return [h = handle, obj = [handle](){
              struct S {
               OVEN_HANDLE h;
               S(int h_) : h(h_) {}
               ~S() { oven_turn_off(h); oven_release(h); }
              };
              return S(handle);
           }()](auto func, auto&&... args) mutable {
                     return func(h, std::forward<decltype(args)>(args)...);
              };
           };
```

```
typedef int OVEN_HANDLE
typedef int OVEN_ERR

OVEN_HANDLE oven_reserve_next_available();
void oven_release(OVEN_HANDLE handle);

OVEN_ERR oven_turn_on(OVEN_HANDLE handle);
OVEN_ERR oven_turn_off(OVEN_HANDLE handle);
OVEN_ERR oven_set_temperature(OVEN_HANDLE handle, int temperature);
OVEN_ERR oven_get_temperature(OVEN_HANDLE handle, int* temperature);

OVEN_ERR oven_set_time(OVEN_HANDLE handle, int num_minutes);
OVEN_ERR oven_get_remaining_time(OVEN_HANDLE handle, int* minutes_remaining);
```

```
int main() {
    auto oven1 = Oven(oven_reserve_next_available());
    oven1(oven_turn_on);
    oven1(oven_set_temperature, 375);

    int temperature = 0;
    if(oven1(oven_get_temperature, &temperature) != OVEN_OK) {
     return -1;
    }
    while(temperature < 375) {
      sleep(60000);
      if(oven1(oven_get_temperature, &temperature) != OVEN_OK) {
        return -1;
      }
    }

    if(oven1(oven_set_time, 60) != OVEN_OK) {
     return -1;
    }

    int minutes_left = 0;
    if(oven1(oven_get_remaining_time, &minutes_left) != OVEN_OK) {
      return -1;
    }

    ....
}
```

Part 3:

Composable Functions

Composable Functions

- **Filter** - take a list of items of one and eliminate items to create a list of the same number or fewer items of the same type.
  - `std::copy_if`
- **Map** - take a list of items of one type and create a list of the same size with all items converted to a new type.
  - `std::transform`
- **Reduce** - take a list of items and create a single value
  - `std::accumulate`
  - `std::reduce` (parallel code)

Ingredient Prep

Barb's Bakery

Barb's banana bread is one of their most popular products, so they are investing in an automated banana preparation system. The goal is to tie it into their inventory system so it can automatically check all the bananas to see which ones are the perfect ripeness for the bread. It will then send those bananas to the Banana Processor 5000 for peeling and mashing.

Composable Functions: Mashing bananas

Current Inventory

filter(IsBanana)

All Bananas

filter(IsRipe)

Ripe Bananas

map(Peel)

Peeled Bananas

reduce(Mash)

Mashed Bananas

```
Ingredient MakeBananaMash() {
    auto currentInventory = GetCurrentInventory();

    std::vector<InventoryItem> allBananas;
    copy_if(begin(currentInventory), end(currentInventory), back_inserter(allBananas),
            IsBanana);

    std::vector<InventoryItem> ripeBananas;
    copy_if(begin(allBananas), end(allBananas), back_inserter(ripeBananas),
            IsRipe);

    std::vector<PeeledBanana> peeledBananas;
    transform(begin(ripeBananas), end(ripeBananas),
              back_inserter(peeledBananas), Peel);

    Ingredient mashedBananas;
    mashedBananas =
        std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                        [](auto ingredient, auto& banana) {
                            ingredient.add(Mash(banana));
                            return ingredient;
                        });
  return mashedBananas;
}
```

Composable Functions: Filter

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

Composable Functions: Filter

```
Ingredient MakeBananaMash() {
 auto currentInventory = GetCurrentInventory();

 std::vector<InventoryItem> allBananas;
 copy_if(begin(currentInventory), end(currentInventory),
            back_inserter(allBananas), IsBanana);

 std::vector<InventoryItem> ripeBananas;
 copy_if(begin(allBananas), end(allBananas), back_inserter(ripeBananas),
            IsRipe);

 std::vector<PeeledBanana> peeledBananas;
 transform(begin(ripeBananas), end(ripeBananas), back_inserter(peeledBananas), Peel);

 Ingredient mashedBananas;
 mashedBananas = std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                  [](auto ingredient, auto& banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
                  });

 return mashedBananas;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

Composable Functions: Filter

```
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 std::vector<PeeledBanana> peeledBananas;
 transform(begin(ripeBananas), end(ripeBananas), back_inserter(peeledBananas), Peel);

 Ingredient mashedBananas;
 mashedBananas = std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                  [](auto ingredient, auto& banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
                  });

 return mashedBananas;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

Composable Functions: Filter, Map

```
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 std::vector<PeeledBanana> peeledBananas;
 transform(begin(ripeBananas), end(ripeBananas), back_inserter(peeledBananas), Peel);

 Ingredient mashedBananas;
 mashedBananas = std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                  [](auto ingredient, auto& banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
                  });

 return mashedBananas;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

```
template <typename T1, typename R1>
using MappingFunction = R1 (*)(const T1&);

template <typename T1, typename R1>
std::vector<R1> Map(const std::vector<T1> sourceList, MappingFunction<T1, R1> mappingFunc) {
 std::vector<R1> mappedList;
 std::transform(begin(sourceList), end(sourceList), std::back_inserter(mappedList),mappingFunc);
 return mappedList;
}
```

Composable Functions: Filter, Map

```cpp
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 const auto peeledBananas = Map(ripeBananas, Peel);

 Ingredient mashedBananas;
 mashedBananas = std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                 [](auto ingredient, auto& banana) {
                  ingredient.add(Mash(banana));
                  return ingredient;
                 });

 return mashedBananas;
}
```

```cpp
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

```cpp
template <typename T1, typename R1>
using MappingFunction = R1 (*)(const T1&);

template <typename T1, typename R1>
std::vector<R1> Map(const std::vector<T1> sourceList, MappingFunction<T1, R1> mappingFunc) {
 std::vector<R1> mappedList;
 std::transform(begin(sourceList), end(sourceList), std::back_inserter(mappedList),mappingFunc);
 return mappedList;
}
```

Composable Functions: Filter, Map, Reduce

```
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 const auto peeledBananas = Map(ripeBananas, Peel);

 Ingredient mashedBananas;
 mashedBananas = std::accumulate(begin(peeledBananas), end(peeledBananas), mashedBananas,
                  [](auto ingredient, auto& banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
                  });

 return mashedBananas;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

```
template <typename T1, typename R1>
using MappingFunction = R1 (*)(const T1&);

template <typename T1, typename R1>
std::vector<R1> Map(const std::vector<T1> sourceList, MappingFunction<T1, R1> mappingFunc) {
 std::vector<R1> mappedList;
 std::transform(begin(sourceList), end(sourceList), std::back_inserter(mappedList),mappingFunc);
 return mappedList;
}
```

```
template<typename SourceType, typename OutType>
using ReductionFunction = OutType (*)(OutType, const SourceType);

template<typename SourceType, typename OutType>
OutType Reduce(const std::vector<SourceType> sourceList,
               ReductionFunction<SourceType,OutType> reductionFunc) {

 OutType reducedValue;
 reducedValue = std::accumulate(begin(sourceList), end(sourceList), reducedValue, reductionFunc)
 return reducedValue;
}
```

Composable Functions: Filter, Map, Reduce

```
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 const auto peeledBananas = Map(ripeBananas, Peel);

 const auto mashedBananas =
             Reduce<PeeledBanana, Ingredient>(peeledBananas,
             [](auto ingredient, const auto banana) {
                  ingredient.add(Mash(banana));
                  return ingredient;
             });

 return mashedBananas;
}
```

```
template <typename F>
using FilterFunction = bool(*)(const F&);

template<typename T>
std::vector<T> Filter(const std::vector<T> sourceList,FilterFunction<T> filter) {
 std::vector<T> filteredList;
 std::copy_if(begin(sourceList), end(sourceList), std::back_inserter(filteredList),filter);
 return filteredList;
}
```

```
template <typename T1, typename R1>
using MappingFunction = R1 (*)(const T1&);

template <typename T1, typename R1>
std::vector<R1> Map(const std::vector<T1> sourceList, MappingFunction<T1, R1> mappingFunc) {
 std::vector<R1> mappedList;
 std::transform(begin(sourceList), end(sourceList), std::back_inserter(mappedList),mappingFunc);
 return mappedList;
}
```

```
template<typename SourceType, typename OutType>
using ReductionFunction = OutType (*)(OutType, const SourceType);

template<typename SourceType, typename OutType>
OutType Reduce(const std::vector<SourceType> sourceList,
               ReductionFunction<SourceType,OutType> reductionFunc) {

 OutType reducedValue;
 reducedValue = std::accumulate(begin(sourceList), end(sourceList), reducedValue, reductionFunc)
 return reducedValue;
}
```

Composable Functions: Eliminate named temporaries

```
Ingredient MakeBananaMash() {
 const auto currentInventory = GetCurrentInventory();

 const auto allBananas= Filter(currentInventory, IsBanana);

 const auto ripeBananas = Filter(allBananas, IsRipe);

 const auto peeledBananas = Map(ripeBananas, Peel);

 const auto mashedBananas = Reduce<PeeledBanana, Ingredient>(peeledBananas,
             [](auto ingredient, const auto banana) {
                  ingredient.add(Mash(banana));
                  return ingredient;
             });

 return mashedBananas;
}
```

```
Ingredient MakeBananaMash() {

 return Reduce<PeeledBanana, Ingredient>(
              Map(
                  Filter(
                         Filter( GetCurrentInventory(), IsBanana),
                         IsRipe),
                  Peel),
              [](auto ingredient, const auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
              });
}
```

Composable Functions: Convert to use `std::ranges`

```
Ingredient MakeBananaMash() {

 return Reduce<PeeledBanana, Ingredient>(
              Map(
                  Filter(
                         Filter( GetCurrentInventory(), IsBanana),
                         IsRipe),
                  Peel),
              [](auto ingredient, const auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
              });
}
```

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

Part 4:

Lazy Evaluation

Lazy Evaluation

Delay actions or calculations until they are needed

```
class Data {
 public:
  DataType GetType();
  char* GetBuffer() { return mBuffer; }
 private:
  std::array<char, 100'000'000> mBuffer;
};




int main() {
 auto maybeNeededData = GetData();

 if(SomeCondition(maybeNeededData.GetType())) {
  return;
 }

 UseData(maybeNeededData.GetBuffer());
}
```

Lazy Evaluation: Allocation

Delay actions or calculations until they are needed

```
class Data {
 public:
  DataType GetType();
  char* GetBuffer() { return mBuffer; }
 private:
  std::array<char, 100'000'000> mBuffer;
};







int main() {
 auto maybeNeededData = GetData();

 if(SomeCondition(maybeNeededData.GetType())) {
  return;
 }

 UseData(maybeNeededData.GetBuffer());
}
```

```
class Data {
 public:
  DataType GetType();
  char* GetBuffer() {
   std::call_once(mBuferFlag, [this](){
    mBuffer.resize(100'000'000);
    FillBuffer(mBuffer);
   });
   return mBuffer;
  }
 private:
  std::vector<char> mBuffer;
  std::once_flag mBufferFlag;
};

int main() {
 auto maybeNeededData = GetData();

 if(SomeCondition(maybeNeededData.GetType())) {
  return;
 }

 UseData(maybeNeededData.GetBuffer());
}
```

Lazy Evaluation: Fetch data lazily

```
std::vector<City>
Map::GetCitiesInBoundary(LatLonBounds boundary) {
  // search through all cities in the map,
  // extracting ones within the specified
  // boundary and saving them in a vector
  return citiesInBounds;
}
```

```
void PrintCities(int maxCount) {
   Map m;
   const auto cities =
    m.GetCitiesInBoundary(GetDisplayBounds());
   const auto numToPrint = std::min(cities.size(),
                           maxCount);
   for(int i = 0; i < numToPrint) {
     std::cout << cities.at(i);
   }
}
```

```
void Map::GetCitiesInBoundary(LatLonBounds boundary,
 std::function<bool(const City&)> callback {

  // find the first city within the bounds
  nextCity = firstCity;
  while(callback( nextCity) && HasMoreCities()) {
   nextCity =  // find the next city within bounds
  }
}
```

```
void PrintCities(int maxCount) {
  Map m;
  m.GetCitiesInBoundary(GetDisplayBounds(),
   [numLeft = maxCount](const City& city) mutable {
    std::cout << city;
    return --numLeft > 0;
   });
}
```

Lazy Evaluation: Efficient Iteration

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

Lazy Evaluation: Efficient Iteration

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

`GetCurrentInventory()`

`itemIterator`

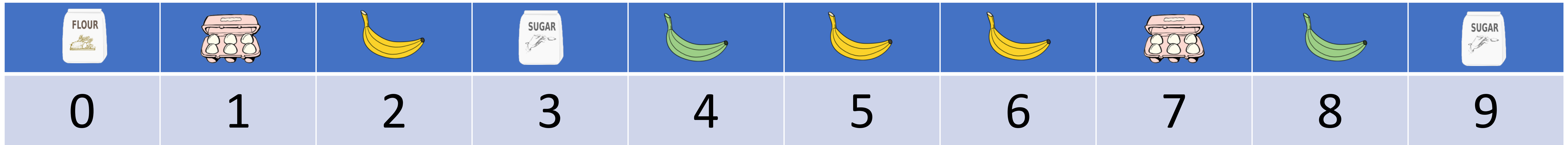
Lazy Evaluation: Efficient Iteration

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

filter(IsBanana)

isBanana(itemIterator.begin())

GetCurrentInventory()



0 1 2 3 4 5 6 7 8 9

itemIterator

begin()

end()

Lazy Evaluation: Efficient Iteration

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

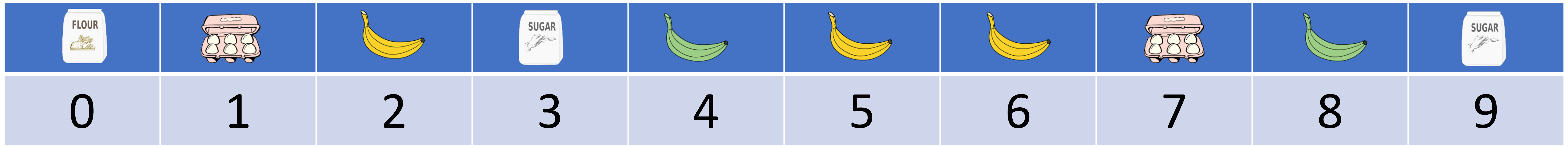
filter(IsRipe)

IsRipe(isBananaIterator.begin())

filter(IsBanana)

isBanana(itemIterator.begin())

GetCurrentInventory()



itemIterator

begin()

end()

Lazy Evaluation: Efficient Iteration

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

transform(Peel)

Peel(isRipeIterator.begin())

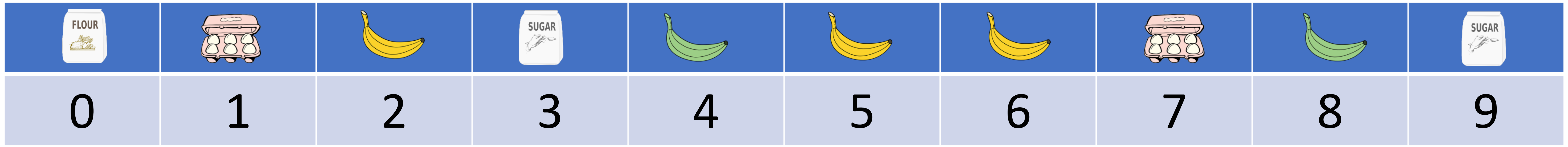
filter(IsRipe)

IsRipe(isBananaIterator.begin())

filter(IsBanana)

isBanana(itemIterator.begin())

GetCurrentInventory()



itemIterator

begin()

end()

Lazy Evaluation: Efficient Iteration

fold_left()

peeledBananas.begin()

transform(Peel)

Peel(isRipeIterator.begin())

filter(IsRipe)

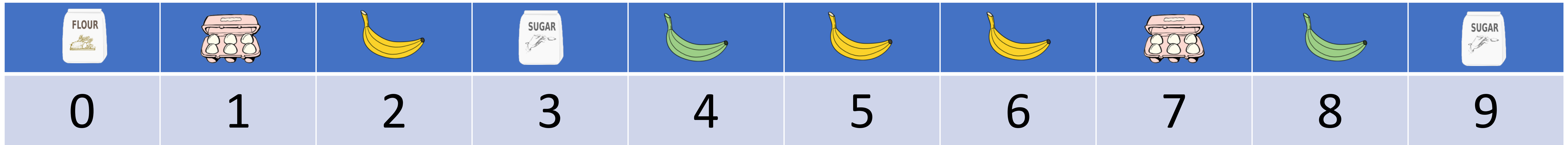
IsRipe(isBananaIterator.begin())

filter(IsBanana)

isBanana(itemIterator.begin())

```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                   ingredient.add(Mash(banana));
                   return ingredient;
          });
}
```

GetCurrentInventory()



itemIterator

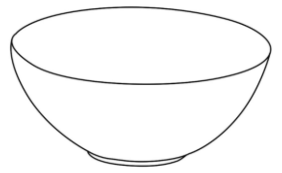
begin()

end()

Lazy Evaluation: Efficient Iteration

fold_left()

peeledBananas.begin()



```
Ingredient MakeBananaMash() {
   auto peeledBananas = GetCurrentInventory()
                        | ranges::views::filter(IsBanana)
                        | ranges::views::filter(IsRipe)
                        | ranges::views::transform(Peel);

   return ranges::fold_left(peeledBananas,Ingredient{},
          [](auto ingredient, auto banana) {
                  ingredient.add(Mash(banana));
                  return ingredient;
          });
}
```

transform(Peel)

Peel(isRipeIterator.begin())

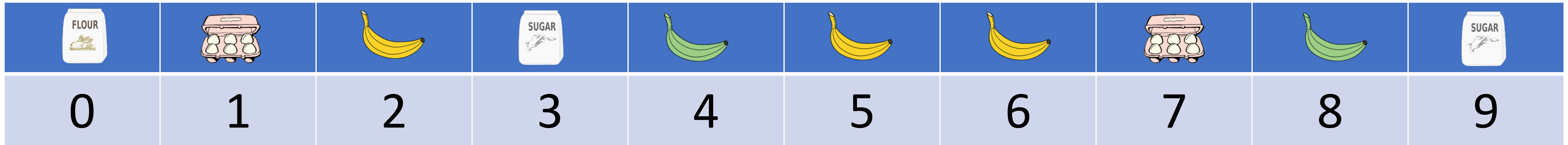
filter(IsRipe)

IsRipe(isBananaIterator.begin())

filter(IsBanana)

isBanana(itemIterator.begin())

GetCurrentInventory()

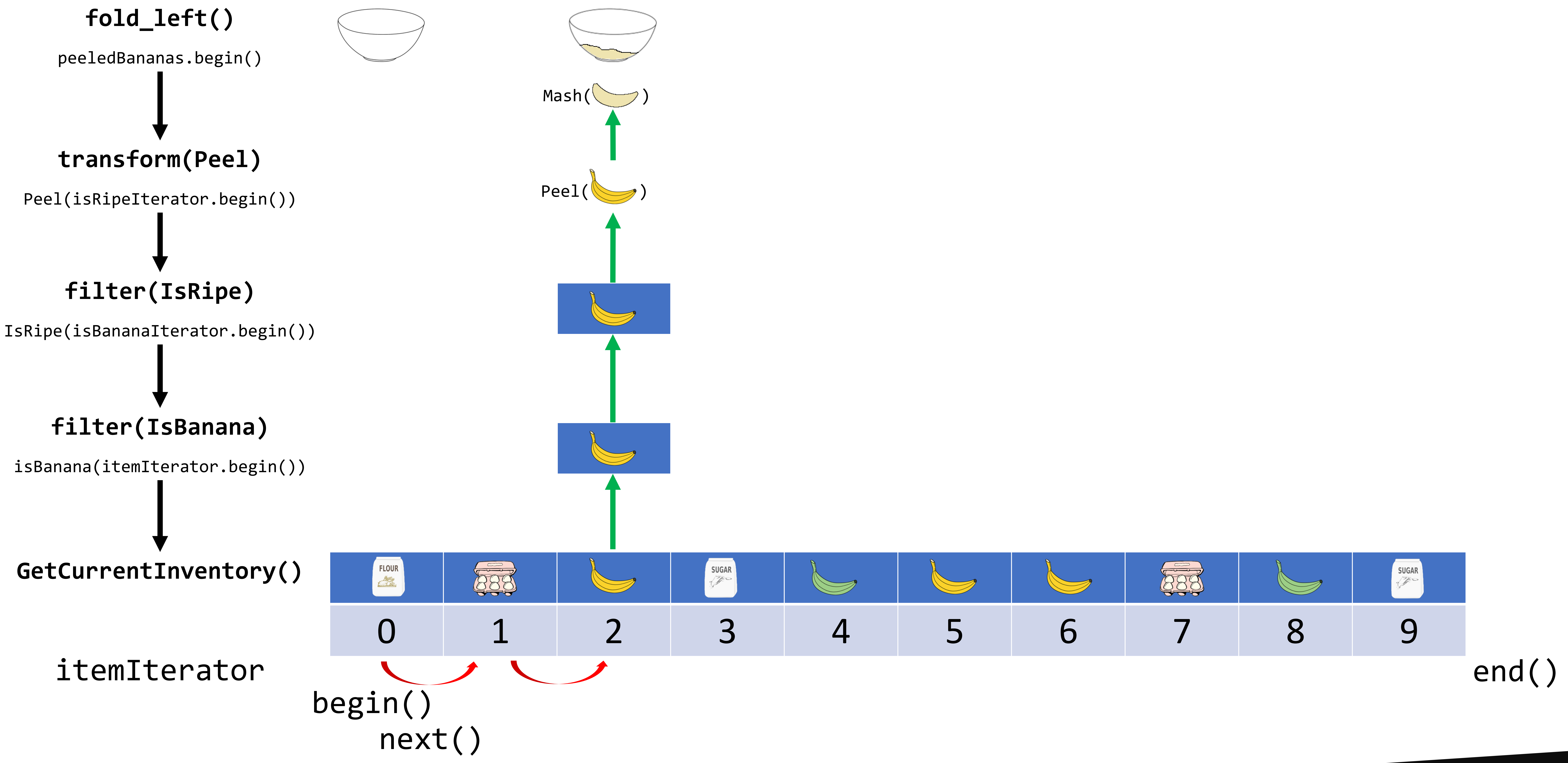


itemIterator

begin()

end()

Lazy Evaluation: Efficient Iteration



fold_left()

peeledBananas.begin()

transform(Peel)

Peel(isRipeIterator.begin())

filter(IsRipe)

IsRipe(isBananaIterator.begin())

filter(IsBanana)

isBanana(itemIterator.begin())

GetCurrentInventory()

itemIterator

Mash( )

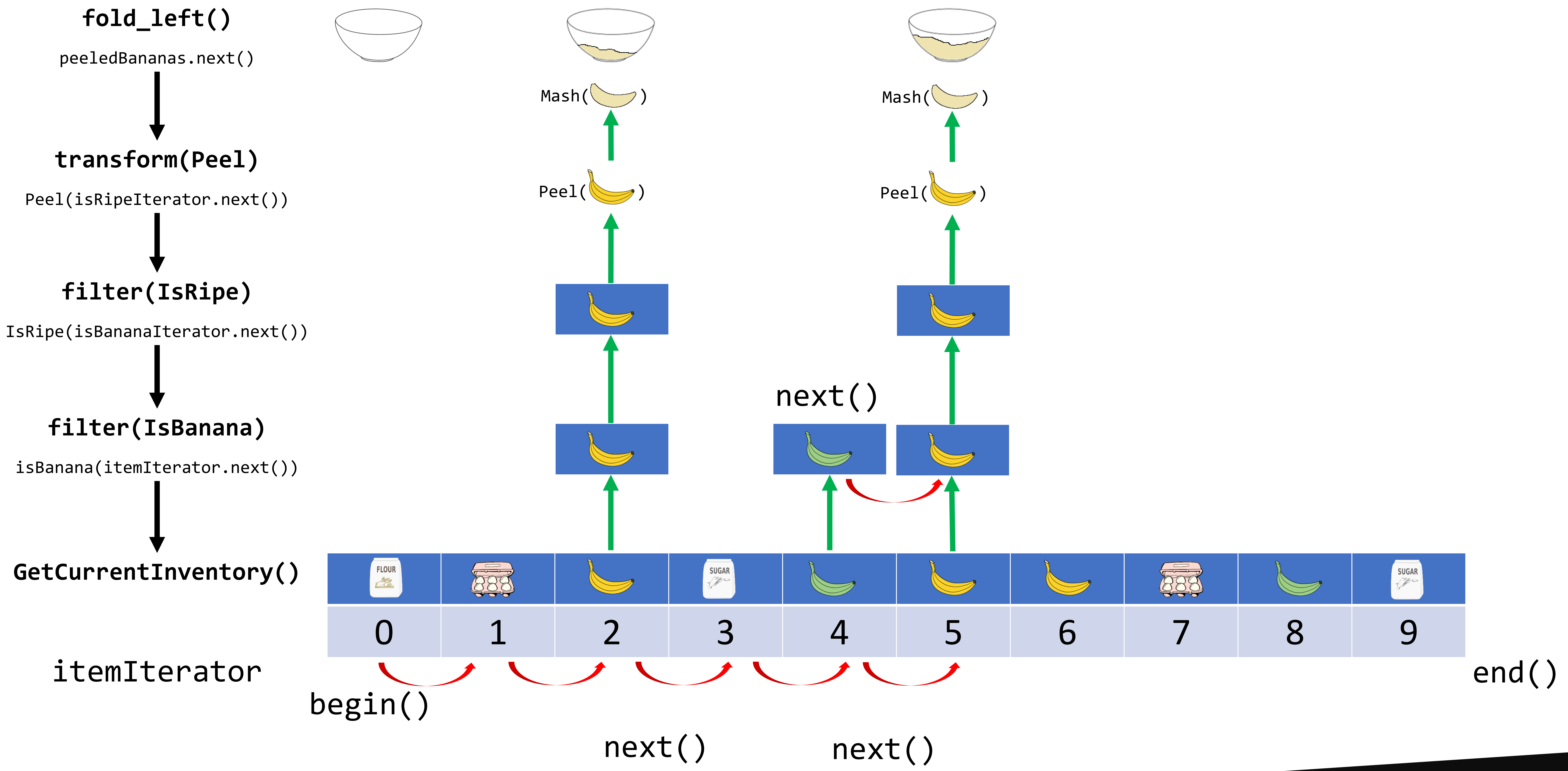
Peel( )

begin()

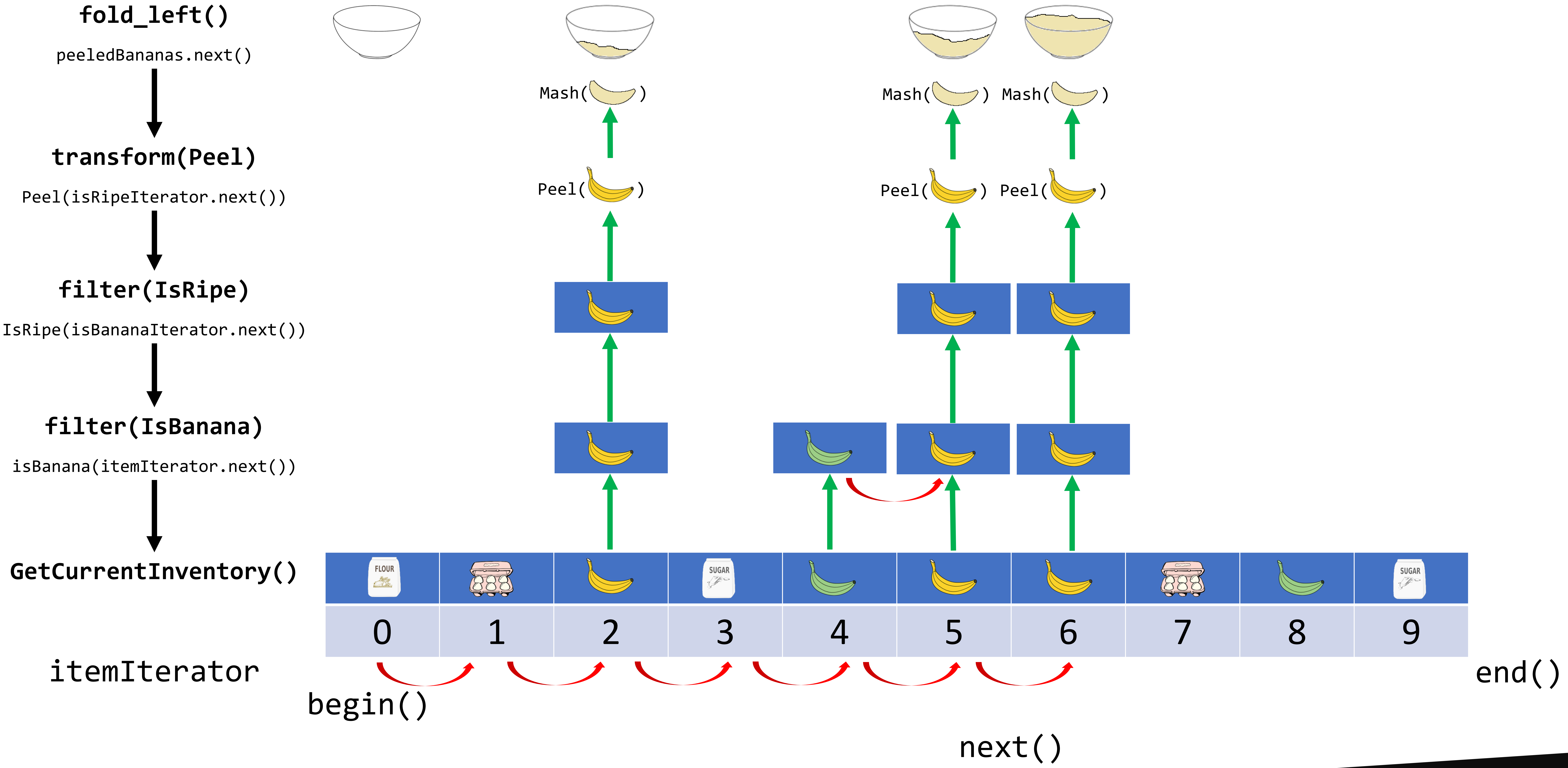
next()

end()

Lazy Evaluation: Efficient Iteration



Lazy Evaluation: Efficient Iteration



fold_left()

peeledBananas.next()

transform(Peel)

Peel(isRipeIterator.next())

filter(IsRipe)

IsRipe(isBananaIterator.next())

filter(IsBanana)

isBanana(itemIterator.next())

GetCurrentInventory()

itemIterator

begin()

end()

next()

Lazy Evaluation: Efficient Iteration

fold_left()
peeledBananas.next()
transform(Peel)
Peel(isRipeIterator.next())
filter(IsRipe)
IsRipe(isBananaIterator.next())
filter(IsBanana)
isBanana(itemIterator.next())
GetCurrentInventory()
itemIterator
Mash( )
Mash( )
Mash( )
Peel( )
Peel( )
Peel( )
FLOUR
SUGAR
SUGAR
0
1
2
3
4
5
6
7
8
9
begin()
next()
next()
next()
end()
end()
end()
end()
end()

Part 5:

Your mileage may vary

Calculations need unchanging data

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee>& employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee>& employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
auto employees = GetCurrentEmployees();

auto background =
  RunOnBackgroundThread(FilterEmployees,
   employees, isBirthday);

employees.clear();
background.join();
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
auto employees = GetCurrentEmployees();

auto background =
  RunOnBackgroundThread(FilterEmployees,
   employees, isBirthday);

employees.clear();
background.join();
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
auto employees = GetCurrentEmployees();

auto background =
  RunOnBackgroundThread(FilterEmployees,
   employees, isBirthday);

employees.clear();
background.join();
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(const std::vector<Employee> employees,
  FilterFunction filter) {
 std::vector<Employee> filteredEmployees;
 filteredEmployees.reserve(employees.size());
 std::copy_if(begin(employees), end(employees), std::back_inserter(filteredEmployees),filter);
 return filteredEmployees;
}
```

```
auto employees = GetCurrentEmployees();

auto background =
  RunOnBackgroundThread(FilterEmployees,
   employees, isBirthday);

employees.clear();
background.join();
```

Calculations need unchanging data

```
std::vector<Employee> FilterEmployees(std::vector<Employee> employees,
  FilterFunction filter) {
 const auto lastValid = std::remove_if(begin(employees), end(employees), [](const auto& item) {
   return !filter(item);});
 employees.erase(lastValid, end(employees);
 return employees;
}
```

```
auto employees = GetCurrentEmployees();

auto background =
  RunOnBackgroundThread(FilterEmployees,
   employees, isBirthday);

employees.clear();
background.join();
```

Calculations need unchanging data

- If the contained type is cheap to copy, copy it
- Guarantee that the data won't change, which requires some type of synchronization
- Create a `vector` like data structure that can create copies lazily or log changes
  - Bitmapped vector trie

Epilogue:

Thinking Functionally

Thinking Functionally

The goal of this talk was to show you different ways of thinking about a problem

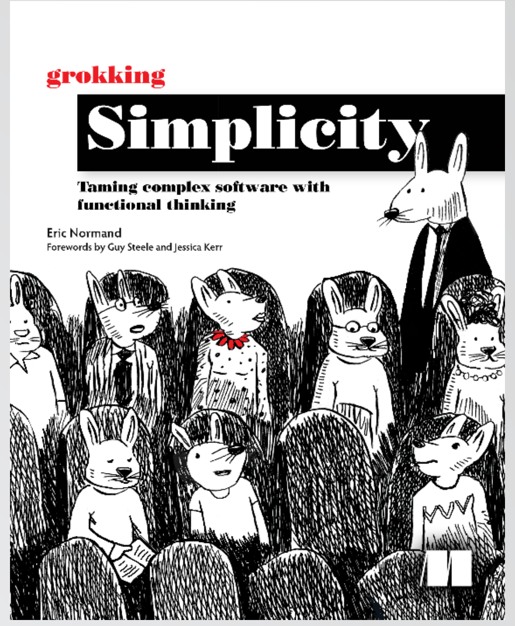
- C++ is a multiparadigm language, leverage it!
- Separate code into **Actions**, **Calculations** and **Data**.
- Isolate **Actions**
- Reuse **Calculations**
- Treat functions as **Data**
- Functions can work together
- Be lazy
- Don't be too smart

"Debugging is twice as hard as writing the code in the first place. Therefore if you write code as cleverly as possible, you are, by definition, not smart enough to debug it."

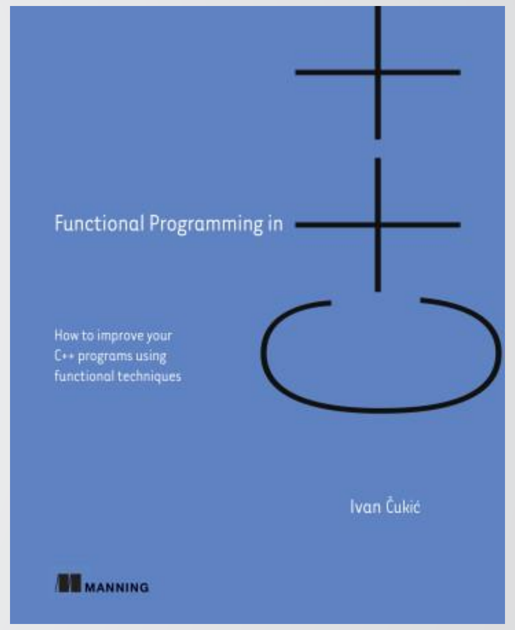
-- Brian Kernighan

Resources

Grokking Simplicity
Eric Normand



Functional Programming in C++
Ivan Čukić


Functional Programming in
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING

Ranges for the Standard Library
Eric Niebler
https://www.youtube.com/watch?v=mFUXNMfaciE

C++ Weekly - Ep 126 - Lambdas With Destructors
Jason Turner
https://www.youtube.com/watch?v=9L9uSHrJA08

THANKS