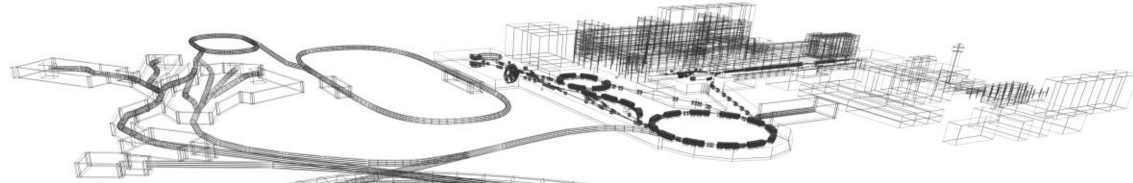


What is a [](){} ?

Matteo Alfonsi (GSI – System Planning SFRS)

What is a []{}() ?

```
void MakeItTwice(double& x)
{ x *= 2; }


for (auto& v : myvec)
{ MakeItTwice(v); }
```

- Normally a function is defined somewhere …
- … and called once or more anywhere else

What is a [](){} ?

```
void MakeItTwice(double& x)
{ x *= 2; }


for (auto& v : myvec)
{ MakeItTwice(v); }


//make use of wonderful <algorithm>
std::for_each(myvec.begin(), myvec.end(),
    MakeItTwice);
```

- Normally a function is defined somewhere …
- … and called once or more anywhere else
- First-class functions in C++, can be passed as argument to other functions, either as template parameters…

What is a []()\{\} ?

FAIR GSI

```
void MakeItTwice(double& x)
{ x *= 2; }


for (auto& v : myvec)
{ MakeItTwice(v); }


//make use of wonderful <algorithm>
std::for_each(myvec.begin(), myvec.end(),
    MakeItTwice);


//Register callback with function address
ActionAtMouseClick( & MakeItTwice );
```

- Normally a function is defined somewhere …
- … and called once or more anywhere else
- First-class functions in C++, can be passed as argument to other functions, either as template parameters…
- … or callbacks via pointers

What is a []( ){} ?

```
void MakeItTwice(double& x)
{ x *= 2; }
```

```
for (auto& v : myvec)
{ MakeItTwice(v); }
```

- As of C++11, lambda expressions can be thought as anonymous functions that can be directly defined in the spot where they are used.

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(),  myvec.end(),
    MakeItTwice);
```

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(),  myvec.end(),
    [](double& x){ x *= 2; } );
```

```
//Register callback with function address
ActionAtMouseClick( & MakeItTwice );
```

```
//Register callback with function address
ActionAtMouseClick([](double& x){ x*=2; } );
```

What is a [](){} ?

- As of C++11, lambda expressions can be thought as anonymous functions that can be directly defined in the spot where they are used.

- The complete purpose of a piece of code all in one spot
- Code of short functions often more explicative than function name

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(),  myvec.end(),
    MakeItTwice);
```

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(),  myvec.end(),
    [](double& x){ x *= 2; } );
```

```
//Register callback with function address
ActionAtMouseClick( & MakeItTwice );
```

```
//Register callback with function address
ActionAtMouseClick([](double& x){ x*=2; } );
```

What is a [](){} ()?

- Funny enough, you can also immediately call it!

```
for (auto& v : myvec)
{ MakeItTwice(v); }
```

```
//Define and call immediately
for (auto& v : myvec)
{ [](double& x){ x *= 2; } ( v ); }
```

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(), myvec.end(),
    MakeItTwice);
```

```
//make use of wonderful <algorithm>
std::for_each(myvec.begin(), myvec.end(),
    [](double& x){ x *= 2; } );
```

```
//Register callback with function address
ActionAtMouseClick( &MakeItTwice );
```

```
//Register callback with function address
ActionAtMouseClick([](double& x){ x*=2; } );
```

Can a [](){} return something?

- Funny enough, but it finds a (debated?) use for complex initialization of consts [1]
  … and I find use of it in this slide to explain that returned type is deduced [2]!

[1] *source: https://www.cppstories.com/2016/11/iife-for-complex-initialization/*
[2] *introduced in C++11 with limitations, fully-featured starting from C++14*

Can a [](){} return something?

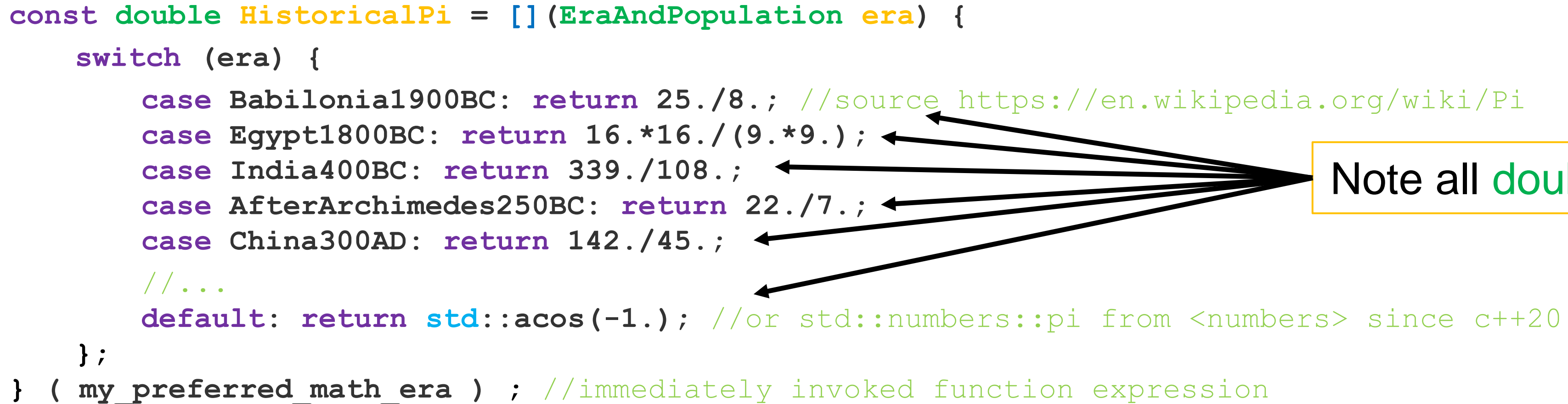
- Funny enough, but it finds a (debated?) use for complex initialization of consts [1]
  … *and I find use of it in this slide to explain that returned type is deduced [2]!*

```
const double HistoricalPi = [](EraAndPopulation era) {
    switch (era) {
        case Babilonia1900BC: return 25./8.; //source https://en.wikipedia.org/wiki/Pi
        case Egypt1800BC: return 16.*16./(9.*9.);
        case India400BC: return 339./108.;
        case AfterArchimedes250BC: return 22./7.;
        case China300AD: return 142./45.;
        //...
        default: return std::acos(-1.); //or std::numbers::pi from <numbers> since c++20
    };
} ( my_preferred_math_era ) ; //immediately invoked function expression
```

[1] *source: https://www.cppstories.com/2016/11/iife-for-complex-initialization/*
[2] *introduced in C++11 with limitations, fully-featured starting from C++14*

Can a [](){} return something?

- Funny enough, but it finds a (debated?) use for complex initialization of consts [1]
  … *and I find use of it in this slide to explain that returned type is deduced [2]!*



[1] *source: https://www.cppstories.com/2016/11/iife-for-complex-initialization/*
[2] *introduced in C++11 with limitations, fully-featured starting from C++14*

Can a [](){} return something?

- Funny enough, but it finds a (debated?) use for complex initialization of consts [1]
  … *and I find use of it in this slide to explain that returned type is deduced* [2]*!*

```
const double HistoricalPi = [](EraAndPopulation era) -> double {
    switch (era) {
        case FictionalStoneAgeMathKnowledge: return 3;
        case Babilonia1900BC: return 25./8.; //source https://en.wikipedia.org/wiki/Pi
        case Egypt1800BC: return 16.*16./(9.*9.);
        case India400BC: return 339./108.;
        case AfterArchimedes250BC: return 22./7.;
        case China300AD: return 142./45.;
        //...
        default: return std::acos(-1.); //or std::numbers::pi from <numbers> since c++20
    };
} ( my_preferred_math_era ) ; //immediately invoked function expression
```

Explicit trailing return type

[1] *source: https://www.cppstories.com/2016/11/iife-for-complex-initialization/*
[2] *introduced in C++11 with limitations, fully-featured starting from C++14*

Not-anonymous [](){}

- As of C++11, lambda expressions can be thought as ~~anonymous~~ functions that can be directly defined in the spot where they are used.

```
auto NameComposer =
  [](const std::string& name, int th_slot){
    return name + "_thread"
           + std::to_string(th_slot);
  };
```

Not-anonymous [](){}

- As of C++11, lambda expressions can be thought as ~~anonymous~~ functions that can be directly defined in the spot where they are used.

```
auto NameComposer =
  [](const std::string& name, int th_slot){
    return name + "_thread"
           + std::to_string(th_slot);
  };
for (int j = 0; j < n_thread; ++j) {
  histosT.emplace_back( //vector<TH1F>
    NameComposer("hTimes",j),
    "Arrival times of…", 100, 0., 1000.);
  histosQ.emplace_back(
    NameComposer("hCharges",j),
    //...
```

Not-anonymous [](){}

- As of C++11, lambda expressions can be thought as ~~anonymous~~ functions that can be directly defined in the spot where they are used.

```
void InitHistos(int n_threads) {
  auto NameComposer =
    [](const std::string& name, int th_slot){
      return name + "_thread"
             + std::to_string(th_slot);
    };
  for (int j = 0; j < n_thread; ++j) {
    histosT.emplace_back( //vector<TH1F>
      NameComposer("hTimes",j),
      "Arrival times of…", 100, 0., 1000.);
    histosQ.emplace_back(
      NameComposer("hCharges",j),
      //...
}
```

Not-anonymous []{}

- As of C++11, lambda expressions can be thought as ~~anonymous~~ functions that can be directly defined in the spot where they are used.

```
void InitHistos(int n_threads) {
  auto NameComposer =
    [](const std::string& name, int th_slot){
      return name + "_thread"
             + std::to_string(th_slot);
    };
  for (int j = 0; j < n_thread; ++j) {
    histosT.emplace_back( //vector<TH1F>
      NameComposer("hTimes",j),
      "Arrival times of…", 100, 0., 1000.);
    histosQ.emplace_back(
      NameComposer("hCharges",j),
      //...
```

```
void DetectorConstruction    //Geant4
::ConstructSupportStructure() {
  //...

  auto p_solid =
    ConstructBeam("leg_SW");
  auto p_logicalv =
    new G4LogicalVolume(p_solid,
            Steel304L, "leg_SW"+"_lv");
  auto p_physical =
    new G4PVPlacement(
      m_parts_db.Rot("leg_SW"),
      m_parts_db.Transl("leg_SW"),
      p_logicalv, "leg_SW"+"_pv",
      mp_mothervolume, false, 0);
```

Do not care about new, that's Geant4, no leaks!

Assume as member a "database" object with all the information for each component

Not-anonymous [](){}

- As of C++11, lambda expressions can be thought as ~~anonymous~~ functions that can be directly defined in the spot where they are used.

```
void InitHistos(int n_threads) {
  auto NameComposer =
    [](const std::string& name, int th_slot){
      return name + "_thread"
             + std::to_string(th_slot);
    };
  for (int j = 0; j < n_thread; ++j) {
    histosT.emplace_back( //vector<TH1F>
      NameComposer("hTimes",j),
      "Arrival times of…", 100, 0., 1000.);
    histosQ.emplace_back(
      NameComposer("hCharges",j),
      //...
```

```
void DetectorConstruction    //Geant4
::ConstructSupportStructure() {
  //...
  auto ConstructAndPlaceBeam =
    [      ](const std::string& db_id) {
      auto p_solid =
        ConstructBeam(db_id);
      auto p_logicalv =
        new G4LogicalVolume(p_solid,
                Steel304L, db_id+"_lv");
      auto p_physical =
        new G4PVPlacement(
              m_parts_db.Rot(db_id),
              m_parts_db.Transl(db_id),
              p_logicalv, db_id+"_pv",
              mp_mothervolume, false, 0);
  };
  ConstructAndPlaceBeam("leg_SW");
  ConstructAndPlaceBeam("leg_NW");
  //...
```

Generic [](){} and more…

- As of C++14, Lambda expressions can be generic (as a function template):

```
auto print = [] (auto arg) { std::cout << arg << "\n"; } ;
print(43); print("is better than 42. Remember"); print(HistoricalPi);
```

Generic [](){} and more…

- As of C++14, Lambda expressions can be generic (as a function template):

```
auto print = [] (auto arg) { std::cout << arg << "\n"; } ;
print(43); print("is better than 42. Remember"); print(HistoricalPi);
```

- Very useful with the C++17 `std::variant` and its visitor pattern

Generic [](){} and more…

- As of C++14, Lambda expressions can be generic (as a function template):

```
auto print = [] (auto arg) { std::cout << arg << "\n"; } ;
print(43); print("is better than 42. Remember"); print(HistoricalPi);
```

- Very useful with the C++17 `std::variant` and its visitor pattern

- As of C++17, Lambda expressions can profit from the fold expressions feature:

```
auto print = [] (auto... args) {
    ( (std::cout << args << "\t"), ...); std::cout << "\n";
} //from www.cppstories.com/2020/08/c-lambda-week-some-tricks.html/
print(43, "is better than 42. Remember", HistoricalPi);
```

Generic [](){} and more…

- As of C++14, Lambda expressions can be generic (as a function template):

```
auto print = [] (auto arg) { std::cout << arg << "\n"; } ;
print(43); print("is better than 42. Remember"); print(HistoricalPi);
```

- Very useful with the C++17 `std::variant` and its visitor pattern

- As of C++17, Lambda expressions can profit from the fold expressions feature:

```
auto print = [] (auto... args) {
    ( (std::cout << args << "\t"), ...); std::cout << "\n";
} //from www.cppstories.com/2020/08/c-lambda-week-some-tricks.html/
print(43, "is better than 42. Remember", HistoricalPi);
```

- As of C++20, they can be more minutely “template-ed” and use concepts (not discussed)

```
auto lambdaCpp20 = []<typename T>(std::vector<T> vec) { /*...*/ }
```

[]( ){} and function objects

```
int oof(int i) { return i+1; } //def
int d = oof(5); //call


struct Rab {
  int operator()(int i)
    { return i+1; }
} rab; //def
int d = rab(5); //call
```

[](){} and function objects

```
int oof(int i) { return i+1; } //def
int d = oof(5); //call
```

```
struct Rab {
  int operator()(int i)
    { return i+1; }
} rab; //def
int d = rab(5); //call
```

```
int oof(int i) //def
{ static int j = 1; return i + (j++); }
int d = oof(5); d = oof(5); //call
```

```
struct Rab {
  int j_;
  Rab(int j) : j_{j} {}
  int operator()(int i)
    { return i + (j_++); }
} rab(1) ; //def
int d = rab(5); d = rab(5); //call
```

[](){} and function objects

```
int oof(int i) { return i+1; } //def
int d = oof(5); //call
```

```
struct Rab {
  int operator()(int i) const
    { return i+1; }
  //plus conversion to function ptr
} rab; //def
```

compilers create
a function object

```
auto rab = [] (int i) { return i+1; };
int d = rab(5); //call
```

```
int oof(int i) //def
{ static int j = 1; return i + (j++); }
int d = oof(5); d = oof(5); //call
```

```
struct Rab {
  int j_;
  Rab(int j) : j_{j} {}
  int operator()(int i)
    { return i + (j_++); }
} rab(1) ; //def
int d = rab(5); d = rab(5); //call
```

[]{}() and function objects

```
int oof(int i) { return i+1; } //def
int d = oof(5); //call
```

```
int oof(int i) //def
{ static int j = 1; return i + (j++); }
int d = oof(5); d = oof(5); //call
```

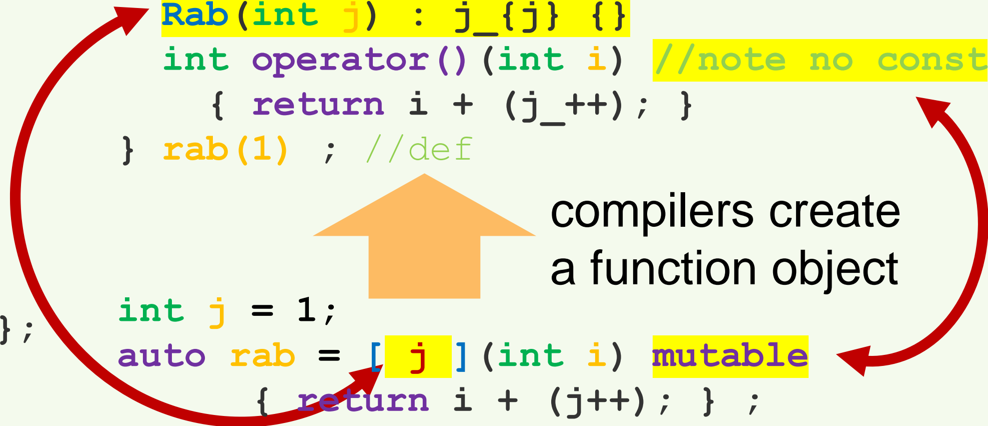
```
struct Rab {
  int operator()(int i) const
    { return i+1; }
 //plus conversion to function ptr
} rab; //def
```

compilers create
a function object

```
auto rab = [] (int i) { return i+1; };
int d = rab(5); //call
```

```
struct Rab {
   int j_;
   Rab(int j) : j_{j} {}
   int operator()(int i) //note no const
     { return i + (j_++); }
} rab(1) ; //def
```

compilers create
a function object



```
int j = 1;
auto rab = [ j ](int i) mutable
    { return i + (j++); } ;
int d = rab(5); d = rab(5); //call
```

Real case: ROOT RDataFrame

```
ROOT::RDataFrame d(nAvailableWfFiles);
auto dd = d.Define("diaS1_wf_filename",
                   [ wffilelist ](ULong64_t entry)
                        { return wffilelist[entry]; } ,
                   {"rdfentry_"} )
  .Define("diaS1_wf",
          [ & tekdec ](const std::string& wf_filename) {
              std::vector<double> res;
              tekdec.ChangeFile(wf_filename)
                 .ProvideSamples(res, 0);
              return res;  } ,
          {"diaS1_wf_filename"} )
  .Filter([](const std::vector<double>& wf)
                { return wf.size() > 0; } ,
          { "diaS1_wf" } )
  .Define("diaS1_baseline",
          [ first = config_baseline_first,
            last = config_baseline_last    ](const std::vector<double>& wf) {

                   //...etc..
```

- A graph of "callables" is created to define the "variables" of your analysis
- Heavily based on generic algorithms, which execute only when needed
- Could not be so "expressive" without lambdas

Capturing local variables

FAIR GSI

`[ n ]`	"Capture by-value", the function object has a member with same name of, type deduced from and initialized to **n**. If the lambda is not `mutable`, this member cannot be altered.
`[ n = f(3) ]`	"Init capture" allows more flexibility as of C++14. It is a "capture by value" as above.
`[ & n ]`	"Capture by-reference", the function object contains a reference member linked to the orginal **n**. Modifying this (`mutable` not required) affect the original variable.
`[ x, &y ]`	**x** is captured by-value, **y** is captured by-reference. Redundancy or ambiguity illegal.
`[ = ]`	Each mentioned variable is implicitly captured by value.
`[ & ]`	Each mentioned variable is implicitly captured by reference.
`[ =, &y ]` `[ &, y ]`	Each mentioned variable is implicitly captured by value, but **y** is captured by-reference. Vice versa. Again, redundancy or ambiguity illegal.
`[ ]`	No variable is captured from enclosing scope.
`[ x = std::move(x) ]`	C++14 "Init capture" comes in aid also for capturing types that cannot be copied but should be moved

Note: pack expansion capture intentionally omitted, too much sorry!

Capturing members

```
void DetectorConstruction    //Geant4
::ConstructSupportStructure() {
  //...
  auto ConstructAndPlaceBeam =
    [     ](const std::string& db_id) {
      auto p_solid =
        ConstructBeam(db_id);
      auto p_logicalv =
        new G4LogicalVolume(p_solid,
                Steel304L, db_id+"_lv");
      auto p_physical =
        new G4PVPlacement(
             m_parts_db.Rot(db_id),
             m_parts_db.Transl(db_id),
             p_logicalv, db_id+"_pv",
             mp_WaterLV, false, 0);
  };
  ConstructAndPlaceBeam("leg_SW");
  ConstructAndPlaceBeam("leg_NW");
  //...
```

Capturing members

- Capturing the **this** pointer gives access to the object members and methods
- Note that despite the “by-copy” look, we get access to the original object!

```
void DetectorConstruction    //Geant4
::ConstructSupportStructure() {
  //...
  auto ConstructAndPlaceBeam =
    [this](const std::string& db_id) {
      auto p_solid =
        ConstructBeam(db_id);
      auto p_logicalv =
        new G4LogicalVolume(p_solid,
                Steel304L, db_id+"_lv");
      auto p_physical =
        new G4PVPlacement(
              m_parts_db.Rot(db_id),
              m_parts_db.Transl(db_id),
              p_logicalv, db_id+"_pv",
              mp_WaterLV, false, 0);
  };
  ConstructAndPlaceBeam("leg_SW");
  ConstructAndPlaceBeam("leg_NW");
  //...
```

Capturing members

- Capturing the **this** pointer gives access to the object members and methods
- Note that despite the “by-copy” look, we get access to the original object!
- **[&]** and **[=]** implicitly captures also **this**

```
void DetectorConstruction    //Geant4
::ConstructSupportStructure() {
  //...
  auto ConstructAndPlaceBeam =
    [this](const std::string& db_id) {
      auto p_solid =
        ConstructBeam(db_id);
      auto p_logicalv =
        new G4LogicalVolume(p_solid,
                Steel304L, db_id+"_lv");
      auto p_physical =
        new G4PVPlacement(
              m_parts_db.Rot(db_id),
              m_parts_db.Transl(db_id),
              p_logicalv, db_id+"_pv",
              mp_WaterLV, false, 0);
  };
  ConstructAndPlaceBeam("leg_SW");
  ConstructAndPlaceBeam("leg_NW");
  //...
```

Capturing members

- Capturing the **this** pointer gives access to the object members and methods
- Note that despite the "by-copy" look, we get access to the original object!
- **[&]** and **[=]** implicitly captures also **this**
- Changes along the path to C++20:
  - C++17 introduces **[*this]** to actually get and work on a copy of **this**.
  - C++20 *deprecates* the **[=]** implicit capture of **this**, now you are allowed to write **[=, this]** if you really mean that.

```
void DetectorConstruction   //Geant4
::ConstructSupportStructure() {
  //...
  auto ConstructAndPlaceBeam =
    [this](const std::string& db_id) {
      auto p_solid =
        ConstructBeam(db_id);
      auto p_logicalv =
        new G4LogicalVolume(p_solid,
            Steel304L, db_id+"_lv");
      auto p_physical =
        new G4PVPlacement(
            m_parts_db.Rot(db_id),
            m_parts_db.Transl(db_id),
            p_logicalv, db_id+"_pv",
            mp_WaterLV, false, 0);
  };
  ConstructAndPlaceBeam("leg_SW");
  ConstructAndPlaceBeam("leg_NW");
  //...
```

Globals and … ROOT quirks

- No, you cannot capture globals, and it does not make sense, since globals can be use anyway anywhere!

Globals and … ROOT quirks

FAIR GSI

```
In [1]:  1 std::vector<int> vec = {1,2,3,4,5,6,7,8,9,10}

(std::vector<int> &) { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

```
In [2]:  1 int n=5;
         2 auto incrementer = [](std::vector<int>& vec) {
         3     for (auto& v : vec) v+=n;
         4 }

input_line_48:4:28: error: variable 'n' cannot be implicitly captured in a lambda with no capture-default specified
    for (auto& v : vec) v+=n;
                           ^
input_line_48:2:6: note: 'n' declared here
 int n=5;
     ^
input_line_48:3:20: note: lambda expression begins here
auto incrementer = [](std::vector<int>& vec) {
                   ^
```

```
In [3]:  1 int n=5;
         2 auto incrementer = [&n](std::vector<int>& vec) {
         3     for (auto& v : vec) v+=n;
         4 }

((lambda) &) @0x7f0930782050
```

```
In [4]:  1 auto multiplier = [&n](std::vector<int>& vec) {
         2     for (auto& v : vec) v*=n;
         3 }

input_line_50:2:22: error: 'n' cannot be captured because it does not have automatic storage duration
 auto multiplier = [&n](std::vector<int>& vec) {
                     ^
input_line_49:2:6: note: 'n' declared here
 int n=5;
     ^
```

- No, you cannot capture globals, and it does not make sense, since globals can be use anyway anywhere!
- If you use ROOT and in particular its cling interpreter, look at this quirk!

New powers, new crashes

- Captures provide new ways to refer to deleted objects!

```
auto LambdaFactory() {
  int counter = 0;
  return [ & ]() {
    std::cout << ++counter
       << " fabricated so far\n";
    /*...*/ } ;
}
//...
int main() {
  //...
  auto callable = LambdaFactory();
  callable(); //counter is deleted!
  //...
```

New powers, new crashes

- Captures provide new ways to refer to deleted objects!

```
auto LambdaFactory() {
  int counter = 0;
  return [ & ]() {
    std::cout << ++counter
       << " fabricated so far\n";
    /*...*/ } ;
}
//...
int main() {
  //...
  auto callable = LambdaFactory();
  callable(); //counter is deleted!
  //...
```

```
struct LambdaFactory {
  int counter = 0;
  std::function<void()> Create() {
    return [ = ]() {
      std::cout << ++counter
        << " fabricated so far\n";
      /*...*/ } ;
  }
};
```

New powers, new crashes

FAIR GSI

- Captures provide new ways to refer to deleted objects!

```
auto LambdaFactory() {
  int counter = 0;
  return [ & ]() {
    std::cout << ++counter
       << " fabricated so far\n";
    /*...*/ } ;
}
//...
int main() {
  //...
  auto callable = LambdaFactory();
  callable(); //counter is deleted!
  //...
```

```
struct LambdaFactory {
  int counter = 0;
  std::function<void()> Create() {
    return [ = ]() {
      std::cout << ++counter
         << " fabricated so far\n";
      /*...*/ } ;
  }
};
int main() {
  //...
  auto callables_vec = []() {
    std::vector<std::function<void()> > res;
    LambdaFactory factory;
    for (int i=0; i<5; ++i)
      res.push_back( factory.Create() );
    return res; } ();
  callables_vec[0](); //counter is deleted!
```

Performance considerations

Performance considerations

1. Information from the web agrees that, for most of the cases lambdas, will produce “in-place” code instead of function calls (“compilers have all the information to do that in the same spot”)

Performance considerations

1. Information from the web agrees that, for most of the cases lambdas, will produce “in-place” code instead of function calls (“compilers have all the information to do that in the same spot”)
2. Captures are making copies etc.. Do they consumes more stack, in case this matters? Are all of them optimized out for most cases, especially for not-mutable lambdas?
3. Other important points?

- Personally, having never been involved in time-critical or embedded applications, I value much more the time spent to think and write the code, or the time to figure out the meaning of a old piece of code when I come back to it after some time… and I found that the “all in the same spot” of the lambda expressions helps a lot!

Summary

function parameters, generic parameters (optional)

`const`, `constexpr` (C++17), `consteval` (C++20) also here! What is the meaning???

function object instance name

captures list `=`, `&`, `this`, `*this` (C++17)

C++20 concepts

```
"constness" auto lambda =
[] <> () specifiers exception attr -> ret requires
{ /*code; */ }
```

C++20 generic lambda fine-tuning

trailing return type (typically not required)

`mutable`
`constexpr` (C++17), `consteval` (C++20)
`noexcept`
standard or compiler-specific attributes

Ah! And you can also immediately invoke them with `(…)` at the end

Ah! And they can be copied or embedded in `std::function`… but no time for that!

Sources

- The series: https://www.cppstories.com/2020/08/lambda-syntax.html/#the-series
- https://en.cppreference.com/w/cpp/language/lambda
- https://www.cppstories.com/2016/11/iife-for-complex-initialization/
- https://www.cppstories.com/2020/07/lambdas5ex.html/
- https://www.nextptr.com/tutorial/ta1430524603/capture-this-in-lambda-expression-timeline-of-change
- GSI C++ User Group

Many thanks to Christian and Dennis for reviewing my slides and giving suggestions!