CppCoreGuidelines, part 3
Functions

Juanmi Huertas
R&D Software Engineer @ HP

meetup C++, Barcelona

April 5th, 2018

CppCoreGuideline index

P: Philosophy
F: Functions
I: Interfaces
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

Previously… on CppCoreGuidelines, part 1…

1) Meaningful code contains it’s own meaning.

P.1: Express ideas directly in code, P.3: Express intent.

2) Legible code is easy to read an to understand.

P.11: Encapsulate messy constructs, rather than spreading through the code.

3) Error-free code has no errors.

P.4: Ideally, a program should be statically type safe, P.5: Prefer compile-time checking to run-time checking,
P.6: What cannot be checked at compile time should becheckable at run time,
P.7: Catch run-time errors early.

4) ‘Cheap’ code perform faster.

P.8: Don't leak any resources, P.9: Don't waste time or space, P.10: Prefer immutable data to mutable data.

5) Standard code follows Software Engineer principles.

P.2: Write in ISO Standard C++, P.12: Use supporting tools as appropriate,
P.13: Use support libraries as appropriate.

Previously... on CppCoreGuidelines, part 2...

1) Use `const` for your types.

Con.1: By default, makes objects immutable.
Con.4: Use `const` define objects with values that do not change after construction.
NR.1: Don't: All declarations should be at the top of a function.
ES.21: Don't introduce a variable (or constant) before you need to use it.

2) Use `const` for your member functions.

Con.2: By default, make member functions `const`.

3) Use `const` for your parameters.

Con.3: By default, pass pointers and referneces to `const`s.
ES.50: Don't cast away `const`.

4) Use `constexpr` as much as you can.

Con.5: Use `constexpr` for values that can be computed at compile time.

CppCoreGuideline index

P: Philosophy
F: Functions
I: Interfaces
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

0) What is a function?

Functions are C++ entities that associate a sequence of statements (a *function body*) with a *name* and a list of zero or more *function parameters*.

```
bool isodd(int n)
{
  return n % 2;
}
```

... the function call expression supports pointers to functions, dereferenced pointers to member functions, lambda-expressions, and any variable of class type that overloads the function-call operator... known as FunctionObjects...

```
auto isOdd = [](int n)
{
    return n%2;
};
```

```
auto glambda = [](auto a, auto b) { return a < b; };
bool b = glambda(3, 3.14); // ok
```

0.a) What is a function? Declaring and defining

A function declaration introduces the function name and its type.
The type is determined by types of parameters, return type, `const/volatile`, ref-qualification, attributes...

noptr-declarator **(** *parameter-list* **)** *cv*(optional) *ref*(optional) *except*(optional) *attr*(optional)

noptr-declarator **(** *parameter-list* **)** *cv*(optional) *ref*(optional) *except*(optional) *attr*(optional) **->***trailing*

```
int f(char s[3]);
int f(char[]);
int f(char* s);
```

A function definition associates the function name/type with the function body.

attr(optional) *decl-specifier-seq*(optional) ***declarator*** *virt-specifier-seq*(optional) *function-body*

```
int max(int a, int b, int c)
{
  int m = (a > b) ? a : b;
  return (m > c)? m : c;
}
```

ctor-initializer(optional) *compound-statement*
function-try-block
= delete ;
= default ;

0.b) What is a function? At the beginning...

A program shall contain a global function named `main`, which is the designated start of the program.

```
int main()
{
  // body
}
```

```
int main(int argc, char* argv[])
{
  return 0;
}
```

```
int main(int ac, char** av)
{
  // body
}
```

It is called at program startup after initialization of the non-local objects with static storage duration.

```
class MyClass
{
  static int iCount;
};
```

```
int MyClass::iCount = 1;

int main()
{
  // body
}
```

F.46: `int` is the return type for `main()`

1) Encapsulate your code!

Express intent. Express ideas through code.
Short and simple ideas are easier to code, to understand, to test, to fix.

```
void read_and_print(istream& is)
{
  int x;
  if (is >> x)
    cout << "the int is " << x << '\n’;
  else
    cerr << "no int on input\n";
}
```

```
int read(istream& is)
{
  int x;
  is >> x;
  // check for errors
  return x;
}

void print(int x)
{
  cout << x << "\n";
}
```

```
void read_and_print(istream& is)
{
  auto x = read(cin);
  print(x);
}
```

Does my function fit on a screen?
That is probably a good indicator to know if you have to refactor your code.

F.1: "Package" meaningful operations as carefully named functions
F.2: A function should perform a single logical operation
F.3: Keep functions short and simple

1.a) Encapsulate your code... efficiently and expressively!

I want my function to be compile-time... help the compiler help you!

```
constexpr int min(int x, int y)
{
  if constexpr (x < y)
    return x;
  else
    return y;
}

void test(int v)
{
  int m1 = min(-1, 2); // probably compile-time evaluation
  constexpr int m2 = min(-1, 2); // compile-time evaluation
  int m3 = min(-1, v); // run-time evaluation
  constexpr int m4 = min(-1, v); // error: cannot evaluate at compile time
}
```

F.4: If a function may have to be evaluated at compile time, declare it `constexpr`

1.b) Encapsulate your code... efficiently and expressively!

Help the compiler help you!

F.6: If your function may not throw, declare it `noexcept`

Exceptions? If you don't like exceptions... don't use them. But say so.

Mark the functions that will not throw with **noexcept**.

```
void my_function (int x);
```

```
void my_exception_free_function (int x) noexcept;
```

F.5: If a function is very small and time-critical, declare it `inline`

```
// header file
#ifndef EXAMPLE_H
#define EXAMPLE_H
// function included in multiple source files must be inline
inline int sum(int a, int b) { return a + b; }
#endif
```

```
#include "example.h"
int a() { return sum(1, 2); }
```

```
#include "example.h"
int b() { return sum(3, 4); }
```

constexpr implies **inline**.

Member functions defined in-class are `inline` by default.

Not to use with functions greater than three/four lines.

2) Parameters or arguments?

A function declaration introduces the function name and its type.
The type is determined by types of parameters, return type, `const/volatile`, ref-qualification, attributes...

```
void print(const char& c);
void print(int x);
```

```
void print(const string& s);
void print(const string& s, format f);
```

```
void print(const string& s, format f = {});
```

DRY
Don't Repeat Yourself. Adding a default value you can share the implementation.

F.51: Where there is a choice, prefer default arguments over overloading

```
X* find(map<Blob>& m, const string& s, Hint);
```

Useless trivia: This was introduced in the early 1980s.

F.9: Unused parameters should be unnamed

2.a) Parameters or arguments? Pure functions.

Pure functions? Impure functions? Total functions? Partial functions?

```
bool not(bool b)
{
  return !b;
}
```

```
static int numDiv = 0;
bool div(int x, int y)
{
  numDiv++;
  return x/y;
}
```

```
static int numNot = 0;
bool count_not(bool b)
{
  numNot++;
  return !b;
}
```

`constexpr` functions are pure!
Pure functions are easy to test.
Pure functions are easy to parallelize.
Total functions are defined by ALL values.
So pure *and* total is awesome!

```
template <typename K, typename V>
V lookup(map<K,V>, K);
```

```
template <typename K, typename V>
optional<V> lookup(map<K,V>, K);
```

```
template <typename K, typename V>
V& map<K,V>::operator[](const K&);
```

```
template <typename K, typename V>
optional<reference_wrapper<V>> map<K,V>::operator[](const K&);
```

Suggested talk: *Using Types Effectively - Ben Deane - CppCon 2016* [slides] [video]

2.b) Parameters or arguments? Input parameters.

```
void in(int x);
```

```
void in(const int& x);
```

```
void in_out(int& x);
```

```
void in_out(const int& x);
```

```
void in(string x);
```

```
void in(string& x);
```

```
void in(const string& x);
```

```
void in_out(const string& x);
```

```
void in_out(string& x);
```

```
void loadQuijote(string& x)
{
  x = “En un lugar de la mancha,...”;
}

void g()
{
  string s = “......”;
  loadQuijote(s);
}
```

F.17: For "in-out" parameters, pass by reference to non-const

F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const

2.c) Parameters or arguments? Pointers and references.

`T*` `T&` `std::unique_ptr<T>` `std::shared_ptr<T>`

I just care for the value. Smart pointers imply some kind of ownership.

R.30: Take smart pointers as parameters only to explicitly express lifetime semantics

```
void just_use(T*);
```

```
void just_use_not_null(T&);
```

```
void sink(unique_ptr<T>);
```

```
void reseat(unique_ptr<T>&);
```

```
void DONT(const unique_ptr<T>&);
```

```
void share(shared_ptr<T>)
{/* then move it inside */}
```

```
void reseat(shared_ptr<T>&);
```

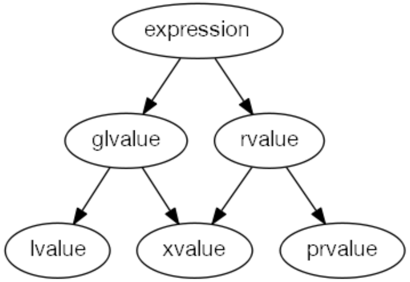
```
void maybe(const shared_ptr<T>&);
```

F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed
F.27: Use a `shared_ptr<T>` to share ownership
F.7: For general use, take `T*` or `T&` arguments rather than smart pointers
F.22: Use `T*` or `owner<T*>` to designate a single object
F.60: Prefer `T*` over `T&` when "no argument" is a valid option

2.e) Parameters or arguments? Moving parameters.

```
struct Y; // has move constructor

struct X
{
  template<typename A, typename B>
  X(A&& a, B&& b):a_{std::forward<A>(a)},b_{std::forward<B>(b)} {}

  Y a_;
  Y b_;
};

template<typename A, typename B>
X factory(A&& a, B&& b)
{
  return X(std::forward<A>(a), std::forward<B>(b));
}

int main()
{
  Y y;
  X two = factory(y, Y());
}
```

Probably we will need a talk to discuss about value categories, move semantics...



`std::move` does not move.

```
void sink(vector<int>&& v)
{
  store_somewhere(std::move(v));
}
```

`std::forward` does not forward.

```
template<class F, class... Args>
inline auto invoke(F f, Args&&... Args)
{
  return f(std::forward<Args>(args)...);
}
```

F.18: For "consume" parameters, pass by `X&&` and `std::move` the parameter
F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter

3) `return` of the type.

We've ignored the "out" parameters.

```
int f(const string& input, /*output only*/ string& output_data)
{
  int status;
  // ...
  output_data = something();
  return status;
}
```

```
string s;
int status = f("things", s);
```

```
string s;
if(f("things", s))
  //do something with s
```

Structure binding. Copy elision.

```
std::tuple<int,string> f(const string& input)
{
  int status;
  // ...
  return make_tuple(status,something());
}
```

```
int status;
struct s;
tie(status,s) = f("things");
```

```
auto [status,s] = f("things");
```

```
if(auto [status,s] = f("things"), status)
  //do something with s
```

```
unordered_map<int,int> m;
for(auto& entry : m)
{
  auto& key = entry->first;
  auto& value = entry->second;
  // do stuff with key and value
}
```

```
unordered_map<int,int> m;
for(auto& [key,value] : m)
// do stuff with key and value
```

F.21: To return multiple "out" values, prefer returning a tuple or struct
F.20: For "out" output values, prefer return values to output parameters

3.a) `return` of the type. Pointers and references.

`T*`

To point...

```
Node* find(Node*  t, int id);
```

It does not transfer ownership.

`T&`

To reference...

```
class Foo
{
  public:
  Foo& operator=(const Foo& r)
  {
    //...
    return *this;
  }
};
```

`std::unique_ptr<T>`

To transfer ownership...

```
unique_ptr<Shape> get_shape(istream& is)
{
  auto kind = read_header(is);
  switch (kind)
  {
    case kCircle:
      return make_unique<Circle>(is);
    case kTriangle:
      return make_unique<Triangle>(is);
  };
}
```

`std::shared_ptr<T>`

To share ownership...

By default, prefer using unique_ptr, and only use shared_ptr when it is obvious and essential to use them.

F.26: Use a `unique_ptr<T>` to transfer ownership where a pointer is needed
F.27: Use a `shared_ptr<T>` to share ownership
F.42: Return a `T*` to indicate a position (only)
F.47: Return `T&` from assignment operators
F.44: Return a `T&` when copy is undesirable and "returning no object" isn't needed

3.b) `return` of the type. Pointers and references: don'ts.

```
int* f()
{
  int fx = 9;
  return &fx;
}

void g(int *p)
{
  int gx;
  cout << *p << endl;
  *p = 999;
  cout << gx << endl;
}

void h()
{
  int *p = f();
  int z = *p;
  g(p);
}
```

```
int& f()
{
  int fx = 9;
  return fx;
}
```

Output:
999
999

```
int* glob;

template<class T>
void steal(T x)
{
  glob = x(); // BAD
}

void f()
{
  int i = 99;
  steal([&] { return &i; })
}

int main()
{
  f();
  cout << *glob << '\n';
}
```

```
template<class F>
auto&& wrapper(F f)
{
  log_call(typeid(f));
  return f();
}
```

```
template<class F>
auto wrapper(F f)
{
  log_call(typeid(f));
  return f();
}
```

F.45: Don't return a T&&

F.43: Never (directly or indirectly) return a pointer or a reference to a local object

2 + 3) Parameters, arguments and return

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don’t know (e.g., unfamiliar type, template)</th><th>Expensive to move
(e.g., BigPOD[],
array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td rowspan="2">f(X&)</td></tr>
<tr><td>In/Out</td><td colspan="2">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2" rowspan="2">f(const X&)</td></tr>
<tr><td>In & retain “copy”</td></tr>
</table>

“Cheap” ≈ a handful of hot int copies
“Moderate cost” ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

F.15: Prefer simple and conventional ways of passing information

2 + 3) Parameters, arguments and return (advanced)

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don’t know (e.g., unfamiliar type, template)</th><th>Expensive to move
(e.g., BigPOD[],
array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td></td></tr>
<tr><td>In/Out</td><td colspan="3">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2">f(const X&)</td></tr>
<tr><td>In & retain copy</td><td>f(const X&) + f(X&&) & move **</td><td></td></tr>
<tr><td>In & move from</td><td colspan="3">f(X&&) **</td></tr>
</table>

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

F.15: Prefer simple and conventional ways of passing information

4) Concerning lambdas.

```
[](){}
```

Have no name, but you can give one.
Can capture local variables.

```
int main()
{
  const int i = std::rand();
  const std::string s = [&](){
    switch(i%2)
    {
      case 0:
        return “long string is mod 0”;
      case 1:
        return “long string is mod 1”;
    }
  }();
}
```

```
template <class ForwardIt>
void quicksort(ForwardIt first, ForwardIt last)
{
 if(first == last) return;
 auto pivot = *std::next(first, std::distance(first,last)/2);
 auto middle1 = std::partition(first, last,
                  [pivot](const auto& em){ return em < pivot; });
 auto middle2 = std::partition(middle1, last,
                  [pivot](const auto& em){ return !(pivot < em); });
 quicksort(first, middle1);
 quicksort(middle2, last);
}
```

F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)

4.a) Concerning lambdas. Capturing...

```
[=](){}

[&](){}

[thing = out](){}

[&thing](){}
```

```
std::for_each(begin(sockets),end(sockets), [message](auto& socket)
{
  socket.send(message);
});
```

```
std::for_each(begin(sockets),end(sockets), [&message](auto& socket)
{
  socket.send(message);
});
```

```
auto createSum5()
{
  int i = 5;
  return [&i](int b){ return b + i; };
});
```

```
auto createSum5()
{
  int i = 5;
  return [i](int b){ return b + i; };
});
```

F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread

F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms

4.b) Concerning lambdas. Capturing `this`...

```
[this](){}
```

```
class wrong
{
  int x = 0;
  void f()
  {
    int i = 0;
    auto lambda = [=]()
                  {use(i, x);};

    x = 42;
    lambda(); // calls use(0,42)
};
```

```
class wrong
{
  int x = 0;
  void f()
  {
    int i = 0;
    auto lambda = [i,this]()
                  {use(i, x);};

    x = 42;
    lambda(); // calls use(0,42)
};
```

F.54: If you capture `this`, capture all variables explicitly (no default capture)

Summing up

0) What is a function?

F.46: `int` is the return type for `main()`.

1) Encapsulate your code. Efficiently and expressively!

F.1: “Package” meaningufl operations as carefully named functions. F.2: A function should perform a single logical operation. F.3: Keep functions short and simple.
F.4: If a function may have to be evaluated at compile time, declare it `constexpr`. F.5: If a function is very small and time-critical, declare it `inline`.
F.6: If your function may not throw, declare it `noexcept`.

2) Arguments or parameters? In, In/out, Pointers?

F.8: Prefer pure functions. F.9: Unused parameters should be unnamed.
F.51: Where there is a choice, prefer default arguments over overloading.

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don’t know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain “copy”			

3) `return` of the type. OUT! Tuples? References?

F.21: To return multiple "out" values, prefer returning a `tuple` or `struct`.

4) Concering lambas. Captures copy and reference.

F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function).
F.54: If you capture `this`, capture all variables explicitly (no default capture).
F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms.
F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread.

Next topic?

P: Philosophy
F: Functions
I: Interfaces
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
E: Error handling
Per: Performance

Con: Constants and immutability
T: Templates and generic programming
CP: Concurrency
SL: The Standard library
SF: Source files
CPL: C-style programming
Pro: Profiles
N: Non-Rules and myths
NL: Naming and layout

References

CppCoreGuidelines (duh!): Link
cppreference.com: Link
Using Types Effectively - *Ben Deane* - CppCon 2016 [slides] [video]
Rvalue references and move semantics in C++11: Link
C++ std::move and std::forward: Link