C++ function design 1

Richel Bilderbeek

October 6, 2015

Functions

- Rules, rules, rules:
- general
- return type
- name
- arguments
- modifiers: constexpt, inline, noexcept
- Applying these
- 9 implementations of `divide`

```
inline constexpr double pi() noexcept;
const T& f(const T& t);
```

F.2

```
void say_hello_and_wait_for_key_press();

int main() {
  say_hello_and_wait_for_key_press();
}
```

F.2

```
void say_hello();
void wait_for_key_press();

int main() {
  say_hello();
  wait_for_key_press();
}
```

- F.2: A function should perform a single logical operation

F.3

```
void f() {
  // 100 lines of code
}
```

F.3

```
void f() {
  g();
  h();
  i();
}
```

- F.3: Keep functions short and simple
- Length: should fit on a screen, 1-5 lines is normal
- Complexity: cyclomatic complexity less than 10

F.5: when to use inline

```
inline void a() { /* One line */ }
inline void b() { /* Two lines */ }
inline void c() { /* Three lines */ }
inline void d() { /* Five lines */ }
inline void e() { /* Ten lines */ }
```

F.5: when to use inline

```
inline void a() { /* One line */ }
inline void b() { /* Two lines */ }
inline void c() { /* Three lines */ }
inline void d() { /* Five lines */ }
inline void e() { /* Ten lines */ }
```

- F.5: If a function is very small and time critical, declare it inline
- Measure!
- There are standards that suggest to always inline below 2,3,5 and 10 lines
- C++ Core Guidelines: 3 lines is max

F.15

Argument passing

- in, e.g. `const T&` as function argument
- out, e.g. `T` as return type
- in/out, e.g. `T&` as function argument

```
void say(const std::string& text);
int get_pin_code();
void sort(std::vector<int>& v);
```

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

F.15 for in-only parameters

T = cheap to copy

```
void a(      T  t);
void b(const T  t);
void c(const T& t);
```

F.15 for in-only parameters

T = cheap to copy

```
void a(      T  t); //Common, copy is const or non-const
void b(const T  t); //Yes, if copy is const
void c(const T& t); //No, use copy
```

- F.21: Use a T parameter for a small object
- Because `sizeof(T&)` > `sizeof(T)`

F.15 for in-only parameters

T = expensive to copy

```
void a(      T  t);
void b(const T  t);
void c(const T& t);
```

F.15 for in-only parameters

T = expensive to copy

```
void a(      T  t); //Yes, if copy is needed and modified
void b(const T  t); //No, expensive to create a copy
                    //that is not modified
void c(const T& t); //Yes, if original is needed
```

- F.20: Use a `const T&` parameter for a large object
- Because `sizeof(T&)` < `sizeof(T)`

F.15 for in-out parameters

T can be anything

```
void f(T& t);
```

Example:

```
void set_to_zero(int& x) { x = 0; }
void sort(std::vector<int>& v);
```

- F.22: Use a T& for an in-out-parameter

F.15 for out-only parameters

Assume T is small.

```
T a();
T& b();
void c(T&); //Make it in-out
```

F.15 for out-only parameters

Assume T is small.

```
T a(); //Yes
T& b(); //No! Dangerous!
void c(T&); //No
```

- F.40: Prefer return values to out-parameters

F.15 for out-only parameters

Assume T is big.

```
T a();
T& b();
void c(T&); //Make it in-out
```

F.15 for out-only parameters

Assume T is big.

```
T a(); //No, expensive
T& b(); //No! Dangerous!
void c(T&); //Yes, make it in-out
```

- F.23: Use T& for an out-parameter that is expensive to move (only)

F.15 for out-only parameters

Assume T is big.

```
T a(); //No, expensive
T& b(); //No! Dangerous!
void c(T&); //Yes, make it in-out
```

- F.23: Use T& for an out-parameter that is expensive to move (only)

Exception:

```
std::ostream& operator<<(
  std::ostream& os,
  const T& t
);
```

I.10

```
//Returns the error code:
// 0: success
// 1: error
int display_temperature(const double kelvin) noexcept;
```

I.10

```
//Throws std::logic_error if kelvin < 0.0
void display_temperature(const double kelvin);
```

- I.10: Use exceptions to signal a failure to perform a required task

F.41 multiple out parameters

```
using V = std::vector<double>;
//Functions to calculate the mean and standard deviation
void a(const V& v, double& mean, double& stddev);
double b(const V& v, double& mean);
double c(const V& v, double& stddev);
std::pair<double,double> d(const V& v);
std::tuple<double,double> e(const V& v);
V f(const V& v);
```

F.41 multiple out parameters

```
using V = std::vector<double>;
std::tuple<double,double> e(const V& v);
```

- F.41: Prefer to return tuples to multiple out-parameters

When to use a const return type?

- Assume T is small

```
const T a();
      T b();
```

When to use a const return type?

- C++98: Yes, as its helps catch errors
- C++11: No, as it hinders rvalue optimalization

const T return type in C++98 1/2

```
struct Int {
  Int(const int any_i = 0) : i(any_i) {}
  operator bool() const { return i==0; }
  int i;
};

/* const */ Int operator+(const Int& lhs, const Int& rhs)
{
  return lhs.i + rhs.i;
}
```

const T return type in C++98 2/2

```
#include <cassert>
#include <iostream>

int main() {
  Int a;
  Int b;
  Int c;
  if (a+b=c) {
    assert(!"Should have used const");
  }
}
```

- Scott Meyers. Effective C++ (3rd edition).ISBN: 0-321-33487-6. Item 3: ‘Use const whenever possible’

F.6: need noexcept?

```
bool is_zero(const int x);
bool is_even(const int x);
bool is_prime(const int x);
double get_square_root(const double x);
int count_urls(const std::string& html_filename);
```

F.6: need noexcept?

- F.6: If your function may not throw, declare it noexcept
- When in doubt: do not mark it `noexcept` (RJCB)

```
bool is_zero(const int x) noexcept { return x == 0; }
bool is_even(const int x) noexcept { return x % 2 == 0; }
bool is_prime(const int x) noexcept;
```

F.6: need noexcept?

- F.6: If your function may not throw, declare it noexcept
- `noexcept` is most useful for frequently used, low-level functions.
- When in doubt: do not mark it `noexcept` (RJCB)

```
//Should throw for x <= 0.0
double get_square_root(const double x);

//Should throw when file does not exist
int count_urls(const std::string& html_filename);
```

F.4: when to use constexpr

```
constexpr double pi() noexcept;
constexpr double square(const double x) noexcept {
  return x * x;
}
constexpr int min(int x, int y) noexcept {
  return x < y ? x : y;
}
constexpr int factorial(const int n) noexcept;
```

F.4: when to use constexpr

```
constexpr double pi() noexcept;
constexpr double square(const double x) noexcept {
  return x * x;
}
constexpr int min(int x, int y) noexcept {
  return x < y ? x : y;
}
constexpr int factorial(const int n) noexcept;
```

- F.4: If a function may have to be evaluated at compile time, declare it constexpr
- A `constexpr` can have no side-effects
- A `constexpr` can only call `constexpr` functions
- Still limited in C++11

I.4

```
void draw_rect(int, int, int, int);
draw_rect(p.x, p.y, 10, 20);
```

I.4

```
void draw_rectangle(Point top_left, Point bottom_right);
void draw_rectangle(Point top_left, Size height_width);

// two corners
draw_rectangle(p, Point{10, 20});

// one corner and a (height, width) pair
draw_rectangle(p, Size{10, 20});
```

- I.4: Make interfaces precisely and strongly typed
- Scott Meyers. Effective C++ (3rd edition). ISBN: 0-321-33487-6. Item 18: Make interfaces easy to use correctly and hard to use incorrectly.

I.4

```
void blink_led(int time_to_blink) {
  // do something with time_to_blink
}

void use() {
  blink_led(2);
}
```

I.4

```
using Duration
  = std::chrono::duration<double>;

void blink_led(const Duration time_to_blink) {
  // do something with time_to_blink
}

void use() {
  blink_led(std::chrono::milliseconds(1500));
}
```

- I.4: Make interfaces precisely and strongly typed
- Scott Meyers. Effective C++ (3rd edition). ISBN: 0-321-33487-6. Item 18: Make interfaces easy to use correctly and hard to use incorrectly.

F.1

```
// Set a value in a y-x-ordered 2D-vector
void checked_set_value(
  std::vector<std::vector<int>>& v,
  const int y,
  const int x,
  const double value
);
```

F.1

```
// Set a value in a y-x-ordered 2D-vector
void checked_set_value(
  std::vector<std::vector<int>>& v,
  const int x,
  const int y,
  const double value
);
```

- People expect to first supply an x.
- F.1: “Package” meaningful operations as carefully named functions

F.1

```
// Assign a color to a certain
// square on a Rubiks' cube
void SetSquare(
  const Square& s,
  const Color& c,
  RubiksCube& c
);
```

F.1

```
void Turn(
  const Position& p,
  const Direction& d,
  RubiksCube& c
) noexcept;
```

- Scott Meyers. Effective C++ (3rd edition). ISBN: 0-321-33487-6. Item 18: Make interfaces easy to use correctly and hard to use incorrectly.

Divide

- How many are incorrect?

```
// n: numerator
// d: denominator
// [args]: const double n, const double d
double divide_a([args]);
double divide_b([args]) noexcept;
std::vector<double> divide_c([args]);
std::vector<double> divide_d([args]) noexcept;
std::vector<double>& divide_e([args]) noexcept;
std::tuple<bool,double> divide_f([args]);
std::tuple<bool,double> divide_g([args]) noexcept;
std::unique_ptr<double> divide_h([args]);
std::unique_ptr<double> divide_i([args]) noexcept;
```

divide_a

```
double
divide_a(const double n, const double d) {
  if (d == 0.0)
    throw std::logic_error(
      "Cannot divide by zero"
    );
  return n/d;
}
```

divide_b

```
double
divide_b(const double n, const double d) noexcept {
  assert(d != 0.0);
  return n/d;
}
```

divide_c

```
std::vector<double>
divide_c(const double n, const double d) {
  if (d == 0.0)
    return std::vector<double>();
  return { n / d };
}
```

divide_d

- Cannot guarantee a noexcept here

```
std::vector<double>
divide_d(const double n, const double d) noexcept {
  if (d == 0.0)
    return std::vector<double>();
  return { n / d }; //May throw here
}
```

divide_e

```
std::vector<double>&
divide_e(const double n, const double d) noexcept {
  static std::vector<double> no_result;
  static std::vector<double> result(1,0.0);
  if (d == 0.0)
    return no_result;
  result[0] = n / d;
  return result;
}
```

divide_f

- Can be guaranteed not to throw, see `divide_g`

```
std::tuple<bool,double>
divide_f(const double n, const double d);
```

divide_g

```
std::tuple<bool,double>
divide_g(const double n, const double d) noexcept {
  if (d == 0.0)
    return std::make_tuple(false,0.0);
  return std::make_tuple(true, n / d);
}
```

divide_h

```
std::unique_ptr<double>
divide_h(const double n, const double d) {
  if (d == 0.0)
    return nullptr;
  return std::make_unique<double>(n / d); //May throw
}
```

divide_i

```
auto no_del() { return  [](double *) { ; }; }

using Ptr = std::unique_ptr<double, decltype(no_del())>;

Ptr divide_i(const double n, const double d) noexcept {
  static double result{0.0};
  if (d == 0.0) {
    Ptr p(nullptr,no_del());
    return p;
  }
  result = n / d;
  Ptr p(&result,no_del());
  return p;
}
```

Conclusion

- There are many rules to design a function
- The prototype of a function tells you a lot about the implementation

More function design

- Function design 2: `T*` and its cousins

Legal stuff