

C++ function design 1

Richel Bilderbeek

October 5, 2015

Functions

- general
- return type
- name
- arguments
- modifiers: constexpr, inline, noexcept

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

```
void say_hello_and_wait_for_key_press();

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

```
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

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

```
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

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  
)
```

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?

```
const T a();  
const T& b(); //Dangerous, but assume correct  
T const& c(); //Dangerous, but assume correct
```

When to use a const return type?

```
const T a();  
const T& b(); //Dangerous, but assume correct  
T const& b(); //Dangerous, but assume correct
```

const T return type?

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

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

```
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
- 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

When to use void main

```
void main() {  
    //Does not return anything,  
    //thus void is ok  
}
```


When to use void main

Never use void main, use int main instead

```
int main() { /* Implicitly returns zero */ }
```

- Herb Sutter. Exceptional C++. ISBN:0-201-61562-2. Item 21: void main() is nonstandard and nonportable.
- Bjarne Stroustrup's homepage
(http://www.research.att.com/~bs/bs_faq2.html#void-main):

The definition

```
void main() { /* ... */ }
```

is not and never has been C++, nor has it even been C.

- Function design 2: T* and its cousins

Copyright by Richel Bilderbeek, 2015



Figure 1: CC-BY-NC-SA

Download at :

`www.github.com/richelbilderbeek/
CppPresentations/cpp_function_design1.pdf`

GitHub

Figure 2: GitHub