POINTLESS POINTERS

HOW TO MAKE OUR INTERFACES EFFICIENT?

Mateusz Pusz

November 15, 2017

WHAT IS A POINTER?

WHAT IS A POINTER?

A **pointer** is a programming language data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address.

-- Wikipedia

WHAT IS A POINTER?

WHAT IS A POINTER?

<u>Pointers</u> due to their ambiguous nature are the source of most problems and bugs in C++ code. Lack of understanding between code architect and the user results in crashes, asserts and memory leaks!!!

-- Mateusz Pusz ☺

POINTERS MISUSE LEADS TO

POINTERS MISUSE LEADS TO

Null Pointer Dereference

POINTERS MISUSE LEADS TO

Null Pointer Dereference

Resource leaks

POINTERS MISUSE LEADS TO

Null Pointer Dereference

Resource leaks

Buffer Overflows

POINTERS MISUSE LEADS TO

Hangs!

Null Pointer Dereference

Resource leaks

Buffer Overflows

POINTERS MISUSE LEADS TO

Hangs!

Null Pointer Dereference

Resource leaks

Buffer Overflows

Crashes!

POINTERS MISUSE LEADS TO

Hangs!

Null Pointer Dereference

Resource leaks

Stability!!!

Buffer Overflows

Crashes!

POINTERS MISUSE LEADS TO

Hangs!

Null Pointer Dereference

Resource leaks

Security!!!

Stability!!!

Buffer Overflows

Crashes!

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
B* c(A* ptr) { /* ... */; return d(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
B* c(A* ptr) { /* ... */; return d(ptr); }
B* d(A* ptr) { /* ... */; return e(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
B* c(A* ptr) { /* ... */; return d(ptr); }
B* d(A* ptr) { /* ... */; return e(ptr); }
B* e(A* ptr) { /* ... */; return f(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
B* c(A* ptr) { /* ... */; return d(ptr); }
B* d(A* ptr) { /* ... */; return e(ptr); }
B* e(A* ptr) { /* ... */; return f(ptr); }
B* f(A* ptr) { /* ... */; return g(ptr); }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; return b(ptr); }
B* b(A* ptr) { /* ... */; return c(ptr); }
B* c(A* ptr) { /* ... */; return d(ptr); }
B* d(A* ptr) { /* ... */; return e(ptr); }
B* e(A* ptr) { /* ... */; return f(ptr); }
B* f(A* ptr) { /* ... */; return g(ptr); }
B* g(A* ptr) { /* ... */; return *ptr;   }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; assert(ptr); return b(ptr); }
B* b(A* ptr) { /* ... */; assert(ptr); return c(ptr); }
B* c(A* ptr) { /* ... */; assert(ptr); return d(ptr); }
B* d(A* ptr) { /* ... */; assert(ptr); return e(ptr); }
B* e(A* ptr) { /* ... */; assert(ptr); return f(ptr); }
B* f(A* ptr) { /* ... */; assert(ptr); return g(ptr); }
B* g(A* ptr) { /* ... */; assert(ptr); return *ptr;   }
```

PROBLEM DEFINITION

```
B* a(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return b(ptr); }
B* b(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return c(ptr); }
B* c(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return d(ptr); }
B* d(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return e(ptr); }
B* e(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return f(ptr); }
B* f(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return g(ptr); }
B* g(A* ptr) { /* ... */; if(!ptr) throw std::invalid_argument{""}; return *ptr;   }
```

QUIZ - GUESS WHAT?

```
B* func(A* arg);
```

QUIZ - GUESS WHAT?

```
person* add(name* n);
```

QUIZ - GUESS WHAT?

```
person* add(name* n);
```

Is it better now?
Is it enough?
Do you know how to *use* that function?
Do you know how to *implement* that function?

ADD() USAGE – TAKE #1

```
person* add(name* n);

void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;
}
```

ADD() USAGE – TAKE #1

```
person* add(name* n);

void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;
}
```

Is that a valid usage of **add()** interface?

What are potential problems with above code?

ADD() USAGE – TAKE #2

```
person* add(name* n);

void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());
}
```

ADD() USAGE – TAKE #2

```
person* add(name* n);

void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());
}
```

Is that a valid usage of **add()** interface?

ADD() USAGE – TAKE #3

```
person* add(name* n);

void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

ADD() USAGE – TAKE #3

```
person* add(name* n);

void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

Is that a valid usage of **add()** interface?

What are potential problems with above code?

ADD() USAGE – TAKE #4

```
person* add(name* n);

void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

ADD() USAGE – TAKE #4

```
person* add(name* n);

void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

Is that a valid usage of **add()** interface?

What are potential problems with above code?

WHICH ONE IS CORRECT?

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

4

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

QUIZ - MATCH IMPLEMENTATION

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
```

```
std::deque<person> people;

person* add(name* n)
{
  people.emplace_back((n != nullptr) ? *n : "anonymous");
  return &people.back();
}
```

```
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

QUIZ - MATCH IMPLEMENTATION

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

```
person* add(name* n)
{
  assert(n);
  int num = 0;
  for(auto ptr = n; *ptr != ""; ++ptr)
    ++num;
  person* p = new person[num];
  for(auto i = 0; i<num; ++i)
    p[i].name(n[i]);
  return p;
}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  )
  (), p->name());
```

```
{"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

QUIZ - MATCH IMPLEMENTATION

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
```

```
person* add(name* n)
{
  assert(n != nullptr);
  return new person{n};
}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

QUIZ - MATCH IMPLEMENTATION

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

```
std::deque<person> people;

person* add(name* n)
{
  if(n == nullptr)
    return nullptr;
  auto it = find_if(begin(people), end(people),
                    [&](person& p)
                    { return p.name() == *n; });
  if(it != end(people))
    return nullptr;
  people.emplace_back(*n);
  return &people.back();
}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};

  ->name());

```

```
eusz Pusz", "Jan Kowalski", ""};
(names);
assert(people != nullptr);
for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
  process(people[i].id(), people[i].name());
delete[] people;
}
```

QUIZ - MATCH IMPLEMENTATION

1

```
void foo()
{
  person* p = add(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;

}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());


}
```

3

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

4

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::unique_ptr<person>
add(std::unique_ptr<name> n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::unique_ptr<person>
add(std::unique_ptr<name> n);
```

Do you know how to *use* that function?
Do you know how to *implement* that function?

USING C++ THE RIGHT WAY – CASE #1

```
person*
add(name* n)
{
  assert(n != nullptr);
  return new person{n};
}
```

```
void foo()
{
  person* p = add(
    new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;
}
```

USING C++ THE RIGHT WAY – CASE #1

```
person*
add(name* n)
{
  assert(n != nullptr);
  return new person{n};
}
```

```
void foo()
{
  person* p = add(
    new name{"Mateusz Pusz"});
  assert(p != nullptr);
  process(p->id(), p->name());
  delete p;
}
```

```
std::unique_ptr<person>
add(std::unique_ptr<name> n)
{
  assert(n != nullptr);
  return std::make_unique<person>(std::move(n));
}
```

```
void foo()
{
  auto p = add(
    std::make_unique<name>("Mateusz Pusz"));
  assert(p != nullptr);
  process(p->id(), p->name());

}
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
person&
add(std::optional<name> n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
person&
add(std::optional<name> n);
```

Do you know how to *use* that function?
Do you know how to *implement* that function?

USING C++ THE RIGHT WAY – CASE #2

```
std::deque<person> people;

person* add(name* n)
{
  people.emplace_back(
    (n != nullptr) ? *n : "anonymous");
  return &people.back();
}
```

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());
}
```

USING C++ THE RIGHT WAY – CASE #2

```
std::deque<person> people;

person* add(name* n)
{
  people.emplace_back(
    (n != nullptr) ? *n : "anonymous");
  return &people.back();
}
```

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = add(&n);
  if(p != nullptr)
    process(p->id(), p->name());
}
```

```
std::deque<person> people;

person& add(std::optional<name> n)
{
  people.emplace_back(
      n ? std::move(*n) : "anonymous");
  return people.back();
}
```

```
void foo()
{
  person& p = add(name{"Mateusz Pusz"});
  process(p.id(), p.name());


}
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::tuple<person&, bool>
add(name n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::tuple<person&, bool>
add(name n);
```

Do you know how to *use* that function?
Do you know how to *implement* that function?

USING C++ THE RIGHT WAY – CASE #3

```
std::deque<person> people;

person* add(name* n)
{
  if(n == nullptr) return nullptr;
  auto it = find_if(
       begin(people), end(people),
       [&](person& p) { return p.name() == *n; });
  if(it != end(people))
    return nullptr;
  people.emplace_back(*n);
  return &people.back();
}
```

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

USING C++ THE RIGHT WAY – CASE #3

```
std::deque<person> people;

person* add(name* n)
{
  if(n == nullptr) return nullptr;
  auto it = find_if(
       begin(people), end(people),
       [&](person& p) { return p.name() == *n; });
  if(it != end(people))
    return nullptr;
  people.emplace_back(*n);
  return &people.back();
}
```

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = add(n);
  if(p != nullptr)
    process(p->id(), p->name());
  delete n;
}
```

```
std::deque<person> people;

std::tuple<person&, bool> add(name n)
{
  auto it = find_if(
       begin(people), end(people),
       [&](person& p) { return p.name() == n; });
  if(it != end(people))
    return { *it, false };
  people.emplace_back(std::move(n));
  return { people.back(), true };

}
```

```
void foo()
{
  auto r = add(name{"Mateusz Pusz"});
  if(std::get<bool>(r))
    process(std::get<0>(r).id(), std::get<0>(r).name());


}
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::vector<person>
add(std::vector<name> names);
```

IS THERE A BETTER SOLUTION?

```
person*
add(name* n);
```

```
std::vector<person>
add(std::vector<name> names);
```

Do you know how to *use* that function?
Do you know how to *implement* that function?

USING C++ THE RIGHT WAY – CASE #4

```
person* add(name* n)
{
  assert(n);
  int num = 0;
  for(auto ptr = n; *ptr != ""; ++ptr)
    ++num;
  person* p = new person[num];
  for(auto i = 0; i<num; ++i)
    p[i].name(n[i]);
  return p;
}
```

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

USING C++ THE RIGHT WAY – CASE #4

```
person* add(name* n)
{
  assert(n);
  int num = 0;
  for(auto ptr = n; *ptr != ""; ++ptr)
    ++num;
  person* p = new person[num];
  for(auto i = 0; i<num; ++i)
    p[i].name(n[i]);
  return p;
}
```

```
std::vector<person> add(std::vector<name> names)
{
  std::vector<person> p;
  p.reserve(names.size());
  for(auto& n : names)
    p.emplace_back(std::move(n));
  return p;



}
```

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = add(names);
  assert(people != nullptr);
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    process(people[i].id(), people[i].name());
  delete[] people;
}
```

```
void foo()
{
  auto people = add({"Mateusz Pusz", "Jan Kowalski"});
  for(auto& p : people)
    process(p.id(), p.name());



}
```


No Excuses
Excuses

POINTERS USAGE IN ANSI C

ARGUMENT TYPE	POINTER ARGUMENT DECLARATION
Mandatory big value	`void foo(A* in);`
Output function argument	`void foo(A* out);`
Array	`void foo(A* array);`
Optional value	`void foo(A* opt);`
Ownership passing	`void foo(A* ptr);`

POINTERS USAGE IN ANSI C

ARGUMENT TYPE	POINTER ARGUMENT DECLARATION
Mandatory big value	`void foo(A* in);`
Output function argument	`void foo(A* out);`
Array	`void foo(A* array);`
Optional value	`void foo(A* opt);`
Ownership passing	`void foo(A* ptr);`

Pointer ambiguity makes it really hard to understand the intent of the interface author.

DOING IT C++ WAY

ARGUMENT TYPE	POINTER ARGUMENT DECLARATION
Mandatory big value	`void foo(const A& in);`
Output function argument	`A foo();` or `std::tuple<...> foo();` or `void foo(A& out);`
Array	`void foo(const std::vector<A>& a);`
Optional value	`void foo(std::optional<A> opt);` or `void foo(A* opt);`
Ownership passing	`void foo(std::unique_ptr<A> ptr);`

DOING IT C++ WAY

ARGUMENT TYPE	POINTER ARGUMENT DECLARATION
Mandatory big value	`void foo(const A& in);`
Output function argument	`A foo();` or `std::tuple<...> foo();` or `void foo(A& out);`
Array	`void foo(const std::vector<A>& a);`
Optional value	`void foo(std::optional<A> opt);` or `void foo(A* opt);`
Ownership passing	`void foo(std::unique_ptr<A> ptr);`

Use above Modern C++ constructs to explicitly state your design intent.

QUIZ – GUESS WHAT?

```
B foo(std::optional<A> arg);
```

```
const A& foo(const std::array<A, 3>& arg);
```

```
std::unique_ptr<B> foo(A arg);
```

```
std::vector<B> foo(const A& arg);
```

C++17 AND C++20: WHAT ABOUT POINTER AND A SIZE

ANSI C -> C++14

```
int count_lower(const char* ptr, size_t size);
```

ANSI C -> C++14

```
void add_2(int* ptr, size_t size);
```

C++17 AND C++20: WHAT ABOUT POINTER AND A SIZE

ANSI C -> C++14

```
int count_lower(const char* ptr, size_t size);
```

C++17

```
int count_lower(std::string_view txt);
```

ANSI C -> C++14

```
void add_2(int* ptr, size_t size);
```

C++20

```
void add_2(std::span<int> array);
```

TAKEAWAYS

C++ IS NOT ANSI C!!!

C++ is a powerful tool:

- strong type system
- better abstractions
- templates
- C++ STD library

CAUTION
Programming is addictive (and too much fun)