Pointless Pointers

A short talk about efficient C++ interfaces

Mateusz Pusz
Intel Technology Poland

To the extent possible under law, Intel Corporation has waived all copyright and related or neighboring rights to this work.

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?

Pointers 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 ☺

Pointer misuse leads to

Hangs!

Null Pointer Dereference

Resource leaks

Security!!!

Stability!!!

Crashes!

Buffer Overflows

Quiz – Guess what?

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

Quiz – Guess what?

```
person* Register(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?*

Register() usage – Take #1

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

```
void foo()
{
  person* p = Register(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  Process(p->Id(), p->Name());
  delete p;
}
```

Is that a valid usage of Register() interface?

What are potential problems with above code?

Register() usage – Take #2

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

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = Register(&n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
}
```

Is that a valid usage of Register() interface?

Register() usage – Take #3

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

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = Register(n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
  delete n;
}
```

Is that a valid usage of Register() interface?

What are potential problems with above code?

Register() usage – Take #4

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

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = Register(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 Register() interface?

What are potential problems with above code?

Which one is correct?

1
```
void foo()
{
  person* p = Register(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  Process(p->Id(), p->Name());
  delete p;
}
```

2
```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = Register(&n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
}
```

3
```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = Register(n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
  delete n;
}
```

4
```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = Register(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 = Register(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  Process(p->Id(), p->Name());
  delete p;
}
```

2

```
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = Register(&n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
```

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

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

3

```
void foo()
{
  name name
  person* p
  assert(people
  for(int i=0; i<sizeof(names)/sizeof(*names) - 1; ++i)
    Process(people[i].Id(), people[i].Name());
  delete[] people;
}
```

```
Pusz"};
    Process(p->Id(), p->Name());
  delete n;
}
```

Quiz – Match Implementation

1

```
void foo()
{
  person* p = Register(new name{"Mateusz Pusz"});
  assert(p != nullp
  Process(p->Id(),
  delete p;
}
```

```
person* Register(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[] = {"
  person* people =
  assert(people !=
  for(int i=0; i<si
    Process(people
  delete[] people;
}
```

3

```
w name{"Mateusz Pusz"};
Register(n);
ptr)
>Id(), p->Name());
}
```

Quiz – Match Implementation

1

```
void foo()
{
  person* p = Register(new name{"Mateusz Pusz"});
  assert(p != nullptr);
  Process(p->Id(), p->Name());
  delete p;
}
```

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

3

```
)
  name* n = new name{"Mateusz Pusz"};
  person* p = Register(n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
  delete n;
}
```

Quiz – Match Implementation

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

person* Register(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

Is there any better solution?

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

```
std::unique_ptr<person>
Register(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*
Register(name* n)
{
  assert(n != nullptr);
  return new person{n};
}
```

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

```
void foo()
{
  person* p = Register(
     new name{"Mateusz Pusz"});
  assert(p != nullptr);
  Process(p->Id(), p->Name());
  delete p;
}
```

```
void foo()
{
  auto p = Register(
     std::make_unique<name>("Mateusz Pusz"));
  assert(p != nullptr);
  Process(p->Id(), p->Name());
}
```

Is there any better solution?

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

```
person&
Register(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

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

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

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

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

```cpp
void foo()
{
  name n{"Mateusz Pusz"};
  person* p = Register(&n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
}
```

```cpp
void foo()
{
  person& p =
     Register(name{"Mateusz Pusz"});
  Process(p.Id(), p.Name());
}
```

Is there any better solution?

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

```
std::pair<person&, bool>
Register(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* Register(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();
}
```

```
std::deque<person> people;
std::pair<person&, bool> Register(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(move(n));
  return { people.back(), true };
}
```

```
void foo()
{
  name* n = new name{"Mateusz Pusz"};
  person* p = Register(n);
  if(p != nullptr)
    Process(p->Id(), p->Name());
  delete n;
}
```

```
void foo()
{
  auto r =
    Register(name{"Mateusz Pusz"});
  if(r.second)
    Process(r.first.Id(), r.first.Name());
}
```

Is there any better solution?

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

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

```
void foo()
{
  name names[] = {"Mateusz Pusz", "Jan Kowalski", ""};
  person* people = Register(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 = Register(
    {"Mateusz Pusz", "Jan Kowalski"});
  for(auto& p : people)
    Process(p.Id(), p.Name());
}
```

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	void foo(**A&** out); or **A** foo();
Array	void foo(**const std::vector<A>&** array);
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)
```

TAKE AWAYS

C++ is not ANSI C!!!

It is a powerful tool

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

Let's speak C++!!!

```
A
A& / const A&
std::optional<A>
std::vector<A>
std::unique_ptr<A>
std::array<A, 256>
std::shared_ptr<A>
A* / const A*
```

Questions?

Thank you

Happy coding!!!

Backup