Zarządzanie pamięcią w C++

Bartosz ’BaSz’ Szurgot
Kamil Szatkowski
Łukasz Ziobroń

bartosz.1.szurgot@nokia.com

kamil.szatkowski@nokia.com

lukasz.ziobron@nokia.com

March 21, 2017

Plan

Agenda

Proste pytanie

```
1  String EvaluateSalaryAndReturnName(Employee e)
2  {
3    if( e.Title() == "CEO" || e.Salary() > 100000 )
4    {
5      cout << e.First() << "␣" << e.Last()
6           << "␣is␣overpaid" << endl;
7    }
8    return e.First() + "␣" + e.Last();
9  }
```

- Ile jest możliwych ścieżek wykonania tego kodu?

Proste pytanie

```
1  String EvaluateSalaryAndReturnName(Employee e)
2  {
3    if( e.Title() == "CEO" || e.Salary() > 100000 )
4    {
5      cout << e.First() << "␣" << e.Last()
6           << "␣is␣overpaid" << endl;
7    }
8    return e.First() + "␣" + e.Last();
9  }
```

- Ile jest możliwych ścieżek wykonania tego kodu?
- 23 (dwadzieścia trzy)

Proste pytanie

```
1  String EvaluateSalaryAndReturnName(Employee e)
2  {
3    if( e.Title() == "CEO" || e.Salary() > 100000 )
4    {
5      cout << e.First() << "␣" << e.Last()
6           << "␣is␣overpaid" << endl;
7    }
8    return e.First() + "␣" + e.Last();
9  }
```

- Ile jest możliwych ścieżek wykonania tego kodu?
- 23 (dwadzieścia trzy)
- (przykład - Herb Sutter [1], GotW#20)

Spróbujmy teraz z zasobami

```
1  MyData* recreate(MyData* p)
2  {
3    auto tmp = new MyData(*p);
4    delete p;
5    return tmp;
6  }
7
8  MyData* doStuff1(void)
9  {
10   MyData md;
11   process(&md);
12   return recreate(&md);
13 }
14
15 MyData* doStuff2(void)
16 {
17   MyData* md = new MyData[2];
18   process(md[0]);
19   process(md[1]);
20   return recreate(md);
21 }
```

Spróbujmy teraz z zasobami

```
1  MyData* recreate(MyData* p)
2  {
3    auto tmp = new MyData(*p);
4    delete p;
5    return tmp;
6  }
7
8  MyData* doStuff1(void)
9  {
10   MyData md;
11   process(&md);
12   return recreate(&md);
13 }
14
15 MyData* doStuff2(void)
16 {
17   MyData* md = new MyData[2];
18   process(md[0]);
19   process(md[1]);
20   return recreate(md);
21 }
```

- Wycieki pamięci
- Polega na dokumentacji (kto usuwa i kiedy?)
- Nieodporne na przyszłe błędy
- Kompilator nie pomoże
- Kiepski design
- Niebezpieczne ze względu na wyjątki
- Prowadzi do błędów
- Po prostu nieprawidłowe

Smoki latają na wolności!

Agenda

Obsługa błędów w C++

Dawne metody obsługi błędów w C/C++:

- goto?
- kody błędów?

Obsługa błędów w C++

Dawne metody obsługi błędów w C/C++:

- **goto?**
- kody błędów?

```
1  #include <iostream>
2  using namespace std;
3
4  bool isValid()
5  {
6    return false;
7  }
8
9  int main()
10 {
11   /* ... */
12
13   if(!isValid())
14   {
15     goto error;
16   }
17
18   /* ... */
19
20 error:
21   std::cerr << "Error occured" << std::endl;
22   return 1;
23 }
```

Obsługa błędów w C++

Dawne metody obsługi błędów w C/C++:

- goto?
- **kody błędów?**

```
1  #include <iostream>
2  using namespace std;
3
4  enum ErrorCode {
5    E_SUCCESS,
6    E_FAIL
7  };
8
9  bool isValid() {
10   return false;
11 }
12
13 ErrorCode foo() {
14   if(!isValid()) {
15     return E_FAIL;
16   }
17   /* ... */
18   return E_SUCCESS;
19 }
20
21 int main() {
22   /* ... */
23   if(foo() == E_FAIL) {
24     return 1;
25   }
26   return 0;
27 }
```

Obsługa błędów w C++

Co z prawdziwym C++?:

- konstruktory,
- operatory?

Obsługa błędów w C++

Co z prawdziwym C++?:

- konstruktory,
- operatory?

```
1   struct FileWrapper
2   {
3     FileWrapper(std::string const& filePath)
4       : m_file(fopen(filePath.c_str(), "rw"))
5     {
6       /* What if file did not open?
7       */
8     }
9
10    ~FileWrapper()
11    {
12      fclose(m_file);
13    }
14
15    FileWrapper & operator<<(std::string const& text)
16    {
17      /* What if file did not open?
18      */
19      fputs(text.c_str(), m_file);
20      return *this;
21    }
22
23  private:
24    FILE* m_file;
25  };
```

Obsługa błędów w C++

Co z prawdziwym C++?:

- konstruktory,
- operatory?

Wyjątki!

```
1   struct FileWrapper
2   {
3     FileWrapper(std::string const& filePath)
4       : m_file(fopen(filePath.c_str(), "rw"))
5     {
6       if(!m_file)
7       {
8         throw std::runtime_error("File not opened");
9       }
10    }
11
12    ~FileWrapper()
13    {
14      fclose(m_file);
15    }
16
17    FileWrapper & operator<<(std::string const& text)
18    {
19      /* Not validation needed because invalid
20         object cannot be created */
21      fputs(text.c_str(), m_file);
22      return *this;
23    }
24
25  private:
26    FILE* m_file;
27  };
```

try/catch - przykład

```
1   #include <iostream>
2   #include <stdexcept>
3   using namespace std;
4
5   void foo()
6   {
7     throw std::runtime_error("Error");
8   }
9
10  int main()
11  {
12    try
13    {
14      foo();
15    }
16    catch(std::runtime_error const&)
17    {
18      std::cout << "std::runtime_error" << std::endl;
19    }
20    catch(std::exception const& ex)
21    {
22      std::cout << "std::exception: " << ex.what() << std::endl;
23    }
24    catch(...)
25    {
26      std::cerr << "unknown exception" << std::endl;
27    }
28  }
```

try/catch - przykład

```
1   #include <iostream>
2   #include <stdexcept>
3   using namespace std;
4
5   void foo()
6   {
7     throw std::runtime_error("Error");
8   }
9
10  int main()
11  {
12    try
13    {
14      foo();
15    }
16    catch(std::runtime_error const&)
17    {
18      std::cout << "std::runtime_error" << std::endl;
19    }
20    catch(std::exception const& ex)
21    {
22      std::cout << "std::exception: " << ex.what() << std::endl;
23    }
24    catch(...)
25    {
26      std::cerr << "unknown exception" << std::endl;
27    }
28  }
```

Wynik wykonania:

- std::runtime_error

Jak wyjątki działają?

- rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku *try/catch* z pasującą klauzulą *catch*,

Jak wyjątki działają?

- rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku *try/catch* z pasującą klauzulą *catch*,
- wyjątek jest dopasowywany do każdej z klauzul *catch* zgodnie z kolejnością ich deklaracji,

Jak wyjątki działają?

- rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku *try/catch* z pasującą klauzulą *catch*,
- wyjątek jest dopasowywany do każdej z klauzul *catch* zgodnie z kolejnością ich deklaracji,
- wyjątek jest niszczony gdy program opuszcza blok *catch* bez ponownego rzucenia wyjątku.

Jak wyjątki działają?

- rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku *try/catch* z pasującą klauzulą *catch*,
- wyjątek jest dopasowywany do każdej z klauzul *catch* zgodnie z kolejnością ich deklaracji,
- wyjątek jest niszczony gdy program opuszcza blok *catch* bez ponownego rzucenia wyjątku.

```
1  struct TalkingObject {
2    TalkingObject() {
3      std::cout << "Default␣constructor" << std::endl; }
4    TalkingObject(TalkingObject const& src) {
5      std::cout << "Copy␣constructor" << std::endl; }
6    TalkingObject(TalkingObject && src) {
7      std::cout << "Move␣constructor" << std::endl; }
8    ~TalkingObject() { std::cout << "Destructor" << std::
9  };
10 void foo() { throw std::runtime_error("Error"); }
11
12 int main() {
13   TalkingObject out;
14   try {
15     TalkingObject inside;
16     foo();
17   } catch(std::runtime_error const& ex) {
18     std::cout << "std::runtime_error:␣" << ex.what() <<
19   } catch(std::exception const&) {
20     std::cout << "std::exception" << std::endl;
21   }
22 }
23 // >>> Output:
24 // Default constructor
25 // Default constructor
26 // Destructor
27 // std::runtime_error: Error
28 // Destructor
```

Jak wyjątki działają?

- wyjątek rzucony ponownie znów startuje mechanizm odwijania stosu, który działa aż do napotkania kolejnego bloku *try/catch* z pasującą klauzulą *catch*,

Jak wyjątki działają?

- wyjątek rzucony ponownie znów startuje mechanizm odwijania stosu, który działa aż do napotkania kolejnego bloku *try/catch* z pasującą klauzulą *catch*,
- klauzula *catch* dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.

Jak wyjątki działają?

- wyjątek rzucony ponownie znów startuje mechanizm odwijania stosu, który działa aż do napotkania kolejnego bloku *try/catch* z pasującą klauzulą *catch*,
- klauzula *catch* dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() { throw std::runtime_error("Error"); }
4
5  void bar() {
6    try {
7      TalkingObject inside;
8      foo();
9    } catch(std::exception const&) {
10     std::cout << "std::exception" << std::endl;
11     throw;
12   }
13 }
14 int main() {
15   TalkingObject outside;
16   try {
17     bar();
18   } catch(std::runtime_error const& ex) {
19     std::cout << "std::runtime_error:␣" << ex.what() <<
20   }
21 }
22 // >>> Output:
23 // Default constructor
24 // Default constructor
25 // Destructor
26 // std::exception
27 // std::runtime_error: Error
28 // Destructor
```

Jak wyjątki działają?

- wyjątek, który nie został złapany przez żaden blok *try/catch* powoduje zatrzymanie programu poprzez wykonanie metody *std::terminate()*

Jak wyjątki działają?

- wyjątek, który nie został złapany przez żaden blok *try/catch* powoduje zatrzymanie programu poprzez wykonanie metody *std::terminate()*

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() {
4    throw std::runtime_error("Error");
5  }
6
7  void bar() {
8    try {
9      TalkingObject inside;
10     foo();
11   } catch(std::exception const&) {
12     std::cout << "std::exception" << std::endl;
13     throw;
14   }
15 }
16
17 int main() {
18   TalkingObject outside;
19   bar();
20 }
21
22 // >>> Output:
23 // Default constructor
24 // Default constructor
25 // Destructor
26 // std::exception
27 // >> abort() <<
```

Jak wyjątki działają?

- wyjątek rzucony podczas działania mechanizmu odwijania stosu powoduje zatrzymanie programu poprzez zawołanie metody *std::terminate()*

Jak wyjątki działają?

- wyjątek rzucony podczas działania mechanizmu odwijania stosu powoduje zatrzymanie programu poprzez zawołanie metody *std::terminate()*

```
1  struct TalkingObject { /*...*/ };
2
3  struct ThrowingObject {
4    ~ThrowingObject() {
5        throw std::runtime_error("error in destructor");
6    }
7  };
8
9
10 void foo() {
11   throw std::runtime_error("Error");
12 }
13
14 int main() {
15   TalkingObject outside;
16   try {
17     ThrowingObject inside;
18     foo();
19   } catch(std::exception const&) {
20     std::cout << "std::exception" << std::endl;
21     throw;
22   }
23 }
24
25 // >>> Output:
26 // Default constructor
27 // >> abort() <<
```

function-try-block

- blok *try/catch* może zostać zastosowany do całego ciała funkcji, jest to tzw. *function-try-block*

function-try-block

- blok *try/catch* może zostać zastosowany do całego ciała funkcji, jest to tzw. *function-try-block*

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() {
4    throw std::runtime_error("Error");
5  }
6
7  void bar()
8  try {
9    TalkingObject inside;
10   foo();
11 } catch(std::exception const&) {
12   std::cout << "std::exception" << std::endl;
13 }
14
15 int main() {
16   TalkingObject outside;
17   bar();
18 }
19
20 // >>> Output:
21 // Default constructor
22 // Default constructor
23 // Destructor
24 // std::exception
25 // Destructor
```

Rekomendacje

- łap wyjątki przy pomocy ***const&***,
- twórz własne wyjątki jako dziedziczące po wyjątkach standardowych,
- używaj wyjątków tylko w sytuacjach wyjątkowych.

Agenda

RAII

- Resource Acquisition Is Initialization [2]
  - Każdy zasób ma konkretny handler
  - Pozyskanie w konstruktorze
  - Zwolnienie w destruktorze
  - (Przemyśl kopiowanie i przenoszenie)

RAII

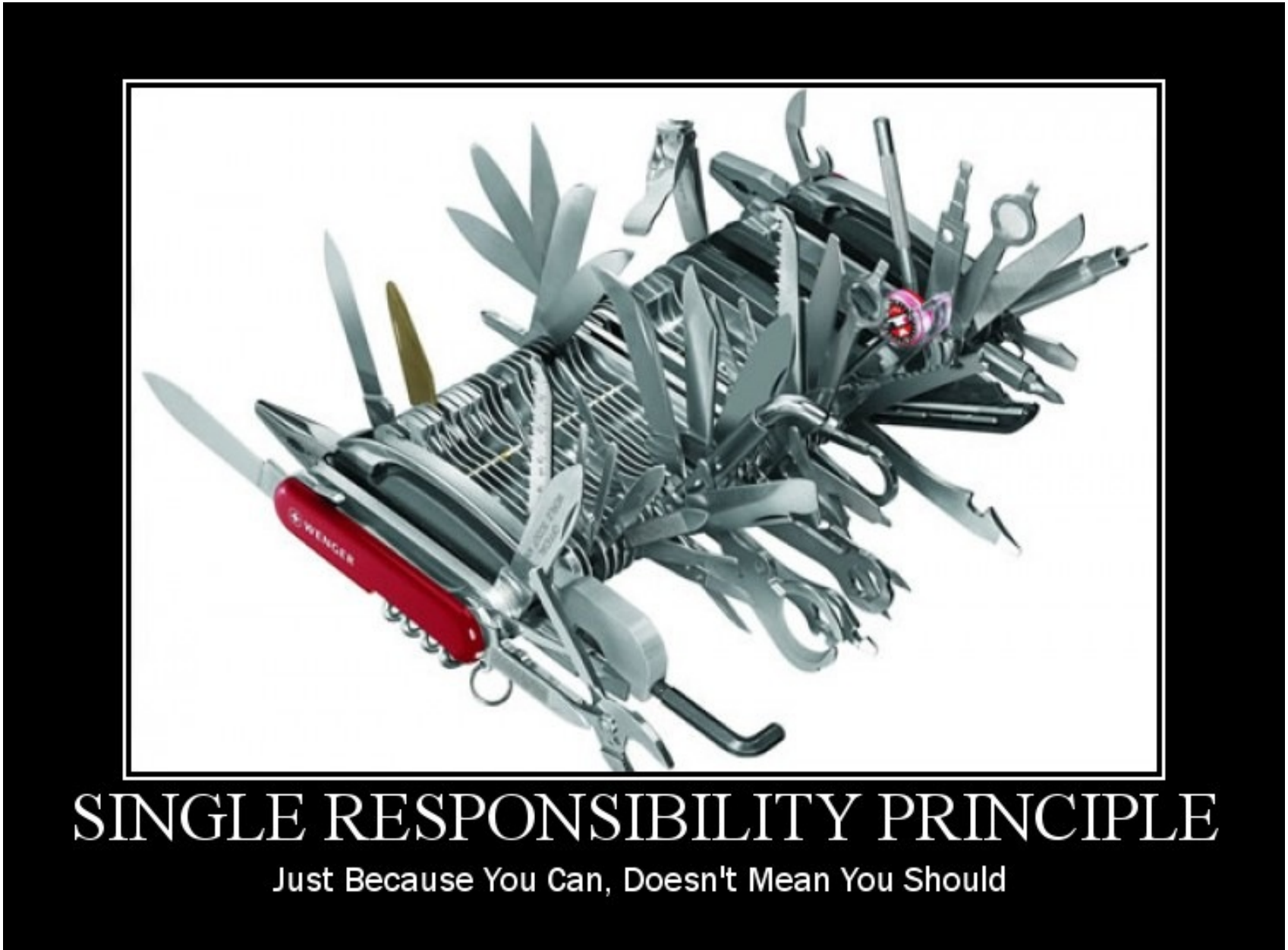
- Resource Acquisition Is Initialization [2]
  - Każdy zasób ma konkretny handler
  - Pozyskanie w konstruktorze
  - Zwolnienie w destruktorze
  - (Przemyśl kopiowanie i przenoszenie)
- Korzyści
  - Gwarancja poprawności na poziomie języka (!)
  - Krótszy kod (automatyzacja)
  - Jasna odpowiedzialność

RAII

- Resource Acquisition Is Initialization [2]
  - Każdy zasób ma konkretny handler
  - Pozyskanie w konstruktorze
  - Zwolnienie w destruktorze
  - (Przemyśl kopiowanie i przenoszenie)
- Korzyści
  - Gwarancja poprawności na poziomie języka (!)
  - Krótszy kod (automatyzacja)
  - Jasna odpowiedzialność
- Większa wydajność niż w językach z VM
  - Stosuje się do każdego zasobu (GC tylko dla pamięci)
  - Nie potrzeba sekcji *finally*
  - Przewidywalny czas zwalniania

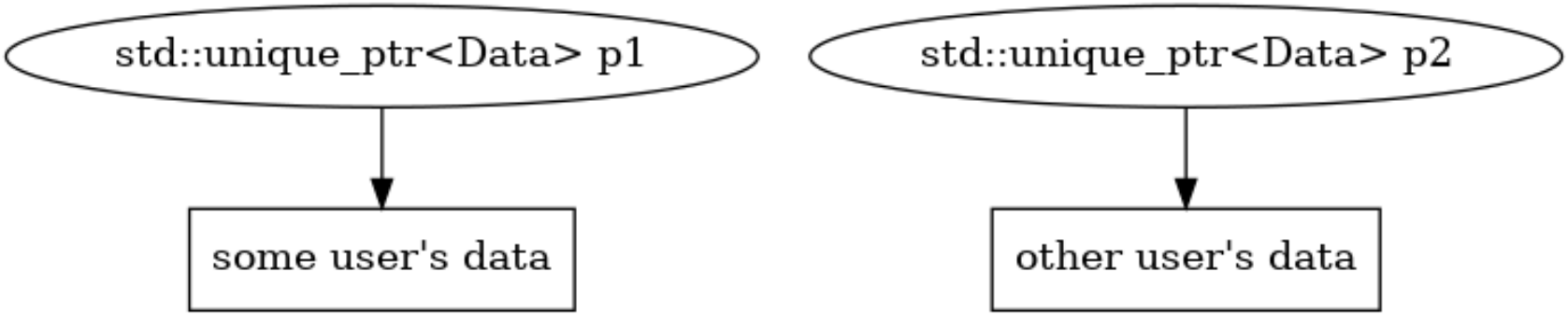
Co poprzednio było źle?

Co poprzednio było źle?

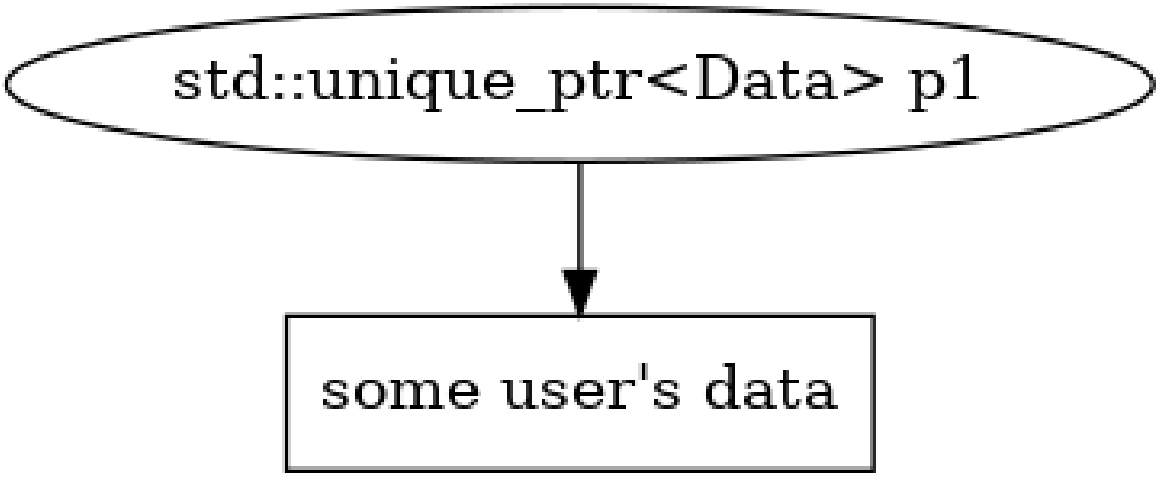


Jeden właściciel

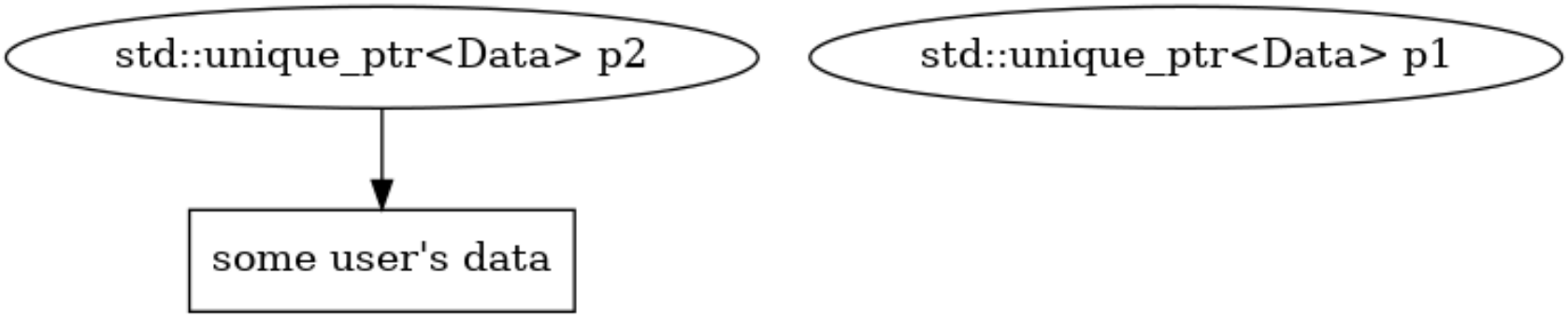
- Najprostszy przypadek
- Jeden obiekt == jeden właściciel
- Przenoszenie możliwe
- Własny deleter możliwy



Przenoszenie własności (p1->p2)



Przenoszenie własności (p1->p2)



Unique pointer

```
1
2
3
4
5
6  MyData* create(void)
7  {
8    auto md = new MyData;
9    process(md);
10   return md;
11 }
```

- Wyciek pamięci

Unique pointer

```
1
2
3
4
5
6  MyData* create(void)
7  {
8    auto md = new MyData;
9    process(md);
10   return md;
11 }
```

- Wyciek pamięci

```
1  #include <memory>
2
3  using DataPtr =
4    std::unique_ptr<MyData>;
5
6  DataPtr create(void)
7  {
8    DataPtr md{new MyData};
9    process( md.get() );
10   return md;
11 }
```

- Ok

Unique pointer - kontynuacja

```
1  using DataPtr =
2    std::unique_ptr<MyData>;
3
4  DataPtr source(void);
5  void sink(DataPtr ptr);
6
7  void simpleUsage(void)
8  {
9    source();
10
11   sink( source() );
12   auto ptr = source();
13   // sink(ptr);   ERROR
14   sink( std::move(ptr) );
15
16   auto p1 = source();
17   //auto p2 = p1; ERROR
18   auto p2 = std::move(p1);
19   //p1 = p2;      ERROR
20   p1 = std::move(p2);
21 }
22
23 void collections(void)
24 {
25   std::vector<DataPtr> v;
26   v.push_back( source() );
27
28   auto tmp = source();
29   //v.push_back(tmp); ERROR
30   v.push_back( std::move(tmp) );
31
32   //sink( v.at(0) );  ERROR
33   sink( std::move(v.at(0)) );
34 }
```

Unique pointer - przykłady

```
1  void f()
2  {
3      std::unique_ptr<Gadget> my_gadget {new Gadget()};
4      my_gadget->use(); // this code may throw exception
5      std::unique_ptr<Gadget> your_gadget = std::move(my_gadget);
6  } // Destructor of std::unique_ptr will execute the delete for pointer
7
8
9  auto ptr = std::make_unique<Gadget>(arg); // C++14 only
10
11 void sink(std::unique_ptr<Gadget> gdgt)
12 {
13     gdgt->call_method();
14     // sink takes ownership - deletes the object pointed by gdgt
15 }
16
17 sink(std::move(ptr)); // explicitly moving into sink
18
19 // pointers to derived classes - SuperGadget derives from Gadget
20 std::unique_ptr<Gadget> pb = std::make_unique<SuperGadget>();
21 auto pb = std::unique_ptr<Gadget>{ std::make_unique<SuperGadget>() };
```

Unique pointer i tablice

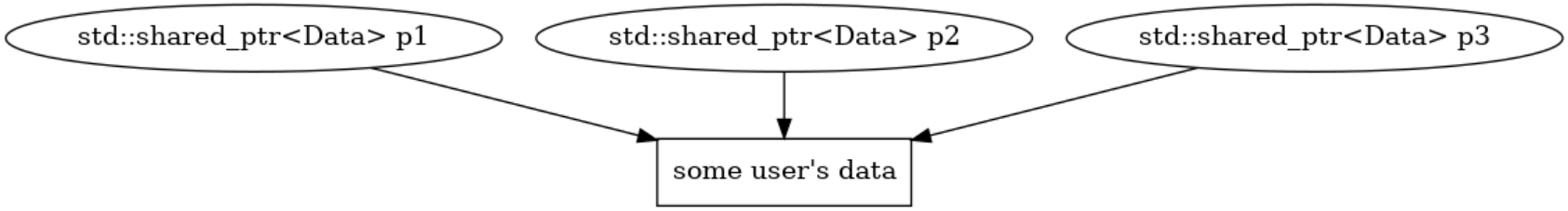
- Podczas niszczenia:
  - *std::unique_ptr<Type>* wywołuje *delete*
  - *std::unique_ptr<Type**[]**>* wywołuje *delete **[]***

```
1 using Array = std::unique_ptr<MyData[]>;
2 void use(void)
3 {
4   Array tab{new MyData[42]};
5   process( tab.get() );
6   process(tab[13]);
7 }
```

- Możliwe
- Zazwyczaj *std::vector<Type>* jest lepszym wyborem

Własność współdzielona

- Brak pojedynczego właściciela
- Wielu użytkowników kodu
- Jeden obiekt == wielu właścicieli
- Przenoszenie i kopiowanie możliwe
- Ostatni sprząta
- Własny deleter możliwy
- Własny alokator możliwy



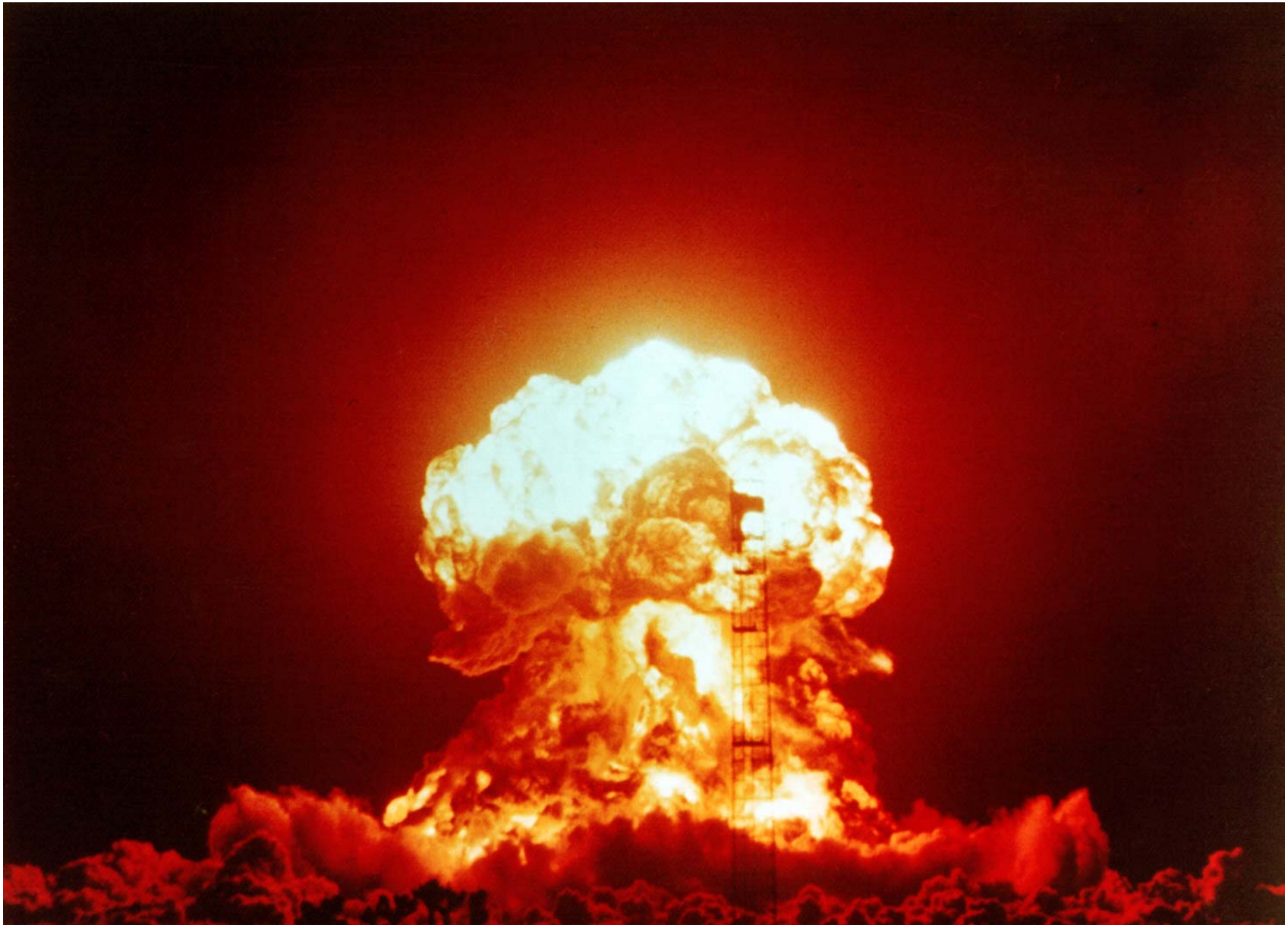
Shared pointer

```
1  using DataShPtr =
2    std::shared_ptr<MyData>;
3
4  DataShPtr source(void);
5  void sink(DataShPtr ptr);
6
7  void simpleUsage(void)
8  {
9    source();
10
11   sink( source() );
12   auto ptr = source();
13   sink(ptr);
14   sink( std::move(ptr) );
15
16   auto p1 = source();
17   auto p2 = p1;
18   auto p3 = std::move(p1);
19   p1 = p2;
20   p1 = std::move(p2);
21 }
22
23 void ownership(void)
24 {
25   auto other1 = source();
26   auto other2 = source();
27
28   auto same1 = source();
29   auto same2 = same1;
30 }
```

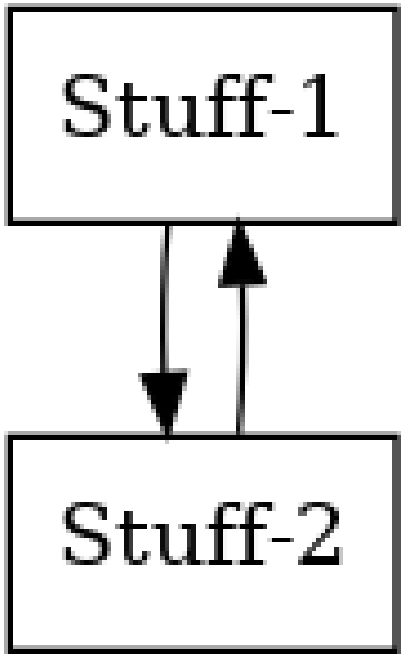
Co jeśli...

```
1  struct Stuff;
2  using StuffShPtr =
3    std::shared_ptr<Stuff>;
4
5  struct Stuff
6  {
7    StuffShPtr p_;
8  };
9
10 void useIt(void)
11 {
12   StuffShPtr p1{new Stuff};
13   StuffShPtr p2{new Stuff};
14   p1->p_ = p2;
15   p2->p_ = p1;
16 }
```

Uważaj na cykle

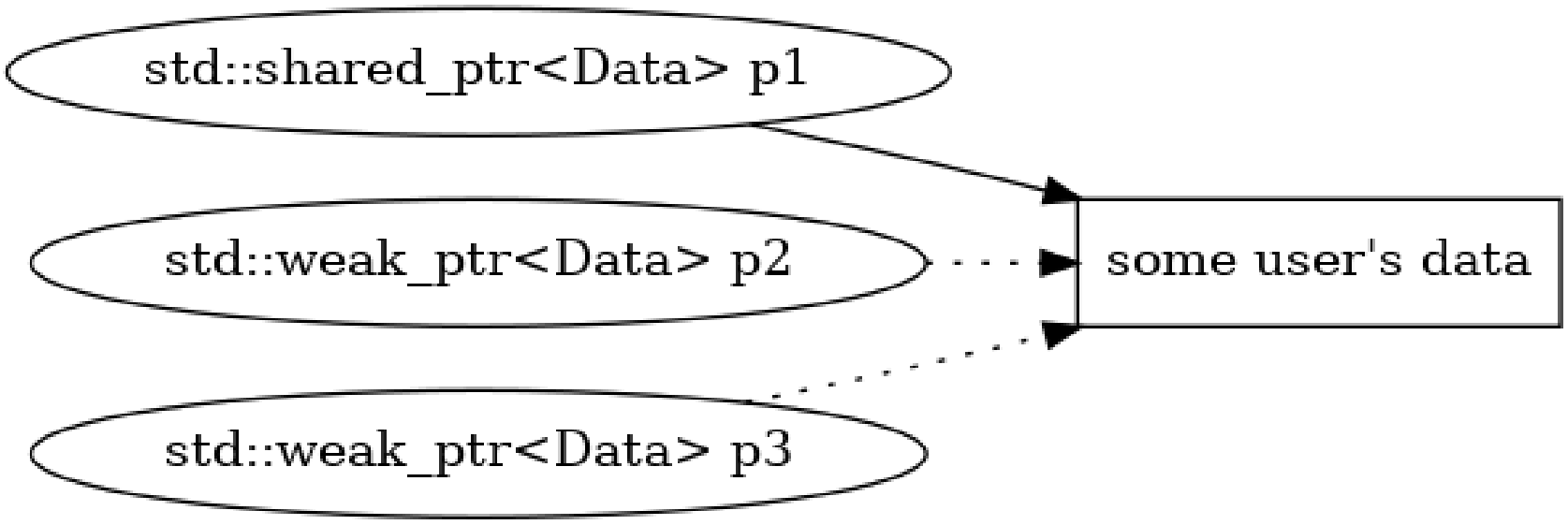


Schematycznie



Zapobieganie cyklom

- Nie jest właścicielem, jeśli nie trzeba
- Jest obserwatorem



Powrót do źródeł

```
1  struct Stuff;
2  using StuffShPtr =
3    std::shared_ptr<Stuff>;
4  using StuffWPtr =
5    std::weak_ptr<Stuff>;
6
7  struct Stuff
8  {
9    StuffWPtr p_;
10 };
11
12 void useIt(void)
13 {
14   StuffShPtr p1{new Stuff};
15   StuffShPtr p2{new Stuff};
16   p1->p_ = p2;
17   p2->p_ = p1;
18 }
19
20 void dereference(void)
21 {
22   StuffShPtr sp{new Stuff};
23   StuffWPtr  wp{sp};
24   auto       p = wp.lock();
25   if(p)
26     process(*p);
27 }
```

To trzeba zapomnieć

- C++98 dostarczał *std::auto_ptr<>*
- Kilka fixów w C++03
- Lecz ciągle. . .

To trzeba zapomnieć

- C++98 dostarczał *std::auto_ptr<>*
- Kilka fixów w C++03
- Lecz ciągle...

```
1 using Ptr = std::auto_ptr<int>;
2 Ptr p{new int{42}};
3 process(p);
4 // assert( p.get() ) ... or is it?
```

To trzeba zapomnieć

- C++98 dostarczał *std::auto_ptr<>*
- Kilka fixów w C++03
- Lecz ciągle...

```
1  using Ptr = std::auto_ptr<int>;
2  Ptr p{new int{42}};
3  process(p);
4  // assert( p.get() ) ... or is it?
```

```
1  using Ptr = std::auto_ptr<int>;
2  using Col = std::vector<Ptr>;
3  Col c;
4  Ptr p{new int{42}};
5  // c.push_back(p); ouch...
```

To trzeba zapomnieć

- C++98 dostarczał *std::auto_ptr<>*
- Kilka fixów w C++03
- Lecz ciągle...

```
1  using Ptr = std::auto_ptr<int>;
2  Ptr p{new int{42}};
3  process(p);
4  // assert( p.get() ) ... or is it?
```

```
1  using Ptr = std::auto_ptr<int>;
2  using Col = std::vector<Ptr>;
3  Col c;
4  Ptr p{new int{42}};
5  // c.push_back(p); ouch...
```

```
1  struct Base { };
2  struct Derived: public Base { };
3
4  std::auto_ptr<Derived> source();
5  void sink(std::auto_ptr<Base>);
6
7  void testIt(void)
8  {
9    // sink( source() ); // error :(
10 }
```

To trzeba zapomnieć

- C++98 dostarczał *std::auto_ptr<>*
- Kilka fixów w C++03
- Lecz ciągle...

```
1  using Ptr = std::auto_ptr<int>;
2  Ptr p{new int{42}};
3  process(p);
4  // assert( p.get() ) ... or is it?
```

```
1  using Ptr = std::auto_ptr<int>;
2  using Col = std::vector<Ptr>;
3  Col c;
4  Ptr p{new int{42}};
5  // c.push_back(p); ouch...
```

```
1  struct Base { };
2  struct Derived: public Base { };
3
4  std::auto_ptr<Derived> source();
5  void sink(std::auto_ptr<Base>);
6
7  void testIt(void)
8  {
9    // sink( source() ); // error :(
10 }
```

- *std::auto_ptr<>* deprecated w C++11
- Nie używać!
- Używać *std::unique_ptr<>*

RAII – więcej niż pamięć

- Co z poniższym kodem?

```
1  std::mutex g_mutex;
2  void work(void)
3  {
4    g_mutex.lock();
5    process(42);
6    g_mutex.unlock();
7  }
```

RAII – więcej niż pamięć

- Co z poniższym kodem?

```
1  std::mutex g_mutex;
2  void work(void)
3  {
4    g_mutex.lock();
5    process(42);
6    g_mutex.unlock();
7  }
```

- Blokada może nie zostać zwolniona (lock-leak)
- RAII-way fix:

```
1  std::mutex g_mutex;
2  void work(void)
3  {
4    std::lock_guard<std::mutex> lock(g_mutex);
5    process(42);
6  }
```

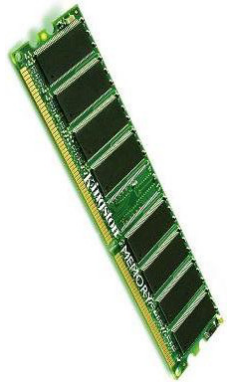
- Poprawne i krótsze!
- Papier N3830 – generic RAII handler

Wskazówki – pamięć

1. Domyślnie używaj *std::unique_ptr*
   - Może być trzymany w kolekcjach
   - Brak narzutu pamięciowego i czasu wykonania
   - Konwertuje się do *std::shared_ptr*

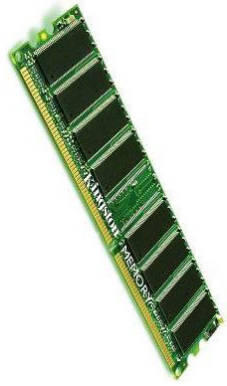
Wskazówki – pamięć

1. Domyślnie używaj *std::unique_ptr*
   - Może być trzymany w kolekcjach
   - Brak narzutu pamięciowego i czasu wykonania
   - Konwertuje się do *std::shared_ptr*
2. Dla współdzielonej własności - *std::shared_ptr*
   - Może być trzymany w kolekcjach
   - Wprowadza narzuty pamięci i czasu wykonania
   - Może być przekonwertowany na/z *std::weak_ptr*

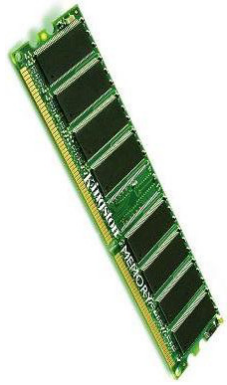


Wskazówki – pamięć

1. Domyślnie używaj *std::unique_ptr*
   - Może być trzymany w kolekcjach
   - Brak narzutu pamięciowego i czasu wykonania
   - Konwertuje się do *std::shared_ptr*
2. Dla współdzielonej własności - *std::shared_ptr*
   - Może być trzymany w kolekcjach
   - Wprowadza narzuty pamięci i czasu wykonania
   - Może być przekonwertowany na/z *std::weak_ptr*
3. Do obserwacji zasobu współdzielonego - *std::weak_ptr*
   - Może być trzymany w kolekcjach
   - Brak dodatkowego narzutu pamięci i czasu wykonania
   - Może być przekonwertowany na/z *std::shared_ptr*



Wskazówki – pamięć

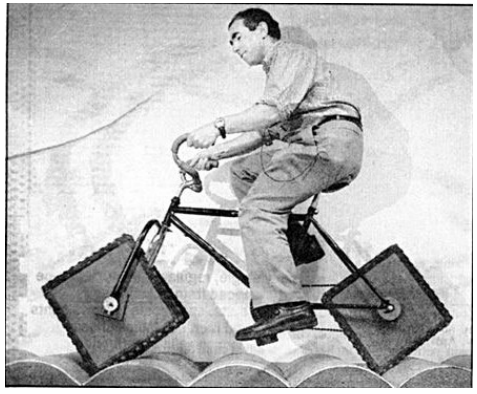


1. Domyślnie używaj *std::unique_ptr*
   - Może być trzymany w kolekcjach
   - Brak narzutu pamięciowego i czasu wykonania
   - Konwertuje się do *std::shared_ptr*
2. Dla współdzielonej własności - *std::shared_ptr*
   - Może być trzymany w kolekcjach
   - Wprowadza narzuty pamięci i czasu wykonania
   - Może być przekonwertowany na/z *std::weak_ptr*
3. Do obserwacji zasobu współdzielonego - *std::weak_ptr*
   - Może być trzymany w kolekcjach
   - Brak dodatkowego narzutu pamięci i czasu wykonania
   - Może być przekonwertowany na/z *std::shared_ptr*
4. Twórz je wykorzystując *std::make_shared* oraz *std::make_unique*
5. **Zwykły wskaźnik oznacza wyłącznie dostęp (brak własności)**
6. Używaj referencji zamiast wskaźnika jeśli to możliwe

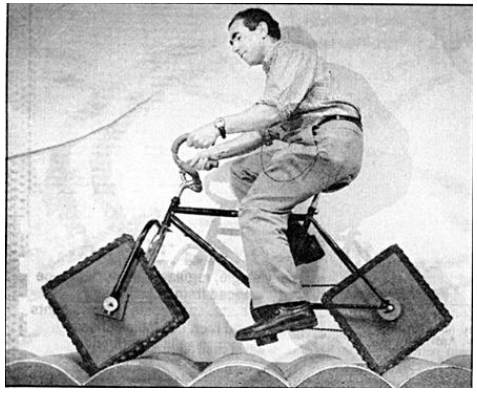
Agenda

Ogólne uwagi

- Nie wynajduj (kwadratowego) koła na nowo
- Używaj STL/Boost

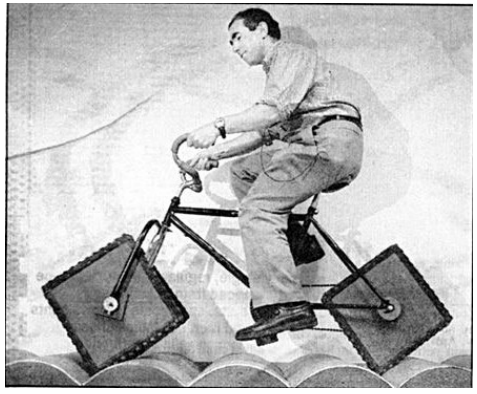


Ogólne uwagi



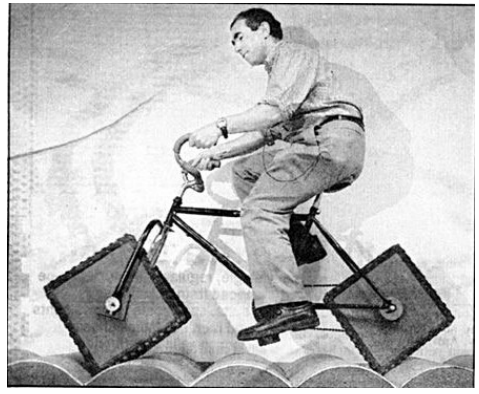
- Nie wynajduj (kwadratowego) koła na nowo
- Używaj STL/Boost
- Reguła 5
  - Destruktor
  - Kopiowanie przez: przypisanie, konstrukcję
  - Przenoszenie przez: przypisanie, konstrukcję
- Reguła zera (pamiętaj o zasadzie pojedynczej odpowiedzialności)

Ogólne uwagi



- Nie wynajduj (kwadratowego) koła na nowo
- Używaj STL/Boost
- Reguła 5
  - Destruktor
  - Kopiowanie przez: przypisanie, konstrukcję
  - Przenoszenie przez: przypisanie, konstrukcję
- Reguła zera (pamiętaj o zasadzie pojedynczej odpowiedzialności)
- Jawne wywoływanie delete
  - Problematyczne przez 99.9% czasu
  - Nie rób tego

Ogólne uwagi



- Nie wynajduj (kwadratowego) koła na nowo
- Używaj STL/Boost
- Reguła 5
  - Destruktor
  - Kopiowanie przez: przypisanie, konstrukcję
  - Przenoszenie przez: przypisanie, konstrukcję
- Reguła zera (pamiętaj o zasadzie pojedynczej odpowiedzialności)
- Jawne wywoływanie delete
  - Problematyczne przez 99.9% czasu
  - Nie rób tego
- Ręczna implementacja nie-domyślnego destruktora
  - Ręczne zarządzanie zasobami?
  - Problematyczne przez 90% czasu
    - Nie dotyczy niestandardowej obsługi zasobów
    - Czasami wykorzystywane jako przerwanie wątku

Nowe jest stare

- Unikaj jawnego użycia operatora *new*
- Widzisz problem tutaj?

```
1  using Ptr = std::shared_ptr<MyData>;
2  void sink(Ptr p1, Ptr p2);
3
4  void use(void)
5  {
6    sink( Ptr{new MyData{41}}, Ptr{new MyData{42}} );
7  }
```

Nowe jest stare

- Unikaj jawnego użycia operatora *new*
- Widzisz problem tutaj?

```
1  using Ptr = std::shared_ptr<MyData>;
2  void sink(Ptr p1, Ptr p2);
3
4  void use(void)
5  {
6    sink( Ptr{new MyData{41}}, Ptr{new MyData{42}} );
7  }
```

- Mała podpowiedź:

```
1  using Ptr = std::shared_ptr<MyData>;
2  void sink(Ptr p1, Ptr p2);
3
4  void use(void)
5  {
6    Ptr p1{new MyData{41}};
7    Ptr p2{new MyData{42}};
8    sink( std::move(p1), std::move(p2) );
9  }
```

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?
- *f1(), f2()* vs. *f2(), f1()* (zależne od implementacji)

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?
- *f1(), f2()* vs. *f2(), f1()* (zależne od implementacji)
- *auto p = new Data;* oznacza:
  1. Zaalokuj *sizeof(Data)* bajtów
  2. Zawołaj *Data::Data()* na tej pamięci
  3. Przypisz adres tej pamięci do *p*

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?
- *f1(), f2()* vs. *f2(), f1()* (zależne od implementacji)
- *auto p = new Data;* oznacza:
  1. Zaalokuj *sizeof(Data)* bajtów
  2. Zawołaj *Data::Data()* na tej pamięci
  3. Przypisz adres tej pamięci do *p*
- Co w przypadku dwóch takich operacji?

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?
- *f1(), f2()* vs. *f2(), f1()* (zależne od implementacji)
- *auto p = new Data;* oznacza:
  1. Zaalokuj *sizeof(Data)* bajtów
  2. Zawołaj *Data::Data()* na tej pamięci
  3. Przypisz adres tej pamięci do *p*
- Co w przypadku dwóch takich operacji?

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

- . . . w jednym *punkcie sekwencji*
- *f(Ptr(new Data), Ptr(new Data));*

Co się dzieje?

- Znasz przypadek *cout«f1()«f2();*?
- *f1(), f2()* vs. *f2(), f1()* (zależne od implementacji)
- *auto p = new Data;* oznacza:
  1. Zaalokuj *sizeof(Data)* bajtów
  2. Zawołaj *Data::Data()* na tej pamięci
  3. Przypisz adres tej pamięci do *p*
- Co w przypadku dwóch takich operacji?

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

1. Zaalokuj *sizeof(Data)*
2. Zawołaj *Data::Data()*
3. Przypisz do *p*

- . . . w jednym *punkcie sekwencji*
- *f(Ptr(new Data), Ptr(new Data));*
- Może wystąpić przeplot!
- Przykład: 1,2,1,2,3,3

Zastępstwo dla ’new’

- W skrócie: *std::make_shared*

```
1  using Ptr = std::shared_ptr<MyData>;
2  void sink(Ptr p1, Ptr p2);
3
4  void use(void)
5  {
6    sink( std::make_shared<MyData>(41),
7          std::make_shared<MyData>(42) );
8  }
```

Zastępstwo dla ’new’

- W skrócie: *std::make_shared*

```
1 using Ptr = std::shared_ptr<MyData>;
2 void sink(Ptr p1, Ptr p2);
3
4 void use(void)
5 {
6   sink( std::make_shared<MyData>(41),
7         std::make_shared<MyData>(42) );
8 }
```

- Wykorzystanie std::make_shared:
  - Naprawia poprzedni błąd
  - Nie powiela w kodzie konstruowanego typu
  - Zezwala na natychmiastowe wykorzystanie przy konstrukcji
  - Optymalizuje wykorzystanie pamięci! (sprawdź *std::allocate_shared*)

Wskazówki

1. Nie wynajduj koła na nowo
2. C++ to nie Java – czy naprawdę potrzebujesz sterty?
3. Zawsze wykorzystuj RAII do zasobów
   - Inteligentne wskaźniki dla pamięci
   - Własne dla własnych zasobów
   - Kod C raczej będzie potrzebował wrapperów
4. Nigdy nie używaj jawnie *delete*
5. Unikaj "własnych" destruktorów
6. Unikaj jawnego *new*

Praca z wodą – stara szkoła

- Praca w stylu C w C++

- Wszędzie wycieki
- Stałe unikanie katastrof

Praca z wodą – poprawione warunki pracy

- Praca w stylu C++ w C++

- Odkrycia i nauka
- Wycieki to elementy przeszłości

Agenda

Unique pointer

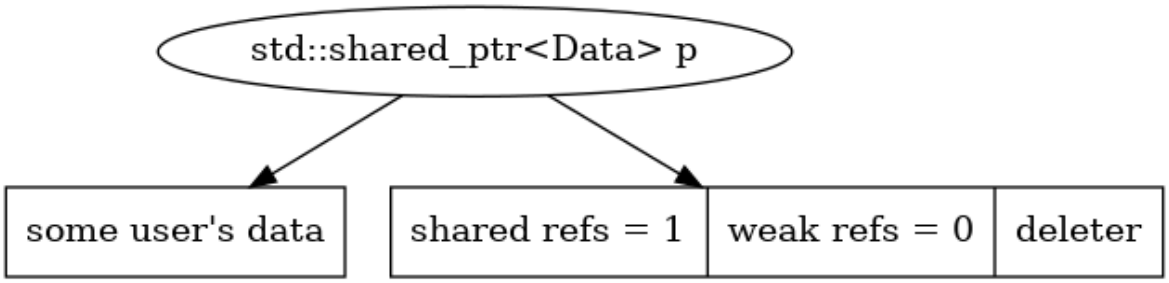
- Prosty wrapper trzymający obiekt wskazanej klasy
- Konstruktor kopiuje wskaźnik
- Destruktor wywołuje właściwe *delete*
- Brak operacji kopiowania
- Przenoszenie oznacza:
  1. Skopiowanie oryginalnego wskaźnika do nowego obiektu
  2. Ustawienie źródłowego wskaźnika na *nullptr*
- Wszystkie metody są *inline*
- Wymuszona składnia (move) zapobiega przypadkowym przypisaniom
- Pod spodem jest zwykłe kopiowanie wskaźników

Shared pointer

- Zawiera wskaźnik na obiekt wskazanej klasy
- . . . i 2 liczniki referencji:
  - Licznik shared pointerów
  - Licznik weak pointerów

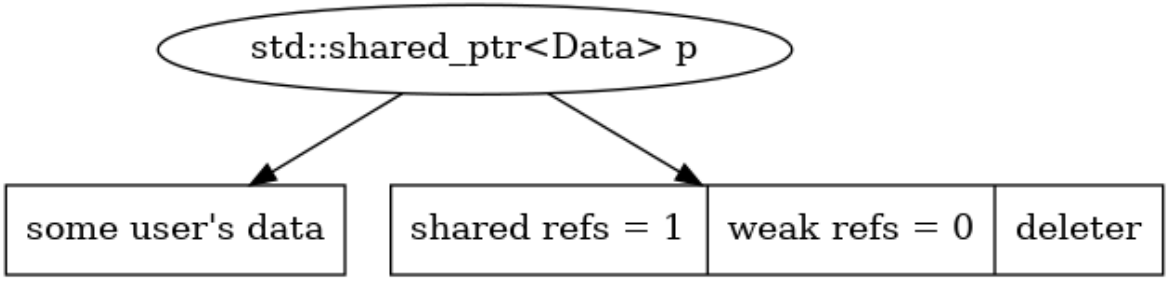
Shared pointer

- Zawiera wskaźnik na obiekt wskazanej klasy
- . . . i 2 liczniki referencji:
  - Licznik shared pointerów
  - Licznik weak pointerów



Shared pointer

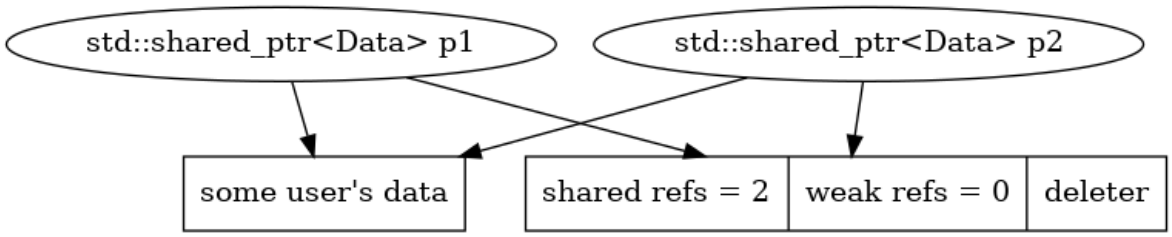
- Zawiera wskaźnik na obiekt wskazanej klasy
- . . . i 2 liczniki referencji:
  - Licznik shared pointerów
  - Licznik weak pointerów



- Destruktor:
  - Dekrementuje *shared-refcount*
  - Usuwa obiekt gdy *shared-refcount==0*
  - Usuwa liczniki gdy *shared-refcount==0* i *weak-refcount==0*
- Dodatkowe miejsce na deleter

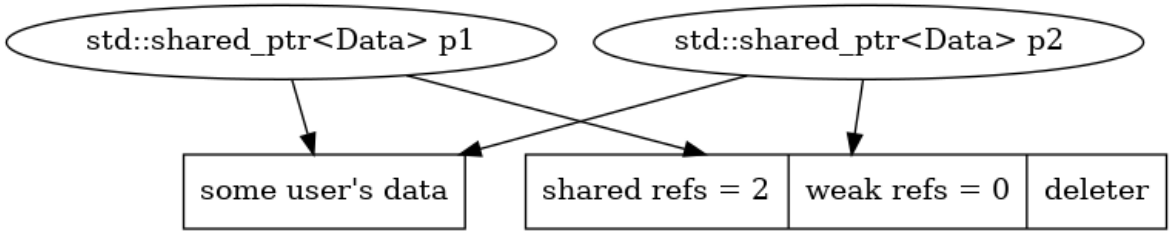
Shared pointer – kopiowanie i przenoszenie

- Kopiowanie oznacza:
  - Kopiowanie wskaźników
  - Inkrementację shared-refcount

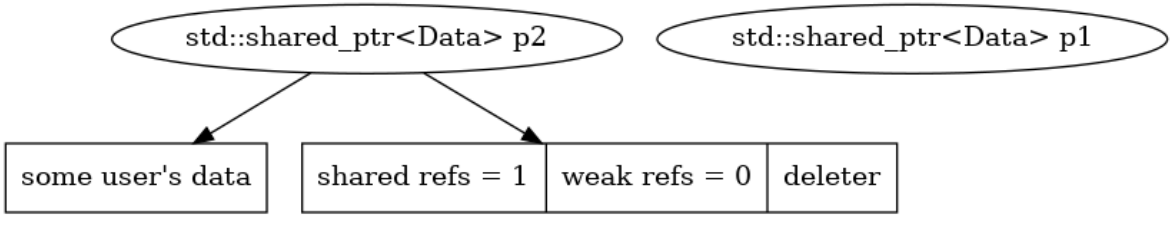


Shared pointer – kopiowanie i przenoszenie

- Kopiowanie oznacza:
  - Kopiowanie wskaźników
  - Inkrementację shared-refcount

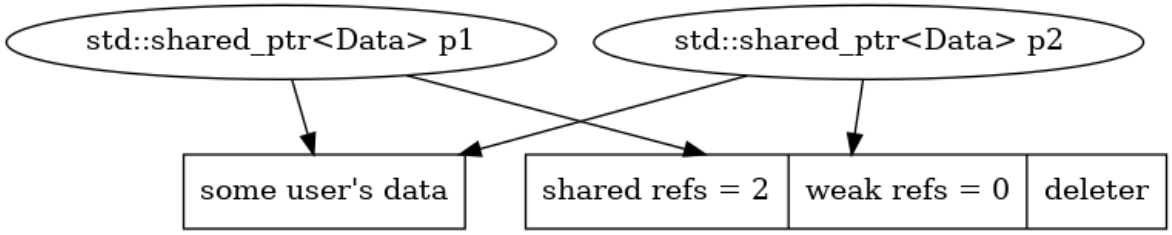


- Przenoszenie oznacza:
  - Kopiowanie wszystkich wskaźników do docelowego shared pointera
  - Ustawianie wskaźników w źródłowym shared pointerze na *nullptr*

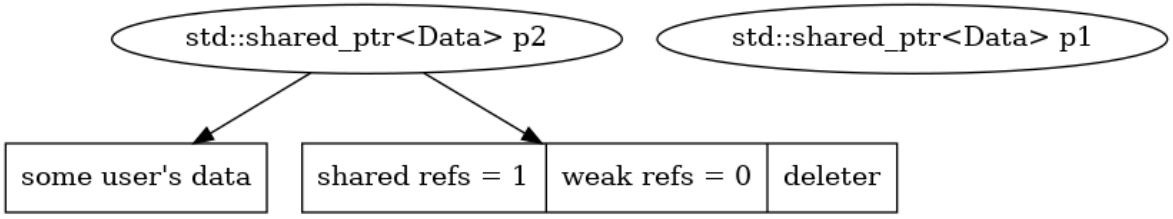


Shared pointer – kopiowanie i przenoszenie

- Kopiowanie oznacza:
  - Kopiowanie wskaźników
  - Inkrementację shared-refcount



- Przenoszenie oznacza:
  - Kopiowanie wszystkich wskaźników do docelowego shared pointera
  - Ustawianie wskaźników w źródłowym shared pointerze na *nullptr*



- Wszystkie metody są *inline*
- Kopiowanie wskaźników i inkrementacja liczników

Shared pointer – kopiowanie

```
1  void foo(std::shared_ptr<MyData> p);
2
3  void bar(std::shared_ptr<MyData> p)
4  {
5      foo(p);
6  }
```

Shared pointer – kopiowanie

```
1  void foo(std::shared_ptr<MyData> p);
2
3  void bar(std::shared_ptr<MyData> p)
4  {
5      foo(p);
6  }
```

- Wymaga inkrementacji/dekrementacji
- Atomics/locks nie są darmowe
- Zawoła destruktory
- Q: Czy można lepiej?

Shared pointer – przenoszenie

```
1  void foo(std::shared_ptr<MyData> p);
2
3  void bar(std::shared_ptr<MyData> p)
4  {
5      foo( std::move(p) );
6  }
```

Shared pointer – przenoszenie

```
1  void foo(std::shared_ptr<MyData> p);
2
3  void bar(std::shared_ptr<MyData> p)
4  {
5      foo( std::move(p) );
6  }
```

- Szybsze niż kopiowanie
- Ciągle wymaga kopiowania wskaźników
- Zawoła destruktory
- Q: Czy można lepiej?

Shared pointer – przekazywanie przez const-ref

```
1  void foo(std::shared_ptr<MyData> const& p);
2
3  void bar(std::shared_ptr<MyData> const& p)
4  {
5      foo(p);
6  }
```

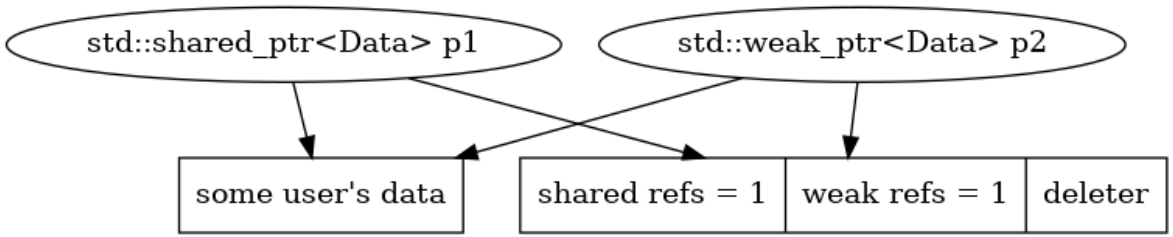
- The Fastest Way©
- Bez dodatkowych operacji
- Tak tanie jak przekazywanie wskaźników
- Może być zoptymalizowane przez kompilator
- Używaj domyślnie dla shared pointerów

Weak pointer

- Trzyma dwa wskaźniki:
  - Obiekt (dane użytkownika)
  - Licznik referencji

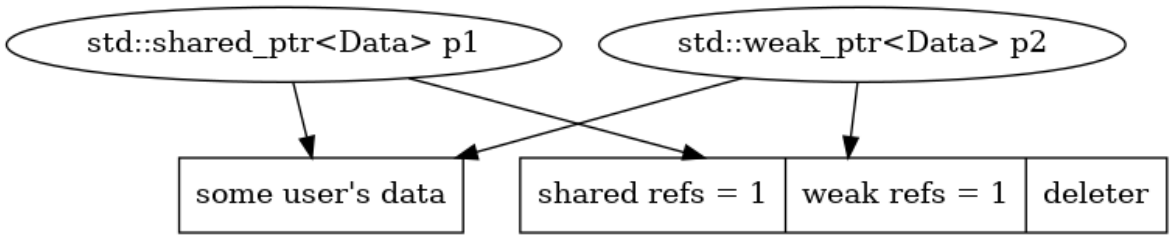
Weak pointer

- Trzyma dwa wskaźniki:
  - Obiekt (dane użytkownika)
  - Licznik referencji



Weak pointer

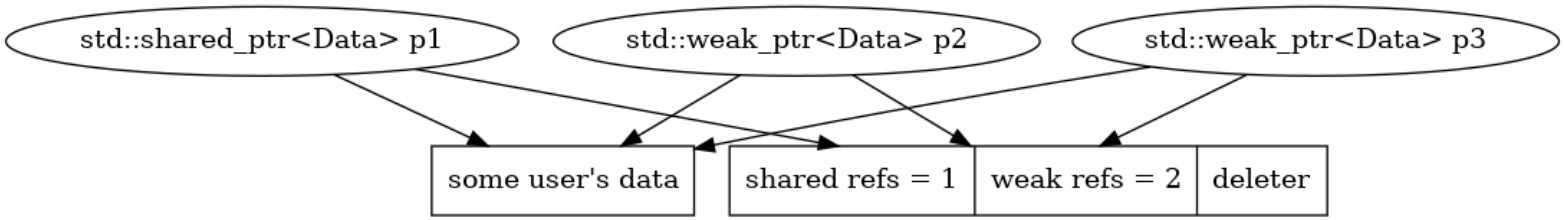
- Trzyma dwa wskaźniki:
  - Obiekt (dane użytkownika)
  - Licznik referencji



- Destruktor:
  - Dekrementuje *weak-refcount*
  - Usuwa liczniki, gdy *weak-refcount==0* i *shared-refcount==0*

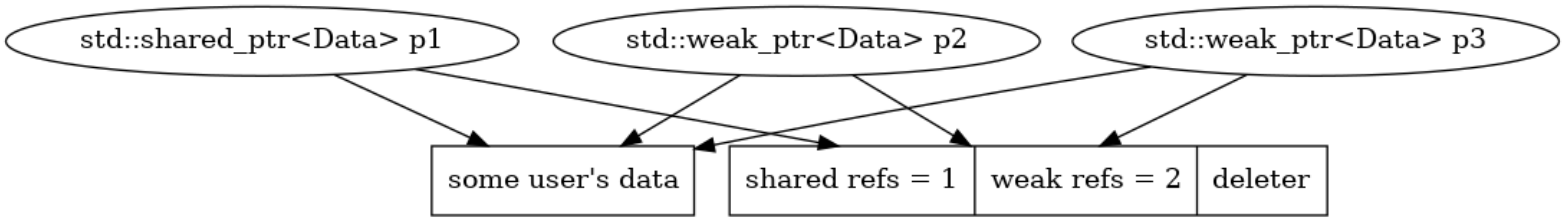
Weak pointer – kopiowanie i przenoszenie

- Kopiowanie oznacza:
  - Kopiowanie wskaźników
  - Inkrementację weak-refcount

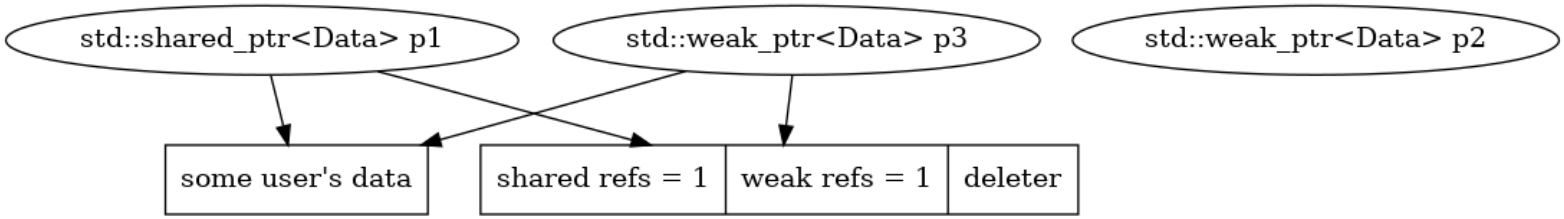


Weak pointer – kopiowanie i przenoszenie

- Kopiowanie oznacza:
  - Kopiowanie wskaźników
  - Inkrementację weak-refcount

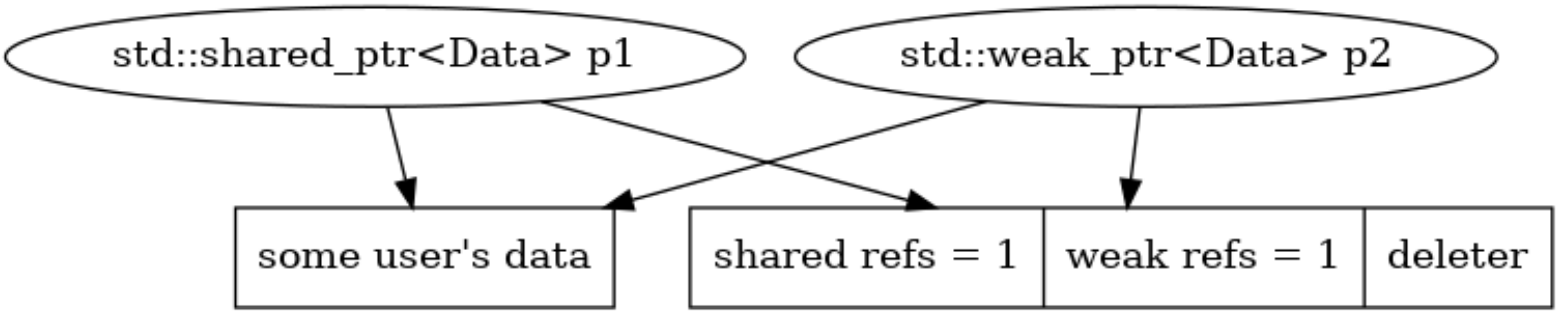


- Przenoszenie oznacza:
  - Kopiowanie wskaźników do docelowego weak pointera
  - Ustawianie wskaźników w źródłowym weak pointerze na *nullptr*



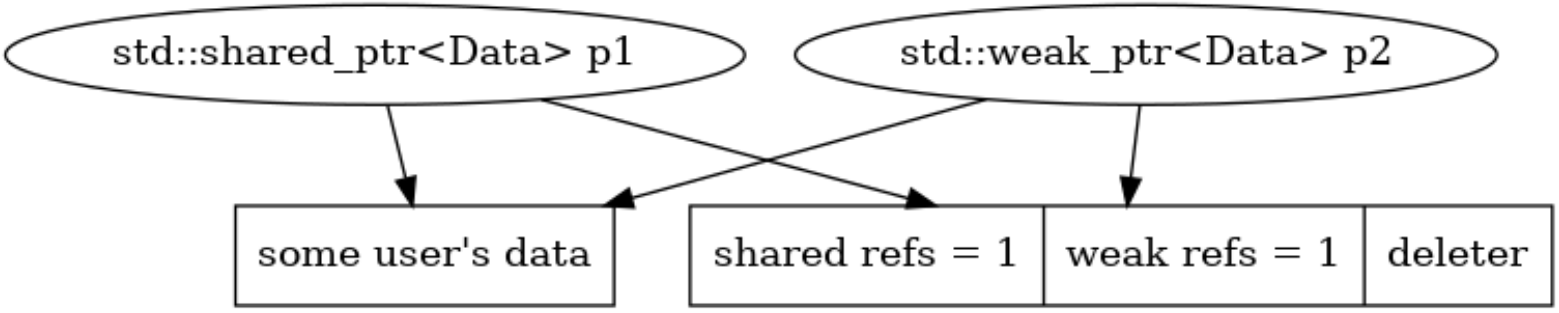
Weak i shared

- Gdy mamy shared pointer i weak pointer:

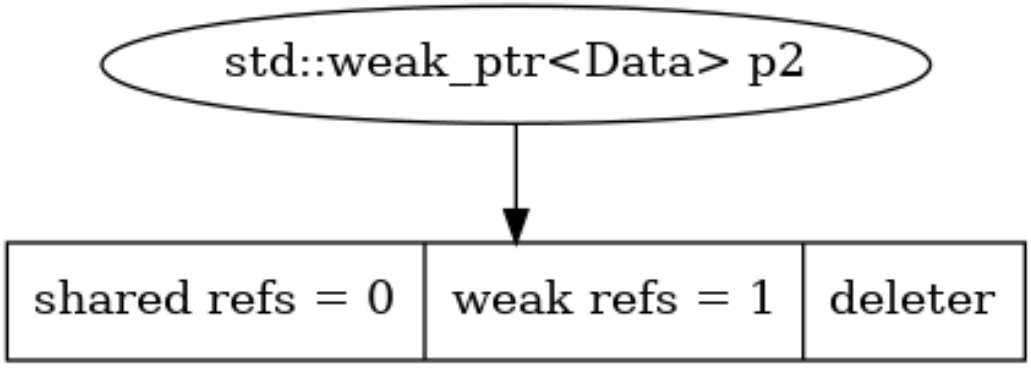


Weak i shared

- Gdy mamy shared pointer i weak pointer:

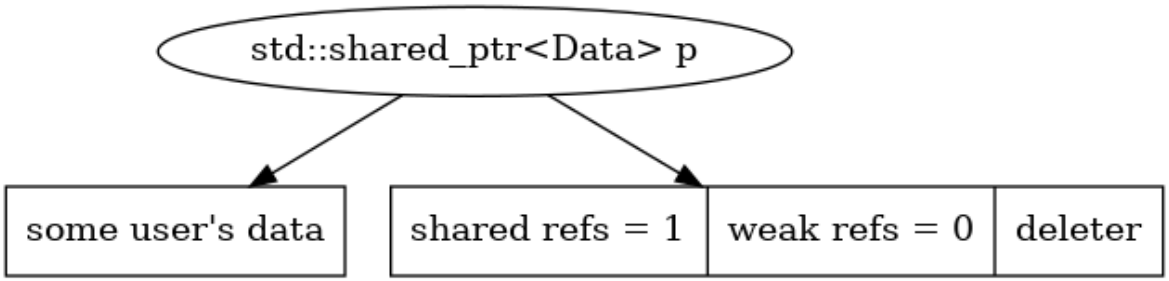


- Po usunięciu shared pointera:



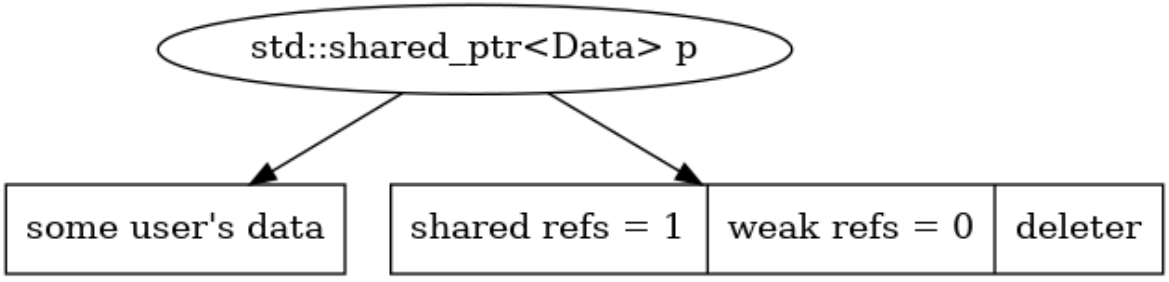
Tworzenie shared pointera

- *std::shared_ptr<Data> p{new Data}*:

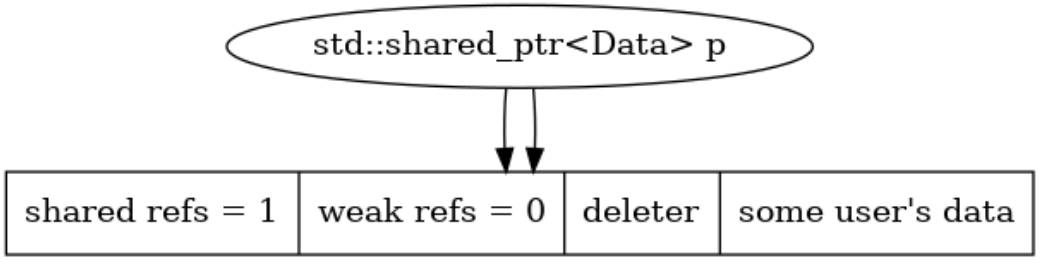


Tworzenie shared pointera

- *std::shared_ptr<Data> p{new Data}*:



- *auto p = std::make_shared<Data>()*:



- Zajmuje mniej pamięci (zazwyczaj)
- Tylko jedna alokacja
- Cache-friendly

Agenda

Zwykły wskaźnik

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6    char tab_[42];
7  };
8
9  int main(void)
10 {
11   constexpr unsigned size = 10u*1000u*1000u;
12   std::vector<Data*> v;
13   v.reserve(size);
14   for(unsigned i=0; i<size; ++i)
15   {
16     auto p = new Data;
17     v.push_back( std::move(p) );
18   }
19   for(auto p: v)
20     delete p;
21 }
```

std::unique_ptr

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6    char tab_[42];
7  };
8
9  int main(void)
10 {
11   constexpr unsigned size = 10u*1000u*1000u;
12   std::vector<std::unique_ptr<Data>> v;
13   v.reserve(size);
14   for(unsigned i=0; i<size; ++i)
15   {
16     std::unique_ptr<Data> p{new Data};
17     v.push_back( std::move(p) );
18   }
19 }
```

std::shared_ptr

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6    char tab_[42];
7  };
8
9  int main(void)
10 {
11   constexpr unsigned size = 10u*1000u*1000u;
12   std::vector<std::shared_ptr<Data>> v;
13   v.reserve(size);
14   for(unsigned i=0; i<size; ++i)
15   {
16     std::shared_ptr<Data> p{new Data};
17     v.push_back( std::move(p) );
18   }
19 }
```

std::weak_ptr

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6    char tab_[42];
7  };
8
9  int main(void)
10 {
11   constexpr unsigned size = 10u*1000u*1000u;
12   std::vector<std::shared_ptr<Data>> vs;
13   std::vector<std::weak_ptr<Data>>   vw;
14   vs.reserve(size);
15   vw.reserve(size);
16   for(unsigned i=0; i<size; ++i)
17   {
18     std::shared_ptr<Data> p{new Data};
19     std::weak_ptr<Data>   w{p};
20     vs.push_back( std::move(p) );
21     vw.push_back( std::move(w) );
22   }
23 }
```

std::shared_ptr – std::make_shared

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6    char tab_[42];
7  };
8
9  int main(void)
10 {
11   constexpr unsigned size = 10u*1000u*1000u;
12   std::vector<std::shared_ptr<Data>> v;
13   v.reserve(size);
14   for(unsigned i=0; i<size; ++i)
15   {
16     auto p = std::make_shared<Data>();
17     v.push_back( std::move(p) );
18   }
19 }
```

Pomiary

- GCC-4.8.2
- Kompilacja z *-std=c++11 -O3 -DNDEBUG*
- Pomiary wykonane przy pomocy:
  - *time* (real)
  - *htop* (mem)
  - *valgrind* (liczba alokacji)

Pomiary

- GCC-4.8.2
- Kompilacja z *-std=c++11 -O3 -DNDEBUG*
- Pomiary wykonane przy pomocy:
  - *time* (real)
  - *htop* (mem)
  - *valgrind* (liczba alokacji)

nazwa testu	**czas [s]**	**alokacje**	**pamięć [MB]**
zwykły wskaźnik	0.54	$10,000,001$	686
std::unique_ptr	0.56	$10,000,001$	686

Pomiary

- GCC-4.8.2
- Kompilacja z *-std=c++11 -O3 -DNDEBUG*
- Pomiary wykonane przy pomocy:
  - *time* (real)
  - *htop* (mem)
  - *valgrind* (liczba alokacji)

nazwa testu	**czas [s]**	**alokacje**	**pamięć [MB]**
zwykły wskaźnik	0.54	$10,000,001$	686
std::unique_ptr	0.56	$10,000,001$	686
std::shared_ptr	1.00	$20,000,001$	1072
std::weak_ptr	1.28	$20,000,002$	1222

Pomiary

- GCC-4.8.2
- Kompilacja z *-std=c++11 -O3 -DNDEBUG*
- Pomiary wykonane przy pomocy:
  - *time* (real)
  - *htop* (mem)
  - *valgrind* (liczba alokacji)

nazwa testu	**czas [s]**	**alokacje**	**pamięć [MB]**
zwykły wskaźnik	0.54	10,000,001	686
std::unique_ptr	0.56	10,000,001	686
std::shared_ptr	1.00	20,000,001	1072
std::weak_ptr	1.28	20,000,002	1222
std::make_shared	0.76	10,000,001	914

Agenda

Zapamiętaj

- Podejście RAII
  - Pozyskanie zasobu w konstruktorze
  - Zwolnienie zasobu w destruktorze
  - Pamiętaj zasadę 5

Zapamiętaj

- Podejście RAII
  - Pozyskanie zasobu w konstruktorze
  - Zwolnienie zasobu w destruktorze
  - Pamiętaj zasadę 5
- Zarządzanie pamięcią:
  1. *std::unique_ptr* – główny wybór
  2. *std::shared_ptr*
  3. *std::weak_ptr*

Zapamiętaj

- Podejście RAII
  - Pozyskanie zasobu w konstruktorze
  - Zwolnienie zasobu w destruktorze
  - Pamiętaj zasadę 5
- Zarządzanie pamięcią:
  1. *std::unique_ptr* – główny wybór
  2. *std::shared_ptr*
  3. *std::weak_ptr*
- Nie wynajduj koła na nowo! Zamiast tego:
  1. Używaj STL i pamiętaj o C++11/14 [18]
  2. Używaj Boosta [19]
  3. Używaj innych popularnych bibliotek
- Jeśli potrzebujesz tablicy, użyj *std::vector*.
- Pamiętaj, że prawie wszystko może rzucić wyjątek
- Ucz się – C++ ciągle się zmienia :)

Słowo na temat obsługi błędów

Error handling and exceptions

Exceptions make the complexity of error handling visible. However exceptions are not the cause of that complexity. Be careful not to blame the messenger for bad news.

- Bjarne Stroustrup
- "The C++ Programming Language", 4th edition [15]

Linki i książki

1. http://www.gotw.ca/gotw/020.htm
2. http://en.wikipedia.org/wiki/RAII
3. "Effective C++: 50 Specific Ways to Improve Your Programs and Designs", Scott Meyers
4. "More Effective C++: 35 New Ways to Improve Your Programs and Designs", Scott Meyers
5. "Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs", Scott Meyers
6. "Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library", Scott Meyers
7. "Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs", Scott Meyers
8. "Modern C++ Design: Generic Programming and Design Patterns Applied", Andrei Alexandrescu
9. "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices", Herb Sutter, Andrei Alexandrescu
10. "Exceptional C++", Herb Sutter
11. "More Exceptional C++", Herb Sutter
12. "Exceptional C++ Style", Herb Sutter
13. http://www.isocpp.org
14. "Inside the C++ Object Model"[1], Stanley B. Lippman
15. "The C++ Programming Language"[2], Bjarne Stroustrup
16. http://en.wikipedia.org/wiki/C++11
17. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf – "Technical report on C++ performance"
18. http://cppreference.com
19. http://www.boost.org

[1] Pre-standardization publication – some parts are outdated

[2] 4th edition rewritten for C++11

Pytania?