NOKIA

Zarządzanie pamięcią w C++

Ewelina Barań, Paweł Krysiak

NOKIA

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

Ile jest możliwych ścieżek wykonania tego kodu?

```
1  #include <iostream>
2
3  String EvaluateSalaryAndReturnName(Employee e)
4  {
5      if (e.Title() == "CEO" || e.Salary() > 100000)
6      {
7          std::cout << e.First() << " " << e.Last()
8                    << " is overpaid" << std::endl;
9      }
10     return e.First() + " " + e.Last();
11 }
```

Ile jest możliwych ścieżek wykonania tego kodu?

```
1  #include <iostream>
2
3  String EvaluateSalaryAndReturnName(Employee e)
4  {
5      if (e.Title() == "CEO" || e.Salary() > 100000)
6      {
7          std::cout << e.First() << " " << e.Last()
8                    << " is overpaid" << std::endl;
9      }
10     return e.First() + " " + e.Last();
11 }
```

Odpowiedź: 23 (3 "typowe", oraz aż 20 "ukrytych")

Źródło: GotW #20

Proste wskaźniki

```
1   struct MyData
2   {
3       int number = 0;
4   };
5
6   void process(MyData& p)
7   {
8       p.number = 77;
9   }
```

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

Proste wskaźniki

- Wycieki i błędy pamięci

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

Proste wskaźniki

- Wycieki i błędy pamięci
- Niebezpieczne ze względu na wyjątki

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

Proste wskaźniki

- Wycieki i błędy pamięci
- Niebezpieczne ze względu na wyjątki
- Trudne dla użytkownika

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

Proste wskaźniki

- Wycieki i błędy pamięci
- Niebezpieczne ze względu na wyjątki
- Trudne dla użytkownika
- Potrzebna dokumentacja

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

Zadanie 1

Skompiluj kod, wykonaj za pomocą valgrinda, następnie popraw błędy pamięci.

Github – zadanie 1

Zadanie 2

Skompiluj kod, wykonaj za pomocą valgrinda, następnie popraw błędy pamięci.

Github – zadanie 2

Zagadka - co się wyświetli na ekranie?

```
1   class IFigure
2   {
3   public:
4       ~IFigure() = default;
5       virtual void printMe() = 0;
6   };
7   class Circle: public IFigure
8   {
9   public:
10      Circle(int p_radius): radius(new int(p_radius)) {}
11      ~Circle()
12      {
13          printMe();
14          delete radius;
15      }
16      void printMe() override
17      {
18          std::cout << "I'm Circle: " << *radius << std::endl;
19      }
20      int* radius;
21  };
22  int main()
23  {
24      IFigure* circle = new Circle(5);
25      delete circle;
26  }
```

Wirtualny destruktor

```
1   class IFigure
2   {
3   public:
4       virtual ~IFigure() = default;
5       virtual void printMe() = 0;
6   };
7   class Circle: public IFigure
8   {
9   public:
10      Circle(int p_radius): radius(new int(p_radius)) {}
11      ~Circle()
12      {
13          printMe();
14          delete radius;
15      }
16      void printMe() override
17      {
18          std::cout << "I'm Circle: " << *radius << std::endl;
19      }
20      int* radius;
21  };
22  int main()
23  {
24      IFigure* circle = new Circle(5);
25      delete circle;
26  }
```

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

Wyjątki

Czym są?

```
1  try
2  {
3      risky_instructions
4  }
5  catch (...)
6  {
7      error_handling
8  }
```

Wyjątki

Czym są?

- W bloku `try` umieszczamy ryzykowne instrukcje.

```
1  try
2  {
3      risky_instructions
4  }
5  catch (...)
6  {
7      error_handling
8  }
```

Wyjątki

Czym są?

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje przerwanie bloku `try`.

```
1  try
2  {
3      risky_instructions
4  }
5  catch (...)
6  {
7      error_handling
8  }
```

Wyjątki

Czym są?

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje przerwanie bloku `try`.
- Rzucony wyjątek leci przez chwilę, aż zostanie złapany w odpowiednim bloku `catch`.

```
1  try
2  {
3      risky_instructions
4  }
5  catch (...)
6  {
7      error_handling
8  }
```

Wyjątki

Blok try-catch na przykładzie

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

Wyjątki

Zapamiętaj!

Wyjątki

Zapamiętaj!

- Umieszczając kilka bloków catch jeden po drugim, zadbaj o to, aby występowały one w porządku rosnącej ogólności.

Wyjątki

Zapamiętaj!

- Umieszczając kilka bloków catch jeden po drugim, zadbaj o to, aby występowały one w porządku rosnącej ogólności.
- Klauzula catch dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.

Wyjątki

Zapamiętaj!

- Umieszczając kilka bloków catch jeden po drugim, zadbaj o to, aby występowały one w porządku rosnącej ogólności.
- Klauzula catch dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.
- Wyjątek, który nie został złapany przez żaden blok try/catch powoduje wykonanie metody `std::terminate()`.

Wyjątki

Zapamiętaj!

- Umieszczając kilka bloków catch jeden po drugim, zadbaj o to, aby występowały one w porządku rosnącej ogólności.
- Klauzula catch dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.
- Wyjątek, który nie został złapany przez żaden blok try/catch powoduje wykonanie metody `std::terminate()`.
- Rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku try/catch.

Wyjątki

noexcept

Wyjątki

noexcept

- Od C++11 mamy możliwość zaznaczenia, że funkcja nie rzuca wyjątku: `noexcept`.

Wyjątki

noexcept

- Od C++11 mamy możliwość zaznaczenia, że funkcja nie rzuca wyjątku: `noexcept`.
- Kompilator sprawdzi czy funkcja może rzucić wyjątkiem.

Wyjątki

noexcept

- Od C++11 mamy możliwość zaznaczenia, że funkcja nie rzuca wyjątku: `noexcept`.
- Kompilator sprawdzi czy funkcja może rzucić wyjątkiem.
- Powoduje optymalizację kodu.

Wyjątki

noexcept

- Od C++11 mamy możliwość zaznaczenia, że funkcja nie rzuca wyjątku: `noexcept`.
- Kompilator sprawdzi czy funkcja może rzucić wyjątkiem.
- Powoduje optymalizację kodu.
- Wołanie `throw` z funkcji noexcept wywoła `std::terminate()`.

Wyjątki

Strefy bezwyjątkowe

Wyjątki

Strefy bezwyjątkowe

- Destruktor - procedura zwijania stosu uruchamia destruktory.

Wyjątki

Strefy bezwyjątkowe

- Destruktor - procedura zwijania stosu uruchamia destruktory.
- Konstruktory kopiujące i przenoszące klas, których obiekty rzucamy jako wyjątki.

Wyjątki

Zarządzanie pamięcią w obliczu wyjątków

```
1   struct MyData {};
2   void foo(){throw std::runtime_error("Error");}
3   int main()
4   {
5       try
6       {
7           MyData* data = new MyData();
8           try
9           {
10              foo();
11          }
12          catch (...)
13          {
14              throw;
15          }
16          delete data;
17      }
18      catch (std::runtime_error const&)
19      {
20          /*handle exception*/
21      }
22  }
```

Zadanie 3

Uruchom program za pomocą valgrinda.

Popraw błędy.

Github – zadanie 3

Zadanie 4

Uruchom program za pomocą valgrinda.

Gdzie jest problem??

Github – zadanie 4

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

Konstruktory

Rodzaje konstruktorów – przed C++11

```
1  Partner(); //default constructor
2  Partner(const Partner& other); //copy constructor
3  Partner& operator=(const Partner& other); //copy assignment
4  ~Partner(); //destructor
```

Konstruktory

Rodzaje konstruktorów – po C++11

```
1  Partner(); //default constructor
2  Partner(const Partner& other); //copy constructor
3  Partner& operator=(const Partner& other); //copy assignment
4  Partner(Partner&& other); //move constructor
5  Partner& operator=(Partner&& other); //move assignment
6  ~Partner(); //destructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{};
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1);
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1); //copy constructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2;
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1;
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1));
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1)); //move constructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1)); //move constructor
13      Partner p5 = std::move(p2);
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1)); //move constructor
13      Partner p5 = std::move(p2); //move constructor
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1); //copy constructor
10     Partner p3 = p2; //copy constructor
11     p3 = p1; //copy assign
12     Partner p4(std::move(p1)); //move constructor
13     Partner p5 = std::move(p2); //move constructor
14     p5 = std::move(p3);
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1); //copy constructor
10     Partner p3 = p2; //copy constructor
11     p3 = p1; //copy assign
12     Partner p4(std::move(p1)); //move constructor
13     Partner p5 = std::move(p2); //move constructor
14     p5 = std::move(p3); //move assign
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1); //copy constructor
10     Partner p3 = p2; //copy constructor
11     p3 = p1; //copy assign
12     Partner p4(std::move(p1)); //move constructor
13     Partner p5 = std::move(p2); //move constructor
14     p5 = std::move(p3); //move assign
15     Partner p6 = make_partner();
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1)); //move constructor
13      Partner p5 = std::move(p2); //move constructor
14      p5 = std::move(p3); //move assign
15      Partner p6 = make_partner(); //default constructor (x1)
```

Konstruktory

Rodzaje konstruktorów – przykład

```
 1  Partner make_partner()
 2  {
 3      return {};
 4  }
 5
 6  int main()
 7  {
 8      Partner p1{}; //default constructor
 9      Partner p2(p1); //copy constructor
10      Partner p3 = p2; //copy constructor
11      p3 = p1; //copy assign
12      Partner p4(std::move(p1)); //move constructor
13      Partner p5 = std::move(p2); //move constructor
14      p5 = std::move(p3); //move assign
15      Partner p6 = make_partner(); //default constructor (x1)
16      p6 = make_partner();
```

Konstruktory

Rodzaje konstruktorów – przykład

```
1  Partner make_partner()
2  {
3      return {};
4  }
5
6  int main()
7  {
8      Partner p1{}; //default constructor
9      Partner p2(p1); //copy constructor
10     Partner p3 = p2; //copy constructor
11     p3 = p1; //copy assign
12     Partner p4(std::move(p1)); //move constructor
13     Partner p5 = std::move(p2); //move constructor
14     p5 = std::move(p3); //move assign
15     Partner p6 = make_partner(); //default constructor (x1)
16     p6 = make_partner(); //default constructor + move assign
17 }
```

Konstruktory

Rule of Three

- Jeśli klasa potrzebuje własnej implementacji jednej z poniższych metod, powinna zapewnić je wszystkie:
  - Destruktor
  - Konstruktor kopiujący *(copy constructor)*
  - Kopiujący operator przypisania *(copy assignment)*
- Kompilator nie wie o specjalnych potrzebach i wygeneruje błędne zachowanie *(shallow copy vs. deep copy)*

Konstruktory

Rule of ~~Three~~ Five

Standard C++11 wprowadził operacje przenoszenia (`std::move()`), stąd potrzeba rozszerzenia wcześniejszej zasady o:

- Konstruktor przenoszący *(move constructor)*
- Przenoszący operator przypisania *(move assignment)*

Sam brak operacji przenoszenia najczęściej nie jest błędem, a straconą szansą na optymalizację.

Konstruktory

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Konstruktory

= delete; = default;

- Łatwe sterowanie zachowaniem klasy

Konstruktory

= delete; = default;

- Łatwe sterowanie zachowaniem klasy
- Bezpieczne, delegujemy czarną robotę z powrotem kompilatorowi

Konstruktory

= delete; = default;

- Łatwe sterowanie zachowaniem klasy
- Bezpieczne, delegujemy czarną robotę z powrotem kompilatorowi
- Dostępne od C++11

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

RAII

RAII

Resource Acquisition Is Initialization

RAII

Resource Acquisition Is Initialization

- RAII w pigułce

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)
  - Operacje na plikach, komunikacja z bazą danych, komunikacja sieciowa, ...

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)
  - Operacje na plikach, komunikacja z bazą danych, komunikacja sieciowa, ...
- Korzyści

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)
  - Operacje na plikach, komunikacja z bazą danych, komunikacja sieciowa, ...
- Korzyści
  - Gwarancja poprawności na poziomie języka

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)
  - Operacje na plikach, komunikacja z bazą danych, komunikacja sieciowa, ...
- Korzyści
  - Gwarancja poprawności na poziomie języka
  - Krótszy kod

RAII

Resource Acquisition Is Initialization

- RAII w pigułce
  - Każdy zasób ma swojego właściciela
  - Konstruktor == pozyskanie zasobu
  - Destruktor == zwolnienie zasobu
- Gdzie RAII jest pomocne?
  - Dynamicznie alokowana pamięć (`std::unique_ptr`, `std::shared_ptr`)
  - Wielowątkowość (`std::lock_guard`)
  - Operacje na plikach, komunikacja z bazą danych, komunikacja sieciowa, ...
- Korzyści
  - Gwarancja poprawności na poziomie języka
  - Krótszy kod
  - Jasna odpowiedzialność

RAII

Resource Acquisition Is Initialization - przykład

```
1  std::mutex m{};
2
3  void bad()
4  {
5      m.lock();
6      f();
7      if(!everything_ok()) return;
8      m.unlock();
9  }
10
11 void good()
12 {
13     std::lock_guard<std::mutex> lg{m}; //RAII is here!
14     f();
15     if(!everything_ok()) return;
16 }
```

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel
- Brak możliwości kopiowania, można przenosić

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel
- Brak możliwości kopiowania, można przenosić
- Własny deleter

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel
- Brak możliwości kopiowania, można przenosić
- Własny deleter
  - Możliwy, najczęściej wystarcza domyślny

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel
- Brak możliwości kopiowania, można przenosić
- Własny deleter
  - Możliwy, najczęściej wystarcza domyślny
  - Jest częścią typu

std::unique_ptr

```
1  template<class T,
2           class Deleter = std::default_delete<T>>
3  class unique_ptr;
```

- Naturalny następca surowych wskaźników
- Tylko jeden właściciel
- Brak możliwości kopiowania, można przenosić
- Własny deleter
  - Możliwy, najczęściej wystarcza domyślny
  - Jest częścią typu
  - Może zwiększyć rozmiar

std::unique_ptr

Unikalny wskaźnik – prosty przykład użycia

```
1   #include <iostream>
2   #include <memory>
3   #include <string>
4   #include <vector>
5
6   using ShoppingList = std::vector<std::string>;
7   struct Partner
8   {
9       std::unique_ptr<ShoppingList> shoppingList;
10  };
11  int main()
12  {
13      Partner boy{}, girl{};
```

std::unique_ptr

Unikalny wskaźnik – prosty przykład użycia

```
1   #include <iostream>
2   #include <memory>
3   #include <string>
4   #include <vector>
5
6   using ShoppingList = std::vector<std::string>;
7   struct Partner
8   {
9       std::unique_ptr<ShoppingList> shoppingList;
10  };
11  int main()
12  {
13      Partner boy{}, girl{};
14
15      boy.shoppingList =
16          std::unique_ptr<ShoppingList>(new ShoppingList{"Beer", "Nachos"});
```

std::unique_ptr

Unikalny wskaźnik – prosty przykład użycia

```
1   #include <iostream>
2   #include <memory>
3   #include <string>
4   #include <vector>
5
6   using ShoppingList = std::vector<std::string>;
7   struct Partner
8   {
9       std::unique_ptr<ShoppingList> shoppingList;
10  };
11  int main()
12  {
13      Partner boy{}, girl{};
14
15      boy.shoppingList =
16          std::unique_ptr<ShoppingList>(new ShoppingList{"Beer", "Nachos"});
17
18      ShoppingList importantItems = {"Pasta", "Toilet paper", "Hand sanitizer"};
19      girl.shoppingList =
20          std::make_unique<ShoppingList>(importantItems); // only from C++14!
```

std::unique_ptr

Unikalny wskaźnik – prosty przykład użycia

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>

using ShoppingList = std::vector<std::string>;
struct Partner
{
    std::unique_ptr<ShoppingList> shoppingList;
};
int main()
{
    Partner boy{}, girl{};

    boy.shoppingList =
        std::unique_ptr<ShoppingList>(new ShoppingList{"Beer", "Nachos"});

    ShoppingList importantItems = {"Pasta", "Toilet paper", "Hand sanitizer"};
    girl.shoppingList =
        std::make_unique<ShoppingList>(importantItems); // only from C++14!

    boy.shoppingList = std::move(girl.shoppingList);
}
```

Zadanie 5

Zamień użycie surowych wskaźników na `std::unique_ptr`

Github – zadanie 5

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>

using ShoppingList = std::vector<std::string>;
struct Partner
{
    std::shared_ptr<ShoppingList> shoppingList;
};
int main()
{
    ShoppingList list = {"Pasta", "Toilet paper", "Hand sanitizer"};
    Partner boy;
    Partner girl;

    boy.shoppingList = std::make_shared<ShoppingList>(list);
    // or std::shared_ptr<ShoppingList>(new ShoppingList{"Pasta", "Toilet paper"});
    girl.shoppingList = boy.shoppingList;

    std::cout << "Girl Ptr: " << girl.shoppingList << std::endl;
    std::cout << "Boy Ptr: " << boy.shoppingList;
}
```

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

Output

```
Girl Ptr: 0xbcbed0
Boy Ptr: 0xbcbed0
```

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

- Używaj `std::shared_ptr` w przypadku pamięci współdzielonej.

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

- Używaj `std::shared_ptr` w przypadku pamięci współdzielonej.
- `std::shared_ptr` działa podobnie jak zwykły wskaźnik.

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

- Używaj `std::shared_ptr` w przypadku pamięci współdzielonej.
- `std::shared_ptr` działa podobnie jak zwykły wskaźnik.
- Możemy kopiować wskaźniki `std::shared_ptr` bez kopiowania całej pamieci.

Shared Pointer

Współdzielony wskaźnik – prosty przykład użycia

- Używaj `std::shared_ptr` w przypadku pamięci współdzielonej.
- `std::shared_ptr` działa podobnie jak zwykły wskaźnik.
- Możemy kopiować wskaźniki `std::shared_ptr` bez kopiowania całej pamieci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Shared Pointer

Co należy wiedzieć?

Shared Pointer

Co należy wiedzieć?

- Wskaźnik ten nie jest właścicielem obiektu lecz przetrzymuje jedynie do niego wskaźnik.

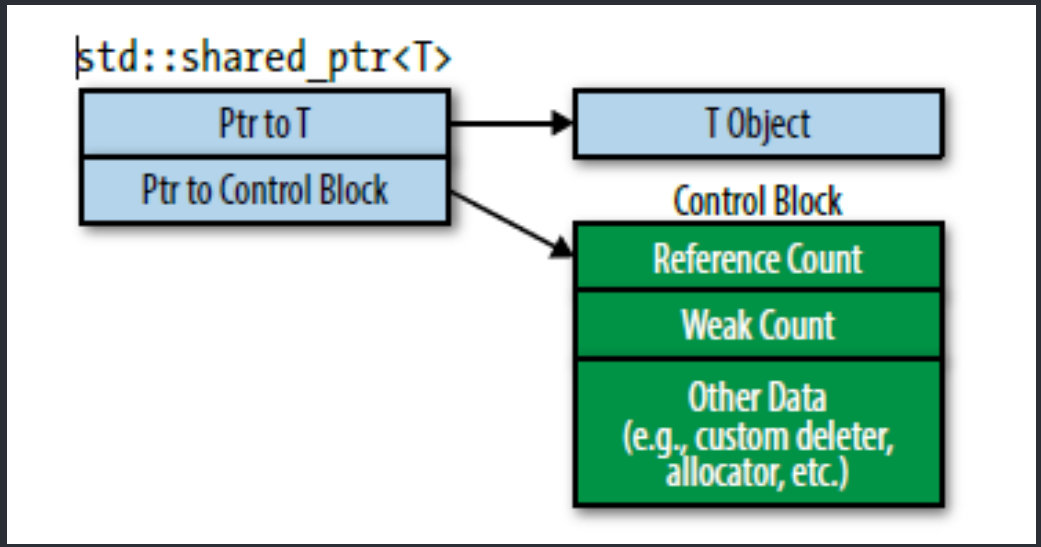
Shared Pointer

Co należy wiedzieć?

- Wskaźnik ten nie jest właścicielem obiektu lecz przetrzymuje jedynie do niego wskaźnik.
- Zwolnienie pamięci, nastąpi gdy ostatni `std::shared_ptr`, wskazujący na ten obiekt, zostanie zniszczony.

Shared Pointer

Skąd std::shared_ptr "wie", że jest ostatni?



Shared Pointer

Licznik odwołań

```
1   struct Partner {...};
2   int main()
3   {
4       ShoppingList list = {"Pasta", "Toilet paper", "Hand sanitizer"};
5       Partner boy;
6
7       boy.shoppingList = std::make_shared<ShoppingList>(list);
8       {
9           Partner girl;
10          girl.shoppingList = boy.shoppingList;
11          std::cout << "Use count: " << boy.shoppingList.use_count() << std::endl;
12      }
13      std::cout << "Use count: " << boy.shoppingList.use_count();
14      return 0;
15  }
```

Output

Licznik odwołań: 2
Licznik odwołań: 1

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny jest przydzielany dynamicznie

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny jest przydzielany dynamicznie
  - `std::shared_ptr` ma rozmiar dwóch wskaźników:
    - wskaźnik na obiekt
    - wskaźnik na blok kontrolny

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny jest przydzielany dynamicznie
  - `std::shared_ptr` ma rozmiar dwóch wskaźników:
    - wskaźnik na obiekt
    - wskaźnik na blok kontrolny
  - dokonujemy dwóch alokacji (dla danych i bloku kontrolnego!)

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny jest przydzielany dynamicznie
  - `std::shared_ptr` ma rozmiar dwóch wskaźników:
    - wskaźnik na obiekt
    - wskaźnik na blok kontrolny
  - dokonujemy dwóch alokacji (dla danych i bloku kontrolnego!)
- Jaka operacja nie zwiększy licznika odwołań?

Shared Pointer

Licznik odwołań wpływa na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny jest przydzielany dynamicznie
  - `std::shared_ptr` ma rozmiar dwóch wskaźników:
    - wskaźnik na obiekt
    - wskaźnik na blok kontrolny
  - dokonujemy dwóch alokacji (dla danych i bloku kontrolnego!)
- Jaka operacja nie zwiększy licznika odwołań?
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Shared Pointer

new vs. std::make_shared

Shared Pointer

new vs. std::make_shared

- Przy użyciu `new` dokonujemy dwóch alokacji pamięci - dla obiektu i bloku kontrolnego.

Shared Pointer

new vs. std::make_shared

- Przy użyciu `new` dokonujemy dwóch alokacji pamięci - dla obiektu i bloku kontrolnego.
- `std::make_shared` wykonuje jedną alokację w jednym bloku pamięci dla obiektu i bloku kontrolnego.

Shared Pointer

new vs. std::make_shared

- Przy użyciu `new` dokonujemy dwóch alokacji pamięci - dla obiektu i bloku kontrolnego.
- `std::make_shared` wykonuje jedną alokację w jednym bloku pamięci dla obiektu i bloku kontrolnego.
- Funkcja `std::make_shared`
  - jest wydajniejsza niż `new`,

Shared Pointer

new vs. std::make_shared

- Przy użyciu `new` dokonujemy dwóch alokacji pamięci - dla obiektu i bloku kontrolnego.
- `std::make_shared` wykonuje jedną alokację w jednym bloku pamięci dla obiektu i bloku kontrolnego.
- Funkcja `std::make_shared`
  - jest wydajniejsza niż `new`,
  - pozwala kompilatorowi na tworzenie mniejszego i szybszego kodu,

Shared Pointer

new vs. std::make_shared

- Przy użyciu `new` dokonujemy dwóch alokacji pamięci - dla obiektu i bloku kontrolnego.
- `std::make_shared` wykonuje jedną alokację w jednym bloku pamięci dla obiektu i bloku kontrolnego.
- Funkcja `std::make_shared`
  - jest wydajniejsza niż `new`,
  - pozwala kompilatorowi na tworzenie mniejszego i szybszego kodu,
  - jest bezpieczna pod względem wyjątków i współbieżności.

Shared Pointer

Przykład złego użycia 1

```
1   struct Partner {...};
2
3   int main()
4   {
5       Partner boy;
6       Partner girl;
7       auto listPtr = new ShoppingList({"Beer", "Wine"});
8
9       boy.shoppingList = std::shared_ptr<ShoppingList>(listPtr);
10      girl.shoppingList = std::shared_ptr<ShoppingList>(listPtr);
11
12      std::cout << "Use count: " << boy.shoppingList.use_count() << std::endl;
13      return 0;
14  }
```

Shared Pointer

Przykład złego użycia 2

```
1   struct Partner {...};
2
3   int main()
4   {
5       ShoppingList list = {"Butter", "Milk"};
6       Partner boy;
7       Partner girl;
8
9       boy.shoppingList = std::make_shared<ShoppingList>(std::move(list));
10
11      auto listPtr = boy.shoppingList.get();
12      girl.shoppingList = std::shared_ptr<ShoppingList>(listPtr);
13
14      std::cout << "Use count: " << boy.shoppingList.use_count() << std::endl;
15      return 0;
16  }
```

Shared Pointer

Zapamiętaj!

Shared Pointer

Zapamiętaj!

- Nie przekazuj surowego wskaźnika do `std::shared_ptr`

Shared Pointer

Zapamiętaj!

- Nie przekazuj surowego wskaźnika do `std::shared_ptr`
- Preferuj `std::make_shared`

Shared Pointer

Zapamiętaj!

- Nie przekazuj surowego wskaźnika do `std::shared_ptr`
- Preferuj `std::make_shared`
- Jeżeli potrzebujesz użyć `new`, zrób to bezpośrednio w konstruktorze `std::shared_ptr`

Shared Pointer

Niestandardowe deletery

```
1   auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe deletery,
2   auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typu
3
4   std::shared_ptr<Partner> girl(new Partner, deleterGirl);
5   std::shared_ptr<Partner> boy(new Partner, delererBoy);
6
7   std::vector<std::shared_ptr<Partner>> vectorOfPartners{girl, boy};
```

Shared Pointer

Niestandardowe deletery

```
1  auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe deletery,
2  auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typu
3
4  std::shared_ptr<Partner> girl(new Partner, deleterGirl);
5  std::shared_ptr<Partner> boy(new Partner, delererBoy);
6  //Deleter nie jest czescia typu shared_ptr
7  //bo zawiera sie w bloku kontrolnym
8
9  std::vector<std::shared_ptr<Partner>> vectorOfPartners{girl, boy};
```

Shared Pointer

Niestandardowe deletery

```
1   auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe deletery,
2   auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typu
3
4   std::shared_ptr<Partner> girl(new Partner, deleterGirl);
5   std::shared_ptr<Partner> boy(new Partner, delererBoy);
6   //Deleter nie jest czescia typu shared_ptr
7   //bo zawiera sie w bloku kontrolnym
8
9   std::vector<std::shared_ptr<Partner>> vectorOfPartners{girl, boy};
10  //Mozemy takie pointery trzymac w kolekcji - std::uniqie_ptr nie!
```

Zadanie 6

Stwórz ciało funkcji makeFile oraz addToFile

- Wykorzystaj std::File:

```
1  std::FILE* fopen( const char* filename, const char* mode )
2
3  int fclose( std::FILE* stream )
4
5  int fprintf( std::FILE* stream, const char* string)
```

Github – zadanie 6

Shared Pointer

Co należy zapamiętać?

Shared Pointer

Co należy zapamiętać?

- Wskaźniki `std::shared_ptr` zapewniają wygodę zarządzania współdzieloną pamięcią.

Shared Pointer

Co należy zapamiętać?

- Wskaźniki `std::shared_ptr` zapewniają wygodę zarządzania współdzieloną pamięcią.
- W porównaniu z `std::unique_ptr`, `std::shared_ptr` jest dwukrotnie większy. Nadmiarowość wynika z potrzeby posiadania bloku kontrolnego.

Shared Pointer

Co należy zapamiętać?

- Wskaźniki `std::shared_ptr` zapewniają wygodę zarządzania współdzieloną pamięcią.
- W porównaniu z `std::unique_ptr`, `std::shared_ptr` jest dwukrotnie większy. Nadmiarowość wynika z potrzeby posiadania bloku kontrolnego.
- Zwolnienie pamięci odbywa się za pomocą `delete`, ale można zastosować własne deletery, które nie wpływają na typ obiektu.

Shared Pointer

Co należy zapamiętać?

- Wskaźniki `std::shared_ptr` zapewniają wygodę zarządzania współdzieloną pamięcią.
- W porównaniu z `std::unique_ptr`, `std::shared_ptr` jest dwukrotnie większy. Nadmiarowość wynika z potrzeby posiadania bloku kontrolnego.
- Zwolnienie pamięci odbywa się za pomocą `delete`, ale można zastosować własne deletery, które nie wpływają na typ obiektu.
- Należy unikać tworzenia `std::shared_ptr` za pomocą wskaźników surowych.

Weak Pointer

Co to jest?

Weak Pointer

Co to jest?

- Działa jak `std::shared_ptr` ale nie uczestniczy w współdzielonym posiadaniu.

Weak Pointer

Co to jest?

- Działa jak `std::shared_ptr` ale nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika.

Weak Pointer

Co to jest?

- Działa jak `std::shared_ptr` ale nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika.
- Boryka się ze śledzeniem kiedy został wiszący.

Weak Pointer

Ciekawy interfejs weak_ptr na podstawie przykładu

```cpp
#include <iostream>
#include <memory>
std::weak_ptr<int> weakPtr;
void observe()
{
    std::cout << "Use count = " << weakPtr.use_count() << ": ";
    if (auto value = weakPtr.lock()) // Has to be copied into a shared_ptr
    {
        std::cout << *value << std::endl;
        return;
    }
    std::cout << "WeakPtr is expired" << std::endl;
}
int main()
{
    {
        auto sp = std::make_shared<int>(42);
        weakPtr = sp;
        observe();
    }
    observe();
}
```

Weak Pointer

Co to jest?

- Możemy sprawdzić czy `std::weak_ptr` wygasł za pomocą funkcji `expired()`.

Weak Pointer

Co to jest?

- Możemy sprawdzić czy `std::weak_ptr` wygasł za pomocą funkcji `expired()`.
- Należy wskaźnik `std::weak_ptr` obudować w `std::shared_ptr` i wtedy odwołać się do danych gdy nie jest `null`.

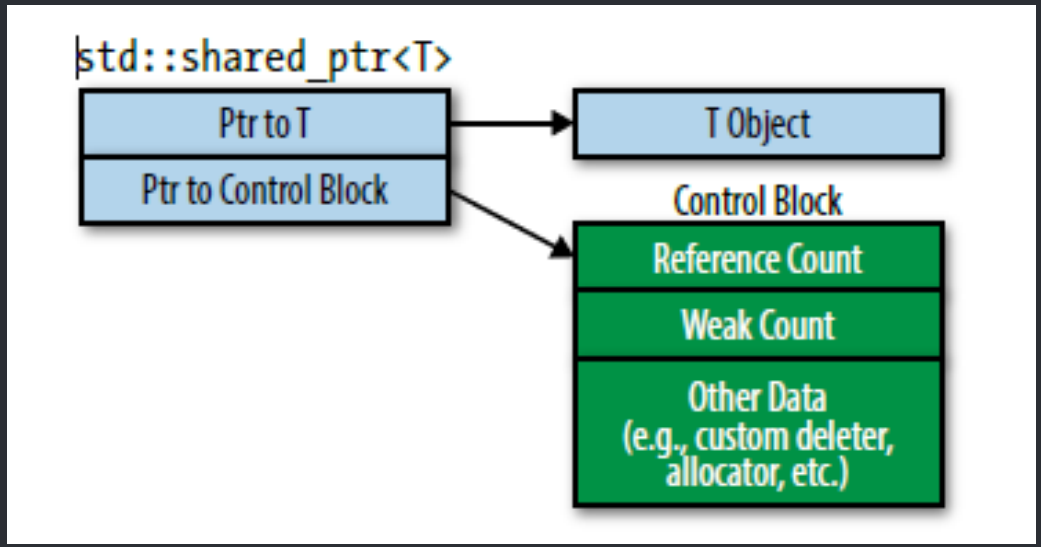
Weak Pointer

Co to jest?

- Możemy sprawdzić czy `std::weak_ptr` wygasł za pomocą funkcji `expired()`.
- Należy wskaźnik `std::weak_ptr` obudować w `std::shared_ptr` i wtedy odwołać się do danych gdy nie jest `null`.
- Służy do tego funkcja `lock()`.

Weak Pointer

Blok kontrolny zawiera weak_count



Weak Pointer

Blok kontrolny zawiera weak_count

- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników wiszących.

Weak Pointer

Blok kontrolny zawiera weak_count

- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników wiszących.
- Aby sprawdzić czy wskaźnik już jest wiszący potrzeba bloku kontrolnego.

Weak Pointer

Blok kontrolny zawiera weak_count

- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników wiszących.
- Aby sprawdzić czy wskaźnik już jest wiszący potrzeba bloku kontrolnego.
- Blok kontrolny zostanie zwolniony gdy ostatni `std::shared_ptr` oraz `std::weak_ptr` zostanie usunięty.

Weak Pointer

Monopoly przykład

```
1   struct Player
2   {
3       std::vector</*???*/Square> mySquares; //Wektor kupionych parceli na planszy;
4   };
5
6   struct Square
7   {
8       std::shared_ptr<Player> owner;  //Musi "wiedzie?" komu zaplaci? czynsz
9                                       //gdy inny gracz stanie na jego parceli
10  };
11
12  struct Game //Posiada wszystkich graczy i pola
13  {
14      std::vector<std::shared_ptr<Player>> players;
15      std::vector<std::shared_ptr<Square>> squares;
16  };
```

Weak Pointer

Blok kontrolny zawiera weak_count

Weak Pointer

Blok kontrolny zawiera weak_count

- Wskaźnik surowy
  - Gdy obiekt Square zostanie zniszczony, użytkownik może próbować wyłuskać ten obiekt.

Weak Pointer

Blok kontrolny zawiera weak_count

- Wskaźnik surowy
  - Gdy obiekt Square zostanie zniszczony, użytkownik może próbować wyłuskać ten obiekt.
- `std::shared_ptr`

Weak Pointer

Blok kontrolny zawiera weak_count

- Wskaźnik surowy
  - Gdy obiekt Square zostanie zniszczony, użytkownik może próbować wyłuskać ten obiekt.
- `std::shared_ptr`
  - Gdy Player i Square zawierają wskaźniki na siebie nawzajem, niszczenie okaże się niemożliwe.

Weak Pointer

Blok kontrolny zawiera weak_count

- Wskaźnik surowy
  - Gdy obiekt Square zostanie zniszczony, użytkownik może próbować wyłuskać ten obiekt.
- `std::shared_ptr`
  - Gdy Player i Square zawierają wskaźniki na siebie nawzajem, niszczenie okaże się niemożliwe.
- `std::weak_ptr`
  - Powyższe problemy zostaną rozwiane bo `std::weak_ptr` jest w stanie wydedukować czy już jest wiszący.

Weak Pointer

Podsumowanie

Weak Pointer

Podsumowanie

- Stosuj obiekty `std::weak_ptr` gdy możesz mieć do czynienia z obiektami `std::shared_ptr`, które mogą zawisnąć.

Weak Pointer

Podsumowanie

- Stosuj obiekty `std::weak_ptr` gdy możesz mieć do czynienia z obiektami `std::shared_ptr`, które mogą zawisnąć.
- Potencjalne przypadki wykorzystania `std::weak_ptr` to zapobieganie cyklom.

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

Unikaj alokacji za pomocą new

Problem 1

```
1  fooProcess(std::shared_ptr<MyData>(new MyData),
2             doSomethingElse());
```

Unikaj alokacji za pomocą new

Problem 1

```
1  fooProcess(std::shared_ptr<MyData>(new MyData),
2             doSomethingElse());
```

- Wykonanie instrukcji `new`.

Unikaj alokacji za pomocą new

Problem 1

```
1  fooProcess(std::shared_ptr<MyData>(new MyData),
2             doSomethingElse());
```

- Wykonanie instrukcji `new`.
- W zależności od kompilatora wykonanie konstruktora `std::shared_ptr` lub funkcji `doSomethingElse`.

Unikaj alokacji za pomocą new

Problem 1

```
1  fooProcess(std::shared_ptr<MyData>(new MyData),
2             doSomethingElse());
```

- Wykonanie instrukcji `new`.
- W zależności od kompilatora wykonanie konstruktora `std::shared_ptr` lub funkcji `doSomethingElse`.
- Funkcja `doSomethingElse`, może rzucić wyjątkiem - potencjalny wyciek pamięci.

Unikaj alokacji za pomocą new

Problem 1 - Gdy użyjemy std::make_shared

```
1  fooProcess(std::make_shared<MyData>(),
2             doSomethingElse());
```

Unikaj alokacji za pomocą new

Problem 1 - Gdy użyjemy std::make_shared

```
1  fooProcess(std::make_shared<MyData>(),
2             doSomethingElse());
```

- Nadal nie mamy gwarancji kolejności wyklonania argumentów.

Unikaj alokacji za pomocą new

Problem 1 - Gdy użyjemy std::make_shared

```
1  fooProcess(std::make_shared<MyData>(),
2             doSomethingElse());
```

- Nadal nie mamy gwarancji kolejności wyklonania argumentów.
- `std::make_shared` zapewni poprawną alokację pamięci bez możliwości jej utraty.

`const std::shared_ptr<T>&`

- Kopiowanie `std::shared_ptr` jest kosztowne (aktualizacja bloku kontrolnego robiona atomowo)
- Przekazywanie przez wartość oznacza faktyczne współdzielenie odpowiedzialności za zasób
- Jeśli chcemy tylko skorzystać z obiektu i możemy zagwarantować jego czas życia, używajmy `const&`

const std::shared_ptr<T>&

Przykład

```
 1  #include <memory>
 2
 3  void take_by_copy(std::shared_ptr<int> p)
 4  {
 5      static int x = 0;
 6      *p = ++x;
 7  }
 8
 9  int main()
10  {
11      auto p = std::make_shared<int>();
12      for (int i = 0; i < 10'000'000; ++i)
13      {
14          take_by_copy(p);
15      }
16  }
```

const std::shared_ptr<T>&

Przykład

```
 1  #include <memory>
 2
 3  void take_by_constref(const std::shared_ptr<int>& p)
 4  {
 5      static int x = 0;
 6      *p = ++x;
 7  }
 8
 9  int main()
10  {
11      auto p = std::make_shared<int>();
12      for (int i = 0; i < 10'000'000; ++i)
13      {
14          take_by_constref(p);
15      }
16  }
```

`const std::shared_ptr<T>&`

Rezultaty

nazwa testu	czas [s]
by value	0.47
by const &	0.11

Ponad czterokrotnie szybciej!

Agenda

1. Wprowadzenie
2. Wyjątki
3. Rule of Three/Five
4. Inteligentne wskaźniki
5. Dobre praktyki
6. Ciekawostki

std::make_shared

Jeden drobny szczegół...

Use case:

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty
- Mało dostępnej pamięci

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty
- Mało dostępnej pamięci
- Krótki czas życia obiektów

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty
- Mało dostępnej pamięci
- Krótki czas życia obiektów
- Obserwatorzy, którzy żyją długo

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty
- Mało dostępnej pamięci
- Krótki czas życia obiektów
- Obserwatorzy, którzy żyją długo

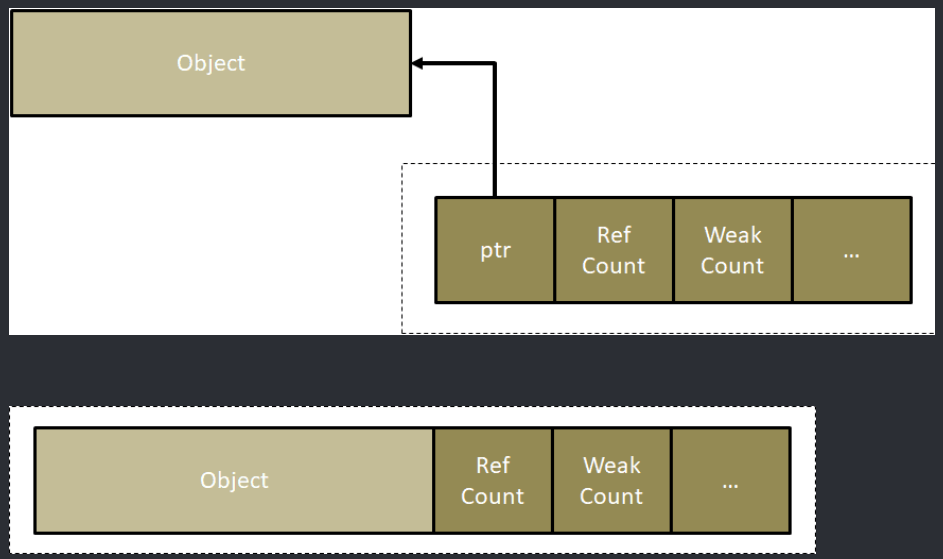
Tylko jedna alokacja, nie można zrobić częściowej dealokacji pamięci.
Trzeba czekać na wszystkich!
Destruktor już dawno zawołany, pamięć się marnuje...

std::make_shared

Jeden drobny szczegół...

Use case:

- Współdzielone, duże obiekty
- Mało dostępnej pamięci
- Krótki czas życia obiektów
- Obserwatorzy, którzy żyją długo



Tylko jedna alokacja, nie można zrobić częściowej dealokacji pamięci.

Trzeba czekać na wszystkich!

Destruktor już dawno zawołany, pamięć się marnuje...

Wydajność

Zwykły wskaźnik

```
1   #include <memory>
2   #include <vector>
3
4   struct Data
5   {
6       char tab_[42];
7   };
8
9   int main()
10  {
11      constexpr unsigned size = 10'000'000u;
12      std::vector<Data*> v;
13      v.reserve(size);
14      for (unsigned i=0u; 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  }
```

Wydajność

std::unique_ptr

```
1   #include <memory>
2   #include <vector>
3
4   struct Data
5   {
6       char tab_[42];
7   };
8
9   int main()
10  {
11      constexpr unsigned size = 10'000'000u;
12      std::vector<std::unique_ptr<Data>> v;
13      v.reserve(size);
14      for (unsigned i=0u; i<size; ++i)
15      {
16          std::unique_ptr<Data> p{new Data};
17          v.push_back(std::move(p));
18      }
19  }
```

Wydajność

std::shared_ptr

```
1   #include <memory>
2   #include <vector>
3
4   struct Data
5   {
6       char tab_[42];
7   };
8
9   int main()
10  {
11      constexpr unsigned size = 10'000'000u;
12      std::vector<std::shared_ptr<Data>> v;
13      v.reserve(size);
14      for (unsigned i=0u; i<size; ++i)
15      {
16          std::shared_ptr<Data> p{new Data};
17          v.push_back(std::move(p));
18      }
19  }
```

Wydajność

std::weak_ptr

```
 1  #include <memory>
 2  #include <vector>
 3
 4  struct Data
 5  {
 6      char tab_[42];
 7  };
 8
 9  int main()
10  {
11      constexpr unsigned size = 10'000'000u;
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=0u; 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  }
```

Wydajność

std::make_shared

```
1   #include <memory>
2   #include <vector>
3
4   struct Data
5   {
6       char tab_[42];
7   };
8
9   int main()
10  {
11      constexpr unsigned size = 10'000'000u;
12      std::vector<std::shared_ptr<Data>> v;
13      v.reserve(size);
14      for (unsigned i=0u; i<size; ++i)
15      {
16          auto p = std::make_shared<Data>();
17          v.push_back(std::move(p));
18      }
19  }
```

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
zwykły wskaźnik	0.59	10’000’001	610

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
zwykły wskaźnik	0.59	10’000’001	610
std:unique_ptr	0.58	10’000’001	610

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
zwykły wskaźnik	0.59	10’000’001	610
std:unique_ptr	0.58	10’000’001	610
std::shared_ptr	1.00	20’000’001	1043

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
zwykły wskaźnik	0.59	10’000’001	610
std:unique_ptr	0.58	10’000’001	610
std::shared_ptr	1.00	20’000’001	1043
std::weak_ptr	1.21	20’000’002	1192

Wydajność

- GCC 9.2.1
- Pomiary wykonane przy pomocy:
  - *time* (real) – czas
  - *valgrind* (memcheck) – alokacje
  - *valgrind* (massif) – zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
zwykły wskaźnik	0.59	10'000'001	610
std:unique_ptr	0.58	10'000'001	610
std::shared_ptr	1.00	20'000'001	1043
std::weak_ptr	1.21	20'000'002	1192
std::make_shared	0.70	10'000'001	839

Pytania

Post Scriptum

- ”Just use the stack”

Post Scriptum

- ”Just use the stack”
- Stos jest szybszy, bezpieczniejszy i po prostu działa.

Post Scriptum

- ”Just use the stack”
- Stos jest szybszy, bezpieczniejszy i po prostu działa.
- Znane również jako ewolucja Rule of Three/Five – Rule of Zero

Zaliczenie wykładu

- W ramach zaliczenia obecności należy wysłać zadanie nr X prowadzącym na maila
  - ewelina.baran@nokia.com
  - pawel.krysiak@nokia.com

Zaliczenie wykładu

- W ramach zaliczenia obecności należy wysłać zadanie nr X prowadzącym na maila
  - ewelina.baran@nokia.com
  - pawel.krysiak@nokia.com
- Losujemy!