Zarządzanie pamięcią w C++

Bogumił Chojnowski, Paweł Krysiak

Nokia

Wrocław, 2022-03-22

Agenda

Agenda

Wprowadzenie

Zarządzanie pamięcią

Zestaw technik kontroli nad zasobami wykorzystywanymi przez program.

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 odwołania do 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 odwołania do 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 odwołania do 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  }
```

Zadanie 1

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

https://github.com/pawelkrysiak/memory_management/blob/master/example1.cpp

Zadanie 2

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

https://github.com/pawelkrysiak/memory_management/blob/master/example2.cpp

Słowo o destrukcji

```
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  }
```

Wirtualny destruktor

Wnioski

- Jeśli klasa ma metody wirtualne - należy zdefiniować wirtualny destruktor
- Niedopełnienie tego obowiązku skutkuje niezdefiniowanym zachowaniem

Wirtualny destruktor

Wnioski

- Jeśli klasa ma metody wirtualne - należy zdefiniować wirtualny destruktor
- Niedopełnienie tego obowiązku skutkuje niezdefiniowanym zachowaniem

Wirtualny destruktor

Wnioski

- Jeśli klasa ma metody wirtualne - należy zdefiniować wirtualny destruktor
- Niedopełnienie tego obowiązku skutkuje niezdefiniowanym zachowaniem

Agenda

Gdy sytuacja wymyka się spod kontroli.

Zgłaszanie błędów

Istnieją trzy sposoby na zgłoszenie nieprawidłowości w trakcie działania programu:

- globalna zmienna z kodem błędu którą należy odczytać
- specjalna wartość zwracana z funkcji
- rzucenie wyjątku

Składnia

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje zakończenie wykonywania 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  }
```

Składnia

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje zakończenie wykonywania 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  }
```

Składnia

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje zakończenie wykonywania 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  }
```

Składnia

- W bloku `try` umieszczamy ryzykowne instrukcje.
- Rzucenie wyjątku powoduje zakończenie wykonywania 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  }
```

Blok try-catch na przykładzie

```
 1  #include <stdexcept>
 2  void foo()
 3  {
 4      throw std::runtime_error("Error");
 5  }
 6  int main()
 7  {
 8      try
 9      {
10          foo();
11      }
12      catch(std::runtime_error const&)
13      {
14          std::cout << "std::runtime_error" << std::endl;
15      }
16  }
```

Zapamiętaj!

- Rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pasującego bloku try/catch.
- Wyjątek, który nie został złapany przez żaden blok try/catch powoduje wykonanie metody `std::terminate()`.

Zapamiętaj!

- Rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pasującego bloku try/catch.
- Wyjątek, który nie został złapany przez żaden blok try/catch powoduje wykonanie metody `std::terminate()`.

Strefy bezwyjątkowe

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

Strefy bezwyjątkowe

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

Strefy bezwyjątkowe

- Destruktor - procedura zwijania stosu uruchamia destruktory.
- Konstruktory kopiujące i przenoszące klas, których obiekty rzucamy jako 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          foo();
 9          delete data;
10      }
11      catch (std::runtime_error const&)
12      {
13          /*handle exception*/
14      }
15  }
```

Zadanie 3

Uruchom program za pomocą valgrinda.

Popraw błędy.

https://github.com/pawelkrysiak/memory_management/blob/master/example3.cpp

Zadanie 4

Uruchom program za pomocą valgrinda.

Gdzie jest problem??

https://github.com/pawelkrysiak/memory_management/blob/master/example4.cpp

Agenda

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{}; //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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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
- Bezpieczne, delegujemy czarną robotę z powrotem kompilatorowi
- Dostępne od C++11

Konstruktory

= delete; = default;

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

Konstruktory

= delete; = default;

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

Agenda

RAII

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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;
```

- 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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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!
21
22      boy.shoppingList = std::move(girl.shoppingList);
23  }
```

Zadanie 5

Zamień użycie surowych wskaźników na `std::unique_ptr`

https://github.com/pawelkrysiak/memory_management/blob/master/example5.cpp

Agenda

Kiedy jest więcej niż jeden właściciel

Unikalny znaczy - unikalny!

`std::unique_ptr` zabrania udzielenia zasobu na równych prawach.

Jak działa std::shared_ptr?

Licznik referencji - `use_count`

Policzmy, ile jest referencji do zasobu, i zwolnijmy go, kiedy licznik spadnie do zera.

Współdzielony wskaźnik – prosty przykład – kod

```
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::shared_ptr<ShoppingList> shoppingList;
10  };
11  int main()
12  {
13      ShoppingList list = {"Pasta", "Toilet␣paper", "Hand␣sanitizer"};
14      Partner boy;
15      Partner girl;
16
17      boy.shoppingList = std::make_shared<ShoppingList>(list);
18      // or std::shared_ptr<ShoppingList>(new ShoppingList{"Pasta", "Toilet paper"});
19      girl.shoppingList = boy.shoppingList;
20
21      std::cout << "Girl␣Ptr:␣" << girl.shoppingList << std::endl;
22      std::cout << "Boy␣Ptr:␣" << boy.shoppingList;
23  }
```

Shared Pointer

Współdzielony wskaźnik – prosty przykład – adresy

```cpp
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;
}
```

x86-64 clang 11.0.1 Executor (Editor #1) C++ ×

A ▾ | Wrap lines | Libraries | Compilation | Arguments | Stdin | Compiler output

x86-64 clang 11.0.1 ▾ | Compiler options...

```
Program returned: 0
Program stdout
Girl Ptr: 0x2081f30
Boy Ptr: 0x2081f30
```

Shared Pointer

Współdzielony wskaźnik – prosty przykład – podsumowanie

- 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 pamięci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Shared Pointer

Współdzielony wskaźnik – prosty przykład – podsumowanie

- 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 pamięci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Shared Pointer

Współdzielony wskaźnik – prosty przykład – podsumowanie

- 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 pamięci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Shared Pointer

Współdzielony wskaźnik – prosty przykład – podsumowanie

- 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 pamięci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Shared Pointer

Współdzielony wskaźnik – prosty przykład – podsumowanie

- 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 pamięci.
- `std::make_shared` zaalokuje odpowiednią pamięć dla obiektu.

Zapamiętaj!

- Wskaźnik ten jest jednym ze współwłaścicieli obiektu.
- Zwolnienie pamięci nastąpi, gdy ostatni `std::shared_ptr` zostanie zniszczony.

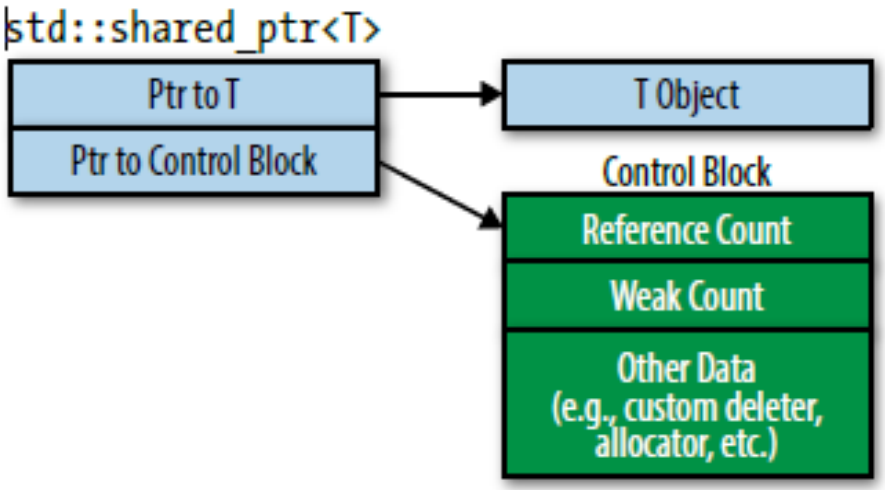
Zapamiętaj!

- Wskaźnik ten jest jednym ze współwłaścicieli obiektu.
- Zwolnienie pamięci nastąpi, gdy ostatni `std::shared_ptr` zostanie zniszczony.

Zapamiętaj!

- Wskaźnik ten jest jednym ze współwłaścicieli obiektu.
- Zwolnienie pamięci nastąpi, gdy ostatni `std::shared_ptr` zostanie zniszczony.

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



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

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

Licznik odwołań – wpływ na wydajność

- Czy blok kontrolny powinien być alokowany dynamicznie?
  - Blok kontrolny musi być 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?
  - Operacja wyłuskania (`operator *`)
  - Konstrukcja przenosząca - przenoszenie `std::shared_ptr` jest szybsze niż kopiowanie!

operator 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.

operator 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.

operator 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.

operator 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.

operator 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.

operator 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.

std::shared_ptr – przykład złego użycia – kod

```
 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:
13      return 0;
14  }
```

std::shared_ptr – przykład złego użycia

Wnioski

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

std::shared_ptr – niestandardowy deleter – 1/3

```
1  auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe
2  auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typ
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};
```

std::shared_ptr – niestandardowy deleter – 2/3

```
1  auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe
2  auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typ
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};
```

std::shared_ptr – niestandardowy deleter – 3/3

```
1  auto deleterGirl = [](Partner* p_partner){/*...*/}; // niestandardowe
2  auto delererBoy = [](Partner* p_partner){/*...*/};  //kazde innego typ
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)
```

https://github.com/pawelkrysiak/memory_management/blob/master/example6.cpp

std::weak_ptr – cichy obserwator

Co to jest?

- Nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika referencji.
- Potrafi stwierdzić fakt fizycznego zwolnienia zasobu.

std::weak_ptr – cichy obserwator

Co to jest?

- Nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika referencji.
- Potrafi stwierdzić fakt fizycznego zwolnienia zasobu.

std::weak_ptr – cichy obserwator

Co to jest?

- Nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika referencji.
- Potrafi stwierdzić fakt fizycznego zwolnienia zasobu.

std::weak_ptr – cichy obserwator

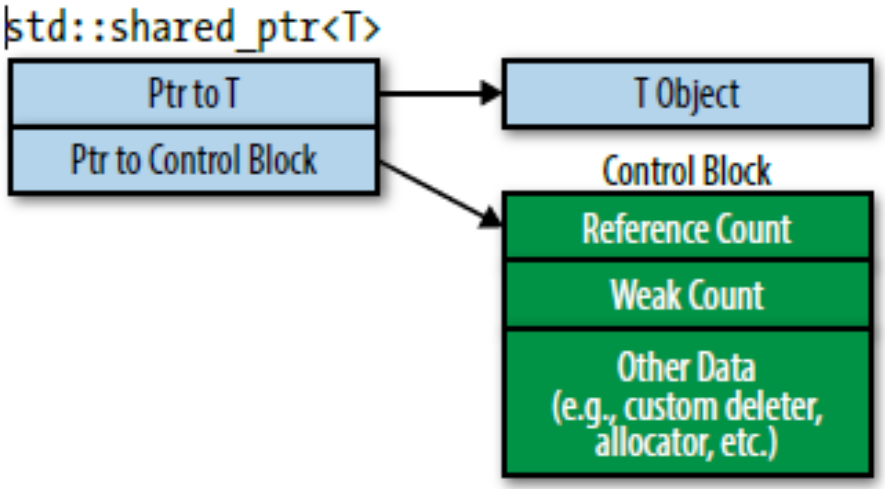
Co to jest?

- Nie uczestniczy w współdzielonym posiadaniu.
- Nie wpływa na powiększenie licznika referencji.
- Potrafi stwierdzić fakt fizycznego zwolnienia zasobu.

std::weak_ptr – przykład obserwowania zasobu

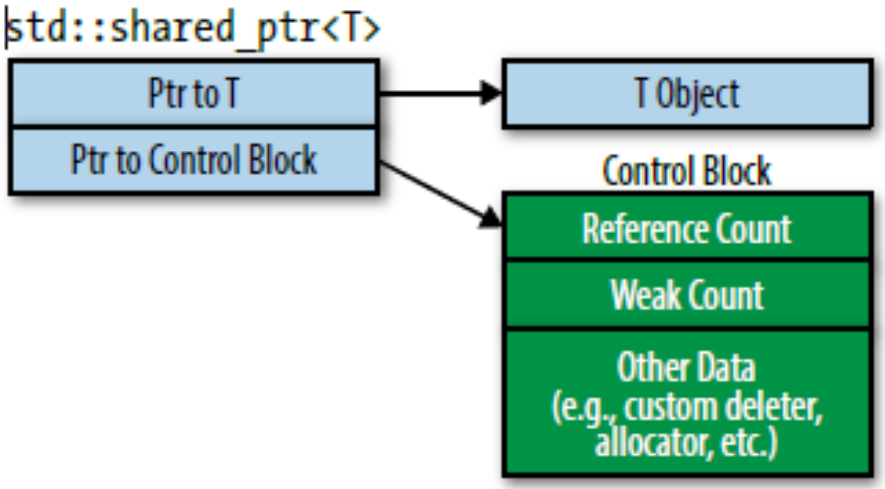
```
1   #include <iostream>
2   #include <memory>
3   std::weak_ptr<int> weakPtr;
4   void observe()
5   {
6       std::cout << "Use␣count␣=␣" << weakPtr.use_count() << ":␣";
7       if (auto value = weakPtr.lock()) // Has to be copied into a shared_ptr
8       {
9           std::cout << *value << std::endl;
10          return;
11      }
12      std::cout << "WeakPtr␣is␣expired" << std::endl;
13  }
14  int main()
15  {
16      {
17          auto sp = std::make_shared<int>(42);
18          weakPtr = sp;
19          observe();
20      }
21      observe();
22  }
```

std::weak_ptr – blok kontrolny zawiera weak_count



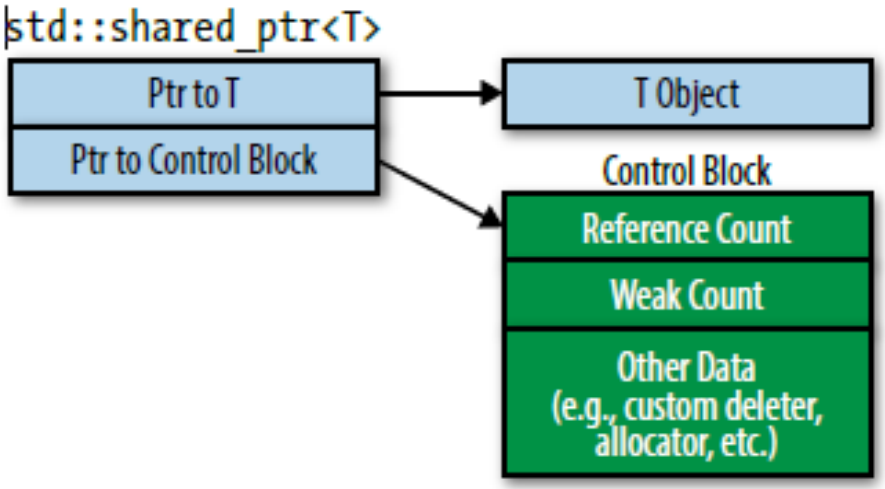
- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników obserwujących.
- Blok kontrolny zostanie zwolniony gdy ostatni `std::shared_ptr` i `std::weak_ptr` zostaną usunięte.
- Niezerowy weak_count nie przeszkadza zwolnić zasobu.

std::weak_ptr – blok kontrolny zawiera weak_count



- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników obserwujących.
- Blok kontrolny zostanie zwolniony gdy ostatni `std::shared_ptr` i `std::weak_ptr` zostaną usunięte.
- Niezerowy weak_count nie przeszkadza zwolnić zasobu.

std::weak_ptr – blok kontrolny zawiera weak_count



- Blok kontrolny z `std::shared_ptr` zawiera również licznik wskaźników obserwujących.
- Blok kontrolny zostanie zwolniony gdy ostatni `std::shared_ptr` i `std::weak_ptr` zostaną usunięte.
- Niezerowy weak_count nie przeszkadza zwolnić zasobu.

std::weak_ptr – zastosowanie

- Kiedy zasób może zostać zwolniony, nie możesz temu zapobiec, jednocześnie chcesz być swiadom że został usunięty
- Potencjalne przypadki wykorzystania `std::weak_ptr` to zapobieganie cyklom.

std::weak_ptr – zastosowanie

- Kiedy zasób może zostać zwolniony, nie możesz temu zapobiec, jednocześnie chcesz być swiadom że został usunięty
- Potencjalne przypadki wykorzystania `std::weak_ptr` to zapobieganie cyklom.

std::optional - kiedy zasobu może nie być

C++17 Alert

std::optional został wprowadzony do biblioteki standardowej w C++17

std::optional - kiedy zasobu nigdy nie było

Null Object

W tej roli wykorzystywany był NULL pointer.

std::optional – przykład użycia

```
1  void process_resource(std::optional<Resource> opt_resource) {
2      if (opt_resource.has_value())
3          std::cout << "I␣can␣process␣resource:␣"
4                    << opt_resource.value()
5                    << std::endl;
6  }
```

std::optional – przykład przenoszenia

```
 1  #include <optional>
 2  #include "Resource.h"
 3
 4  void process_resource(std::optional<Resource> opt_resource) {
 5      if (opt_resource.has_value())
 6          std::cout << "I␣can␣process␣resource:␣"
 7                    << opt_resource.value()
 8                    << std::endl;
 9  }
10
11  int main(int ac, char *av[]) {
12      std::optional<Resource> opt_resource = std::nullopt;
13      process_resource(std::move(opt_resource));
14
15      auto moved_resource = std::make_optional<Resource>(6);
16      process_resource(std::move(moved_resource));
17  }
```

std::optional vs wskaźniki

- std::optional realizuje koncept pustej wartości - robi to zwykle lepiej niż NULL
- wystrzegaj się tworzenia opcjonalnych wskaźników - każdy wskaźnik już modeluje Null Object
- standard C++17 rozmyślnie odrzucił ideę tworzenia opcjonalnych referencji znanych z boost::optional - referencja która może być NULL ma już nazwę - jest to **wskaźnik**.

std::optional vs wskaźniki

- std::optional realizuje koncept pustej wartości - robi to zwykle lepiej niż NULL
- wystrzegaj się tworzenia opcjonalnych wskaźników - każdy wskaźnik już modeluje Null Object
- standard C++17 rozmyślnie odrzucił ideę tworzenia opcjonalnych referencji znanych z boost::optional - referencja która może być NULL ma już nazwę - jest to **wskaźnik**.

std::optional vs wskaźniki

- std::optional realizuje koncept pustej wartości - robi to zwykle lepiej niż NULL
- wystrzegaj się tworzenia opcjonalnych wskaźników - każdy wskaźnik już modeluje Null Object
- standard C++17 rozmyślnie odrzucił ideę tworzenia opcjonalnych referencji znanych z boost::optional - referencja która może być NULL ma już nazwę - jest to **wskaźnik**.

Agenda

std::make_{unique,shared} versus new

```
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.

std::make_{unique,shared} versus new

```
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.

std::make_{unique,shared} versus new

```
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.

std::make_{unique,shared} versus new

```
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.

std::make_{unique,shared} versus new

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

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

std::make_{unique,shared} versus new

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

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

std::make_{unique,shared} versus new

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

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

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

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

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

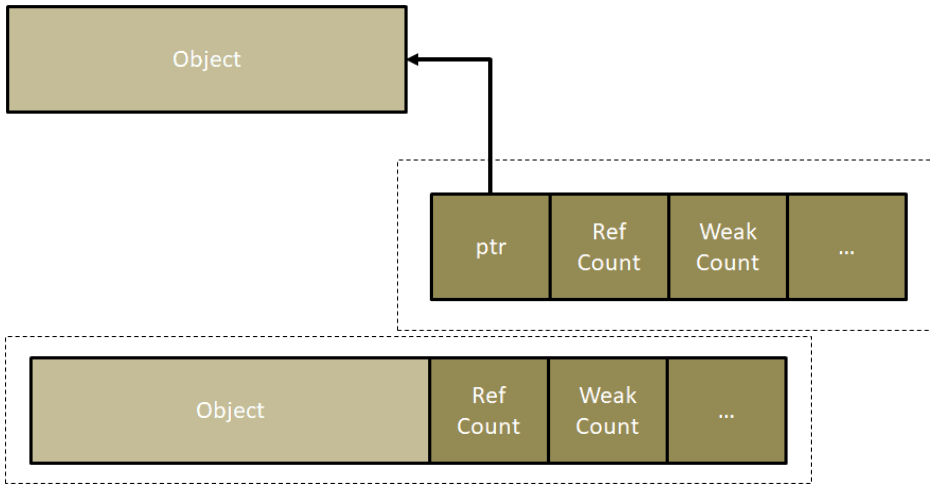
Tylko jedna alokacja, nie można zrobić częściowej dealokacji pamięci.
Trzeba czekać na wszystkich!
Destruktor już dawno zawołany, pamięć się marnuje...

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]**
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”
  - Stos jest szybszy, bezpieczniejszy i po prostu działa.
  - Znane również jako ewolucja Rule of Three/Five – Rule of Zero

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

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 należy wysłać rozwiązane zadania oraz sprawozdanie (<1000 słów) z zajęć mailem[1] do obu prowadzących:
  - bogumil.chojnowski@nokia.com
  - pawel.krysiak@nokia.com

[1] Dodatkowe punkty będą przyznane za umieszczenie w tytule e-maila: [PARO2022] Memory Management C++