Pytanie 1:

Co pojawi się na ekranie po wykonaniu poniższego kodu?

```
1  #include <iostream>
2  using namespace std;
3
4  struct Base
5  {
6      Base() { cout << "Base "; }
7      ~Base() { cout << "~Base "; }
8  };
9
10 struct Derived : public Base
11 {
12     Derived() { cout << "Derived "; }
13     ~Derived() { cout << "~Derived "; }
14 };
15
16 int main() {
17     Derived d;
18     return 0;
19 }
```

A. Base Derived ~Derived ~Base
B. Derived Base ~Base ~Derived
C. Base Derived ~Base Derived
D. Derived Base ~Derived ~Base

Typowo rekrutacyjne pytanie. Znajomość kolejności wywoływania konstruktorów i destruktorów jest kluczową wiedzą.
Napierw zawsze wołany jest konstuktor klasy bazowej. Niszczenie obiektu przebiega w odwrotnej kolejności, czyli destuktor klasy bazowej zawoła się na końcu.

Pytanie 2:

Poniższy kod:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {5, 4, 8, 2, 9, 10, 3};
    std::sort(numbers.begin(), numbers.end());
    for(auto number : numbers)
    {
        std::cout << number << " ";
    }
    return 0;
}
```

A. Kompiluje się i działa poprawnie. Na wyjściu zobaczymy 2 3 4 5 8 9 10

B. Kompiluje się, ale nie działa poprawnie, bo nie zapisaliśmy wyniku działania funkcji std::sort do zmiennej. Na wyjściu zobaczymy 5 4 8 2 9 10 3.

C. Nie kompiluje się - pętla for ma nieprawidłową składnię

D. Nie kompiluje się - funkcja std::sort przyjmuje tylko jeden argument - kontener do sortowania

E. Nie kompiluje się - wektora nie można zainicjalizować w podany sposób

Jednolita inicjalizacja zmiennych i pętle po kolekcji to elementy standardu C++11. Obecnie trudno wyobrazić sobie pisanie kodu w starszy sposób, gdzie używamy iteratorów do przechodzenia po kolekcji, a na wektor dodajemy elementy tylko metodą push_back().
Zaskoczony, że tak się da? Magia C++ 11. Pojawił się nowy rodzaj pętli po kolekcji. Sam algorytm sortowania opisany jest tutaj: https://en.cppreference.com/w/cpp/algorithm/sort

Pytanie 3:

Co się stanie podczas wywołania funkcji lessThanFive?

```
1  #include <vector>
2  #include <algorithm>
3
4  auto lessThanFive()
5  {
6      std::vector<int> v{3,1,4,2,5};
7      return std::none_of(v.begin() ,v.end(), [](const int e){ return e<5; });
8  }
```

A. Nic, kod się nie skompiluje
B. Zwróci true
C. Zwróci false
D. Zwróci iterator do elementu na 4-tej pozycji
E. Segmentation fault

O tym, jak działają algorytmy dowiesz się z lekcji STL (Standard Template Library). Biblioteka ta musi być dobrze znana każdemu programiście C++.
O tym co się stanie podczas tego wywołania można pocytać w dokumentacji biblioteki standardowej języka C++: https://en.cppreference.com/w/cpp/algorithm/all_any_none_of

Pytanie 4:

Jakiego typu będzie zmienna d?

```
1  #include <list>
2
3  int main() {
4      const std::list<int> list;
5      auto& d = list;
6      return 0;
7  }
```

A. std::list<int>
B. const std::list<int>&
C. std::list<int>&
D. const std::list<int>
E. Kod się nie skompiluje, bo zmienna list ma taką samą nazwę jak typ

O tym, jaki typ zostanie wydedukowany przez kompilator dowiesz się na szkoleniu z C++11. Obecnie mamy już standard C++17, czyli od 6 lat możemy używać automatycznej dedukcji typów. Nie uczyli o tym na uczelni? Hmm... a to już antyczna wiedza.
W C++11 słówko kluczowe auto oznacza automatyczną dedukcję typu. Zasady dedukcji mogą nie być intuicyjne na pierwszy rzut oka, ale dedukcja działa tak samo jak w przypadku szablonów.

Pytanie 5:

Jaki jest problem z poniższym kodem?

```
1   class Msg{};
2
3   void processMsg(Msg* msg)
4   {
5       // ...
6       delete msg;
7   }
8
9   int main() {
10      Msg m;
11      processMsg(&m);
12      return 0;
13  }
```

A. Wyciek pamięci
B. Nie ma problemu, kod się kompiluje i działa
C. Usuwamy dane zaalokowane na stosie
D. Usuwamy dane zaalokowane na stercie

O tym, dlaczego ten program jest niepoprawny i jak odróżnić stos od sterty dowiesz się w części Zarządzanie pamięcią.
Programista C++ powinien rozróżniać stos i stertę oraz wiedzieć, czemu nie usuwa się ręcznie danych zaalokowanych na stosie. Ręcznie można usuwać tylko dane ze sterty.

Pytanie 6:

Czy poniższy kod jest prawidłowy?

```
1   void processArray(int* array)
2   {}
3
4   int main() {
5       int* array = new int[10];
6       processArray(array);
7       delete array;
8       array = nullptr;
9       delete array;
10      return 0;
11  }
```

A. Tak
B. Nie, są wycieki pamięci
C. Nie, usuwamy tablicę zaalokowaną na stercie
D. Nie, usuwamy dwukrotnie te same dane
E. Nie, po usunięciu danych nie zerujemy wskaźnika array

W module Zarządzanie pamięcią dowiesz się dlaczego programy pisane w ten sposób często wydają się poprawne, tylko trzeba je co jakiś czas restartować, bo system operacyjny nie pozwala im alokować więcej pamięci. Niestety wielu programistów nie potrafi sprzątać pamięci po sobie.
W tym kawałku kodu alokujemy tablicę 10 intów. Musimy w tym celu zwolnić zaalokowaną pamięć używając tablicowego operatora delete[]. Usuwanie pustego wskaźnika nie jest szkodliwe, więc możemy to robić nawet 10 razy :)

Pytanie 7:

Jak najlepiej poprawić ten program?

```
1  #include <stdexcept>
2
3  struct Resource {
4      void use(char N) {
5          if (N == 'd') {
6              throw std::logic_error("Passed d. d is prohibited.");
7          }
8      };
9  };
10
11 int main() {
12     try {
13         auto rsc = new Resource();
14         rsc->use('d');
15         delete rsc;
16     }
17     catch (std::logic_error & e) {
18         e.what();
19     }
20     return 0;
21 }
```

A. Zamienić rsc na std::unique_ptr<Resource>
B. Zamienić rsc na std::shared_ptr<Resource>
C. Dodać zwalnianie pamięci rsc w sekcji catch
D. Zamiast rzucać wyjątkiem funkcja use powinna zwracać kod błędu

W module o inteligentnych wskaźnikach dowiesz się, jak można ich używać, aby zapobiegać wyciekom pamięci. Dowiesz się też, kiedy stosować unique_ptr, a kiedy shared_ptr. Współczesny programista C++ korzysta z inteligentnych wskaźników, gdyż zapewniają one prawidłowe zarządzanie pamięcią na poziomie języka C++.
Unique_ptr<T> to jeden z inteligentnych wskaźników, które weszły do standardu języka C++. Automatycznie zarządza on za nas pamięcią, w związku z czym nie musimy pamiętać o tym, aby zawołać delete :). W przypadku powyższego kodu nastąpi wyciek pamięci, jeśli zostanie rzucony wyjątek. Gdy będzie tam unique_ptr<Resource> to wyciek nie nastąpi.

Pytanie 8:

Poniższy kod:

```
1  #include <iostream>
2  #include <stdexcept>
3
4  struct ThrowingObject {
5      ~ThrowingObject() {
6          throw std::runtime_error("error in destructor");
7      }
8  };
9
10 void foo() {
11     throw std::runtime_error("Error");
12 }
13
14 int main() {
15     try {
16         ThrowingObject inside;
17         foo();
18     } catch(std::exception const&) {
19         std::cout << "std::exception" << std::endl;
20     } catch(std::runtime_error const&) {
21         std::cout << "std::runtime_error" << std::endl;
22     }
23 }
```

A. Wypisze "std::exception"
B. Wypisze "std::runtime_error"
C. Wypisze "error in destructor"
D. Wypisze "Error"
E. Jest niepoprawny i nastąpi zawołanie std::terminate()

O tym, jak działają wyjątki i dlaczego nie powinno się rzucać wyjątków w destruktorze, dowiesz się w module o Wyjątkach.
W związku z tym że mechanizm odwijania stosu musi usunąć obiektu, które w momencie rzucenia wyjątku znajdywały się na stosie, to będą wołane ich destruktory. W tym kodzie rzucamy wyjątkiem podczas gdy obsługujemy już inny wyjątek. Coś takiego powoduje ubicie programu poprzez zawołanie std::terminate().

Jeśli dużo rzeczy w tych pytaniach było dla Ciebie nowe, to nie przejmuj się. Wiemy czego uczą prowadzący na uczelniach i jakie są programy nauczania. Uczelnia nigdy nie była nastawiona na to, aby student nauczył się programowania, tylko na to, aby zrealizować program. Nas wyróżnia to, nasz program nauczania jest modyfikowany i jest tak naprawdę inny na każdym kursie. Jest on dostosowywany do poziomu grupy, więc jeśli komuś trzeba jakiś temat wytłumaczyć bardziej od podstaw, to właśnie to robimy. Coders School jest szkołą nastawioną na nauczenie, a nie zrealizowanie programu. Gwarantujemy że cały materiał zapamiętasz na długo. Mamy odpowiednie techniki w zanadrzu - bez kar cielesnych ;)

W razie pytań zachęcam do kontaktu.

Łukasz Ziobroń
Coders School
lukasz@coders.school