Programowanie obiektowe pierwsze kolokwium teoretyczne – zbiór zadań - rozwiązania

Zadanie 1

Podaj wyniki pracy następujących programów. Następnie odpowiedz na pytania znajdujące się poniżej każdego z przykładów. Odpowiedź uzasadnij.

Program 1 (dst)

```
#include <iostream>
using namespace std;
class X {
private: // 1
int a;
double b;
public:
X(int, double);
~X( );
void zmien(int);
void wypisz( );
};

X::X(int aa, double bb){
this->a=aa;
this->b=bb;
cout<<"konstruktor\n";
}

X::~X( ) {
      cout<<"destruktor "<<this->a<<" "<< this->b<<endl;
      }

void X::wypisz( ){
      cout<<this->a<<" *** "<< this->b<<endl; //2
      }

void X::zmien(int liczba){
      this->a=liczba* this->a;
      }

int main( ) {
      X x{2, 7.2};
      X *y=new X{3, 8.9};
      x.wypisz( );
      y->wypisz( );
      x.zmien(2);
      y->zmien(-3);
      x.wypisz( );
      y->wypisz( );
      delete y;
      return 0;
      }
```

Wymagania na ocenę:

dostateczną:
- definicja i korzystanie z prostej klasy (w tym konstruktor, destruktor, metody i mechanizm przyjaźni) - słowa kluczowe static, const w klasach

dobrą i bardzo dobrą
- wymagania na ocenę dostateczną - słowa kluczowe mutable, default, delete, explicit w klasach
- kopiowanie obiektów (tzn. konstruktor kopiujący i operator przypisania kopiujący),
- przenoszenie obiektów (tzn. konstruktor przenoszący, operator przypisania przenoszący)
- przeciążanie operatorów

Zadania oznaczone przez **(dst)** są zadaniami na ocenę dostateczną.

Pytania:

* Czy w linii oznaczonej **1** można opuścić słowo kluczowe **`private`**? Odpowiedź uzasadnij. * Tak, słowo kluczowe "private" jest opcjonalne, ponieważ domyślnie wszystkie elementy klasy są prywatne. Można by je opuścić, a kod zachowałby się tak samo. Jednak dodanie "private" jest dobrym zwyczajem programistycznym, ponieważ wyraźnie wskazuje na dostępność tylko wewnątrz klasy.

Czy w linii oznaczonej **2** można opuścić słowo kluczowe **`this`**? Odpowiedź uzasadnij. * Nie, słowo kluczowe "this" jest konieczne, aby odwołać się do składowych obiektu wewnątrz klasy. Bez "this" kompilator nie będzie wiedział, do którego obiektu odnosi się dana składowa.

Czy metoda wypisz z klasy X mogłaby być metodą stałą? Odpowiedź uzasadnij. *
Tak, metoda wypisz mogłaby być metodą stałą (const). Oznacza to, że metoda nie zmienia stanu obiektu i może być wywoływana na stałych obiektach klasy. Oznaczenie jej jako stałej zapewnia, że nie zmienia ona żadnych składowych obiektu, co jest dobrym praktyką w programowaniu.

Czy konstruktor w klasie X jest konstruktorem domyślnym?
Nie, konstruktor w klasie X nie jest konstruktorem domyślnym. Konstruktor domyślny to konstruktor, który nie przyjmuje żadnych argumentów i jest generowany automatycznie przez kompilator, jeśli nie zdefiniowano innych konstruktorów w klasie. Konstruktor w klasie X przyjmuje argumenty (int i double), więc nie jest domyślnym konstruktorem.

* Napisz definicję konstruktora klasy X z listą inicjalizacji składowych.
```
X::X(int aa, double bb) : a(aa), b(bb) {
      cout << "konstruktor\n";
 }
```

Program 2 (dst)
```
#include <iostream>
using namespace std;

class X {
      private:
      int a;
      double b;

      public:
      X(int aa=2, double bb=3):a{aa}, b{bb}
      {cout<<"konstruktor\n";}//1 ~X( ) {cout<<"destruktor\n";}
      void zmien(int);
      void wypisz( );
       };

void X::wypisz( ){
      cout<< this->a<<" *** "<< this->b<<endl;
       }

void X::zmien(int liczba){
      this->b=liczba* this->a;
      this->a=liczba;
       }

int main( ){
      X *tab=new X[2]{{8,9}};
      tab[0].wypisz( );
      tab[1].wypisz( );
      tab[0].zmien(2);
      tab[1].zmien(-1);
      tab[0].wypisz( );
      tab[1].wypisz( );
      delete [ ]tab; //2
      return 0;
       }
```
Pytania:
* Czy w konstruktorze klasy X (linia oznaczona 1) można opuścić wartości domyślne argumentów? Odpowiedź uzasadnij.
Tak, w konstruktorze klasy X można opuścić wartości domyślne argumentów. Wartości domyślne są używane wtedy, gdy konstruktor jest wywoływany bez podawania argumentów. Jeśli wartości domyślne są określone, to konstruktor może być wywołany bez argumentów, ale można także podać

własne argumenty, które nadpiszą wartości domyślne. W przypadku tego konstruktora, argumenty domyślne są 2 i 3.

* Czy w definicji konstruktora klasy X wymagana jest lista inicjalizacji składowych?
W tym przypadku, lista inicjalizacji składowych nie jest wymagana, ale jest zalecana. Lista inicjalizacji jest używana do bezpośredniego inicjowania składowych obiektu, co może być bardziej efektywne niż przypisywanie wartości wewnątrz ciała konstruktora. W tym przypadku, lista inicjalizacji została użyta, aby zainicjować składowe a i b za pomocą argumentów konstruktora, co jest dobrym praktyką programistyczną.

Odpowiedź uzasadnij. * Czy w linii oznaczonej **2** można napisać:
```
delete tab;
```
Odpowiedź uzasadnij.
Nie, nie można użyć "delete tab;" w linii oznaczonej 2. W przypadku dynamicznie alokowanej tablicy, takiej jak "tab", do zwolnienia pamięci należy użyć "delete [] tab;", gdzie "[]" informuje kompilator, że zwalniamy pamięć po tablicy, a nie po pojedynczym obiekcie. Użycie "delete tab;" w takim przypadku spowoduje niezdefiniowane zachowanie.

* Czy do klasy X możemy dodać konstruktor jednoargumentowy? Odpowiedź uzasadnij.
Tak, do klasy X można dodać konstruktor jednoargumentowy. Konstruktor jednoargumentowy pozwoliłby tworzyć obiekty klasy X z jednym argumentem, co może być przydatne w niektórych sytuacjach. Jeśli zostanie dodany konstruktor jednoargumentowy, to należy dostosować definicje obiektów w funkcji main, aby pasowały do nowego konstruktora. To może zwiększyć elastyczność klasy i umożliwić tworzenie obiektów z różnymi konfiguracjami.

Program 3 (dst)
```
#include <iostream>
using namespace std;

class X {
      private:
      int a;
      double b;

      public:
      X(int aa, double bb):a{aa}, b{bb} {cout<<"konstruktor\n";}
      ~X( ) {cout<<"destruktor "<<this->a<<" "<<this->b<<endl;}
      void zmien(int);
      void wypisz( );
       };
```


```
void X::wypisz( ){
      cout<< this->a<<" *** "<< this->b<<endl;
       }

void X::zmien(int liczba){
      this->a=liczba* this->a;
       }

int main( ){
      X x{8, 4.2};
      x.wypisz( );
      x.zmien(2);
```

```
      x.wypisz( );
      X y{x};
      y.wypisz( );
      y.zmien(-1);
      y.wypisz( );
      return 0;
      }
```

Pytania:

* Uwzględniając powyższe definicje odpowiedz na pytanie. Czy poprawna jest poniższa linia? `X x{8};`

Odpowiedź uzasadnij.

Nie, linia ta nie jest poprawna. Konstruktor klasy X oczekuje dwóch argumentów: int i double. Przy próbie utworzenia obiektu x przy użyciu jednego argumentu (8) nastąpi błąd kompilacji. Musisz dostarczyć oba oczekiwane argumenty lub użyć konstruktora z wartościami domyślnymi, jeśli takie są dostępne (w tym przypadku nie ma).

* Czy z pola b klasy X można uczynić pole stałe? Odpowiedź uzasadnij. *

Tak, z pola b klasy X można uczynić pole stałe, o ile pole to nie będzie modyfikowane po zainicjowaniu. Aby to zrobić, wystarczy dodać specyfikator "const" do deklaracji pola w klasie X:

Dodaj do klasy X konstruktor domyślny.

```
X() : a(0), b(0.0) {
      cout << "konstruktor domyślny\n";
 }
```

Program 4 (dst)

```
#include <iostream>
using namespace std;

class X {
      private:
      int a;
      double b;

      public:
      X(int aa, double bb):a{aa}, b{bb}
       {cout<<"konstruktor"<<a<<" "<<b<<endl;}
      ~X( ) {cout<<"destruktor\n";}
      void wypisz() {cout<<this->a<<" @ "<<this-
      >b<<endl;} friend void wypisz(const X&); //1
      };

void wypisz(const X& arg ){
      cout<<arg.a<<" %%% "<<arg.b<<endl;
      }

int main( ){
      X x{20, 17.2}; //2
      x.wypisz();
      wypisz(x);
      X y{23, -6};
      y.wypisz();
      wypisz(y);
      x=y;
      x.wypisz();
      y.wypisz();
      return 0;
      }
```

Pytania:

* Co się stanie, gdy w deklaracji przyjaciela (linia oznaczona 1) zostanie usunięte słowo kluczowe **friend**?

Usunięcie słowa kluczowego "friend" z deklaracji funkcji "wypisz" spowoduje, że ta funkcja przestanie być funkcją przyjacielską klasy X. To oznacza, że nie będzie miała dostępu do prywatnych składowych klasy X, takich jak a i b. Kompilacja zakończy się błędem, gdyż funkcja "wypisz" próbuje uzyskać dostęp do prywatnych składowych klasy X.

* Co się stanie, gdy obiekt `x` (linia oznaczona 2) uczynimy obiektem stałym?

Jeśli obiekt x zostanie uczyniony obiektem stałym poprzez dodanie specyfikatora "const" w deklaracji, np. `const X x{20, 17.2};`, to nie będzie można wywoływać na nim metod niemodyfikujących obiektu. Jednakże, nadal będzie można wywoływać funkcję "wypisz" z parametrem const (przyjaciela klasy), która może wyświetlać składowe obiektu x.

* Stosując inicjalizację wewnątrzklasową dodaj do klasy X konstruktor domyślny.

```
X() : a(0), b(0.0) {
      cout << "konstruktor domyślny " << a << " " << b <<
      endl;
}
```

Program 5 (dst bez konstruktora kopiującego)

```
#include <iostream>
using namespace std;

class X {
      private:
      int a;
      double b;

      public:
      X(int aa, double bb):a{aa}, b{bb} {cout<<"konstruktor\n";}
      ~X( ) {cout<<"destruktor\n";}
      X(const X& wzor): a{wzor.a}, b{wzor.b+1} { cout<<"&&&\n";}
      void wypisz() {cout<<this->a<<" @ "<<this->b<<endl;}
      friend void wypisz(X); //1
      };

void wypisz(X arg ){
      cout<<arg.a<<" %%% "<<arg.b<<endl;
      }

int main( ){
      X x{20, 17.2}; //2
      x.wypisz();
      wypisz(x);
      x.wypisz();
      X y{x};
      y.wypisz();
      wypisz(y);
      y.wypisz();
      return 0;
      }
```

Pytania:

* W jaki sposób można pominąć wywołanie konstruktora kopiującego w przyjacielu wypisz?

Aby pominąć wywołanie konstruktora kopiującego w przyjacielu wypisz, można przekazywać obiekt "X" jako referencję lub wskaźnik zamiast kopiować go jako argument. To oznacza, że funkcja "wypisz" nie tworzy nowego obiektu, tylko operuje na oryginalnym obiekcie przekazanym jako referencja lub wskaźnik.

Zmodyfikowany przyjaciel wypisz:

```
void wypisz(const X& arg) {
      cout << arg.a << " %%% " << arg.b << endl;
 }
```
Teraz przyjaciel przyjmuje obiekt "X" jako stałą referencję, co oznacza, że nie będzie tworzył kopii obiektu.

* W jaki sposób można zablokować wykonywanie kopii obiektów?

Aby zablokować wykonywanie kopii obiektów, można usunąć konstruktor kopiujący i operator przypisania albo zadeklarować je jako prywatne i nie dostępne z zewnątrz klasy. W przypadku klasy "X" z programu 5, konstruktor kopiujący jest już zdefiniowany i można go usunąć w całości, co spowoduje, że nie będzie można tworzyć kopii obiektów tej klasy.
Usunięcie konstruktora kopiującego (można to zrobić poprzez "= delete"):

```
X(const X& wzor) = delete;
```

* Zdefiniuj operator przypisania dla powyższej klasy.

```
X& operator=(const X& other) {
      if (this != &other) {
            this->a = other.a;
            this->b = other.b;
      }
 return *this;
 }
```
Ten operator przypisania kopiuje składowe obiektu "other" do bieżącego obiektu, o ile nie są to te same obiekty (sprawdzane za pomocą operatora "!="). Operator przypisania zwraca referencję do samego siebie, umożliwiając łańcuchowe przypisania.

Program 6

```
#include <iostream>
using namespace std;

class Liczba {
     double x;

     public:
     Liczba (double xx=12.5) : x{xx} {cout<<"***\n"; } //1
     Liczba (const Liczba& wzor) : x{wzor.x*10} { }
     Liczba& operator = (const Liczba & wzor)
           {cout<<"$$$\n"; x=wzor.x-5; return *this;}
     void wypisz( ) {cout<<x<<endl;}
      };
```



```
int main( ){
     Liczba obiekt{13.2};
     obiekt.wypisz( );
     Liczba x{obiekt};
     x.wypisz( );
     obiekt=x;
     obiekt.wypisz( );
     return 0;
      }
```

Pytania:

* Czy w powyższym programie konstruktor kopiujący jest poprawnie zdefiniowany? Odpowiedź uzasadnij. * Czy w powyższym programie operator przypisania jest poprawnie zdefiniowany? Odpowiedź uzasadnij. **Program 7**

```
#include <iostream>
using namespace std;

class Liczba {
     static int licznik;
     int *wsk;
     public:
     Liczba (int a=5) : wsk{new int} {*wsk=a; licznik++;}
     Liczba (const Liczba& wzor) : wsk{new int}
      {*wsk=*(wzor.wsk); licznik++;}
     ~Liczba ( ) {delete wsk; licznik--;}
     void wypisz ( ) {cout<<*wsk<<" "<<licznik<<endl;}
     void zmien (int b) {*wsk=b;}
      };

int Liczba::licznik{2}; //1

int main( ){
     Liczba x{12};
     Liczba y{x};
     x.wypisz( );
     y.wypisz( );
     y.zmien(200);
     x.wypisz( );
     y.wypisz( );
     return 0;
      }
```

Pytania:

* Jaki będzie wynik pracy powyższego programu, jeśli w linii 1 zostanie usunięta wartość początkowa tzn. linia 1 jest postaci:

```
int Liczba::licznik;
```

Jeśli usuniemy wartość początkową z linii 1, to zmienna statyczna "licznik" nie zostanie zainicjowana, i jej początkowa wartość będzie nieokreślona. To spowoduje, że program będzie miał niezdefiniowane zachowanie i wyniki będą niestabilne.

* Jakie mechanizmy powinny zostać zdefiniowane w klasie X? Odpowiedź uzasadnij.

W klasie Liczba nie ma klasy X, dlatego pytanie to nie dotyczy programu.

* W klasie Liczba zdefiniuj kopiujący operator przypisania.

```
Liczba& operator=(const Liczba& other) {
      if (this != &other) {
            delete wsk;
            wsk = new int(*other.wsk);
      }
 return *this;
 }
```

* W klasie Liczba zdefiniuj przenoszący operator przypisania.

```
Liczba& operator=(Liczba&& other) {
      if (this != &other) {
            delete wsk; wsk = other.wsk;
            other.wsk = nullptr;
      }
 return *this;
 }
```

* W klasie Liczba zdefiniuj przenoszący konstruktor.

```
Liczba(Liczba&& other) {
      wsk = other.wsk; other.wsk = nullptr;
 }
```

* Czy w konstruktorze kopiującym klasy można opuścić referencję w argumencie? Odpowiedź uzasadnij.

Nie, w konstruktorze kopiującym klasy nie można opuścić referencji w argumencie. Konstruktor kopiujący powinien przyjmować referencję do obiektu, który ma być skopiowany, aby uniknąć

rekurencyjnego wywoływania konstruktora kopiującego w trakcie tworzenia kopii. Bez referencji kopiowalibyśmy obiekt w nieskończoność, co doprowadziłoby do przepełnienia stosu.

Program 8

```
#include <iostream>
using namespace std;

class X {
      private:
      int a{0};
      double b{0};

      public:
      X()=default;//2
      X(int aa, double bb):a{aa}, b{bb}
      {cout<<"konstruktor\n";} ~X( ) {cout<<"destruktor "<<a<<"
      "<<b<<endl;}
      X(const X& wzor): a{wzor.a}, b{wzor.b} {
      cout<<"\n&&&\n";} void wypisz( );
      friend const X operator*(const X& arg1, const X&
      arg2); };

void X::wypisz( ) {
      cout<<a<<" *** "<<b<<endl;
      }

const X operator*( const X& arg1, const X& arg2) {
      return {arg1.a*arg2.a, arg1.b*arg2.b};
      }

int main( ){
      X x{2, 3.2}, y{1, 5},z{};
      x.wypisz( );
      y.wypisz( );
      z=x*y;//1
      z.wypisz( );
      return 0;
      }
```

Pytania:

* Czy w definicji klasy X można pominąć wartości początkowe pól klasy? Odpowiedź uzasadnij.

Tak, w definicji klasy X można pominąć wartości początkowe pól klasy, ale jeśli nie zostaną one podane, to zostaną użyte wartości domyślne, które są 0 dla inta i double. Dlatego można pominąć wartości początkowe pól, ale nie jest to

* Napisz jak wygląda jawne wywołanie operatorów w linii 1.

```
z = operator*(x, y);
```

Oznacza to, że obiekty x i y są przekazywane jako argumenty do funkcji operator*(), a wynik jest przypisywany do obiektu z.

* Czy w linii 2 można słowo kluczowe **default** zastąpić słowem kluczowym **delete**

Nie, słowo kluczowe "default" i "delete" mają zupełnie różne znaczenia w kontekście deklaracji konstruktorów. "default" oznacza, że konstruktor domyślny zostanie zainicjowany domyślnie, podczas gdy "delete" oznacza, że dany konstruktor jest usuwany (nie jest dostępny). W tym przypadku, "default" oznacza, że konstruktor domyślny jest dostępny i zostanie zainicjowany domyślnie, jeśli nie dostarczymy własnej definicji.

Usunięcie konstruktora domyślnego za pomocą "delete" jest konieczne, jeśli chcemy go wyłączyć.

* Przeciąż na rzecz klasy X operator<<

```
friend ostream& operator<<(ostream& os, const X& obj) {
        os << obj.a << " *** " << obj.b;
        return os;
  }
```

* Dodaj do klasy konstruktor jednoargumentowy w którego definicji skorzystasz z konstruktora dwuargumentowego.

```
X(int aa) : a(aa), b(0.0) {
        cout << "konstruktor jednoargumentowy\n";
  }
```

Zadanie 1

dodaj konstruktor, metodę zmieniającą wartość pola bok, stałą metodę wyznaczającą obwód prostokąta, przyjaciela wypisującego informacje nt. obiektów klasy `Kwadrat`.

Fragment 1 (dst)

Do klasy

```
class Kwadrat
{
double bok;
double obwod; };
```

Opdpowiedz:

```
class Kwadrat {
private:
   double bok;
   double obwod;

public:
   Kwadrat(double _bok) : bok(_bok), obwod(4 * _bok) {}
   void zmienBok(double nowyBok) { bok = nowyBok; obwod = 4 * nowyBok; }
   double obliczObwod() const { return obwod; }

   friend void wypiszKwadrat(const Kwadrat& k);
};

void wypiszKwadrat(const Kwadrat& k) {
   cout << "Kwadrat o boku: " << k.bok << ", obwód: " << k.obwod << endl;
}
```

Zadanie 2

dodaj konstruktor, metodę zmieniającą wartość pola bok, stałą metodę wyznaczającą obwód prostokąta, przyjaciela wypisującego informacje nt. obiektów klasy Kwadrat. dodaj konstruktor, destruktor, konstruktor kopiujący, operator przypinania oraz metodę wypisującą informacje na temat obiektów klasy Moja

Fragment 2

Do klasy

```
class Moja
{
int a;
double *b;
};
```

Opdpowiedz:

```
class Moja {
private:
    int a;
    double* b;

public:
    Moja(int _a, double _b) : a(_a), b(new double(_b)) {}
    ~Moja() { delete b; }
    Moja(const Moja& wzor) : a(wzor.a), b(new double(*(wzor.b))) {}
    Moja& operator=(const Moja& wzor) {
        if (this != &wzor) {
            a = wzor.a;
            delete b;
            b = new double(*(wzor.b));
        }
        return *this;
    }
    void wypiszInfo() const {
        cout << "a: " << a << ", b: " << *b << endl;
    }
};
```

Zadanie 3

Napisz jakie są różnice między:

1. Klasą a strukturą.

Klasa i struktura w C++ są niemal identyczne pod względem definicji i składni. Różnica polega na domyślnym dostępie do składowych: w klasie jest to "private", a w strukturze "public". Można nadawać różne etykiety dostępu (public, private, protected) zarówno składowym klas jak i struktur.

2. Obiektem a klasą.

- Klasa to abstrakcyjny szablon opisujący strukturę danych i zachowanie.
- Obiekt to konkretne wystąpienie klasy, który przechowuje dane i umożliwia wywoływanie metod opisanych w klasie.

3. Etykietą public a private (protected).

- "public" oznacza dostępność składowej zarówno wewnątrz klasy, jak i z zewnątrz.
- "private" oznacza, że składowa jest dostępna tylko wewnątrz klasy.
- "protected" jest podobne do "private", ale pozwala na dostęp dla klas dziedziczących.

4. Metodą klasy a funkcją, która nie jest metodą. dodaj konstruktor, destruktor oraz stałą metodę wypisującą informacje na temat składowych obiektów klasy Produkt. Zabroń wywoływania konstruktora kopiującego oraz operatora przypisania. Przeciąż na rzecz tej klasy operator wyjścia.

- Metoda klasy jest funkcją, która jest związana z daną klasą i ma dostęp do jej składowych.
- Funkcja, która nie jest metodą, to zwykła funkcja, która nie jest przypisana do klasy i nie ma dostępu do jej składowych.

5. Definicją metody w klasie a definicją metody poza klasą.

- Metodę klasy definiuje się wewnątrz deklaracji klasy. Jest to związane z daną klasą i ma dostęp do jej składowych.
- Funkcję, która nie jest metodą, definiuje się poza deklaracją klasy. Nie jest związana z daną klasą i nie ma dostępu do jej składowych bez argumentów.

6. Konstruktorem a destruktorem klasy.

- Konstruktor jest wywoływany podczas tworzenia obiektu i inicjalizuje jego składowe.
- Destruktor jest wywoływany podczas zniszczenia obiektu i może służyć do zwalniania zasobów lub wykonywania innych czynności sprzątających.

7. Klasą z konstruktorem a klasą bez żadnego konstruktora.

- Klasa z konstruktorem ma zdefiniowany konstruktor, który może inicjować obiekty.
- Klasa bez konstruktora nie ma zdefiniowanego konstruktora i obiekty tej klasy są inicjowane domyślnie (jeśli to możliwe) lub przy użyciu inicjalizatorów domyślnych.

8. Konstruktorem domyślnym klasy a konstruktorem który domyślny nie jest.
   - Konstruktor domyślny jest konstruktorem, który jest wywoływany bez argumentów lub z argumentami o domyślnych wartościach.
   - Konstruktor, który domyślny nie jest, to konstruktor, który nie jest zdefiniowany w klasie.

9. Konstruktorem ze słowem kluczowym explicit i konstruktorem bez tego słowa kluczowego.
   - Konstruktor z "explicit" nie pozwala na niejawne rzutowanie typów przy tworzeniu obiektu, co może zapobiec nieoczekiwanym konwersjom.
   - Konstruktor bez "explicit" pozwala na niejawne rzutowanie typów, co może prowadzić do nieintuicyjnych zachowań.

10. Konstruktorem delegowanym a konstruktorem który delegowany nie jest.
    - Konstruktor delegowany to konstruktor, który wywołuje inny konstruktor wewnątrz swojej definicji.
    - Konstruktor, który delegowany nie jest, to konstruktor, który nie wywołuje innych konstruktorów wewnątrz swojej definicji.

11. Konstruktorem (destruktorem) a metodą klasy.
    - Konstruktor jest wywoływany podczas tworzenia obiektu i inicjalizuje go.
    - Destruktor jest wywoływany podczas zniszczenia obiektu i może służyć do zwalniania zasobów.
    - Metoda klasy to funkcja związana z daną klasą i ma dostęp do jej składowych.

12. Funkcją, która przyjaźni się z klasą a funkcją, która nie przyjaźni się z klasą.
    - Funkcja, która jest przyjacielem klasy, ma dostęp do jej prywatnych składowych.
    - Funkcja, która nie jest przyjacielem klasy, nie ma dostępu do jej prywatnych składowych.

13. Polem z modyfikatorem const i polem bez tego modyfikatora.
    - Pole z modyfikatorem "const" jest niezmienne i nie można zmieniać jego wartości po zainicjowaniu.
    - Pole bez "const" może zmieniać swoją wartość po inicjacji

14. Metodą stałą a metodą, która stała nie jest.
    - Metoda stała jest metodą, która nie zmienia stanu obiektu i jest oznaczona jako "const".
    - Metoda, która stała nie jest, może zmieniać stan obiektu i nie jest oznaczona jako "const".

15. Polem z modyfikatorem static i polem bez tego modyfikatora.
    - Pole z modyfikatorem "static" jest wspólne dla wszystkich obiektów klasy, a nie dla każdego obiektu osobno.
    - Pole bez "static" jest osobne dla każdego obiektu i każdy obiekt ma swoją kopię tego pola.

16. Metodą statyczną a metodą, która nie jest statyczna.
    - Metoda statyczna nie jest związana z konkretnym obiektem i może być wywołana bez utworzenia obiektu.
    - Metoda, która nie jest statyczna, jest związana z konkretnym obiektem i jest wywoływana na rzecz tego obiektu.

17. Polem z modyfikatorem mutable a polem bez tego modyfikatora.
    - Pole z modyfikatorem "mutable" może być modyfikowane, nawet w metodach stałych.
    - Pole bez "mutable" nie może być modyfikowane w metodach stałych.

18. Słowami kluczowymi default i delete w kontekście klas.
    - default" oznacza, że dana funkcja (konstruktor, destruktor, operator) będzie generowana automatycznie przez kompilator.
    - "delete" oznacza, że dana funkcja zostaje zablokowana i nie będzie dostępna.

19. Kopiującym operatorem przypisania a konstruktorem kopiującym.
    - Kopiujący operator przypisania pozwala na przypisanie wartości jednego obiektu do drugiego po ich inicjacji.
    - Konstruktor kopiujący jest wywoływany podczas tworzenia kopii obiektu.

20. Konstruktorem kopiującym a konstruktorem przenoszącym.
    - Konstruktor kopiujący jest wywoływany podczas tworzenia kopii obiektu i tworzy głęboką kopię.
    - Konstruktor przenoszący jest wywoływany podczas przenoszenia zasobów z jednego obiektu do drugiego i jest wydajniejszy niż kopiowanie.

21. Kopiującym operatorem przypisania a przenoszącym operatorem przypisania
    - Kopiujący operator przypisania przypisuje kopię jednego obiektu do drugiego.
    - Przenoszący operator przypisania przekazuje zasoby z jednego obiektu do drugiego, co jest bardziej wydajne.

22. Przekazywaniem argumentów reprezentujących klasy przez wartość a przez referencję.
    - Przekazywanie przez wartość kopiuje obiekt do funkcji, co może być kosztowne.
    - Przekazywanie przez referencję pozwala na manipulację oryginalnym obiektem i jest bardziej wydajne.