Zadanie 1

Co pojawi się na ekranie po wykonaniu następujących linii kodu? Następnie odpowiedz na pytania znajdujące się poniżej każdego z przykładów. Odpowiedź uzasadnij.

Wymagania na ocenę:
dostateczną:
- definicja i korzystanie z prostej hierarchii dziedziczenia klas (w tym konstruktor i destruktor klasy bazowej i kasy pochodnej, metody, wywoływanie metod, standardowe konwersje w górę hierarchii dziedziczenia)
- dziedziczenie publiczne, prywatne, chronione
- ogólna wiedza dotycząca metod wirtualnych w tym koncepcja klas abstrakcyjnych

dobrą i bardzo dobrą
- wymagania na ocenę dostateczną
- słowo kluczowe virtual w klasach (metody wirtualne, metody „czysto" wirtualne, klasy abstrakcyjne, rzutowanie w dół hierarchii dziedziczenia - operator **dynamic_cast**)
- słowa kluczowe **final** i **override**
- dziedziczenie wielokrotne – wirtualne klasy bazowe

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

Program 1 (dst)

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

class Baza {
protected: //1
int a;

public:
Baza(int x=5, int y=2) : a{x*y} //2
    {cout<<"\nBaza konstruktor\n" ; }
~Baza( )
    { cout<<"Baza destructor\n";}
void wypisz( ) const
    {cout<<"( "<<a<<" )\n"; }
void ustaw_a(int arg) {a+=arg ;}
};

class Pochodna: public Baza {
  private: //3
  int b;
  double c;

  public:
  Pochodna(int p, int q, double r):Baza{p,p}, b{q}, c{r} //4
                {cout<<"\nPochodna konstruktor\n";}
  ~Pochodna( ) { cout<<"Pochodna destruktor\n";}
  void ustaw_c(double arg) {c=b/arg;}
  void wypisz( ) const;
  };

void Pochodna::wypisz( ) const {
     Baza::wypisz();
     cout<<" "<<b<<"; "<<c<<endl;
     }

int main( ) {
     Baza obiekt1{5};
     obiekt1.wypisz( );
     obiekt1.ustaw_a(3);
     obiekt1.wypisz( );
     Pochodna obiekt2{2, 7, 3.4};
     obiekt2.wypisz( );
     obiekt2.ustaw_a(5);
     obiekt2.ustaw_c(10);
     obiekt2.wypisz( );
     return 0;
     }
```

Pytania:

* Czy w linii oznaczonej **1** można opuścić słowo kluczowe **protected**? Odpowiedź uzasadnij.

Nie, nie można opuścić słowa kluczowego `protected` w linii oznaczonej 1. W tej linii jest deklaracja zmiennej `a`, która ma być dostępna dla klas pochodnych, więc konieczne jest użycie modyfikatora dostępu `protected`. Pozwala to klasom dziedziczącym mieć dostęp do zmiennej `a`, ale jednocześnie chroni ją przed bezpośrednim dostępem z poziomu innych klas.

* Czy w linii oznaczonej **2** można opuścić wartości domyślne argumentów konstruktora? Odpowiedź uzasadnij.

Tak, można opuścić wartości domyślne argumentów konstruktora w linii oznaczonej 2. Domyślne wartości argumentów są używane, gdy nie są one dostarczone przy wywoływaniu konstruktora. W tym przypadku, jeśli nie zostaną dostarczone żadne argumenty, wartości domyślne `x=5` i `y=2` zostaną użyte.

* Czy w linii oznaczonej **3** można zamiast **private** napisać **protected**? Odpowiedź uzasadnij.

Tak, można zmienić `private` na `protected` w linii oznaczonej 3, ale należy pamiętać, że to zmienia dostępność zmiennych `b` i `c` w klasie `Pochodna`. Jeśli zmienna jest oznaczona jako `private`, to jest ona dostępna tylko w obrębie klasy, natomiast `protected` pozwala na dostęp również dla klas dziedziczących. Decyzja zależy od intencji projektanta, czy chce udostępnić te zmienne klasom pochodnym.

* Czy w linii oznaczonej **4** można zrezygnować z wywołania konstruktora klasy bazowej? Odpowiedź uzasadnij.

Nie, nie można zrezygnować z wywołania konstruktora klasy bazowej w linii oznaczonej 4. W konstruktorze klasy pochodnej `Pochodna` wywoływany jest konstruktor klasy bazowej `Baza` przy użyciu składni inicjalizacji listy (`: Baza{p, p}`). To jest konieczne do poprawnego zainicjowania składowej `a` klasy `Baza`. Omitowanie tego wywołania mogłoby prowadzić do nieprawidłowego stanu obiektu klasy pochodnej.

Program 2 (dst)

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

class Baza
{
protected: //1
int a;
double b;
public:
Baza(int x, double y) : a{x}, b{y}
        {cout<<"\nkonstruktor "<<a<<" "<<b<<"\n" ; }
~Baza( )
        {cout<<"destruktor "<<a<<" "<<b <<endl;}
void wypisz( ) const
        {cout<<endl<<a<<" # "<<b<<endl; }
void zmiany(int arg)
        {a=arg ; b+=arg;}
};

class Pochodna: public Baza //2
  {
  private:
  int c;
  double d;

  public:
  Pochodna(int p, double q):Baza{p,q}, c{p+1}, d{q/10} //3
          {cout<<"\nKONSTRUKTOR "<<c<<" "<<d<<"\n";}
  ~Pochodna( )
          {cout<<"DESTRUKTOR "<<c<<" "<< d<<endl;}
  void zmiany(int arg) { c*=arg; d+= d ;}
  void wypisz( ) const;
  };

void Pochodna::wypisz( ) const
```

```
    {
    cout<<endl<< a<<", "<< b
         <<", "<< c<<", "<< d<<endl;
    }

int main(){
    Baza obiekt1{5, 4.9};
    obiekt1.zmiany(3);
    obiekt1.wypisz( );
    Pochodna obiekt2{2, 11.5};
    obiekt2.wypisz( );
    obiekt2.zmiany(-2);
    obiekt2.wypisz( );
    obiekt2.Baza::zmiany(4);
    obiekt2.wypisz( );
    obiekt2.Baza::wypisz( );
    return 0;
    }
```

Pytania:

* Czy w linii oznaczonej **1** można zrezygnować z etykiety **protected**? Odpowiedź uzasadnij.

Tak, można zrezygnować z etykiety `protected` w linii oznaczonej 1, ale należy pamiętać, że składowa `a` będzie wtedy prywatna, co oznacza, że nie będzie dostępna dla klas pochodnych. Zazwyczaj, gdy zmienna ma być dziedziczona, stosuje się `protected`, aby umożliwić dostęp klasom pochodnym, ale jednocześnie utrudnić dostęp z zewnątrz klasy.

* Czy na liście dziedziczenia klasy Pochodna (linia oznaczona **2**) można opuścić etykietę **public**? Odpowiedź uzasadnij.

Nie, nie można opuścić etykiety `public` na liście dziedziczenia w linii oznaczonej 2. Domyślnie dziedziczenie jest prywatne, co oznaczałoby, że wszystkie składowe klasy bazowej `Baza` byłyby prywatne w klasie `Pochodna`, co mogłoby prowadzić do błędów dostępu. Stosuje się `public`, aby umożliwić dostęp do dziedziczonych składowych w klasie pochodnej.

* Czy w linii oznaczonej **3** można zrezygnować z wywołania konstruktora klasy bazowej? Odpowiedź uzasadnij.

Nie, nie można zrezygnować z wywołania konstruktora klasy bazowej w linii oznaczonej 3. Wywołanie konstruktora klasy bazowej `Baza` jest konieczne do poprawnej inicjalizacji składowych `a` i `b` w klasie `Baza`. Pomija się to wywołanie, można wprowadzić nieprawidłowy stan obiektu klasy pochodnej `Pochodna`. Wywołanie konstruktora klasy bazowej jest często niezbędne, aby inicjalizować dziedziczone składowe z klasy bazowej

Program 3 (dst)

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

 class X
 {
 protected:
 int a;
 public:
 X(int x) : a{x} { cout<<a<<" konstruktor X\n";}
 ~X( ) { cout<< a<<" destruktor X\n";}
 void metoda(int aa) { a=aa;}
 void wypisz( ) const {cout<<"\n( "<< a<<" )\n"; }
 };
 class Y : public X
 {
 protected:
 int b;
 public:
 Y(int x,int y):X{x},b{y} { cout<< a<<" "<< b<<" konstruktor Y\n";}
 ~Y( ) {cout<< a<<" "<< b<<" destruktor Y\n";}
 void metoda(int x) { b= b+x;}
 void wypisz( ) const {X::wypisz( ); cout<< b<<endl; }
 };

 int main()
 {
 X o1{45};
 o1.wypisz();
 o1.metoda(-4);
 o1.wypisz();
 Y o2{1,7};
 o2.wypisz();
 o2.metoda(3);
 o2.wypisz();
 X *wsk1=new X{23};
 X *wsk2=new Y{2,4};
 wsk1->wypisz();
 wsk2->wypisz();
 wsk1->metoda(9);
 wsk2->metoda(12);
 wsk1->wypisz();
 wsk2->wypisz();
 delete wsk1; delete
 wsk2; return 0;
 }
```

Pytania:

* Czy w powyższej hierarchii dziedziczenia dziedziczenie publiczne można zastąpić dziedziczeniem prywatnym? Odpowiedź uzasadnij.

Tak, można zastąpić dziedziczenie publiczne dziedziczeniem prywatnym, ale to zmieni dostępność dziedziczonych składowych. W tym przypadku, składowe klasy `X` byłyby prywatne w klasie `Y`, co oznaczałoby, że nie byłyby dostępne publicznie w klasie `Y`. Wprowadzenie dziedziczenia prywatnego jest możliwe, ale zmienia interfejs dostępu do składowych.

* Czy w powyższej hierarchii dziedziczenia dziedziczenie publiczne można zastąpić dziedziczeniem chronionym? Odpowiedź uzasadnij.

Tak, można zastąpić dziedziczenie publiczne dziedziczeniem chronionym. Wówczas składowe klasy `X` byłyby dostępne jako chronione w klasie `Y`, co oznaczałoby, że byłyby dostępne dla klas pochodnych, ale nie byłyby publicznie dostępne. To może być

uzasadnione, jeśli projektant chce udostępnić składowe dla klas pochodnych, ale jednocześnie zabezpieczyć je przed bezpośrednim dostępem z zewnątrz.

* Czy powyższa hierarchia dziedziczenia jest poprawnie zdefiniowania? Odpowiedź uzasadnij.

Tak, powyższa hierarchia dziedziczenia jest poprawnie zdefiniowana. Klasa `Y` dziedziczy publicznie po klasie `X`, co oznacza, że składowe chronione klasy `X` są dostępne w klasie `Y`. Konstruktor klasy `Y` wywołuje konstruktor klasy `X` poprzez listę inicjalizacyjną, co jest zalecaną praktyką. Klasa `Y` przesłania metodę `wypisz` klasy `X` i dodaje własną funkcjonalność. W mainie obiekty klasy `X` i `Y` są tworzone i niszczone poprawnie, a wskaźniki klasy bazowej (`X`) są używane do obsługi obiektów klasy pochodnej (`Y`).

Program 4 (dst)

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

class X
{
protected:
int a, b;

public:
X(int x=50, int y=6) : a{x}, b{y} {cout<<"\n***\n" ; }
~X( ) { cout<<"destruktor\n";}
void wypisz( ) const {cout<<"( "<< a<<", "<< b<<" )\n"; }
};

class Y: public X
  {
  private:
  int c;

  public:
  Y(int p, int q=11):X{p,q}, c{p+q} {cout<<"\n+++\n";}
  ~Y( ) { cout<<"DESTRUKTOR\n";}
  friend void pisz(const Y& co );
  };

void pisz(const Y& co ) {
     co.wypisz( );
     cout<<" "<<co.c<<endl;
     }
int main( )
     {
     X P{30, 2};
     P.wypisz( );
     Y S{70};
     S.wypisz( );
     pisz(S);
     return 0;
     }
```

Pytania:

* Czy przyjaciela pisz można wywołać w następujący sposób?: pisz(P);

Odpowiedź uzasadnij.

Nie, nie można wywołać funkcji `pisz` dla obiektu klasy `X` w następujący sposób: `pisz(P);`. Powód tego wynika z faktu, że funkcja `pisz` została zadeklarowana jako funkcja przyjacielska w klasie `Y`. Funkcje przyjacielskie mają dostęp do prywatnych i chronionych składowych klasy, ale są związane z konkretną klasą, dla której zostały zadeklarowane.

W tym przypadku, `pisz` jest funkcją przyjacielską klasy `Y`, więc ma dostęp do prywatnej składowej `c` klasy `Y`. Natomiast klasa `X` nie ma dostępu do składowych prywatnych klasy `Y`, więc próba wywołania `pisz(P);` dla obiektu klasy `X` zakończy się błędem kompilacji.

Aby użyć funkcji `pisz` dla obiektu klasy `X`, funkcję tę należy zadeklarować jako funkcję przyjacielską również w klasie `X` lub umieścić ją w globalnym kontekście (bez deklaracji jako funkcja przyjacielska w klasie). Jednakże, wówczas funkcja `pisz` nie miałaby dostępu do prywatnej składowej `c` klasy `Y`.

Program 5

```
#include <iostream>
using namespace std;
class X {
protected:
int a;
public:
X(int x) : a{x} { cout<<a<<" konstruktor X\n";}
virtual ~X( ) { cout<<a<<" destruktor X\n";}
virtual void metoda(int aa) {a=aa;} // 1
virtual void wypisz( ) const // 2
        {cout<<"\n( "<<a<<" )\n"; }
}
```

```
class Y : public X {
 protected:
 int b;
 public:
 Y(int x,int y):X{x},b{y} { cout<<a<<" "<<b<<" konstruktor Y\n";}
 ~Y( ) { cout<<a<<" "<<b<<" destruktor Y\n";}
 void metoda(int x) override {b=b+x;} // 3
 void wypisz( ) const {X::wypisz(); cout<<b<<endl; // 4
 };

 int main( ) {
 X *wsk1=new X{23};
 X *wsk2=new Y{2,4);
 wsk1->wypisz( );
 wsk2->wypisz( );
 wsk1->metoda(9);
 wsk2->metoda(12);
 wsk1->wypisz( );
 wsk2->wypisz( );
 delete wsk1; delete
 wsk2; return 0;
 }
```

Pytania:

* Czy w linii oznaczonej liczbą **1** można dodać słowo kluczowe **final**? Odpowiedź uzasadnij.

Czy w linii oznaczonej liczbą 1 można dodać słowo kluczowe final?

* Nie, nie można dodać słowa kluczowego `final` w linii oznaczonej liczbą 1. Słowo kluczowe `final` stosuje się do wirtualnych funkcji w klasie bazowej, oznaczając, że nie mogą one być przesłaniane w klasach pochodnych. Jednak w przypadku deklaracji funkcji czysto wirtualnej (oznaczonej `= 0`), `final` jest już implicit, co oznacza, że nie ma potrzeby jawnego dodawania go.

* Napisz, jaki będzie wynik pracy powyższego programu w przypadku, gdy w linii oznaczonej liczbą **2** zostanie pominięte słowo kluczowe **virtual**?

Napisz, jaki będzie wynik pracy powyższego programu w przypadku, gdy w linii oznaczonej liczbą 2 zostanie pominięte słowo kluczowe virtual?

* Jeśli pominiemy słowo kluczowe `virtual` w linii oznaczonej liczbą 2, to funkcja `wypisz` nie będzie uznana za funkcję wirtualną. W rezultacie, dla wskaźnika `wsk2` typu `X*` wywołanie `wsk2->wypisz();` nie skorzysta z polimorfizmu, co oznacza, że będzie używana implementacja funkcji z klasy `X`, a nie klasy `Y`. Efekt będzie taki, jakby funkcja `wypisz` nie była przesłonięta w klasie `Y`.

* Napisz, jaki będzie wynik pracy powyższego programu w przypadku, gdy w linii oznaczonej liczbą **2** zostanie pominięte słowo kluczowe **const**?

Napisz, jaki będzie wynik pracy powyższego programu w przypadku, gdy w linii oznaczonej liczbą 2 zostanie pominięte słowo kluczowe const?

* Jeśli pominiemy słowo kluczowe `const` w linii oznaczonej liczbą 2, to program nadal będzie działać poprawnie, ale będzie to traktowane jako inna sytuacja. Jeśli nie zadeklarujemy funkcji `wypisz` jako `const` w klasie `X`, to nadal będzie działać polimorfizm, ale dostęp do tej funkcji dla obiektu `wsk2` będzie niemożliwy, ponieważ `wsk2` jest wskaźnikiem typu `X*`, a nie `const X*`.

* Czy w linii oznaczonej liczbą **3** można pominąć słowo kluczowe **override**? Odpowiedź uzasadnij.

Czy w linii oznaczonej liczbą 3 można pominąć słowo kluczowe override?

* Tak, w linii oznaczonej liczbą 3 można pominąć słowo kluczowe `override`, ale zaleca się jego użycie. Słowo kluczowe `override` służy do jawnej oznaczenia, że dana funkcja ma zamiar przesłaniać funkcję wirtualną z klasy bazowej. Pominięcie tego słowa nie spowoduje błędu, ale może prowadzić do błędów w przyszłości, gdyby sygnatura funkcji w klasie pochodnej różniła się od sygnatury funkcji w klasie bazowej.

* Czy w linii oznaczonej liczbą **4** można pominąć słowo kluczowe **const**? Odpowiedź uzasadnij.

Czy w linii oznaczonej liczbą 4 można pominąć słowo kluczowe const?

- Nie, w linii oznaczonej liczbą 4 nie można pominąć słowa kluczowego `const`. Jest to część deklaracji funkcji w klasie `X`, a później przesłonięta w klasie `Y`. Omitowanie `const` w klasie `Y` spowoduje błąd kompilacji, ponieważ przesłonięcie funkcji musi mieć taką samą sygnaturę, co obejmuje także specyfikator `const` dla funkcji `wypisz`.

Program 6

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

class X
{
protected:
int a;
public:
X(int x=8):a{-x} {cout<<"konstruktor X\n";} // 1
virtual ~X() {cout<<"destruktor X\n";} // 2
virtual void wypisz() const {cout<<a<<"\n";}
 };

class Y:public X
{
protected:
int b;
public:
Y(int x, int y):X{x}, b{y} {cout<<"konstruktor Y\n";}
~Y() {cout<<"destruktor Y\n";}
void wypisz() const {X::wypisz();cout<<"-> "<<b<<"\n";}
};

class YY: public X {}; //3
```

```
int main()
{
X *a=new X{-23};
X *b=new Y{2, 4};
X *c=new YY;
a->wypisz();
b->wypisz();
c-    >wypis
z(); delete
a; delete b;
delete c;
return 0;
}
```

Pytania:

* Czy w definicji konstruktora klasy X w linii **1** można pominąć wartość domyślną argumentu? Odpowiedź uzasadnij.

Czy w definicji konstruktora klasy X w linii 1 można pominąć wartość domyślną argumentu?

* Tak, wartość domyślną argumentu w linii 1 można pominąć, ale zastosowanie jej jest dość przydatne. W przypadku, gdy konstruktor jest wywoływany bez podania argumentu, wartość domyślna `8` zostanie użyta. Dzięki temu konstruktor może być wywołany zarówno bez argumentu, jak i z argumentem.

* Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii **2** pominiemy słowo kluczowe **virtual**.

Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii 2 pominiemy słowo kluczowe virtual.

* Jeśli pominiemy słowo kluczowe `virtual` w linii 2, to destruktor klasy `Y` nie będzie wirtualny. W rezultacie, podczas usuwania obiektu typu `X*`, wskazującego na obiekt klasy `Y`, destruktor klasy `Y` nie zostanie wywołany. To może prowadzić do wycieków pamięci i błędów w zarządzaniu zasobami.

* Uczyń z metody wypisz metodę czysto wirtualną oraz wprowadź niezbędne zmiany w programie, tak żeby można było program skompilować i uruchomić? Napisz, jaki będzie wynik programu po wprowadzonych zmianach.

Uczyń z metody wypisz metodę czysto wirtualną oraz wprowadź niezbędne zmiany w programie, tak żeby można było program skompilować i uruchomić? Napisz, jaki będzie wynik programu po wprowadzonych zmianach.

* Wprowadźmy metodę `wypisz` jako metodę czysto wirtualną w klasie `X`. Zmiany w kodzie będą następujące:

virtual void wypisz() const = 0; // zmiana w klasie X

Zmiana w programie:

```
int main()
{
   X *a = new X{-23};
   X *b = new Y{2, 4};
   X *c = new YY;

   a->wypisz();
   b->wypisz();
   c->wypisz();

   delete a;
   delete b;
   delete c;

   return 0;
```

}
W wyniku tych zmian, klasa `X` staje się klasą abstrakcyjną, co oznacza, że nie można tworzyć bezpośrednio obiektów tej klasy. Klasa `YY` musi zaimplementować metodę czysto wirtualną `wypisz`, aby stała się ona klasą konkretnej implementacji.

* Czy klasa YY mogłaby być pochodną klasy Y, bez zmiany definicji klasy YY? Odpowiedź uzasadnij.

Czy klasa YY mogłaby być pochodną klasy Y, bez zmiany definicji klasy YY?

- Nie, klasa `YY` nie mogłaby być pochodną klasy `Y` bez zmiany definicji klasy `YY`. Klasa `Y` dziedziczy publicznie po klasie `X`, co oznacza, że wszystkie składowe chronione i publiczne z klasy `X` są dostępne w klasie `Y`. Klasa `YY` dziedziczy również publicznie po klasie `X`, ale nie ma dostępu do składowych chronionych klasy `X`, ponieważ są one oznaczone jako `protected`. Aby klasa `YY` mogła być pochodną klasy `Y`, składowe chronione klasy `X` powinny być dostępne dla klas dziedziczących, co wymagałoby zmiany modyfikatora dostępu w klasie `X`.

Program 7

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

class X {
protected:
int a;
public:
X(int x=2):a{x} {cout<<"konstruktor X\n";}
virtual ~X() {cout<<"destruktor X\n";} //0
virtual void wypisz() const {cout<<a<<"\n";} //1
};

class Y:public X {
protected:
int b;
public:
Y(int x, int y):X{x}, b{y} {cout<<"konstruktor Y\n";}
~Y() {cout<<"destruktor Y\n";}
void wypisz() const {cout<<a<<" "<<b<<"\n";}
};

class YY:public X {};

void wywolaj(const X& arg) //2
 {arg.wypisz(); }

int main()
{
X *a[3]={new X{23}, new Y{2, 4}, new YY}; // 3
for (int i=0; i<3; ++i) wywolaj(*a[i]);
for (int i=0; i<3; ++i) delete a[i];
return 0;
}
```

Pytania:

* Czy w linii **0** można pominąć słowo kluczowe **virtual**? Odpowiedź uzasadnij.

Czy w linii 0 można pominąć słowo kluczowe virtual?

* W linii 0 nie można pominąć słowa kluczowego `virtual`. Bez użycia słowa kluczowego `virtual`, destruktor klasy `X` nie będzie działał polimorficznie. W rezultacie, podczas usuwania obiektów za pomocą wskaźników na klasę bazową `X`, nie zostaną wywołane destruktory klas pochodnych, co może prowadzić do wycieków pamięci.

* Jaki będzie wynik pracy powyższego programu, jeśli w linii **1** pominiemy słowo kluczowe **virtual**?

Jaki będzie wynik pracy powyższego programu, jeśli w linii 1 pominiemy słowo kluczowe virtual?

* Jeśli pominiemy słowo kluczowe `virtual` w linii 1, to destruktor klasy `Y` nie będzie działał polimorficznie. W konsekwencji, podczas usuwania obiektu `Y` za pomocą wskaźnika na klasę bazową `X`, destruktor `Y` nie zostanie wywołany, co może prowadzić do wycieku pamięci.

* Jaki będzie wynik pracy powyższego programu, jeśli w linii **2** pominiemy referencję?

Jaki będzie wynik pracy powyższego programu, jeśli w linii 2 pominiemy referencję?

* Jeśli w linii 2 pominiemy referencję, a zamiast tego użyjemy argumentu typu `X` (tj. `void wywolaj(const X arg)`), to skutkowałoby to przekazaniem obiektu przez wartość. W rezultacie, funkcja `wypisz` byłaby wywoływana dla kopii obiektu, co mogłoby prowadzić do błędów, jeśli przekazywany obiekt jest obiektem klasy pochodnej (np. `Y`). Użycie referencji jest preferowane, aby uniknąć kopiowania obiektu.

* Czy w przypadku abstrakcyjnej klasy X linia **3** będzie poprawna? Odpowiedź uzasadnij.

Czy w przypadku abstrakcyjnej klasy X linia 3 będzie poprawna?

- Nie, linia 3 nie będzie poprawna w przypadku abstrakcyjnej klasy `X`. Wprowadzając obiekt klasy abstrakcyjnej `X` do tablicy wskaźników, możemy utworzyć obiekty klas pochodnych, ale nie można utworzyć bezpośrednio obiektu klasy abstrakcyjnej. Jednakże, można utworzyć obiekty klas pochodnych i przypisać im adresy do wskaźników klasy bazowej. Jednak używając obiektu klasy `X` bezpośrednio w tablicy nie będzie możliwe, ponieważ klasa `X` jest abstrakcyjna (ma czysto wirtualną funkcję).

Program 8

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

class X
{
protected:
int a;
public:
X(int x=2):a{x} {cout<<"#";}
virtual ~X() {cout<<"destruktor X\n";}
virtual void wypisz() const {cout<<a<<"\n";} //1
};
class Y: public X //2
{
protected:
int b;
public:
Y(int x, int y):X{x}, b{y} {cout<<"$\n";}
~Y() {wypisz();cout<<"destruktor Y\n";}
void wypisz() const {cout<<a<<" "<<b<<"\n";}
virtual void zmiany(int x, int y) {a+=x; b=y;} //3
};

class YY: public Y
{
int c;
public:
YY(int x):Y{x,-x}, c{2*x} {cout<<"%\n";}
void zmiany(int x, int y) {a=y; b=x; c=x*y;}
void wypisz() const {cout<<a<<" "<<b<<" "<<c<<"\n";}
};

void wywolaj(X& arg) //4
{
cout<<"\n***\n";
arg.wypisz();
```

```
if (Y *wsk=dynamic_cast<Y*>(&arg))
    {
    wsk->zmiany(3,2);
    wsk->wypisz();
    }
}

int main()
{
X *a[3]={new X{23}, new Y{2, 7}, new YY{6}};
for (int i=0; i<3; ++i) wywolaj(*a[i]);
for (int i=0; i<3; ++i) delete a[i];
return 0;
}
```

Pytania:

* Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii **1** pominiemy słowo kluczowe **virtual**.

Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii 1 pominiemy słowo kluczowe `virtual`.

* Jeśli w linii 1 pominiemy słowo kluczowe `virtual`, destruktor klasy `X` nie będzie działał polimorficznie. W rezultacie, podczas usuwania obiektów za pomocą wskaźników na klasę bazową `X`, destruktory klas pochodnych (`Y` i `YY`) nie zostaną wywołane, co może prowadzić do wycieku pamięci.

* Czy w linii 2 można dziedziczenie publiczne zastąpić dziedziczeniem prywatnym (ew. chronionym)? Odpowiedź uzasadnij.

Czy w linii 2 można dziedziczenie publiczne zastąpić dziedziczeniem prywatnym (ew. chronionym)? Odpowiedź uzasadnij.

* Nie, w linii 2 dziedziczenie musi pozostać publiczne. W przeciwnym razie, jeżeli zastąpimy dziedziczenie publiczne dziedziczeniem prywatnym lub chronionym, to nie będzie możliwe użycie obiektów klas pochodnych, gdy wskaźnik do klasy bazowej jest używany. W przypadku dziedziczenia prywatnego lub chronionego, obiekty klas pochodnych nie będą dostępne publicznie poprzez wskaźnik do klasy bazowej, co ograniczy funkcjonalność polimorfizmu.

* Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii **3** pominiemy słowo kluczowe **virtual**.

Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii 3 pominiemy słowo kluczowe `virtual`.

* Jeśli w linii 3 pominiemy słowo kluczowe `virtual`, to funkcja `zmiany` w klasie `Y` nie będzie przesłaniana (będzie cicha wymiana funkcji, a nie przesłanianie). W rezultacie, wywołanie `wywolaj(*a[i])` dla obiektu klasy `YY` nie spowoduje wywołania funkcji `zmiany` z klasy `YY`, co prowadzi do utraty części funkcjonalności polimorfizmu.

* Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii **4** pominiemy referencję.

Napisz, jaki będzie wynik pracy powyższego programu, jeśli w linii 4 pominiemy referencję.

* Jeśli w linii 4 pominiemy referencję i użyjemy zwykłego argumentu (np. `void wywolaj(X arg)`), to funkcja `wywolaj` będzie operować na kopii obiektu przekazanego do niej. W takim przypadku, polimorfizm nie zadziała poprawnie, a funkcja `zmiany` z klasy `Y` nie zostanie wywołana dla obiektu klasy `YY` podczas wywołania `wywolaj(*a[i])`.

* Czy w powyższej hierarchii dziedziczenia można zrezygnować z wirtualności metod? Odpowiedź uzasadnij.

Czy w powyższej hierarchii dziedziczenia można zrezygnować z wirtualności metod? Odpowiedź uzasadnij.

- Nie, wirtualność metod jest istotna w tej hierarchii dziedziczenia ze względu na wykorzystanie polimorfizmu. Dzięki deklaracji funkcji `wypisz` jako wirtualnej w klasie `X`, możliwe jest dynamiczne wiązanie funkcji podczas działania programu, co pozwala na przesłanianie tej funkcji w klasach pochodnych (`Y` i `YY`). Jeżeli zrezygnowano by z wirtualności, to nie można by

było korzystać z polimorfizmu, a wywołanie funkcji `wypisz` dla obiektów klas pochodnych nie działałoby poprawnie.

Zadanie 3 (dst, db, bdb) Napisz

Jakie są różnice (ew. podobieństwa) między?

1. Klasą bazową a klasą, która jest jej klasą pochodną.

Klasa bazowa vs. Klasa pochodna:

- Klasa bazowa: Jest to klasa nadrzędna lub podstawowa, która zawiera wspólne właściwości i metody dla klas pochodnych.
- Klasa pochodna: Jest to klasa, która dziedziczy cechy (właściwości i metody) z klasy bazowej, a także może mieć swoje własne unikalne cechy.

2. Obiektem klasy bazowej a obiektem reprezentującym jej klasę pochodną.

Obiekt klasy bazowej vs. Obiekt klasy pochodnej:

- Obiekt klasy bazowej: Instancja klasy bazowej.
- Obiekt klasy pochodnej: Instancja klasy pochodnej, która dziedziczy cechy klasy bazowej, ale może mieć również swoje własne cechy.

3. Etykietą private a protected.

Private vs. Protected:

- Private: Składowe oznaczone jako prywatne są dostępne tylko w obrębie tej samej klasy.
- Protected: Składowe oznaczone jako chronione są dostępne w obrębie tej samej klasy i klas pochodnych.

4. Dziedziczeniem publicznym a dziedziczeniem prywatnym.

Dziedziczenie publiczne vs. Dziedziczenie prywatne:

- Dziedziczenie publiczne: Publiczne składowe klasy bazowej pozostają publiczne w klasie pochodnej.
- Dziedziczenie prywatne: Publiczne składowe klasy bazowej stają się prywatne w klasie pochodnej.

5. Dziedziczeniem publicznym a dziedziczeniem chronionym.

Dziedziczenie publiczne vs. Dziedziczenie chronione:

- Dziedziczenie publiczne: Publiczne i chronione składowe klasy bazowej pozostają publiczne i chronione w klasie pochodnej.
- Dziedziczenie chronione: Publiczne składowe klasy bazowej stają się chronione w klasie pochodnej.

6. Dziedziczeniem chronionym a dziedziczeniem prywatnym.

Dziedziczenie chronione vs. Dziedziczenie prywatne:

- Dziedziczenie chronione: Chronione składowe klasy bazowej pozostają chronione w klasie pochodnej.
- Dziedziczenie prywatne: Publiczne i chronione składowe klasy bazowej stają się prywatne w klasie pochodnej.

7. Metodą klasy bazowej, a metodą, która jest składową klasy pochodnej.

Metoda klasy bazowej vs. Metoda klasy pochodnej:

- Metoda klasy bazowej: Metoda zdefiniowana w klasie bazowej.
- Metoda klasy pochodnej: Metoda przesłonięta lub dodana w klasie pochodnej.

8. Konstruktorem klasy bazowej a konstruktorem jej klasy pochodnej.

Konstruktor klasy bazowej vs. Konstruktor klasy pochodnej:

- Konstruktor klasy bazowej: Inicjalizuje obiekt klasy bazowej.
- Konstruktor klasy pochodnej: Inicjalizuje obiekt klasy pochodnej, może korzystać z konstruktora klasy bazowej.

9. Wywołaniem metody za pomocą adresu a wywołaniem metody za pomocą wartości.

Wywołanie metody za pomocą adresu vs. Wywołanie metody za pomocą wartości:

- Wywołanie metody za pomocą adresu: Wywołanie metody poprzez wskaźnik do obiektu.
- Wywołanie metody za pomocą wartości: Bezpośrednie wywołanie metody na obiekcie.

10. Metodą czysto wirtualną a metodą wirtualną.

Metoda czysto wirtualna vs. Metoda wirtualna:

- Metoda czysto wirtualna: Funkcja wirtualna bez implementacji w klasie bazowej, wymaga implementacji w klasach pochodnych.
- Metoda wirtualna: Funkcja wirtualna z domyślną implementacją, której można dostarczyć alternatywną implementację w klasie pochodnej.

11. Destruktorem wirtualnym a destruktorem czysto wirtualnym.

Destruktor wirtualny vs. Destruktor czysto wirtualny:

- Destruktor wirtualny: Zwalnia zasoby i jest wywoływany przy usuwaniu obiektu.
- Destruktor czysto wirtualny: Destruktor wirtualny z funkcją czysto wirtualną, bez domyślnej implementacji.

12. Klasą abstrakcyjną a klasą, która nie jest abstrakcyjna.

Klasa abstrakcyjna vs. Klasa nieabstrakcyjna:

- Klasa abstrakcyjna: Klasa, która zawiera co najmniej jedną czysto wirtualną funkcję i nie może być instancjonowana.
- Klasa nieabstrakcyjna: Klasa, która może być instancjonowana i niekoniecznie posiada czysto wirtualne funkcje.

13. Wirtualną klasą bazową a zwykłą klasą bazową.

Wirtualna klasa bazowa vs. Zwykła klasa bazowa:

- Wirtualna klasa bazowa: Klasa używana w dziedziczeniu diamentowym, gdzie jest tylko jedna instancja wspólna dla obiektów klas pochodnych.
- Zwykła klasa bazowa: Klasa, z której dziedziczą inne klasy, bez uwzględnienia problemów dziedziczenia diamentowego.

14. Konwersją w górę hierarchii dziedziczenia a konwersją w dół hierarchii dziedziczenia.

Konwersja w górę hierarchii dziedziczenia vs. Konwersja w dół hierarchii dziedziczenia:

- Konwersja w górę: Konwersja wskaźnika lub referencji do obiektu klasy bazowej.
- Konwersja w dół: Konwersja wskaźnika lub referencji do obiektu klasy pochodnej.

15. Słowem kluczowym **override** a słowem kluczowym **final**

Słowo kluczowe override vs. Słowo kluczowe final:

- Override: Wskazuje, że metoda w klasie pochodnej przesłania identyczną metodę z klasy bazowej.
- Final: Uniemożliwia dziedziczenie lub przesłanianie metody lub klasy.

Zadanie 2

Fragment 1 (dst) Do klasy

```
class Osoba
{
string nazwisko;
int wiek;
};
```

dodaj konstruktor, destruktor, metodę zmieniającą wartość pola wiek, oraz metodę wypisującą informacje na temat osoby. Na podstawie klasy Osoba stwórz klasę Pracownik z dodatkowym polem pensja, dodaj do tej klasy konstruktor, destruktor, metodę zmieniającą wartość pola pensja, oraz metodę wypisującą informacje na temat pracownika. Napisz program, w którym przetestujesz wszystkie składowe zdefiniowanej hierarchii dziedziczenia klas.

Fragment 2 Dodaj do hierarchii dziedziczenia z klasą bazową Osoba (Fragment 1) wirtualność. Przeciąż na rzecz klasy Osoba operator <<. Napisz program, w którym przetestujesz dodane funkcje/metody.