Polymorphism

מאפיינים של polymorphism

- המאפשר כתיבת תוכנה ברמות הפשטה שונות.
- פולימורפיזם מהווה תכונה חשובה במתודולוגית תכנות מונחה עצמים. המנגנון נותן למתכנת את היכולת לכתוב אלגוריתמים ומבני נתונים לשימוש כללי, ולגזור מהם צורות שימוש שונות בהתאם לעצמים ולנסיבות המשתנות.
- אפשרות שלמשתנה מסוים יהיו צורות שונות והתוכנית יכולה לקבוע באיזה מהצורות של המשתנה יעשה שימוש במהלך הריצה.
- פונקציה ספציפית יכולה להשתנות בהתאם לפרמטרים שנותנים לה.

מנגנוני ה-polymorphism

- **דריסה** ((Override)שבה ניתן להחליף פונקציה של מחלקת בסיס בפונקציה שונה במחלקה הנגזרת ממנה.
טכניקה נוספת היא **הצללה** שדומה ליכולת הדריסה.
באמצעות מנגנון זה ניתן להתאים את הפעולה לסוג המיוחד של האובייקט. שם הפעולה נשמר והיא מתבצעת בדרכים שונות בהתאם לאובייקט אליו היא מתייחסת.

- **העמסת פרמטרים**, שבו ניתן ליצור כמה פונקציות בעלות שם זהה אך בעלות פרמטרים שונים. מנגנון זה שימושי לריכוז כל הפונקציות תחת אותה קורת גג, במקום לכתוב פונקציות רבות בעלות שם שונה שמבצעות למעשה את אותה פעולה מבחינה לוגית.

- **העמסת אופרטורים**, שבו אופרטור מסוים שומר על צורתו הלוגית, אך מקבל מכניזם פנימי שונה בתוך אובייקט. למשל אופרטור + שנועד לחבר מספרים, במחלקה של מחרוזות מקבל יכולת שרשור, ובמחלקה של מערכים יכול לקבל יכולת לחבר בין שני מערכים.

Virtual functions

- By default, C++ matches a function call with the correct function definition at compile time.
- This is called static binding.
- You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding.
- You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

Default : no virtual


```
#include <iostream>
using namespace std;
class A {
void f() {  cout << "Class A" << endl;  }
};
class B: A {
void f() {  cout << "Class B" << endl;  }
 };
void g(A& arg) { arg.f(); }
int main() {
B x;
g(x);
}
```

The following is the output of the above example:

Class A

- **When function g() is called, function A::f() is called, although the argument refers to an object of type B.**
- **At compile time, the compiler knows only that the argument of function g() will be a reference to an object derived from A; it cannot determine whether the argument will be a reference to an object of type A or type B.**
- **However, this can be determined at run time.**

Virtual definition


```
#include <iostream>
using namespace std;
class A {
   virtual void f() {   cout << "Class A" << en
};
class B: A {
   void f() {  cout << "Class B" << endl;  }
};
void g(A& arg) { arg.f(); }
int main() {
   B x;
   g(x);
}
```

The following is the output of the above example:

Class B

- **This example is the same as the previous example, except that A::f() is declared with the virtual keyword.**
- **The virtual keyword indicates to the compiler that it should choose the appropriate definition of f() not by the type of reference, but by the type of object that the reference refers to.**

Polymorphism ?

- A class that declares or inherits a virtual function is called a *polymorphic class*.

Overloading virtual functions


```
class A {
   virtual void f() { cout << "void A::f()" << endl; }
   virtual void f(int) { cout << "void A::f(int)" << endl; }
 };
class B : A {
   using A::f;
   void f(int) { cout << "void B::f(int)" << endl; }
};
int main() {
   B obj_B;
   B* pb = &obj_B;
   pb->f(3);
   pb->f();
}
```

```
void B::f(int)
void A::f()
```

Example 1


```
class A {
    virtual void f() { cout << "Class A" << endl; }  };
class B: A {
    void f(int) { cout << "Class B" << endl; }   };
class C: B {
    void f() {
    cout << "Class C" << endl; }   };

int main() {
B b;
C c;
A* pa1 = &b;
A* pa2 = &c;
// b.f();
pa1->f();
pa2->f();
}
```

Class A
Class C

- The function B::f is not virtual. It hides A::f.
- Thus the compiler will not allow the function call b.f().
- The function C::f is virtual; it overrides A::f even though A::f is not visible in C.

Example 2


```
class CPolygon {
   protected:
         int width, height;
   public:
         void set_values (int a, int b) {
                  width=a; height=b; } };
class CRectangle: public CPolygon {
   public:
         int area () { return (width * height); } };
class CTriangle: public CPolygon {
   public:
         int area () { return (width * height / 2); } };
```

```
int main () {
   CRectangle rect;
   CTriangle trgl;
   CPolygon * ppoly1 = &rect;
   CPolygon * ppoly2 = &trgl;
   ppoly1->set_values (4,5);
   ppoly2->set_values (4,5);
   cout << rect.area() << endl; cout << trgl.area() << endl;
   return 0;
}
```

```
20
10
```

Example 3


```
class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) {
            width=a; height=b; }
    virtual int area () { return (0); }
};
class CRectangle: public CPolygon {
public:
    int area () {  return (width * height); }
};
class CTriangle: public CPolygon {
public: int area () { return (width * height / 2); }
};
```

Example 3 -A


```
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
    return 0;
}
```

```
20
10
 0
```

Example 2 - B


```
int main () {
   CPolygon * ppoly1 = new CRectangle;
   CPolygon * ppoly2 = new CTriangle;
   ppoly1->set_values (4,5);
   ppoly2->set_values (4,5);
   ppoly1->printarea();
   ppoly2->printarea();
   delete ppoly1;
   delete ppoly2;
   return 0;
}
```

```
20
10
```

Notice that the ppoly pointers:
CPolygon * ppoly1 = *new* CRectangle;
CPolygon * ppoly2 = *new* CTriangle;

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.

Example 4


```
class Car {...};
class Bus: public Car{...};
class PrivateCar: public Car{...};
void changeWheel(Car *p) {
    p->stop();
    p->placeJack();
    p->lift(); ... }
int main() {
    Car* array[3];
    array[0]=new Bus();
    array[1]=new PrivateCar();
    array[2]=new Bus();
    for(int i=0;i<3; ++i) changeWhe
}
```

הפונקציה מקבלת מצביע למכונית כללית. למעשה מדובר בסוג מסוים של מכונית, מכונית ספציפית, אבל changeWheel מתייחסת אליה פשוט כמכונית ומבצעת עליה רק פעולות שאפשר לעשות על כל מכונית. כאשר changeWheel תבקש ממכונית לעשות פעולה מסוימת, המכונית תבצע את הפעולה באופן ייחודי לה