Polymorphism

הקדמה

בשיעור זה נכיר כלי רעיון (או מנגנון) מרכזי בתכנות מונחה עצמים: רב צורתיות או Polymorphism. שימוש בו יאפשר לנו לכתוב אלגוריתמים ומבני נתונים כלליים. המנגנון יאפשר לנו ביטוי במושגים מופשטים וכך לכתוב קוד המתאים להרבה מיקרים פרטיים.

התייחסות למתכנת כאל אדם

נתבונן בדוגמה שראינו בשעור הקודם. בדוגמה, למחלקה היורשת יש הגדרה מחודשת של אחת מהפונקציות המוגדרות במחלקה המורישה:

```
class Person {
…
        void outputDetails(std::ostream& os=std::cout) const;
…
};
void Person::outputDetails(std::ostream& os) const {
        os << "I'm just a person\n";
}

class Programmer : public Person {
…
        void outputDetails(std::ostream& os=std::cout) const;
…
};
void Programmer::outputDetails(std::ostream& os) const {
        os << "I'm a programmer!!\n";
}
```

כעת, נתייחס לאובייקט של מתכנת בעזרת מצביע לאדם:

```
int main() {
        Programmer yoram("Yoram",1226611,"N.G.C ltd.");
        Person* p;
        p = &yoram;
}
```

לכאורה, היינו מצפים שקוד זה לא יהיה חוקי מכיוון שקיימת אי התאמה בין הטיפוסים בהשמה. הטיפוס של &yoram הוא *Programmer ואילו הטיפוס של p הוא *Person ולכן נראה שהקוד של ההשמה היה אמור להכתב כך:

```
        p = (Person*)&yoram;
```

למרות שההשמה האחרונה מהווה קוד חוקי, גם צורת ההשמה הראשונה היא תקינה. מכיוון שמתכנת הוגדר להיות סוג של אדם, השפה מאפשרת לנו להתייחס למתכנת כאל אדם, ללא צורך לבצע המרת טיפוסים.

כעת נפעיל את הפונקציה outputDetails() בעזרת המצביע:

```
int main() {
        Programmer yoram("Yoram",1226611,"N.G.C ltd.");
        Person* p = &yoram;
        p->outputDetails();
}
```

הפלט יהיה: "I'm just a person\n". הקוד שיפעל אם כן הוא הקוד של המחלקה Person (בהתאם לטיפוס המצביע), למרות שהאובייקט הנימצע בזיכרון הוא מסוג Programmer.

פונקציות וירטואליות

המחלקה Person מהווה מכנה משותף לכל סוגי האנשים המיוצגים ע"י מחלקות. לכל סוג של אדם תהיה פונקצית הדפסת פרטים. לחלק מסוגי האנשים תהיה פונקציה ייחודית וחלק פשוט יירשו את הפונקציה של המכנה המשותף. יוצא אם כן, שהמכנה המשותף לכל האנשים הוא קיום

פונקצית הדפסת הפרטים אבל לא המימוש שלה. **המשותף** במקרה זה **הוא הממשק ולא האמפלמנתציה**.

נניח כעת קיום של מחלקה נוספת המייצגת סטודנט:

```
class Student : public Person {
…
        void outputDetails(std::ostream& os=std::cout) const;
…
};
void Student::outputDetails(std::ostream& os) const {
        os << "I'm a poor student\n";
}
```

וכעת נכתוב:

```
int main() {
        Programmer yoram("Yoram",1226611,"N.G.C ltd.");
        Student moni;
        Person *p1 = &yoram, *p2 = &moni;
        p1->outputDetails(); // "I'm just a person\n"
        p2->outputDetails(); //"I'm just a person\n"
}
```

שוב, נקבל הדפסות של אנשים כלליים.
למרות שהטיפוס של p1,p2 הוא *Person, בזמן הריצה הם יצביעו לאובייקטים מסוג Student ו-Programmer. האובייקטים שבזכרון מכילים את כל האינפורמציה היחודית לכל מחלקה ולפיכך אין מניעה עקרונית שההפעלה של ההדפסה מתוך המצביעים תפעיל את הפונקציות היחודיות לכל מחלקה. בכדי שדבר זה באמת יקרה, אנו צריכים להכריז על outputDetails **כפונקציה וירטואלית** של המחלקה Person:

```
class Person {
…
        virtual void outputDetails(std::ostream& os=std::cout) const;
…
};
```

הוספת מילה שמורה יחידה זו, תשנה את התנהגות התוכנית שלנו:

```
int main() {
        Programmer yoram("Yoram",1226611,"N.G.C ltd.");
        Student moni;
        Person *p1 = &yoram, *p2 = &moni;
        p1->outputDetails(); //"I'm a programmer!!\n"
        p2->outputDetails(); //"I'm a poor student\n"
}
```

כעת, פונקציות ההדפסה שיופעלו הן הפונקציות היחודיות לכל סוג של אדם.
אם נוסיף מחלקה חדשה היורשת מ - Person ולא נגדיר עבורה פונקצית הדפסה, היא תירש את הפונקציה הווירטואלית. לדוגמה:

```
class Actor : public Person {};

int main() {
        Student moni;
        Actor yosi;
        Person *p1 = &moni, *p2 = &yosi;
        p1->outputDetails();//"I'm a poor student\n"
        p2->outputDetails();"I'm just a person\n"
        yosi.outputDetails();"I'm just a person\n"
}
```

כפי שראינו, בעזרת פונקציות וירטואליות ניתן להפעיל דרך מצביע ל – Base פונקציות של ה – Derived. למרות העובדה הזאת, אוסף הפונקציות של ה – Derived שנוכל להפעיל מוגבל לפונקציות המוצהרות ב – Base. לדוגמה:

```
int main() {
        Programmer yoram;
        Person* p = &yoram;
        P->getCompany(); // c.error : getCompany is not a member function of Person
}
```

כעת ניתן ליכתוב אלגוריתם כללי העובד עם בני אדם. על כל סוג של אדם, האלגוריתם יפעל אחרת:

```
void algorithm(Person* p) {
        p->outputDetails();
        …
}
int main() {
        Programmer yoram;
        Student moni;
        algorithm(&yoram);
        algorithm(&moni);
}
```

יכולנו גם לקבל את הפרמטר בפונקציה – by refernce:

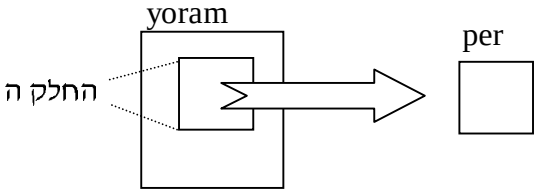
```
void algorithm(Person& person) {
        person.outputDetails();
        …
}
int main() {
        Programmer yoram;
        algorithm(yoram);
        …
}
```

הדבר אפשרי מכיוון שהעברה by-reference, מבוצעת למעשה ע"י שליחה של מצביע.

אם היינו מקבלים את הפרמטר – by value היינו מאבדים את ה – polymorphism:

```
void algorithm(Person per) {
        per.outputDetails();
        …
}
int main() {
        Programmer yoram;
        algorithm(yoram);
        …
}
```

מכיוון ש – person הוא אובייקט מסוג Person ואיננו מצביע או reference ל – Programmer, אין בזיכרון את האינפורמציה הדרושה להפעלת הפונקציות היחודיות ל – Programmer. ה – sizeof של per יהיה sizeof(Person) והוא יהווה בעצם העתק של החלק ה – Person-י של המתכנת שנישלח:



דוגמה "המחיים"

בשביל להבין טוב יותר את הטעם ב – polymorphism ננסה לחשוב על אלגוריתם להחלפת גלגל במכונית:

1. הוצאי את ה – ג'ק
2. הוצאי את הגלגל הרזרבי
3. הגביהי את הרכב בעזרת הג'ק
4. פרקי את הגלגל המורם
5. הרכבי את הגלגל הרזרבי
6. הורדי את הרכב מה – ג'ק
7. החזרי את הג'ק למקומו
8. החזרי את הגלגל שהחלפת למקום הגלגל הרזרבי

האלגוריתם הזה נכון לכל סוגי הרכב, אפילו למשאיות ואוטובוסים. עבור כל מקרה פרטי של רכב, הפעלה של האלגוריתם הלכה למעשה, תגרום לפעילות שונה. ה – ג'ק של משאית ממוקם ומופעל בצורה שונה מזו של מכונית פרטית. הפעולה של הוצעת הג'ק והרמת הרכב, יהיו אם כן פעולות שונות עבור סוגי רכב שונים. זהו בעצם הרעיון המרכזי: אפשרות לכתוב קוד אבסטרקטי שיהיה נכון להרבה מיקרים פרטיים אבל יתבטא בצורה שונה . מכאן גם השם: רב צורתיות – כל פעם הקוד מקבל צורה אחרת כאשר הוא מופעל על מיקרה פרטי חדש. את הדוגמה המילולית שהבאנו כאן ניתן בקלות להפוך לקוד. לכל סוג של מכונית תהיה מחלקה שתייצג אותה, וכל המחלקות יירשו ממחלקה של מכונית. האלגוריתם של החלפת הגלגל יראה כך:

```
void changeWheel(Car& aCar) {
        aCar.takeJackOut();
        aCar.getExtraWheelOut();
        aCar.liftCar();
        aCar.decomposeWheel();
        aCar.installExtraWheel();
        aCar.decreaseLift();
        aCar.returnJackToPlace();
        aCar.returnWheelToPlace()
}
```

ה – main יכול להראות כך:

```
int main() {
    DeffTrack aTrack;
    SchoolBus schoolBus;
    ToyotaCorola gorget;

    changeWheel(aTrack);
    changeWheel(schoolBus);
    changeWheel(gorget);
}
```

לכל מכונית ימומשו הפונקציות הוירטואליות בצורה אחרת. לדוגמה:

```
void DeffTrack::takeJackOut() {
    … implementation of taking out the jack of a 'Deff' track ..
}
void ToyotaCorola::takeJackOut()  {
    … implementation of taking out the jack of a Corola ..
}
```

מבנה נתונים כללי

עד כה בנינו מבני נתונים שונים המתאימים לאכסון טיפוסים מסוימים. כתבנו עץ בינארי המחזיק int-ים, רשימה משורשרת המחזיקה double-ים וכ׳. אם רצינו רשימה משורשרת של double-ים או של Points היינו צריכים לכתוב קוד חדש עבור כל סוג של רשימה.

בעזרת ה - plolymorphism ניתן לכתוב מבנה נתונים שיתאים לטיפוסים שונים. הדוגמה הפשוטה ביותר היא מבנה נתונים של מערך. הקוד הבאה מדגים שימוש במערך של סוגים שונים של אנשים:

```
int main() {
	Person* array[9];
	for(int i=0; i<9; i+=3) {
		array[i] = new Programmer("yoram",i,"e.c.i");
		array[i+1] = new Student("moni",i);// assuming a proper constructor
		array[i+2] = new Actor("yosi",i);
	}
	for(int j=0; j<9; j++)
		array[j]->outputDetails();
	// memory cleanup …
}

output:
I’m a programmer !!
I'm a poor student
I’m just a person
I’m a programmer !!
I'm a poor student
I’m just a person
I’m a programmer !!
I'm a poor student
I’m just a person
```

מכיוון שמדובר במערך של מצביעים לאנשים ולא לאנשים ספציפיים, ניתן להחזיק בו סוגים שונים של אנשים. הפעולות שאותן ניתן לבצע על האנשים במבנה הנתונים, מוגבלות לאותן פעולות המשותפות לכל האנשים (אך כאמור, יכולות להיות ממומשות בצורה שונה בכל סוג).

כעת נתבונן בדוגמה מעט מורכבת יותר. נבנה מבנה נתונים של עץ בינארי ממוין של אנשים. המיון יעשה לפי מספר תעודת הזהות של האנשים:

```
class Node {
public:
	Person* _dataPtr;
	Node *_ls,*_rs;
	Node(Person* dataPtr, Node* ls = NULL, Node* rs = NULL);
};
Node::Node(Person* dataPtr, Node* ls, Node* rs):
	_dataPtr(dataPtr),_ls(ls),_rs(rs) {} // using the same person (no duplication)

class Tree {
	friend ostream& operator <<(ostream& os, const Tree& tree);
	Node* _root;
	static void insert(Person& person, Node** p);
	static ostream& print(ostream& os, Node* p);
public:
	Tree();
	void insert(Person& person);
```

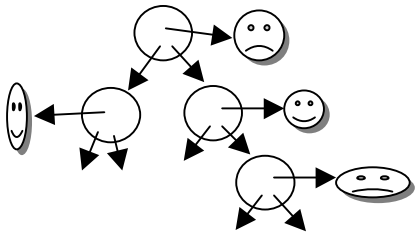
```
};

Tree::Tree() : _root(NULL) {}

ostream& Tree::print(ostream& os, Node* p) {
	if(!p)
		return os<<"null";
	os << '<';
	print(os,p->_ls)<<',';
	p->_dataPtr->outputDetails(os);
	os << ',';
	print(os,p->_rs);
	return os<<'>';
}

ostream& operator <<(ostream& os, const Tree& tree) {
	return tree.print(os,tree._root);
}

void Tree::insert(Person& person) {
	insert(person,&_root);
}

void Tree::insert(Person& person, Node** p) {
	if(!*p) {
		*p = new Node(&person);
		return;
	}
	if(person.getId() < (*p)->_dataPtr->getId())
		insert(person,&((*p)->_ls));
	else
		insert(person,&((*p)->_rs));
}
// dtor ..

int main() {
	Person gil("Gill",5);
	Programmer yori("Yoram",8);
	Student moni("mon",1);
	Actor yosi("Yos",2);
	Student dan("Dan",7);
	Tree tree;
	tree.insert(gil);
	tree.insert(yori);
	tree.insert(moni);
	tree.insert(yosi);
	tree.insert(dan);
	std::cout << tree << std::endl ;

}
```

פלט:

<<null,{stud 1},<null,{per 2},null>>,{per 5},<<null,{stud 7},null>,{prog 8},null>>

(זאת בהנחה ששיננו את outputDetails של המחלקות לפורמט המודפס כאן)

בזמן ריצת התוכנית, מבנה הנתונים שבנינו יראה כך:



שימו לב שהאובייקטים של האנשים לא שוכפלו ע"י מבנה הנתונים.
אם נוסיף לדוגמה, את השורה:

```
yori.changeId(17000);
```

לפני שורת ההדפסה:

```
std::cout << tree << std::endl ;
```

ניראה את השינוי של תעודת הזהות גם בתוך מבנה הנתונים:

<<null,{stud 1},<null,{per 2},null>>,{per 5},<<null,{stud 7},null>,{prog 17000},null>>

אם נרצה שמבנה הנתונים יחזיק שכפולים של האנשים המוכנסים אליו, נצטרך להוסיף קוד הדואג לכך. מכיוון ששכפול כל סוג אדם, נעשה בצורה שונה, יש צורך להפעיל את מנגנון ה – polymorphism גם לשם השכפול. נוסיף פונקציה וירטואלית בשם clone המחזירה העתק של אדם ונשתמש בה כך:

```
Node::Node(Person* dataPtr, Node* ls, Node* rs):
        _dataPtr(dataPtr->clone()),_ls(ls),_rs(rs) {}
```

נצהיר על הפונקציה באופן הבא:

```
class Person {
…
public:
virtual Person* clone() const;
…
};
```

ונממש אותה:

```
Person* Person::clone() const {
        return new Person(_name,_id);
}
```

ניתן גם לממש את הפונקציה בעזרת הפעלה של copy-ctor:

```
Person* Person::clone() const {
        return new Person(*this);
}
```

כעת יש לממש את הפונקציה עבור כל סוגי האנשים:

```
class Programmer: public Person {
…
public:
        virtual Person* clone() const; // the same as: Person* clone() const;
….
```

```
};
Person* Programmer::clone() const {
        return new Programmer(_name,_id,_company);
}
```

או

```
Person* Programmer::clone() const {
        return new Programmer(*this);
}

// the same to Student and Actor …
```

המילה השמורה virtual איננה חייבת להופיע שוב בהצהרת הפונקציה של המחלקה הנורשת, הפונקציה היא בכל מיקרה וירטואלית כיוון שהיא מוצהרת ככזו במחלקת האב. למרות הנאמר, נהוג להוסיף את המילה virtual גם במחלקה הנורשת לשם הבהירות.

כעת פלט התוכנית (לאחר שינוי תעודת הזהות של yori) יהיה:

```
<<null,{stud 1},<null,{per 2},null>>,{per 5},<<null,{stud 7},null>,{prog  8},null>>
```

virtual destructor

כאשר מנסים לחסל אובייקט ממחלקה נורשת ע"י מצביע למחלקה המורישה, יש סכנה לשחרור לא שלם של הזיכרון. לדוגמה:

```
#include <iostream>
class A{
public:
  ~A();
};
A::~A(){std::cout << "A dtor \n"; }

class B : public A {
  int* _p;
public:
  B();
  ~B();
};
B::B(): _p(new int(7)){}
B::~B() { std::cout << "B dtor\n";
 delete _p;
}
int main() {
  A* ap = new B();
  delete ap;
}
```

הפלט של התוכנית יהיה: `A dtor`

מכיוון שה – `dtor` של A לא הוגדר כווירטואלי, אין אפשרות לדעת בזמן ריצה שצריך להפעיל את ה `dtor` של B. במקרה זה, מופעל ה – `dtor` של A וההקצאה הדינמית המוצבעת ע"י `_p` לא שוחררה.

אם נשנה את ה – `dtor` של A להיות וירטואלי, שחרור הזיכרון יעשה כהלכה:

```
class A{
public:
  virtual ~A();
```

```
};
```

פלט:

```
B dtor
A dtor
```

מדוגמה זו ניתן להסיק, כי כאשר כותבים מחלקה אותה מתכוונים לרשת ומתכוונים להשתמש ב – polymorphism צריך להצהיר על ה – `dtor` כ – `virtual`.

pure virtual

כאמור, מחלקות מתארות מושגים בעולם הבעיה. לעיתים, המושג שאותו אנו מעונינים לתאר בעזרת המחלקה הוא מושג מופשט לחלוטין, כל תפקידו הוא למצות מכנה משותף והוא לא אמור להוות תבנית ליצירת אובייקטים. נחשוב לדוגמה על מחלקה המייצגת ישות וניח שכל ישות בעולם שלנו, ניתנת להדפסה בצורה כלשהי, השונה לכל ישות. במצב זה, אין משמעות מוגדרת למימוש של פונקצית הדפסה במחלקה של הישות ולכן נשאיר אותה לא ממומשת. המחלקה 'ישות' במקרה שלנו תייצג רק את המכנה המשותף לכל הישויות בעולם, היא תאפשר להתייחס אליהם כישויות אך לא תאפשר יצירת אובייקטים מסוגה:

```
#include <iostream>
enum Color {blue,green};
class Entity {
  Color _color;
public:
  void printColor();
  virtual void print()=0;
};
void Entity::printColor() {
  //...
}
class Box :public Entity {
public:
  virtual void print();
};
void Box::print() {
  std::cout << "A Box\n";
}
int main() {
  // Entity e; // can't make an instance of an abstract class
  Box b;
  b.print();
  b.printColor();
  Entity* p = &b; // ok: p is just a pointer
  p->print(); // polymorphism
}
```

כאשר כותבים '=0' בסוף הצהרה של פונקציה וירטואלית, הכוונה היא שלפונקציה אין מימוש במחלקה והיא תמומש רק במחלקות היורשות. פונקציה כזאת נקראת פונקציה וירטואלית טהורה – pure virtual. כאשר למחלקה מסוימת יש פונקציה pure virtual אחת או יותר, המחלקה היא מחלקה אבסטרקטית והשימוש היחידי המותר בה הוא ליצירת מצביעים ו – references. אין אפשרות ליצור אובייקטים מסוג מחלקה אבסטרקטית.

יכולנו כמובן לתת מימוש ריק או מימוש אחר לפונקציה print() אבל אז המשמעות הייתה שונה: ישות איננה מחלקה מופשטת שמטרתה מיצוי של מכנה משותף, אלא מחלקה רגילה, ניתן ליצור אובייקטים מסוגה והדפסת אובייקטים כאלה, מוגדרת. כאשר הגדרנו את print להיות פונקציה וירטואלית טהורה, כפנו על כל מחלקה יורשת גשמית (שאיננה אבסטרקטית) של Entity לספק

מימוש ל - print. אם היינו מספקים מימוש ריק ל print ב - Entity המימוש הזה היה המימוש של כל מחלקה יורשת שלא מספקת מימוש משלה.

כפי שרואים בדוגמה, ישות יכולה להכיל members שונים המשותפים לכל הישויות בעולם, במקרה זה color_ ו-()printColor. למרות שלא ניתן ליצור אובייקטים מסוג מחלקה אבסטרקטית, היא יכולה להכיל משתנים ופונקציות ממשיים המהווים גם הם מכנה משותף.

מחלקה היורשת ממחלקה אבסטרקטית יכולה לממש את הפונקציות הווירטואליות הטהורות שירשה ואז להפך למחלקה גשמית. אם היא מממשת רק את חלקן היא נהיית מחלקה אבסטרקטית בעצמה:

```
#include <iostream>
class A {
  virtual void foo1()=0;
  virtual void foo2()=0;
};

class B : public A {
  virtual void foo1();
};
void B::foo1() {
  //...
}
class C : public B {
  virtual void foo2();
};
void C::foo2() {
  //...
}
int main() {
  // A a; // c.error: A is an abstract class
  // B b; // c.error: B is an abstract class
  C c; // ok: C is a concrete class
}
```

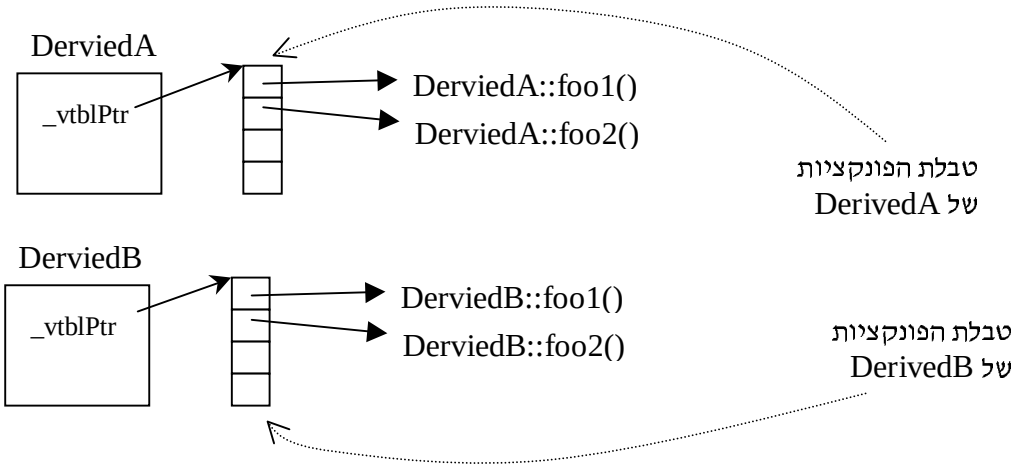
אופן המימוש של polymorphism ב – C++

כאמור, מנגנון ה – polymorphism מאפשר הפעלה של פונקציה מתוך מצביע ל – Base, הפונקציה המופעלת תלויה בסוג האובייקט המוצבע בזמן ריצה ואיננו ידוע בזמן קומפילציה:

```
void foo(Base* p) {
        p->foo1(); // compiler: should I call DeriveA::foo1() or DerivedB::foo1() ?
        p->foo2(); // compiler: should I call DeriveA::foo2() or DerivedB::foo2() ?
}
```

נשאלת השאלה איך נקראת הפונקציה המתאימה בזמן הריצה ?

מימוש נפוץ של המנגנון הוא בעזרת טבלה הנקראת virtual function table או vtbl. לכל מחלקה בעלת פונקציה וירטואלית מוסף מצביע נוסף, סמוי, לטבלה של מצביעים לפונקציות (מצביעים לפונקציות הוא חומר שיילמד בהמשך). בזמן ריצה, כאשר מופעלת פונקציה וירטואלית, שם הפונקציה קובע את המיקום בטבלה, ואז דרך המצביע הנמצא במקום זה, מופעלת הפונקציה המתאימה.



הקומפיילר, אם כן, פותר את הבעיה באופן הבא:

```
void foo(Base* p) {
        p->foo1(); // compiler generates code like: (p->_vtblPtr[0]) ()
        p->foo2(); // compiler generates code like: (p->_vtblPtr[1]) ()
}
```

(הדבר יהיה מובן טוב יותר כאשר תלמדו על מצביעים לפונקציות)

המצביע לטבלת הפונקציות הווירטואליות דורש זיכרון נוסף עבור כל אובייקט מסוג המחלקה. גודל הזיכרון הנוסף הוא גודל כגודל הזיכרון הדרוש לאכסון מצביע בודד. הזמן הדרוש להפעלת פונקציה ווירטואלית הוא רק הזמן הדרוש למעבר דרך שתי ההצבעות.
הרצת התוכנית הבאה תדגים את עלות הזיכרון של המצביע לטבלה:

```
#include <iostream>
class A{
  void foo();
};
class B{
  virtual void foo();
};

int main() {
  std::cout << sizeof(A) << std::endl << sizeof(B) << std::endl;
}
```

חשוב להבין שמלבד העלות בזיכרון, קיימת גם עלות קטנה בזמן הריצה. בכל פעם שמפעילים פונקציה ווירטואלית, יש מעבר דרך מצביע נוסף.

upward casting ו – downward casting

כאמור, מנגנון ה – polymorphism, מבחינה טכנית, הוא הפעלת פונקציה של derived class מתוך מצביע מטיפוס ה – base class. בכדי שמצביע מסוג derived יוכל להצביע על אובייקט מסוג base, דרושה המרת טיפוסים. ראינו כבר שהמרה זו יכולה להתבצע בצורה לא מפורשת:

```
Base* p = new Derived(..); // same as: Base* p = (Base*) new Derived(..);
```

ההמרה שהתבצעה כאן (מ – *Dervied ל- *Base), המרת טיפוסים במעלה היררכית הירושה ניקראת upward casting. upward casting מאפשרת להתייחס לאובייקט בצורה אבסטרקטית יותר ואין בה סכנה לדריסת זיכרון. ההמרה ההפוכה, במורד היררכית הירושה, ניקראת downward casting:

```
Derived* p = (Derived*) new Base();
// Derived* p = new Base();// this will not pass compilation
```

downward casting דורש המרה מפורשת כיוון שהוא עלול להיות מסוכן. מכיוון שלא כל Base הוא Derived (למרות שכל Derived הוא Base), התייחסות ל- Base כאל Derived עלולה לגרום לדריסת זיכרון.

סיכום

מנגנון ה – polymorphism מאפשר להתייחס לאובייקט ברמות הפשטה שונות ולהפעיל דרך ההתייחסות פונקציות ספציפיות למחלקת האובייקט. המנגנון מאפשר ניסוח של אלגוריתמים ומבני נתונים במושגים מופשטים ובכך להפוך אותם לכלליים.
כאשר מנסים למדל עולם בעיה מסוים בעזרת מחלקות, כדאי לנסות להעלות כל מכנה משותף (מימושי או ממשקי) כמה שיותר גבוהה בהיררכית הירושה.