# מחלקות

## הגדרה

מחלקה היא תוכנית שבאמצעותה אפשר לייצר אובייקטים. בתוך המחלקה נגדיר מהם השדות של האובייקט, כיצד בונים אותו, ואת כל השיטות (מתודות) שניתן להפעיל עליו. לאחר שכתבנו את המחלקה ניתן ליצור אובייקטים רבים ממחלקה זו שהם בעצם משתנים מהטיפוס של המחלקה. שם המחלקה תמיד מתחיל באות גדולה. כדי להצהיר על מחלקה נכתוב:

class ClassName {

//Methods, constructors, class variables

};

יש לשים לב שבשונה מ-Java מחלקות ב-C++ מסתיימות ב-' ; ' ולא רק בסוגריים מסולסלות.

### Struct

ב++C, ההבדל היחיד בין struct לבין class הוא בהרשאות הגישה כברירת מחדל. ב-struct, הרשאת הגישה כברירת מחדל היא public, וב-class היא private.

## מחלקות בC++

בשפת C, הדרך המקובלת ליצור מבנים מורכבים היא להגדיר struct. struct הוא אוסף של משתנים שנמצאים **ברצף בזיכרון**. אפשר גם להגדיר משתנים מורכבים יותר על ידי struct בתוך struct וכו'. אבל ב-struct אין שיטות, אם רוצים לעשות איתו משהו צריך לכתוב שיטות חיצוניות.

בשפת Java יש מחלקות (class) שאפשר לשים בהן גם משתנים וגם שיטות. היתרון הוא שהקוד קריא יותר, כי כל המידע והפעולות הדרושות למימוש האובייקט נמצאים במקום אחד. החיסרון ב-Java שבכל אובייקט שיוצרים, המשתנים שלו לא נמצאים ברצף בזיכרון, אלא כל משתנה הוא פוינטר אל הערכים שאליהם אנו רוצים להגיע. הפוינטרים שמורים ב-stack, והערכים אליהם מצביעים שמורים ב-heap.

המחלקות בשפת ++C הן שילוב של כל היתרונות בשתי השפות. מצד אחד יש בה struct כמו ב-C, אבל אפשר לשים בו גם שיטות (נלמד בהמשך). למעשה, המטרה העיקרית ביצירת C++ הוא הוספה של תכנות מונחה עצמים שבמרכזו עומדים יצירת מחלקות. מצד שני, יש בה מחלקות כמו ב-Java, אבל המשתנים נשמרים ברצף בזיכרון ולא על ידי פוינטרים. ביצירת אובייקט ניתן להגדיר האם נרצה לשמור אותו ב-stack או ב-heap (נלמד בהמשך). למה זה משנה?

* בלי מצביעים התוכנה תופסת **פחות זיכרון** ודורשת **פחות זמן** כשניגשים למשתנים.
* כשכל המידע נמצא באותו מקום בזיכרון, **ניהול הזיכרון מהיר יותר**. בפרט, כשמשתמשים בזיכרון מטמון (cache), זה מאד יעיל שכל המבנה נטען בבת-אחת לאותו בלוק בזיכרון.

## משתני המחלקה

משתני המחלקה הם המשתנים אותם מכיל האובייקט בתוכו. הם יכולים להיות מכל סוג שהוא: משתנים פשוטים כמו int או string, מערכים, או אף אובייקטים אחרים. מצהירים עליהם בתוך בלוק המחלקה. לאחר שבנינו אובייקט מטיפוס המחלקה נוכל לפנות אל משתני המחלקה (אלא אם כן המשתנה הוגדר כ-private) על ידי שנרשום אופרטור נקודה לאחר האובייקט - object.class\_var.

### משתנים סטטיים

משתני מחלקה סטטיים אלו הם משתנים שנוצרים פעם אחת בלבד בתחילת התוכנית, והם אינם שייכים לאף אובייקט שנוצר מהמחלקה. משתמשים במשתנים סטטיים כדי לתאר את המחלקה כולה או משהו שמשותף לכל האובייקטים. כדי להגדיר שמשתנה הוא סטטי, יש להוסיף לתחילת השורה שבה הוא מוצהר את המילה השמורה static. static int var = 0;

משתנה סטטי, אלא אם כן הוא const, לא ניתן לאתחל בתוך המחלקה אלא אך ורק מחוץ למחלקה. הסיבה היא שכל עוד לא יצרנו אובייקט מסוג המחלקה, לא ניתן להקצות זיכרון. בשורת האתחול מחוץ למחלקה יש לרשום את סוג המשתנה, מרחב השם, שם המשתנה, ולאחר מכן את ההשמה.

int ClassName::static\_var = 0;

כל פעם שאובייקטים פונים למשתנה סטטי הם בעצם פונים לאותו שטח בזיכרון. מחוץ למחלקה, ניתן לפנות למשתנה סטטי ולשנות אותו דרך שם המחלקה ופעמיים נקודותיים. ניתן גם לפנות דרך אובייקט מהמחלקה אך נחשב לסגנון גרוע.

ClassName::static\_var // good

Object\_name.static\_var // bad style

## בנאים - Constructors

בנאי הוא שיטה שתפקידה ליצור ולאתחל אובייקט חדש ולתת ערכים לכל משתני המחלקה שאינם סטטיים. שמו של הבנאי זהה לשם של המחלקה. יכולים להיות מספר בנאים שמקבלים פרמטרים שונים, כך שכל בנאי יוצר אובייקט לפי הפרמטרים שהוא מקבל. פעולה זו נקראת "העמסת בנאים". בתוך הבנאי אסור שתהיה פקודת return ואפילו לא return void.

ClassName(type arg, …) {}

### בנאי ברירת מחדל

הוא בנאי שאינו מקבל פרמטרים כלל או בנאי שלכל פרמטר בו יש ערך ברירת מחדל. אם אנו לא מגדירים אף בנאי במחלקה, אזי הקומפיילר יוסיף באופן אוטומטי בנאי ברירת מחדל (Default Constructor) שקורא לבנאי ברירת המחדל של כל שדה באובייקט. אמנם אם הגדרנו בנאי עם פרמטרים ולא בנאי ברירת מחדל, אם ננסה לקרוא לבנאי ברירת מחדל נקבל שגיאת קומפילציה. לכן חשוב שאנו נגדיר תמיד בנאי ברירת מחדל.

ClassName() {…}

ClassName(int a=1, double b=2.0, …) {…}

### רשימת אתחול

כאשר יש מחלקה שאחד או יותר מהשדות שלה גם הוא עצם מורכב, כלומר יש כאן עצם קטן בתוך עצם גדול, אזי בבניית העצם הגדול קודם צריך לאתחל את העצם הקטן על ידי קריאה לבנאי ברירת מחדל שלו, ולאחר מכן לבנאי של העצם הגדול, שיגדיר בתוכו לאיזה בנאי של העצם הקטן לקרוא. בפירוק זה בדיוק הפוך, קודם יפורק העצם הגדול ולאחר מכן העצם הקטן. אך מה קורה כאשר בעצם הקטן אין בנאי ברירת מחדל אלא יש בנאי אחר? במקרה זה הקומפיילר לא יבנה לו בנאי ברירת מחדל אלא תתקבל שגיאת קומפילציה. הפתרון היעיל ביותר לבעיה זו הוא רשימת אתחול.

בכל בנאי נוכל להגדיר רשימת אתחול, שבה אנו יכולים עבור כל משתנה מחלקה להכניס ערכים. בפרט, עבור כל עצם פנימי נוכל להפעיל את הבנאי שלו. כך אנו מאתחלים את העצם הקטן עם הבנאי שאנו רוצים וחוסכים הפעלה פעמיים של הבנאי על העצם הקטן, וגם מונעים שגיאת קומפילציה במקרה שלא מוגדר בנאי ברירת מחדל.

כדי להגדיר רשימת אתחול, לאחר הצהרת הבנאי ולפני פתיחת הסוגריים המסולסלים נכתוב נקודותיים, ואז עבור כל משתנה מחלקה שנרצה לאתחל נרשום את שמו ובסוגריים את הערכים שנרצה להכניס לו. משתני המחלקה יופרדו באמצעות פסיק. עבור עצם פנימי הכנסת הערכים תגדיר לאיזה בנאי שלו אנו קוראים. נוכל בתוך הסוגריים גם לבצע פעולות אריתמטיות.

class A {

A(int arg) {…}

A(int arg1, arg2) {…}

}

Class B {

A a1, a2;

B(int i) : a1(i), a2(i, 2\*i){…}

}

## יצירת אובייקט

כאשר יוצרים אובייקט ישנה אפשרות ליצור אותו על המחסנית או על הערימה על ידי קריאות שונות לבנאי.

### יצירה על המחסנית

כדי ליצור אובייקט באמצעות **בנאי עם פרמטרים**, יש לרשום את שם המחלקה (שהוא גם שם הבנאי), השם שנרצה לאובייקט, ובסוגריים מסולסלות להכניס את הערכים המתאימים לפרמטרים של הבנאי. לדוגמא, אם נרצה ליצור שני אובייקטים של point משני בנאים שונים נרשום:

1. Point p1 {1.0};
2. Point p2 {1.0, 2.0};

כדי ליצור אובייקט באמצעות **בנאי ללא פרמטרים**, יש לעשות אותו הדבר רק לא לשים סוגריים בכלל ואפילו לא ריקים, זה יגרום לשגיאת קומפילציה כי הקומפיילר יחשוב שמנסים להגדיר פה פונקציה.

1. Point p;

ניתן גם ליצור מערך של אובייקטים, ובאותה שורה לאתחל את כולם באמצעות בנאים שונים.

Point p[3] { {1.0},{1.0, 2.0} } //call (1) (2) and (3)

עד כה הראינו קריאה לבנאי באמצעות סוגריים מסולסלים { } ובתוכם הפרמטרים, אמנם אפשר לקרוא לבנאי באותה הצורה גם באמצעות סוגריים עגולים ( ). למעשה עבור בנייה של אובייקט אחד אין בין שתי השיטות הבדל. אמנם כשרוצים לבנות מערך של אובייקטים זה אך ורק אם סוגריים מסולסלים, כמו שהגדרנו לעיל. בפועל מקובל יותר להשתמש בסוגריים מסולסלים גם עבור אובייקט יחיד.

### יצירה על הערימה

כאשר רוצים ליצור אובייקט על הערימה נרשום =, אופרטור new ולאחריו שם המחלקה. אם נרצה לקרוא לבנאי עם פרמטרים נשים לאחר מכן סוגריים מסולסלות שבהם נשים את הערכים המתאימים לפרמטרים של הבנאי הרצוי. כדי להגדיר מערך של אובייקטים נרשום [size] לאחר שם המחלקה.

Point \*p1 = new Point;

Point \*p2 = new Point {1.0};

Point \*p3 = new Point {1.0, 2.0};

Point \*p4 = new Point[3] { {1.0},{1.0, 2.0} }

## מפרקים - Destructors

ב-C++ יש עקרון חשוב בבניית אובייקטים. כל אובייקט שנוצר חייב לעבור תהליך של בנייה שמתבצע באמצעות הבנאי, וכאשר הוא מפסיק להתקיים הוא חייב לעבור תהליך של פירוק שמתבצע באמצעות המפרק. מטרת המפרק היא למחוק את כל שדות המחלקה שהן מצביעים או מערכים שהוקצו באופן דינמי באמצעות אופרטור new. בכל מחלקה בה יש לנו מצביעים שהוקצו באופן דינאמי נצטרך למחוק אותם באמצעות המפרק.

המתכנת אף פעם לא צריך לקרוא למפרק באופן ידני, זוהי האחריות של הקומפיילר להכניס קריאה למפרק ברגע שהעצם מפסיק להתקיים.

* אובייקט שנשמר על המחסנית (ללא אופרטור new), מפסיק להתקיים בסוף התוכנית או כאשר יוצאים מה-scope של האובייקט (הבלוק שבו הוגדר). המפרק יופעל אוטומטית במצב זה.
* אובייקט שנשמר על הערימה (באמצעות אופרטור new), מפסיק להתקיים אך ורק כאשר נפעיל עליו אופרטור delete. רק פעולה זו תפעיל את המפרק של האובייקט.

מפרק הוא שיטה שאינה מקבלת פרמטרים ולא מחזירה ערך, ששמה מתחיל בטילדה ~ (אופרטור המציין bitwise not), ואחריו שם המחלקה. יכול להיות רק מפרק אחד בכל מחלקה. אם לא הגדרנו מפרק אזי הקומפיילר יגדיר מפרק ברירת מחדל משלו, אמנם הוא לא יעבוד עבור אובייקטים שיש בהם פוינטר לזיכרון דינמי. לאובייקטים כאלו חייבים להגדיר במחלקה מפרק אחרת המשאבים לא ישתחררו ויהיו דליפות זיכרון. פוינטר לא מפורק אלא רק האובייקט אליו מצביע מפורק.

class String {

**char \*s**;

String(char \*c){ //constructor

size = strlen(c);

**s = new char[size+1]; //allocated memory**

     strcpy(s, c);

}

~String(){ //destructor

**delete [] s;**

}

};

כדי לפרק משתנה או אובייקט שנוצר באמצעות new נפעיל עליו אופרטור delete. וכדי לפרק מערך של משתנים או אובייקטים שנוצרות בעזרת new[] נפעיל עליו אופרטור delete[], הקורא למפרק של כל אחד מהעצמים במערך. אם נפעיל delete ולא delete[] על מערך, אזי רק העצם הראשון יפורק ותהיה דליפת זיכרון או שגיאת קומפילציה.

### עותק שטחי ב-scope פנימי

כאשר מגדירים בתוך scope פנימי עותק שטחי של האובייקט, אזי ביציאה מה-scope הפנימי האובייקט כולו יפורק. כתוצאה מכך כאשר נצא מה-scope החיצוני הקומפיילר ינסה לפרק את העותק המקורי, מה שיגרום לשגיאת קומפילציה, כי לא ניתן לפרק אובייקט שכבר פורק. כדי להימנע מכך לא ניצור עותק שיטחי בתוך scope פנימי.

## שיטות - Methods

שיטות או מתודות הן פונקציות שניתן להפעיל על האובייקט שניצור מהמחלקה. גם בתוך המחלקות ניתן לבצע העמסה של שיטות (Overloading). יש שני דרכים לממש שיטות: מימוש מבפנים ומימוש מבחוץ, נפרט על כך בהמשך.

ישנם שיטות שאם אנו לא מגדירים אותם אזי הקומפיילר מגדיר להם ברירת מחדל, כמו בנאי, בנאי מעתיק, מפרק, אופרטור השמה, וכו'. נוכל למנוע מהקומפיילר להגדיר שיטת ברירת מחדל על ידי אופרטור delete.

Class\_name(Class\_name& other) = delete;

### שיטות סטטיות

בדומה לקיומם של משתנים סטטיים כך גם קיימות שיטות סטטיות, שאינן שייכות לאף אובייקט אלא למחלקה. שיטות אלו קיימות וניתנות להפעלה עוד בתחילת התוכנית לפני שבכלל נוצר אובייקט. כלומר, הן אינן צריכות אובייקט כדי להפעיל אותן, אלא אפשר להפעילם ישירות באמצעות שם המחלקה ולאחר מכן פעמיים נקודותיים.

ClassName::static\_func(arg1, arg2, …);

חשוב לשים לב ששיטות סטטיות יכולות לגשת רק למשתנים סטטיים ושיטות סטטיות אחרות בתוך המחלקה, אך לא למשתנים ושיטות אחרות. כמו כן, אינן יכולות לגשת למצביע this, ואין יכולות להיות וירטואליות.

### שיטות קבועות

אם מגדירים את המילה const בכותרת של שיטה לאחר הסוגריים, אזי נאמר ששיטה זו היא שיטה קבועה. המשמעות היא שבתוך השיטה הזאת המשתנה this יהיה מצביע לעצם קבוע. במילים אחרות, השיטה לא תוכל לשנות את השדות של האובייקט עליו הופעלה (מלבד שדות המוגדרות mutable). מתי נשתמש בשיטות קבועות:

* מתוך שיטות קבועות אי אפשר לקרוא לשיטות שאינן קבועות, אפילו כאלו שאינן משנות את השדות של האובייקט. לכן מלכתחילה כל שיטה שאנו מגדירים שאינה משנה את שדות האובייקט נגדיר const.
* אם יצרנו אובייקט קבוע אפשר להפעיל עליו רק שיטות קבועות. אם בתוכנית הראשית מגדירים עצם כ-const, אפשר לקרוא על העצם הזה רק לשיטות שהוגדרו כ-const.
* זה עוזר לאתר באגים ותקלות. אם שיטה מסוימת מוגדרת כ-const, אפשר להיות בטוחים שהיא לא משנה את העצם, ולכן אם העצם משתנה כנראה התקלה במקום אחר. בנוסף, זה מאפשר לנו לקבל אזהרות על תופעות לוואי לא רצויות. אם נגדיר את השיטה כ-const, הקומפיילר יזהה את הבעיה ויזהיר אותנו.

Void foo(//parameters) const {…}

כשמוסיפים את המילה const לשיטה היא הופכת לחלק מהחתימה של השיטה. מכאן שאפשר ליצור שתי שיטות שונות עם אותו שם ואותם פרמטרים, אלא שאחת עם const ואחת בלי. הקומפיילר ישתמש בשיטה הנכונה לפי ההקשר, אם משתמשים בשיטה כ-lvalue הוא יקרא לשיטה הלא קבועה, ואם משתמשים בשיטה כ-rvalue הוא יקרא לשיטה הקבועה. באותו אופן, אם מפעילים את השיטה על אובייקט רגיל הוא יקרא לשיטה הלא קבועה, ואם מפעילים על אובייקט const הוא יקרא לשיטה הקבועה.

## מימוש מבפנים (Inline) ומבחוץ (Outline)

יש שתי דרכים לממש מחלקות ב-++C:

* מבפנים (Inline) - בדרך זו מימוש המחלקה הוא ישירות בתוך בלוק המחלקה (בדומה לJava). כלומר, עם הצהרת הבנאים והשיטות נפתח סוגריים מסולסלות ונממש את כל מה שנרצה בפנים.
* מבחוץ (Outline) - דרך שניה היא לשים בתוך המחלקה רק **הצהרות** של השיטות, בלי גוף. את המימוש עצמו שמים מחוץ למחלקה. כדי לפנות אל השיטה מחוץ למחלקה נצטרך לרשום את שם המחלקה ולאחר מכן פעמיים נקודותיים (כמו במרחבי שם. דוגמא בהמשך הסעיף).

### חלוקה לקבצים

עקרונית, ניתן לערבב בין המימושים כך שחלק מבפנים וחלק מבחוץ, וכן את המימוש החיצוני אפשר לשים באותו קובץ. אבל, מקובל ורצוי מאוד להגדיר כל מחלקה בשני קבצים נפרדים:

**קובץ הכותרת** - בדרך כלל עם סיומת hpp או h. כולל את הצהרות המחלקה, השדות, הבנאים והשיטות, אבל בלי מימושים. בקובץ hpp בדרך כלל גם נממש שיטות שלא נרצה לשנות.

class Complex {

double re, im;

public:

Complex(double re, double im); //outline constructor

string to\_string(); //outline method

};

**קובץ התוכן** - בדרך כלל עם סיומת cpp. כולל את המימוש של השיטות.

#include "Complex.hpp"

Complex::Complex(double re, double im) {

(\*this).re = re;

this->im = im;

}

Complex::string to\_string() {

return to\_string(re) + "+" + to\_string(im) + "i"

}

חלוקה זו עדיפה משתי סיבות:

1. **הנדסת תוכנה.** אם ניתן את המחלקה שלנו למישהו אחר, הוא ירצה לראות איזה שיטות יש בה, אבל לא יעניין אותו כיצד מימשנו אותן. לכן הוא יסתכל בקובץ הכותרת, ועדיף שהקובץ הזה יהיה קטן ופשוט ככל האפשר. בנוסף, כך אנו שומרים על עקרון הכימוס האומר שכל מתכנת יודע רק מה שהוא צריך לדעת ולא יותר.
2. **זמן קומפילציה.** בקובץ main המשתמש במחלקה שבנינו מספיק לייבא את הקובץ hpp בלבד, הוא לא צריך להכיר את cpp. כתוצאה מכך, גם אם המימוש ב-cpp ישתנה, כל עוד קובץ ה-hpp לא ישתנה אין צורך לקמפל מחדש את main, אלא רק לקשר בין הקבצי o שלהם. וכך אנו חוסכים זמן קומפילציה שיכול להיות מאוד משמעותי במערכות תוכנה מורכבות בהם קומפילציה לוקחת הרבה זמן.

שם הקובץ לא חייב להיות כמו שם המחלקה, ואף בכל קובץ ניתן להגדיר מספר מחלקות שונות. אך מקובל לכתוב כל מחלקה בשתי קבצים נפרדים ששמם הוא שם המחלקה - ClassName.hpp, ClassName.cpp.

## הרשאות גישה - Modifiers

ניתן להגדיר למשתני מחלקה, בנאים ומתודות הרשאות גישה, כך שניתן לגשת אליהן לפי שלושת האפשרויות הבאות:

1. **private** - ניתן לגשת אך ורק מתוך מחלקה עצמה או פונקציות "חברות", אך לא מחוץ למחלקה. זהו גם **הברירת מחדל** כאשר לא מגדירים הרשאת גישה.
2. **protected** - ניתן לגשת רק מהמחלקה עצמה, או ממחלקות מאותו package, או ממחלקות שיורשות ממנה.
3. **public** - ניתן לגשת מכל מקום.

כדי להגדיר הרשאת גישה נרשום את אחת האפשרויות לעיל ולאחריו נקודותיים. כעת כל המשתנים, הבנאים, והשיטות שיבואו אחריו, עד שנגדיר הרשאת גישה אחרת או עד סיום הבלוק, יוגדרו לפי הרשאת גישה זו.

class ClassName {

public:

//everything public

private:

//everything private

};

### עקרון הכימוס - Encapsulation

ככלל, עקרון חשוב בתכנות מונחה עצמים הוא עקרון הכימוס. העיקרון אומר שיש להשתדל שכמה שיותר מידע יהיה פרטי כדי למנוע שימוש שגוי בו. תמיד נרצה להציג את הממשק למשתמש אך נחביא את המימוש.

יש להשתדל להעניק את הרשאת הגישה private לכמה שיותר משתני מחלקה. הסיבה לכך היא כדי שמתכנתים אחרים יוכלו לגשת למשתני המחלקה אך ורק דרך מתודות שאנחנו כתבנו ולא יהיה להם גישה ישירה אליהם. כך אנו בעצם מבטיחים שמשתני מחלקה רגישים מכילים כל העת ערכים חוקיים בלבד. מתודות לעומת זאת בדרך כלל נגדיר כ-public, אולם לעתים גם אותם נגדיר כ-private. אנו נעשה זאת כאשר מתודה נותנת שירות חשוב ורגיש ללוגיקה הפנימית של המחלקה, כך שלא נרצה לאפשר לכל מתכנת להפעיל אותה.

## this

המילה השמורה this מכילה מצביע קובע לאובייקט הנוכחי, כלומר האובייקט שעליו המתודה פועלת. this הוא מצביע למצביע, שהרי גם אובייקט הוא מצביע, ולכן כדי לפנות אל המשתנים והמתודות של האובייקט נשתמש באופרטור חץ, או באופרטור נקודה על \*this. לדוגמא:

Point::Point(int x, int y) {

this->x = x;

(\*this).y = y;

}

נשתמש במילה השמורה this בשני מצבים:

1. במימוש המחלקה, כאשר רוצים לפנות אל משתני המחלקה מספיק לרשום את שם המשתנה, לדוגמא var. אמנם כאשר כבר יש משתנה בשם זה (כגון שאנחנו הגדרנו אותו או שהשיטה מקבלת ארגומנט בשם זהה, כמו בדוגמא לעיל), אזי אם נשתמש בשם var הקומפיילר לא יפנה אל משתנה המחלקה. במצב זה נכתוב this->var כדי לפנות אל משתנה המחלקה var.
2. כאשר בהרצה של מתודה רוצים שבתוכה היא תפעיל מתודה נוספת, ולשלוח אל המתודה השנייה את המצביע של האובייקט שעליו המתודה הראשונה פועלת. במקרה כזה נוכל לשלוח אל המתודה השנייה את המילה \*this בלבד, כך שהקומפיילר יבין שנשלח האובייקט שעליו המתודה הראשונה פועלת.

## פונקציה friend

במקרים מסוימים יש פונקציה שקשורה למחלקה אך מסיבות שונות לא נרצה להגדיר אותה בתוך המחלקה אלא בחוץ, אמנם כן נרצה שיהיה לה גישה למשתני המחלקה. פתרון אפשרי הוא להגדיר את משתני המחלקה כציבוריים (public), הבעיה היא שאז כל פונקציה תוכל לגשת אליהן, מה שסותר את עקרון הכימוס. בC++ יש פתרון לבעיה זו והוא להגדיר פונקציה זו כפונקציה "חברה", כך שאפילו אם משתני המחלקה פרטיים (private) לפונקציה זו יהיה גישה אליהן. כדי להגדיר פונקציה חברה יש להצהיר בתוך המחלקה (בקובץ h) על הפונקציה ולהוסיף בתחילת ההצהרה את המילה "friend". המימוש יהיה בחוץ.

friend void foo (…); //hpp

void foo (…){//implementation} //cpp

באותו אופן, אפשר להגדיר מחלקה חברה. מחלקה אחת יכולה להגדיר מחלקה אחרת כ"חברה" ובכך לאפשר לה לגשת למשתנים הפרטיים שלה. זה שימושי כשיש שתי מחלקות הקשורות באופן הדוק זו לזו. למשל מחלקה של וקטור ומחלקה של איטרטור על הוקטור הזה.

## העתקה

ישנם ארבעה מקרים בהם מעתיקים אובייקט:

1. כשמגדירים אובייקט חדש באמצעות אובייקט קיים. לדוגמא: complex a=b, או complex a(b).
2. כששולחים אובייקט כפרמטר לפונקציה by value.
3. כשמחזירים אובייקט מפונקציה by value.
4. בפעולת השמה של אובייקט קיים לאובייקט **קיים** אחר. לדוגמא: complex a; a=b.

בשלושת המקרים הראשונים ההעתקה מתבצעת באמצעות בנאי מעתיק, ואילו במקרה הרביעי באמצעות אופרטור השמה '='. נשים לב שהמקרה הראשון שונה מהרביעי בכך שבראשון האובייקט נוצר כמועתק. כאשר אנו לא מגדירים בנאי מעתיק או מעמיסים אופרטור השמה, הקומפיילר מוסיף אותם כברירת מחדל. במקרה זה, כל מה שהם עושים הוא העתקת סיביות מוחלטת מהאובייקט המועתק אל המעתיק. אמנם ברירת מחדל זו יכולה לגרום לשגיאות במחלקות מורכבות שמכילות מצביעים ומערכים שנוצרו על הערימה ולא במחסנית. בעיות נפוצות הן:

* כאשר ניצור אובייקט b באמצעות אובייקט קיים a שיש לו מצביע על הערימה (מקרה 1 לעיל), אזי a ו-b שניהם מצביעים אל אותו מקום בערימה. כאשר יופעל המפרק ~ הוא ימחק קודם את הזיכרון ש-b מצביע עליו בערימה. אך כשינסה למחוק לאחר מכן גם את a תתקבל שגיאת זמן ריצה, שכן לא ניתן למחוק זיכרון שכבר נמחק.
* כאשר ניצור אובייקט b עם מצביע על הערימה ולאחר מכן נבצע השמה עליו מאובייקט אחר a מאותה מחלקה (מקרה 4 לעיל). אזי כעת b מצביע על המקום ש-a מצביע עליו בערימה. במקרה זה לא ניתן לגשת אל הזיכרון ההתחלתי שהוקצה ל-b וכן לא ניתן למחוק אותו. זיכרון זה נקרא "בלוק אבוד" ומהווה דליפת זיכרון.

כדי לפתור בעיות אלו, בכל פעם שנרצה להגדיר אובייקטים עם מצביעים על הערימה ולבצע השמה בין אובייקטים שונים, נצטרך להגדיר שלוש שיטות: בנאי מעתיק, אופרטור השמה '=', ומפרק.

### בנאי מעתיק

עבור מחלקה T, בנאי מעתיק הוא בנאי שמקבל פרמטר אחד בלבד שהוא רפרנס קבוע לאובייקט מסוג המחלקה, **const T&**. חייב שהפרמטר יהיה מסוג רפרנס קבוע, שכן אם נעביר פרמטר T בלבד נצטרך לקרוא לבנאי מעתיק, שאותו אנו הרגע מגדירים. לכן אם הפרמטר לא מסוג רפרנס קבוע ניכנס ללולאה אינסופית. בתוך הבנאי נעתיק את כל השדות של האובייקט המועתק. מערך או מבנה נתונים אחר נעתיק איבר אחר איבר ולא את המערך כולו. הקומפיילר קורא לבנאי הזה אוטומטית בכל פעם שצריך להעתיק עצם מסוג T.

ClassName(const ClassName& obj){

//copy values

}

כאשר לא מוגדר בנאי מעתיק, הקומפיילר מגדיר בנאי מעתיק משלו. בנאי זה קורא לבנאי המעתיק עבור כל אחד מהשדות של המחלקה. אם גם להם אין בנאי מעתיק יקרא לבנאי ברירת מחדל שלהם.

### אופרטור השמה '='

גם כן מקבל רפרנס קבוע של אובייקט מסוג המחלקה. בתוך האופרטור השמה יש קודם לבדוק האם ההשמה היא מהאובייקט אל עצמו. אם כן להחזיר this. לאחר מכן נמחק את המערך שנמצא על הערימה, ניצור אותו מחדש ונשווה אותו למערך באיבר המועתק. לבסוף נחזיר את this כדי לאפשר שרשור על האופרטור.

ClassName& operator=(const ClassName& obj) {

if (this==&obj) return \*this; // preventing problems in a=a

delete[] array;

//copy values

return \*this; // allows a=b=c=…

}

## המרה

עבור מחלקה שהגדרנו, לעיתים נרצה לבצע המרה מטיפוס כלשהו אל המחלקה שלנו, כך שנוכל לבצע השמה של אובייקט מהטיפוס אל אובייקט מהמחלקה שלנו. וכן ההיפך, נרצה להמיר אובייקט מסוג המחלקה שלנו אל אובייקט מטיפוס כלשהו, כך שנוכל לבצע השמה ביניהם. המרה מטיפוס כלשהו אל המחלקה שלנו מתבצעת באמצעות בנאי המרה, ואילו המרה מהמחלקה שלנו אל טיפוס אחר מתבצעת באמצעות אופרטור המרה.

### בנאי המרה

בנאי המרה ממיר מטיפוס כלשהו למחלקה שלנו. הוא מקבל כפרמטר אובייקט מטיפוס כלשהו, ובתוכו נגדיר כיצד לאתחל אובייקט מסוג המחלקה שלנו באמצעות הפרמטר. לדוגמא, בהגדרת המחלקה fraction המייצגת שבר a/b, נבצע המרה מ-int אל fraction כך:

fraction(int n)){

a = n; b = 1;

}

### אופרטור המרה

אופרטור המרה ממיר מהמחלקה שלנו לטיפוס כלשהו. כדי להגדיר אופרטור המרה נרשום operator, לאחר מכן את שם הטיפוס או המחלקה אליה אנו רוצים להמיר, ולבסוף סוגריים ריקים () כי אופרטור המרה לא מקבל שום פרמטר. נרצה מאוד ששיטה זו תהיה גם const כי בכל מצב אינו מעוניינים לשנות את האובייקט שאותו אנו ממירים. בנוסף, לאופרטורים אלה אין צורך לציין את הטיפוס המוחזר כיוון שהוא חייב להיות זהה לטיפוס שאליו אנו ממירים. בתוך האופרטור נאתחל אובייקט מסוג הטיפוס שאליו או מעוניינים להמיר ונחזיר אותו. לדוגמא המרה מ-fraction אל double:

Operator double() const{

return (double)a / (double)b;

}

### Implicit ו-explicit

כל המרה יכולה להתבצע בצורה מפורשת (explicit) או בצורה לא מפורשת (implicit). המרה בצורה מפורשת היא המרה שבה אנו מציינים את הבנאי או האופרטור שאיתו אנו מבצעים את ההמרה. ואילו המרה בצורה לא מפורשת היא המרה שבה אנו מבצעים השמה ישירות בין הערכים. לדוגמא:

fraction f(3); \\ explicit

fraction f = 3; \\ implicit

double d = (double)f; \\ explicit

double d = f; \\ implicit

fraction farray[3] {2,3,4}; \\ implicit

כאשר אנו מגדירים בנאי המרה או אופרטור המרה, על ידי המילה השמורה explicit אנו יכולים להגדיר האם ההמרה שלהם היא רק בצורה מפורשת או גם בצורה לא מפורשת. כלומר, כיצד הקומפיילר יכול להשתמש בשיטת ההמרה שהגדרנו. אם נצהיר בתחילת הבנאי או האופרטור explicit אזי הקומפיילר יוכל להפעיל את הבנאי או האופרטור רק כאשר יופעלו באופן מפורש.

**explicit** fraction(int n)){…}

**explicit** Operator double() const{…}

נרצה לעיתים להגדיר שיטת המרה כ-explicit בלבד. יש שתי סיבות לכך: הסיבה הראשונה היא כדי למנוע המרה שלא התכוונו אליה שתוכל לגרום בהמשך לשגיאות. הסיבה השנייה כדי למנוע שגיאות קומפילציה. לדוגמא: עבור הביטוי f+d נקבל שגיאה, שכן הקומפיילר יכול להמיר את f והוא יכול להמיר את d. מכיוון שאין לו איזו עדיפות מהם לבחור הוא יחזיר שגיאת ambiguous. אמנם אם נגדיר אחת משיטות ההמרה explicit תהיה רק אופציה אחת ולא תיזרק שגיאה.

# ירושה

## הגדרה

הורשה הינה דרך להגדרת מחלקות על ידי שימוש בתכונות ושיטות אשר הוגדרו על ידי מחלקות אחרות, כלליות יותר. מחלקה אשר יורשת ממחלקה אחרת, מקבלת ממנה את משתני המחלקה, הבנאים, ואת השיטות שלה. פרט לכך המחלקה עצמה יכולה להגדיר תכונות ושיטות נוספות. הורשה אינה קשר היא מנגנון. מחלקה A שיורשת ממחלקה B מקיימת את הקשר "A הוא סוג של B", ובאנגלית "Is a". לדוגמא, מחלקת "מתכנת" תירש ממחלקת "אדם", מפני שמתכנת הוא סוג של אדם, והיינו רוצים שלמתכנת יהיה את כל התכונות של אדם עם הרחבות המיוחדת למתכנת. המחלקה שממנה יורשים נקראת "בסיס" (base) והמחלקה היורשת נקראת "נגזרת" (derived). אמנם מקובל גם לקרוא להם "אבא ובן".

### למה זה טוב

* תכנות מונחה עצמים מאפשר לנו לכתוב תוכנה שמשקפת את המציאות באמצעות אובייקטים. מנגנון הירושה ממשיך עקרון זה ומאפשר לנו לבנות מחלקה שהיא מקרה פרטי של מחלקה אחרת.
* קוד קריא וברור יותר.
* מאפשר למחזר קוד ולא לממש אותו מחדש.
* כדי להשיג פולימורפיזם בזמן ריצה על ידי דריסת שיטות (Overriding), בניגוד לפולימורפיזם בזמן קומפילציה על ידי העמסת שיטות (Overloading).

### הבדלים בין C++ ל-Java

1. ב-C++ אפשר לרשת ממספר רב של מחלקות בניגוד ל-Java שבה אפשר לרשת ממחלקה אחת ויחידה.
2. ב-C++ כל מחלקה בפני עצמה ולא כמו Java שברירת המחדל היא לרשת מ-Object.
3. ב-C++ אין ממשקים.

## כיצד יורשים

בהמשך שורת ההצהרה של המחלקה היורשת יש לרשום נקודותיים ':' ומייד לאחר מכן את שמה של המחלקה המורישה. אם יש מספר מחלקות שיורשים מהן יש להבדיל ביניהן באמצעות פסיק ולתת הרשאת גישה לכל מחלקת בסיס. רק לאחר מכן, בתוך הסוגרים המסולסלות של המחלקה המוגדרת ניתן להוסיף משתנים ומתודות אשר יתווספו לאלה שהגיעו בהורשה.

class Programmer: public Person {…}

מאחרי הקלעים, מנגנון הירושה עובד כך שכל עצם מסוג Programmer למעשה כולל שדה נסתר מסוג Person. כשקוראים לשיטה שהועברה בירושה על עצם מסוג Programmer, הקומפיילר למעשה קורא לשיטה המתאימה של השדה הנסתר הזה. לכן מבחינת הזיכרון ירושה שקולה להגדרת מחלקה רגילה עם שדה של מחלקה אחרת.

כאשר יורשים ממספר מחלקות יש לשים לב שאין שדות או שיטות עם שם זהה בשני מחלקות מורישות. הסיבה לכך היא שאם אכן יהיו שני שדות או שיטות כאלו, אזי כאשר נקרא להן הקומפיילר לא ידע איזה מהן להפעיל ונקבל שגיאת ambiguous. לכן יש להיזהר מאוד בהורשה מרובה. למעשה, כאשר תכנון המחלקות נכון ומסודר, אין צורך בהורשה מרובה.

## הרשאת גישה

בהצהרה המחלקה היורשת מיד לאחר הנקודותיים, צריך להגדיר הרשאת גישה לשדה הנסתר שנמצא במחלקה היורשת. ברירת המחדל כאשר לא מציינים הרשאת גישה היא private, כלומר ניתן לגשת למה שיש במחלקה המורישה אך ורק מתוך המחלקה היורשת עצמה או פונקציות "חברות", אך לא מבחוץ. אם נגדיר את הירושה כ-public יהיה ניתן לגשת באמצעות אופרטור נקודה על אובייקט מסוג המחלקה היורשת לכל מה שיש במחלקה המורישה.

class Programmer: **public** Person {…}

מחלקה יורשת רק את המשתנים והשיטות המוגדרות public או protected במחלקה המורישה. מה שמוגדר כ-private אין אפשרות לרשת.

## בנאים ומפרקים

בנאים ומפרקים לא עוברים בירושה. אם למחלקת הבסיס יש כמה בנאים ולא הגדרנו שום בנאי למחלקה היורשת, אזי למחלקה היורשת יהיה רק בנאי בלי פרמטרים, מפני שלא יורשת את הבנאים של הבסיס.

כשבונים עצם ממחלקה יורשת, חייבים להגדיר בבנאי שלה כיצד לבנות את השדה הנסתר של מחלקת הבסיס, כמו שאנו עושים עבור שאר השדות. אפשרות אחת היא לקרוא בפירוש לבנאי של מחלקת הבסיס ברשימת האתחול, כמו שקוראים לבנאים של כל שאר השדות. אפשרות שנייה היא לא לקרוא לבנאי של מחלקת הבסיס, ואז הקומפיילר יקרא אוטומטית לבנאי הברירת מחדל שלו. אמנם אם לא קיים כזה נקבל שגיאת קומפילציה. לכן תמיד חשוב להגדיר בכל מחלקה בנאי ברירת מחדל. כתיבה אלגנטית של בנאי מחלקה יורשת היא כזו שכל השדות מאותחלים ברשימת אתחול, והבנאי עצמו ריק.

Programmer(string name, int id, string company) : Person(name, id), \_company(company) { // EMPTY. Considered elegant}

כאמור, גם המפרק לא עובר בירושה, אלא לכל מחלקה יש להגדיר מפרק בנפרד. כאשר מפרקים מחלקה יורשת הקומפיילר מפרק אותו ואחר כך קורא אוטומטית למפרק של מחלקת הבסיס, בדיוק כמו שהוא קורא למפרקים של שאר השדות.

סדר הבניה של אובייקט ממחלקה יורשת הוא כך: בונים שדות מחלקת הבסיס -> אובייקט מחלקת הבסיס -> שדות מחלקה יורשת -> אובייקט מחלקה יורשת. ואילו סדר הפירוק הוא הפוך: מפרקים אובייקט מחלקה יורשת-> שדות מחלקה יורשת-> אובייקט מחלקת הבסיס -> שדות מחלקת הבסיס.

**אופרטור השמה** - טכנית עובר בירושה, אבל מעשית אי אפשר להשתמש באופרטור השמה של המחלקה המורישה, כי הקומפיילר יוצר אוטומטית אופרטור השמה חדש שמסתיר אותו.

## דריסה (Overriding)

המחלקה היורשת יכולה להגדיר שיטות חדשות או לתת משמעות חדשה לשיטות שנורשו על ידי הגדרתם מחדש. פעולה זו מכונה "דריסה" (Overriding). כדי לבצע דריסה צריך להגדיר במחלקה היורשת שיטה חדשה שהחתימה שלה זהה לשיטה הקיימת כבר במחלקה המורישה. לאחר שנעשה זאת השיטה שהגדרנו תחליף את השיטה שירשנו. כאשר נקראת שיטה על עצם ממחלקה מסוימת, נבחרת הגרסה העדכנית ביותר של השיטה. למשל, אם המחלקה Person מגדירה שיטה output, והמחלקה Programmer יורשת אותה ומגדירה שיטה output עם חתימה זהה, אז עצמים מסוג Programmer ישתמשו בשיטה המחליפה.

חשוב לשים לב שהחתימה של השיטה המחליפה חייבת להיות **זהה לגמרי**, כולל ה-const וכל שאר פרטים, אחרת זו לא תהיה החלפה אלא העמסה. כדי שהקומפיילר יוודא עבורנו שדייקנו בחתימה ואכן ביצענו דריסה, ניתן להשתמש במילת המפתח override בתחילת השיטה במחלקה היורשת. אם זוהי לא שיטה דורסת נקבל שגיאת קומפילציה (דומה לתג @Override ב-Java).

אם רוצים להשתמש בשיטה של המחלקה המורישה מתוך המחלקה היורשת, אפילו שיטה שדרסנו, נכתוב את שם המחלקה המורישה, ארבע נקודות '::', ושם השיטה הרצויה. למשל, מתוך Programmer כדי לגשת לשיטה output שירשנו מ-Person נכתוב:

Person::output(...)

ניתן גם לגשת לאופרטורים שירשנו. למשל, אם כותבים אופרטור השמה '=' ב-Programmer ורוצים לגשת לזה של Person, נכתוב:

Person::operator=(...)

לדוגמא, ראה מצגת 13 שקופיות 20-22.

## בנייה השמה והמרה

ניתן לבצע השמה בין אובייקט מסוג מחלקת האב לאובייקט במחלקת הבן. מה שיקרה הוא שהאובייקט מסוג האב יקבל את השדה הנסתר שיש באובייקט מסוג הבן (slicing). לאובייקט מסוג זה כמובן יהיו רק את השיטות שיש למחלקת האב בלבד. כמו כן, ניתן לשלוח לפונקציה המצפה לקבל מצביע מסוג מחלקת האב, מצביע ממחלקת הבן. זוהי המרת up-cast שהיא טבעית וחוקית. אמנם פעולה הפוכה של השמה או המרה בין אובייקט מסוג האב למחלקת הבן תגרור שגיאת קומפילציה, כי לא ניתן לאתחל את כל השדות. פעולה הפוכה זו ניתן לבצע באמצעות המרה דינאמית עליה נלמד בהמשך.

Person p1 = Programmer p2(); // good

Programmer p1 = Person p2(); // bad

## פולימורפיזם

פולימורפיזם (בעברית: "רב-צורתיות") הוא אחד מאבני היסוד של תכנות מונחה עצמים (יחד עם כימוס, הורשה והפשטה). פולימורפיזם מתרחש כאשר ישנה היררכיה של מחלקות הקשורות על ידי הורשה, ממשקים וכדומה. באופן כללי, פולימורפיזם אומר שבכל פעם שאנו מפעילים שיטה על אובייקט של מחלקה שהיא חלק מהיררכיה של מחלקות, תופעל השיטה הרלוונטית שמומשה עבור האובייקט שהפעיל שיטה זו.

שפת תכנות התומכת בפולימורפיזם נותנת למתכנת את היכולת לממש אלגוריתמים ומבני נתונים לשימוש כללי, ולגזור מהם צורות שימוש שונות בהתאם לעצמים ולנסיבות המשתנות, באופן מפורש או מובלע, ובכך להעלות את רמת ההפשטה של התוכנה.

## שיטה וירטואלית

השימוש הנפוץ בפולימורפיזם הוא כאשר יוצרים אובייקט מסוג מחלקת הבסיס ומכניסים לתוכו אובייקטים מסוג המחלקה היורשת, אך עדיין מעוניינים שהפעלת שיטות על אובייקט זה תהיה לפי המימוש העדכני שנדרס במחלקה היורשת. לדוגמא: עבור מחלקת "צורה" עם שיטת חישוב שטח, שיורשות ממנה מחלקות "ריבוע" ו"משולש" שבהם מימוש עדכני לשיטת חישוב השטח, נרצה שהקוד הבא ייתן לנו את חישוב השטח המתאים לכל אובייקט.

class Shape {

int width, height;

int area() { return 0; }

};

class Rectangle: public Shape {

int area () { return (width \* height); }

};

class Triangle: public Shape {

int area () { return (width \* height / 2); }

};

int main() {

Shape shapes[2] = {Rectangle(10,10), Triangle(10,10)};

shapes[0].area(); // will print 0

shapes[1].area(); // will print 0

}

אמנם בדרך זו לא נצליח לקבל פולימורפיזם. הסיבה לכך היא שבמקרה זה תתבצע העתקה של השדה הנסתר אל shapes (slicing), ולכן הקומפיילר כלל לא ידע שיש כאן פולימורפיזם. גם אם נשתמש במערך של מצביעים לרפרנס של המחלקות היורשות, שאז אין slicing, לא נצליח לקבל את התוצאה הרצויה. זאת מפני שהקריאה לשיטות נקבעת **לפי** **טיפוס המשתנה בזמן קומפילציה**, שהוא מחלקת הבסיס במקרה זה. פעולה זו נקראת "קישור סטטי" (static/early resolution). לכן הקומפיילר שוב לא יודע שיש פולימורפיזם.

### virtual

הפתרון לבעיה זו הוא להגדיר את כל השיטות במחלקת הבסיס עליהן נרצה לבצע פולימורפיזם באמצעות המילה השמורה virtual. פעולה זו מאותת לקומפיילר שאנו לא מעוניינים בקישור סטטי עבור שיטות אלו, אלא שההחלטה לאיזה מימוש של השיטה לקרוא תהיה **בזמן ריצה לפי סוג האובייקט** עליו הפעלנו את השיטה. זה נקרא "קישור דינמי" (dynamic/late resolution).

**virtual** int area() { return 0; }

שיטה וירטואלית אינה יכולה להיות סטטית מפני שצריכה גישה לאובייקט שהפעיל אותה. יש לשים לב שאם יוצרים אובייקט מסוג מחלקת הבסיס **ומעתיקים** לתוכו אובייקט מהסוג היורש (slicing), האובייקט החדש הוא מסוג הבסיס. מסיבה זו גם אם השיטה היא וירטואלית, הקומפיילר יקרא לשיטה של מחלקת הבסיס. כדי ליהנות מהיתרונות של שיטות וירטואליות, צריך להשתמש ברפרנס או בפוינטר.

כל שיטה המוגדרת כוירטואלית במחלקת הבסיס, היא וירטואלית באופן אוטומטי גם בכל המחלקות היורשות ממנה. עם זאת, מקובל לסמן גם את השיטות במחלקות היורשות כ-virtual כדי שהקוד יהיה ברור יותר. אין חובה לדרוס כל שיטה וירטואלית, אמנם בדרך כלל זו כוונתנו בהצהרתה ככזו. כדי לוודא שאכן דרסנו את השיטה הווירטואלית ולא טעינו בהצהרה על השיטה, נוסיף לשיטה הדורסת override בסוף ההצהרה.

virtual int area () override { return (width \* height); }

### קריאה לשיטות וירטואליות משיטות אחרות

אם שיטה f מוגדרת כוירטואלית במחלקת הבסיס ושיטה g במחלקת הבסיס קוראת לה, אזי הגרסה של שיטה f שתיקרא בפועל תלויה בסוג **העצם** בזמן ריצה. אמנם יש לזה שני יוצאי דופן:

* אם השיטה g היא בנאי של מחלקת הבסיס.
* אם השיטה g היא מפרק של מחלקת הבסיס.

בשני המקרים האלה, שיטה f שתיקרא בפועל היא זו של מחלקת **הבסיס**. הסיבה לכך היא בגלל סדר הבניה והפירוק. כשבונים את מחלקת הבסיס - המחלקה היורשת עדיין לא בנויה ולכן מסוכן לקרוא לשיטות שלה, מפני שאולי יש שדות חשובים שעדיין לא מאותחלים. וכן כאשר מפרקים את מחלקת הבסיס - המחלקה היורשת כבר מפורקת, ולכן שוב מסוכן לקרוא לשיטות שלה, מפני שאולי יש שדות חשובים שכבר נמחקו.

### מפרקים וירטואליים

כאשר מגדירים מצביע או רפרנס מסוג מחלקת הבסיס אל אובייקט מסוג המחלקה היורשת, כאשר נקרא למפרק על אובייקט זה הקומפיילר יקרא למפרק של מחלקת הבסיס. אם למחלקה היורשת יהיה גוש זיכרון בערימה שצריך למחוק נקבל זליגת זיכרון. כדי לפתור בעיה זו נגדיר גם את המפרק במחלקת הבסיס כווירטואלי. למרות שהמפרקים בשם שונה לא נקבל שגיאה, כי יש רק מפרק אחד בכל מחלקה. חשוב לשים לב שכאשר מגדירים במחלקת הבסיס מפרק וירטואלי כל המחלקות היורשות ממנה חייבות להגדיר מפרק.

### מאחורי הקלעים

כיצד הקומפיילר יודע עבור כל אובייקט לאיזה גרסה של השיטה הווירטואלית לקרוא?

* + עבור כל מחלקה עם שיטות וירטואליות, מוגדרת **טבלת שיטות וירטואליות VTBL** אחת ויחידה. הטבלה הזאת היא למעשה מערך של מצביעים לפונקציות. עבור כל פונקציה וירטואלית, יש בטבלה הזאת מצביע למימוש שלה בפועל. השיטות הווירטואליות צריכות להיות ממומשות במחלקת הבסיס כדי שטבלה זו תיווצר.
  + לכל אובייקט שנוצר באמצעות בנאי מהמחלקה, הבנאי מכניס לתוכו **מצביע נסתר VPTR** לטבלת השיטות הווירטואליות VTBL.
  + כשאנחנו כותבים קוד שקורא לשיטה וירטואלית, הקומפיילר למעשה כותב קוד שקורא את המצביע לטבלת השיטות הווירטואליות מהאובייקט שנמצא בזיכרון, הולך לכניסה המתאימה בטבלה (לפי שם הפונקציה שקראנו לה), ומפעיל את המימוש המתאים. לדוגמא, ראה מצגת 14 שקופיות 21-22.

בשפות אחרות כגון Java, Smalltalk, Python, כל השיטות וירטואליות. לכן, בכל אובייקט יש מצביע לטבלה הווירטואלית, וכל קריאה לכל שיטה על אובייקט כלשהו עוברת דרך הטבלה הווירטואלית. זה מבזבז גם מקום וגם זמן. לעומת זאת בשפת ++C אפשר לבחור איזה שיטות יהיו וירטואליות ואיזה לא. הבחירה תלויה באופן שבו אנחנו רוצים להשתמש בשיטות. שיטה לא וירטואלית היא מהירה יותר וגם חסכונית יותר בזיכרון, אבל שיטה וירטואלית מאפשרת פולימורפיזם. לכן אך ורק כאשר נרצה לבצע פולימורפיזם על שיטה נגדיר אותה וירטואלית, אחרת נמנע מכך כדי לחסוך זמן וזיכרון. זה מתכתב עם הפילוסופיה של שפת :C++ "לא השתמשת - לא שילמת".

### לסיכום

כדי לבצע פולימורפיזם צריכים להתקיים ארבעה תנאים:

1. השיטה צריכה להיות וירטואלית במחלקת הבסיס.
2. המחלקה היורשת צריכה לדרוס את השיטה במחלקת הבסיס.
3. אנו לא קוראים לשיטה דרך בנאי או מפרק.
4. אנו קוראים לשיטה דרך מצביע או רפרנס מסוג מחלקת הבסיס שמצביע אל עצם המחלקה היורשת.

## שיטות וירטואליות טהורות

לפעמים רוצים להגדיר מחלקה מופשטת, שאין אובייקטים השייכים אליה ישירות. אלא יהיו מחלקות שיורשות אותה ויממשו חלק מהשיטות, או כל השיטות שלה, בהתאם לסוג המחלקה היורשת. למשל מחלקת Shape. אנו רוצים שהמתכנתים שישתמשו בחבילה שלנו, לא יגדירו עצם מסוג Shape אלא יגדירו מחלקות שיורשות אותה, כגון Rectangle, Triangle, Circle וכו', ויממשו את הפונקציה area בהתאם.

הדרך לעשות זאת היא להגדיר במחלקת הבסיס שיטות וירטואליות טהורות, כלומר שיטה וירטואלית ללא מימוש. הקומפיילר **יחייב** אותנו לממש שיטות אלו בכל מחלקה יורשת, אחרת נקבל שגיאת קומפילציה. כדי להגדיר שיטה וירטואלית טהורה יש להחליף את הסוגריים המסולסלות ב-"= 0".

class Shape {

int width, height;

**virtual int area() = 0;**

**virtual ~Shape(){}**

};

### מחלקה אבסטרקטית

מחלקה שכוללת שיטה וירטואלית טהורה נקראת מחלקה אבסטרקטית (מופשטת), ולא ניתן ליצור אובייקטים ממחלקה זו. מספיק שיטה אחת כזו במחלקה כדי שהקומפיילר לא ידע כיצד ליצור ממנה אובייקט. ניסיון ליצור אובייקט ממחלקה אבסטרקטית יגרור שגיאת קומפילציה. כן ניתן ליצור מצביע או רפרנס למחלקה יורשת שאינה אבסטרקטית. מחלקה אבסטרקטית מכילה רק קובץ כותרת hpp.

Shape s; // bad

Shape\* s; // good

בכל מחלקה אבסטרקטית יש להגדיר מפרק וירטואלי ריק, כדי שלא ייווצר מפרק ברירת מחדל שהוא לא וירטואלי, ואז במחיקת אובייקט מסוג המחלקה היורשת יקרא למפרק מסוג מחלקת הבסיס. המפרק צריך להיות ריק כי המפרק במחלקה היורשת קורא לו, לכן חייב לעשות משהו. אם במחלקה האבסטרקטית יש שדות, אז באובייקט מסוג המחלקה היורשת יהיה שדה נסתר מסוג המחלקה האבסטרקטית. אך אם אין שדות, לא יהיה שדה נסתר כזה.

מחלקה שיש בה **רק** שיטות וירטואליות טהורות היא המקבילה ב++C לממשק (interface) ב-Java, אלא שכאן ניתן גם להוסיף שדות למחלקה.

### החלפת שיטות פרטיות

כפי שלמדנו, אם שיטה מוגדרת כפרטית (private) במחלקת הבסיס, המחלקות היורשות לא יכולות להשתמש בה. אבל, המחלקות היורשות עדיין יכולות לדרוס אותה. במקרים מסוימים במחלקת הבסיס יהיו שיטות פרטיות המגדירות את פרטי המימוש הפנימי של המחלקה, ונרצה שבמחלקות היורשות אותה לא יהיה גישה לשיטות אלו, אך כן נרצה לחייב אותן לממש אותן מחדש כדי להתאים אותן למחלקה היורשת. לשם כך במחלקות בסיס אלו נגדיר את שיטות הפרטיות כווירטואליות טהורות, כך שהמחלקות היורשות אינן יכולות לקרוא להן אך יהיו חייבות לממש אותן.

## Pimpl

זוהי תבנית עיצוב שמשתמשת בירושה ושיטות וירטואליות טהורות כדי להסתיר את פרטי המימוש של מחלקה. ב++C השדות הפרטיים מוגדרים בתוך הגדרת המחלקה שנמצאת בתוך קובץ הכותרת (.h) שלה. הלקוחות של המחלקה שלנו המקבלים את קובץ הכותרת, יודעים מה השדות הפרטיים של המחלקה. אמנם לרוב הם לא אמורים לדעת על שדות אלו. כיצד נוכל להסתיר זאת מהם?

דרך מקובלת היא להשתמש בתבנית עיצוב הנקרא Pimpl (pointer to implementation). בתבנית זו בקובץ h יהיו כל השיטות הציבוריות של המחלקה (בלי השדות הפרטיים) מוגדרות כשיטות וירטואליות טהורות במחלקה אבסטרקטית. בקובץ cpp נגדיר מחלקה יורשת שבה יהיה המימוש של כל השיטות והשדות הפרטיים. נוסיף בקובץ h שיטות סטטיות make, שבאמצעותן ניתן יהיה ליצור אובייקט מהמחלקה בקובץ h. כך באמצעות קובץ ה-h בלבד, ובלי ידע של המימוש והשדות הפרטיים, המשתמש יכול ליצור אובייקט ולהפעיל עליו את כל השיטות בקובץ cpp באמצעות פולימורפיזם.

לדוגמה, נניח שרוצים להגדיר רשימה מקושרת שאפשר להוסיף לה איברים, אבל שהלקוחות לא יידעו באיזה שדות אנו משתמשים. אזי בקובץ list.h נגדיר מחלקה בשם List שתכלול רק שיטות וירטואליות טהורות (כגון Add) ללא כל שדות. בנוסף, תהיה לה שיטה סטטית בשם make. בקובץ list.cpp נגדיר מחלקה בשם ListImpl, שתירש את List, תוסיף שדות ותממש את השיטות הווירטואליות הטהורות. שם יהיה גם המימוש של השיטה הסטטית make, שייצור עצם חדש מסוג ListImpl ויחזיר פוינטר עבורו. ראה מצגת 13 שקופית 37.

## מידע והמרה על טיפוסים בזמן ריצה - RTTI

RTTI (Run Time Type Information) הוא מנגנון בC++ המאפשר לתוכנית לקבל מידע בזמן ריצה על טיפוסי אובייקטים. השאלות הבסיסיות עליהן מנסה מנגנון זה לענות הם:

* + האם אובייקט נתון הוא ממחלקה נתונה?
  + האם שני אובייקטים נתונים הם מאותה מחלקה?
  + מהו שם המחלקה של אובייקט נתון?
  + האם ניתן להמיר מצביע לאובייקט אחד למצביע מטיפוס אחר? כלומר, האם קיימת התאמת מצביעים בירושה המאפשרת לבצע המרה בטוחה?

המידע בשאלות 1-3 מסופק באמצעות המחלקה type\_info והאופרטור typeid, ואילו ההמרה בשאלה 4 מבוצעת על ידי האופרטור dynamic\_cast. כדי להשתמש ב-RTTI יש לכלול #include <typeinfo>.

שימו לב: אפשר להגיד לקומפיילר שלא ישמור בכלל מידע על סוגים בזמן ריצה, כך שלא יהיה לנו dynamic\_cast ולא typeid. בקומפיילר של Visual Studio זו ברירת המחדל כדי לחסוך זמן ומקום בזיכרון. אם רוצים להשתמש באפשרויות האלו צריך לשנות את ההגדרות של הקומפיילר.

## type\_info ו-typeid

המחלקהtype\_info  מייצגת את מידע הטיפוס הניתן לשליפה מכל עצם נתון. כדי לקבל אובייקט מסוג type\_info עבור אובייקט כלשהו x, יש להפעיל עליו את האופרטור typeid. הערך המוחזר מהאופרטור הוא אובייקט מסוג type\_info המכיל מידע על x.

typeid(x); // return type\_info of x

הפעולות אותן ניתן לבצע על אובייקט type\_info הם אופרטורי השוואה '==' ו-'!=', כדי להשוות בין שני אובייקטים האם מאותו טיפוס, או להשוות בין אובייקט למחלקה ולגלות האם הוא מופע של מחלקה. בנוסף, ניתן לשלוף את שם המחלקה של אובייקט נתון על ידי הפונקציה name(). לא בהכרח שפונקציה זו תחזיר את שם המחלקה המדויק, אלא איזשהו מזהה ייחודי שלה.

X x1, x2;

if(typeid(x1)==typeid(X)) {…] // return true

if(typeid(x1)==typeid(x2)) {…} // return true

cout << typeid(x1).name() << endl; // print X

אם האובייקט מסוג סטטי המידע ש-typeid משתמש בו נקבע כבר בזמן קומפילציה. אך אם האובייקט הוא פולימורפי, כלומר ירש ממחלקה עם שיטות וירטואליות, אז הקריאה ל-typeid יוצרת פעולה בזמן ריצה הפונה לטבלה הווירטואלית VTBL באמצעות המצביע VPTR, ושולפת משם את טיפוס האובייקט המתאים. אין צורך לעבור על כל עץ הירושה אלא רק ללכת לטבלה הווירטואלית של המחלקה הנוכחית. החיסרון ב-typeid שהוא נותן מידע רק על המחלקה ממנה נבנה האובייקט ולא על המחלקות מהם יורשת או על מחלקות היורשות ממנה. לשם כך נשתמש ב-dynamic\_cast.

נשתמש ב-typeid בדרך כלל למיפוי שגיאות ולא להשוואת טיפוסים. זאת מפני שניתן להשתמש באופרטור dynamic\_cast לאותן פעולות, והוא אף יעיל יותר.

## dynamic\_cast

האופרטור dynamic\_cast משמש להמרה בטוחה של מצביע/רפרנס למחלקת בסיס עם שיטות וירטואליות, אל מצביע/רפרנס למחלקה יורשת שלה. ההמרה תצליח רק אם המצביע מסוג מחלקת הבסיס נוצר מבנאי של הבן אליו אנו ממירים. לדוגמה, אם יש לנו מצביע מסוג Shape\* לאובייקט מסוג Rectangle, אפשר להמיר אותו על ידי המרה דינמית למצביע מסוג Rectangle. אבל, אם ננסה להמיר אותו למצביע מסוג Circle, האופרטור יחזיר null.

Shape \*s = Rectangle();

Rectangle \*r = dynamic\_cast<Rectangle \*>(s);

כשמשתמשים באופרטור זה, הקומפיילר מכניס בדיקה המתבצעת בזמן ריצה, הפונה לטבלה הווירטואלית VTBL באמצעות המצביע VPTR, שם היא בודקת אם האובייקט שמצביעים אליו אכן מתאים לסוג שאליו רוצים להמיר. לכן, אפשר לבצע המרה דינאמית רק אם האובייקט הוא ממחלקה שיש לה טבלת שיטות וירטואליות. אחרת הקומפיילר לא ייתן לנו להשתמש ב-dynamic\_cast, ותתקבל שגיאת קומפילציה. בדיקה זו היא "יקרה" בזמן ריצה, מפני שצריכה לעבור על כל עץ הירושה כדי לקבוע אם ההמרה תקינה או לא.

הצורך בהמרה דינאמית הוא כאשר יש לנו מצביע/רפרנס למחלקת הבסיס, ואנו רוצים להפעיל עליו שיטות המוגדרות במחלקה יורשת, במידה והוא אכן מופע של המחלקה היורשת. נוכל לעשות זאת באמצעות typeid, אך בדרך זו תתעורר בעיה כאשר למחלקה היורשת גם יש מחלקה שיורשת ממנה. במקרה זה אם המצביע המקורי הוא מסוג מחלקת הנכד, הבדיקה typeid תחזיר תשובה שלילית למרות שאכן מוגדרת בו שיטה זו. לעומת זאת, המרה דינאמית בודקת את כל עץ הירושה ולכן תדע לזהות זאת. לדוגמא: עבור היררכיית המחלקות Shape: Rectangle: Square

Shape \*s = Square();

if(typeid(\*s) == typeid(Rectangle)) // return false

((Rectangle\*)s)->area();

Rectangle \*r = dynamic\_cast<Rectangle \*>(s);

if(r) // return true

r->area();

שימוש נוסף להמרה דינאמית הוא לדמות את אופרטור instanceof ב-Java.

### המרה דינאמית לרפרנס

האופרטור dynamic\_cast מאפשר לבצע גם המרת רפרנס לאובייקט. אלא שכאן אם ההמרה נכשלת, בגלל שלא ניתן להחזיר null לרפרנס, נזרקת חריגה bad\_cast. לכן השימוש שלנו בהמרת דינאמית לרפרנס יהיה באמצעות try ו-catch.

try {

Rectangle &r = dynamic\_cast<Rectangle &>(s);

     r.area();

} catch(bad\_cast) {  // cast failed

cout << "cannot cast: r is not of type Rectangle" << endl;

}

## המרות בזמן קומפילציה

### const\_cast

האופרטור const\_cast משמש להמרת מצביע/רפרנס קבוע (const) למצביע/רפרנס רגיל (לא const). הוא למעשה אומר לקומפיילר להתעלם מהבדיקה של const. בדרך כלל, שימוש באופרטור זה מראה על שגיאה בתכנון המחלקה. צריך לתכנן את המחלקה נכון כך שכל מה שצריך להיות const אכן יהיה const. בכל זאת יש בו שימוש כשאנחנו מקבלים קוד ישן של מתכנת שעשה שגיאה ולא סימן פונקציה מסוימת כ-const, למרות שבפועל היא לא משנה את האובייקט שהיא מקבלת כפרמטר. לכן לא נוכל להפעיל פונקציה זו על אובייקט שהוא כן קבוע. אם יש לנו גישה לקוד המקור, נשנה אותו ונסמן את הפונקציה כ-const. אבל לפעמים אין לנו גישה לקוד המקור אלא רק לקובץ הבינארי המקומפל. במקרה זה, נשתמש ב-const\_cast על האובייקט כדי שנוכל להפעיל עליו את הפונקציה. מומלץ כאמור להימנע מאופרטור זה ככל האפשר.

const int\* cp;

int\* p = const\_cast<int\*>(cp);

### reinterpret\_cast

האופרטור reinterpret\_cast משמש להמרת פוינטר/רפרנס מסוג אחד לסוג אחר. האופרטור לא מבצע שום בדיקה, לא בזמן קימפול ולא בזמן ריצה. כמו const\_cast, הוא למעשה אומר לקומפיילר להתעלם מבדיקות הטיפוסים הרגילות שהוא מבצע, ולסמוך עלינו שאנחנו יודעים מה אנחנו עושים. דומה לאופרטור (type\*). זה מסוכן ועלול לגרום לשגיאות לוגיות וערכים לא מוגדרים, אמנם בכל זאת ישנם מקרים שנצטרך אותו. לדוגמה, כאשר רוצים לכתוב מערך של RGB לקובץ בינארי כדי ליצור תמונה, נצטרך להמיר אותו באמצעות reinterpret\_cast ל-char\*, כי פונקציה write מצפה לקבל טיפוס מסוג זה.

imageFile.write(reinterpret\_cast<char\*>(&image), 3\*dimx\*dimy);

### static\_cast

האופרטור static\_cast משמש להמרה בין שני סוגים שהמידע כיצד להמיר אותם ידוע בזמן קומפילציה. לדוגמה, המרה מ-double ל-int מתבצעת על ידי פעולה ידועה של לקיחת החלק השלם בלבד. מידע נוסף על כיצד להמיר יכול להיות אופרטור המרה שהגדרנו.

כל המרות מסוג זה יכולות להתבצע גם באמצעות סוגריים לפני האובייקט, כמו שמקובל בשפת C, אך המרת static\_cast עדיפה משתי סיבות:

* 1. קל יותר למצוא המרות בקוד, פשוט מחפשים את המחרוזת static\_cast.
  2. ההמרה בעזרת סוגריים עלולה להיות שגויה. למשל, בעזרת סוגריים אפשר להמיר מצביע double למצביע int. ההמרה זו שגויה והתוצאה תהיה זבל. לעומת זאת, המרה סטטית תציל אותנו מכך ותכריז על שגיאת קומפילציה, מפני שאין המרה סטטית המאפשרת להמיר מצביע ממשי למצביע לשלם:

double\* dp;

int\* ip = (int\*)dp; // hold garbage

int\* ip = static\_cast<int\*>(dp); // compilation error

המרה סטטית של מחלקות:

* + - * static\_cast חוסם המרות בין מצביעי מחלקות שאין ביניהן קשר, אף אחת לא יורשת מהשנייה.
      * static\_cast מאפשר המרה של מצביע למחלקה יורשת אל מצביע למחלקת הבסיס. המרה כזאת היא בטוחה ואף מתבצעת אוטומטית גם בלי static\_cast.
      * static\_cast מאפשר גם המרה של מצביע למחלקת בסיס אל מצביע למחלקה יורשת. המרה כזאת היא מסוכנת, משום שהיא נכונה רק אם המצביע המומר אכן הצביע לעצם מהסוג של המחלקה היורשת. לא מתבצעת כל בדיקה לנושא זה ולכן זו סכנה. עדיף במקרה זה להשתמש ב-dynamic\_cast.

כשמשתמשים ב-static\_cast, בדרך-כלל הקומפיילר מכניס פקודה כלשהי שתתבצע בזמן ריצה. לעומת זאת ב-const\_cast או reinterpret\_cast, הקומפיילר לא מכניס שום פקודה לביצוע בזמן ריצה. ההוראה משפיעה על הקומפילציה בלבד.