גירסה 2.00 – 4.10.2004

C++ Tutorial

מסמך זה הורד מהאתר http://underwar.livedns.co.il.

Nir Adar

Email: underwar@hotmail.com

Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

תוכן עניינים

תוכן עניינים **3**

הקדמה **4**

הכרות ראשונה עם מחלקות **5**

הרשאות **9**

תרגיל 12

קלט ופלט בשפת C++ **14**

פונקציות חופפות **15**

הקצאת זיכרון דינמית **16**

DEFAULT PARAMETERS **17**

משתני התייחסות **19**

הגדרת משתנים, CONSTRUCTORS & DESTRUCTORS **21**

הגדרת משתנים 22
CONSTRUCTORS & DESTRUCTORS 22
תחביר בסיסי ומוטיבציה *22*
מחלקות בעלות מספר בנאים, רשימות אתחול *26*
Default Constructor *29*

דוגמא מסכמת **31**

סיום **34**

הקדמה

במסמך זה נסקור את שפת C++. מסמך זה מיועד לאנשים הבקיאים היטב בשפת C. המסמך אינו מתיימר להיות מדריך שלם לכל שפת C++, אלא מיועד להעביר בצורה טובה את העקרונות והשיטות לגשת לתכנות ב-C++.
מסמך זה הוא שדרוג של המסמך הישן שהיה בפרויקט UnderWarrior, והוא למעשה מצב ביניים של המטרה אליה אני שואף להגיע. בימים אלו אני משפר את המסמך על מנת שיהווה מדריך שלם לשפה. שינוי זה הולך לשנות את אופי המסמך, את סדר הנושאים בו, וכדו'. גירסה זו של המסמך היא הרחבה של המסמך המקורי שכתבתי, שעדיין שומרת על אופיו. לדעתי הזווית המצומצמת שמסמך זה מציג, גם לה יש מקום, ולכן אני משחרר מסמך זה כמסמך נפרד.

אנשים רבים מתייחסים אל שפת C++ כהרחבה של שפת C. למעשה שפות אלו שונות ביותר אחת מהשניה. C++ היא שפה המציגה תפיסה חדשה ושונה לחלוטין מזו של שפת C. הסיבה להתייחסות לעתים לשפת C++ כהרחבה של C, היא הסיבה שכמעט כל תוכנה שנכתבה עבור C, תרוץ גם בשפת C++. המקור הוא היסטורי – מתכנני C++ רצו שיהיה ניתן להשתמש בקוד שכבר היה קיים בשפת C גם בשפה החדשה, ולכן תכננו אותה תואמת ככל הניתן לשפת C, בלי לפגוע בהגבלות וברעיונות של C++.
מדוע אם כך לא כל תוכנית בשפת C תרוץ בשפת C++? הסיבה היא שבשפת C++ הוספו מילים שמורות (keywords) חדשות, שלמתכנת אסור להשתמש בהן בתור משתנים, כגון new, delete ועוד. תוכניות בשפת C בהן נעשה שימוש בשמות מזהים שב C++ הם מילים שמורות לא יעברו קומפילציה.

במסמך זה נכיר את שפת C++ בעזרת מספר רב של דוגמאות.
רוב הנושאים יוצגו בעזרת דוגמא, ולאחריה הסבר.
הדגשנו במסמך את צורת החשיבה הדרושה לתכנות נכון ב C++. במספר מקומות במסמך - ראשית אנו מציגים קטע קוד ב-C, מציגים בעייתיות הקיימת בו, ורואים כיצד הכלים השונים ב C++ פותרים את הבעיה הנדונה.

הכרות ראשונה עם מחלקות

שפת C++ בנויה מסביב **למחלקות** (class).

מחלקה היא הרחבה של מושג המבנים (structs) הקיים בשפת C.
מחלקה היא למעשה הגדרת טיפוס חדש בו מרוכזים פונקציות ומשתנים תחת שם אחד. לאחר שהגדרנו מחלקה, נוכל ליצור אובייקטים מסוג המחלקה שהגדרנו, ולהשתמש בהם בתוכניות שלנו.

רעיון המחלקות הוא ניסיון לקרב את שפת התכנות למחשבה האנושית.
אנו מגדירים עצמים ואוסף פעולות עליהם.
למשל, נוכל להגדיר עצם מסוג בן אדם. נגדיר (בעזרת פונקציות) פעולות כגון הליכה, דיבור וכדומה. לאחר שהגדרנו אדם, נוכל להגדיר למשל גם זמר, שיקבל את כל התכונות של העצם אדם, ולהוסיף עליהן תכונות ופעולות נוספות, למשל, שירה.

נראה במסמך זה כיצד אנו נעביר רעיונות כאלו ואחרים אל שפת התכנות.

נתחיל בהכרת C++, ונפתח בדוגמא:

```
class MyClass
{
	public:
		int i;
		void func1();
		int Start();
};

class Car
{
	public:
		int Drive();
};

class Horse
{
	public:
		int Drive();
};

int main()
{
	MyClass a;
	Car Volvo;
	Horse Moshe;
	a.i = 3;
	a.func1();
	Moshe.Drive();
	Volvo.Drive();
	return 0;
}
```

נביט בקטע מהדוגמא:

```
class MyClass
{
	public:
		int i;
		void func1();
		int Start();
};
```

התחלנו קטע זה בהגדרת מחלקה חדשה בשם `MyClass`.
לאחר מכן, המילה השמורה `public` מגדירה את ההרשאות שיהיו למשתנים ולפונקציות שיופיעו אחריה. בנושא ההרשאות נעסוק מאוחר יותר. רק נציין שבדרך כלל לא נהוג להגדיר משתנים בשטח ה-`public` של המחלקה. כרגע כתבנו את המשתנה תחת ההרשאה `public` רק כדי לשמור על פשטות הדוגמא.
לאחר מכן הגדרנו שהמחלקה `MyClass` מכילה משתנה בשם `i` מסוג `int`, ושתי פונקציות - `func1()` ו-`Start()`.
נשים לב לנקודה הבאה: כאשר הגדרנו מחלקה, הגדרנו למעשה תבנית, ולא יצרנו משתנים. בדומה ל-`struct` בשפת `C`, כאשר מגדירים `struct` אנחנו לא מקצים זיכרון עבור המשתנים. הזיכרון יוקצה רק עבור כל משתנה שניצור מסוג ה-`struct` (ובמקרה שלנו עבור כל משתנה שניצור מסוג של המחלקה).

צורת דיבור: כאשר, למשל, אנחנו כותבים את הקוד הבא:

```
int i;
```

אנחנו אומרים שהגדרנו *משתנה* בשם `i`, *שהסוג* שלו הוא `int`.
כאשר נגדיר משתנה שהוא מסוג מחלקה, נכנה את המשתנה בשם **אובייקט**.
לדוגמא:

```
MyClass a;
```

בדוגמא זו יצרנו אובייקט בשם `a` שסוגו הוא המחלקה `MyClass`.

נזכור זאת להבא: כאשר מדברים על מחלקה, מדברים על התבנית שהגדרנו. כאשר מדברים על אובייקט, אנו מתייחסים לעותק של התבנית, הנמצא בזיכרון.

```
class Car
{
	public:
		int Drive();
};

class Horse
{
	public:
		int Drive();
};
```

בקטע קוד זה הגדרנו שתי מחלקות, אחת בשם `Car` והשניה בשם `Horse`. בכל אחת מהן קיימת פונקציה בשם `Drive()`.
נשים לב שעבור כל מחלקה, יצרנו פונקציה `Drive()` שונה.
בהצהרה זו הכרזנו על פונקציה `Drive()` ששייכת למחלקה `Car` ועל פונקציה `Drive()` ששייכת למחלקה `Horse`.

```
int main()
{
	MyClass a;
	Car Volvo;
	Horse Moshe;
	a.i = 3;
	a.func1();
	Moshe.Drive();
	Volvo.Drive();
	return 0;
}
```

כמו ב-`C`, הפונקציה ממנה מתחילה התוכנית לרוץ היא `main()`.
עם תחילת התוכנית, אנו יוצרים אובייקט מסוג `MyClass` בשם `a`, אובייקט בשם `Volvo` מסוג `Car` ואובייקט מסוג `Horse` בשם `Moshe`.
לאחר מכן, ניגשנו, בדומה לצורה בה אנו ניגשים לשדות ב-`struct`, אל השדה `i` הנמצא באובייקט `a` ושמנו בו את הערך 3. לאחר מכן קראנו לפונקציה `func1()` השייכת למחלקה `MyClass` על האובייקט `a`.
כאשר השתמשנו פעמיים בפונקציה `Drive()`, פעם על `Moshe` ופעם על `Volvo`, קראנו לשתי פונקציות. בפעם הראשונה קראנו לפונקציה של המחלקה `Horse` ובפעם השניה קראנו לפונקציה של המחלקה `Car`.

הערה: בדוגמא זו לא הצגנו כיצד ממשים את הפונקציות המוגדרות בתוך המחלקות, אלא רק כיצד מצהירים עליהן.

נביט בדוגמא נוספת ונראה כיצד ניתן לממש פונקציות השייכות למחלקה.

```
class Rectangle
{
	public:
		int x, y;
		int getArea()
		{
			return x*y;
		}
};

class Circle
{
	public:
		int x, y, radius;
		int getArea();

};
```

```
int Circle::getArea()
{
	return PI*radius*radius;
}
```

(אנו מניחים כי PI הוא קבוע שהוגדר במקום כלשהו בתוכנית).

ניתן לממש פונקציה בתוך הגדרת המחלקה, או לחילופין לממש אותה בקטע קוד נפרד מחוץ למחלקה. נביט במחלקה הראשונה בדוגמא:

```
class Rectangle
{
	public:
		int x, y;
		int getArea()
		{
			return x*y;
		}
};
```

הפונקציה getArea() ממומשת מיד אחרי ההצהרה עליה. היא מחזירה את המכפלה של x ב-y. נשים לב לדוגמא לשימוש בפונקציה:

```
Rectangle rec1, rec2;
int area1, area2;
...
area1 = rec1.getArea();
area2 = rec2.getArea();
```

בקריאה הראשונה הפונקציה פעלה על ה-x וה-y השייכים ל-rec1 ובקריאה השני היא פעלה על אלו השייכים ל-rec2.

כעת נביט בצורה השניה למימוש פונקציות:

```
class Circle
{
	public:
		int x, y, radius;
		int getArea();

};

int Circle::getArea()
{
	return PI*radius*radius;
}
```

בדוגמא זו הצהרנו על הפונקציה getArea() בתוך המחלקה Circle, ומימשנו אותה מחוץ למחלקה. הכוונה בצורת הכתיבה הבאה: Circle::getArea() היא שאנו מממשים את הפונקציה getArea **ששייכת** למחלקה Circle. כבר ראינו שאותו שם לפונקציה יכול לשמש במחלקות שונות. לכן כשאנו באים לממש את הפונקציה מחוץ למחלקה, אנו צריכים להגיד במפורש לאיזו מחלקה פונקציה זו שייכת.

הרשאות

כאשר עבדנו בשפת C, עבדנו לעיתים קרובות על ידי הגדרת ADTs - Abstract Data Types.
הרעיון היה כזה: הגדרנו מבנה נתונים (struct) ואז הגדרנו פונקציות שפעלו עליו, והיו למעשה פעולות שמיועדות למבנה הנתונים שיצרנו.
ב-C++ למעשה הפכנו את מודל מבנה הנתונים המופשט לחלק מהשפה.
כאשר אנו שמים למעשה את הפונקציות בהגדרת המחלקה אנחנו אומרים: הפונקציה X הנמצאת במחלקה Y, שייכת למחלקה Y, והיא תעבוד על אובייקטים הנמצאים במחלקה. יתכן ש-X תקבל פרמטרים ויתכן שלא, אולם תהיה לה גישה בכל מקרה לאובייקט עליו היא פועלת. למשל, בדוגמא הקודמת, כאשר כתבנו Moshe.Drive() הפעלנו את הפונקציה Drive() על האובייקט Moshe, מבלי להעביר את Moshe כפרמטר.
בצורה זו C++ מחברת את רעיון ה-ADT לשפה.

אחת הבעיות הגדולות שהיו עם ADT, היא שלמרות שהגדרנו פונקציות בהן המשתמש צריך להשתמש כדי לעבוד עם מבנה הנתונים שיצרנו, לא הייתה מניעה שהוא ייגש ישירות למשתנים וישנה אותם. במקרה כזה, יתכן שהאובייקט היה מגיע למצב לא תקף בעקבות שינויי המשתמש, והפונקציות שיועדו לפעול עליו היו נכשלות.
בעיה נוספת, גדולה אף יותר, היא שבכך שהמשתמש ניגש ישירות לשדות של מבנה הנתונים, הוא נעשה תלוי במימוש של מבנה הנתונים. שינוי במימוש במקרה כזה ידרוש שינוי בקוד שהתבסס עליו.
C++ באה לעזור לנו במקרה זה. מנגנון ההרשאות ב-C++ בא למנוע מהמשתמש לגשת לחלק מהשדות של מבנה הנתונים, שלא בעזרת הפונקציות שמתכנת המחלקה בחר.
ישנם 3 סוגי הרשאות ב-C++: public, private ו-protected.
נציג כעת את שני הסוגים הראשונים, ובהרשאה מסוג protected ניתקל עוד בהמשך.

```
class permissionClass
{
	public:
		int iNumber1;
	private:
		int iNumber2;
};

int main()
{
	permissionClass per;
	per.iNumber1 = 10;
	per.iNumber2 = 10;
	return 0;
}
```

הרשאה מסוג `public` היא ההרשאה שראינו עד כה. הרשאה זו אומרת: כל אחד יכול לגשת למשתנה או לפונקציה המוגדרת כ-`public` ולעשות בעזרתה ככל העולה על רוחו.
לעומתה, הרשאה מסוג `private` אומרת: אל המשתנים או הפונקציות המוגדרים כ-`private`, יוכלו לגשת רק פונקציות השייכות לאותה מחלקה.

בדרך כזו נוכל להפריד בין הממשק אל המשתמש לבין המימוש של המחלקה:
הממשק יהיה בחלק ה-`public` של המחלקה, ופונקציות ומשתנים הקשורים למימוש יהיו ב-`private`.

בדוגמא לעיל, `per.iNumber1 = 10;` היא שורה חוקית, כי `iNumber1` נמצא תחת ההרשאה `public` במחלקה. לעומתה, השורה `per.iNumber2 = 10;` איננה שורה חוקית, כי `iNumber2` איננו נמצא בחלק ה-`public` של המחלקה.
שורה זו לא תעבור כלל קומפילציה.

נביט בדוגמא נוספת, ונתחיל לראות כיצד משתמשים בהרשאות השונות.

```
#include <iostream.h>

class MyClass
{
      private:
            int j;
      public:
            int func1(int i);
            void print() { cout << j << endl; }
};

int MyClass::func1(int i)
{
      j = i;
      return j;
}

int main()
{
      MyClass class1, class2;

      // The following line is illegal:
      class1.j = 10;

      class1.func1(10);
      class2.func1(20);
      class1.print();
      class2.print();
      return 0;
}
```

הספרייה `iostream.h` היא המחליפה של הספרייה `stdio.h` משפת `C`, והיא מכילה מחלקות שונות לקלט ופלט.
הפונקציה `print` הממומשת בתוך המחלקה `MyClass` משתמשת באובייקט `cout` על מנת לשלוח פלט למסך, ולכן היינו צריכים לעשות `include` לספריה זו.

השימוש באובייקט `cout` זוהי הדרך שלנו בשפת `C++` לשלוח פלט למסך.
`endl` הוא המחליף ל`'\n'` של `C`.

ב-`MyClass` יצרנו משתנה עם הרשאה `private` בשם `j`.
כמו כן יצרנו את הפונקציה `print()` שמדפיסה את ערכו, וכן יצרנו פונקציה בשם `func1()` שמקבלת מספר ומחליפה את ערכו של `j` במספר זה.
התוכנית מדגימה כיצד אנו יוצרים שני אובייקטים מסוג `MyClass`.
איננו יכולים להציב לתוך המשתנה `j` בתוך כל אחד מהם ערכים ישירות, (עקב ההרשאה שנתנו למשתנה זה), לכן אנו משתמשים בפונקציה `func1()` כדי לקבוע את ערכי `j`.
בתחילה תהליך זה נראה מסורבל משהו, הרי בסופו של דבר בכל מקרה אנו משנים את המשתנה `j`, אז למה לעשות את זה דרך פונקציה ולא ישירות?
יש לכך שתי סיבות חשובות:
סיבה ראשונה, המוכרת למתכנתי `C` שעבדו עם `ADT`, היא הצורך להפריד בין הממשק של התוכנית למימוש שלה.
הפונקציה `func1()` שייכת ל-`public` של הפונקציה, כלומר לממשק.
אנו מניחים שכאשר קוראים לה היא משנה משתנה בתוך המחלקה, וכאשר קוראים ל-`print()` משתנה זה מודפס.
אם המשתמש משתמש רק בפונקציות אלו ולא ניגש ישירות ל-`j`, נוכל למשל לשנות את שם המשתנה `j`, או לשנות פרטי אחרים הקשורים למימוש, ותוכניתו של המשתמש במחלקה לא תושפע.
לעומת זאת אם המשתמש במחלקה ניגש ישירות לשדות שלה ואז נשנה את המחלקה, גם תוכניות המשתמש ידרשו שינוי.
סיבה שניה, חשובה לא פחות להרשאות, היא ביקורת על המידע.
נניח שנרצה לכתוב מחלקה שתתאר אדם.

```
class Person
{
	private:
		char *name;
		int age;
	public:
		void SetAge(int newAge);
};
```

נרצה להיות מסוגלים לשנות את גילו של האדם.
פתרון אפשרי יכול להיות הפונקציה הבאה:

```
void Person::SetAge(int newAge)
{
	age = newAge;
}
```

אולם, נוכל גם להציע את הפתרון הבא:

```
void Person::SetAge(int newAge)
{
      if (newAge > 0) age = newAge;
}
```

כאן אנו מתחילים לראות את היתרון שמשתנה private נתן לנו.
פונקציה זו משנה את גילו של האדם רק אם הגיל החדש שרוצים לקבוע לו גדול מ-0, כי הרי לא אפשרי שלאדם יהיה גיל שלילי.
מכיוון שכל שינוי של המשתנה age צריך לעבור דרך הפונקציה SetAge(), אנו יכולים לבדוק ולהיות בטוחים שגילו של האדם הוא גיל תקף כל הזמן.

נושא שיכול לבלבל בהתחלה הוא מתי ראוי להשתמש בכל אחת מההרשאות. האם המשתנה יכול להיות public? private? ומה לגבי פונקציות? מה לגבי ההרשאה הנוספת שאמרנו שקיימת, protected? הכלל הבא יסכם את הנושא:

כלל אצבע לגבי הרשאות

- משתני המחלקה יוגדרו תמיד בתור private.
- פונקציות הנגישות למשתמש יוגדרו public, ופונקציות פנימיות ימוקמו לרוב כ-private.

המשתמש אף פעם אינו צריך לגשת אל משתני המחלקה. אלו שייכים למימוש ולכן נמקם אותם כפרטיים. לכל משתנה שהוא private, במידה ונרצה לתת למשתמש גישה אליו, נגדיר פונקציה שתחזיר את ערכו למשתמש (get) ופונקציה שתאפשר למשתמש להזין ערך חדש עבור המשתנה (set).

תרגיל

כתוב מחלקה בשם Point המייצגת נקודה במישור. לנקודה ערכי x ו-y שלמים אותם ניתן לקבל ולקבוע. כמו כן ניתן לקבל את מרחק הנקודה מהראשית.

פתרון לתרגיל מוצג בעמוד הבא.

פתרון אפשרי לתרגיל:

```
#include <math.h>

class Point
{
	public:
		int GetX();
		int GetY();

		void SetX(int new_x) { x = new_x; };
		void SetY(int new_y) { y = new_y; };

		double Length();
	private:
		int x, y;
};

int Point::GetX()
{
	return x;
}

int Point::GetY()
{
	return y;
}

double Point::Length()
{
	return sqrt(x*x+y*y);
}
```

יש לציין כי מימוש נכון ומלא של מחלקה זו הינו שונה מעט מהפתרון המוצג לעיל.
הפתרון המוצע מתבסס על הידע אותו הצגנו עד כה במסמך.

קלט ופלט בשפת C++

נציג מספר דוגמאות נוספות לשימוש ב-cout, אוביקט המאפשר גישה אל הפלט של שפת C++.
נניח למשל שנרצה להדפיס שלושה מספרים אחד אחרי השני, נוכל לכתוב את הקוד הבא:

```
int x = 4, y = 5, z = 6;
cout << x << y << z;
```

על המסך יודפס: "456".

אם נרצה להדפיס גם רווחים בין המספרים, נעשה זאת כך:

```
int x = 4, y = 5, z = 6;
cout << x << " " << y << " " << z;
```

או כך:

```
int x = 4, y = 5, z = 6;
cout << x << ' ' << y << ' ' << z;
```

נסכם: האובייקט cout שולח פלט אל הפלט הסטנדרטי. ניתן לשרשר מספר איברים להדפסה, ולהדפיסם בשימוש אחד באובייקט cout, על ידי שימוש באופרטור << בין איבר לאיבר. האובייקט cout מזהה את סוגי המשתנים שנשלחים אליו. אין צורך להעביר (כמו ב-printf() ב-C) מהו הסוג של כל משתנה אותו רוצים להדפיס.
שפת C++ מאפשרת לנו את התנהגותו של האופרטור << כך שנוכל בעזרת cout להדפיס גם אובייקטים מסוג של מחלקות שאנו ניצור.

קלט בשפת C++ מתבצע על ידי האובייקט cin, וצורת השימוש בו מזכירה קצת את השימוש ב-cout. הקוד הבא, לדוגמא, קולט מספר שלם מן המשתמש לתוך המשתנה i:

```
int i;
cin >> i;
```

פונקציות חופפות

אחת מההרחבות של C++ לשפת C, היא הוספת פונקציות חופפות לשפה.
בשפת C++ ניתן להגדיר פונקציות בעלות אותו שם, אך בעלות פרמטרים פורמליים שונים. פונקציות אלו נקראות פונקציות חופפות, והקומפיילר מתייחס אליהן כאל פונקציות נפרדות. לדוגמא:

```
void func1(char c)
{
	cout << "Char: " << c << endl;
}

void func1(char *sz)
{
	cout << "String: " << sz << endl;
}

int main()
{
	func1("Hello");
	func1('W');
	return 0;
}
```

התוכנית תדפיס:

```
String: Hello
Char: W
```

בקריאה הראשונה לפונקציה העברנו לה מחרוזת, לכן נקראה הפונקציה המטפלת בסוג הנתונים char *. בפונקציה השניה העברנו תו, ולכן נקראה הפונקציה המקבלת char.

יתכנו מקרים בהם הקומפיילר לא ידע לאיזה מהפונקציות החופפות לפנות.
נביט למשל בדוגמא הבאה:

```
void func1(float f);
void func1(double d);

int main()
{
	int x = 10;
	func1(x);
	return 0;
}
```

התוכנית הנ"ל לא תעבור קומפילציה. כאשר מעבירים לפונקציה פרמטרים מסוג שונה מזה שהפונקציה מקבלת, מתבצעת המרה של הפרמטר הנשלח לזה שהפונקציה מצפה לקבל. אולם, במקרה לעיל הקומפיילר לא ידע להחליט לאיזה סוג משתנה להמיר את x - ל-float או double.
לפיכך, אם ננסה לקמפל את הקוד הנ"ל נקבל הודעת שגיאה שתודיע על כך.

הקצאת זיכרון דינמית

אחת האפשרויות החשובות בכל שפת תכנות, היא האפשרות להקצות לתוכנית זיכרון נוסף בזמן ריצה.
ב-C, השתמשנו בפונקציה `malloc()` על מנת להקצות זיכרון, ובפקודה `free()` על מנת לשחרר זיכרון זה.
ב-C++, נשתמש במילים השמורות `new` ו-`delete` על מנת להקצות ולשחרר זיכרון.

ה-`syntax` של `new`:

```
<pointer> = new <requested type>;
```

ואם נרצה להקצות מערך של אלמנטים:

```
<pointer> = new <requested type>[number of items];
```

למשל, נניח שנרצה להקצות משתנה מסוג `int`, נכתוב:

```
int *pi;
pi = new int;
```

אם נרצה להקצות מחרוזת באורך 20, נכתוב:

```
char *szString;
szString = new char[20];
```

זיכרון שהוקצה באופן דינמי, יש לשחרר במפורש, על ידי `delete`.

ה-`syntax` של `delete`:

```
delete <pointer>;
```

ובמקרה שנרצה לשחרר מערך שהוקצה:

```
delete []<pointer>;
```

למשל, על מנת לשחרר את הזיכרון אותו הקצנו בדוגמא הקודמת בעזרת `new`, נכתוב:

```
delete pi;
delete []szString;
```

Default parameters

אחת האפשרויות החדשות, שלא היו קיימות בשפת C, ששפת C++ מספקת לנו, היא האפשרות לתת ערך ברירת מחדל לפרמטרים של פונקציות.
אם פרמטר כלשהו של פונקציה בדרך כלל מקבל ערך קבוע, ורק לעיתים רחוקות נרצה לשים ערך במקומו, נוכל להשתמש במנגנון זה.

נביט בדוגמא הבאה:

```
#include <iostream.h>

void func1(int i = 10)
{
	cout << i << endl;
}

int main()
{
	func1(23);
	func1();
	return 0;
}
```

פלט התוכנית יהיה: "23" ושורה לאחר מכן "10".
הפונקציה func1 מקבלת פרמטר בשם i. ערך ברירת המחדל שלו נקבע להיות 10.
כאשר אנו קוראים לפונקציה עם הערך 23, אנו מדפיסים את המספר 23 על המסך ומתעלמים לגמרי מערך ברירת המחדל. כאשר אנו קוראים לפונקציה שורה לאחר מכן ללא כל פרמטר, אזי הפונקציה משתמשת בערך ברירת המחדל של i ומדפיסה על המסך 10.

כאשר פונקציה מקבלת מספר ערכים, אשר לחלקם ישנה ברירת מחדל ולחלקם אין ברירת מחדל, יופיעו המשתנים ללא ברירת המחדל ראשונים.
לדוגמא, הפונקציה הבאה איננה כתובה בצורה חוקית:

```
void PrintNumbers(int i2 = 10, int i3 = 20, int i1)
{
	cout << i1 << " " << i2 << " " << i3 << endl;
}
```

הפונקציה הבאה הינה חוקית:

```
void PrintNumbers(int i1, int i2 = 10, int i3 = 20)
{
	cout << i1 << " " << i2 << " " << i3 << endl;
}
```

נביט בפונקצית ה-`main()` הבאה המשתמשת בפונקציה `PrintNumbers`:

```
int main()
{
	PrintNumbers(10);
	PrintNumbers(10, 20);
	PrintNumbers(10, 20, 30);
	PrintNumbers();
	return 0;
}
```

כל השורות, מלבד השורה המודגשת באדום, יעברו קומפילציה. השורה המודגשת לא תעבור קומפילציה, מכיוון שלא הגדרנו ערך ברירת מחדל ל-`i1`, ולכן אי אפשר לקרוא לפונקציה ללא העברת ערך אליו.

פלט התוכנית ללא השורה הבעייתית:

```
10 10 20
10 20 20
10 20 30
```

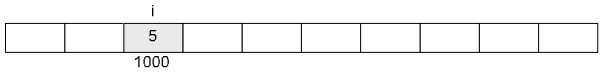
גם לפונקציות של מחלקה ניתן להגדיר ערכי ברירת מחדל. נראה זאת בהמשך המסמך.

משתני התייחסות

אלמנט שפה חדש נוסף בשפת C++ הוא **משתני התייחסות** – **references**.
ניתן לחלק את אלמנטי המידע בשפת C לשתי קטגוריות: משתנים ומצביעים.
משתנה הכיל מידע, ומצביע הינו משתנה שמכיל כתובת בזכרון, בה נמצא משתנה אחר.

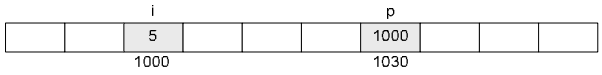
נביט איך זה נראה בזכרון. נניח כי הגדרנו משתנה i מסוג מספר שלם. ל-i הוקצאו מספר תאים בזכרון (לצורך פשטות נניח כי הוקצה תא אחד). נניח, למשל, של-i הוקצאה הכתובת 1000 בזכרון.

```
int i = 5;
```



כעת נגדיר מצביע p, שיצביע אל i. מצביע הוא משתנה בעצמו, ולכן יש לו כתובת.
נניח כי p קיבל את הכתובת 1030. p יצביע אל i, ולכן p יכיל את הערך 1000.

```
int *p = &i;
```



C++ מציגה צורת עבודה נוספת - משתני התייחסות. **משתנה התייחסות** הינו שם נוסף למשתנה קיים. לא יוקצה זכרון נוסף עבור משתנה ההתייחסות. שינוי של משתנה ההתייחסות ישנה את המשתנה המקורי.

מבחינת תחביר, משתנה התייחסות יוגדר כך:

```
<type>& <reference_name> = var_name;
```

לדוגמא:

```
int &j = i;
```

שינוי של j יגרום לשינוי של i, ולהפך.
נביט כעת בדוגמא פשוטה לשימוש במשתני התייחסות:

```
#include <iostream.h>

int main()
{
	int i = 5;
	int &j = i;

	j = 7;
	cout << i << endl;

	return 0;
}
```

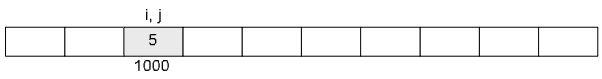
פלט התוכנית יהיה 7.

משתני התייחסות יכולים להיות פרמטרים של פונקציות, וכן ערכים מוחזרים של פונקציות.
יתרונות השימוש במשתני התייחסות:

1. נוחות שימוש: השימוש בהם הינו כמו במשתנה לוקלי, ולא צריך להסתבך עם התחביר של מצביעים.
2. יעילות: לא נוצר העתק של המשתנה המועבר אל הפונקציה (פעולה שיכולה להיות יקרה), אלא ממש המשתנה מועבר.

נשים לב כי לאחר שהגדרנו את `j` להתייחס אל `i`, לא ניתן לגרום לו להתייחס למשתנה אחר. שינוי של `j` ישנה את הערך שהוא מתייחס אליו. כלומר: הקביעה לאן מתייחס ה-`reference` נקבעת עם אתחולו.

משתנה התייחסות הינו שם נוסף למשתנה קיים. לא מוקצה עבורו עוד זכרון.




```
#include <iostream.h>

int main()
{
	int i = 5;
	int &j = i;

	cout << &i << endl << &j << endl;

	return 0;
}
```

פלט התוכנית יכול להיות, למשל:

```
0x0012FF7C
0x0012FF7C
```

הגדרת משתנים, Constructors & Destructors

ADT הינה אחת מדרכי התכנות השימושיות ביותר בשפת C. עם זאת, כבר ציינו שבשפת C היו בגישה זו מספר בעיות. נציג כעת בעיה נוספת, ונראה כיצד C++ פותרת אותה.
כפי שכבר צוין, הרעיון שמאחורי ADT הוא הגדרת מבנה נתונים (struct) ופעולות (פונקציות) שפועלות עליו.
דוגמא ל-ADT בשפת C היא מנגנון העבודה עם קבצים בשפת C.
הגדרנו struct בשם FILE.
כאשר היינו רוצים לגשת לקובץ, היינו ראשית מצהירים על מצביע מסוג FILE.
לאחר מכן, היינו משתמשים בפעולה fopen על מנת לפתוח את הקובץ.
היינו מבצעים פעולות קלט פלט שונות, ובסיום העבודה היינו סוגרים את הקובץ עם הפקודה fclose.
באופן דומה, לכל ADT בשפת C, יצרנו פונקציות "Create" ו-"Destroy" ששימשו לאתחול והריסת מבנה הנתונים.
תהליך זה, של יצירת מצביע, איתחולו בעזרת פונקציה ומחיקתו בסיום על ידי פונקציה אחרת אינו בטוח. נביט בדוגמא הבאה כדי לראות למה:
נניח שנתונה התוכנית הבאה:

```
#include <stdio.h>
int main()
{
	FILE *fin;
	char szFileName[20];
	printf("Enter file name: ");
	scanf("%s", szFileName);
	fin = fopen(szFileName, "r");
	...
	fclose(fin);
	return 0;
}
```

למתכנתי C, תוכנית זו נראית תקינה, אולם היא מכילה מצב שבC++ אנו מנסים להימנע ממנו: אנו מגדירים את המצביע fin כבר בשורה הראשונה של הפונקציה, אולם אנו שמים בו ערך משמעותי רק עמוק בתוך הפונקציה. הבעייתיות במצב זה שאם יוסף קוד בין הרגע שהצהרנו על fin עד לרגע שאתחלנו את המשתנה, אזי נקבל זבל. נרצה למנוע את קיומם של משתנים או אובייקטים לא מאותחלים.

הגישה של C++: בתוכנית C++ הבנויה כראוי, בכל רגע נתון כל המשתנים מכילים ערכים תקפים (ערכים אלו יכולים להיות גם NULL או ערך שיאמר שאין במשתנה מידע משמעותי, אולם לא יהיו משתנים שיכילו זבל).
האמצעים לעמוד בדרישה זו:

- הגדרת משתנים בכל מקום בפונקציה.
- Constructors ו-Destructors.

הגדרת משתנים

בשפת C++, אנו יכולים להגדיר משתנים בכל קטע של הפונקציה, ולא רק בראש בלוק. אי לכך, נוכל להגדיר על המשתנה, מיד לפני שנרצה להשתמש בו, ולאו דווקא בראש הפונקציה. למשל, התוכנית הבאה היא תוכנית חוקית לגמרי ב-C++, אולם לא ב-C:

```
int main()
{
	printf("Please enter 2 numbers: ");
	int x, y;
	scanf("%d%d", &x, &y);
	int sum = x+y;
	printf("The sum of x+y is %d\n", sum);
	return 0;
}
```

למרות שזו תוכנית חוקית ב-C++, נזכור כי אמרנו שב-C++ נשתמש בפונקציות הקלט והפלט של C++, ולא באלו של שפת C, ולכן הדרך הנכונה לכתוב את התוכנית הינה:

```
int main()
{
	cout << "Please enter 2 numbers: ";
	int x, y;
	cin >> x >> y;
	int sum = x+y;
	cout << "The sum of x+y is " << sum << endl;
	return 0;
}
```

על ידי כך שאנו מגדירים את המשתנים קרוב ככל הניתן למקום בו אנו משתמשים בהם, התקדמנו צעד לעבר המצב הרצוי לנו - המשתנים מצויים כעת זמן מינימלי במצב לא מאותחל. אנו מגדירים אותם רגע לפני שאנו ממלאים בהם תוכן, ולכן הסיכוי שנשתמש במשתנה בו יש זבל קטן, אם זאת, עדיין לא הגענו למצב שבכלל אין משתנים המכילים זבל. נביט כעת כיצד אנו פותרים בעיה זו.

Constructors & Destructors

תחביר בסיסי ומוטיבציה

לכל מחלקה, ישנן מספר פונקציות מיוחדת - הבנאים (Constructors) וההורסים (Destructor).

מבחינת תחביר, הבנאי הוא פונקציה ששמה כשם המחלקה. הבנאי יכול לקבל פרמטרים כלשהם כרצוננו. ההורס הוא פונקציה ששמה מתחיל ב~ (גל), ולאחריו שם המחלקה. ההורס אינו מקבל לעולם פרמטרים.

למשל, נביט במחלקה הבאה. שם המחלקה הוא `MyClass`, ויש לה בנאי והורס, ששניהם אינם מקבלים פרמטרים:

```
class MyClass
{
	public:
		MyClass();
		~MyClass();
};
```

הבנאי וההורס נמצאים לרוב בחלק ה-`public` של המחלקה. ניתן לשים אותם גם תחת הרשאה אחרת, אולם הרווחים מפעולה כזו הם מחוץ להיקפו של מסמך זה. אנו נניח כי תמיד הבנאי וההורס נמצאים תחת הרשאה `public`.

הבנאי היא פונקציה הנקראת ברגע שאובייקט מסוג המחלקה נוצר. מטרת פונקציה זו היא לאתחל את שדות האובייקט בערכים התחלתיים, ולבצע פעולות אתחול נוספות אם יש צורך בכך. למשל, נביט בקטע הקוד הבא:

```
class Rectangle
{
	private:
		int x, y;
	public:
		Rectangle();
}

Rectangle::Rectangle()
{
	x = y = 4;
	cout << "Rectangle Ctor" << endl;
}

int main()
{
	Rectangle x1;
	cout << "Main Program" << endl;
	Rectangle x2;
	return 0;
}
```

יצרנו מחלקה בשם `Rectangle` (מלבן). הגדרנו במחלקה זו את השדות `x` ו-`y`, שהם שדות פרטיים של המחלקה.
הבנאי של המחלקה מאתחל את `x,y` ושם בהם את הערך 4.

כאשר נריץ את התוכנית, על המסך יודפס:

```
Rectangle Ctor
Main Program
Rectangle Ctor
```

מיד כאשר הצהרנו על אובייקט של המחלקה, נקרא הבנאי על מנת לאתחל את השדות שלו.

ההורס של המחלקה נקרא רגע לפני שהזיכרון שהוקצה עבור אובייקט כלשהו משתחרר. אם נרצה לעשות פעולות שונות לפני שחרור הזיכרון של האובייקט, נבצע אותן כאן. למשל, נניח שאחת מהפונקציות של המחלקה הקצתה זיכרון דינמי במהלך ריצת התוכנית, ונרצה לשחרר זיכרון זה, לפני שנשחרר את זיכרון המחלקה.

ניקח כעת את הגדרת המחלקה `Rectangle` ונוסיף לה `Destructor`:

```
class Rectangle
{
	private:
		int x, y;
	public:
		Rectangle();
		~Rectangle();
}

Rectangle::Rectangle()
{
	x = y = 4;
	cout << "Rectangle Ctor" << endl;
}

Rectangle::~Rectangle()
{
	cout << "Rectangle Dtor" << endl;
}
```

נביט כעת במספר תוכנית, שיתייחסו כולן למימוש זה של `Rectangle`:

ראשית נביט באותה דוגמא שראינו קודם, טרם הוספנו את ההורס ל-`Rectangle`.

```
int main()
{
	Rectangle x1;
	cout << "Main Program" << endl;
	Rectangle x2;
	return 0;
}
```

כעת תוכנית זו תדפיס:

```
Rectangle Ctor
Main Program
Rectangle Ctor
Rectangle Dtor
Rectangle Dtor
```

שלושת השורות הראשונות הן אותן שלושת השורות שהיו בדוגמא הקודמת.
בסוף הפונקציה `main()`, משוחררים כל המשתנים המקומיים שהוקצו בה. מכיוון ש-`x1, x2` הינם משתנים מקומיים, הם משוחררים, ורגע לפני שהם משוחררים נקרא ההורס שלהם.

נביט בדוגמא שניה:

```
int main()
{
	Rectangle x1, *x2;
	cout << "Stage 1" << endl;
	x2 = new Rectangle;
	x1 = *x2;
	delete x2;
	cout << "Final stage" << endl;
	return 0;
}
```

על המסך יודפס:

```
Rectangle Ctor
Stage 1
Rectangle Ctor
Rectangle Dtor
Final stage
Rectangle Dtor
```

כאשר הגדרנו את `x1` ואת `x2`, נקרא הבנאי רק פעם אחת, רק עבור `x1`.
`x1` הוא אובייקט מסוג `Rectangle`, ולכן אתחלנו אותו. `x2` הוא מצביע, והוא איננו אובייקט של המחלקה (הוא רק מצביע אל אחד כזה) ולכן אין צורך לקרוא לבנאי עבור `x2`.
כאשר הקצנו זיכרון דינמי, נקרא הבנאי על מנת להקצות את הזיכרון החדש שהוקצה. תופעה זו היא אחת החידושים של הפקודה `new`, לעומת הפקודה `malloc()` של `C`. זיכרון שהוקצה על ידי `malloc()` לא יאותחל (לא יקרא הבנאי בצורה אוטומטית). לעומת זאת, אם אנו מקצים זיכרון בעזרת `new`, נקרא הבנאי של המחלקה.
הפקודה `delete`, בצורה דומה, קראה להורס של `Rectangle` כדי לבצע פעולות ניקוי רגע לפני ששוחרר הזיכרון של האובייקט.
בסוף הפונקציה `main()` שוחרר `x1`, שהוא משתנה מקומי.

כיצד בנאים עוזרים לנו לעמוד בעקרונות של `C++`, שקודם דיברנו עליהם?
ניזכר במטרה אותה רצינו להשיג:
בתוכנית `C++` הבנויה כראוי, בכל רגע נתון כל המשתנים מכילים ערכים תקפים.
כבר ראינו שהגדרת משתנים רגע לפני שמשתמשים בהם, מצמצמת למינימום את הזמן שיש באובייקטים זבל. בנאים עושים יותר מכך - ברגע שאובייקט נוצר, נקרא הבנאי שלו, שתפקידו לאתחל את השדות השונים באובייקט. אם נכתוב את הבנאי כראוי, נוכל להבטיח שכל אובייקט מהמחלקה שאנו כותבים, יהיה מאותחל בכל רגע.
בתוכנית `C++`, האובייקטים השונים יוגדרו כאשר יהיה צריך להשתמש בהם. מיד עם ההצהרה עליהם, הם יאותחלו ללא התערבות המתכנת המשתמש באובייקט, על ידי הבנאים.

מחלקות בעלות מספר בנאים, רשימות אתחול

נחדד כעת מספר נקודות הקשורות לבנאים, ולדרכים בהן כותבים אותם ומשתמשים בהם.
נביט בדוגמא הבאה. אנו נשפר דוגמא זו בהדרגה בעמודים הבאים:

```
#include <iostream.h>

class myClass
{
	private:
		int p_i, p_j;
	public:
		myClass();
		myClass(int i, int j);

		int GetI() { return p_i; }
};

// Constructors Implematation: (first way)
myClass::myClass()
{
	p_i = p_j = 0;
}

myClass::myClass(int i, int j)
{
	p_i = i;
	p_j = j;
}

int main()
{
	myClass my1;

	cout << my1.GetI() << endl;
	// Will print 0 (default constructor was called)

	myClass my2(6, 4);
	cout << my2.GetI() << endl;
	// Will print 6 (second constructor was called)

	return 0;
}
```

למחלקה `myClass` ישנם שני בנאים:

```
myClass();
myClass(int i, int j);
```

כאן אנו רואים שלמחלקה יכול להיות יותר מבנאי אחד. אלו למעשה פונקציות חופפות, אותן כבר ראינו.

כאשר יצרנו את האובייקט `my1`, נקרא הבנאי ללא הפרמטרים. במקרה זה לא העברנו אף פרמטר למחלקה. לעומת זאת, כאשר יצרנו את האובייקט `my2`, יצרנו אותו כך:

```
myClass my2(6, 4);
```

הערכים שבסוגריים הם הערכים שמועברים אל הבנאי המתאים. במקרה זה יקרא הבנאי המקבל שני מספרים שלמים.
לעומת הבנאי, ההורס הוא פונקציה שאיננה מקבלת פרמטרים אף פעם, ולפיכך ההורס הוא יחיד.

נשתמש במספר בנאים כאשר נרצה לאתחל אובייקטים במספר דרכים - למשל, נניח שנרצה ליצור מחלקה שמטפלת באורכים - נוכל לכתוב בנאי שיקבל `int`, לכתוב בנאי אחר שיקבל `double` וכו', וכך לגרום למחלקה שלנו להיות מאותחלת על ידי סוגים שונים של ערכים.

נביט כרגע במימוש נוסף של אותה מחלקה:

```
#include <iostream.h>

class myClass
{
      private:
            int p_i, p_j;
      public:
            myClass();
            myClass(int i, int j);

            int GetI() { return p_i; }
};
// Constructors Implematation: (second way)
myClass::myClass() : p_i(0), p_j(0) { }

// Will be the same as writing inside the constractor:
// p_i = 0; p_j = 0;   -- but this way is faster and recommended

myClass::myClass(int i, int j) : p_i(i), p_j(j) { }

int main()
{
      myClass my1;
      cout << my1.GetI() << endl;
      myClass my2(6, 4);
      cout << my2.GetI() << endl;

      return 0;
}
```

כאן אנו משתמשים **ברשימת אתחול** בבנאי. נביט בצורה בה הבנאי מומש:

```
myClass::myClass(int i, int j) : p_i(i), p_j(j) { }
```

מיד לאחר הצהרת הבנאי, ולפני הסוגריים המסולסלות, מופיעה רשימת האתחול של הבנאי. אנו מתחילים את רשימת האתחול ב"׃" (נקודותיים). לאחר מכן, כאשר אנו כותבים `p_i(i)`, אנו אומרים "אתחל את המשתנה `p_i` לערך `i`.
כל הפעולות שנבצע ברשימת האתחול יתרחשו לפני הכניסה לגוף הבנאי (הבלוק שבתוך הסוגריים המסולסלות).
מימוש זה זהה בפעולתו למימוש הקודם, אולם הוא מהיר יותר, ונחשב לתכנות נכון יותר ב`C++`.
כפי שאנו רואים בשורה הבאה, ניתן לאתחל משתנים לא רק על ידי משתנים אחרים, אלא גם על ידי קבועים:

```
myClass::myClass() : p_i(0), p_j(0) { }
```

נביט כעת בשיפור נוסף ואחרון לתוכנית קצרה זו.

```
#include <iostream.h>

class myClass
{
	private:
		int p_i, p_j;
	public:
		myClass(int i=0, int j=0) : p_i(i), p_j(j) { }
		int GetI() { return p_i; }
};

int main()
{
	myClass my1;

	cout << my1.GetI() << endl;
	myClass my2(6, 4);
	cout << my2.GetI() << endl;
	return 0;
}
```

בדוגמא זו איחדנו את שני הבנאים לבנאי אחד, בעזרת שימוש במנגנון ה `Default parameters` של `C++`. כאשר יצרנו אובייקט ללא פרמטרים, נלקחו ערכי ברירת המחדל בתור הערכים לאתחול האובייקט, וכאשר נתנו ערכים מפורשים, `C++` השתמשה בהם כדי לאתחל את האובייקט החדש.
לכל פונקציה (פרט ל`Destructor` של מחלקה, שהינו פונקציה חסרת פרמטרים) נוכל להגדיר ערכי ברירת מחדל.
בפרט נוכל להגדיר ערכי ברירת מחדל לכל פרמטר בפונקציה של מחלקה, וכן גם לבנאי שלה.

בזאת סיימנו עם שיפור הדוגמא הראשונה הקשורה לבנאים. נעבור למספר דוגמאות נוספות, ונציג בהן נקודות נוספות הקשורות לבנאים.

Default Constructor

לכל מחלקה ב-C++ קיים בנאי, בין אם אנו כותבים אחד כזה במפורש, ובין אם לא.
נניח למשל שניצור את המחלקה הבאה:

```
class MyClass
{

};
```

אם לא נצהיר על בנאי במפורש, C++ תוסיף בזמן הקומפילציה בנאי ברירת מחדל (Default Constructor).
הבנאי יהיה בנאי ריק, שלא יבצע כלום, והוא זהה למעשה למחלקה בה קיימת ההצהרה הבאה:

```
MyClass::MyClass() { }
```

אם נממש בנאי משלנו למחלקה, C++ לא תוסיף למחלקה את בנאי ברירת המחדל.
נביט בקוד הבא:

```
class Man
{
	private:
		char *szName;
	public:
		Man(const char *name);
};
int main()
{
	Man person("Nir");
	Man no_one;
	return 0;
}
```

בדוגמא זו, השורה באדום לא תעבור קומפילציה.
מכיוון שמימשנו בנאי למחלקה, C++ לא תוסיף למחלקה בנאי ללא פרמטרים, ולכן אי יהיה אפשר ליצור אובייקט בלי לאתחל אותו.
ניתן לאמר שבדוגמא זאת מימשנו את הרעיון - "לכל אדם יש שם". C++ לא תיתן לנו להגדיר משתנה מסוג אדם, אם לא ניתן לו שם.

באופן דומה, נביט בדוגמא הבאה:

```
class MyClass1
{
	public:
		MyClass1(int i);
};

class MyClass2
{
	public:
```

```
            MyClass2();
};

class MyClass3
{
      public:
};

int main()
{
      MyClass1 m[100];   /* Illegal */
      MyClass2 m1[100]; /* Legal */
      MyClass3 m2[100]; /* Legal too - MyClass3 has
default constructor that doesn't get any parameters */
      return 0;
}
```

השורה המודגשת באדום לא תעבור קומפילציה. כאשר אנו יוצרים מערך, נקרא הבנאי עבור האובייקטים השונים שאנו יוצרים. מכיוון שאנו לא מסוגלים (מבחינת תחבירית) להגדיר את הערכים שיעברו אל הבנאי, כאשר אנו יוצרים מערכים תמיד יקרא בנאי ברירת המחדל, או הבנאי ללא פרמטרים אם הגדרנו כזה.
לפיכך, לא ניתן ליצור מערכים של אובייקטים ממחלקות שיש להן רק בנאים בעלי פרמטרים.

נביט בדוגמא נוספת הקשורה למערכים.

מה תדפיס התוכנית הבאה?

```
#include <iostream.h>

class A
{
      public:
      A() { cout << "A"; }
};

int main()
{
      A blah[100];
      cout << "Hello World!!!" << endl;
      return 0;
}
```

התשובה:
התוכנית תדפיס את התו A מאה פעמים, ולאחריו ייכתב Hello World!!!.
עבור כל איבר ואיבר במערך, נקרא הבנאי. הבנאי נקרא עבור כל איבר במערך, על מנת שכל התאים במערך יכילו עצמים תקפים מסוג A.

דוגמא מסכמת

לאחר שראינו את נושא הבנאים מכיוונים שונים, נציג תוכנית דוגמא שתסכם את רוב הנושאים שראינו במסמך זה.
נציג כעת את המטרה אליה נגיע בתוכנית, ונבנה את התוכנית בשלבים, פונקציה אחר פונקציה.
דוגמא זו באה להציג כיצד יש לגשת למימוש מחלקה פשוטה ב-C++.

בדוגמא זו נרצה ליצור מחלקה שתייצג בני אדם. לכל בן אדם יש שם, והמחלקה תאפשר לקבל את שמו של בן אדם כלשהו, או להחליף לו את שמו.
נביט בהצהרה על המחלקה:

```
class Person
{
	char *Name;
	public:
		Person(const char *szName);
		~Person();
		void print();
		void SetName(const char *szNewName);
		const char *GetName() const;
};
```

הגדרנו שדה מסוג `char*` בשם `Name`, שיכיל את שמו של האדם. הבנאי יתאחל שדה זה, הפונקציה `GetName()` תחזיר לקורא לה את שמו של האדם, והפונקציה `SetName()` תאפשר לנו לקבוע לאדם שם חדש. הפונקציה `Print()` תדפיס את שמו של האדם על המסך.

נתחיל במימוש הבנאי.
על הבנאי להקצות זיכרון עבור `Name`, ולהעתיק אל `Name` את המחרוזות אותה הוא מקבל כפרמטר.
לכן, נממש את הבנאי בצורה הבאה:

```
Person::Person(const char *szName)
{
	Name = new char[strlen(szName)+1];
	strcpy(Name, szName);
}
```

ראשית הקצנו זיכרון עבור שמו של האדם (פלוס 1 עבור התו `'\0'` המסיים את המחרוזת). לאחר מכן העתקנו את המחרוזת הנתונה כפרמטר אל `Name`.

מימוש הפונקציה `Print()` הוא טריוויאלי. נדפיס את ערכו של `Name` אל הפלט הסטנדרטי:

```
void Person::print()
{
	cout << "My Name: " << Name << endl;
}
```

גם מימוש הפונקציה `GetName()` איננו קשה במיוחד. הפונקציה פשוט מחזירה את ערכו של `Name`:

```
const char *Person::GetName() const
{
	return Name;
}
```

ההורס של המחלקה צריך לשחרר את הזיכרון הדינמי שהקצנו עבור `Name`:

```
Person::~Person()
{
	delete[] Name;
}
```

מכיוון שהקצנו מערך של `char`, נשתמש ב- `delete[]` על מנת לשחרר אותו.

עד כה, ראינו נקודות פשוטות, אך חשובות, כאשר אנו יוצרים מחלקה, המשתמשת בזיכרון דינמי. נביט כעת בפונקציה המעניינת ביותר במחלקה, `SetName()`.
מהן הפעולות שעל פונקציה זו לבצע?
ראשית, נזכור שאנו עובדים ב`C++`, ולכן אנו מניחים כי האובייקט נמצא כל העת במצב תקין, מאותחל. ומכאן: כאשר קוראים לפונקציה `SetName()`, מוקצה כרגע כבר זיכרון עבור השם הקודם.
גישה נאיבית תאמר כעת: הזיכרון כבר מוקצה, ולכן אין צורך להקצות שוב זיכרון, לכן רק נעתיק את המחרוזת החדשה במקום המחרוזת הישנה, ונסיים את הפונקציה.
למשל, נוכל לממש גישה זו בצורה הבאה:

```
void Person::SetName(const char *szNewName)
{
	strcpy(Name, szNewName);
}
```

פתרון זה <u>איננו נכון!</u>
איננו מתייחסים כאן לעובדה, כי יכול להיות שהשם החדש אותו נרצה לתת לאדם, הינו בגודל שונה מהשם הישן שלו.
אם השם החדש קטן מהשם הישן, לא תהיה בעיה במימוש, מלבד העובדה שמעט מהזיכרון שהקצנו יישאר מוקצה אך ללא שימוש מעשי.
אם השם החדש גדול מהשם הישן, תתרחש דליפת זיכרון.
לפיכך, הגישה הנכונה תהיה ראשית לשחרר את הזיכרון שהקצנו עבור השם הישן, ואז להקצות זיכרון לשם החדש.

להלן מימוש נכון של הפונקציה `SetName()`:

```
void Person::SetName(const char *szNewName)
{
	delete[] Name;
	Name = new char[strlen(szNewName) + 1];
	strcpy(Name, szNewName);
}
```

ראשית אנו משחררים את הזיכרון הישן, ואז אנו מקצים זיכרון חדש בגודל הרצוי ומציבים אותו במשתנה `Name`.

לסיום, נציג את המחלקה שבנינו בשלמותה:

```
#include <iostream.h>
#include <string.h>

class Person
{
	char *Name;
	public:
		Person(const char *szName)
		{
			Name = new char[strlen(szName)+1];
			strcpy(Name, szName);
			cout << "ctor" << endl;
		}

		void print()
		{
			cout << "MyName: " << Name << endl;
		}

		~Person()
		{
			delete[] Name;
		}

		void SetName(const char *szNewName)
		{
			delete[] Name;
			Name = new char[strlen(szNewName) + 1];
			strcpy(Name, szNewName);
		}

		const char *GetName() const
		{
			return Name;
		}
};
```

סיום

מסמך זה מגיע אל סיומו. במסמך זה סקרנו בקצרה את הנושאים הבסיסיים ביותר בשפת C++, אולם ל-C++ יש הרבה יכולות להציע מעבר לאלו שראינו במסמך זה, ואף הנושאים שהוצגו במסמך זה, לא מוצו עד תום.
לסיום המסמך, נציג דוגמא אחרונה, ללא הסברים רבים, שתציג את אחת התכונות החשובות ב-C++ - הורשה.
רעיון ההורשה הוא אחד הרעיונות החזקים ביותר של C++:
לאחר שהגדרנו מחלקה, נרצה להשתמש בה גם במחלקות אחרות.
דרך אחת הינה להגדיר אובייקט מסוג מחלקה אחת בתוך מחלקה אחרת.
אולם, לפעמים נרצה להגדיר מחלקה אחת, ולאחר מכן להגדיר מחלקה שניה, שהיא מקרה פרטי של המחלקה הראשונה. לדוגמא: לאחר שהגדרנו בדוגמא הקודמת את המחלקה "בן אדם", יתכן שנרצה להגדיר עכשיו את המחלקה "זמר".
"זמר" הוא בפרט "בן אדם", וכולל את כל התכונות של "בן אדם". בנוסף לכך הוא כולל תכונה נוספת. "זמר" יודע לשיר.
C++ נותנת לנו אפשרות להגדיר מחלקה נוספת - שתקבל את כל המשתנים והפונקציות (עם הגבלות מסוימות) של המחלקה המקורית, וכך לא נצטרך לממש שוב עבור המחלקה השניה.

נציג דוגמת קוד לכך:
נרצה לממש מחלקה שתייצג מלבן, ושתוכל להדפיס מלבן על המסך, וכן לחשב את שיטחו של המלבן. לאחר מכן נממש בעזרתה מחלקה מסוג ריבוע, שהוא מקרה פרטי של מלבן.

להלן מימוש מחלקת המלבן:

```
#include <iostream.h>

class myRectangle
{
	public:
		int Area();
		void SetX(int vX) { if (vX >= 0) m_x = vX; }
		void SetY(int vY) { if (vY >= 0) m_y = vY; }
		void Print();
		myRectangle(int x, int y) : m_x(x), m_y(y) { }
		~myRectangle() { }
	private:
		int m_x, m_y;
};

int myRectangle::Area()
{
	return m_x * m_y;
}
```

```
// Function that prints the Rectangle on the screen
void myRectangle::Print()
{
	for (int x = 0; x < m_x; ++x)
	{
		for (int y = 0; y < m_y; ++y)
		{
			cout<<'*';
		}
		cout<<endl;
	}

	// New line at the end of the shape
	cout<<endl;
}
```

פרטי המימוש של המחלקה אינם מסובכים, והקורא יוכל לעמוד על הדקויות בהן לבדו. נממש כעת את המחלקה ריבוע, **שתיגזר** (תקבל את כל התכונות) של המחלקה מלבן. על מנת לציין שהמחלקה ריבוע נגזרת מהמחלקה מלבן, נצהיר על המחלקה ריבוע כך:

```
class Square : public myRectangle
```

בצורה כזו המחלקה קיבלה את המתודות והשדות השונים של מחלקת המלבן.
כאשר אנו כותבים `public myRectangle`, אנו אומרים כי כל שדות ה-`public` של המחלקה `myRectangle`, יהיו גם בשדות ה-`public` של המחלקה `Square`.
אם היינו כותבים `private myRectangle`, היינו אומרים כל שדות ה-`public` של המחלקה `myRectangle`, יהפכו להיות שדות `private` במחלקה `Square`.
לרוב נשתמש ב-`public` כאשר נגזור מחלקה ממחלקה.
להלן מימוש המחלקה:

```
class Square : public myRectangle
{
	public:
		Square(int x) : myRectangle(x, x) { }

		void SetX(int vX)
		{
			myRectangle::SetX(vX);
			myRectangle::SetY(vX);
		}

		void SetY(int vY)
		{
			SetX(vY);
		}

};
```

נשים לב למספר דברים חדשים במימוש זה:
הבנאי הוגדר בצורה הבאה:

```
Square(int x) : myRectangle(x, x) { }
```

משמעות כתיבה זו: הבנאי של `Square` קורא לבנאי של `myRectangle`, ומעביר לו את הפרמטרים `x, x`.

```
void Square::SetX(int vX)
{
	myRectangle::SetX(vX);
	myRectangle::SetY(vX);
}
```

אנו מממשים מחדש עבור הריבוע את הפונקציות `SetX()` ו-`SetY()`, מכיוון שבריבוע, שינוי אורך צלע גורר שינוי אורך של כל הצלעות.
כאשר אנו כותבים את הביטוי **`myRectangle::SetX(vX)`** בתוך הפונקציה, אנו למעשה אומרים ל-C++ להשתמש בפונקציה `SetX()` שירשנו מהמחלקה `myRectangle`, ולא לקרוא לפונקציה `SetX()` שמוגדרת מחדש במחלקה `Square`.

נדגים כעת שימוש בתוכנית:

```
int main()
{
	myRectangle myRec(5, 4);
	myRec.Print();
	Square mySqr(7);
	mySqr.Print();
	mySqr.SetX(2);
	mySqr.Print();
	cout << "Square Area is : " << mySqr.Area() << endl;

	return 0;
}
```

כעת אנו רואים את יתרון תכונת ההורשה של C++ בבירור.
את הפונקציות `Area()` ו-`Print()` לא מימשנו במפורש עבור הריבוע, אולם מנגנון ההורשה מאפשר לנו להשתמש בפונקציות אלו, השייכות למלבן. בצורה זו, חסכנו לעצמנו כתיבה. מימשו במחלקה ריבוע רק את הפונקציות הקשורות ספציפית לריבוע, ולא למלבן, ואת כל השאר קיבלנו בצורה שקופה מהמלבן.

-EOF-