סדנת תכנות בשפת C++ (67315)

מסכם:

יחיאל מרצבך

עקרונות בסיסיים במחלקות

הקדמה למחלקות

החידוש המרכזי של שפת c++ הינו השימוש במחלקות (classes).

מה הן מחלקות? לעתים נרצה לאגד איזשהו רעיון או אובייקט, סביב מתודות ומשתנים מסוימים. למשל, נרצה להגדיר **חיה** בעלת זנב וראש, שמסוגלת לאכול, להשמיע קול ועוד. לשם כך, נגדיר את המושג מחלקה (class), אובייקט בשפה שיש לו מתודות ומשתנים. הדבר הדומה ביותר לזה ב-c הינו struct, אלא שבלתי אפשרי לעשות בו **פעולות**. בנוסף, ב-struct לא ניתן **להסתיר מידע**; לפעמים נרצה להסתיר מידע, מבחינת העיצוב של התוכנה ועל מנת להגן על עצמנו מטעויות.

כך נראית מחלקה:

```
//file: Point.h
#ifndef _POINT_H_
#define _POINT_H_
class Point // capital 'P'
{ public:
 Point(int x, int y); // Constructor
int get_x() const; // A "getter" method
int get_y() const; // A "getter" method
 void setY(int y); // A "setter" method
private:
 int _x, _y;
};
#endif // _POINT_H_
```

נציג כאן מספר עקרונות מרכזיים, בהם ניגע בהעמקה בהמשך. נוכל לראות את הבנאי (constructor) וgetter וsetter. הבנאי למעשה מאתחל את האובייקט, והמתודות האחרות שמופיעות כאן מטרת להגדיר API (במקרה והן בצורת public). אם מדובר ב-private" לא ניתן לראות אותם מחוץ לסקופ.

במקרה בו יש const בסוף הפונקציה, דבר זה מסמן **שלא ניתן לשנות** את משתני המחלקה דרך מתודה זו. דבר זה מאפשר לנו 'להגן' על הקוד.

על מנת לממש את מתודות אלו, נכנס לקובץ cpp עצמו, ונממש כך:

```
//file Point.cpp
#include "Point.h"
Point::Point(int x, int y) {
 _x = x; _y = y;
}
int Point::get_x() const
{
 return _x;
}
```

```
void Point::set_x(int x)
{
 _x = x;
}
```

בנאים

בנאי פשוט

כפי שהקדמנו, מטרת הבנאי הינה לאתחל את המחלקה. הוא איננו מחזיר דבר והוא נראה כך:

```
Point::Point(int x, int y)
{
 _x = x; _y = y;
}
```

כלומר, שמו הוא בדיוק כמו שם המחלקה. אם נרצה ליצור אכן אובייקט:

```
Point p(1, 3); // constructor!
```

נעיר כי משתנה זה מאוחסן על הstack.

על מנת לקמפל, נכתוב בדומה לשפת c:

```
> g++ -c -Wall Point.cpp –o Point.o
> g++ -c -Wall app.cpp –o app.o

> g++ Point.o app.o –o app
```

בנאי דיפולטיבי

מה יקרה אם לא יהיו לאובייקט מסוים "הוראות בנייה"? תהיה שגיאת קומפליציה. לכן פעמים רבות נעדיף להגדיר "בנאי דיפולטיבי" שקובע מה יהיו הערכים כאשר לא נכניס פרמטרים בעת הגדרת האובייקט. בצורה הבאה, הבנאי איננו מקבל ערכים:

```
class MyClass
{
 public:

 MyClass();\\default constructor

 MyClass( int i );

 MyClass( double x, double y );

 ...
};
MyClass a; // Calls 1

MyClass b( 5 );// Calls 2

MyClass c( 1.0, 0.0 ); // Calls 3
```

בבנאי דיפולטיבי, לא נקבל שום פרמטר. אם אין שום בנאים, הקומפיילר מייצר אחד דיפולטיבי. נוכל להגדיר גם בנאי דיפולטיבי עם ערכים דיפולטיבים בפרמטרים של אחד הבנאים האחרים (x=5 למשל בבנאי השני).

רשימת אתחול (Initializer List)

אם ישנם משתנים שהם בעצמם מופעים של אובייקט, על כל הבנאים של האובייקט להיות קיימים **לפני שנאתחל** את האובייקט. נוכל גם לאתחל את המשתנים והאובייקטים הללו, בעת יצירת האובייקט עצמו. כך מבצעים זאת:

```
Circle::Circle(int x, int y, double r):_center(x,y) {
}
```

נצטרך להשתמש ברשימת האתחול עבור:

1. אובייקטים **ללא** בנאי דיפולטיבי.
2. משתנים קבועים – `static` שאינם יכולים להשתנות לאחר אתחולם, ואינם מאותחלים.
3. משתני הפנייה (`reference` – ניגע בזה בהמשך).

דלגציה של בנאים

על מנת שלא לשכפל קוד, נוכל לקרוא לבנאי אחד מתוך בנאי השני, למשל במקרה של בנאי דיפולטיבי:

```
private:
 void init(int x, int y)
 {
 this->_x = x;
 this->_y = y;
 }
public:
 Point()
 {
 init(0,0);
 }
 Point(int x)
 {
 init(x,0);
 }
…
 };
```

בצורה זו, אנחנו 'חוסכים' ביצירה של בנאים ומונעים כפל קוד, שעלול לגרום לבעיות במידה ונשנה משהו.

בנאי העתקה – copy constructor

לעתים רבות נרצה לממש בנאי העתקה, שתפקידו לקבל אובייקט מסוים ולהעתיק את כל המידע שיש במחלקה אחת למחלקה האחרת. למשל, באובייקט הקודם, היינו מקבלים אובייקט מסוג `point` ולוקחים את המשתנים שלו להיות משתני המחלקה שלנו. או בדוגמה הבאה:

```
Complex::Complex(const Complex &c)
: _real(c._real), _img(c._img)
{ }
```

Destructors – השמדה

מדוע שנצטרך להשמיד את המחלקה? נצטרך לבצע זאת כאשר נסיים להשתמש בה. אם כך, האובייקט 'מושמד', כאשר מסתיים הסקופ, למשל. המטרה של destructor היא לאפשר לנו למחוק את כל הזיכרון של האובייקט לאחר השימוש. המתודה של ה-destructor צריכה 'לשחרר' את כל המשאבים בהם השתמשנו במחלקה. נסמן את ה destructor באמצעות ~ לפני שם המחלקה.

למשל, אם נגדיר malloc איכנשהו (שימו לב, זו פקודה ב-c, בשפת c++ ישנן דרכים אחרות להקצות זיכרון):

```
#include <cstdlib>
 class MyClass
{
 public:

 MyClass();
 ~MyClass();
 // destructor
 private:
 char* _mem;

};
 MyClass::MyClass()

{
 _mem=malloc(1000);

} MyClass::~MyClass()
{

 free(_mem);

}
```

ניתן לראות כאן כיצד הקצינו זיכרון למשתנה, וב-destructor אנו משנים זיכרון זה.

גישות למשתנים – public ו-private

באמצעות המילים private ו-public נוכל לקבוע האם נוכל לגשת למשתנים מסוימים או לא:

```
class MyClass
{
 public:


 // Variables and methods defined here can be
// called and set by MyClass users.
private:
//
// Variables and methods defined here can only be
// called and set by MyClass methods, by Foo's
// writer.
};
```

כלומר, אם רשמנו משתנה או מתודה שהינן private, לא ניתן לגשת אליו מחוץ למחלקה, ואם רשמנו public אז ניתן לגשת. בהמשך, נראה עוד צורת גישה נוספת.

אם לא נכתוב כלום, נקבל באופן דיפולטיבי משתנה שהינו private.

הקשר בין struct ובין מחלקות

למעשה, ההבדל היחיד בין struct ובין class ב++c הינו שבstruct הדיפלוטיבי הינו Public. גם אין צורך כלל ב-typedef, ובשונה מבשפת c, במקרה שלנו ניתן להגדיר גם מתודות (פונקציות) ב-struct.

אלמנטים נוספים במחלקות

friend

לפעמים נרצה "לרמות" ולגשת למשתנים פרטיים של מחלקה מסוימת. נוכל לעשות זאת באמצעות המילה השמורה friend אותה נרשום לפני השם של המתודה – כך למעשה מחלקה אחת הינה 'חברה' של מחלקה אחרת.

this

מדובר על מצביע ל'אובייקט' עצמו. ניתן להחזיר גם מצביע למידע עצמו באמצעות ->. כך למשל:

```
void S::increment(S* this)
{
this->_a++;
}
```

לעתים נשתמש בביטוי זה כאשר נרצה להחזיר את האובייקט עצמו לאחר שינויים.

סינטקס נוסף בשפת c++

המילה השמורה static

אם נרצה למשל לספור את 'מספר הפעמים שהמחלקה נוצרה', נגדיר משתנה מחלקה עם המילה השמורה static. משתנה זה נוצר **פעם אחת בלבד**, ונוכל לעדכן אותו בכל פעם שהאובייקט נוצר. במילים אחרות, המילה static במחלקה באה לומר כי המשתנה מקושר **למחלקה עצמה** ולא לאובייקט ספציפי. בעקבות כך, עלינו לעדכן אותו מחוץ למחלקה עצמה:

```
//cpp file
int Point::object_count = 0;
```

בעקבות העובדה שהוא לא מקושר לאובייקט ספציפי, לא ניתן לגשת אליו באמצעות המילה this.

בצורה דומה, פונקציה יכולה להיות סטטית, כלומר היא מקושרת למחלקה עצמה ולא למופעים ספציפיים – ולכן היא לא יכולה לגשת למשתנים ספציפיים שקשורים למופע של המחלקה, אלא רק **למשתני המחלקה עצמם**.

Const

כפי שכבר הזכרנו, באמצעות הוספת המילה השמורה const בסוף הגדרה של פונקציה, לא נוכל לשנות אף אחד ממשתני המחלקה:

```
int get_x() const;
```

בהרבה פעמים נרצה שתי גרסאות, אחת למקרה בו יהיה 'מותר' לשנות את המחלקה, ואחרת במקרה בו זה בלתי אפשרי. כאשר למשל תהיה לנו מחלקה שהיא בעצמה מוגדרת בתור const, נקרא למתודה שהינה const. דוגמה טובה לכך:

```
class A
{
 public:
A(){}

void foo() const;
 void foo();

}; void A::foo()
const
{

 cout << "const foo\n";

} void A::foo()
{

 cout << "foo\n";
```

```
}
int main()
{

 A a;

 const A ca;

 a.foo ();

 ca.foo();
}
// output-overload
foo
const foo
```

המחלקה string

במקום `char*` , ב-c++ יש לנו את המחלקה `string`, שהינה חלק מהמחלקה הסטנדרטית.

נצטרך לבצע `#include <string>` ואז נבצע `std::string`.

משתני הפנייה - Reference variables

מה המשמעות של משתנה מסוג `reference`? מדובר למעשה ב'שם נרדף' למשתנה אחר.

לעתים לא נרצה לשמור את הכתובת בזכרון כי זה דבר יקר. עלינו להגדיר אותו בזמן האתחול,

ולא ניתן לשנות אותו לאחר מכן:

```
int i= 10;
int& ref= i; // ref is an int reference
// initialized at definition,
// only once!

int& ref2; // error: initializer missing
```

למעשה, משתנה זה שקול למצביע קבוע.

נוכל אם כך להעביר בשלושה דרכים משתנים לפונקציות: באמצעות מצביע, באמצעות הפנייה (by reference) ובאמצעות ערך (by value).

Reference במחלקות

המוטיבציה המרכזית שלנו לשימוש ב-reference היא אם נרצה לשנות משתנה מסוים במתודה, ולא ליצור אותו מחדש. כמו שעשינו עם מצביעים, אנחנו מעבירים אותו by reference:

```
void add(Point& a, const Point& b)

{
 //
// a is a reference,
// b is a const ref
```

```
a._x+= b._x;

 a._y+= b._y;
}
```

נוכל גם להחזיר 'הפנייה' למשתנה מסוים:

```
Point& add(Point& a, const Point& b
)
{
 //
// a is reference, b is a const ref
a._x+=b._x;

a._y+=b._y;
 return a;
}
```

הסיכון בהחזרה של משתנה מסוג `reference` דומה לסיכון שהיה לנו במצביעים, אם נחזיר `reference` למשתנה שנמצא על ה`stack` ונמחק בסוף הסקופ, תתקבל שגיאת זיכרון.

נוכל גם לשנות משתנה פנימי דרך הreference:

```
class Buffer
{

 public:


 Buffer (int l){

 _length = l;
 }

 int& get()
 {

 return _length;
 }

private:
int _length;
};

int main ()
{


 Buffer buf(5);

 buf.get() = 3;
}
```

לפעמים נרצה גם להגדיר שם נרדף לאובייקט, כך:

```
Point const& _center;
```

נצטרך לעדכן אותו גם ב'רשימת האתחול'.

מילות שמורות אחרות

mutable

המילה השמורה mutable מאפשרת לנו לשנות משתנה על ידי פונקציה שהינה const(אפילו אם האובייקט הוא const). למשל:

```
class X {
public:
X() : m_flag(true)
{}
 bool getFlag() const {

 m_accessCount++;


 return m_flag;
 }

 private:

 bool m_flag;

 mutable int m_accessCount;
};
 int main()
{
 const X x;

 x.getFlag();

}
```

בגדול לא מומלץ להשתמש במילה זו, כי היא 'שוברת' את הבטיחות של השפה, ויש להשתמש בה רק בסיטואציות מאוד מסוימות.

NameSpaces

המילה השמורה namespace מאפשרת לנו לאגד יחד משתנים ופונקציות. נוכל גם להשתמש בדבר זה על מנת להבדיל בין משתנים ופונקציות בעלי אותו השם. מבחינת סינטקס, נגדיר את הביטוי למשל namespace Alpha ואז ניגש אליו פנימה עם .::. מחלקות הן אמנם לא נחשבות בתור namespace, אבל כפי שראינו גם הגישה אליהן מתבצעת בדרך זו.

המילה השמורה using

באמצעות המילה השמורה using נוכל להגדיר למעשה מהו הערך שאליו אנו מתכוונים, במקום שנשתמש בכל פעם בגישה. כך למשל:

```
namespace Alpha //global in namespace Alpha
{

 char c_a;
 namespace Beta {

 char c_b;
```

```
 }
}

 int main ()
{

 Alpha::Beta::c_b = 0;

 using Alpha::Beta::c_b;
 c_b = 5;
 using namespace Alpha;

 c_a = 5; //instead of Alpha::c_a = 5;
}
```

חשוב להדגיש שאסור לעשות namespace למחלקה הסטנדרטית std. זה גורם לו להיות עצום ולפלוט לתוך הסקופ שלנו מלא דברים לא נצרכים.

המילה השמורה auto

באמצעות המילה השמורה auto ניתן לקבוע באופן אוטומטי מה הסוג של המשתנה, באמצעות המאתחל שלו. שימו לב כי היא מחזירה ערך by value ולא by reference.

דוגמה קטנה:

```
auto a = 5; // int a
```

הקצאת זיכרון דינמית

כיצד נוכל להקצות זיכרון דינמי ב-c++? באמצעות המילה השמורה new. כך למשל:

```
IntList *L = new IntList;
```

על מנת למחוק את הזיכרון שהקצנו, עלינו לבצע delete. אם נרצה לבצע מחיקה לזיכרון של מערך, נצטרך לכתוב את הפעולה delete[].

בדרך כלל, איננו צריכים לבדוק האם הקצאת הזיכרון בוצעה בהצלחה, כיוון שתיזרק שגיאה (exception) מהשפה עצמה. אם נוסיף את הדגל std::nothrow, לא תיזרק שגיאה ויתקבל מצביע ל-null במקרה בו ההקצאה לא בוצעה בהצלחה.

אם יש לנו מערך במחלקה שהקצינו לו זיכרון, כדאי למחוק את הקצאת הזיכרון ב-destructor. אם נשתמש בבנאי העתקה, נצטרך להקצות זיכרון מחדש בכל פעם, בשביל שהמערכים והאובייקטים יהיו 'עצמאיים'.

Exceptions

במקום לקרוס, בשפת c++ נזרקות 'החרגות' (exceptions). נוכל להשתמש בבלוקי try-catch על מנת ליצור החרגות ולהתמודד איתן. למשל:

```
try { // code that might throw an exception
throw 20;
}
 catch (const int &e)
 // caught exception type
{

std::cout << "Caught exception: "
<< e<< std::endl;
}
```

ניתן לבצע מספר 'תפיסות' על `try` ספציפי. ניתן, באמצעות הסימון של '...' לתפוס את כל סוגי החריגות, או להתעלם מהשגיאה.

נבחי כי בספרייה הסטנדרטית ישנן חריגות מובנות, שניתן להשתמש בהן. למשל:

```
try
{
MyClass *p1 = new MyClass;
}
catch (const std::bad_alloc& ba)
{
std::cerr << "bad_alloc caught: " << ba.what() << std::endl
}
```

אם לא נבצע 'תפיסה', אז תיזרק שגיאה ונצא מהתוכנית.

אין לבצע חריגות בפונקציית ה-`destructor` - עלולה להיווצר התנהגות בלתי רצויה.

stream ופעולות על מחלקות

Streams

Stream הוא למעשה אבסטרקציה של קלט ופלט. יש לנו מקור או יעד שממנו או אליו אנחנו מקבלים או פולטים.

ככלל, על מנת לקבל קלט נשתמש באופרטור ההכנסה <<. על מנת לייצא פלט, נשתמש באופרטור החילוץ >>. מדובר למעשה בסימון של בייטים בגודל לא מוגבל.

ישנן מספר סוגי מחלקות לקבלת קלט או לייצוא פלט:

1. ofstream – נועד להכנסת פלט לתוך קובץ.
2. iftream - נועד לייצא מידע מתוך קבצים.
3. fstream – נועד גם לקלטים וגם לפלטים.
4. Iostrem – האובייקט הסטנדרטי של stream.

עלינו לכלול את הספריות הדרושות לנו בקובץ.

למעשה, גם cout, cerr הם סוגי של ostream שאנחנו כותבים אליהם.

כל אחד מהאיברים ב-stream משורשרים אחד אחרי השני. כך למשל, אם נגדיר std::cout << 17 אנחנו מחזירים reference ל-std::cout, אליו נוכל לשרשר עוד דברים, למשל << "hi".

בעזרת std::cin נקבל למעשה קלט **מהמשתמש** עצמו – זה כקלט שלנו.

נראה דוגמה של קריאה מקובץ:

```
std::ofstream outFile(“myOut.txt", std::ios::out);"

outFile << "Hello world" << std::endl;
 int i;
std::ifstream inFile("in.txt");
 while (inFile >> i)
{

// your code with the I you read
}
outFile.close();
```

נגדיר את מצבי הפתיחה של הקובץ:

1. ios::out – מאפשר לפתוח קובץ לקריאה וכתיבה.
2. ios::in – מאפשר לפתוח קובץ לקריאה.
3. ios::app – מוסיף דברים בסוף הקובץ (כביכול עומדים בסוף הקובץ).

`ios::trunc` – מוחק את התוכל של הקובץ.

4. `ios::ate` – פותח בסוף הקובץ ומאפשר ללכת אחורה.

נוכל גם לשלוט כביכול ב'סמן', באמצעות הפקודות הבאות:

1. `tellg()` – מחזיר את המיקום של הסמן בקלט.
2. `tellp()` – מחזיר את המיקום של הסמן בפלט.
3. `seekg(n, location)` – משנה את הסמן בקלט בהסטה של n.
4. `seekp(n, location)` – משנה את הסמן בפלט בהסטה של n.

ואם נרצה לוודא את הקלט:

1. מתודת `bad()` – מקבלת true אם הקריאה או הכתיבה נכשלים.
2. מתודת `fail()` – מקבלת true אם bad() מקבל true או שיש בעייה בפורמט (מספר ביטים לא מתאים לקריאה)
3. מתודת `eof()` – מקבלת true אם הגענו לסוף הקובץ.
4. מתודת `clear()` – מאתחלת את דגלים אלו.

במקום דגלים, נוכל גם להשתמש גם בערך אליו ננסה להכניס את הקובץ. למשל:

```
std::ifstream fileIn("myFile.txt");
 int val;
 while (fileIn >> val)
{

std::cout << val << std::endl;

}
```

דבר זה שקול ל-`!fail()`.

למעשה, כאשר אנחנו כותבים ל-`stream`, מערכת ההפעלה משתמשת בבאפר, כשהיא בוחרת מתי להעביר את התוכן שלו. נוכל לעשות זאת באופן ידני באמצעות פעולת `flush()` וגם כאשר אנו יורדים שורה באמצעות `std::endl` אז מתבצעת ריקון של הבאפר, זאת בשונה מפעולת `\n`.

Operator Overloading

באופן עקרוני, שפת `c++` לא יודעת לבצע פעולות אופרטריות בין מחלקות. אמנם, נוכל להגדיר מחדש עבור המהדר כיצד יוגדר חיבור בין מחלקות. דבר זה יתבצע באמצעות התבנית הבאה:

```
<return value> operator <operator symbol> (<Class_name> const& a,
<Class_name> const&b)
```

כך למשל:

```
// header file
class Complex {
 public:
 bool operator==(const Complex& rhs) const;
};

// cpp file
bool Complex::operator==(const Point& rhs) const {
 return _r == rhs._r &&
_i == rhs._i;
```

אם נרצה לבצע הדפסה של משתנה כלשהו של המופע, כיצד נוכל לעשות זאת באופן מוגדר? אם נרצה להדפיס את המחלקה, כלומר להעביר ל`std::output`, נצטרך להשתמש במימוש אופרטור ההכנסה `<<`. נעיר כי אנחנו תמיד מעבירים את ה-`Stream` בצורת ה- `by reference`.

נשתמש במילה השמורה `operator` שלמעשה משנה את הגדרת האופרטור. נוכל לבצע זאת על כל האופרטורים המוכרים לנו, למשל `+=` או `*` או כל מה שבא לנו. בדרך אגב נעיר כי היינו יכולים לקרוא לפונקציה operator באופן מפורש, וכי במקרה של `+=` כדאי להשתמש בהחזרה של מצביע למופע (על מנת לשרשר).

אם נרצה 'לחלץ' מstream, נשתמש באופרטור החילוץ `>>` שמקבל מצביע ל-`istream` ומחזיר גם שרשור של מצביע כזה.

על מנת שנוכל לבצע הגדלה רק לאחר הכרזת אובייקט באמצעות ++, נשתמש בהוספת int בחתימה הפונקציה. למשל operator++(int).

אופרטור ההשמה מול copy constructor

מה ההבדל בין השמה ובין בנאי ההעתקה? בבנאי העתקה אנחנו ממש יוצרים אובייקט חדש. במקרה של השמה אנחנו משימים זאת לגבי אובייקט קיים.

החתימה של אופרטור ההשמה נראית כך:

```
X& X::operator=(const X& x);
```

אנחנו מחזירים `reference` לאובייקט שנמצא בצד שמאל למעשה, והחזרת ה-`reference` נועדה על מנת לשרשר.

חוק השלושה (rull of three)

חוק השלושה למעשה בא לומר לנו כי אם נצטרך לממש אחד משלושת הבנאים, סביר להניח שנצטרך לממש את שלושתם:

1. `Destructor`.
2. `Copy constructor`.
3. `Copy assignment`.

הרעיון הוא כי אם נצטרך להקצות זיכרון במקום מסוים, סביר להניח כי נצטרך להקצות בהעתקה וב-assign.

STL - Standard Template Library

עד כה, היינו צריכים בעצמנו מחלקות מורכבות כמו וקטורים, מילונים, וכו'. הספרייה הסטנדרטית של c++ מאפשרת לנו להשתמש במבני נתונים קבועים שאיננו צריכים לקבוע בכל פעם מחדש.

בגדול, STL מורכב משלושה מרכיבים מרכזיים: איטרטורים, מיכלים (containers) ואלגוריתמים.

מיכלים הם למשל כמו וקטורים, רשימות, מילונים וכו', ואלגוריתמים אלו אלגורתימי מיון או כל מיני פעולות על המיכלים. מיכלים מורכבים משני סוגים מרכזיים: בעלי חשיבות לסדר וללא חשיבות לסדר, למשל ההבדל בין set ובין list. הקונטניירים המרכזיים בעלי חשיבות לסדר הם list, vector, deque.

Vector

על מנת ליצור וקטור, נצטרך לגשת לספרייה הסטנדרטית std::vector<T>. ה-T מסמן לנו את הטיפוס שאנו רוצים בתוך הוקטור או הרשימה. בהמשך ניגע בכך כשנדבר על טמפלייטים.

לוקטור יש מספר מתודות:

1. בנאים:
   a. בנאי דיפולטיבי שיוצר וקטור ריק.
   b. בנאי שמאפשר להכניס n ערכים. למשל std::vector<int> v(n).
   c. בנאי שמאפשר להכניס את האיברים עצמם.
   d. בנאי העתקה.
   e. בנאי שמאפשר להכניס מספר איברים מסוג מסוים.
2. מתודות מרכזיות:
   a. push_back(k) הוספת איבר לסוף הוקטור.
   b. pop_back() – מחיקת האיבר האחרון.
   c. clear() – ריקון הוקטור.
   d. at(i) – קבלת האיבר במקום ה-i.
   e. operator [] – מחזיר את האינדקס ה-i.
   f. IsEmpty() – בודק האם הוא ריק.
   g. size() – מחזיר את הגודל של הוקטור.

כדאי להבחין כי push_back יעיל יותר מ-push_front.

לעומת זאת, בdeque אין הבדל בין push_back ו-push_front. אם כך, כיצד הוא עובד? על שני מערכים שונים, כלומר לוקח יותר מקום.

Container Adapter

ב-STL ישנן מספר קונטיינירים שאינם מחלקות בפני עצמן אלא המטרה שלהם הוא להגדיר את הפונקציונלית של מחלקה קיימת. כלומר, למשל נוכל להגדיר את stack בתור צמצום הפונקציונליות של vector ל-push_back ו-pop_back ואת תור (queue) בתור צמצום הפונקציונלית של deque לpush_back ו-pop_front.

קונטיינר אסוציאטיבי

בדומה למילון בפייתון, יש לנו סוג של מילון בשפת c++. כך למשל:

```
std::map<T1, T2>

 std::set<T>

std::multimap<T1, T2>

 std::multiset<T>

std::unordered_map<T1, T2>

 std::unordered_set<T>
```

או אם נרצה דוגמה קונקרטית לשימוש במפה:

```
months["january"] = 31;
months["february"] = 28;
months["march"] = 31;
```

כל אחד ממבני הנתונים שלעיל מכיל הבדלים, מבחינת המימוש, או מבחינת מספר האיבירים שנמצאים בו.

איטרטורים

כיצד נוכל לרוץ מעל מיכלים אסוציאטיבים למשל? הרי, אין שם אינדקסים! באמצעות איטרטורים, נוכל לרוץ על כל container שנרצה, בעל אינדקסים או ללא אינדקסים, אף אם יש חשיבות לסדר או אף אם אין.

לשם כך, נצטרך את שתי הפעולות הבאות: begin() ו-end() שמחזירות איטרטורים לתחילת הסדרה ולסופה, בהתאמה:

```
std::set<int> mySet;
// add some integers to mySet
std::set<int>::iterator i1 = mySet.begin();
std::set<int>::iterator i2 = mySet.end();
```

איטרטורים הם **סוג** של מצביעים, ולכן נבצע dereference על מנת לקבל את האיבר באותו המקום.

ניתן לקדם אותו גם עם ++. למשל:

```
while (i1 != i2) {
std::cout << *i1 << ' ';
++i1;
}
```

או לרוץ באמצעות לולאת for:

```
std::set<int> mySet;
// add some integers to mySet

for (std::set<int>::iterator it = mySet.begin();
it != mySet.end(); it++)
{
std::cout << *it << ' ';
}
```

אגב, במקום לעשות זאת בצורה זו, ניתן להמיר לתצורה הבאה, שנקראת range-based for loop:

```
std::map<std::string,int> dict;
for (auto entry : dict) {
std::cout << entry.first << " "
<< entry.second << std::endl;
}
```

לרוב הקונטיינרים של STL ישנו איטרטור מובנה.

למעשה, כאשר נממש איטרטור, נצטרך גם למשש מתודות קבועות שאינם משנים את הערכים:

```
const_iterator begin() const
const_iterator end() const
```

אך במקרה זה הקומפיילר לא יודע להכניס לאיטרטור איברים שאינם קבועים, ולכן נצטרך לייצר עוד שתי מתודות, על מנת לאפשר שימוש באיברים שאינם קבועים באיטרטור קבוע:

```
const_iterator cbegin() const
const_iterator cend() const
```

סך הכל נצטרך שש מתודות.

עלינו להבחין בין איטרטורים שרציפים בזיכרון או שאינם רציפים בזיכרון.

Iterator traits

לאיטרטור יש מספר תכונות שעלינו למשש על מנת שייקרא איטרטור סטנדרטי של סי. עלינו למשש 5 typedefים:

1. הערך של האיטרטור - value_type, למשל int.
2. הרפרנס של האיטרטור – reference – הסוג של הרפרנס.
3. המצביע של האיטרטור – pointer - סוג המצביע על האיטרטורים.

4. הגדרת ההבדל בין האיטרטורים – diffrenece_type – דרך להגדיר את המרחק בין האיטרטורים.
5. הסוג של האיטרטור – iterator_category. אחד מה-input_iterator_tag**:
   a. std::input_iterator_tag.
   b. std::output_iterator_tag.
   c. std::forward_iterator_tag.
   d. std:bidrectional_itrerator_tag.
   e. std::random_access_iterator.

אם נרצה לממש איטרטור, נצטרך לממש את כל הtraits הללו. למשל, עבור רשימה מקושרת:

```
class Iterator {
private:
 friend class ConstIterator; // to allow Conversion constructor from
Iterator to ConstIterator
 Node *node_;

public:
 /* ITERATOR TRAITS - must be defined (and public) in the iterator,
for it to work with all STL algorithms.
 * Comment this out and attempt to call std::find with this iterator
to watch the runtime error
 * you will get if you don't define iterator traits */
 typedef int value_type;
 typedef int &reference;
 typedef int *pointer;
 typedef std::ptrdiff_t difference_type; // irrelevant here, as we
have no difference - but still required
 typedef std::forward_iterator_tag iterator_category;

 Iterator(Node *node) : node_(node) {}

 Iterator &operator++() {
 node_ = node_->next_;
 return *this;
 }

 Iterator operator++(int) {
 Iterator it(node_);
 node_ = node_->next_;
 return it;
 }

 bool operator==(const Iterator &rhs) const { return node_ ==
rhs.node_; }

 bool operator!=(const Iterator &rhs) const { return node_ !=
rhs.node_; }

 reference operator*() { return node_->data_; }

 pointer operator->() { return &(operator*()); }
};
```

אלגוריתמים

ישנם מספר אלגוריתמים שמבוססים על איטרטורים. למשל:

```
auto result = std::find(v.begin(), v.end(), n1);
```

או:

```
std::sort(v.begin(), v.end());
```

למעשה, כמעט כל פעולה שנרצה לעשות קיימת באלגוריתמים של השפה, ולכן היתרון שבמימוש או בתמיכת איטרטורים הוא ענק.

מחלקות מקוננות

כך נוכל לייצר מחלקה בתוך מחלקה:

```
class List
{
 private:

 class Node
 {
 public:

 Node(int d):data_(d),next_(nullptr) {}

 int data_;

 Node* next_;
 };

 Node* pHead;
...
};
```

דבר זה יכול להיות משמעותי אם נרצה למשל לממש איטרטור בתוך המחלקה עצמה.

גנריות

Function overloading

בשונה מב-c, בשפת סיפיפי אפשר לכתוב אותו שם של פונקציה עם פרמטרים **שונים**. הקומפיילר למעשה יוצר שתי שמות שונים לפונקציות כאשר נבצע Overloading. למשל:

```
f(double) → _Z1fd
f(int) → _Z1fi
f(int, double) → _Z1fid
```

כיצד הלינקר יודע לאיזה פונקציות לקרוא? הלינקר קורא לפונקציה המתאימה ביותר:

1. קודם כל בוחר את כל הפונקציות עם **אותו שם**.
2. אחרי זה בוחר את **מספר הארגומנטים** המתאים ביותר.
3. לאחר מכן בוחר את **סוג המשתנים** המתאים ביותר.

ישנם כל מיני קונבנציות לגבי מעבר בין סוגים של משתנים. **אסור** לעשות overloading על ערך החזרה בלבד.

Templates

כאשר רצינו לייצר גנריות ב-c, השתמשנו ב*void - ובהמרה. ב++c נוכל להשתמש בגנריות בצורה שונה. באמצעות המילה השמורה template<typename T> נוכל להשתמש בסוג כללי בפונקציה עצמה (ניתן גם להחליף את typename ב-class):

```
void swap( T& a, T &b )

{
 T tmp = a;

 a = b;

 b = tmp;
}
```

כאשר הקומפיילר מזהה פונקציה גנרית, הוא מייצרת גרסה שעובדת לפונקציה עצמה לפי הסוג הספציפי שהפעלנו עליו. הרבה הפעלות עלולות לגרום לניפוח של הקוד.

עלינו לשים לב איזה **הנחות** אנחנו מניחים כשאנחנו מפעילים את הפונקציה בצורה גנרית. למשל בפונקציה לעיל הנחנו כי אופרטור ההשמה קיים.

נבחין כי קוד מסוג טמפלייט חייב להיות בקובץ h בעצמו, כי למעשה הוא סוג של הכרזה. בנוסף, שגיאות קומפליציה עלולות להתרחש על סוג ספציפי כפי שראינו קודם.

מחלקות גנריות

מה אם נרצה מחלקה שהיא בעצמה גנרית? נוכל לעשות זאת כך:

```
template <typename T>
class genericClass {
 T _member;
public:
 void print() const;
};
```

ובמימוש עצמו:

```
template <typename T> void genericClass<T>::print()
const
{
 std::cout << _member;
}
```

אם נגדיר שתי מחלקות עם פרמטרים גנריים שונים, נקבל למעשה שתי מחלקות שונות לחלוטין:

```
int main() {
 genericClass<int> intObj;
 genericClass<float> floatObj;
 ...
}
```

ואם נרצה להגדיר מספר פרמטרים?

```
template <class T1, class T2>
 class A {

 T1 _d;

 T2 _c;
};
```

נוכל להגדיר ערכים דיפולטיביים למחלקות:

```
template<class T = char, int Size = 1024>
 class Buffer
{

 T m_values[Size];
};
 Buffer<char> buff1;
 Buffer<> buff2;
// same as Buff1, needs <>
/Buffer<int, 256> buff3
```

אם נרצה, נוכל לממש גרסה עבור סוג ספציפי:

```
template <class Type> class A { ... };
template<> class A<char*>

{
 public:
 char* _data;

 A(char* data) : _data(data) { }

 bool isBigger(char* data);
};
```

ירושה

מחלקה היא למעשה קונספט כללי, למשל סטודנט, פרי, ועוד. הקונספט של ירושה מאפשר לנו לקחת מחלקה מסוימת ולירש ממנה תכונות מסוימות. למשל המחלקות תפוח ובננה יירשו מהמחלקה פרי, והמחלקה 'מתכנת' תירש מהמחלקה 'איש':

```
class Programmer : public Person
 {
 ...
};
```

לירושה יכולות להיות מספר modifiers שהם: private, public, protected. אם נשים משתנים מסוימים בתור Protected המחלקה היורשת תוכל לגשת אליהם. אם מדובר במשתני public במחלקת האב, כולם יכולים לגשת אליהם, ובפרט אם נגדיר את המחלקה היורשת להיות public.

העקרונות המשמעותיים בשלושת הגישות:

- אין גישה למשתנים הפרטיים מתוך המחלקה היורשת.
- מחלקה שיורשת public יורשת את משתני האב הpublic כpublic והProtected כprotected.
- מחלקה שיורשת protected יורשת את כל המשתנים כמו Protected.
- מחלקה שיורשת private יורשת את כל המשתנים בתור private.

דוגמה קטנה לאחד מהעקרונות הללו:

```
class A { public:
 int x;
};

class B: private A { public:
 int x;
 void F() {
 cout << x << ","
 << A::x << endl;
 }
};

int main ()
{
 A a;
a.x = 5; // OK
B b;
b.x = 2; // OK
 b.A::x = 5; // ERROR
b.F();
}
```

בלתי אפשרי לעשות ירושה מעגלית. כדאי להשתמש בירושה כאשר יש יחס של is a.

סדר הבנייה של המחלקות

כמו שהיינו חושבים, אכן קודם כל נבנית מחלקת האב וכל המשתנים שלה, ורק לאחר מכן נוצרת המחלקה היורשת. ב-destructor מדובר על סדר הפוך.

דוגמה לבנייה של שתי מחלקות, אב ויורשת:

```
class A
{
 int a_;
public:
 A(int a);
 ~A();
};

A::A(int a):a_(a)
{
 std::cout << “A ctor\n”;
}
A::~A()
{
 std::cout << “A dtor\n”;
}
class B : public A
{
 int b_;
public:
 B(int a, int b);
 ~B();
};
B::B(int a, int b) : A(a), b_(b)
{
 std::cout << “B ctor\n”;
}
B::~B()
{
 std::cout << “B dtor\n”;
}
```

דריסת פונקציות אב

במקום ליצור פונקציות חדשות מהיסוד, ניתן 'לדרוס' את מתודת האב באמצעות הרחבה של המתודה הקיימת. כלומר, קריאה למתודת האב, והוספה של פרטים אליה:

```
void Person::OutputDetails(ostream& os) const

{
 if(name_ != "") os<< " name: " << name_;

if(id_ != kNoIdVal) os << ", I.D: " << id_;
}

void Programmer::OutputDetails(ostream& os) const
{

Person::OutputDetails(os);

 os << “ [“ << company_ << “]“;
}
```

נבחין שקראנו למחלקת האב באמצעות Namespace שלה.

מתודיות וירטואליות ופולימורפיזם

לאיזה פונקציה מחליט הקומפיילר לקרוא? תלוי בסוג המצביע. למשל, בחלק הקודם אם יצרנו מצביע שהינו מסוג Person, גם אם נכניס אובייקט מסוג Progarmmer, עדיין הקומפיילר יתייחס אליו בתור Person.

לכך שהמצביע יכול להתייחס למחלקה היורשת בתור מחלקת האב, אנחנו קוראים פולימורפיזם – ריבוי צורות.

אם נסתכל בדוגמה הבאה (נניח כי Circle ו-Square יורשת מ-Shape):

```
Shape* MyShapes[2];

MyShapes[0] = new Circle();

MyShapes[1] = new Square();
MyShapes[0]->Draw();
```

יודפסו לנו המתודות של מחלקת האב, ולא המתודות הדורסות. הסיבה לכך קשורה למה שאמרנו קודם, שהמבציע הוא על מחלקת האב.

רזולוציה דינמית

על מנת למנוע בעיות כמו שקרו קודם, ננסה למצא פיתרון, באמצעות החלטה תוך כדי זמן הריצה לאיזו פונקציה לקרוא. דבר זה מתאפשר באמצעות המילה השמורה `virtual` – כך הקומפיילר יודע לקרוא לפונקציה לפי הטיפוס של האובייקט. למשל בדוגמה הקודמת, אם היינו מוסיפים `virtual` לכל המתודות, אזי הייתה נקראת המתודה של המחלקות היורשות ולא של האב.

חשוב להבחין כי אם אנחנו מבצעים השמה של מחלקה יורשת באמצעות מחלקת האב, אזי מתבצע 'חיתוך' וכל המתודות יקראו לאב. אמנם, השתמשנו ברפרנס בלבד, אזי נקרא למחלקה היורשת:

```
int main()
{
D d1;

 B b1 = d1;

 b1.F(); // prints B

D d2;

 B& b2 = d2;
```

```
 b2.F(); // prints D
}
```

גם כאשר נעביר ממתודה שאיננה וירטואל, למתודה שהינה וירטואל, נקרא למתודה המתאימה. מסיבה זו לא מומלץ לקרוא למתודיות וירטואליות מתוך הבנאי (עלול ליצור בעיות, מבחינה הוירטואליות וכו').

טבלה וירטואלית

כיצד זה מתקיים בפועל? אם מדובר במתודה שהינה וירטואל, כל אובייקט מחזיק מצביע למימוש של הפונקציה, בדומה למה שהיה לנו ב-c.

הבעיה היא כי כל מתודה וירוטואלית מחזיקה בעצם מצביע, ויש בזבוז של מקום. הפתרון הוא שכל אובייקט מחזיק מצביע לתוך טבלת מצביעי הפונקציות, וכך למעשה כל אובייקט מחזיק מצביע אחד, וכל מחלקה בפני עצמה מחזיקה את כל המצביעים (רק פעם אחת). דבר זה נקרא `virtual table`.

Destructors

מה קורה במצב בו נקרא לאובייקט היורש, באמצעות מחלקת האב, בעת מחיקת האובייקט?

```
class Base
{
 public:

 ~Base();

};
 class Derived : public Base
 { public:
 ~Derived();

}; Base *p = new Derived;
 delete p;
```

לכאורה, נקרא ל-destructor של האב! אבל זה לא מה שאנחנו רוצים. ולכן עלינו להפוך גם את הdestructor להיות וירטואלי. בעקבות כך, תמיד, אם יש לנו מתודות וירטואליות, צריך להפוך את הdestructor להיות וירטואלי.

Override

אם למשל ניצור מתודות במחלקה היורשת שדורסת את מחלקת האב, אבל עם פרמטר שונה (למשל `int` במקום `float`), אז הקומפיילר לא יזהה שמדובר באותה מתודה והוא יצור מתודה חדשה. בשביל לפתור בעיה זו, נשתמש במילה השמורה `override` על מנת לומר לקומפיילר שאנחנו אכן דורסים את המתודה – במצב זה, הקומפיילר יכעס אם לא נשתמש באותו פרמטר ("אתה לא באמת דורס את המתודה"). לכן כדאי לשים בכל מצב override.

Final

אם נרצה ליצור מצב בו המחלקה היורשת לא תוכל לדרוס את המתודה של האב, נשתמש במילה השמורה `final`. ניתן גם להשתמש במילה השמורה `final` על מחלקה, ואז בלתי אפשרי לרשת ממנה. נשתמש בזה רק אם נהיה חייבים.

מחלקות אבסטרקטיות

אם נתבונן בדוגמה שראינו מקודם, shape הינו למעשה אבסטרקטי ואין לו באמת מתודה של draw. לכן, נרצה למנוע מהמשתמש ליצור אובייקט מסוג זה. ב-c השתמשנו ב-asser(false), אבל זה דרך של c. ב-++c נוכל להגדיר מחלקה שהיא כולה וירטואלית:

```
class Shape
{ public:
 virtual ~Shape();
Virtual destructor
virtual void Draw() = 0; //Pure virtual
virtual double Area() = 0; //Also
};
```

כעת, אם נרצה ליצור את האובייקטים:

```
Shape* p; // legal

Shape s; // illegal

Circle c; // legal

p = &c; // legal
Shape& s = c; // legal

 p = new Shape; // illegal
```

מצביעים חכמים – Smart pointers

אחד האתגרים המרכזיים בשפות c ו-c++ היא הניהול של הזיכרון.

עד כה, המודל שאיתו עסקנו היה המודל של 'אחריות אישית' – אחרי שהקצנו זיכרון דינמי, היה עלינו למחוק אותו. אם למשל למחלקה string יש משתנה דינמי בשם data, אז string הוא הבעלים של data. אם הגדרנו את כל הדברים כפי שלמדנו, אז אנחנו לא צריכים לחשוש לזיכרון, ולמעשה נאמר כי המחלקה היא memory-tight. הבעיה היא שזה מונע מאיתנו גמישות מסוימת.

בעקבות כך, אפשר להציע רעיון אחר שנקרא dynamic ownership. מצביעים צריכים 'בעלים' שישחררו את הזיכרון שלו, אך כעת הבעלות משתנה באופן דינמי. אמנם, מה שראינו עד עכשיו היה שהבעלות השתנתה בזמן הקומפליציה בלבד, אך גם זה עלול ליצור בעיות. למשל, אם ישנה פונקציה שמחזירה מצביעה מסוים, אז הפונקציה שמקבלת את הפונקציה הקודמת צריכה לשחרר את הזיכרון.

הפיתרון שלנו הוא ליצור אובייקט שבזמן ריצה מנהל את הזיכרון של האובייקט האחר, ושיש לאובייקט זה גמישות. כלומר, נרצה מצביע חכם, שידאג לשחרור זיכרון וכו' למצביע המקורי. על מנת ליצור מצביע חכם כזה, נצטרך את כל המתודות הבאות:

```
class pointer {   public:
        pointer( T* );
        pointer( pointer const& );
        pointer& operator=( pointer const& );
        T& operator*() const;
        T* operator->() const;
}
```

נוכל להבחין שיש כאן המון מתודות שנותנים 'תחושה' של מצביע. ל-STL ישנם מספר סוגים של מצביעים חכמים:

1. std::unique_ptr – מחלקה שמאפשרת להעביר בעלות באופן דינמי, אבל רק מצביע אחד יכול להיות בעלים של מצביע כלשהו.
2. std::shared_ptr – מאפשר למספר בעלים לחלוק בעלות על מצביע בודד.
3. std::weak_ptr – הוא איננו הבעלים אבל הוא מאפשר לעשות מספר פעולות.

נתחיל להתבונן ב-unique_ptr.

std:unique_ptr

נתבונן במימוש הבא של המצביע:

```
template <typename T>
class unique_ptr
{
public:
  explicit unique_ptr( T* p ) : _p (p) {}
  T& operator*() const { return *_p; }

  T* operator->() const { return _p; }
  T* get() const { return _p; }
  ~auto_ptr(){ delete _p; }
  //other methods
private:
  T* _p; // actual pointer ("raw" pointer)
};
```

ונשתמש בו כך:

```
#include <memory>
struct bar { /* ... */ };
void foo()
{
   std::unique_ptr<bar> my_bar
( new bar(arg1, arg2) );
   // use my_bar as any raw pointer
bar->do_something();
}  // my_bar is deleted automatically
```

בעקבות כך, איננו צריכים לזכור למחוק את המצביע, כיוון שהאובייקט החדש ששמנו בתור דואג לעשות זאת היטב.

בדרך אחרת, נוכל להשתמש כך:

```
#include <memory>
struct bar { /* ... */ };
void foo()
{

   std::unique_ptr<bar> my_bar =  std::make_unique<bar>(arg1, arg2);

   // use my_bar just like any raw pointer
bar->do_something();
}  // my_bar is deleted automatically
```

`make_unique` הוא סוג של `new` משודרג.

העברת בעלות

על מנת למנוע מחיקות מיותרות, ישנה בעלות יחידה. לכן עלינו ליצור העברת בעלות, באמצעות std::move(..). מבחינה תכנותית, נבצע זאת כך:

```
template <typename T>
class unique_ptr
{
public:
    explicit unique_ptr( T* p ) : _p (p) {}
```

```
  // move-constructor:

    std::unique_ptr( std::unqiue_ptr&& rhs );
    // move-assignment operator:

    unique_ptr& operator=( unique_ptr&& rhs );      …
private:
    T* _p; // actual pointer (“raw” pointer)
};
```

הסימן && הוא מעין סוג משוכלל של רפרנס. במקום להעתיק, אנחנו מעבירים בעלות. במימוש עצמו, מתבצעת קריאה לפונקציית `release()` של האובייקט המקורי, שבעצם מוחקת את הבעלות של הקודם. גם האורפטורים האחרים מתנהגים בצורה דומה.