## תכנות בסיסי ב - C++

### ההיסטוריה של C++ על קצה המזלג

בתחילת שנות השמונים חיפש חוקר בשם Bjarne Stroustrup במעבדות Bell דרכים לשפר את שפת C ולהוסיף לה מאפיינים שימושיים שהכיר משפות אחרות. התוספת העיקרית שהחליט להוסיף לשפה היא תמיכה באובייקטים ולכן השפה שהגדיר נקראה "C with classes" .
בהתחלה, Stroustrup לא כתב קומפיילר מ- "C with classes" לשפת מכונה, אלא פשוט מ – "C with classes" ל – C. בשביל להריץ תוכנית בשפה החדשה היה צורך להפעיל שני קומפיילרים בשרשרת : את הקומפיילר החדש וקומפיילר רגיל של C על התוצאה. בשלב מאוחר יותר, שונה שם השפה ל – C++ ונכתב קומפיילר ישירות לשפת מכונה. השם C++ הוא מעין הלצה המסמלת את השדרוג של C ב-1. יש הטוענים שהשפה הייתה צריכה להיקרא C++ מכיוון שערך השפה הוא הערך לאחר ההוספה. המקפידים ביותר טוענים שהשפה הייתה צריכה להיקרא C+1 מכיוון שהשפה C לא שונתה.

### C++ ביחס ל – C

במקור, C++ נבנתה כהרחבה של – C. גם היום, למעט מספר יוצאים מהכלל, אפשר ליראות ב – C++ הרחבה של - C. בגלל תכונה זו, כמעט כל תוכנית ב – C יכולה להיחשב תוכנית C++ תקינה, ולהתקמפל ע"י קומפיילר של C++. מובן שההפיך אינו נכון.

C++ הוסיפה ל - C מאפיינים רבים. אחד המאפיינים החשובים שהוסיפה הוא מושג האובייקט ומושג המחלקה. ב - C קיימים מבנים (struct) המאפשרים הגדרת טיפוסים שמשתנה מסוגם מהווה קיבוץ של משתנים קיימים. ב- C++ אפשר להגדיר מחלקות המאפשרות לקבץ משתנים **ופונקציות** הפועלות עליהם. C++ הוסיפה גם תמיכה באנקפסולציה, הסתרת מימוש, ירושה ופולימורפיזם. מלבד תמיכה בתכנות O.O נוספו לשפה גם templates, כלי המאפשר לכתוב קוד גנרי בעזרת compile-time-polymorphism. תוספות רבות של C++ נועדו לשפר את השפה ולאו דווקא לתמוך בפרדיגמות תכנות חדשות. דוגמאות לתוספות כאלה הם : const ,references ,variables ,inline functions ,default parameters values ו- operators overloading.
הספרייה הסטנדרטית של C++ כוללת את הספרייה הסטנדרטית של C ומוסיפה שירותים רבים נוספים.
כפי שאפשר להבין C++ היא שפה מורכבת בהרבה מ - C.

### "hello lord" ב - C++

בגלל התוספות הרבות של C++, סגנון הכתיבה המקובל לתוכניות שכבר ראינו, הוא שונה.
לדוגמה, האופן המקובל לכתוב תוכנית המדפיסה "hello lord" על המסך ב - C++ הוא כזה :

```
#include <iostream>

int main() {
   std::cout << "hello lord!" << std::endl;
}
```

הסבר :
הקובץ iostream הוא קובץ header הכולל שירותי קלט\פלט באמצעות זרמים. הקובץ הוא חלק מהתוספות שנוספו לספרייה הסטנדרטית. השירותים שהוא מציע מהווים אלטרנטיבה לשירותי stdio.h.
std::cout הוא אובייקט גלובאלי המייצג זרם פלט לערוץ הפלט הסטנדרטי. הרישא std היא מרחב השם שהוא שייך עליו ונועד לסמן שהאובייקט הוא חלק מהספרייה הסטנדרטית. הסימון '<<' הוא אופרטור שקיבל משמעות מיוחדת בהקשר לזרמי קלט. std::endl הוא אלמנט הגורם לירידת שורה בהדפסה.

המושגים "זרם פלט", "אובייקט גלובאלי", "מרחב שם", "אופרטור שקיבל משמעות חדשה" עדיין לא נלמדו בצורה מסודרת ולכן קשה להבין את התוכנית בצורה מלאה. בשלב זה מספיק לדמיין את std::cout כמעיין צינור המוביל אל המסך ואנו שולחים לו אלמנטים שונים להדפסה בעזרת האופרטור '<<'.
שימו לב שבתוכנית זו לא כתבנו return 0 בסוף ה - main. ב - C++ מחזירה main את הערך 0 גם אם לא כותבים זאת מפורשות.

**קומפילציה**

הקומפיילר שבו נשתמש לקומפילציה של תוכניות C++ ניקרא 'g++'. קומפיילר זה שייך לאוסף הקומפיילרים של GNU שגם gcc שייך אליו. למעשה, לא מדובר בקומפיילרים שונים לגמרי אלא במנגנון זהה שהותאם לשפות שונות. השימוש ב - g++, דומה מאוד לשימוש ב – gcc. בפרט, לדגלים Wall-, c- ו - o- יש אותה המשמעות בשניהם.

הסיומות המקובלות לקובצי C++ הם : cpp, cc, c++ ו cxx. בקורס זה נשתמש בסיומת : cpp.
בהנחה שהתוכנית המדפיסה !hello lord נמצאת בקובץ בשם "check.cpp" , הקומפילציה תתבצע באופן הבא :

```
$ g++ -Wall check.cpp -o check
```

וההרצה :

```
$ check
```

**קלט ב - C++**

בתוכנית הקודמת השתמשנו בזרם הפלט std::cout ובאופרטור '>>' למטרות פלט. באופן דומה, נשתמש בזרם הקלט std::cin ובאופרטור '<<' , למטרות קלט. הקוד הבא מדגים שימוש פשוט בקלט מהמשתמש :

```
#include <iostream>
#include <cmath>

int main() {
   std::cout << "please enter a,b and c of 'ax^2+bx+c:'" << std::endl;
   double a,b,c;
   std::cin >> a >> b >> c;
   double discriminant = sqrt(b*b-4*a*c);
   std::cout << "z1 = " << (-b+discriminant)/(2*a) << std::endl
         << "z2 = " << (-b-discriminant)/(2*a) << std::endl;
}
```

הסבר :
כאמור הכללת iostream מאפשרת שירותי זרמי קלט\פלט. cmath הוא קובץ המאפשר שירותים זהים לקובץ math.h הזכור מ - C. הסיבה לכפילות היא ניסיון ליצור סטנדרט שמות חדש לקובצי הספרייה הסטנדרטית של C++.

ב- C יש דרישה להצהיר על כל המשתנים הלוקאליים בתחילת הפונקציה. כפי שניתן ליראות בדוגמה זו, ב - C++ אין דרישה כזו.
הפקודה :

```
std::cin >> a >> b >> c;
```

גורמת לקליטה של שלושה משתני double מהמשתמש. המספר הראשון יוכנס ל - a, השני ל-b והשלישי ל - c.
אין הגבלה למספר האלמנטים הניתנים לשרשור בעזרת >> ו - <<.

כפי שניתן להבין, תוכנית זו קולטת מקדמים של משוואה ריבועית ופולטת את שורשיה.
כאשר משתמשים ב - g++ אין צורך לכתוב במפורש שאנו מבצעים linking לחלק המתמטי של הספרייה הסטנדרטית, כלומר אין צורך בתוספת : lm- בשלב ה - linking.

בניגוד לפונקציות printf ו- scanf, אין צורך לציין במפורש מהו הטיפוס של המשתנה הנקלט או הנפלט. לדוגמה:

```
#include <iostream>

int main() {
   char buf[100];
   int a;
   double b;
   std::cin >> buf >> a >> b;
   if(std::cin.fail()) {
      std::cerr << "input problem\n";
      return 1;
   }
   std::cout << "I got: "<< buf << ' '
         << a << ' ' << b << std::endl;
}
```

התוכנית קולטת מחרוזת, מספר שלם ומספר double ומדפיסה אותם. במקרה שהאינפורמציה שהוכנסה לא מתאימה לטיפוס המשתנה המקבל אותה, תודפס הודעת שגיאה וההרצה תיגמר.
דוגמה למקרה כזה הוא הקלט הבא:

```
32 hello 32
```

במקרה זה, תיווצר בעיה בקליטת המשתנה a. הדבר יגרום לאובייקט std::cin למצב פנימי לא תקין. הפונקציה הבולאנית (הפרדיקט) fail תציין שארעה בעיה בקליטת משתנה.
האובייקט הגלובאלי std::cerr מייצג זרם פלט לערוץ השגיאות, אותו ערוץ אליו נהגנו להתייחס דרך stderr. ההדפסה לערוץ השגיאות תופיע על המסך גם אם הפלט הסטנדרטי מופנה לקובץ.

## Refrences

Reference הוא מאפיין ב – C++ המאפשר מתן שם נוסף למשתנה מסוים.
נתבונן בדוגמה:

```
int main() {
        int a = 5;
        int& b = a;
        b++;
        std::cout << a;
}
```

השורה: int& b=a לא הגדירה משתנה חדש, אלא רק שם חדש למשתנה a. לאחר ביצוע שורה זו, יהיו לאותו תא בזיכרון שני שמות: a ו-b. לאחר ש – b הוצהר כשם נירדף, ההתייחסות אליו היא כאל משתנה רגיל מסוג int (ולא – int&). הפלט של הקוד הנ"ל יהיה לכן: 6.

אם נוסיף את השורה:

```
std::cout << &a << " " << &b;
```

נקבל הדפסה כפולה של אותה כתובת בזיכרון.
יש לשים לב למשמעויות השונות של התו - '&' (ampersand):
פעם ב- "int& b" המציין "b הוא שם נירדף" ופעם ב – "&a" המציין "הכתובת של a".

לאחר ששם מסוים הוגדר כשם נוסף למשתנה, אי אפשר להעביר אותו להיות שם נוסף של משתנה אחר. הדוגמה הבאה ממחישה זאת:

```
int main() {
        int a = 5,c=17;
        int& b = a;
        b=c;
        c = 3;
        std::cout << a <<" " << b <<" " << c << std::endl;
}
```

הפקודה: "b=c" איננה הופכת את b לשם נירדף של c אלא משימה את הערך שב – c לתוך התא בזיכרון אליו ניתן להתייחס ע"י b וע"י a. הפלט לכן יהיה: 17,17,3.

reference הוא אחד התוספות ש – C++ הוסיפה על C, ב- C הוא איננו קיים.

השימוש הנפוץ ביותר ל – references הוא בהעברת פרמטרים לפונקציה ובקבלת ערך מוחזר מפונקציה.

**references כפרמטר לפונקציה**

ניתן להשתמש ב – reference על מנת להעביר משתנה לפונקציה ללא שיכפול שלו:

```
void f(double& a) {
	a*=2;
}
int main() {
	double t=3.5;
	f(t);
	std::cout << t;
}
```

במקרה זה a יהיה שם נירדף ל – t, והכפלתו פי 2, תגרום ל-t להיות 7.
למעשה, ממומשת העברת פרמטרים by reference ע"י העברת מצביע. קומפיילר של C++ יתרגם את הקוד הנ"ל לקוד דומה לקוד שייצור כתוצאה מתרגום הקוד הבא:

```
void f(double* a) {
	(*a)*=2;
}
int main() {
	double t=3.5;
	f(&t);
	std::cout << t;
}
```

יש הטוענים שהקוד האחרון עדיף מכיוון שהוא מבהיר לקורא כי המשתנה שנישלח (t) יכול להשתנות.

**reference כערך מוחזר מפונקציה**

reference יכל לשמש גם כערך מוחזר מפונקציה. המשמעות היא שהקריאה לפונקציה תהיה שם נרדף למשתנה שהוחזר ממנה. לדוגמה:

```
int a = 6; // global variable
int& f() {
	return a;
}
int main() {
	f()++; // 'f()' is another name for a
	std::cout << a;
}
```

מכיוון ש – f() משמשת כאן כשם נירדף למשתנה a, הפלט יהיה 7.

בתוכנית זו a הוא משתנה גלובאלי. משתנה גלובאלי הוא משתנה הקיים כל זמן ריצת התוכנית ונגיש מכל מקום בקוד (במקרים מסוימים, צריך רק להצהיר שוב על קיומו).

ב - C++ , כל הכתוב מהסימן // עד סוף השורה, נחשב כהערה.

בהמשך ניראה דוגמאות בהן השימוש ב - reference כערך מוחזר יכל להיות נח מאד.

### ביטויים בולאניים ב C++

ב - C++ קיים טיפוס לערכים בולאניים ששמו bool. הערכים שביטוי מטיפוס bool יכל לקבל הם true ו false. המוסכמות הנהוגות ב - C לגבי התייחסות למספרים כביטויים בולאניים, קיימות גם ב - C++. לדוגמה:

```
#include <iostream>

int main() {
   int a = 5;
   bool isZero = (a == 0);
   if(!isZero && isZero==false && isZero!=true && !!!isZero && a ) // all the conditions are the same
      std::cout << "a is not zero\n";
}
```

### הקצאות דינמיות ב - C++

כזכור, ב - C הקצאת דינאמית של זיכרון התבצעה ע"י שימוש בפונקציה malloc ושחרור זיכרון ע"י free. ב - C++ הוספו האופרטורים new ו - delete שנועדו להחליף את malloc ו- free. ניתן דוגמה לאופן השימוש בהם:

```
int main()
{
   int *p = new int;
   delete p;
   p = new int(7);
   delete p;
   p = new int[70];
   delete [] p;
}
```

הפקודה הראשונה מקצה דינאמית int בודד שערכו אינו מוגדר. הפקודה השניה משחררת את הזיכרון שהוקצה בראשונה.

כפי שניתן ליראות, השימוש ב - new נח יותר מהשימוש ב - malloc. ב - new אין צורך לפרט כמה byte רוצים להקצות ואין צורך לבצע המרה של טיפוסים.

הפקודה השלישית מקצה int בודד שערכו מאותחל ל - 7 והפקודה הרביעית משחררת אותו.

הפקודה החמישית מקצה רצף של 70 int לא מאותחלים והפקודה השישית משחררת את רצף הזיכרון הזה. יש לשים לב לתחביר השונה עבור שחרור מערכים. שימוש בתחביר רגיל עבור שחרור מערך או שימוש בתחביר של מערך עבור שחרור הקצאה יחידה, אינו שחרור תקין.

### Functions overloading

ב - C++ פונקציה מאופיינת ע"י שמה ורשימת הטיפוסים שהיא מקבלת. ניתן להגדיר שתי פונקציות שונות בעלות אותו השם אך נבדלות ברשימת הטיפוסים שהן מקבלות. לדוגמה:

```
#include <iostream>

void foo() { std::cout << "foo()\n"; }
void foo(int n) { std::cout << "foo(" << n << ")\n"; }

int main() {
```

```
   foo(12);
   foo();
}
```

פלט:

```
foo(12)
foo()
```

הגדרת מספר פונקציות עם אותו השם נקראת function oveloading, והיא שימושית מאד כאשר מדובר בפונקציות המבצעות פעולה דומה מבחינה קונצפטואלית אך על טיפוסים שונים.

**מחלקות ואובייקטים ב - C++**

כאמור, הוספת התמיכה באובייקטים היוותה את המוטיבציה העיקרית להרחבת C ל - C++.
ניתן דוגמה בסיסית וסינתטית להגדרת מחלקה ויצירת אובייקט ב - C++:

```
#include <iostream>

// 1. the class declaration

class A {
   int _a;
   double *_ptr;
   void foo();
 public:
   float _b;
   A(int a);
   void foo1();
   ~A();
 private:
   void foo(int n);
}; // the ';' is mandatory

// 2. the implementation of the class's functions

A::A(int a): _a(a),_ptr(new double(5.7)) {
   std::cout << "A ctor was invoked\n";
}

A::~A() {
   std::cout << "A dtor was invoked\n";
   delete _ptr;
}

void A::foo() { std::cout << "bla bla\n"; }

void A::foo1() {
   foo();
   std::cout << *_ptr << ' ' << _a << ' ' << _b << std::endl;
   foo(2);
}

void A::foo(int n) {
   std::cout << n << std::endl;
}

// 3. main

int main() {
   A a(7);
   a._b = 66;
```

```
   a.foo1();

   A* p = new A(8);
   p->_b = 77;
   p->foo1();
   delete p;
}
```

פלט:

```
A ctor was invoked
bla bla
5.7 7 66
2
A ctor was invoked
bla bla
5.7 8 77
2
A dtor was invoked
A dtor was invoked
```

הסבר:
התוכנית מורכבת מהצהרה על מחלקה, מימוש של הפונקציות שלה ומהפונקציה main.

1. ההצהרה
ההצהרה גורמת ליצירת טיפוס חדש בשם A. משתנה (אובייקט) מסוג A יכיל בתוכו משתנים מסוג int, *double ו - float. משתנה כזה ניתן יהיה להיבנות, להיהרס ולהפעיל את הפונקציה foo1 שלו.
לצורכי מימוש פנימי, קיימות לאובייקט הפונקציות foo() ו- foo(int).
המילים private ו public מגדירות אזורים של members פרטיים וציבוריים בהתאמה. הרשאת ברירת המחדל היא private. ה - members: _a, _ptr, foo() ו - foo(int) הם בעלי הרשאת private ואילו ה members: _b, A(int), ~A() ו - foo1() הם בעלי הרשאת גישה public.
כפי שניתן ליראות, ניתן להגדיר כמה אזורים שונים בעלי אותה הרשאה.
הפונקציה A(int a) היא constructor (בקיצור ctor) המפרט איך בונים אובייקט מסוג A כאשר ניתן int בודד. הפונקציה ~A() נקראת destructor והיא אחראית על הריסת האובייקט. הריסת אובייקט לוקאלי מתבצעת בזמן שניגמר ה - scope שלו. הריסה של אובייקט המוקצה דינמית מתבצעת כאשר מבצעים delete על מצביע עליו.

2. מימוש הפונקציות
בניגוד לתחביר של Java, ב - C++ ימומשו הפונקציות של המחלקה בד"כ מחוץ להצהרתה. בכדי לציין לאיזה מחלקה שייכת פונקציה מסוימת, יש להוסיף את שמה בליווי :: לפני שם הפונקציה. זהו בעצם שמה המלא של הפונקציה. בתוך ההצהרה יכולנו להסתפק בשם מקוצר (ללא ה - A::) מכיוון שהיה ברור לאיזה מחלקה הפונקציה שייכת.
סדר מימוש הפונקציות בשלב זה אינו חשוב. ההצהרה על כל הפונקציות בשלב ההצהרה על המחלקה, גרמה לכולן להיות מוכרות לקומפיילר ולכן במימוש ניתן לקרא לכל פונקציה מכל פונקציה ללא קשר לסדר מימושן.
נתבונן על מימוש ה - ctor:

```
A::A(int a) : _a(a),_ptr(new double(5.7)) {
   std::cout << "A ctor was invoked\n";
}
```

קטע הקוד המודגש, ניקרא רשימת אתחול (initialization list). רשימת האתחול מתחילה בתו ':' המופיע לאחר שם הפונקציה ולאחר מכן יש סידרה של הפעלות ctor-ים של data members. תחילת מימוש הפונקציה ('{') מסיימת את רשימת האתחול.
רשימת האתחול אחראית לפרוט אופן הבניה של המרכיבים מהם בנוי האובייקט. אופן הבניה נקבע ע"י הפעלות ctor.
מכיוון שכאן מדובר רק באתחול של int ו - pointer, ניתן היה לכתוב את הקוד גם כך:

```
A::A(int a)  {
  _a = a;
  _ptr = new double(5.7);
  std::cout << "A ctor was invoked\n";
}
```

בהמשך ניראה מקרים אחרים בהם חייבים להשתמש ברשימת האתחול.

המימוש של שאר ה - member functions אינו כולל מאפיינים חדשים.

3. main

בפונקצית ה - main נוצרים שני אובייקטים מסוג A, האחד לוקאלי והשני מוקצה דינמית.
בכל פעם שנוצר אובייקט חדש מסוג A, ה ctor האחראי לבנייתו מדפיס הודעה על המסך.
ההדפסה הראשונה

```
A ctor was invoked
```

היא תוצאה של בניית המשתנה הלוקאלי.
שלוש ההדפסות הבאות הן תוצאה של פעולת foo1 של המשתנה הלוקאלי. הפונקציה foo1 מפעילה שתי פונקציות פרטיות הגורמות לחלק מהדפסות.
ההדפסה הבאה:

```
A ctor was invoked
```

היא תוצאה של בניית המשתנה הדינמי. הפעלת foo1 שלו גורמת לשלוש ההדפסות הבאות.
בשלב זה יש הריסה יזומה של האובייקט הדינאמי.
כעת יש דרישה מפורשת להריסה של האובייקט הדינאמי. ה - destructor של A ניקרא וההודעה הבאה מודפסת:

```
A dtor was invoked
```

לאחר מכן, הפונקציה main מסתיימת, האובייקט הלוקאלי a נהרס אוטומטית, ה - dtor שלו מופעל ושוב מודפסת הודעת ה - dtor על המסך.

### עיקרון האנקפסולציה encapsulation

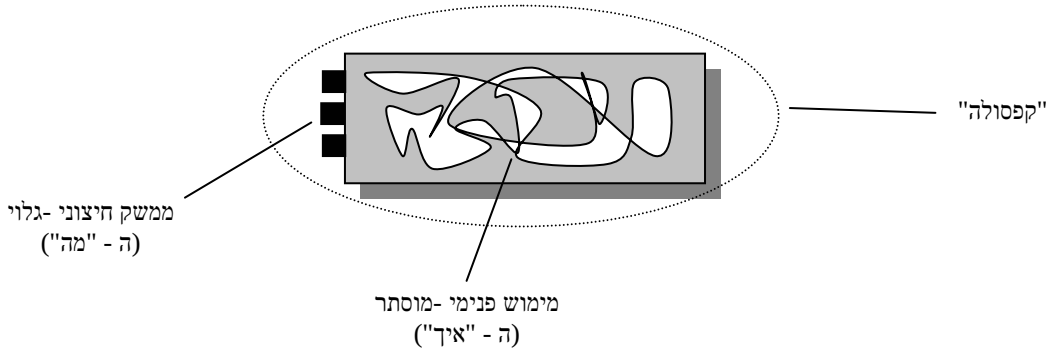
הרשאות הגישה מאפשרות יישום של עיקרון מרכזי בתכנות מונחה עצמים – עיקרון האנקפסולציה. הרעיון הוא כתיבה של קוד המהווה מעיין יחידה סגורה ושלמה בעלת תפקוד מוגדר היטב. ליחידה כזו אנחנו מתייחסים כאל "קפסולה" סגורה בעלת ממשק נוקשה וידוע אך מימוש פנימי, מוסתר ולא ידוע.

לרעיון של הסתרת המימוש יש שני יתרונות חשובים:

1. **שמירת מצב פנימי תקין**. כאשר מקפידים על ממשק קשיח המאפשר אוסף פעולות מוגבל על האובייקט, אפשר להבטיח כי כל שימוש באובייקט לא יפגום בעקביות המצב הפנימי שלו. בדוגמה של רשימה משורשרת, הממשק שנגדיר, מבטיח רשימה רצופה ללא הצבעות שגויות. ליתרון הזה יש חשיבות גדולה בפרוייקטים גדולים הנכתבים ע"י מספר אנשים. האובייקט מתוכנת פעם אחת בצורה מסודרת עם ממשק קשיח ואז ניתן למעשה לשכוח מהמימוש הפנימי שלו ולהשתמש בו כקופסא שחורה. אם לא דואגים לממשק כזה, אז המבנה הפנימי של האובייקט עלול להיפגע משימוש שנכתב ע"י אנשים שאינם מתמצאים בפרטי המימוש או סתם בגלל טעות.
2. **מודולריות.** כאשר אנו מפרידים את **מה** שהאובייקט עושה מ **איך** שהוא עושה זאת, אנחנו מרוויחים את האפשרות להחליף אובייקטים המתנהגים בצורה זהה אך ממומשים בצורה שונה. אם לדוגמה נכתוב קוד ארוך מאד המשתמש ברשימה שכתבנו ואז נהייה מעוניינים להשתמש ברשימה המתנהגת באופן זהה אך ממומשת ע"י מערך, נוכל פשוט להחליף את ה - class List שלנו ב - class List האחרת. לא יהיה צורך לשנות את כל הקוד הארוך המשתמש ברשימות כיוון שכל ההתייחסות אליהן היא דרך ממשק שגם הרשימה החדשה עונה עליו.

היתרון הזה מאפשר שיפור מערכות ללא שינויים גדולים. לפעמים יש trades off בין יעילות ריצה ובין צריכת זיכרון, המודולריות מאפשרת החלפה בין אובייקטים בעלי יתרונות שונים. ניתן ליראות את היתרונות שבתכנון מודולרי גם בתחומי הנדסה שאינם הנדסת תוכנה, גלגל

מכונית לדוגמה מתחבר לרכב באופן המוגדר היטב ולכן ניתן להרכיב ברכב גלגלים המיוצרים ע"י חברות שונות.




**המצביע this**

נתבונן בשתי השורות הבאות של ה - main האחרון:

```
a.foo1();
…
p->foo1();
```

שתי הפקודות גורמות להפעלה של הקוד הבא:

```
void A::foo1() {
   foo();
   std::cout << *_ptr << ' ' << _a << ' ' << _b << std::endl;
   foo(2);
}
```

בפעם הראשונה ש foo1 הופעלה, היא הופעלה דרך האובייקט הלוקאלי של main. המשתנים a_ ו - b_ הם משתנים הנמצאים על המחסנית. בפעם השניה ש foo1 הופעלה a_ ו b_ הם משתנים **אחרים** הנמצאים על הערמה. אם היינו מגדירים אובייקטים נוספים (על הערמה או המחסנית) ומפעילים דרכם את foo1, a_ ו - b_ היו משתנים אחרים במקומות שונים בזיכרון. נשאלת השאלה איך הפעלת **אותו** קוד מתייחס כל פעם ל a_ ו b_ אחרים.
התשובה היא שבכל הפעלה של member function נישלח משתנה נוסף, סמוי. משתנה זה הוא מצביע לאובייקט הספציפי שדרכו הופעלה הפונקציה. מצביע זה ניקרא this.
את ההפעלה הראשונה של foo1, מתרגם הקומפיילר לקוד הדומה לקוד הבא:

```
A::foo1(&a);
```

את ההפעלה השניה הוא מתרגם באופן דומה:

```
A::foo1(p);
```

את הפונקציה foo1 מבין הקומפיילר כך:

```
void A::foo1(A *this) {
   foo(this);
   std::cout << *(this->_ptr) << ' ' << this->_a << ' ' << this->_b << std::endl;
   foo(2,this);
}
```

כעת ניתן להבין איך שתי הפעלות של אותה פונקציה foo1 מאפשרות התייחסות למשתנים שונים. הביטוי this->_a לדוגמה, יתייחס למקום בזיכרון התלוי בערך של this.

בשיעור הקודם ראינו דוגמה לרשימה משורשרת בשפת C. בדוגמה זו, כל פונקציה המבצעת מניפולציה ברשימה, קיבלה מצביע לרשימה הספציפית עליה היא צריכה לעבוד. כפי שניתן להבין, הקומפיילר שכתב stroustrup עבור הגרסאות הראשונות של C++ (הקומפיילר מ - C with classes ל-C), פשוט היה צריך לבצע תרגום דומה לתרגום שביצענו כאן.

this היא מילה שמורה ב - C++ המאפשרת התייחסות לאותו מצביע סמוי. ניתן לדוגמה, להוסיף ל - A את ה - member function הבאה:

```
bool A::isSameObject(A &other) {
        std::cout << "my address is: " << this << "\n his address is: " << &other << std::endl;
        return this == &other;
```

```
}
```

ואז אם מוסיף ב - main את השורות הבאות:

```
A a1(1),a2(2);
if(a1.isSameObject(a1)) {…}
if(a1.isSameObject(a2)) {…}
```

יבוצע הקוד של ה - if הראשון ולא יבוצע הקוד של ה - if השני.

**חלוקה לקבצים ב - C++**

החלוקה זהה רעיונית לחלוקה ב - C. הצהרות בקבצי header בעלי סיומת h. או hh. ומימושי הפונקציות והגדרות של משתנים גלובאליים בקבצי cpp. את הדוגמה הסינתטית שלנו ניתן לחלק לקבצים באופן הבא:

## file A.h:

```
#ifndef A_H
#define A_H
class A {
   int _a;
   double *_ptr;
   void foo();
 public:
   float _b;
   A(int a);
   void foo1();
   ~A();
 private:
   void foo(int n);
};
#endif
```

## file A.cpp:

```
#include "A.h"
A::A(int a): _a(a),_ptr(new double(5.7)) {
   std::cout << "A ctor was invoked\n";
}

A::~A() {
   std::cout << "A dtor was invoked\n";
   delete _ptr;
}

void A::foo() { std::cout << "bla bla\n"; }

void A::foo1() {
   foo();
   std::cout << *_ptr << ' ' << _a << ' ' << _b << std::endl;
   foo(2);
}

void A::foo(int n) {
   std::cout << n << std::endl;
}
```

## file main.cpp:

```
#include "A.h"
int main() {
   A a(7);
   a._b = 66;
   a.foo1();

   A* p = new A(8);
```

```
    p->_b = 77;
    p->foo1();
    delete p;
}
```

ה - makefile המתאים לפרוייקט כזה ייכתב בדיוק באותו אופן שבו נכתב makefile לפרוייקט ב - C למעט סיומות קבצי המימוש : cpp. במקום c.

**עץ בינארי ממוין ב - C++**

נביא כעת דוגמה מעשית לאובייקט המייצג מבנה נתונים ב - C++. ניתן היה להמיר בקלות יחסית את הקוד של הרשימה המשורשרת מהשיעור הקודם אך במקום זאת נביא דוגמה לקוד של עץ בינארי ממוין :

```
#include <iostream>

struct Node {
    Node *_ls, *_rs;
    int _data;
    Node(int data);
};

Node::Node(int data):_ls(0),_rs(0),_data(data) {}

class Tree {
    Node* _root;
    void print(Node* p);
    void destruct(Node* p);
  public:
    Tree();
    void insert(int data);
    void print();
    ~Tree();
};

Tree::Tree():_root(0) {}

Tree::~Tree() {
    destruct(_root);
}

void Tree::destruct(Node* p) {
    if(!p)
        return;
    destruct(p->_ls);
    destruct(p->_rs);
    delete p;
}

void Tree::print() {
    print(_root);
}

void Tree::print(Node* p) {
    if(!p)
        return;
    std::cout << '{';
    print(p->_ls);
    std::cout << ',' << p->_data << ',';
```

```
    print(p->_rs);
    std::cout << '}';
}

void Tree::insert(int data) {
    if(!_root) {
        _root = new Node(data);
        return;
    }
    Node* p = _root;
    while(true) {
        if(data <= p->_data) {
            if(p->_ls)
                p = p->_ls;
            else {
                p->_ls = new Node(data);
                break;
            }
        }
        else {
            if(p->_rs)
                p = p->_rs;
            else {
                p->_rs = new Node(data);
                break;
            }
        }
    }
}

int main() {
    int array[] = {5,3,7,1,4,6};
    Tree tr;
    for(size_t i=0; i<6; i++)
        tr.insert(array[i]);
    tr.print();
}
```

פלט:

```
{{{,1,},3,{,4,}},5,{{,6,},7,}}
```

ל- insert קוד מסורבל יחסית.
שימוש ברקורסיה היה מאפשר קוד פשוט יותר:

```
void Tree::insert(int data) {
  insert(data,_root);
}

void Tree::insert(int data, Node*& p) {
  if(!p) {
    p = new Node(data);
    return;
  }
  if(data <= p->_data)
    insert(data,p->_ls);
  else
    insert(data,p->_rs);
}
```

למרות שהקוד אולי אלגנטי יותר, הוא יעיל פחות. הקוד מממש רקורסיית זנב ואין שום צורך אמיתי בתהליך הריק של החזרה מהרקורסיה.

אותו קוד, ללא שימוש ב - refernces, יראה כך:

```
void Tree::insert(int data) {
  insert(data,&_root);
}

void Tree::insert(int data, Node** p) {
  if(!(*p)) {
    *p = new Node(data);
    return;
  }
  if(data <= (*p)->_data)
    insert(data,&((*p)->_ls));
  else
    insert(data,&((*p)->_rs));
}
```

קוד זה כבר ניתן להמיר בקלות יחסית ללולאה יעילה:

```
void Tree::insert(int data) {
  Node** p = &_root;
  while(*p) {
    if(data <= (*p)->_data)
      p = &((*p)->_ls);
    else
      p = &((*p)->_rs);
  }
  (*p) = new Node(data);
}
```

כדאי להשוות בין הגרסה הראשונה לאחרונה של insert.

תרגיל: נסו לחשוב על גרסה אלגנטית ל - clone של העץ.

**Struct ב - C++**

ב - C++ גם ל – struct ניתן להוסיף פונקציות. ההבדל **היחיד** בינו לבין class הוא שב- struct הרשאת ברירת המחדל היא public:

```
struct A {
        int _data; // a public data member
        void foo(); // a public member function
private:
        double _d; // a private data member
};
```

כפי שניתן להבין, באופן זה עדיין נשמרת התאימות ל - C.