Template
(חלק ב')

רשימת הארגומנטים של template

רשימת הארגומנטים של Template יכולה להכיל מספר טיפוסים:

file A.h:
```
template <class T1, class T2>
class A {
        T1 _d;
        T2 _c;
};
```

file main.cc:
```
int main() {
        A<double, int> a;
}
```

ברשימת הארגומנטים של template ניתן לרשום גם ערכים ולא רק טיפוסים. לדוגמה:

file A.h:
```
template<class T, int size>
class A {
        T _array[size];
}
```

file main.cc:
```
int main() {
        A<double,70> a1;
        A<double,100> a2;
}
```

ה – template הנ"ל יפרש פעמיים: פעם בשביל אובייקט המכיל מערך של 70 double, ופעם בשביל אובייקט המכיל מערך של 100 double. ה – sizeof של a1 יהיה: 70*sizeof(double) ו ה-sizeof של a2 יהיה 100*sizeof(double).

אם למשל נכתוב את השורה הבאה:

```
A<int,60> array[100];
```

יוקצו על המערך 100*60*sizeof(int) בתים. יצירת מערך של אובייקטים גורמת לקריאות של default ctor, לקריאות אלה לא ניתן להוסיף פרמטרים. אם היינו מבצעים ב – A הקצאה דינמית של מערך, אי אפשר היה לציין בשורת הקוד האחרונה, מה יהיה גודלו.

ארגומנט ערך יכל להיות מטיפוס ארגומנט קודם. לדוגמה:

```
template <typename T, T val>
class A
{
  public:
    void foo() {
       std::cout << val << std::endl;
    }
};

int main() {
   A<int,7> a;
```

```
    A<char,'h'> b;
    // A<char,7.8> c; // compilation error
    a.foo();  b.foo();
}
```

פלט:

```
7
h
```

ערכי default לארגומנטים של template

בדומה לערכי default שניתן לתת לפרמטרים של פונקציה, ניתן לתת ערכי default לארגומנטים של template :

file A.h:

```
 template <class T = double, int size = 50>
class A {
        T _array[size];
};
```

file main.cc:

```
int main() {
        A<> d;  // similar to: A<double,50>
        A<int> h; // similar to: A<int,50>
        A<float,70> z;
}
```

template specialization

לפעמים אנחנו כותבים קוד כללי ב – template שאיננו מתאים למקרים פרטיים אחדים. לדוגמה:

file A.h:

```
class A {
public:
        Type _data;
        A(Type data) : _data(data) {}
        bool isBegger(Type data);
};


template <class Type>
bool A<Type>::isBegger(Type data) {
        return data > _data;
}
```

file main.cc:

```
#include “A.h”
int main() {
        A<char*> a(“hi”);
        std::cout << a.isBigger(“bye”);
}
```

מה יקרה בהרצה ? לפי הקוד שכתוב ב – isBigger(), יושווה הערך של data לערך של _data.
הערכים של משתנים אלה הם כתובות של המקומות בזיכרון בהם מתחילות המחרוזות ”hi“ ו – ”bye“. הפונקציה isBigger() תשווה את הכתובות האלה ותבדוק איזה מהן גדולה יותר, כמספר. סביר שהכוונה שלנו בכתיבת הקוד הייתה אחרת, רצינו שתהיה השוואה לקסיקוגראפית בין

המחרוזות. אפשר להוסיף ל – template קוד ספציפי המטפל במקרה הפרטי שהטיפוס הוא char*:

file A.h (תוספת):
```
bool A<char*>::isBigger(char *data) {
        return strcmp(data, _data) > 0;
}
```
(יש להוסיף #include <string.h> בתחילת הקובץ)

file main.cc:
```
int main() {
        A<int> ai(5);
        A<char*> as("hi");
        ai.isBigger(7); // generic isBigger()
        as.isBigger("bye"); // specific isBigger()
}
```
הוספת כזו של פונקציה לטיפול במקרה פרטי נקראת 'specialization'.
כאשר הקומפיילר מגיע לקטע קוד בו מופעלת פונקציה, הוא צריך להחליט לאיזה פונקציה לקרא.
המנגנון שדואג לכך ניקרא 'matching' והאופן בו הוא פועל מאפשר את ה – specialization.

הערה: אם היינו יוצרים משתנה מסוג A<String> כאשר String היא מחלקה עם חפיפה של האופרטור <, לא היינו זקוקים ל – specialization.

global template function

כבר ראינו שניתן לכתוב פונקציה גלובאלית שהיא template (ב – << operator של הדפסת רשימה משורשרת).
פונקצית template, בדומה למחלקת template, נפרשת בפעם הראשונה שקוראים לה:

file main.cc:
```
template <class Type>
void foo(Type d) {…}

int main() {
        foo(5); // the function foo(int) will be expand and called
        foo(5.5); // the function foo(double) will be expand and called
        foo(2.1); // the function foo(double) will be called but will not be expand again
        int i = 5;
        foo(&i); // the function foo(int*) will be expand and called  (Type = 'int*')
}
```
מנגנון template מאפשר גם להגדיר מצביע לטיפוס כללי:

file main.cc:
```
template <class Type>
void foo(Type* p) {…}

int main() {
        int d;
        foo(&d); // foo(int *) will be expand and called (Type = int)
}
```
במקרה זה, בהפעלה של הפונקציה יש לשלוח כפרמטר מצביע לטיפוס מסוים. מנגנון ה – template, לפי הטיפוס של המצביע, פורש את הפונקציה כאשר Type מזוהה אם הטיפוס עליו מצביע הפרמטר.

מנגנון ההתאמה בין קריאה לפונקצית template לבין המימוש יכל להיות מורכב יותר. לדוגמה:

```
#include "LinkedList.h"
template <typename T>
void foo1(LinkedList<T*> list) {...}

template <typename T>
void foo2(LinkedList<LinkedList<T>* > list) {...}

int main() {
  LinkedList<int*> list;
  LinkedList<LinkedList<int>* > ll;
  foo1(list); // T = int
  foo1(ll); // T = LinkedList<int>
  foo2(ll);; // T = int
}
```

בפונקצית template , חייבים כל הטיפוסים המופיעים כארגומנטים ל – template להופיע ברשימת הפרמטרים של הפונקציה. הסיבה לכך היא שבפרישת קוד הפונקציה, השמת הטיפוסים לארגומנטים של ה – template, נעשית לפי הטיפוסים של הפרמטרים בקריאה לפונקציה:

file main.cc:

```
template <class T1, class T2>
void foo(T1 t) {
	T2 data;
}
int main() {
	foo(5); // T2 = ??
}
```

בדוגמה זו, כיוון שלא כללנו משתנה מטיפוס T2 ברשימת הפרמטרים שמקבלת foo() , הטיפוס שמקבל T2 איננו מוגדר. אפשר היה לכתוב את foo() כך:

```
template <class T1, class T2>
void foo(T2 t2, T1 t1) {
	T2 data;
}
```

אין דרישה כמובן, שרשימת הארגומנטים של ה – template תהיה באותו הסדר של רשימת הפרמטרים לפונקציה.

Templates כ – compile-time polymorphism

כזכור, polymorphism הוצג כמנגנון המאפשר כתיבת קוד כללי מבחינת הטיפוס עליו הוא עובד. קוד כללי כזה אמור להיות תקין עבור מספר טיפוסים, אך לפעול בצורה שונה בהתאם לטיפוס.
הדוגמה הבאה מראה כיצד מנגנון ה – templates מאפשר polymorphism שונה מזה שהכרנו, polymorphism הנקבע בזמן קומפילציה ולא בזמן הריצה:

```
#include <iostream>

template <typename T>
void foo(T t) {
	t.print(); // compile time polymorphism
}

class A {
```

```
public:
	void print() {
		std::cout << "I'm a happy instance of A" << std::endl;
	}
};

class B {
public:
	void print() {
		std::cout << "I'm a sad instance of B" << std::endl;
	}
};

int main() {
	A a;
	B b;
	foo(a);
	foo(b);
}
```

דוגמה לשילוב בין templates ל – run-time-polymorphism:

```
#include <iostream>

class DataStructure {
public:
	virtual ostream& toStream(ostream& os) const = 0;
};

ostream& operator << (ostream& os, const DataStructure& ds) {
	return ds.toStream(os);
}

template <typename T>
class List :public DataStructure {
public:
	virtual ostream& toStream(ostream& os) const {
		return os<<"I'm a list of " << T::getType() << std::endl;
	}
};

template <typename T>
class Tree : public DataStructure {
public:
	virtual ostream& toStream(ostream& os) const {
		return os<<"I'm a tree of " << T::getType() << std::endl;
	}
};

class A {
public:
	static char* getType() {return "A";}
```

```
};
class B {
public:
	static char* getType() {return "B";}
};

int main() {
	Tree<A> ta;
	Tree<B> tb;
	List<A> la;
	List<B> lb;
	std::cout << ta << tb << la << lb;
	return 0;
}
```

בכל פעם שנפרש קוד לטיפוס של רשימה של משהו חדש, נפרש קוד של מחלקה היורשת מ – DataStructure.

דוגמה נוספת לשילוב בין run-time ל – compile-time פולימורפיזם

נשתמש ברשימה המשורשרת שבנינו בשביל להחזיק מצביעים לטיפוס פולימורפי:

```
#include <iostream>
#include "LinkedList.h"

class A {
public:
  virtual void print(ostream& os) const =0;
};

ostream& operator << (ostream& os, const A* a) {
  a->print(os);
  return os;
}

class B : public A {
  virtual void print(ostream& os) const  {
    os << "B print";
  }
};

class C : public A {
  virtual void print(ostream& os) const  {
    os << "C print";
  }
};

int main() {
  B b1,b2;
  C c1,c2;
  LinkedList<A*> list;
  list.insert(&b1);
  list.insert(&c1);
  list.insert(&c2);
  list.insert(&b2);
  std::cout << list << std::endl;
```

```
}
```

פלט:

```
-> B print -> C print -> C print -> B print
```

שימו לב שהיחס בין הרשימה המשורשרת לאובייקטים שהיא מחזיקה הוא "ללא שיכפול וללא חיסול".

דוגמה לאלגוריתם גנרי על מבנה נתונים גנרי המקבל אובייקט השוואה גנרי

מיון לפי פונקציות השוואה שונות:

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

template <typename T>
class Less {
public:
  bool operator() (const T& t1, const T& t2) const {
    return t1 < t2;
  }
};

template <typename T>
class More {
public:
  bool operator() (const T& t1, const T& t2) const {
    return t1 > t2;
  }
};

class DigitLess {
  int digitSum(const int& t) const {
    return t-9*(t/10);  // good only for two digits number
  }
public:
  bool operator() (const int& t1, const int& t2) const {
    return digitSum(t1) < digitSum(t2);
  }
};

template <typename T>
void swap(T& t1, T& t2) {
  T tmp(t1);
  t1 = t2;
  t2 = tmp;
}

template <typename DS, typename CMP>
void sort(DS& ds, const CMP& cmp) {
  for(int i=0; i<ds.size(); i++)
    for(int j=i+1; j<ds.size(); j++)
      if(cmp(ds[j],ds[i]))
            swap(ds[j],ds[i]);
}
```

```
template <typename T, int SIZE>
class Array {
  T _arr[SIZE];
public:
  const T& operator[](int i) const { return _arr[i]; }
  T& operator[](int i) { return _arr[i]; }
  int size() {return SIZE; }
};

template <typename T, int SIZE>
ostream& operator << (ostream& os, const Array<T,SIZE>& arr) {
  for(int i=0; i<SIZE; i++)
    os << '|' << arr[i];
  return os << '|';
}

int main() {
  Array<int,20> ar1;
  Array<double,20> ar2;
  for(int i=0; i<20; i++)
    ar2[i] = ar1[i] = i;
  sort(ar1,Less<int>());
  sort(ar2,More<double>());
  std::cout << ar1 << " " << std::endl <<  ar2 << std::endl;
  sort(ar1,DigitLess());
  std::cout << ar1 << std::endl;
}
```

פלט:

```
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|
|19|18|17|16|15|14|13|12|11|10|9|8|7|6|5|4|3|2|1|0|
|0|1|10|2|11|3|12|4|13|5|14|6|15|7|16|8|17|9|18|19|
```