מבני נתונים גנריים ו – Templates
(חלק א')

כמעט כל מי שהתנסה בתכנות, נתקל בצורך לכתיבת קוד כללי, קוד שאיננו ספציפי לטיפוסים מסוימים אלא מתייחס לטיפוס כלשהו. עץ בינארי ממוין לדוגמה, הוא מבנה נתונים שימושי מאד, אנחנו עשויים להשתמש בעץ המכיל אנשים, נקודות, מספרים או רשימות משורשרות. בכל המקרים הלוגיקה והמבנה של מבנה הנתונים זהים, ההבדל הוא רק בטיפוסים שהוא מחזיק.

אפשר היה לכתוב את הקוד הדרוש לעץ של אנשים, לקרא למחלקה המתאימה PersonTree ואז לשכפל את הקוד לשנות את שם המחלקה ל – PointTree ולהחליף כל מופע של המחלקה Person במחלקה Point. פתרון זה מביא לתוכניות המכילות הרבה קוד משוכפל ואוסף גדול של שמות מחלקות (מסורבלים לרוב). שכפול קוד מקשה על התחזוקה ומזמין שגיאות.

ב – C++ ישנם פתרונות אלגנטיים יותר.

שימוש בירושה ופולימורפיזם

אפשרות אחת היא להשתמש במנגנוני הירושה והפולימופיזם. אם כל הטיפוסים המיועדים להכלה במבנה הנתונים שלנו, יורשים ממחלקה אחת, נניח Object אז אפשר לכתוב קוד של מבנה נתונים המחזיק מצביעים ל Object. במבנה נתונים זה ניתן יהיה להכניס כל אחד מהטיפוסים המיועדים, ניתן יהיה אפילו 'לערבב' והכניס לאותו מבנה מספר אובייקטים מטיפוסים שונים. הקוד הכללי שיהיה כתוב במבנה הנתונים, יתייחס לאובייקטים המוכלים בו כאל Object, והוא יוכל להפעיל רק את אותן פונקציות המשותפות לכל הטיפוסים המיועדים (כלומר מוצהרות ב – Object). זאת בדיוק הגישה בה בחרו מעצבי שפת התכנות Java, להתמודד עם הצורך בתכנות כללי.
לגישה זו מספר חסרונות כאשר מעוניינים במבני נתונים הומוגניים (כאלה שכל הנתונים שלהם הם מאותו הטיפוס):

- צורך ב downcast כאשר רוצים להתייחס ל – members שאינם במכנה המשותף.
- צורך לרשת את המחלקה של האובייקט המוכל במבנה הנתונים ממחלקת אב משותפת.
- אי יכולת ליצור מבנה נתונים של טיפוסים פרימיטיביים.
- תשלום בזמן ריצה כאשר מפעילים פונקציות וירטואליות.

שימוש ב – void *

אפשרות אחרת היא לכתוב מבנה נתונים המחזיק משתנים מסוג void*. void* הוא טיפוס שמשמעו : 'מצביע כללי' או 'מצביע סתמי'. כיוון שכל מצביע הוא למעשה **מספר** המייצג כתובת בזיכרון, כל מצביע (ולא משנה לאיזה טיפוס) דורש את אותה כמות זיכרון לשם אכסונו. גם המצביע הכללי מכיל למעשה מספר המייצג כתובת, אך אין בטיפוס שלו את האינפורמציה איזה טיפוס נימצא בכתובת הזאת.

דוגמה למבנה נתונים כללי הממומש בעזרת void * :

```
#include <iostream>
#include "Point.h" // this class represents a two dimensional point (an element of IR^2)

int main() {
	void* arr[10]; // a generic data structure

	for(int i = 0; i<10; i++)
		arr[i] = new Point(i,i+1); // implicit cast: Point* -> void* (like up-cast)
	for(i = 0; i<10; i++)
```

```
        std::cout << *((Point*)arr[i]) << " "; // explicit cast was needed (like down-cast)

    std::cout << std::endl;
}
```

במקרה זה יצרנו מערך המחזיק מצביעים כלליים, באותו אופן יכולנו ליצור כל מבנה נתונים אחר המחזיק מצביעים כלליים. יש לשים לב ל – cast שנעשה כאן בזמן השימוש בנתון השמור במבנה הנתונים. ההמרה הייתה מ – * void ל – *Point. המתכנת הוא זה שיודע על איזה סוג אובייקט מצביע arr[i] ולא הקומפיילר, ולכן עליו להמיר את המצביע מפורשות. העובדה שהאחריות על ההמרה נמצאת בידי המתכנת ושהקומפיילר לא מבצע בדיקת התאמת טיפוסים, יכולה לגרום בעיות. אם arr[i] מצביע לאובייקט מסוג אחר, אז ההמרה למצביע ל – Point, תגרום בזמן ריצה להתייחסות לא נכונה לאינפורמציה שבזיכרון או גרוע מזה – לדריסת זיכרון.

Templates

הכלי העיקרי אתו יוצרים ב – C++ מבני נתונים הומוגניים כלליים, הוא ה - template . המנגנון דומה מאד לפתרון הראשון שהוצע (של שיכפול הקוד ושינויו), אך הוא מבוצע בצורה אוטומטית ע"י הקומפיילר.

הקוד הבא הוא דוגמה ל – template פשוט המכיל משתנה כללי יחיד:

file: A.h

```
template <class Type>
class A {
private:
    Type _data;
public:
    A(Type data) : _data(data) {}
    Type getData() { return _data; }
};
```

המילה השמורה – template מכריזה על תחילתו של קוד המכיל טיפוס (לכן המילה 'class') כללי הנקרא פה בשם Type (Type הוא שם שרירותי, אפשר היה לבחור כל שם אחר).

כעת ניתן לכתוב main כזה:

file: main.cc

```
#include "Point.h"
#include "A.h"

int main() {
    A<double> ad(5.5);
    Point p;
    A<Point> ap(p);

}
```

A<double> הוא טיפוס חדש שמשמעו: 'A עם משתנה מסוג double'.

בזמן הקומפילציה קורה תהליך דומה לתהליך שתיארנו של שיכפול קוד. הצהרת ה – Template (כל הקוד שניכלל כתוצאה מהכללת הקובץ 'A.h') **איננה גורמת ליצירת שום קוד**, הקומפיילר רק משתמש באינפורמציה זו בהמשך. רק כאשר מגיע הקומפיילר לדרישה ליצירת משתנה מסוג A<double> נוצר הקוד הבא:

```
class A<double> {
private:
	double _data;
public:
	A(double data) : _data(data) {}
	double getData() { return _data; }
};
```

כל מופע של 'Type' הוחלף במופע של 'double'.
כאשר הקומפיילר מגיע לדרישה ליצר משתנה מסוג A<Point> נפרש הקוד הבא:

```
class A<Point> {
private:
	Point _data;
public:
	A(Point data) : _data(data) {}
	double getData() { return _data; }
};
```

אם תגיע דרישה נוספת ליצירת משתנה מסוג A<double>, לא יפרש שוב הקוד של הצהרת המחלקה, כיוון שקוד זה כבר נפרש בפעם הראשונה שנידרש משתנה מאותו הטיפוס. הטיפוס A<double> כבר הוצהר ולכן לא מצהירים עליו שוב.

למעשה, דרך המימוש של מנגנון ה – templates יכלה להיות שונה בין קומפיילרים שונים, פריסת הקוד איננה בהכרח פריסה של ה - text המקורי ואפשר לחשוב על אפשרות של פריסה של קוד שעבר קומפילציה מסוימת.

כפי שתואר מנגנון ה – templates כאן, ניתן ליראות שאפשר היה לממש אותו גם ע"י ה – preprocessor. כל העבודה הכרוכה במימוש המנגנון יכולה להיעשות ע"י שיכפול והחלפה של text. למעשה ממומש המנגנון ע"י חלקים מאוחרים יותר בקומפיילר, חלקים המתייחסים גם להתאמה בין טיפוסים ולא סתם מעבדים text.

ניתן לכתוב template גם בכתיב הבא:

```
template <typename T>
class A {…}
```

המילה השמורה typename מתפקדת כאן בדיוק בתפקיד שהיה למילה class בכתיב הקודם. משמעות שתי צורות הכתיבה זהה.

בדוגמה שהבאנו, הקוד של הפונקציות נכתב בתוך הצהרת ה – class ולכן היה – inline. אם נרצה שלא יהיה inline, נצטרך להוציא אותו מתוך ההצהרה אבל לא נוכל לשים אותו ביחידת קוד המתקמפלת בנפרד:

file A.h:

```
template <class Type>
class A {
private:
	Type _data;
public:
	A(Type data);
	Type getData();
```

```
};

template <class T>
A<T>::A(T data) : _data(data) {}

template <class T>
T A<T>::getData() { return _data; }
```

הסיבה שאנחנו כותבים "… A<T>::" היא שהפונקציה שייכת ל – class "A עם T ".

מדוע כללנו גם את מימושי הפונקציות בקובץ ה – 'h.' ולא כתבנו אותם ביחידת קומפילציה נפרדת ('cc.') כרגיל ?

כזכור, במודל הקומפילציה הרגיל שהכרנו, כל קובץ cc. עובר קומפילציה נפרדת, בלתי תלויה. כל קומפילציה נפרדת מייצרת object file ו ה- linker מחבר את כל ה – object files ויוצר קובץ הרצה.
נניח שהיינו ממשים את ה – template member function בקובץ cc. :

```
file A.cc:
template <class T>
A<T>::A(T data) : _data(data) {}

template <class T>
T A<T>::getData() { return _data; }
```

איך הייתה מתבצעת קומפילציה של קובץ כזה ?
כאשר רואים רק את ה – template ללא השימוש בה, אין אפשרות לדעת איזה קוד צריך לפרוש.
האם יש לפרוש את הפונקציה:

```
int A<int>::getData() { return _data; }
```

או אולי את:

```
SMatrix A<SMatrix>::getData() { return _data; }
```

לקומפיילר העובד על A.cc אין דרך לדעת.

הקומפיילר שיודע איזה פונקציה באמת יש לפרוש הוא הקומפיילר העובד על main.cc:

file: main.cc

```
#include "A.h"
int main() {
        A<double> a(7.8);
        a.getData();
}
```

ניתן לחשוב על מודל קומפילציה בו המידע איזה סוגים של פונקצית ה – template לפרוש יעבור בין הקומפילציות השונות. זה יהיה מודל מסובך יותר של קומפילציה שבו תהיה תלות בין הקומפילציות השונות. במודל שבו אנו עובדים, הפתרון הוא פשוט למקם את מימוש פונקצית ה – template ב header הנשתל בקובץ המשתמש. באופן זה, הקומפיילר העובד על main.cc יכל לפרוש את פונקציות ה – template הנחוצות, מכיוון שגם השימוש וגם המימוש של הפונקציה, גלוי לפניו. לפיכך ברור שה – linker צריך להתייחס לפונקציות ה- template כבעלות internal-linkage, כלומר להתעלם מהן.

העובדה שהפונקציה getData() נפרשת עבור כל סוג של A איננה צריכה להטעות אותנו לחשוב כי מדובר בפונקציה inline. הפונקציה הזאת מופעלת כפונקציה רגילה, והיא לא נפרשת עבור כל הפעלה.

רשימה משורשרת כדוגמה

כדוגמה לשימוש ב – template למבנה נתונים גנרי, נראה קוד בסיסי ל - template של רשימה משורשרת:

file ListNode.h:

```
#ifndef _LISTNODE_H
#define _LISTNODE_H

template <class Type>
class ListNode {
public:
	Type *_data;
	ListNode *_next; // " ListNode<Type> *_next " is also correct
	ListNode(Type *data = NULL, ListNode* next = NULL)
		: _data(data), _next(next) {}
};
#endif
```

file LinkedList.h:

```
#ifndef _LINKEDLIST_H
#define _LINKEDLIST_H

#include <iostream>
#include "ListNode.h"


template <class Type>
class LinkedList {
friend ostream& operator << <>(ostream& os, const LinkedList& list); // *see remark
private:
	ListNode<Type>* _head; // the '<Type>' is needed here
public:
	LinkedList() : _head(NULL) {}
	void insert(Type& data);
};


template <class Type> // *template function*
ostream& operator <<(ostream& os, const LinkedList<Type>& list) {
	ListNode<Type> *tmp = list._head;
	while(tmp != NULL) {
		os << " -> " << *(tmp->_data);
		tmp = tmp->_next;
	}
	return os;
```

```
}

template <class Type>
void LinkedList<Type>::insert(Type& data) {
	_head = new ListNode<Type>(&data,_head);
}
#endif
```

ואז ניתן לכתוב main כזה:

file main.cc:

```
#include <iostream>
#include "LinkedList.h"
#include "Point.h"
int main() {
	LinkedList<int> iList;
	LinkedList<Point> pList;
	int i1=2, i2 = 3, i3=-33;
	iList.insert(i1);
	iList.insert(i2);
	iList.insert(i3);
	Point p1(2,1),p2(5,7);
	pList.insert(p1);
	pList.insert(p2);

	std::cout << iList << std::endl << pList << std::endl;

	LinkedList<LinkedList <Point> > ll; // the space '<Point> >' is essential
	ll.insert(pList);
	ll.insert(pList);
	std::cout << ll << std::endl;
}
```

* הערה: תוספת ה - <> בהצהרת החברות של פונקצית ה – template, נדרשת ע"י קומפיילרים מסוימים בשביל לציין שמדובר באמת בפונקצית template ולא בפונקציה רגילה.

כדאי לנסות לחשוב מה בעצם קורה בהצהרה של רשימת הרשימות, וכן איך פועלת פונקצית ההדפסה שלה.

בבחינה מדוקדקת של הקוד הנ"ל מתגלות כמה בעיות עקרוניות: לרשימה המשורשרת אין copy ctor, אין dtor וכל הכנסה של ערך חדש למבנה הנתונים איננו משוכפל אלא פשוט מוצבע. למעשה רשימת הרשימות בקוד הנ"ל הוא פיקציה, זאת בעצם אותה הרשימה המוצבעת ע"י שני אברים של רשימת הרשימות.

הנה הגרסה המתוקנת:

file ListNode.h:

```
#ifndef _LISTNODE_H
#define _LISTNODE_H

template <typename Type>
class ListNode {
public:
	Type* _data;
	ListNode* _next;
	ListNode(const Type* data = NULL, ListNode* next = NULL)
```

```
            :  _next(next) {
            if(data)
                  _data = new Type(*data); // assuming copy ctor of
Type
            else
                  _data=NULL;
      }
      ~ListNode() {
            if(_data)
                  delete _data;
      }
};
#endif
```

file LinkedList.h:

```
#ifndef _LINKEDLIST_H
#define _LINKEDLIST_H

#include <iostream>
#include "ListNode.h"


template <typename Type>
class LinkedList {
friend ostream& operator << <>(ostream& os, const LinkedList& list);
private:
      ListNode<Type>* _head; // the '<Type>' is needed here
public:
      LinkedList() : _head(NULL) {}
      LinkedList(const LinkedList& other);
      void insert(const Type& data);
      ~LinkedList();
};

template <typename T>
LinkedList<T>::LinkedList(const LinkedList& other) {
// better to implement it using ListNode<T>**
      if(other._head == NULL) {
            _head=NULL;
            return;
      }
ListNode<T>* tmpT = _head = new ListNode<T>(other._head->_data);
      ListNode<T>* tmpO = other._head;
      while(tmpO->_next) {
            tmpT->_next = new ListNode<T>(tmpO->_next->_data);
            tmpO = tmpO->_next;
            tmpT = tmpT->_next;
      }
}

template <typename Type> // *template function*
ostream& operator <<(ostream& os, const LinkedList<Type>& list) {
      ListNode<Type> *tmp = list._head;
      while(tmp != NULL) {
            os << " -> " << *(tmp->_data);
            tmp = tmp->_next;
      }
      return os;
```

```
}

template <typename Type>
void LinkedList<Type>::insert(const Type& data) {
	_head = new ListNode<Type>(&data,_head);
}

template <typename T>
LinkedList<T>::~LinkedList() {
	ListNode<T>* t = _head;
	while(_head) {
		_head = _head->_next;
		delete t;
		t = _head;
	}
}
#endif
```

וקוד המדגים את ההבדל (נסו גם על הגרסה הקודמת):

file main.cc:

```
#include <iostream>
#include "LinkedList.h"
#include "Point.h"
int main() {
	LinkedList<int> iList;
	LinkedList<Point> pList;
	int i1=2, i2 = 3, i3=-33;
	iList.insert(i1);
	iList.insert(i2);
	iList.insert(i3);
	Point p1(2,1),p2(5,7);
	pList.insert(p1);
	pList.insert(p2);

	std::cout << iList << std::endl << pList << std::endl;

	LinkedList<LinkedList <Point> > ll;
	ll.insert(pList);
	p1.setX(7777);
	pList.insert(p1);
	ll.insert(pList);
	std::cout << ll << std::endl;
}
```