

Generics (Γενικότητα)

Μπορούμε να κατασκευάσουμε ΑΤΔ που επιτρέπει χρήση με διαφορετικούς Τύπους Δεδομένων στο ίδιο πρόγραμμα;

Π.χ. Μια στοίβα με `char*` και μια με `int`

Goals of this Lecture

- Help you learn about:
 - Generic modules
 - Data structures that can store multiple types of data
 - Functions that can work on multiple types of data
 - How to create generic modules in C
 - Which wasn't designed with generic modules in mind!
- Why?
 - Reusing existing code is easier/cheaper than writing new code; reuse reduces software costs
 - Generic modules are more reusable than non-generic ones
 - A power programmer knows how to create generic modules
 - A power programmer knows how to use existing generic modules to create large programs

Generic Data Structures Example

- Stack

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void Stack_free(Stack_T s);
void Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void Stack_pop(Stack_T s);
int Stack_isEmpty(Stack_T s);
```

- Items are strings (type char*)

Generic Data Structures Example

- Stack operations (push, pop, top, etc.) make sense for items ***other than*** strings too
- So Stack module could (and maybe should) be generic
- Problem: How to make Stack module generic?
 - How to define Stack module such that a Stack object can store items ***of any type***?

Generic Data Structures via typedef

- Solution 1: Let clients define item type (G2)

```
/* client.c */  
  
struct Item {  
    char *str; /* Or whatever is appropriate */  
};
```

```
...  
Stack_T s;  
struct Item item;  
  
item.str = "hello";  
s = Stack_new();  
Stack_push(s, item);  
...
```

```
/* stack.h */  
  
typedef struct Item *Item_T;  
typedef struct Stack *Stack_T;  
  
Stack_T Stack_new(void);  
void Stack_free(Stack_T s);  
void Stack_push(Stack_T s, Item_T item);  
Item_T Stack_top(Stack_T s);  
void Stack_pop(Stack_T s);  
int Stack_isEmpty(Stack_T s);
```

Generic Data Structures via typedef

- Problem: Awkward for client to define structure type and create structures of that type
- Problem: Client (or some other module in same program) might already use “Item_T” for some other purpose!
- Problem: Client might need two Stack objects holding different types of data!!!
- We need another approach...

Generic Data Structures via void*

- Solution 2: The generic pointer (void*) (G3)

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void      Stack_free(Stack_T s);
void      Stack_push(Stack_T s, const void *item);
void      *Stack_top(Stack_T s);
void      Stack_pop(Stack_T s);
int        Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Can assign a pointer of any type to a void pointer

```
/* client.c */  
  
...  
Stack_T s;  
s = Stack_new();  
Stack_push(s, "hello");  
...
```

OK to match an actual parameter of type char* with a formal parameter of type void*

```
/* stack.h */  
  
typedef struct Stack *Stack_T;  
  
Stack_T Stack_new(void);  
void Stack_free(Stack_T s);  
void Stack_push(Stack_T s, const void *item);  
void *Stack_top(Stack_T s);  
void Stack_pop(Stack_T s);  
int Stack_isEmpty(Stack_T s);
```


Generic Data Structures via void*

- Can assign a void pointer to a pointer of any type

```
/* client.c */  
  
char *str;  
...  
Stack_T s;  
s = Stack_new();  
Stack_push(s, "hello");  
...  
str = Stack_top(s);
```

OK to assign
a void* return value
to a char*

```
/* stack.h */  
  
typedef struct Stack *Stack_T;  
  
Stack_T Stack_new(void);  
void Stack_free(Stack_T s);  
void Stack_push(Stack_T s, const void *item);  
void *Stack_top(Stack_T s);  
void Stack_pop(Stack_T s);  
int Stack_isEmpty(Stack_T s);
```

Generic Data Structures via void*

- Problem: Client must know what type of data a void pointer is pointing to

```
/* client.c */
int *i;

...
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
...
i = Stack_top(s);
```

Client pushes a string

Client considers retrieved value to be a pointer to an int! Legal!!! Trouble!!!

- Solution: None
 - Void pointers subvert the compiler's type checking
 - Programmer's responsibility to avoid type mismatches

Generic Data Structures via void*

- Problem: Stack items must be pointers
 - E.g. Stack items cannot be of primitive types (int, double, etc.)

```
/* client.c */  
...  
int i = 5;  
...  
Stack_T s;  
s = Stack_new();  
...  
Stack_push(s, 5);  
...  
Stack_push(s, &i);
```

Not OK to match an actual parameter of type int with a formal parameter of type void*

OK, but awkward

- Solution: none
 - In C, there is no mechanism to create truly generic data structures
 - (In C++: Use [template classes](#) and [template functions](#))
 - (In Java: Use [generic classes](#))

Generic Algorithms Example

- Suppose we wish to add another function to the Stack module

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void      Stack_free(Stack_T s);
void      Stack_push(Stack_T s, const void *item);
void      *Stack_top(Stack_T s);
void      Stack_pop(Stack_T s);
int        Stack_isEmpty(Stack_T s);
int        Stack_areEqual(Stack_T s1, Stack_T s2);
```

Returns 1 (TRUE) iff s1 and s2 are equal,
that is, contain equal items in the same order

Generic Algorithm Attempt 1

- Attempt 1

```
/* stack.c */  
  
...  
  
int Stack_areEqual(Stack_T s1, Stack_T s2) {  
    return s1 == s2;  
}
```

Returns 0 (FALSE)
Incorrect!

```
/* client.c */  
  
char str1[] = "hi";  
char str2[] = "hi";  
Stack_T s1 = Stack_new();  
Stack_T s2 = Stack_new();  
Stack_push(s1, str1);  
Stack_push(s2, str2);  
  
if (Stack_areEqual(s1, s2)) {  
    ...  
}
```

- Checks if s1 and s2 are *identical*, not *equal*
 - Compares pointers, not items
- That's not what we want

Addresses vs. Values

- Suppose two locations in memory have the same value

```
int i=5;  
int j=5;
```

i	5
j	5

- The addresses of the variables are *not* the same
 - That is “(&i == &j)” is FALSE
- Need to compare the values themselves
 - That is “(i == j)” is TRUE
- Unfortunately, comparison operation is type specific
 - The “==” works for integers and floating-point numbers
 - But not for strings and more complex data structures

Generic Algorithm Attempt 2

- Attempt 2

```
/* stack.c */
...
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;
    while ((p1 != NULL) && (p2 != NULL)) {
        if (p1 != p2)
            return 0;
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL))
        return 0;
    return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
    ...
}
```

Returns 0 (FALSE)
Incorrect!

- Checks if *nodes* are *identical*
 - Compares pointers, not items
- That is *still* not what we want

Generic Algorithm Attempt 3

- Attempt 3

```
/* stack.c */
...
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;
    while ((p1 != NULL) && (p2 != NULL)) {
        if (p1->item != p2->item)
            return 0;
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL))
        return 0;
    return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
    ...
}
```

Returns 0 (FALSE)
Incorrect!

- Checks if *items* are *identical*
 - Compares pointers to items, not items themselves
- That is *still* not what we want

Generic Algorithm Attempt 4

- Attempt 4

```
/* stack.c */
...
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;
    while ((p1 != NULL) && (p2 != NULL)) {
        if (strcmp(p1->item, p2->item) != 0)
            return 0;
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL))
        return 0;
    return 1;
}
```

```
/* client.c */

char str1[] = "hi";
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1,s2)) {
    ...
}
```

- But `strcmp()` works only if items are strings! We need `Stack_areEqual()` to be *generic*
- How to compare values when we don't know their type?

Returns 1 (TRUE)
Correct!

Generic Algorithm via Function Pointer

- Approach 5

```
/* stack.h */

typedef struct Stack *Stack_T;

Stack_T Stack_new(void);
void Stack_free(Stack_T s);
void Stack_push(Stack_T s, const void *item);
void *Stack_top(Stack_T s);
void Stack_pop(Stack_T s);
int Stack_isEmpty(Stack_T s);
int Stack_areEqual(Stack_T s1, Stack_T s2,
    int (*cmp)(const void *item1, const void *item2));
```

- Add parameter to **Stack_areEqual()**
 - Pointer to a compare function, which...
 - Accepts two const void * parameters, and...
 - Returns an int
- Allows client to supply the function that **Stack_areEqual()** should call to compare items

Generic Algorithm via Function Pointer

- Approach 5 (cont.)

```
/* stack.c */
...
int Stack_areEqual(Stack_T s1, Stack_T s2,
    int (*cmp)(const void *item1, const void *item2)) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;
    while ((p1 != NULL) && (p2 != NULL)) {
        if ((*cmp)(p1->item, p2->item) != 0)
            return 0;
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL))
        return 0;
    return 1;
}
```

- Definition of **Stack_areEqual()** uses the function pointer to call the client-supplied compare function
- **Stack_areEqual()** “calls back” into client code

Generic Algorithm via Function Pointer

- Approach 5 (cont.)

```
/* client.c */
```

```
int strCompare(const void *item1, const void *item2) {  
    char *str1 = item1;  
    char *str2 = item2;  
    return strcmp(str1, str2);  
}
```

```
...  
char str2[] = "hi";  
Stack_T s1 = Stack_new();  
Stack_T s2 = Stack_new();  
Stack_push(s1, str1);  
Stack_push(s2, str2);
```

```
if (Stack_areEqual(s1, s2, strCompare)) {  
    ...  
}
```

Returns 1 (TRUE)
Correct!

Client passes address
of **strCompare()** to
Stack_areEqual()

- Client defines “callback function”, and passes pointer to it to **Stack_areEqual()**
- Callback function must match **Stack_areEqual()** parameter exactly

Generic Algorithm via Function Pointer

- Alternative: Client defines more “natural” callback function, and casts it to match **Stack_areEqual()** parameter
- Approach 5 (cont.)

```
/* client.c */

int strCompare(const char *str1, const char *str2) {
    return strcmp(str1, str2);
}

...
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
    (int (*)(const void*, const void*))strCompare)) {
    ...
}
```

Cast
operator

Generic Algorithm via Function Pointer

- Approach 5 (cont.)

```
/* client.c */
...
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);

if (Stack_areEqual(s1, s2,
    (int (*)(const void*, const void*))strcmp)) {
    ...
}
```

Cast
operator

- Alternative (for string comparisons only): Simply use **strcmp()** with cast!

Summary

- Generic data structures
 - Via item typedef
 - Safe, but not realistic
 - Via the generic pointer (void*)
 - Limiting: items must be pointers
 - Dangerous: subverts compiler type checking
 - The best we can do in C
 - (See C++ template classes and Java generics for other approaches)
- Generic algorithms
 - Via function pointers and callback functions

Υλοποίηση Ενοτήτων (Modules) στην C και Γενικότητα (Genericity)

		Modularity				
		No Modularity (0)	Απόκρυψη Πράξεων (MA) (1)	Απόκρυψη Δεδομένων και Πράξεων (ΠΑ)(2,3)	+	-
Genericity	ΤΣ καθορισμένος στον κώδικα, χωρίς αφαίρεση	ΟΧΙ (Demo Mod 0)	ΟΧΙ λόγω G (Demo Mod 1)	ΟΧΙ λόγω G (Demo Mod 3+2)	έλεγχος μεταγλώττισης	1 μόνο ΤΣ, Αλλαγή ΤΣ απαιτεί Αλλαγή υλοποίησης ΑΤΔ, απαιτεί recompilation
	ΤΣ #include σε .h (και αντίστοιχο .c)	ΟΧΙ λόγω Modularity	ΝΑΙ με Προσοχή όπως ουρά 1ης άσκησης	ΝΑΙ (απλό με πολλά πλεονεκτήματα) ουρά 2ης άσκησης	Αλλαγή ΤΣ δεν απαιτεί αλλαγή υλοποίησης ΑΤΔ, έλεγχος μεταγλώττισης	1 μόνο ΤΣ, Αλλαγή ΤΑ απαιτεί recompilation
	ΤΣ (Πλήρης απόκρυψη και ΤΣ *T)(παράμετροι συναρτήσεων και data του ΑΤΔ είναι *T)	ΟΧΙ λόγω Modularity	Χωρίς ιδιαίτερο όφελος	ΝΑΙ, απαιτεί καλή χρήση δεικτών Mod 3 Gen 1-T (SetValue απλουστεύεται)(compare)	Αλλαγή ΤΣ δεν απαιτεί αλλαγή υλοποίησης ΑΤΔ, δεν απαιτεί recompilation, έλεγχος μεταγλώττισης	1 μόνο ΤΣ
	ΤΣ (Πλήρης απόκρυψη ΤΣ *T)(παράμετροι συναρτήσεων και data του ΑΤΔ είναι void*)	ΟΧΙ λόγω Modularity	Χωρίς ιδιαίτερο όφελος	Χρήσιμη για βιβλιοθήκες, απαιτεί καλούς ελέγχους. Mod 3 Gen n-Type (compare)	πολλά-ΤΣ, σταθερή υλοποίηση ΑΤΔ, δεν απαιτεί recompilation	χωρίς έλεγχο μεταγλώττισης
+			Καλή προσέγγιση, προσοχή στην άμεση χρήση στον Τύπο.	Η καλύτερη απόκρυψη	1 ΤΣ μπορεί να καλύπτει πολλούς τύπους με union	
-		Να αποφεύγεται εντελώς, καμία απόκρυψη	Κίνδυνοι άμεσης πρόσβασης στον ΤΑ, αλλοίωση απόκρυψης	Διαχειριζόμαστε δείκτες και όχι structs		