

Generic Data Structures

Σέργιος - Ανέστης Κεφαλίδης
Κωνσταντίνος Νικολέτος
Κώστας Πλας

One issue remains...

- We have studied and implemented a bunch of different data structures.
- All of our implementations have the same problem, the data type is hardcoded.
 - We use *typedef* to be able to quickly change the hardcoded type, but it is still hardcoded.
- In the real world we want the ability to create instances of a data structure that contain different types, so we must get rid of the hardcoded data type.
 - For example, a List that holds Integers and a List that holds Floats in the same program.
- This is the last issue that we are going to tackle, so let's have some fun ;-)

Let's examine the problem with an example

- Consider that a University needs to save information about students in the form a record. Let's assume that this record is called Student.
- Record Student:
 - id(String)
 - gradeAverage(Double)
 - fullName(String)
 - yearOfEntry(Int)
- To maintain an efficient searching time for every student the university uses a hash table, to store the records based on their id(of type String).
- We implement a hash table that stores Students and hashes Strings
- So far everything is great!

Let's examine the problem with an example

Hashtable



Records:

("1240529", 8.3, "Panos Cosmatos", 2018)

("1240572", 9.1, "Gaspar Noe", 2016)

Stored Records(Students) .
Hashing based on id(String) .
. .

Let's examine the problem with an example

- Let's assume that the University wants to utilize a similar system for professors.
- Record Professor:
 - id(Int)
 - fullName(String)
 - course(String)
- We want to store the Professors on a different hashtable and hash their id which is now an Integer. However, our hashtable cannot store records of type Professor.
- What can we do???

Let's examine the problem with an example

Student Hashtable



Student Records:

("1240529", 8.3, "Panos Cosmatos", 2018)

("1240572", 9.1, "Gaspar Noe", 2016)

.

.

Stored Records(Students)
Hashing based on id(String)

.

.

.

Prof. Records:

(0, "John Smith", "Data Structures")

(1, "Jane Doe", "Discrete Mathematics")

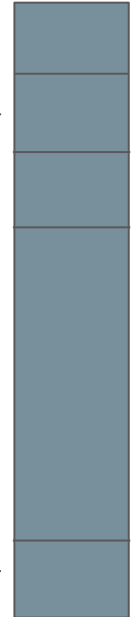
.

.

.

Stored Records(Professors)
Hashing based on id(int)

Prof. Hashtable



Proposed solutions

- There are two schools of thought for dealing with this problem.
 - a. Use pointers to handle data. This approach essentially bypasses the type system.
 - b. Create multiple implementations of the data structure, one for each type that we want to use the data structure with (**code generation approach**).

Outline


1. Using ***void**** to create generic data structures.
 - Presentation.
 - Live Coding: Implementing a generic Linked List using ***void****.
2. Using code generation methods to create generic data structures.
 - ***C Macros***
 - i. Presentation.
 - ii. Live Coding: Implementing a generic Linked List using ***Macros***.
 - ***C with Templates*** (a mix of C and C++)
 - i. Presentation.
 - ii. Live Coding: Implementing a generic Linked List using ***Templates***.
3. Comparison and discussion.

Using void* to create generic data structures

- The basic idea is to use *void ** to create a data structure that can store every type (custom or not), instead of creating different data structures to deal with different data types.
- Let's assume we have implemented a linked list that uses integers as keys. It is easy to convert the basic struct definition using *void **

```
typedef struct list{  
    int data;  
    struct list *next;  
}List;
```

```
typedef struct list{  
    void *data;  
    struct list *next;  
}List;
```

A diagram consisting of two rectangular boxes with a light gray background and a thin black border. The left box contains C code defining a struct 'list' with an 'int data' field and a 'struct list *next' pointer. The right box contains the same code but with 'void *data' instead of 'int data'. A horizontal arrow points from the right side of the left box to the left side of the right box, indicating a transformation or conversion.

Using `void*` to create generic data structures

- Inserting elements is straight forward, using `void *`. Remember that we need to use casts!
- How can we search or delete elements, since we do not know what type of data was inserted in the list?

Using void* to create generic data structures

- Remember that we can create pointer types for functions in c!
 - E.g. `typedef void (*Visitor)(Node *)`;
- Using custom functions types we can provide data comparison and deletion functions to the data structure, to handle different data types.

```
typedef int (*Compare)(void *, void *);
```

```
int CompareInteger(void *a, void *b) {  
    int *ia = (int*)a;  
    int *ib = (int*)b;  
  
    if(*ia == *ib) return 0;  
    else if(*ia > *ib) return 1;  
    else return -1;  
}  
  
int CompareString(void *a, void *b) {  
    char *sa = (char *)a;  
    char *sb = (char *)b;  
    return strcmp(sa, sb);  
}
```

Using void* to create generic data structures

- The aforementioned functions can either be passed into the definition to the structure:

```
typedef struct list{  
    void *data;  
    struct list *next;  
    Compare compare;  
}List;
```

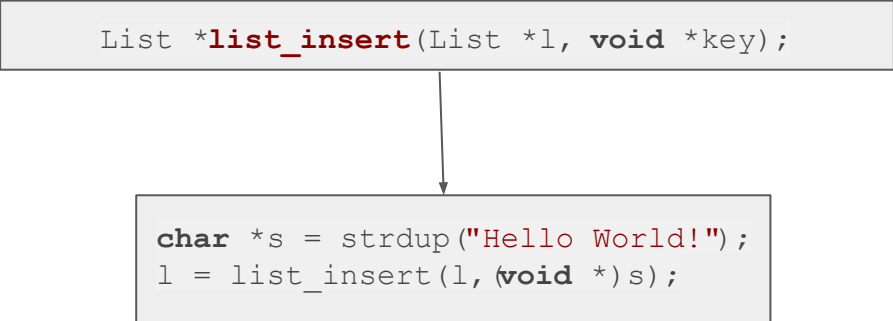
- Or in function definition. In this case the comparison function must be passed in every call of the function:

```
int list_search(List *l, void *key, Compare compare);
```

Using void* to create generic data structures

- Combining the above we can create a generic data structure with general ease.
- For insertion: data insertion is up to the struct. All we need to do is cast the element when calling the function:

```
List *list_insert(List *l, void *key);
```



The diagram consists of two rectangular boxes. The top box contains the function signature `List *list_insert(List *l, void *key);`. A vertical arrow points from the bottom center of this box to the top center of the bottom box. The bottom box contains two lines of code: `char *s = strdup("Hello World!");` and `l = list_insert(l, (void *)s);`. The `(void *)` cast in the second line is positioned directly below the `void *` in the function signature above.

```
char *s = strdup("Hello World!");  
l = list_insert(l, (void *)s);
```

Using void* to create generic data structures

- For search ,deletion, etc: the main operations are up to the struct again. We now use comparison and deletion functions. (Function in function definition)

```
int list_search(List *l, void *key, Compare compare){
    ...
    if(compare(l->data, key) == 0){
        return 1;
    }
    ...
}
```

Using void* to create generic data structures

- For search ,deletion, etc: the main operations are up to the struct again. We now use comparison and deletion functions. (Function in struct definition)

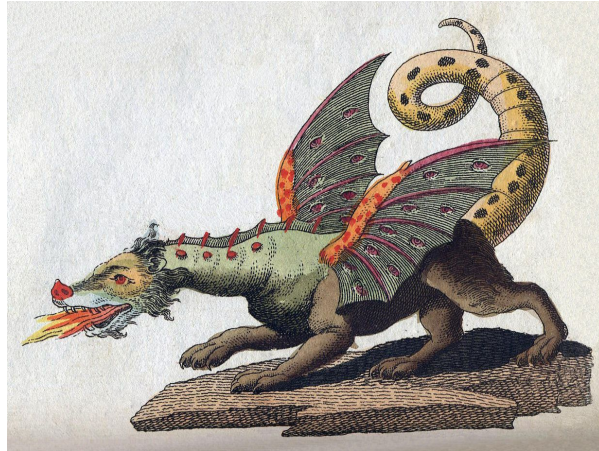
```
int list_search(List *l, void *key){  
    ...  
    if(l->compare(l->data, key) == 0){  
        return 1;  
    }  
    ...  
}
```

Using `void*` to create generic data structures

- Using custom function types combined with `void *`, we can create flexible data structures able to handle different data types, without rewriting vast amounts of code.
- Pros:
 - Only need to write comparison, deletion etc. functions to support new data types.
 - Easily reusable data structures, that do not need multiple definitions for multiple data types.
- Cons:
 - Ownership of data is passed to the structure. The structure is responsible for whatever happens to these memory locations (Memory deallocation etc.)!!!
 - Some implementations may not be able to deal with data stored in the stack.
 - We need a lot of casts to deal with conversion of data types to `void *`.
 - Compiler will miss a lot of type errors, that could otherwise be avoided.
 - We Must be very careful when inserting data into the structure! No one can stop us from inserting data of different types!!!

Playing with fire: Macros

hic sunt dracones



Using macros to create multiple implementations...

- We can create multiple implementations manually, by copying and pasting structs and functions:

```
typedef struct list_struct_int {
    int data;
    struct list_struct_int* next;
} List_int;

typedef struct list_struct_double {
    double data;
    struct list_struct_double* next;
} List_double;
```

- This is a lot of manual effort and is unmaintainable. **Why?**
- Automate this process by automating code generation. **Which C facility allows us to modify our source files before compilation?**

Using macros to create multiple implementations...

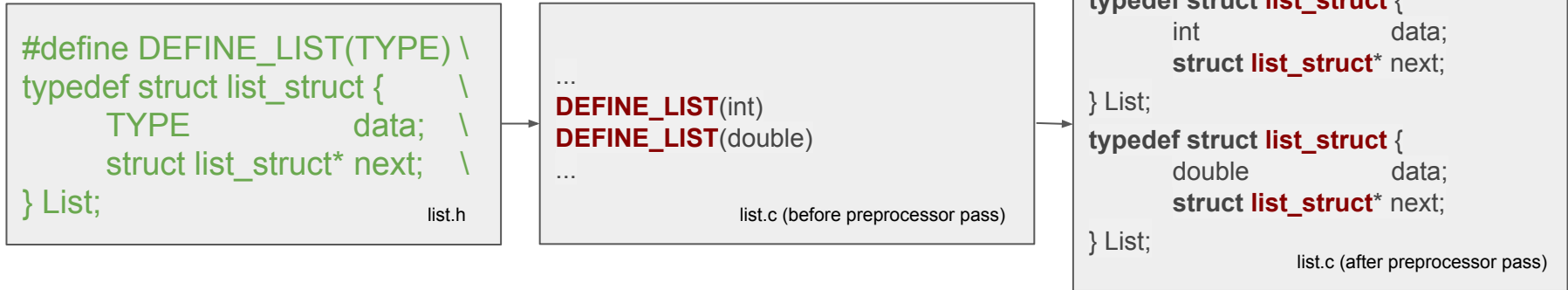
- Macros give us the ability to modify our source files before they are passed to the compiler.
 - The C preprocessor expands all macros before the code is compiled.
- We can define macros that generate code.



- What happens if we try to define multiple lists this way?

Using macros to create multiple implementations...

- What happens if we try to define multiple lists this way?
- The preprocessor modifies our source file.
- The compiler complains! **Why?**

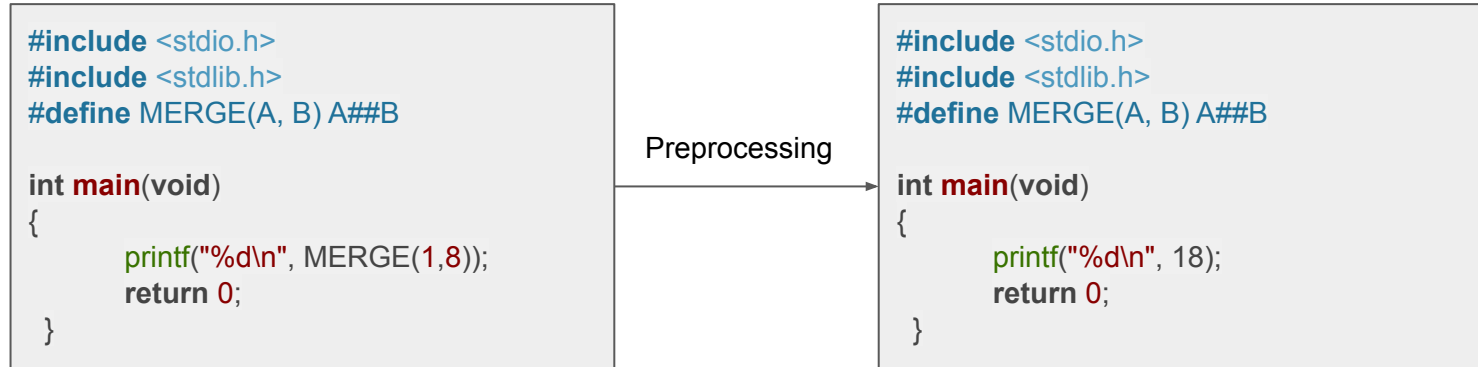


Compiler output:

error: redefinition of 'struct list_struct'
error: conflicting types for 'List'

Using macros to create multiple implementations...

- We can use macros to easily generate multiple implementations of a data structure, but we need a way to differentiate them.
- We can use the token-pasting operator (**##**) to merge two tokens into one while expanding macros. This allows us to assign different names to each of our implementations.



Using macros to create multiple implementations...

- Let's try again!
- The Preprocessor expands our macros and the compiler has no reason to complain!

```
#define DEFINE_LIST(TYPE)    \
typedef struct list_struct_##TYPE { \
    TYPE                    data; \
    struct list_struct_##TYPE* next; \
} List_##TYPE;
```

list.h

```
...  
DEFINE_LIST(int)  
DEFINE_LIST(double)  
...
```

list.c (before preprocessor pass)

```
typedef struct list_struct_int {  
    int                    data;  
    struct list_struct_int* next;  
} List_int;  
typedef struct list_struct_double {  
    double                    data;  
    struct list_struct_double* next;  
} List_double;
```

list.c (after preprocessor pass)

- There is an important limitation! Can you find it?
 - Hint: try using **DEFINE_LIST** with different types.

Using macros to create multiple implementations...

- Let's try again!
- The Preprocessor expands our macros and the compiler has no reason to complain!

```
#define DEFINE_LIST(TYPE)    \
typedef struct list_struct_##TYPE { \
    TYPE                      data; \
    struct list_struct_##TYPE* next; \
} List_##TYPE;
```

list.h

```
...  
DEFINE_LIST(int)  
DEFINE_LIST(double)  
...
```

list.c (before preprocessor pass)

```
typedef struct list_struct_int {  
    int                      data;  
    struct list_struct_int* next;  
} List_int;  
typedef struct list_struct_double {  
    double                    data;  
    struct list_struct_double* next;  
} List_double;
```

list.c (after preprocessor pass)

- There is an important limitation!
 - **Structs and pointers are not supported!** Luckily you can use **typedef!**

Using macros to create multiple implementations...

- Now that we are able to generate the structs required by our Data Structure we will also generate its functions in similar fashion.
- Let's see a more complete example...

Using macros to create multiple implementations...

```
#define DEFINE_LIST(TYPE) \
typedef struct list_##TYPE { \
    TYPE          data; \
    struct list_##TYPE* next; \
} List_##TYPE; \
DEFINE_LIST_PREPEND(TYPE)

#define DEFINE_LIST_PREPEND(TYPE) \
List_##TYPE* list_##TYPE##_prepend(List_##TYPE* list, TYPE data) \
{ \
    List_##TYPE* newList = malloc(sizeof(List_##TYPE)); \
    newList->data = data; \
    newList->next = list; \
    return newList; \
}
```

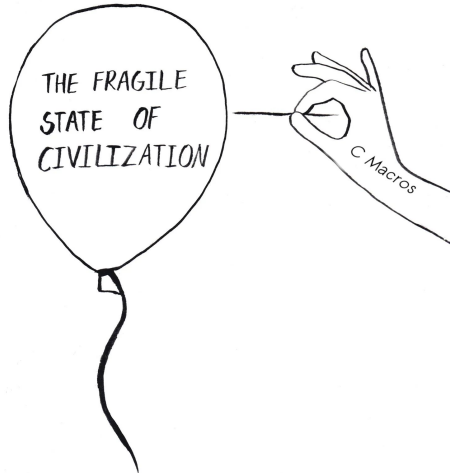
Do note that the **DEFINE_LIST** macro creates both the required node struct and calls the macros that generate the functions.

This is done for convenience. You can generate the functions manually.

If you choose to generate the functions manually, you can bypass unused functions, reducing the size of the codebase.

Using macros to create multiple implementations...

- Pros:
 - Type safe
- Cons:
 - Increased code size.
 - Confusing to write. It is easy to make mistakes or forget syntactical edge cases.
 - Some limitations do exist.



A taste of C++: Templates

What is better: to be born good or to overcome your evil nature through great effort?



What is C++?

- General purpose, multi-paradigm programming language.
- Originally developed as an **extension of C** by Bjarne Stroustrup.
 - Has seen significant expansion and growth.
 - **No longer a strict superset of C.**
- Very big and complex language.
- You will learn more about it in the “Object-oriented Programming” course, next semester.

With great power comes great responsibility

What is C++?

- A lot of C code is valid C++ code. This means that it can be compiled with a C++ compiler.
- This is not always the case!
 - C++ has additional keywords like *'delete'* and *'new'*.
 - C++ does not support Variable Length Arrays and Array Initializers.
 - There are other facilities to handle these usecases which we will not present.
 - C++ does not allow implicit pointer casts.
 - `int* a = malloc(sizeof(int));` → `int* a = (int*) malloc(sizeof(int));`
- Makefiles must be modified to use:
 - `CC=g++`
 - `CXXFLAGS` instead of `CFLAGS`

Using templates to create multiple implementations...

- To create generic types C++ provides the *Template* feature. We want to use this feature in our C code to create generic types.
 - This combination of C and C++ is known as 'C with Templates'.
 - We will not use any other C++ feature.
- Templates work by generating code at compile time.
 - The end result of template generation is similar to what we saw in the previous section.
 - Templates are more powerful and better suited for generic types, compared to the simple text processing done by macros.
 - Templates are easier to debug because compiler errors refer to the template instead of the expanded version of the code.

Using templates to create multiple implementations...

- Every generic struct or function must be preceded by one of:
 - `template<typename T>` where *T* is the generic type that will be used (like *TYPE* in our macros).
 - `template<>` when we want to manually create an instance of our template (we will see the need for that in the live coding segment).

```
#define DEFINE_LIST(TYPE) \
typedef struct list_struct_##TYPE { \
    TYPE data; \
    struct list_struct_##TYPE* next; \
} List_##TYPE;
```

list.h

```
template<typename T>
struct List {
    T data;
    List<T> *next;
};
```

list.h

Note: notice the subtle difference in the definition of structs in C++.

Using templates to create multiple implementations...

- Every generic struct or function must be preceded by one of:
 - `template<typename T>` where *T* is the generic type that will be used (like *TYPE* in our macros).
 - `template<>` when we want to manually create an instance of our template (we will see the need for that in the live coding segment).

```
#define DEFINE_LIST_PREPEND(TYPE) \
List_##TYPE* list_##TYPE##_prepend(List_##TYPE* list, TYPE data) \
{\
    List_##TYPE* newList = malloc(sizeof(List_##TYPE)); \
    newList->data = data; \
    newList->next = list; \
    return newList; \
}
```

list.h

```
template<typename T>
List<T>* list_prepend(List<T> *list, T data)
{
    List<T>* newList = (List<T>*)malloc(sizeof(List<T>*));
    newList->data = data;
    newList->next = list;
    return newList;
}
```

list.h

Using templates to create multiple implementations...

- There is no need to manually create the implementations for the different types, the compiler takes care of that.
- The type of the data structure is denoted inside angled brackets <...>.

```
#include "list.h"

int main(void)
{
    List<int> myIntList = list_prepend(NULL, 1);
    ...
}
```

main.c

Using templates to create multiple implementations...

- Pros:
 - Type safe.
 - The compiler is able to help you write them.
 - Very powerful feature.
 - With great power comes...
- Cons:
 - Increased code size. Can lead to an exponential increase of compilation time.
 - As a C++ feature, templates require a C++ compiler. C++ is not fully compatible with C which means that you are sacrificing some of C's functionality (e.g., variable length arrays, array initializers) to use templates.
 - Very powerful feature.
 - ... great responsibility.

Comparison



Comparison

- Using ***void****
 - Pros:
 - A single implementation works for all types.
 - Cons:
 - Bypasses the type system. The burden of handling types and casts falls on the programmer.
- Using ***Macros***
 - Pros:
 - Type safe
 - Cons:
 - Increased code size.
 - Confusing to write. It is easy to make mistakes or forget syntactical edge cases.
- Using ***Templates***
 - Pros:
 - Type safe.
 - Significantly easier to write compared to macros.
 - Cons:
 - Increased code size.
 - As a C++ feature, templates require a C++ compiler. C++ is not fully compatible with C which means that you are sacrificing some of C's functionality (e.g., variable length arrays, array initializers) to use templates.

Let's discuss :-)

Σας ευχαριστούμε!

Καλή επιτυχία στην εξεταστική!

