C/UNIX Seminar - January 2022

Day 2: Genericity and Data Structures

Assistants Salle Machine 2024

- Void Pointers
  - Genericity
  - Example: Linked list
- Functional Programming
- Enumerations
  - Principle
  - Values of enumerations

- Unions
  - Principle
  - Tagged unions
- Macros
  - What is a macro?
  - Genericity
  - Multi lines
  - Macros operators
  - Token-pasting
  - Warning

Void Pointers

Genericity

Genericity in programming is the idea of writing a single piece of code that can handle many different situations and types.
This is important when you want to avoid writing similar code multiple times; that is to say always, or at least we hope so.

Genericity

One of the ways we can do this in C is using void pointers.
Void pointers are declared like any other pointer:

```
void *ptr = NULL;
```

The main difference being that void pointers do not retain any information about the type of the data they point to nor about its size.

Genericity

While this means that you can have a void pointer point to anything, it also means that once you have made it point to some data, you have to find a way to remember what type that data is.
Another issue is that you cannot dereference a void pointer as the compiler has no idea how many bytes it should dereference and in what way.

Genericity

Putting those concerns aside, say you knew what type something is.
Then, you could dereference it by simply casting the pointer beforehand.

Genericity

```c
int integer = 42;
char *str = "Hello, world!";

void *ptr = &integer;

ptr = str;
// ptr cannot be dereferenced like this
// printf("%c\n", *ptr);

char *get = ptr;
printf("%c\n", *get);
```

Example: Linked list

```c
struct list
{
    struct list *next;
    void *data;
};

void list_free(struct list *list);
struct list *list_add(struct list *list,
                      void *elem,
                      size_t elem_size);
struct list *list_remove(struct list *list,
                         size_t index);
```

```c
struct list *list_add(struct list *list,
                      void *elem,
                      size_t elem_size)
{
    ...
    // We cannot dereference a void pointer so
    // instead of doing janky casts lef and
    // right we memcpy(3) the desired amount
    // of bytes
    new->data = calloc(1, elem_size);
    memcpy(new->data, elem, elem_size);
    ...
}
```

The full code will be available later on.

Functional Programming

Functional Programming

While C is a decidedly imperative language, there are ways to imitate the logic of functional languages and their higher-order functions to make code even more generic.
Let's take as an example the bubble sort algorithm, which I'm sure you all know by know.

Functional Programming

If we wanted to implement this algorithm for different data types and structures, with a strictly imperative and procedural mindset we would have to write the whole function multiple times with slight changes; this happens to be a cardinal sin for any self-respecting programmer.
So, how should one go about doing that?

Functional Programming

The answer to this question lies in the magic of function pointers.
Function pointers are just like the other, "normal" pointers you've seen so far.
The difference lies in the fact that the function pointed to by said pointer can be called using it, which makes passing a function to another one possible.

Functional Programming

What this means for our bubble sort example is that you could build the bubble sort function in such a way that it accepts both a set of data of a certain type *and* a function that can order said data. Couple this with void pointers and the possibilities are limitless... well, almost, but pretty great either way.

Enumerations

Principle

Enumerations are a user defined data type which can be used to name arbitrary values in order to make code easy to read and maintain.

```
enum direction {
    UP,
    DOWN,
    LEFT,
    RIGHT
};
```

By convention, enumerations are written in uppercase.

Values of enumerations

Even if they are named, enumerations remain values. You can choose to change these values, or you can keep the default values. By default the first value will be 0 and the next will be plus one.

```
enum fruits
{
    BANANA,       // = 0
    APPLE,        // = 1
    PEACH,        // = 2
    MANGO = 10    // = 10
    STRAWBERRY,   // = 11
};
```

Unions

Principle

An union is data stucture useful in order to produce generic code. It can take a huge amount of fields, but can contain only one at a time. The size of an union is the size of its biggest element.

To declare an union, you have to do the same as structures:

```
union integer
{
    char small;
    int medium;
    long big;
};
```

Tagged unions

In order to know which field is assigned to the union we have to tag it. You can use the enumerations in order to tag you union.

```
union int_string
{
    int int_t;
    char *str_t;
};

enum name_height
{
    NAME,
    HEIGHT
};
```

Tagged unions

```c
struct patient
{
    enum name_height tag;
    union int_string data;
};

int main(void)
{
    struct patient patient_1;
    patient_1.tag = NAME;
    patient_1.data.str_t = "John";
}
```

Macros

What is a macro?

A macro is a piece of code that will be replaced by its value at compile-time.

The main interests are **genericity** and **getting rid of magic values**

The syntax of the macros is the following:

```
#define MACRO_NAME([[Parameter, ]* Parameter]) replacement-text
```

Genericity

You can call a macro with any type: the same macro can be used for multiple types.

Genericity: example

```
#define MULT(A, B) ((A) * (B))
int int_mult = MULT(10, 2);
float float_mult = MULT(4.5, 3.0);
```

As you can see, the macro MULT is working for both int and float.

Multi lines

In order to write a macro on multiple lines, you just have to finish your line with \.

Here is an example:

```
#define POW(X) \
int res = 1 << (int)X; \
printf("%d\n", res)
```

Stringify

There are some operators about the macros that you need to know.
The stringify operator converts a macro argument into a string. If a parameter appears with a prefixed #, the preprocessor places the argument between ".

```
#define PRINT(A) puts(#A)
```

Token-pasting

The binary operator ## joins its left and its right operands together into a single token. Whitespace characters that appear before and after ## are removed along with the operator itself.

```
#define JOIN(A, B) A ## B
JOIN(123, 456) //expanded as 123456
```

Warning

The token-pasting operator doesn't work with strings, and the result is not a string.

```
JOIN("hello", "world") // doesn't work
#define STRFY(A) #A
char *s = STRFY(JOIN(hello, world)) //works
```

asm 2024