Real time audio programming in C

28.05.2020 – Object oriented programming in C

Some object-oriented programming paradigms can be imitated in C. The Pure Data SDK uses this object-oriented programming approach in C for:

- (A kind of) classes and class member functions in C.
- (A kind of) overloading.
- Inheritance

C has no classes and therefore no class member functions.
But structs are basically classes and have struct members (variables).
They can be accessed with the . (dot/point) or the -> (arrow) operator, depending on whether a pointer or a value is dereferenced:

```c
#include <stdio.h>

typedef struct aStruct {
    int memberOne;
    int memberTwo;
} aStruct;

int main()
{
    aStruct myStruct;
    myStruct.memberOne = 1;
    myStruct.memberTwo = 2;

    printf("Member one: %d\n", myStruct.memberOne);
    printf("Member two: %d\n", (&myStruct)->memberTwo);

    return 0;
}

/*

 Output:
 Member one: 1
 Member two: 2

 */
```

In object-oriented programming it is possible to define member functions. In C that would mean defining a function inside a struct. **That is <u>not</u> possible.** Instead we have to imitate, what compilers in object-oriented programming languages do for us and pass our struct as a pointer argument to our class function:

```c
#include <stdio.h>

typedef struct aStruct {
    int memberOne;
    int memberTwo;
} aStruct;

void aStruct_aMemberFunctionThatPrints(aStruct *x)
{
    printf("Member one: %d\n", x->memberOne);
    printf("Member two: %d\n", x->memberTwo);
}

int main()
{
    aStruct myStruct;
    myStruct.memberOne = 1;
    myStruct.memberTwo = 2;
    aStruct_aMemberFunctionThatPrints(&myStruct);

    return 0;
}

/*
    Output:
    Member one: 1
    Member two: 2
*/
```

Task: Write the corresponding getter and setter member functions for the two struct members. The setter functions should set the member variables and the getter functions should get the values and therefore return an integer value.

Overloading is a feature that allows for implementing the same function with the **same function name** for different data types. The compiler will check at compile time (at least in C++) which function to call. In C this already done for basic operators and simple data types. For example, it is possible to call the + function for float and integer values:

```c
#include <stdio.h>

int main() {

    float a, b;
    float resultFloat;
    int c, d;
    int resultInt;

    a = 1.1; b = 2.2;
    c = 1; d = 2;

    resultFloat = a+b;
    resultInt = c+d;

    printf("%f\n", resultFloat);
    printf("%d\n", resultInt);

    return 0;
}
```

However, overloading for your own functions is not supported in C. There are different approaches for achieving this kind of behavior, all have advantages and disadvantages, depending on what you are trying to do.

```c
#include <stdio.h>

int addArray_int(int *myArray, int size) {
    int sum = 0;
    for (int i=0; i<size;i++)
          sum+=myArray[i];

    return sum;
}

float addArray_float(float *myArray, int size) {
    float sum = 0;
    for (int i=0; i<size;i++)
          sum+=myArray[i];

    return sum;
}

int main() {

    int arraySize = 2;
    float floatArray[arraySize];
    int intArray[arraySize];

    floatArray[0] = 1.1;
    floatArray[1] = 2.2;
    intArray[0] = 1;
    intArray[1] = 2;

    printf("%f\n", addArray_float(floatArray, arraySize));
    printf("%d\n", addArray_int(intArray, arraySize));

    return 0;
}

/*
    Output:
    3.300000
    3
*/
```

Task: Write a function, which takes the type as a first argument (a char pointer) and then chooses between the *int* and *float* version.

Hint: The array itself must be passed as a void pointer and you also need to pass a void pointer for the result.

A possible Solution:

```
void addArray(char *type, void *myArray, int size, void *result)
{
    if(!strcmp(type, "float")) {
        for (int i=0; i<size;i++) {
            *((float *)result) += ((float *)myArray)[i];
        }
    }

    if(!strcmp (type, "int")) {
        for (int i=0; i<size;i++) {
            *((int *)result) += ((int *)myArray)[i];
        }
    }
}


int main()
{
    // Size of arrays
    const unsigned int size = 3;

    // Init arrays
    int i_arr[size] = {1, 2, 3};
    float f_arr[size] = {1., 2., 3.};

    // Result var
    int i_result;
    float f_result;

    addArray("int", (void*)(i_arr), size, &i_result);
    addArray("float", (void*)(f_arr), size, &f_result);

    printf("Sum of int array: %d\n", i_result);
    printf("Sum of float array: %f\n", f_result);

    return 0;
}
```

```
/*
    Output:
    Sum of int array: 6
    Sum of float array: 6.000000
 */
```

Pure Data uses a different approach to achieve similar behaviour. It defines a new data type:

t_atom

The t_atom struct contains a member for a float, for a symbol and for several other things. It also contains a member for its type and corresponding setter functions (implemented as macros), for example

SETFLOAT(atom, f)

and getter functions

*t_float atom_getfloat(t_atom *a)*

When *SETFLOAT* is called, the atom's type is set to float and the value f stored in the float member of the atom struct. When atom_getfloat() is called, the atom's type is checked and if it is indeed a float it returns the float member of the atom. Among other things this technique allows for working with lists of mixed data (usually strings and floats).

Task: Pure Data works internally with floats only, but let's implement our own simple atom type which contains three members: the type (int or float as an enum value), and a member for a float and a member for an integer. Write the corresponding getter and setter

functions and a function, that adds all elements of an atom array together.

A possible solution:

```c
#include <stdio.h>

enum simpleAtomTypes {UNDEFINED, INTEGER, FLOAT};

typedef struct simpleAtom
{
    int type;
    int intVal;
    float floatVal;
} simpleAtom;

void simpleAtom_setFloat(simpleAtom *a, float val)
{
    a->type = FLOAT;
    a->floatVal = val;
}

void simpleAtom_setInt(simpleAtom *a, int val)
{
    a->type = INTEGER;
    a->intVal = val;
}

float simpleAtom_getFloat(simpleAtom *a)
{
    if(a->type == FLOAT)
        return a->floatVal;
    else
    {
        printf("Atom does not contain a float value!\n");
        return 0;
    }
}

int simpleAtom_getInt(simpleAtom *a)
{
    if(a->type == INTEGER)
        return a->intVal;
    else
    {
        printf("Atom does not contain a int value!\n");
        return 0;
    }
}
```

```c
float simpleAtom_addAll(simpleAtom *a, int size)
{
    float sum = 0;
    for(int i=0; i<size;i++)
    {
        if(a[i].type == FLOAT)
            sum+=a[i].floatVal;
        if(a[i].type == INTEGER)
            sum+=a[i].intVal;
    }

    return sum;
}

int main() {

    int size = 2;
    float result = 0;

    simpleAtom myAtom[size];
    simpleAtom_setFloat(&myAtom[0], 5.5);
    simpleAtom_setInt(&myAtom[1], 67);
    result = simpleAtom_addAll(myAtom, size);
    printf("%f\n", result);

    return 0;
}

/*
    Output:
    72.500000
*/
```