Real time audio programming in C

28.05.2020

Object oriented programming in C Part II – Inheritance

Pointer to functions, Lookup table

Every object in Pure Data must contain the *t_object x_obj* as its first member. This allows for a very basic inheritance mechanism, because in memory structs are exactly represented as programmed. Therefore functions, that only work with the first object member *t_object x_obj* will work with all Pure Data objects.

Task: Write a base class and two derived classes (structs). The base class should only contain an integer member type (UNDEFINED, INTEGER or FLOAT as an enum). The first derived class (name it for example derivedClassFloat) contains also the type and additionally a float value, the second class contains the type and an integer value. Write a function that works on the base class and returns the type. In the main function create for example a derivedClassFloat value and then get its type with the written baseClass_getType() function. You have to cast the derived class to the base class.

A possible solution:

```c
#include <stdio.h>


/**
 * @brief Enum that is used in an atom struct type.
 * @enum Holds the defined data types of an atom struct
type.
 */
enum simpleAtomTypes {UNDEFINED, INTEGER, FLOAT};


/**
 * @brief Base class.
 * @typedef A base class atom object type.
 */
typedef struct baseClass
{
    int type;       /**< @c int variable, that defines
the atom type. */

} baseClass;


/**
 * @brief Derived class.
 * @typedef A derived class atom object type that owns
the base class member.
 */
typedef struct derivedClassFloat
{
    int type;       /**< @c int variable, that defines
the atom type. */
    float floatVal; /**< @c float variable of the atom
type. */

} derivedClassFloat;
```

```c
/**
 * @brief Derived class.
 * @typedef A derived class atom object type that owns the
base class member.
 */
typedef struct derivedClassInt
{
    int type;       /**< @c int variable, that defines the
atom type. */
    int intVal;     /**< @c int variable of the atom type. */

} derivedClassInt;

/**
 * @brief Function that gets the atom type.
 * @param x Pointer to an atom type object.
 * @return int member of the @c enum @c simpleAtomTypes.
 */
int baseClass_getType(baseClass *x) {
    return x->type;
}

int main() {
    // Declare a derived class object
    derivedClassFloat someThing;
    // Set type and var member
    someThing.type = FLOAT;
    someThing.floatVal = 5.5;

    // Get the type of the derived class object using base
    // class casting
    int type = baseClass_getType((baseClass *) &someThing);

    // Print the type
    printf("The simple atom type of someThing is ");
    switch (type) {
        case 0:
            printf("UNDEFINED.\n");

            break;
        case 1:
            printf("INTEGER.\n");

        case 2:
            printf("FLOAT.\n");

        default:
            break;
    }
    return 0;
}
```

```
/*
    Output:
    The simple atom type of someThing is FLOAT.
 */
```

So far, our written C structs/classes neither have a constructor nor a destructor, we created them on the stack without dynamic memory allocation. So either we do it every time manually using malloc() and free() or we write a constructor/destructor function, that returns a pointer to our object. We can also use the constructor to initialize our members with default values.

```
/**
 * @brief A simple class example.
 * @typedef @c struct type as workaround for the missing
 * class feature in C
 */
typedef struct myClass
{
    int type;
    float floatVal;

} myClass;


/**
 * @brief Constructor function for myClass.
 * @return Pointer to the initialized myClass object.
 */
myClass *myClass_new()
{
    myClass *x = (myClass *)malloc(sizeof(myClass));
    x->type = 0;
    x->floatVal = 0;
    return x;
}
```

Task: Write the corresponding destructor which calls the free() function on our class.

Solution:

```
/**
 * @brief Destructor function for myClass.
 * @param x Pointer to myClass object to be freed.
 */
void myClass_free(myClass *x)
{
    if(x != NULL)
        free(x);
}
```

Object life cycle

```
int main() {
    // Initialize a new object of myClass
    myClass *classInstance = myClass_new();

    // Use the object
    printf("%d\n", classInstance->type);

    // Free the object
    myClass_free(classInstance);

    return 0;
}
```

To provide an SDK for external objects, Pure Data implements a tricky class register mechanism. Within the object setup function (which is called on Pure Data's startup), class_new() is called. Arguments are the object's name, its new function, its free function, its size, its type and the type of arguments it expects.
class_new returns a pointer which is stored in a global pointer.

```
static t_class *helloworld_class;

void helloworld_setup(void) {
     helloworld_class = class_new(gensym("helloworld"),
                        (t_newmethod)helloworld_new,
                        0,
                        sizeof(t_helloworld),
                        CLASS_DEFAULT,
                        0,
                        0);

     class_addbang(helloworld_class, helloworld_bang);
}
```

What we haven't seen so far is passing a function name as an argument to another function. In this case the helloworld_new() function.

The syntax for this dealing with function pointers is shown in the following examples.

```c
#include <stdio.h>

/**
 * @brief Function that prints a value.
 * @param a Value to print.
 */
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main()
{
    // void pointer to the function fun.
    void (*fun_ptr)(int) = &fun;

    // Call the function fun via passing a value to
    // the function pointer
    (*fun_ptr)(10);

    return 0;
}
```

A function pointer is not compatible with a regular pointer. You are not allowed to cast a function pointer to a simple void pointer. Casting a function pointer to a function pointer of another kind is allowed, however, correct behaviour is only guaranteed if it is casted back to its original type.

```c
/**
 * @brief Function that sums two @c int values.
 * @param a First summand.
 * @param b Second summand.
 * @return int Sum of the both @c int parameter values.
 */
int sum(int a, int b) {
    return a + b;
}


/**
 * @brief Function that multiplies two @c int values.
 * @param a First factor.
 * @param b Second factor.
 * @return int Product of the both @c int parameter
 * values.
 */
int mul(int a, int b) {
    return a * b;
}


/**
 * @brief Function that takes a pointer to another math
 * function that will process the two other passed
 * parameter values.
 * @param OpType Pointer to a function the takes two @c
 * int as parameter and returns an @c int as result.
 * @param a First @c int paramter to process.
 * @param b Second @c int paramter to process.
 * @return int Result of the math operation.
 */
int mathOp(int (*OpType)(int, int), int a, int b) {
    return OpType(a, b);
}
```

```c
int main()
{
    printf("%i,%i\n", mathOp(sum, 10, 12), mathOp(mul, 10,
    2));

    return 0;
}


/*  Output:
 *  22,20
 */
```

```c
#include <stdio.h>
#include <stdlib.h>


/** @typedef @c pointer to a @c void function. */
typedef void* (*new)(void);

/** @typedef @c pointer to a function, that takes one
    @c int parameter. */
typedef void* (*methodInt)(int);

/** @typedef @c pointer to a function, that takes two
    @c int parameter. */
typedef void* (*methodIntInt)(int, int);


/** @typedef struct with one @c int element only. */
typedef struct _oneInt
{
    int val1;   /**< Struct @c int variable. */
} t_oneInt;


/**
 *  @brief Constructor function that takes a @c int value
 *  and returns a pointer to a new @c t_oneInt struct.
 *  @param v1 @c int value of the new struct element
 *  @return Pointer to the new struct.
 */
void *oneInt_new(int v1)
{
    t_oneInt *x = (t_oneInt *)malloc(sizeof(t_oneInt));
    x->val1 = v1;
    return x;
}
```

```c
/** @typedef struct with two @c int elements. */
typedef struct _twoInt {
    int val1;   /**< Struct @c int variable. */
    int val2;   /**< Struct @c int variable. */
} t_twoInt;


/**
 *  @brief Constructor function that takes two @c int
 *  values and returns a pointer to a new @c t_twoInt
 *  struct.
 *  @param v1 @c int value of the new struct element.
 *  @param v2 @c int value of the new struct element.
 *  @return Pointer to the new struct.
 */
void *twoInt_new(int v1, int v2) {
    t_twoInt *x = (t_twoInt *)malloc(sizeof(t_twoInt));
    x->val1 = v1;
    x->val2 = v2;
    return x;
}


/**
 *  @brief Constructor function that takes a pointer to a
 *  constructor function and an argument list/count
 *  @param m The objects method pointer.
 *  @param args Arguments list used at function call.
 *  @param argc Arguments count of argument list used at
 *  function call.
 *  @return Pointer to the new struct object depending on
 *  arguments count @c argc.
 *  @todo Error handling if function returns @c NULL.
 */
void *intObject_new(new m, int *args, int argc) {
    if(argc == 1)
    {
        void* (*new_ptr)(int) = (methodInt)m;
        return (new_ptr)(args[0]);
    }

    else if(argc == 2)
    {
        void* (*new_ptr)(int, int) = (methodIntInt)m;
        return (new_ptr)(args[0], args[1]);
    }

    else return NULL;

}
```

```c
int main()
{
    // Prepare data to pass to to the initializer
    // function.
    int oneElementArray[1];
    int twoElementsArray[2];

    oneElementArray[0] = 1;
    twoElementsArray[0] = 2;
    twoElementsArray[1] = 4;

    // Call init functions of t_oneInt and t_twoInt
    // objects.

    // Back casted new method pointer
    // Pointer to the argumuents list resp. argv
    // Number of arguments resp. argc
    t_oneInt *a = intObject_new((new)oneInt_new,
                                oneElementArray,
                                1);

    // Back casted new method pointer
    // Pointer to the argumuents list resp. argv
    // Number of arguments resp. argc
    t_twoInt *b = intObject_new((new)twoInt_new,
                                twoElementsArray,
                                2);

    // Print results.
    printf("%d\n", a->val1);
    printf("%d %d\n", b->val1, b->val2);


    return 0;
}
```

Task: Instead of calling *intObject_new()* and passing the constructor directly, implement a global lookup table (with fixed length) where objects can be registered with their name, their new_method() and their number of arguments. The corresponding *object_new()* function then needs to arguments: the name of the object and the arguments for the object. On calling object_new() the function searches in the lookup table for the right object by comparing the char arrays and calls the corresponding contructor.

A few hints in C and pseudo code (in upper case):

```
...
...

typedef struct RegisteredIntObject
{
    char name[25];
    method newMethod;
    int argc;
} registeredIntObject;

GLOBAL ARRAY registeredObject[10];
GLOBAL INT currentIndex = 0;

REGISTEROBJECT(OBJECT_NAME, NEW_METHOD, ARGC)
{
     INSERTOBJECT_AT_CURRENT_INDEX_INTO(registeredObjects);
     INCREASE_CURRENT_INDEX;
}

NEWOBJECT(OBJECT_NAME, ARGV)
{
     SEARCH_IN_REGISTEREDOBJECTS_FOR_OBJECT();
     CALL_THE_CORRESPONDING_NEW_METHOD_WITH_ARGV();
}

...
...
```

A solution:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXOBJECTNAMESIZE 24
#define MAXNUMBEROFOBJECTS 10

typedef void* (*method)(void);
typedef void* (*methodInt)(int);
typedef void* (*methodIntInt)(int, int);

typedef struct RegisteredIntObject
{
    char name[MAXOBJECTNAMESIZE];
    method newMethod;
    int argc;
} registeredIntObject;

registeredIntObject objectLookupTable[MAXNUMBEROFOBJECTS];
int currentIndex = 0;

typedef struct oneInt
{
    int val1;
} oneInt;

void *oneInt_new(int val1)
{
    oneInt *x = (oneInt *)malloc(sizeof(oneInt));
    x->val1 = val1;
    return x;
}

typedef struct twoInt
{
    int val1;
    int val2;
} twoInt;

void *twoInt_new(int val1, int val2)
{
    twoInt *x = (twoInt *)malloc(sizeof(twoInt));
    x->val1 = val1;
    x->val2 = val2;
    return x;
}
```

```
void registerObject(char *name, method m, int argc)
{
    strcpy(objectLookupTable[currentIndex].name, name);
    objectLookupTable[currentIndex].newMethod = (method)m;
    objectLookupTable[currentIndex].argc = argc;
    currentIndex++;
}

void *newObject(char *name, int *argv)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;

        i++; // This loop is not save, why?
    }

    if(objectLookupTable[i].argc == 1) // why check for argc?
    {
        void* (*new_ptr)(int) =
              (methodInt)objectLookupTable[i].newMethod;
        return (new_ptr)(argv[0]);
    }

    if(objectLookupTable[i].argc == 2) // why check for argc?
    {
        void* (*new_ptr)(int, int) =
              (methodIntInt)objectLookupTable[i].newMethod;
        return (new_ptr)(argv[0], argv[1]);
    }
}

int main()
{
    int anArray[1];
    int anotherArray[2];

    anArray[0] = 1;
    anotherArray[0] = 2;
    anotherArray[1] = 4;

    registerObject("oneint", (method)oneInt_new, 1);
    registerObject("twoint", (method)twoInt_new, 2);

    oneInt *a = newObject("oneint", anArray);
    twoInt *b = newObject("twoint", anotherArray);
    printf("%d\n", a->val1);
    printf("%d %d", b->val1, b->val2);
    return 0;
}
```

Task: Add a member for a bang method to the registeredObject struct and a function for adding the bang method. Our bang method does not return anything and gets its instance as void pointer.

```
typedef void (*bang)(void *);
```

Also implement the bang methods for *oneInt* & *twoInt* which should simply print their member values to the command line and attach them to the objects. Instead of calling printf call the bang methods from inside the main function.

A solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXOBJECTNAMESIZE 24
#define MAXNUMBEROFOBJECTS 10

typedef void* (*method)();
typedef void* (*methodInt)(int);
typedef void* (*methodIntInt)(int, int);
typedef void (*bang)(void *);

typedef struct registeredIntObject
{
    char name[MAXOBJECTNAMESIZE];
    method newMethod;
    bang bangMethod;
    int argc;
} registeredIntObject;

registeredIntObject objectLookupTable[MAXNUMBEROFOBJECTS];
int currentIndex = 0;

typedef struct oneInt
{
    int val1;
} oneInt;

void *oneInt_new(int val1)
{
    oneInt *x = (oneInt *)malloc(sizeof(oneInt));
    x->val1 = val1;
    return x;
}

void *oneInt_bang(void *x)
{
    printf("%d\n", ((oneInt *)x)->val1);
}
```

```c
typedef struct twoInt
{
    int val1;
    int val2;
} twoInt;

void *twoInt_new(int val1, int val2)
{
    twoInt *x = (twoInt *)malloc(sizeof(twoInt));
    x->val1 = val1;
    x->val2 = val2;
    return x;
}

void *twoInt_bang(void *x)
{
    printf("%d\n", ((twoInt *)x)->val1);
    printf("%d", ((twoInt *)x)->val2);
}

void registerObject(char *name, method m, int argc)
{
    strcpy(objectLookupTable[currentIndex].name, name);
    objectLookupTable[currentIndex].newMethod = (method)m;
    objectLookupTable[currentIndex].argc = argc;
    currentIndex++;
}

void addBang(char *name, bang b)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;

        i++;  // Again, this loop is not save..
    }

    objectLookupTable[i].bangMethod = b;
}

void object_bang(char *name, void *x)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;
        i++; // Again, this loop is not save..
    }

    (objectLookupTable[i].bangMethod)(x);
}
```

```
void *newObject(char *name, int *argv)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;
        i++;
    }

    if(objectLookupTable[i].argc == 1)
    {
        void* (*new_ptr)(int) =
               (methodInt)objectLookupTable[i].newMethod;
        return (new_ptr)(argv[0]);
    }

    if(objectLookupTable[i].argc == 2)
    {
        void* (*new_ptr)(int, int) =
               (methodIntInt)objectLookupTable[i].newMethod;;
        return (new_ptr)(argv[0], argv[1]);
    }
}

int main()
{
    int anArray[1];
    int anotherArray[2];

    anArray[0] = 1;
    anotherArray[0] = 2;
    anotherArray[1] = 4;

    registerObject("oneint", (method)oneInt_new, 1);
    addBang("oneint", (bang) oneInt_bang);
    registerObject("twoint", (method)twoInt_new, 2);
    addBang("twoint", (bang) twoInt_bang);

    oneInt *a = newObject("oneint", anArray);
    twoInt *b = newObject("twoint", anotherArray);
    object_bang("oneint", a);
    object_bang("twoint", b);

    return 0;
}
```

Task: Now we have to pass the class name together with the object to the object_bang() function which is not really necessary. Instead use the described inheritance mechanism and create a base class *object* which only holds the class name as a char array. Add the class name member to both structs *intOne* and *intTwo* too.
In the *newObject()* function store the name argument in the class name member.

A solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXOBJECTNAMESIZE 24
#define MAXNUMBEROFOBJECTS 10

typedef void* (*method)();
typedef void* (*methodInt)(int);
typedef void* (*methodIntInt)(int, int);
typedef void (*bang)(void *);

typedef struct registeredIntObject
{
    char name[MAXOBJECTNAMESIZE];
    method newMethod;
    bang bangMethod;
    int argc;
} registeredIntObject;

registeredIntObject objectLookupTable[MAXNUMBEROFOBJECTS];
int currentIndex = 0;

typedef struct object
{
    char className[MAXOBJECTNAMESIZE];
} object;

typedef struct oneInt
{
    char className[MAXOBJECTNAMESIZE];
    int val1;
} oneInt;

void *oneInt_new(int val1)
{
    oneInt *x = (oneInt *)malloc(sizeof(oneInt));
    x->val1 = val1;
    return x;
}

void *oneInt_bang(void *x)
{
    printf("%d\n", ((oneInt *)x)->val1);
}
```

```
typedef struct twoInt
{
    char className[MAXOBJECTNAMESIZE];
    int val1;
    int val2;
} twoInt;

void *twoInt_new(int val1, int val2)
{
    twoInt *x = (twoInt *)malloc(sizeof(twoInt));
    x->val1 = val1;
    x->val2 = val2;
    return x;
}

void *twoInt_bang(void *x)
{
    printf("%d\n", ((twoInt *)x)->val1);
    printf("%d", ((twoInt *)x)->val2);
}

void registerObject(char *name, method m, int argc)
{
    strcpy(objectLookupTable[currentIndex].name, name);
    objectLookupTable[currentIndex].newMethod = (method)m;
    objectLookupTable[currentIndex].argc = argc;
    currentIndex++;
}

void addBang(char *name, bang b)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;
        i++;
    }

    objectLookupTable[i].bangMethod = b;
}

void object_bang(void *x)
{
    int i = 0;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(((object *)x)->className,
           objectLookupTable[i].name))
            break;
        i++;
    }

    (objectLookupTable[i].bangMethod)(x);
}
```

```
void *newObject(char *name, int *argv)
{
    int i = 0;
    void *x = NULL;
    while(i<MAXNUMBEROFOBJECTS)
    {
        if(!strcmp(name, objectLookupTable[i].name))
            break;
        i++;
    }

    if(objectLookupTable[i].argc == 1)
    {
        void* (*new_ptr)(int) =
           (methodInt) objectLookupTable[i].newMethod;
        x = (new_ptr)(argv[0]);
        strcpy(((object *)x)->className, name);
    }

    if(objectLookupTable[i].argc == 2)
    {
        void* (*new_ptr)(int, int) =
           (methodIntInt)objectLookupTable[i].newMethod;
        x = (new_ptr)(argv[0], argv[1]);
        strcpy(((object *)x)->className, name);
    }
    return x;
}

int main()
{
    int anArray[1];
    int anotherArray[2];

    anArray[0] = 1;
    anotherArray[0] = 2;
    anotherArray[1] = 4;

    registerObject("oneint", (method)oneInt_new, 1);
    addBang("oneint", (bang) oneInt_bang);
    registerObject("twoint", (method)twoInt_new, 2);
    addBang("twoint", (bang) twoInt_bang);

    oneInt *a = newObject("oneint", anArray);
    twoInt *b = newObject("twoint", anotherArray);
    object_bang(a);
    object_bang(b);

    return 0;
}
```

And now we now everyting in order to understand the Pure Data external object structure☺

```c
#include "m_pd.h"

static t_class *helloworld_class;

typedef struct _helloworld
{
    t_object  x_obj;
} t_helloworld;

void helloworld_bang(t_helloworld *x)
{
    post("Hello world !!");
}

void *helloworld_new(void) HIER FEHLT WAS!!
{
    return (void *)x;
}

void helloworld_setup(void) {
    helloworld_class = class_new(gensym("helloworld"),
                        (t_newmethod)helloworld_new,
                        0,
                        sizeof(t_helloworld),
                        CLASS_DEFAULT,
                        0,
                        0);

    class_addbang(helloworld_class, helloworld_bang);
}
```

```c
void *myClass_free(myClass *x)
{
    if(x != NULL)
        free(x);
}
```