4. Object-oriented programming

The given overview of lexical and grammatical rules focused on the common points between C/C++ and AleC++. This chapter will also focus on similarities between AleC++ and its basis languages. If the reader is not familiar with the object programming introduced by C/C++, and this chapter is too succinct to explain it completely, the authors suggest consulting one of the numerous manuals on the subject.

Object-orientation enables better reusability of already developed program parts. With this concept software is developed in phases, layer-by-layer. Diverse, multifarious building blocks are developed first, and then used to put together to form larger blocks, which can be used again to build some other blocks. Alecsis was developed to model large number of various electronic components and systems. Many of them can be considered as composed of some building blocks. That is why the authors included the construct of object programming in AleC, and turned it into AleC++.

Before we approach object programming we will discuss some rules of C++, not necessarily directly connected to object programming, since we will need them in the coming chapters. Those rules address functions overload, standardized constants, inline functions, initial values of formal parameters, etc.

4.1. Constant variables

You already read about constants in the chapter on lexical rules. It is possible to name a constant by using the preprocessor command `define`, and then use that name in the program.

```
#define PI 3.141
```

ANSI-C and C++ introduce variables whose value should not be changed in the program.

```
const double PI = 3.141;        // constant of type double
const No = 123;                 // constant of type int
```

These constant has to be initialized immediately upon declaration (unless it is an external variable), since that is the only place initialization can be done.

Interesting effects can be produced using combinations of qualifier `const` and various pointers, as the following example shows:

```
const c=2;             // constant, int type
const *pc = &c;        // pointer to a constant
int *const cp = pc;    // constant pointer
const *const cpc = &c; // constant pointer to a constant
int *vp = &c;   // error - pointer must not point to a constant
```

Pointer `pc` can be changed afterwards in the program, but not the value stored on the address it points to. This is possible to do in the case of the pointer `cp`, but in this case the value of the pointer cannot be changed. Finally, neither the value of the pointer `cpc`, nor the value on the address it points to can be changed. The last example illustrates a semantic error -- if it was allowed for the pointer `vp` to be set to the address of constant `c`, the value of the constant can be changed indirectly using the pointer.

Qualifier `const` can be used with function declaration, too. The declaration of standard library function `strcpy`, given below, indicates that only string `dst` changes within the declaration block. Using this technique, an error can be detected if a pointer to a constant is passed to a function where it can be changed.

```
char* strcpy (char* dst, const char* src);
```

4.2. References

All arguments are passed to functions by value, which means local copies of original arguments are created, and a change in a copy does not affect the original. The pointer to the variable needs to be passed to the function, if the programmers want to change the variable inside the function. A certain number of situations exist where we want the same effect without the use of operator `&`. We refer to passing of large object by reference in order to save time. It is possible to declare object in AleC++ as a reference to another object of known type. We will encounter this situation when we declare *formal parameters*, *local object*, and *the type of the data that the function returns.*

4.2.1. Formal references

If you declare a formal parameter of a function as a reference, the address of the argument will be passed to the function and not the copy of the argument. Then, every change of the parameter changes the argument. Declaration of a reference is the same as in the case of the pointer, except we use the operator `&` instead of `*`:

```
f1 (int pi)  { pi++; }
f2 (int& pi) { pi++; }
f3 () {
    int i=1;
    f1 (i);    // after return, i is still 1
    f2 (i);    // after return, i is 2
    f2 (i);    // i is now 3
}
```

If you do not like the idea of being able to change an argument in a function, you can always declare the argument as `const`. You cannot change the argument now in the function, but using references, you can still save some time in the transfer of large objects.

4.2.2. Local references

Reference object inside a function is practically another name for the initializing object. **Local reference has to be initialized immediately after declaration**, bearing in mind that every change in the reference will affect the original object.

```
int i; int& ri = i;    // ri is a synonym for i
ri = 2;                // i is now 2, as well
```

4.2.3. Reference-returning functions

A function can return a reference under its name. In this case the function call can be an *l-value*, meaning that you can write into the function call. This property is used mostly with the overloading of some operators. **The function has to return a valid reference, which is not a local object**, because local object is erased during the return from the function (dangling reference).

```
int &f1(int, int);
...
f1 (2,3) = 4;
```

In all three cases given above, if the initializing object is not an *l-value*, or if the rules demand type conversion, a temporary object will be created, and its address will be used. As this is usually unwanted, the compiler will warn you.

4.3. Function overload

Multiple functions of the same name can exist in C++ and AleC++. The compiler will not report redeclaration if all of them have different parametric profile (type, and number of parameters). In the moment of the function call, compiler decides which function satisfies the actual call the best, taking into consideration the number of implicit conversions that will have to be applied to arguments.

```
double f1 (int);      // the first declaration
double f1 (int);      // copy - O.K.
int f1 (int);         // error - difference is in the returned type
double f1 (int, const char*); // O.K. - different profile
...
double d1 = f1 (2);           // call to f1(int)
double d2 = f1(3, "string");      // call to f1(int, const char*)
```

Compiler will easily find its way around an overload of the name of the function, if it has the information about the profile. The linker, however, cannot determine the version based solely on the name of the symbol.

Alecsis linker uses **type-safe linkage**. It secures the uniqueness of the function name by adding an especially coded parameteric profile (name mangling) to that name. For instance, the first version of `f1` in the symbol-table of linker will be `f1_i` (having one integer parameter), the second `f1_iPc` (having one integer and one pointer to character as parameters).

One should note that the function overloading is very useful for simulation, especially logic simulation. One can create a single name for a logic operator (for instance, `and` gate) which can have two or three inputs.

4.4. Default values of function parameters

We expanded the ability to have default value of parameters in Alecsis to include specific simulation constructs. If an actual argument does not appear on the appropriate place, its default value is placed there. This enables function calls in many different ways, with the parameters always defined.

```
f1 (int i, double j, int k=2);  // k has the default value 2
int p = f1 (2,2.3);             // parameter k is set to 2
int q = f1 (2, 5.1, 7);         // here, k equals 7
```

The only restriction in the number and type of the parameters with initial values is that **all parameters to the right from the first parameter with the initial value have to have initial values**, as well. An overload of this function can confuse the compiler:

```
char *f (int, int=2, int);     // error - the third argument does
                               // not have a default value
int *fp (int), *fp(int, double=2.2);   // two versions of fp
int *p1 = fp(1, 5.6); // O.K. - call of the second version of fp
int *p2 = fp(1); // ??? - both variants are suitable - conflict
```

Note: All initial values have to be constant expressions.

4.5. Inline functions

Ordinary functions are translated and stored in the library, and linker finds them if used in the code. It pays off to insert the function in the code instead a function call in case when the function is only a few lines long, as the time needed to for the function call is longer that the time of function execution in such case. The **precondition is that the function is already defined in the same file** using the key word `inline` before its definition. This is used to speed up significantly the execution of the program. However, certain restrictions apply, such as: inline functions cannot have any loops or unconditional jump commands, cannot declare static variables (local yes), cannot pass their address, cannot be recursive, etc. They are most useful when working with small, but often called functions.

```
inline double sum_a_b(double a, double b) { return a + b; }
```

4.6. Functions with variable number of arguments

This feature is a part of C, and it enables the writing of functions like `printf`.

```
#include <varargs.h>
void print_menu (const char *title, ...) {
    char *field;
    char *args;

    printf(menu %s options:\n", title):

    va_start(args, title);

    int nmenu=0;
    while ((field=va_arg(args, char*))!=0) {
        printf("\t%d: %s\n", ++nmenu, field);
    }
}
```

At least one argument must be given in the list of formal arguments. The last given parameter (in our example, there is only one given parameter -- `title`) is used as the argument for macro `va_start` from `varags.h` file. Macro `va_start` sets pointer `args` to the memory location **after** the parameter title, i.e. to the memory where the next argument is. All other parameters will result from the call of macro `va_arg` by passing the initialized pointer `args`, and the expected parameter type. The type has to agree with the type of the argument, since the compiler is not able to perform implicit conversions in this case.

Note: If the parameter to be read using `va_arg` is of type `char` or of some other short type, the second parameter of `va_arg` must be `int`.

Note: Coma is not necessary in between the last defined parameter and the symbol '...':

```
void print_menu (const char *title ...);
```

Implementation of functions with the variable number of arguments utilizes the fact that all formal parameters of the function are stored in consecutive memory locations. However, some computers have special alignment rules. Computers *Silicon Graphics* and *HP 9000 s700/800* store parameters of type `double` on the memory locations that can be divided by 8. This option is implemented in `varargs.h` file, you just have to define `DWORD_ALIGNMENT` flag before including this file:

```
#define DWORD_ALIGNMENT
#include <varargs.h>
```

For other computers where Alecsis until now installed, such alignment is not necessary. If you install Alecsis on some new type of computer, that is not predefined in `Makefile`, you should see the alignment rules for that computer in file:

```
/usr/include/varargs.h
```

that is used by C. You can then adapt Alecsis `varargs.h` to fit your needs.

4.7. Visibility area resolution operator

This operator is the first in the series of operators not used in C. This operator (two consecutive colons (`::`)) was designed to solve the situations when it is not clear which variant of an object is used, in case when symbols with the same name exists in different visibility areas. It is widely used as a part of objects whose type is a class, but it can be used independently:

```
int i;              // global i
main () {
    int i, j;       //local i and j

    j = i;          // refers to local i
    j = ::i;        // refers to global i
}
```

4.8. Classes

Classes are a special case of composite types, similar to structures. They allow the expansion of the existing system of types. Classes can have members -- data, but unlike structures, classes can encompass a number of appended functions. These appended functions are methods, which manipulate the objects, overload existing operators to make them applicable to newly defined types, as well as provide mechanisms of implicit conversion in case of dealing with "changed" types. Fully defined class represents a type, equal in status to already existing types.

We emphasize that classes differ form structures in that classes can have functions for members (in version 2.0 C++ structures can too, but this was left out in AleC++). Access to members stays the same as in C - operators '`.`' for objects, and '`->`' for pointers to objects.

```
class X {
    int a; double b;
    char s[20];
};

main () {
    X x, *px;

    x.a; x.b;
    px->a; px->b;
}
```

4.8.1. Access to class members

Members of classes are data and functions. They are **local** in reference to the class, and cannot be referenced outside the function. These functions are methods, since they define the operations with the objects of the class. The access to the members of a class is not simple doe to the need to control the access. Members of a class are initialized as `private` (that is no one except their own methods have the right to access them). Members and methods have to be defined as `public` to be accessible. We will talk about the third type of access -- `protected` in the chapter on inheritance.

4.8.2. Declaration and definition of methods

Methods declared in a class have to be defined somewhere. Shorter methods can be defined within the class, which makes them `inline` functions. It is possible to use members of the class not yet declared (as in the case of definition outside of class) within the methods defined within the class, since those methods are not translated until the class is entirely defined. Methods defined outside a class can be `inline`, too, but the key word has to be used in this case.

The following is an example of a class:

```
class Point {
        int xVal, yVal;
   public:
     void set () { xVal = yVal = 0; }
     void set (int x, int y) { xVal = x; yVal = y; }
     void show() { printf("xval = %d, yval = %d\n", xVal, yVal); }
     int Xval () { return xVal; }
     int Yval () { return yVal; }
};
```

The class `Point` has only two data of `int` type - `xVal` and `yVal`. These are private, and cannot be accessed from outside, since only class methods have the right of access. All methods are defined within the class, and are therefore inline. Overloaded method `set` gives them their values, `show` displays them on the screen, while methods `Xval` and `Yval` return their values. These are the only legal operations with the object of this class. Restriction of possible operations on members of a class is a valuable advantage of object programming, since it limits who and how can change objects of a particular class. Errors are easier to find, since it is always specified who is responsible for a particular operation.

```
f1 () {
    Point p1, p2;
    int x,y;

    p1.set();             // p1.xVal and p1.Yval are set to 0
    p2.set(2,3);          // p2.xVal = 2, p2.yVal = 3
    p1.show();
    p2.show();
    x = p1.xVal;          // error, xVal is a private member
    y = p1.Yval();        // O.K. - Yval is a public method
}
```

4.8.3. Keyword this

In the previous paragraph methods have referenced members of the class, as if these were known to them in advance. All methods have an additional level (besides usual levels of visibility) -- class level. This level is narrower in the order of searching than the global, but wider than the level of formal parameters. This means that a class member masks global variable of the same name, while a formal parameter or a local variable masks the class member. **The class object, whose members are used is passed as a hidden parameter in every appended function.** This can be made visible using the keyword `this`. This keyword represents exactly the passed object, and is manipulated as any other object of the same type.

In C, the keyword `this` refers to the **pointer of the object** passed to the appended functions, so the access to the individual members requires the operator of indirection `->` (eg. `this->xVal`, `this->yVal`). Alecsis virtual processor uses a table of pointers to the sources of the most frequent operands, and is able to **treat `this` as an object**, and not an operand. This renders dereferencing of the word `this` unnecessary, which can save time in larger methods. Access to particular members is possible using operator "`.`" (eg. `this.xVal`, `this.yVal`).
Nevertheless, we intend to change this to be the same as in C in the following releases of Alecsis, not the confuse the user.

You can explicitly use `this` in order to differentiate the names of the class members from names of other variables, in case of masking of the members of the class. The second option is to list names of classes and using the resolution operator:

```
class X {
        int x, y;
    public:
        void set1 (int x, int y) { this.x = x; this.y = y; }
        void set2 (int x, int y) { X::x = x; X::y = y; }
};
```

4.8.4. Static methods and class members

It is not legal to use all specifications the method of allocation (extern, register, auto) inside the body of a class. Mentioned specification can be applied to the members and functions, with different affects.

4.8.4.1. Static members

When objects of a class are defined, every object generates its own copy of the individual members, which occupy different memory locations. If we want the present state of object characteristics, including number, state, etc., static members can provide the necessary information. **There exists only one copy of the static member of the class, no matter how many copies class objects exist**. These members are internally implemented as static, or global variables, and do not affect the size of the class (only non-static members do). Declaration of the static element is not a definition, therefore in the case of the class on the global level it is the same as for any other global variable, but using access operator:

```
class X {
    int a, b;
    static s;   // declaration of the static member of the class
                // of int type
};              // class occupies 8 bytes

int X::s = 0;   // definition of the static member of class X
```

Access is the same for static members as for any other member of the class:

```
X::s = 3;       // access beyond any object
X x;            // object x class X
int xs = x.s;   // access using object x
```

4.8.4.2. Static methods

These functions are defined as methods, but are very similar in behaviour to global functions. These functions do not accept current class object as a hidden parameter, since they are used outside an object, so key word `this` cannot be used with these functions. They have to accept a pointer to an object explicitly in order to use it:

```
class X {
        int xval;
    public:
        static void fx (class X*);
};
...
X x;                // object x

X::fx (&x);//call of static function independently from the object
x.fx (&x);          // classical  method of access

void X::fx (X *x) { // definition of static fn. fx, class X
    xval = 1;       // error - which xval is this referring to?
    this.xval = 1;  // error, as well
    x->xval = 2;    // O.K.
}
```

Static member and functions lessen the congestion of the global area with symbols, help define which global or static symbols belong to which class, and offer the ability to limit the access using `private` keyword.

4.8.5. Class friends

One may need that other functions, not only the class methods, are allowed access the private members of the class. To enable that, one may declare functions or classes as *friends*.

4.8.5.1. Friendly functions

A function can be defined as a “friend” of the class by using the key word `friend`. A friendly function has the right of access to private members of the class:

```
class Y;                  // incomplete Y class declaration
class X {
        int xval, yval;
    public:
        set (int x, int y) { xval = x; yval = y; }
        friend int f1 (X *);  // global function f1 is a friend
```

```
        friend Y::fy (X *);   // method fy of class Y is a friend
};

class Y {                  // ending of class Y declaration
        int Yy;
    public:
        fy (X*);
};

int f1 (X* x) { return x->xval + x->yval; }     // definition f1
Y::fy (X* x) { return x->xval + x->yval; } // definition Y::fy
```

Friendship declaration does not mean passing an object as an implicit argument, thus an object must be passed explicitly to the function. Therefore, global function cannot use `this` even when declared as a friend. The example above shows that a method of one class can be friend of another. The friendship declaration does not change the meaning no matter what area of the class it is situated in (public or private). A method and a friendly class with the same parametric profile do not cause warning messages about redeclaration.

4.8.5.2. Friendly classes

If the intention is for all the methods of one class to have access to all the members of another class, the first class can be declared as friendly:

```
class X;
class Y {
        int xval, yval;
        ...
        friend class X;       // can do without "class"
};
```

This approach makes the job easy, but you need to be careful, since if everyone is defined as friend than the whole system loses meaning since everyone can change data.

4.9. Constructors and destructors

It is possible to declare arbitrary number of methods inside a class. Those methods are called explicitly, as any other function. Some other functions have special meaning, and are the basis for object-oriented programming in AleC++.

Variables declared within an area of visibility (most common way - using '{') have existence period ending with the closing of the area of visibility (most likely -- using '}'). Compiler allocates enough space for those variables, and consequently frees the space after the area of visibility is closed. In order to allow the class object to behave as object of predefined types (`int`, `double`, etc.) we need to define constructors and destructors as separate methods.

The purpose of the constructor is to allocate memory space for pointers, which are members of classes, and to initialize them whenever an object of that class is declared. Destructor does the reverse -- it frees the memory occupied by a constructor (deinitialization is not necessary). Destructor is not necessary for class objects without pointers, because there is nothing to free, but the constructor is.

4.9.1. Constructors

Constructor is the method that has the same name as the class. Constructor does not return anything, but it can have parameters. It can be overloaded. Constructor needs not to be called explicitly, since compiler calls it whenever it creates an object. In order to make that happen, constructor has to be defined as `public`.

```
class Point {
     int xVal, yVal;
  public:
     Point (int x, int y) { xVal = x; yVal = y; }
     Point () { xVal = yVal = 0; }
     void show () { printf("xval= %d, yval = %d\n", xVal, yVal); }
};
```

Arguments are defined in case the constructor has parameters:

```
Point p1, p2(2,3), p3 = p2;
```

Object `pl` does not have arguments, so compiler calls constructor without parameters. Object `p2` uses constructor with two `int` type parameters, while `p3` is initialized by copying the object `p2` (constructor is not invoked). If one of the mentioned ways is not used, compiler will report an error.

In the case of object `p3`, compiler made a shallow copy of the object `p2` by copying its bit-pattern. If an object involves a pointer pointing at allocated memory, it is necessary to define a special type of constructor -- copy constructor, which accepts reference to a class, e.g. `Point (Point&)`. The object `p2` would be passed by reference into that constructor, where space would be allocated for the memory pointed by the pointer. In that way, the deep copy of the object would be made.

Constructor can be called in expressions as an ordinary function. Than it creates temporary, nameless object, which is destroyed after the exit from the visibility area. Initialization can be done using this temporary object.

```
Point p1(2,3), p2 = Point(4,5);
```

This means going the long way, but temporary objects play an important role in the in operator overloads and implicit conversions.

Constructors for the objects created on the stack are called every time the control gets to their declaration place. Constructors for global, or static variables are called before the beginning of the simulation (if one uses Alecsis simulator), that is before the `main` function (if one uses AleC++ for C++-like programs). Constructors cannot be static functions. They can have parameters with initial values, just as any other function.

4.9.2. Destructors

Destructors bear the same name preceded with the character ‘~’ as the originating class. Destructor do not return any result, nor accept any parameters, which is the reason why destructors cannot be overloaded. Compiler calls them immediately before an object is destroyed in order to free the memory occupied by the object, except in the case of global, or static variable when they are used at the and of the simulation (or on leaving the `main` function). If the command return appears before the end of a block, destructors are called before the exit from a function. As in the case of constructors, destructors cannot be static objects.

```
class X {
        ...          // class members
    public:
```

```
        ~X();        // Destructor for class X
};
```

4.10. Operator overload

C programmers know it is possible to declare variable types of structures, but that little can be done with them using current C operators. Structures can be copied into each other, can be passed to functions, returned using `return`, and that is it. (We are describing operations with objects, and not their particular members.) Structures cannot be added because compiler does not know how to do it, that is instructions for addition of operands that are not integer or real do not exist. Operator overload mechanism "explains" to compiler how to apply the existing operators to object of class type.

4.10.1. Global level overload

Overload of operators is possible by defining identically named functions to be called when needed. Word `operator` should be put before the operator name, which is the special signal for compiler to transform it into a function name.

```
class complex {
       double re, im; // real and imaginary part of complex number
    public:
       complex (double r=0.0, double i=0.0) { re = r; im = i; }
       friend complex operator+ (complex, complex);
};
...

inline complex operator+ (complex l, complex r) {
    return complex (l.re+r.re, l.im+r.im);
}
...
foo () {
    complex c1 (2,3), c2 (7,8), c3 = c1 + c2;
}
```

Class `complex` from this example is a new type of data that represents complex numbers. The class constructor defines the number, and becomes a complex zero in the case of left out arguments. Usually, compiler would not know to add two complex numbers, and would report an error. However, operator '+' overloads on the global level if used on two objects of the same type. Since the operator function is global, it could not access to private members unless previously defined as friendly within the class. The function is defined as inline**,** due to its shorth length. Object passes to it using shallow copying (by value), which is satisfactory in this case, since they are only 16 bytes long, without pointers. The temporary object created in the operator function returns to the environment using command `return` (shallow copying). Here, it initializes object `c3`, which gets the value of 9+11i.

All operators in AleC++ can be overloaded, except:

```
.   .*   ::   ?:   <-   now   $   $$   @   ddt   d2dt2   idt   sdt
```

In general, binary operator ♦ in the expression `op1♦op2` can be overloaded using global function `operator♦(top1,top2)`, where `top1`, and `top2` are appropriate types of operands with the allowed

implicit conversions. Unary operator ◊ in the expression ◊ op can be overloaded using global function `operator◊(top)`, where `top` is the type of operand with the allowed implicit conversions. If the objects are large, it pays off to pass them to the function by address (binary operator function would be declared as `operator♦(top1&,top2&))`.

Operator function can be called as an ordinary function, i.e. as affix:

```
c3 = operator+(c1,c2);
```

but it is a matter of style to use it as an infix operator.

4.10.2. Overload using methods

An operator function can be a method of the class, in which case all rules for methods apply.

```
class complex {
    ...          // the same as in the class "complex" given before
    complex operator+ (complex r)
        { return complex (re + r.re, im + r.im); }
};
```

The method `operator+` is implicitly defined as *inline*. In binary operations, those methods have **one parameter only**, since the left operand is implicitly passed (`this`). Unary operator methods **do not have parameters** because the only operator is passed implicitly, too. You can call these methods either explicitly (as members), or as operators:

```
foo {
    complex c1(2,3), c2(1,1);
    complex c3;

    c3 = c1.operator+(c2);       // a classical method call
    c3 = c1 + c2;                // a real overload
}
```

AleC++ allows overloa of existing operators, but it does not allow for a definition of new operators because that would introduce confusion into the rules concerning association and priority.

4.11. Overload of operator =

Assignment operator (=) can be overloaded as any other operator, but certain restrictions apply. When inheriting (see the section on inheritance), every derived class has to give its own version of the operator, since the definition of the operator is not transferable. Further on, to cover all the applications of this operator you need to define a copy constructor (constructor takes the reference of the source class). These definitions apply to the following cases:

- initialization of objects (e.g. `X x1, x2 = x1;`)
- copying of objects (`x2 = x1;`)

- passing the object as an argument of the function call (transfer by value)
- return of the object using command `return`

4.12. Overload of implicit conversions

You are, by now, familiar with the rules regarding the conversion of operands of different types. The user can add a new conversion to the list of legal conversions, so that the new type (class) functions in the same manner as an inrinsic type. For that, you need appropriate constructors and conversion functions.

```
class X {
        ...
    public:
        X (int);
        operator int (&X);
        X operator+ (X&, X&);
};

X x1(2), x2(3);
int i, *p;

x2 = x1 + 2; // O.K. - 2 is converted into X(2) using constructors
i = x2;      // O.K. - x2 is converted into int using operator int
x2 = x1 + p; // error - types do not agree
```

Appropriate constructors convert common variables into objects of desired type. Conversion of an object into another type requires a conversion function. That function is defined the same way as the operator function, but instead the name of the operator you need to give the name of the target type (it is legal to give a pointer to the type, e.g. `operator int*`). In the example above, `x2` is converted to `int` and the result is assigned to variable `i`. The last line of the example is an error, since the compiler does not know how to add an object of class `X` and the pointer to type `int`.

4.13. Dynamic allocation of memory

Dynamic allocation of memory is known to the users of C. In C++ this feature is raised on the level of a language. Instead of library functions `malloc` and `free`, we have new operators `new` and `delete`. This feature exist in AleC++, too.

4.13.1. Allocation - operator new

Pointers can point to address of a static variable, but can also point to a part of memory allocated under **heap** (free memory whose addresses increase toward to other end of the stack). Allocation can be sufficient for an object of a certain type, as well as for a number of such objects.

```
int *p = new int;              // 4 bytes allocated
int size = 20;
```

```
char *s1 = new char[size+1];   // allocated (size+1)*sizeof(char)
char *s2 = (char *)malloc((size+1) * sizeof(char));   // C style
Point *p1 = new Point (2,3);   // new + constructor call
Point *p2 = new Point;         // default constructor
int *i = new int (5);          // pointer i allocated and set to 5
```

Command `new` is used for memory allocation regardless of type. In contrast to `malloc`, you define a type and not a number of bytes, because the compiler calculates that number automatically. In case you are allocating pointers to classes, you can pass a list of arguments to the constructor, too. You can also set a value pointed by some pointer to a particular value.

Allocation of a vector has a restriction: constructors with arguments are not allowed, only default constructors.

```
Point *vp = new Point [5];
```

Operator `new` can be overloaded to increase the control over allocation. If you do that, you can access the standard (global) operator using access resolution (`::new`).

4.13.2. Deallocation - operator delete

Already allocated pointer to an object needs to be deallocated when leaving the visibility area (in contrast to objects, pointers to objects need to be explicitly allocated using `new`, and deallocated using `delete`), otherwise the memory they point to will not be accessible nor free, which can cause unpleasant consequences.

```
delete i;          // frees the pointer i
delete p1;         // call destructor ~Point, frees p1
delete [5] vp;   // calls destructor for every index of vector vp
                 // and then frees the vector
```

4.14. Inheritance

An analysis of a large number of programs and the experience of programmers revealed that the largest portion of time needed for the writing of a new program is spent on the same routines such as functions for list, stack, tree, graph manipulations. The mechanism of inheritance is a characteristic of object-oriented programming, and it uses these functions as building elements of programs. The essence of inheritance is that “children” inherit “parents” with possible changes. In C++ and AleC++ the children are **derived** and the parents are the **base** classes. There is a mono inheritance (one base class) and multiple inheritance (more base classes). Since derived classes can be base classes for some other classes, the whole system can be very complex as a tree or **hierarchy of inheritance** similar to a family tree.

4.14.1. Inheritance and rules concerning access rights

In a declaration of a derived class, base classes are listed after the derived class name and the colon “`:`”.

```
class A {
        int a1, a2;
```

```
    public:
        int a3;
};

class B {
        int b1;
    public:
        int b2;
};

class C : private A, public B {
        int c1, c2;
    public:
        int c3;
};
```

Class `C` is a derived class, which inherits characteristics of classes `A` and `B`. This means that object of class `C` would be 12+8+12=32 bytes. The problem arises with the access rights. The `private` members in the base class are inaccessible in the derived class, and if the class is inherited as private no member (even if declared `public`) is accessible. The situation is somewhat better with the class `B`. Member `b2` is accessible, but `b1` is not. These restrictions can be circumvented f the members of base classes are declared as `protected` instead of `private`. This makes them accessible for methods and friends of the derived class only.

4.14.2. Access to members in the hierarchy bearing the same name

Every class on the inheritance tree has its own area of visibility, which enables the appearance of the members and methods of the same name in many places and their mutual masking. Access to masked parameters is still possible using the operator of resolution:

```
class A {
        int a;
    public:
        void show () ;
};

class B : public A {
        int b;
    public:
        void show ();
};

foo {
        B b;

        b.show();             // calls show of class B
        b.B::show();          // same
        b.A::show();          // show of base class A
}
```

4.14.3. Virtual base classes

Virtual classes are still **not functioning** in the current release of Alecsis.

No class can be repeated in the list of base classes (after the character ":"). However, the same class can appear more than once if it was the base class for more than one base class.

```
class A; class B: A {...}; class C: A {...};
class D: B, C {...};
```

Class A appears twice in D - once in B and another time in C. If that creates problems, you can declare A as `virtual` class. Regardless of the complexity of the hierarchy, this class can appear only once in the final hierarchy.

4.14.4. Construction and destruction of derived classes

A derived class and its base classes can have constructors. When an object of the derived class type is declared, the compiler calls the constructors of the base classes in the order of inheritance, and then it calls the constructor for the derived class. However if a virtual class exists on the tree, its constructor is called first. The opposite happens with the destructor call - first the derived class, than base, and finally virtual.

The situation complicates if constructors of base classes require arguments. The arguments are listed in the definition (not declaration) of the derived class constructor:

```
class X {
        int xval;
    public:
        X (int x) { xval = x; }
};

class Y {
        int yval;
    public:
        Y (int y) { yval = y; }
};

class Z : public X, public Y {
        int zval;
    public:
        Z (int x, int y, int z) : X(x), Y(y) { zval = z; }
};
```

Arguments for base constructors are listed after the colon ":" and can include all legal expressions. The visibility area after ":" includes formal parameters of the derived constructor, source class and global variables. Symbols from those regions can participate in the forming of the argument. If a member of a derived class is also a class, the arguments for the constructor can be passed using the same syntax. Finally, common members of classes can be initialized this way, too. For more details on this subject, consult a manual of C++.

```
class X {
        int xval;
        Y y;            // y has type Y (class)
    public:
        X (int x) : xval(x), y(x+2) {}  // xval is set to x,
                                        // constructor for y
                                        // obtaines argument x+2
};
```

4.15. Virtual functions

Virtual functions are still **not functioning** in the current release of Alecsis.

The mechanism of **late linking** is a fundamental characteristic (if not the requirement) of object-oriented programming. All references to a method and functions are treated as global symbols used by linker in the early phases of the simulation. If the linker is not able to resolve such references the program stops (**early linking**). Contrary to this, **late linking** allows a late decision on the choice of the method, even during the execution of the program (when the method is invoked)

A method needs to be declared as `virtual` in the base class if the mechanism of late linking is to be used. This allows redefinition of that function in a derived class. A redefined function becomes virtual (the word `virtual` is not necessary for it), with the condition that the returned type and the parametric profile is the same as in the base class. **Virtual function cannot be static**, but can be a friend. Global functions cannot be virtual.

Both base and the derived versions have to be defined, or the base function can be declared as purely virtual, but the whole class becomes **abstract**.

```
class Base {
        ...
    public:
        virtual int foo (int);  // base version of foo
};

class A: public Base {
        ...
     public;
        int foo (int);      // redefinition of foo - derived class
};
...

Base b;
A a;
Base *bp;

bp = &a;
bp ->foo(2);                        // calls A::foo
bp = &b;
bp ->foo(2);                        // calls Base::foo
```

Note that the assignment of pointer to a derived class to a pointer of a base class is legal and that the compiler does the implicit conversion. The reverse is not legal without the explicit conversion (**`cast`** operator).

Abstract classes have at **least one purely virtual function**. Those classes serve as the basis for inheriting. It is not legal to define an object, declare a formal parameter, or return the result using return if the abstract class is the type (it is possible with the pointers or references of abstract classes).

```
class object {
        ...
    public:
        virtual void draw() = 0;   // purely virtual function
};
class circle : public object {
        ...
    public:
        void draw() { ... }        // definition circle::draw
};
class rectangle : public object {
        ...
    public:
        void draw() { ... }        // definition rectangle::draw
};

circle c1, c2;         // two objects, class cicrle
rectangle r1, r2;      // two objects, class rectangle
static object *ob[] = { &c1, &r2, &c2, &r1 }; // vector of pointer
        // Assignment of addresses of derived objects is
        // performed using implicit conversion into the base class

ob[0].draw();          // calling circle::draw
ob[1].draw();          // calling rectangle::draw
ob[2].draw();          // calling circle::draw
ob[3].draw();          // calling rectangle::draw
```

If the base class is not abstract, then you need to define both the base and the derived implementation of the virtual function.