SECURE APPLICATIONS PROGRAMMING

Structure of the lecture:

Teaching: Course + Laboratory

Evaluation method: Written Exam in exam session

Objectives:

- Learning theoretical elements and practical principles of application development using object oriented technology;
- Secure application development using OpenSSL library.
- GUI using Qt (?!)

SECURE APPLICATIONS PROGRAMMING

Part I – C Programming Language

- **Reviewing Pointers, Functions – Transfer of parameters, Compiling in command line and IDE**
- **Reviewing Arrays: definition, use of standard and user-defined elements, memory allocation, functions using arrays**
- **Reviewing Data Structures: simple linked lists – stacks, queues, functions for operations, binary search trees**
- **Reviewing Files: LIB and DLL generation, types of files, work on BMP files – digital Watermarking, ECB and CBC encryption/decryption, RLE compression, preprocessing**

SECURE APPLICATIONS PROGRAMMING

Part II – C++ Programming Language

- Reviewing Structure with pointers to functions C
- Concepts of class, object, access methods, this pointer in C++
- Classes having extensions in dynamic memory - Constructor, copy constructor, overloading the operator =, destructor in C++
- Namespaces and overloading the unary and binary operators – Work with LIBs and DLLs in C++
- Conversions between different types of objects (cast operator, operator = and copy constructor), object arrays, const modifier, static member typology (static), constant objects, constant pointers to objects and pointers la constant objects in C++
- Derived classes, inheritance, polymorphism and class hierarchies using pointer to table of pointers to virtual functions in C++
- Template functions and classes in C++
- STL – Standard Template Library in C++

SECURE APPLICATIONS PROGRAMMING

Part III – OpenSSL Library

- Use of hash functions – MD5, SHA-1, SHA-256
- Use of algorithms with symmetrical keys – AES in ECB and CBC, DES in CFB and 3DES
- Use of algorithms with asymmetrical keys – RSA
- Creating digital certificates – X509 v3

SECURE APPLICATIONS PROGRAMMING

Part IV – Qt (?!)

- Cross-platform development framework
- C++ application design
- C++ application using OpenSSL library with GUI in Windows

BIBLIOGRAPHY

Part I – C Programming Language

- http://ism.ase.ro
- Ion Smeureanu – "Programarea in limbajul C/C++", Editura CISON, 2001

Part II – C++ Programming Language

- http://ism.ase.ro
- Ion Smeureanu, Marian Dardala – "Programarea orientata obiect in limbajul C++", Editura CISON, 2002
- Bjarne Strastroup – The Creator of C++, "The C++ Programming Language"-3rd Edition, Editura Addison-Wesley, http://www.research.att.com/~bs/3rd.html

Part III – OpenSSL Library

- http://www.openssl.org

Part IV – Qt Framework

- http://qt-project.org/

I.1 Reviewing Pointers, Functions

Fundamental elements

What is a pointer?

What should the pointer use for?

What is the role of a function?

What means pointer arithmetic?

Examples:
- pointer to char and int
- sum of two values
- compiling at command line and IDE

I.2 Reviewing Arrays unallocated dynamically

Bi-dimensional array – data structure:

- *Homogenous;*
- *Contiguous;*
- *Linear.*

a_{11}	a_{12}	. . .	a_{1n}	a_{21}	. . .	a_{2n}	a_{31}	. . .	…	a_{m1}	. . .	a_{mn}

a_{11}	a_{21}	. . .	a_{m1}	a_{12}	. . .	a_{m2}	a_{13}	. . .	…	a_{1n}	. . .	a_{mn}

C/C++ source code

```
#include<stdio.h>

void main()
{
        int y[10][10],m,n;
        scanf("%d",&m);
        scanf("%d",&n);
       for(int i=0;i<m;i++)
            for(int j=0;j<n;j++)
                y[i][j]=0;
}
```

Initializing values?

Conversion ASCII-to-INT

Problems?

Y vs. Y+i
Y vs. (Y+i)+j
Y vs. Y[i][j]

I.2 Reviewing Arrays unallocated dynamically

Fundamental elements

What is an array?

What is an array use for?

How does it transmit an array as parameter to a function?

What is the relation between an array and a pointer?

Example:
- sum of items returning a temporary address
- compiling in Visual Studio

I.3 Reviewing Arrays allocated dynamically

HEAP *memory allocation:*

- *Function* **malloc***;*
- *malloc.h;*
- *Allocation inside vs. outside.*

void *malloc(size_t size);

- HEAP address
- NULL – there is not available memory

No. of bytes to be allocated

Standard type on 32/64 bits

I.3 Reviewing Arrays allocated dynamically

Deallocation of **HEAP** *memory by standard functions:*

- *Function* **free***;*
- *malloc.h;*
- *Deallocation inside vs. outside.*

Problems? → Run-Time Error

void free(void *memblock);

It does not return a result

Memory block allocated in ***HEAP***

I.3 Reviewing Arrays allocated dynamically

Uni-dimensional array in **HEAP***:*

C/C++ source code

```
#include<stdio.h>
void main(){
        int *pV,n;
        scanf("%d",&n);
        pV=(int *)malloc(n*sizeof(int));
        ...
        free(pV);
}
```

Variable to manage the array in HEAP

Allocation of HEAP for the array

Deallocation of HEAP for the array

&pV vs. pV
pV vs. pV+i
***pV vs. *pV+i**
***pV vs. *(pV+i)**

I.3 Reviewing Arrays allocated dynamically

Bi-dimensional array in **HEAP***:*

C/C++ source code

```
#include<stdio.h>
void main(){
        int **pM,m,n;
        scanf("%d",&m);
        scanf("%d",&n);
        pM=(int **)malloc(m*sizeof(int*));
        for(int i=0;i<m;i++)
            *(pM+i)=(int*)malloc(n*sizeof(int));
        ...
        for(int i=0;i<m;i++)
            free(*(pM+i));
        free(pV);
}
```

Variable to manage the array in HEAP

Allocation of HEAP for the array

Deallocation of HEAP for the array

&pM vs. pM
pM vs. pM+i
***pM+i vs. *(pM+i)**
***(pM+i)+j vs. *(*(pV+i)+j)**

I.3 Reviewing Arrays allocated dynamically

Struct – data structure:

- *Defined by user;*
- *Heterogeneous;*
- *Template to describe each element from collectivity;*
- *List of characteristics.*

C/C++ source code

```
#include<stdio.h>

typedef struct angajat{
      char nume[20];
      long int salariu;
      int ore;
};
long int salariu_total;
angajat companie[30][20];
```

Array as field in struct

Initializing methods

Array of structures struct

30: maximum number of departments

20: maximum number of employees in a department

I.3 Reviewing Arrays allocated dynamically

Fundamental Elements

What is an array allocated dynamically?

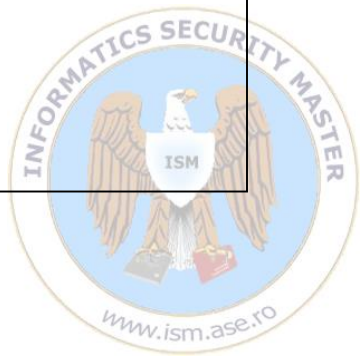
What is the target of working with arrays allocated dynamically?

How does it transmit an array as parameter to the functions?

What is the relation between an array allocated dynamically and a pointer?

What is the advantage to work with structures?

Example:
- sum of elements per row with memory leak
- compiling in Visual Studio

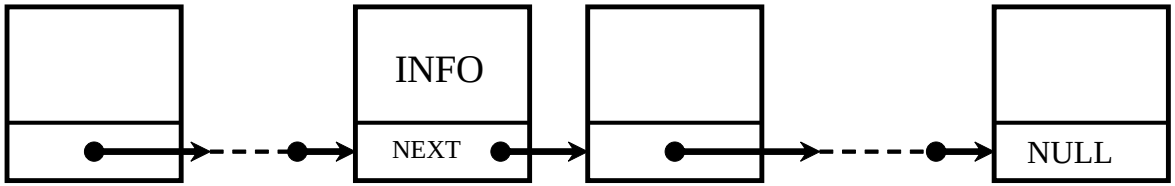


I.4 Reviewing auto-referred dynamic data structures

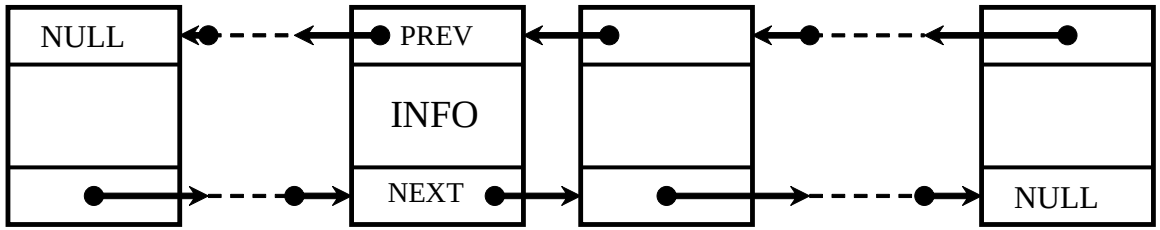
Linked lists:

- *Collection of elements – nodes;*
- *Order relation – criterion: relative pozition;*
- *Elements of same type: data, linking information (one direction, two directions);*
- *Dynamic character of the length;*
- *Use of* **HEAP** *memory to store elements;*
- *Sequential referring of the elements.*

I.4 Reviewing auto-referred dynamic data structures



Simple linked list



Double linked list

I.4 Reviewing auto-referred dynamic data structures

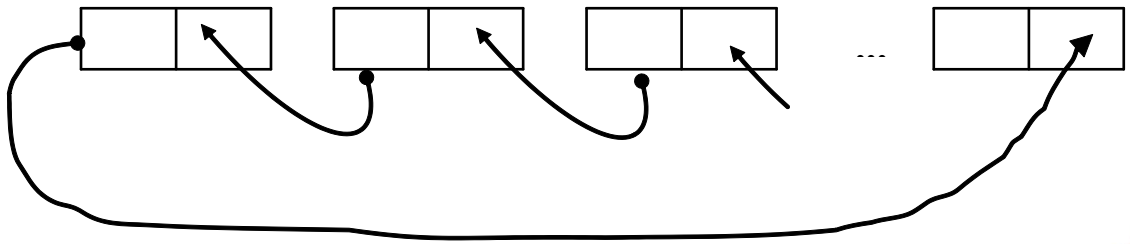
Basic operations with lists:

- inserting a node;
- creating a list;
- deleting a node;
- deleting a list;
- interchange nodes.

I.4 Reviewing auto-referred dynamic data structures

Circular list:

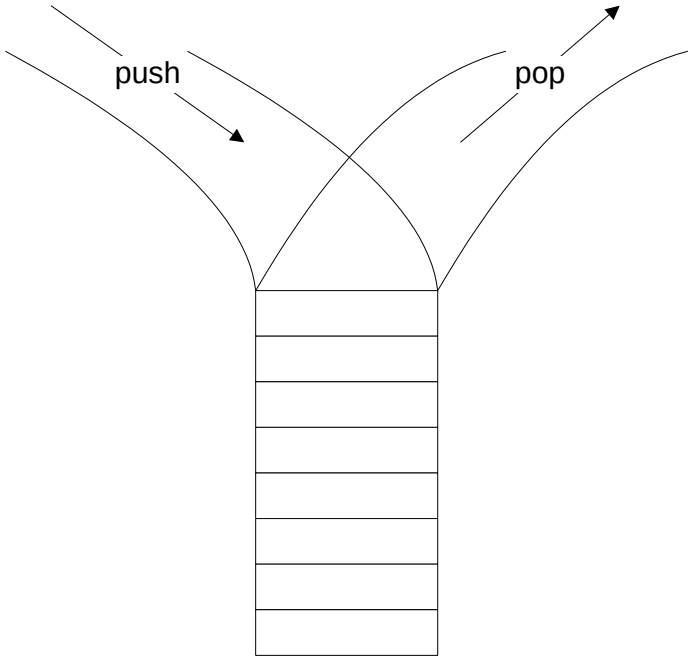
- simple/double;
- closing the linking of the nodes;
- there is not the start node and end node;
- it is managed by the address of a single node.



I.4 Reviewing auto-referred dynamic data structures

Stack:

- simple list; there is no relation with functions stack;
- working rule LIFO – inserting (push) and deleting (pop) of the node to the same end;
- it is managed by the address of a node called the stack top.



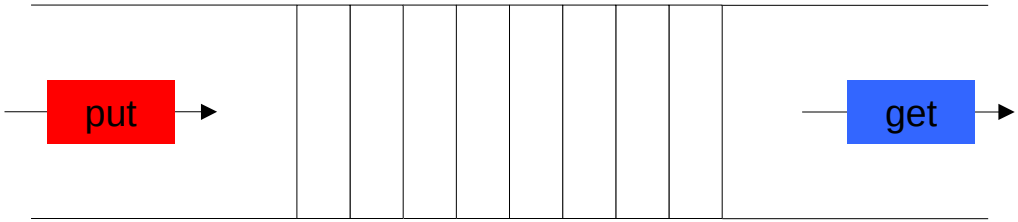
I.4 Reviewing auto-referred dynamic data structures

Queue:

• simple list;

• working rule FIFO – inserting of the node (put) at one end (end of queue) and deleting (get) of the node at the other end (beginning of queue);

• it is managed by the addresses of the first node (beginning of queue) and the last node (end of queue).



I.4 Reviewing auto-referred dynamic data structures

Fundamental elements

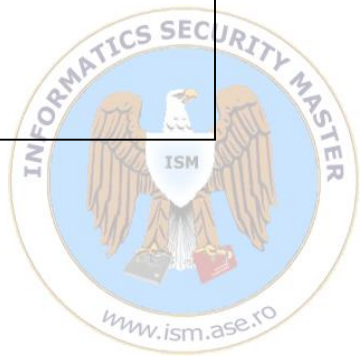
What is a simple linked list?

What is a double linked list?

What is a binary search tree?

What is the goal to work with auto-referred structures?

Examples:
- Simple linked list – buffer overrunning
- binary search tree and stack overflow
- compiling in Visual Studio



I.5 File and Preprocessor

PREPROCESSING

- *Stage that precedes compiling*
- *Based on symbols defined on #*
- *It is NOT executable code*
- *Conditioned compiling of the code*
- *Symbolic substitution*
- *Enumerated type*
- *Macro definitions*

I.5 File and Preprocessor

Symbolic substitution:

- *Based on directive #define*

```
#define NMAX 1000
#define then
#define BEGIN {
#define END }

void main()
BEGIN
int vb = 10;
int vArray[NMAX];
if(vb < NMAX) then printf("less");
else printf("greater");
END
```

I.5 File and Preprocessor

Symbolic substitution:

- *Availability of the symbol:*
  - *End of source code;*
  - *Redefinition of the symbol;*
  - *Removing the current definition.*

```
#define NMAX 1000
….
#define NMAX 10
…
#undef NMAX
```

I.5 File and Preprocessor

Enumerated type :

enum typeName {symbolList} variablesList

- Values are in sequence
- Value of each symbol can be specified explicitly

enum supplies {book, notebook, pencil = 4, pen = 6, crete}

I.5 File and Preprocessor

Macro definitions:

#define macroName(symbolList) expression

Example:

```
#define POW(X) X*X
#define ABS(X) (X) < 0 ? – (X) : (X)
```

C/C++ source code

```
...
int x=POW(3);
int y=POW(3+2);
...
```

I.5 File and Preprocessor

Macro definitions generating functions:

```
#define SUM_GEN(TYPE) TYPE sum(TYPE vb2, TYPE vb2) \
        { return vb1 + vb2; }
```

Conditioned compiling:

```
        #if expression_1
                        sequence_1
        #elif expression_2
                        sequence_2
        …
        #else
                        sequence_n
        #endif
```

I.5 File and Preprocessor

Conditioned compiling:

```
#ifdef macroName
        ...
#else
        ...
#endif
```

or

```
#ifndef macroName
        ...
#endif
```

I.5 File and Preprocessor

Operators # and ##:

- are used with #define
- operator # transforms the arguments into a string with "";

```
#define macro1(s) # s
```

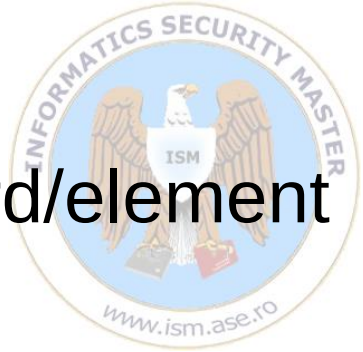
- operators ## concatenates 2 elements

```
#define macro2(s1, s2)  s1 ## s2
```

I.5 File and Preprocessor

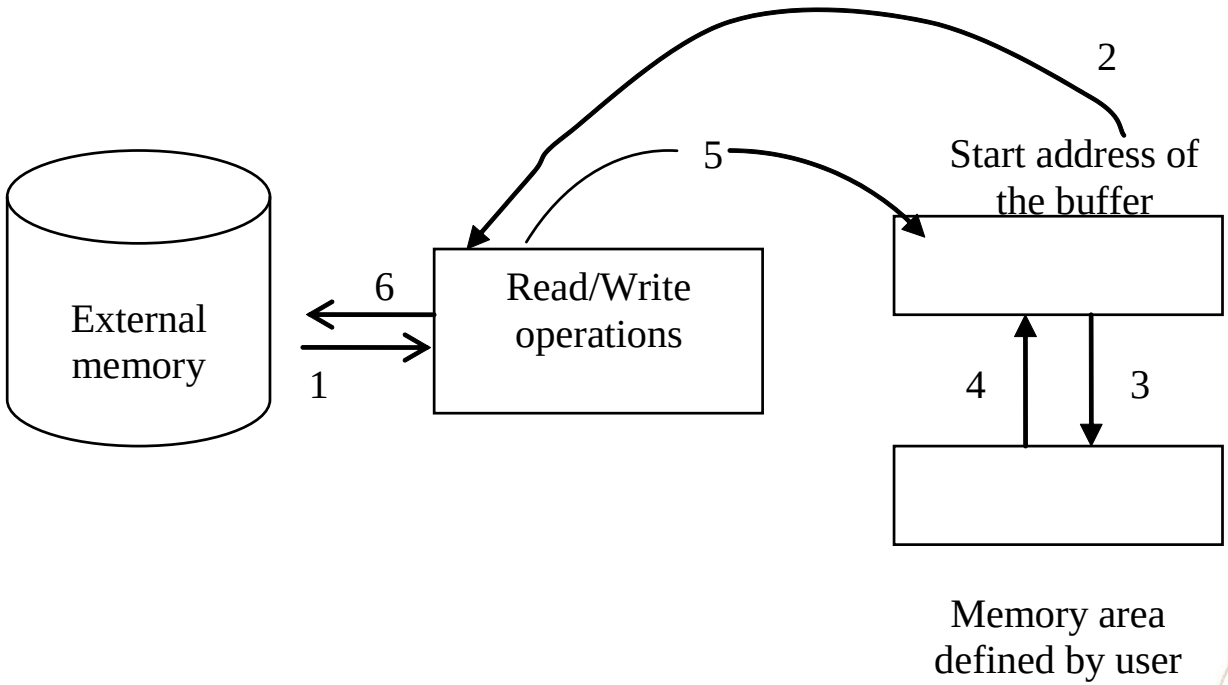
Files:

- organization system of data (logically, physicaly);
- external data structure;
- organization methods: positional (sequential, indexed sequential, random), complex ring, hierarchical, multiple double-linked chain, attributed string, inverted list ;
- access types: sequentially, directly;
- processing operations: file level, record/element level.



I.5 File and Preprocessor

Input/Output operations at record/element level:



I.5 File and Preprocessor

File processing at high level:

- structure FILE;
- file handling as pointer variable to FILE;

```
typedef struct  {
        int level;                  /* fill/empty level of buffer*/
        unsigned flags;      /* file status flags */
        char fd;                    /* file descriptor */
        unsigned char hold;  /* ungetc char if no buffer */
        int bsize;                 /* buffer size */
        unsigned char *buffer;    /* data transfer buffer */
        unsigned char *curp;      /* current active pointer */
        unsigned istemp;   /* temporary file indicator */
        short token;         /* Used for validity checking */
}       FILE;
```

I.5 File and Preprocessor

File processing functions:

- opening and assigning;

FILE* fopen(const char* externName, const char* openMode);

File identifier

Physical file name

File access mode

C/C++ source code

```
FILE * f, * g;
f=fopen("file.txt","a+");
g=fopen("file.bin","r+");
```

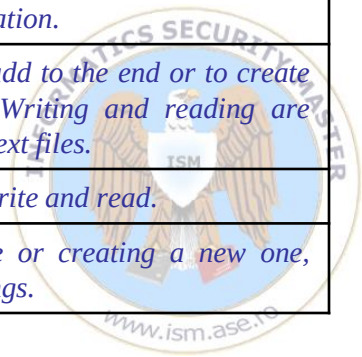
Assigning text file to **f**

Assigning binary file to **g**

_fmode:

- global variable defined stdlib.h
- default type of the assigned file
- possible values: _O_TEXT sau _O_BINARY

Mode	*Description*
a	*Opening an existing file to add data to the end or creating file if the file is not created. Only writing is allowed. Available only for text files.*
r	*Opening an existing file only for reading.*
w	*Overwriting an existing file or creating a new one, allowing only the write operation.*
a+	*Opening an existing file to add to the end or to create if the file is not on disk. Writing and reading are allowed. Available only for text files.*
r+	*Opening an existing file to write and read.*
w+	*Overwriting an existing file or creating a new one, allowing readings and writings.*



I.5 File and Preprocessor

File processing functions:

- file closing;

```
int fclose(FILE* id_file);
```

- 0, successful closing
- -1, failed closding

File identifier

C/C++ source code

```
fclose(f);
fclose(g);
```

I.5 File and Preprocessor

File processing functions:

• moving the file pointer to the beginning of the file;

```
void rewind(FILE* id_file);
```

File identifier

C/C++ source code

```
rewind(f);
rewind(g);
```

I.5 File and Preprocessor

File processing functions:

- verifying the end of file;

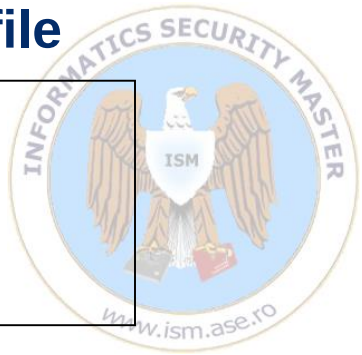
```
int feof(FILE* id_file);
```

- not null if the reading from file jumps the physical end of file mark
- 0, otherwise

File identifier

C/C++ source file

```
...
if(!feof(f){
...
}
...
```



I.5 File and Preprocessor

File processing functions:

- cleaning the file buffer;

```
int fflush(FILE* id_file);
```

File identifier

- 0, if the buffer was cleaned
- EOF, error (defined in stdio.h)

C/C++ source code

```
fflush(stdin);
gets_s(string,sizeof(string));
```

I.5 File and Preprocessor

File processing functions:

- current position of the file pointer;

```
int fgetpos(FILE* id_file, fpos_t* position);
```

- 0, successful positioning;
- not null, otherwise; assign to the variable errno the EBADF(FILE* inaccessible) or EINVAL (disabled id_file or position)

File identifier

Position of I/O pointer in the file, as relative number of the current byte

C/C++ source code

- long int
- __int64
- structure

```
fpos_t poz;
FILE * f;
...
fgetpos(f,&pos):
```

I.5 File and Preprocessor

File processing functions:

- position of the file pointer;

```
long ftell(FILE* id_file);
```

- no. of bytes from the beginning of the file
- -1L, otherwise

File identifier

C/C++ source file

```
long pos;
FILE * f;
...
pos=ftell(f):
...
```

I.5 File and Preprocessor

File processing functions:

• change position of file pointer;

```
int fsetpos(FILE* id_file, const fpos_t *position);
```

• 0, successful change

• EBADF (FILE * disabled) or EINVAL (disabled values for id_file/position)

File identifier

New position of file pointer

C/C++ source file

```
fpos_t pos;
FILE * f;
...
fsetpos(f,&pos);
...
```

I.5 File and Preprocessor

File processing functions:

- change position of file pointer;

```
int fseek(FILE* id_file, long offset, int origin);
```

- 0, successful change
- not null, otherwise

File identifier

No. of bytes of the offset to move the file pointer

Position against which the file pointer is moved:

- SEEK_SET (0);
- SEEK_CUR (1);
- SEEK_END (2).

C/C++ source file

```
long offs;
FILE * f;
...
fseek(f,offs,SEEK_SET);
...
```

I.5 File and Preprocessor

File processing functions:

• renaming/moving a file;

```
int rename(const char * oldName, const char * newName);
```

• 0, successful operation

• otherwise, EACCES (disabled path), ENOENT (non-existent oldName file), EINVAL (irregular symbols)

Old file name (complete path can be given)

New file name (complete path can be given)

C/C++ source code

```
char  *oldN,*newN;
...
rename(oldN,newN);
...
```

I.5 File and Preprocessor

File processing functions:

- deleting a file;

```
int remove(const char * fileName);
```

- 0, successful deleting
- -1, otherwise

Path of the file (it must be closed before removing)

C/C++ source file

```
char *fis;
...
remove(fis);
...
```

I.5 File and Preprocessor

File processing functions:

- reading without conversion;

```
size_t fread(void *buffer, size_t dim, size_t n, FILE *id_file);
```

- Storing location
- Block dimension
- No. of blocks
- File identifier
- no. of read blocks, successful operation
- 0 or negative value, otherwise

C/C++ source code

```
char string[10];
FILE *f;
int nr_name;
...
nr_name=fread(string,sizeof(char),7,f);
...
```

I.5 File and Preprocessor

File processing functions:

- writing without conversion;

```
size_t fwrite(void *buffer, size_t dim, size_t n, FILE *id_file);
```

- Storing location
- Block dimension
- No. of blocks
- File identifier
- no. of read blocks, successful operation
- 0 or negative value, otherwise

C/C++ source code

```
char str[10];
FILE *f;
int nr_name;
...
nr_name=fwrite(str,sizeof(char),7,f);
...
```

I.5 File and Preprocessor

File processing functions:

- reading/writing with conversion – transfer of character symbols;

```
int fgetc(FILE* f);
int fputc(int c, FILE *f);
int getc(FILE* f);
int putc(int c, FILE *f);
```

- reading/writing with conversion – transfer of strings

```
char* fgets(char* s,int n,FILE* f);
int fputs(const char* s,FILE* f);
```

- data transfer with controlled format:

```
int fprintf(FILE* f,const char* format[,…]);
int fscanf(FILE* f,const char* format[,…]);
```

II.1 Reviewing structures having pointers to functions

Fundamental elements

There is reference type in C (&)?

There is the concept and key word class in C?

How can be made the inclusion of the behavior in structure in C?

How are differences between C and C++ regarding syntax and functionality of the binary file?

Examples:
- passing function parameters by value, pointer and reference
- structure and independent function
- structure with pointer to the function
- class and objects in C++
- compiling Visual Studio

II.2 Classes and Objects

- Data structures including data and functions;
- Development of new data types – ADT (Abstract Data Types);
- Management of very large programs;
- Code reusability;
- Implementation of the OOP concepts – data encapsulation, polymorphism (“onr interface, many methods”), inheritance.

II.2 Classes and Objects

- Each object contains data (attributes/properties/ data members/fields) defined in class;
- Class defines functions (methods/operations) applied to the objects; these functions defines the object interface;
- Data are hidden in object and they can be accessed by functions defined inside the class – data encapsulation;
- Object are created by instantiation of the class;
- Data members and methods used by objects are defined by abstracting (class definition);
- State of the object is established by its data members (properties);
- Behavior of the object is established by its methods (functions);

II.2 Classes and Objects

Syntax to define a class:

```
class className
{
   accessType:
        properties;
        methods;
    accessType :
         properties;
         methods;
};
```

II.2 Classes and Objects

accessType:

- Describes the access type to the properties and methods of the class;
- Covering area is closed by definition of another access type or class end;

```
class Test{
        public:
                        …
        private:
                        …
        public:
                        …
}
```

II.2 Classes and Objects

accessType:

- private
  - Default access type assigned by compiler to the beginning of the class;
  - Allows the access only from inside the class;
- protected
  - Is used in class hierarchies built by inheritance implementation;
  - Allows the access from inside the class and derived classes;
- public
  - Allows the access from inside and outside of the class;

II.2 Classes and Objects

Data members (properties):

- Define the state of the object;
- Are initialized by instantiation of the object;
- Are defined in private area and they are referred by public methods;
- Define memory space reserved by an instance of the class (exception: static properties)
- Particular types: constant, static;

II.2 Classes and Objects

Static data members (properties):

- Define properties that are not belong to an object;
- They are used by all instances of a class;
- Represents "global variables" belonging to a class;
- Specifying the static modifier does NOT represent a data definition (it is only a description);
- Initialization is made outside the class

II.2 Classes and Objects

Static properties:

- syntax:

```
class Test
{
  public:
        static int vb_1;
        static char vb_2;
};
```

II.2 Classes and Objects

Static properties :

- initialization outside the class
- used with operator of scope (class_name ::)

```
Test::vb_1;
```

II.2 Classes and Objects

this pointer:

- For Test class, this pointer has the type Test *;
- Represents the object address calling the member function of the class;
- All member functions has this pointer by default.

II.2 Classes and Objects

Member functions:

- Define the object interface;
- Allow the access to the object properties – data encapsulation;
- define the behavior of the object;
- Special classes of member function: constructor, destructor, copy constructor;
- Particular types of functions: static, inline;

II.2 Classes and Objects

Member functions:

- Can be defined inside the class

```
class Test {
        void memberFunction( ) { …};
};
```

- Can be defined outside the class using the operator of scope ::

```
class Test {
        void memberFunction( );
};
void Test:: memberFunction( ){…};
```

II.3 Classes and Objects

Constructor functions:

- Main role: allocation of the memory area for an object;
- Secondary role: initialization of the properties for an object;
- Types of constructors :
  - Default constructor
  - Parameter list constructor
  - Parameter list with default values constructor
  - Copy construction

II.3 Classes and Objects

Constructor functions:

- They have the same name with the class name;
- They have NOT return type because they return by default the memory area address of the built object;
- They are defined in public sections of the class;
- Default form is generated by compiler when a constructor is not defined by programmer;

II.3 Classes and Objects

Constructor functions:

- syntax:

```
class className{
public:
        className( ){…}
        className(parameterList){…}
};
```

- call:

```
void main () {
className obj_1;
className obj_2(constructorParameters);
}
```

II.3 Classes and Objects

```
class Test {
private:
        int prop_1;
public:
        …
};
```

Default constructor:

```
Test ( ) { prop_1 = 0; }
```

Parameter list constructor:

```
Test ( int val ) { prop_1 = val ; }
Test ( int val ): prop_1(val ) {}
```

II.3 Classes and Objects

Parameter list with default values constructor:

```
Test ( int val  = 0) { prop_1 = val ; }
```

IMPORTANT. This type of constructor replaces the previous classes of constructors.

II.3 Classes and Objects

Destructor functions:

- Main role: deallocation the memory of an object;
- They have the same name with the class name; to differ from constructors, their names are prefixed by operator ~;
- They have NOT return type because they return void by default;

II.3 Classes and Objects

Destructor functions:

- They are defined in public section of the class;
- Default form is generated by compiler when the programmer does not define a destructor;
- They are called by default for all objects before the end of program running;
- For local objects, they are executed in reverse order from execution order of the constructors;

II.3 Classes and Objects

- syntax:

```
class className {
public:
        ~className( ){...}
};
```

- Default call:

```
void main () {
        className obj_1;
}
```

II.3 Classes and Objects

Destructor functions:

IMPORTANT ! For member data dynamically allocated in constructor functions, it is MANDATORY to deallocate them in destructor function. Otherwise, the application generates memory leaks.

II.3 Classes and Objects

Static member functions:

- They define functions that do not belong to an object;
- They are used by all objects;
- They are "global functions" belonging to a class;
- They have access ONLY to other static members of the class;
- They are called by operator of scope ::
- They do NOT accept this pointer in parameter list.

II.3 Classes and Objects

Static member functions:

- syntax:

```
class className {
public:
        static void Fct_1( ){…}
};

void main( ) {
        className::Fct_1( );
}
```

II.3 Classes and Objects

Inline member functions:

- Small functions that are not called;
- At compiling time, the inline function calling is replaced by its source code in the same way like macro definitions;
- They allow quick execution of the code by elimination the effort of a function call;
- They lead to increase the dimension of the executable code;

II.3 Classes and Objects

Inline member functions:

- By default, the member functions defined inside the class are inline functions (it is NOT a rule, depending on compiler version);
- explicitly, a member function is defined as inline when the specifier inline is used to define the function;

```
class Test {
        void Fct( );
};
inline void Test:: Fct( ){…};
```

II.3 Classes and Objects

Access member functions:

- They allow the access (read / write operations) to the private properties of the class;
- Validation of the input data must be implemented;
- They are defined in public section;
- generally, the read functions are prefixed by word get and the write functions are prefixed by word set;

II.3 Classes and Objects

Access member functions:

```
class className {
private:
        int prop_1;
public:
        int Get_prop_1( ) {
                return prop_1;
        }

        void Set_prop_1(int val) {
                //validation of val
                prop_1 = val;
        }
};
```

II.3 Classes and Objects

Transfer of the objects as parameters to/from functions:

- By value (IMPORTANT: copy constructor and operator =)

```
class className {
…
};
className Fct1 (className obj);
```

- By reference (IMPORTANT: modifications and return)

```
void Fct2 (className & obj);
```

- By address (IMPORTANT: modifications and return)

```
void Fct3 (className * obj);
```

II.3 Classes and Objects

Copy constructor:

- Main role: allocation the memory area for an object and its initialization by an existent object;
- It has default definition given by compiler which copy bit by bit the existent object value in memory area of the built object;
- It is called by default in all cases of defining and initialization of new objects;

II.3 Classes and Objects

Copy constructor:

- syntax:

```
class className {
        public:
    className(className & existentObj){…}
};
```

- call:

```
void main () {
className obj_1(…);
className obj_2 = obj_1;
}
```

II.3 Classes and Objects

Copy constructor:

- Default call: compiler calls copy constructor to copy on function stack the values of the objects from parameter list (if these ones are passed by value);
- Default call : compiler calls copy constructor to copy on stack of the function which calls another one the value of the object returned by child function (if the object is returned by value);

II.3 Classes and Objects

Copy constructor:

- Default call:

```
class Test {
public:
        Test (Test & existentObj){…}
        void Fct1(Test obj1, Test *obj2) {…}
        Test Fct2(Test obj1) {…}
};

        void main () {
        Test obj_1, obj_2, obj_3, obj_4;
        obj_1.Fct1(obj_2, &obj_3);
        obj_4 = obj_1.Fct2(obj_2);
        }
```

Copy constructor called by default

II.3 Classes and Objects

Operator =

- Main role: it copies bit-by-bit the value of source memory area into destination memory area (the both memory areas are identical as structure and type);

- For objects, it copies the value of source object into destination object

II.3 Classes and Objects

Operator =

- Explicit call:

```
class className {
        …
    };

    void main () {
    className obj_1(…);
    className obj_2(…);
    obj_2 = obj_1;
    }
```

II.3 Classes and Objects

Operator =

- Mandatory overloading by member function

```
class className {
        className operator = (className obj)
        {
                //copying from obj to this pointer;
        }
};

void main () {
className obj_1(…);
className obj_2(…);
obj_2 = obj_1;
}
```

II.3 Classes and Objects

Pointers to member data (properties):

- They store "address" of a property within an object – as offset;
- Definition syntax:

```
propType className:: * propPointerName;
```

- Initialization:

```
propPointerName = &className::propName;
```

- Use:

```
className obj, *pobj = & obj;
propType var = obj.* propPointerName
propType var = pobj->* propPointerName
```

II.3 Classes and Objects

Pointers to member functions:

- They store "address" of a member function in member function list of the class – as offset;
- Definition syntax:

```
returnType (className:: * functionPointerName) (parameters) ;
```

- Initialization:

```
functionPointerName = & className:: functionPointerName;
```

- Use:

```
className obj, *pobj = & obj;
returnType var = (obj.* functionPointerName)(parameters)
returnType var = (pobj->* functionPointerName)(parameters)
```

II.3 Classes and Objects

Fundamental elements

Role of the class.

What is the requirement of the classes having pointer properties and what are the effects? – constructor, destructor, copy constructor, overloading of operator =

Members – properties and static member functions?

Examples:
- class and objects
- requirement of copy constructor and destructor
- complete class having pointer properties
- friend member functions
- compiling in Visual Studio

II.4 Inheritance and Polymorphism

Function overloading:

- **It implements the concept of polymorphism ("same thing, many views")**
- **Assigning of a symbol (function name) with more meanings;**
- **Difference is made by function signature = number and types of parameters;**
- **Return type does NOT represent selection criterion when the function is called**

```
int sum(int a, int b)
{
    return a+b;
}
```

Error at compiling time

Ambiguity

```
double sum(int a, int b)
{
    return a+b;
}
```

II.4 Inheritance and Polymorphism

Operators overloading:

- They are implemented by functions:

```
class Test{
        …
};
void main()
{
        Test t1, t2, t3;
        t1 = t2 + t3;
}
```

Interpretation

operator+(t2,t3)

(overloading by global function)

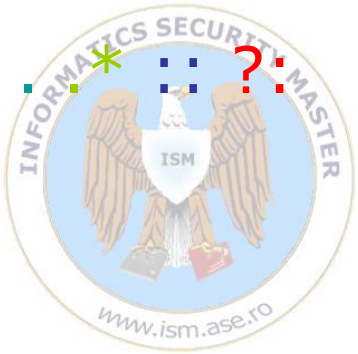
t2.operator+(t3)

(overloading by member function)

II.4 Inheritance and Polymorphism

Requirements of operator overloading:

- It does NOT change the operator priority;
- It does NOT change associativity;
- It keeps the number of parameters (cardinality);
- It does NOT create new operators;
- Overloaded forms does not automatically constitute;
- It does NOT overload the operators: . .* :: ?:



II.4 Inheritance and Polymorphism

Requirements of operator overloading:

- It is made by member functions or global functions;

EXCEPTIONS:

- Member function: ( ) [ ] -> =
- Global function: new delete

- It does NOT ensure commutativity;
- For operators like ++, -- the forms "before" and "after" are differently overloaded;

II.4 Inheritance and Polymorphism

IMPORTANT!

Operators overloaded by member functions have this pointer on first position!

```
class Test{
        Test operator+(Test t, int vb){
        ...
        }
};
```

operator + with 3 parameters !!!!!

II.4 Inheritance and Polymorphism

IMPORTANT!

Requirements of return type choosing:

- Waterfall call of the operator;
- If the return type is an object reference, then the object has not to be a temporary one;
- If the return type is an object, then the copy constructor must be defined by developer;

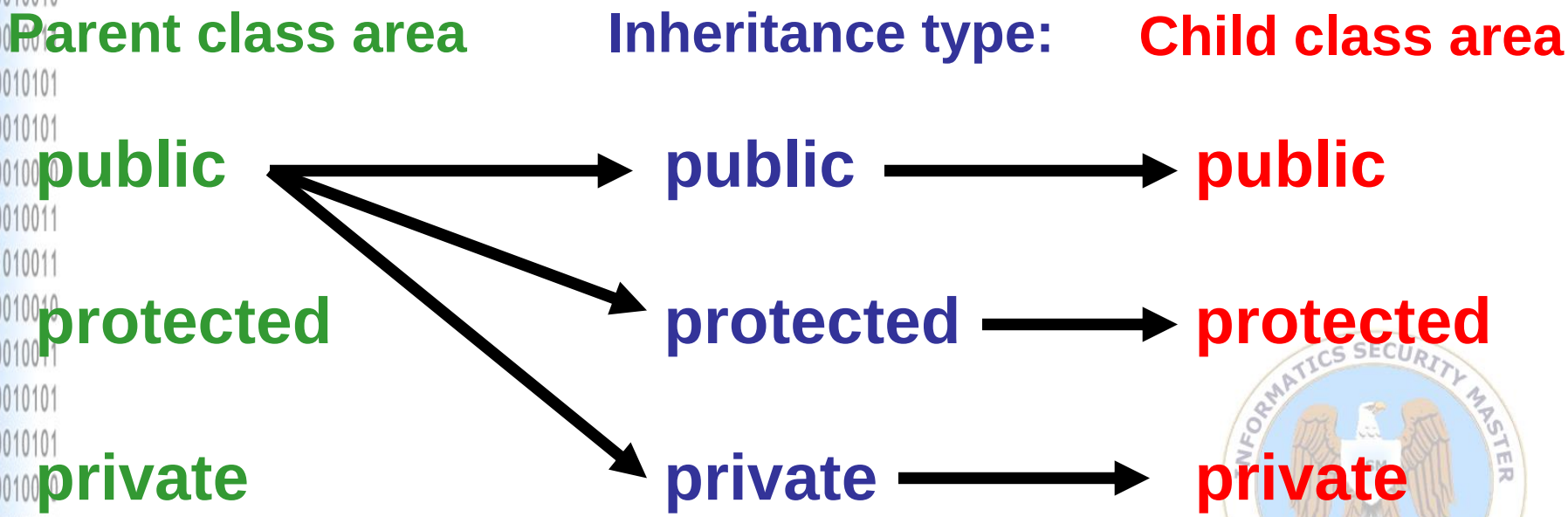
II.4 Inheritance and Polymorphism

- REUSE OF SOURCE CODE;
- Development of new classes starting from the existent ones;
- Derivation – existent class is derived into a new class;
- Inheritance – new class inherits the properties and member functions of the derived class (parent class);

```
class Parent{
};
class Child : inheritanceType Parent{
};
```

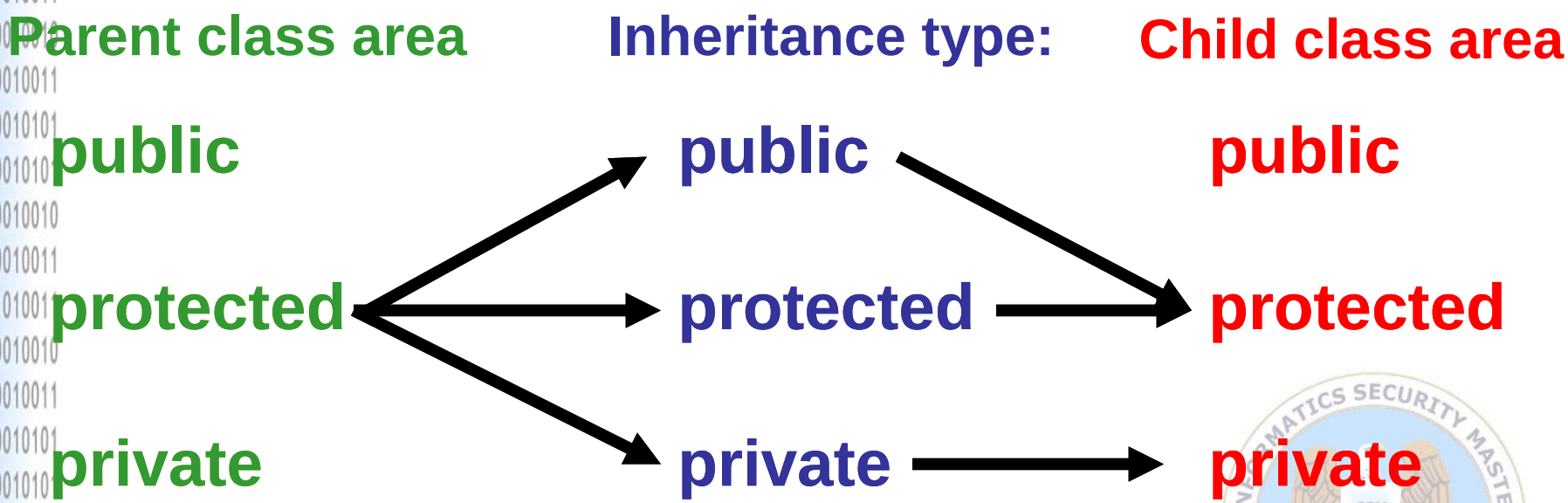
II.4 Inheritance and Polymorphism

- Access requirements are NOT removed in parent class;



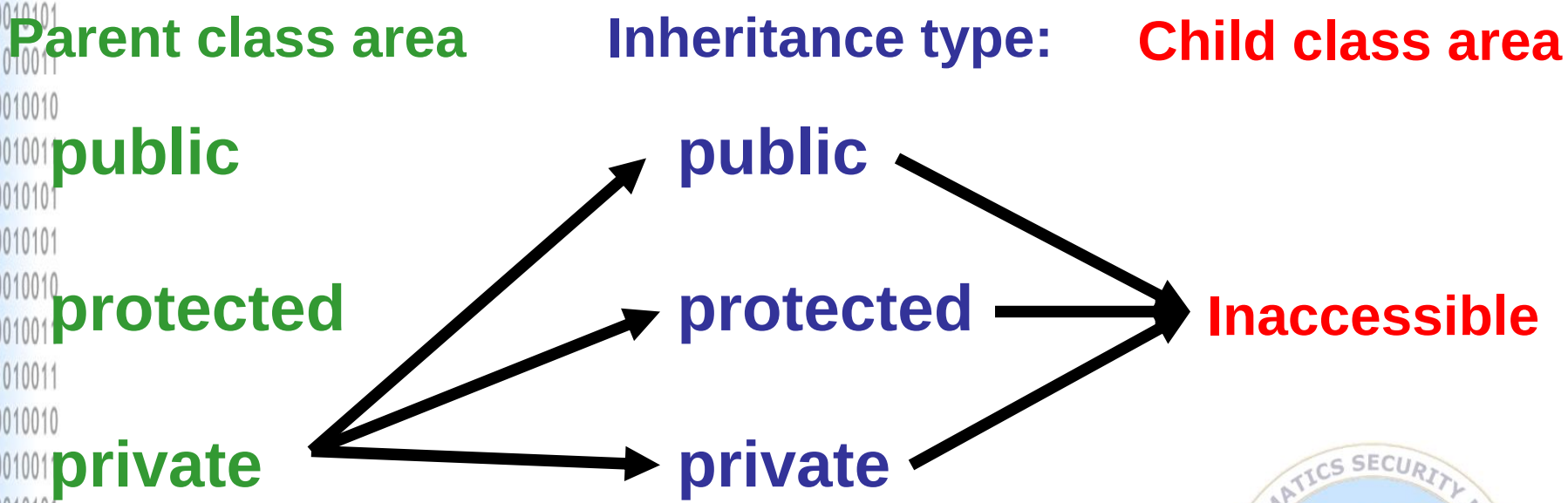
II.4 Inheritance and Polymorphism

- Access requirements are NOT removed in parent class;



II.4 Inheritance and Polymorphism

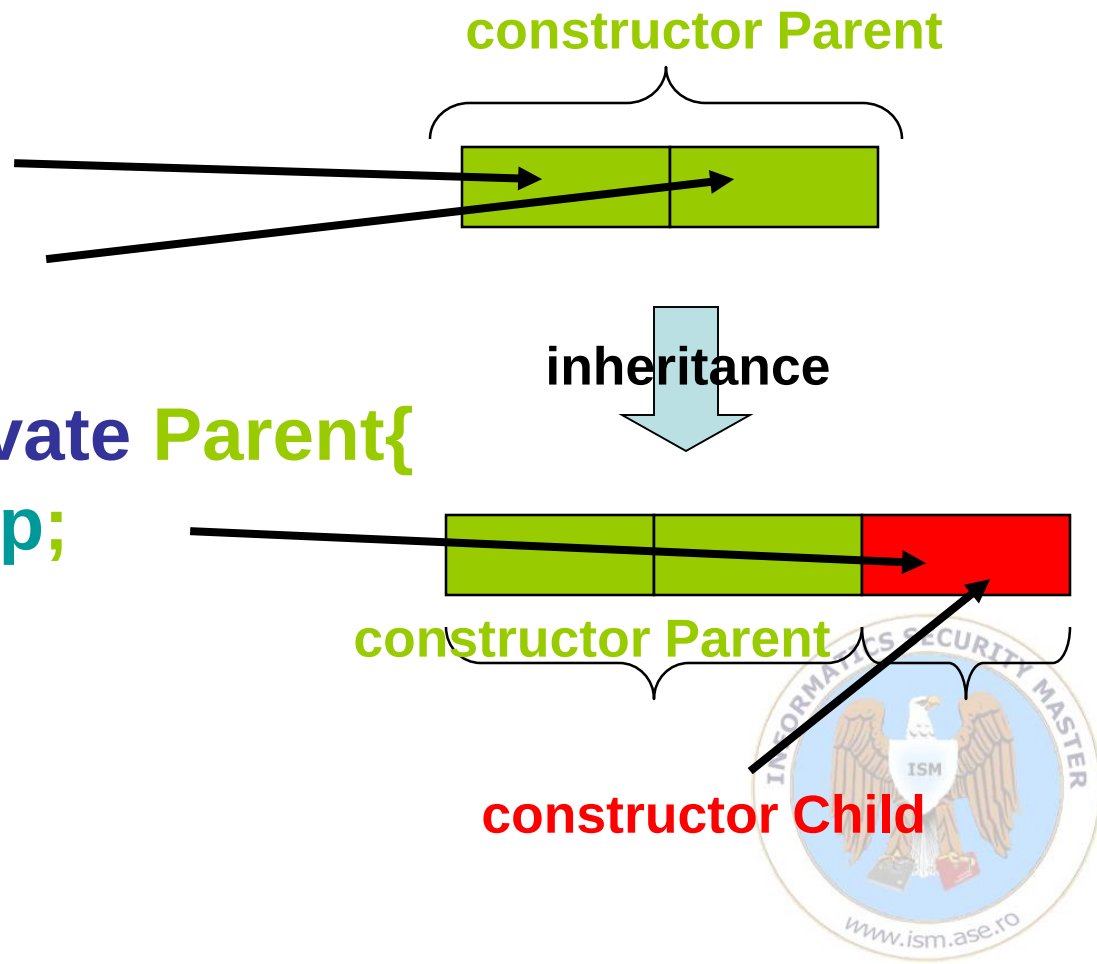
- Access requirements are NOT removed in parent class;



II.4 Inheritance and Polymorphism

- each constructor is strictly responsible for the class area represented by it.

```
class Parent{
        int prop1;
        int prop2;
};
class Child : private Parent{
        int newProp;
};
```



II.4 Inheritance and Polymorphism

- Building a child object = CONSTRUCTOR PARENT CLASS + CONSTRUCTOR CHILD CLASS

```
class Parent{
        Parent(){…}
        Parent(parametersList){…}
};

class Child : inheritanceType Parent{
        Child(){…};
        OR
        Child() : Parent(parametersList) {…}
};
```

Default call Parent()

Explicit call
:Parent(parametersList)

II.4 Inheritance and Polymorphism

- Destruction child object = DESTRUCTOR CHILD CLASS + DESTRUCTOR PARENT CLASS

IMPORTANT ! Each destructor must strictly focus on what class constructors made.

2 – deallocation property !!!!!!
(is it still allocated ???)

```
class Parent{
        int * sp;
        ~Parent(){delete [ ] sp; }
};
class Child : inheritanceType Parent{
        ~Child(){delete [ ] sp; }
};
```

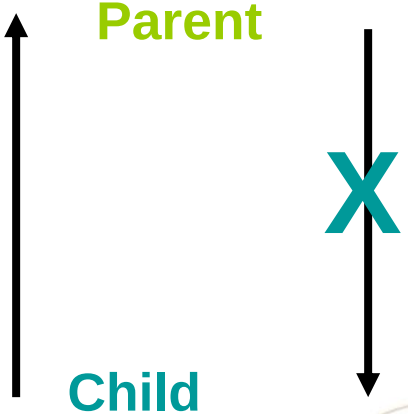
1- deallocation of inherited property

II.4 Inheritance and Polymorphism

- UPCASTING – default transformation of the child object or child object pointers into parent object or parent object pointers is allowed.

```
class Parent{
        ...
};

class Child : public Parent{
        ...
};
```



II.4 Inheritance and Polymorphism

- Functions having the same header in parent class and child lass can be built.

```
class Parent{
        int Fct1(int a){…}
};
class Child : private Parent{
        int newProp;
        int Fct1(int a){…}
};
```

```
void main(){
Child d1;

d1.Fct1(5);

d1.Parent::Fct1(5);
}
```

II.4 Inheritance and Polymorphism

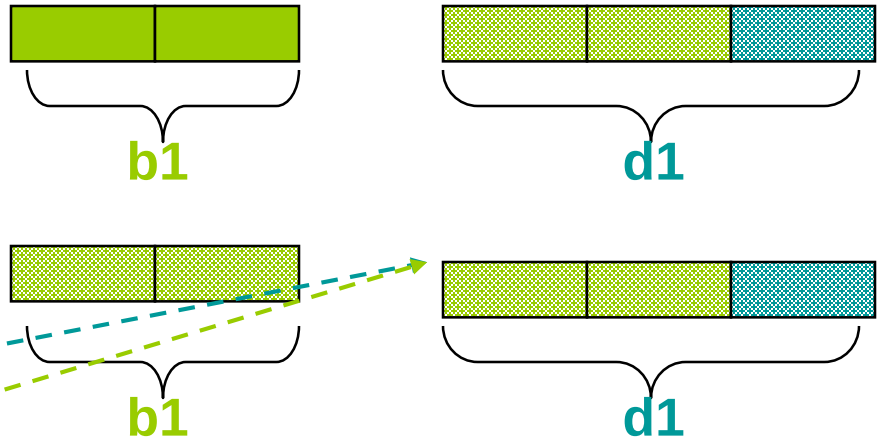
- *UPCASTING + member function overriding*

```
void main(){
    Child d1, *pd1;
    Parent b1, *pb1;

    b1 = d1;

    pd1 = &d1;
    pb1 = pd1;

    b1.Fct1(5);
    pb1->Fct1(5);
}
```



ALWAYS member function defined in Parent;

II.4 Inheritance and Polymorphism

Fundamental elements

Why the class inheritance is required?

Why polymorphism is required?

Types of polymorphism: overloading and overriding (object pointer, derivation-inheritance, virtual member function overriding)

Examples:
- classes Employee, Manager and SoftDev in namespace ismase
- polymorphism
- compiling in Visual Studio

II.5 Template and STL

Fundamental elements

What are template functions?

What are template classes?

Advantages and disadvantages? – DLL

Standard Template Library

Examples:
- template functions
- class Vector in namespace roase
- class std::vector in STL
- class std::list in STL with members pointers to Student
- compiling in Visual Studio

III.1 OpenSSL

Fundamental elements

www.openssl.org

Requirement to use OpenSSL library?

Advantages and disadvantages? – DLL

Examples:
- Building hash with MD5
- Encryption and decryption with AES - Rijndael
- Generating public and private keys for RSA
- Electronic signature with RSA
- Encryption for privacy with RSA
- Building X509 v3 certificate and reading ASN.1 DER
- compiling in Visual Studio