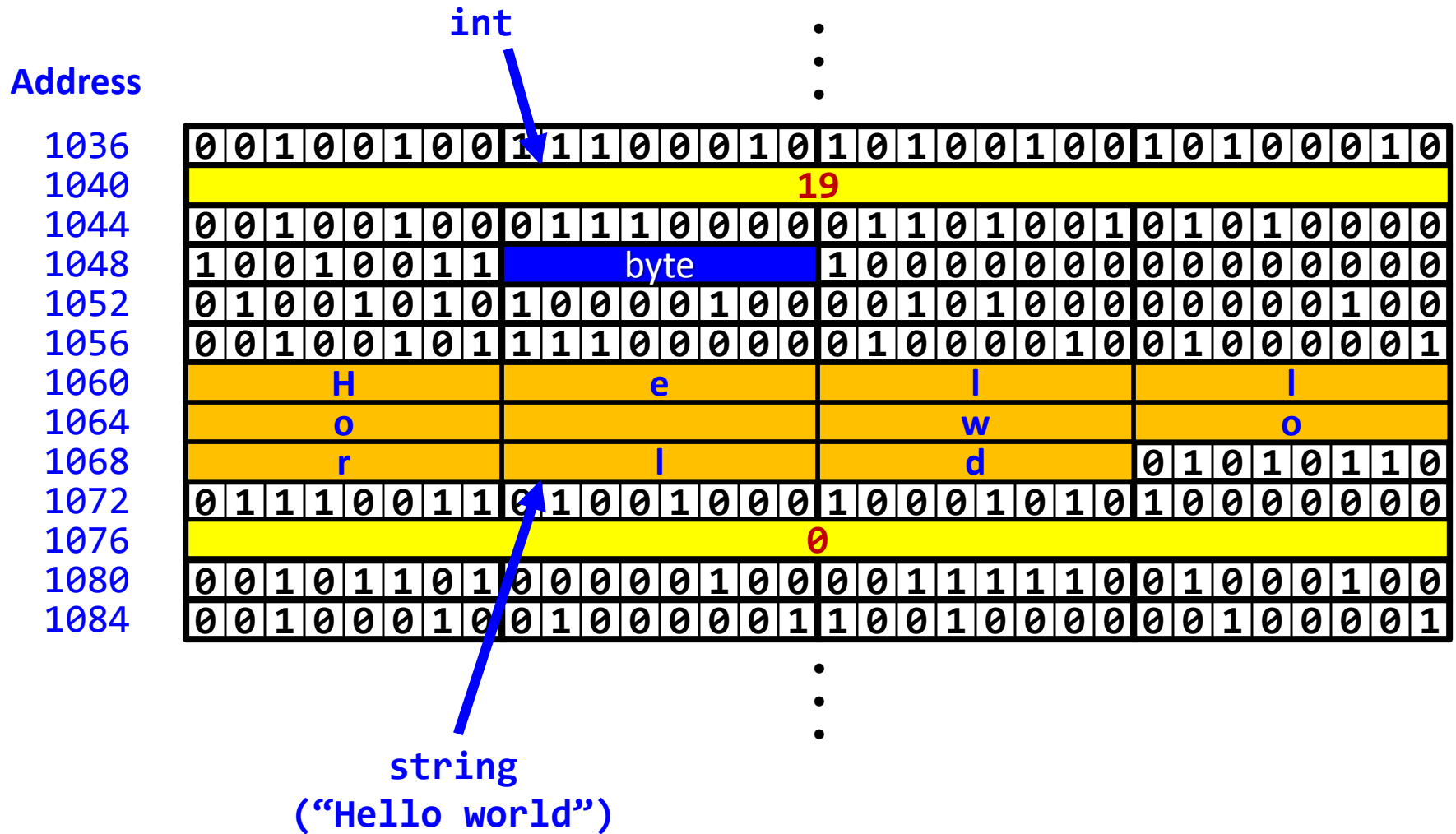


# *Memory management*



Jordi Cortadella and Jordi Petit  
Department of Computer Science

# The memory



# Pointers

- Given a type  $T$ ,  $T^*$  is another type called “pointer to  $T$ ”. It holds the memory address of an object of type  $T$ .
- Examples:

```
int* pi;  
myClass* pC;
```

- Pointers are often used to manage the memory space of dynamic data structures.
- In some cases, it might be convenient to obtain the address of a variable during runtime.

# Pointers

- Address of a variable (reference operator &):

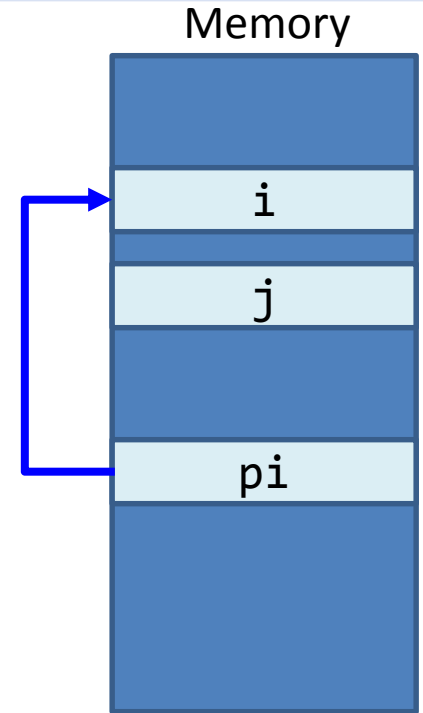
```
int i;  
int* pi = &i;  
// &i means: “the address of i”
```

- Access to the variable (dereference operator \*)

```
int j = *pi;  
// j gets the value of the variable pointed by pi
```

- Null pointer (points to nowhere; useful for initialization)

```
int* pi = nullptr;
```



# Dynamic Object Creation/Destruction

- The *new* operator returns a pointer to a newly created object:

```
myClass* c = new myClass( );  
myClass* c = new myClass{ }; // C++11  
myClass* c = new myClass;
```

- The *delete* operator destroys an object through a pointer (deallocates the memory space associated to the object):

```
delete c; // c must be a pointer
```

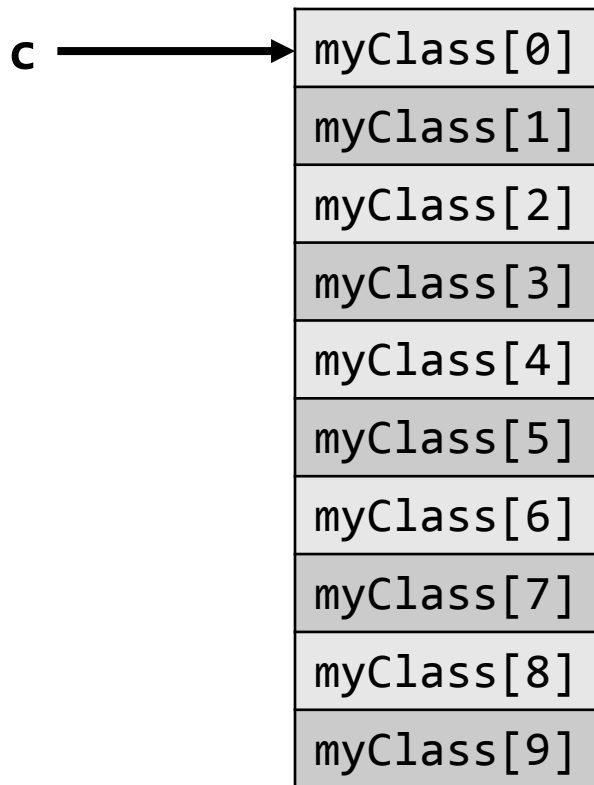
# Access to members through a pointer

The members of a class can be accessed through a pointer to the class via the `->` operator:

```
vector<int>* vp = new vector<int> (100);  
...  
vp->push_back(5);  
...  
int n = vp->size();
```

# Allocating/deallocating arrays

*new* and *delete* can also create/destroy arrays of objects



```
// c is a pointer to an
// array of objects
myClass* c = new myClass[10];

// c[i] refers to one of the
// elements of the array
c[i].some_method_of_myClass();

// deallocating the array
delete [] c;
```

# References

- A reference defines a new name for an existing value (a synonym).
- References are not pointers, although they are usually implemented as pointers (memory references).
- Typical uses:
  - Avoiding copies (e.g., parameter passing)
  - Aliasing long names
  - Range *for* loops



# References: examples

```
auto & r = x[getIndex(a, b)].val;  
...  
r.defineValue(n); // avoids a long name for r  
...  
  
// avoids a copy of a large data structure  
bigVector & V = myMatrix.getRow(i);  
...  
  
// An alternative for the following loop:  
// for (int i =0; i < a.size(); ++i) ++a[i];  
  
for (auto x: arr) ++x;    // does not work (why?)  
  
for (auto & x: arr) ++x; // it works!
```

# Pointers vs. references

- A *pointer* holds the memory address of a variable. A *reference* is an alias (another name) for an existing variable.
- In practice, references are implemented as pointers.
- A pointer can be re-assigned any number of times, whereas a reference can only be assigned at initialization.
- Pointers can be NULL. References always refer to an object.
- You can have pointers to pointers. You cannot have references to references.
- You can use pointer arithmetic (e.g., `&object+3`). Reference arithmetic does not exist.

# The Vector class

(an approximation to the STL vector class)

# The Vector class

- The natural replacement of C-style arrays.
- Main advantages:
  - It can dynamically grow and shrink.
  - It is a *template* class (can handle any type T)
  - No need to take care of the allocated memory.
  - Data is allocated in a contiguous memory area.
- We will implement a **Vector** class, a simplified version of STL's vector.
- Based on Weiss' book (4<sup>th</sup> edition), see Chapter 3.4.

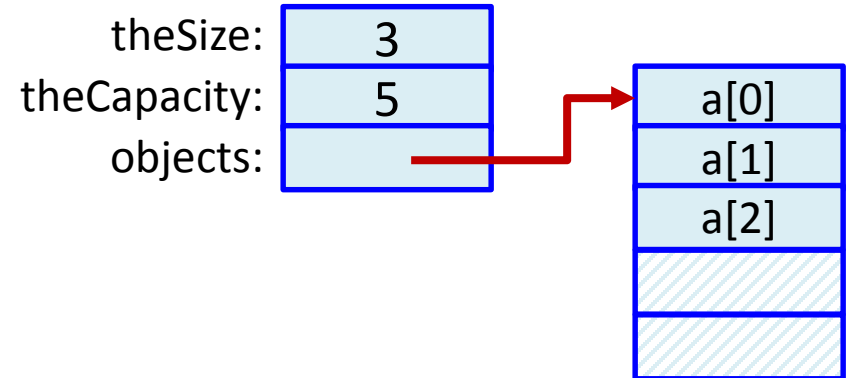
# The Vector class

```
template <typename Object>
class Vector {
    public:
        ...
    private:
        ...
};
```

- What is a class template?
  - A generic abstract class that can handle various datatypes.
  - The template parameters determine the genericity of the class.
- Example of declarations:
  - `Vector<int> V;`
  - `Vector<polygon> Vp;`
  - `Vector<Vector<double>> M;`

# The Vector class

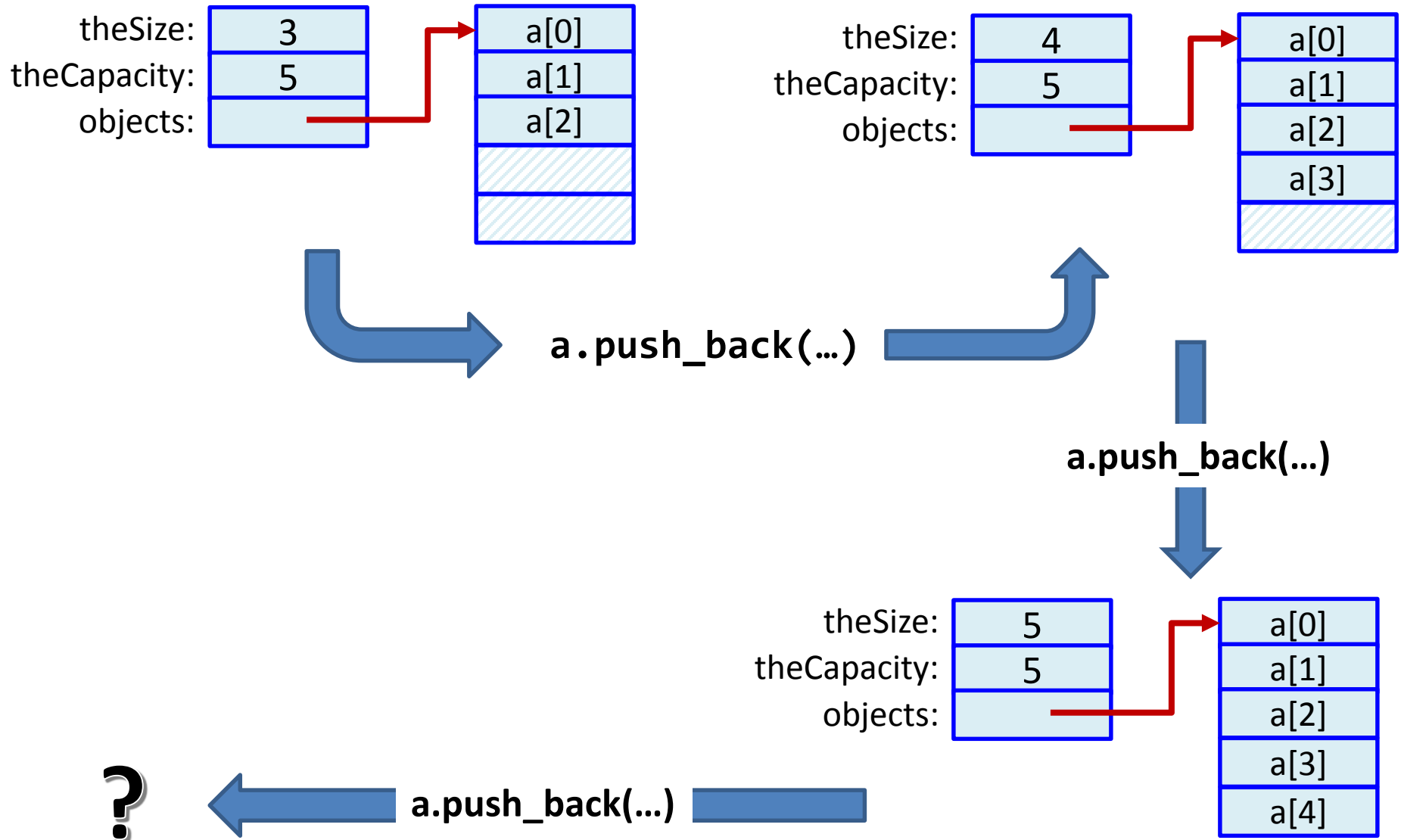
```
template <typename Object>
class Vector {
public:
    ...
private:
    int theSize;
    int theCapacity;
    Object* objects;
};
```



← **a pointer !**

- A Vector may allocate more memory than needed (size vs. capacity).
- The memory must be reallocated when there is no enough capacity in the storage area.
- The pointer stores the base memory address (location of objects[0]). Pointers can be used to allocate/free memory blocks via new/delete operators.

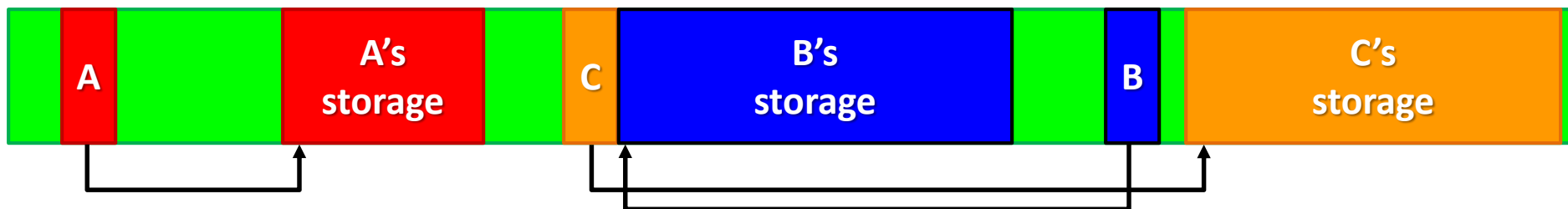
# The Vector class



# Memory management



After resizing B's storage  
(e.g., `B.push_back(...)`)



- Data structures usually have a descriptor (fixed size) and a storage area (variable size). Memory blocks cannot always be resized. They need to be reallocated and initialized with the old block. After that, the old block can be freed.
- Programmers do not have to take care of memory allocation, but it is convenient to know the basics of memory management.
- Beware of memory leaks, fragmentation, ...



# The Vector class

```
public:
```

```
// Returns the size of the vector
```

```
int size() const  
{ return theSize; }
```

```
// Is the vector empty?
```

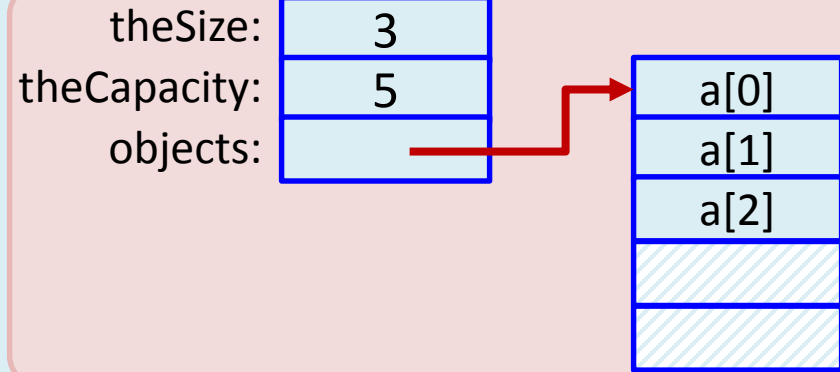
```
bool empty() const  
{ return size() == 0; }
```

```
// Adds an element to the back of the vector
```

```
void push_back(const Object & x) {  
    if (theSize == theCapacity) reserve(2*theCapacity + 1);  
    objects[theSize++] = x;  
}
```

```
// Removes the last element of the array
```

```
void pop_back()  
{ theSize--; }
```



see later

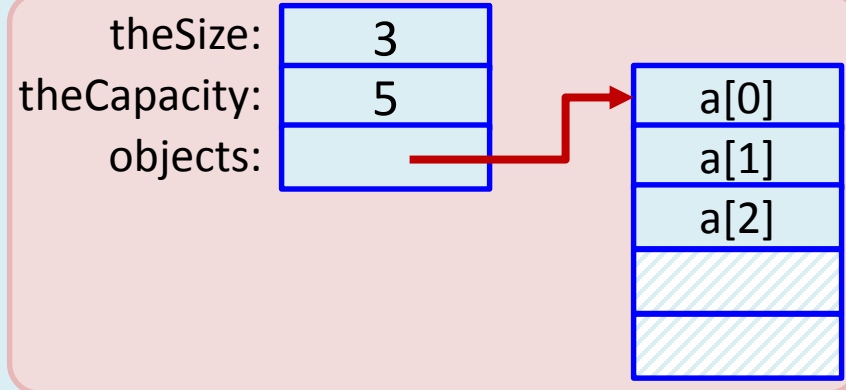
# The Vector class

```
public:
    // Returns a const ref to the last element
    const Object& back() const
    { return objects[theSize - 1]; }

    // Returns a ref to the i-th element
    Object& operator[](int i)
    { return objects[i]; }

    // Returns a const ref to the i-th element
    const Object& operator[](int i) const
    { return objects[i]; }

    // Modifies the size of the vector (destroying
    // elements in case of shrinking)
    void resize(int newSize) {
        if (newSize > theCapacity) reserve(newSize*2);
        theSize = newSize;
    }
```



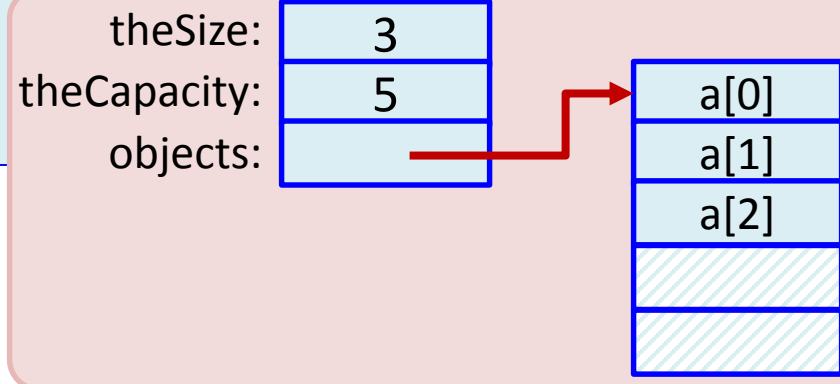
# The Vector class

```
// Reserves space (to increase capacity)
void reserve(int newCapacity) {
    if (newCapacity <= theCapacity) return;

    // Allocate the new memory block
    Object* newArray = new Object[newCapacity];

    // Copy the old memory block
    for (int k = 0; k < theSize; ++k)
        newArray[k] = objects[k];

    theCapacity = newCapacity;
    // Swap pointers and free old memory block
    std::swap(objects, newArray);
    delete [] newArray;
}
```



# Constructors, copies, assignments

```
MyClass a, b; // Constructor is used
```

```
MyClass c = a; // Copy constructor is used
```

```
b = c; // Assignment operator is used
```

```
// Copy constructor is used when passing  
// the argument to a function (or returning  
// the value from a function)
```

```
void foo(Myclass x);
```

```
...
```

```
foo(c); // Copy constructor is used
```

# The Vector class

```
// Default constructor with initial size
```

```
Vector(int initSize = 0) : theSize{initSize},  
    theCapacity{initSize + SPARE_CAPACITY}  
{ objects = new Object[theCapacity]; }
```

```
// Copy constructor
```

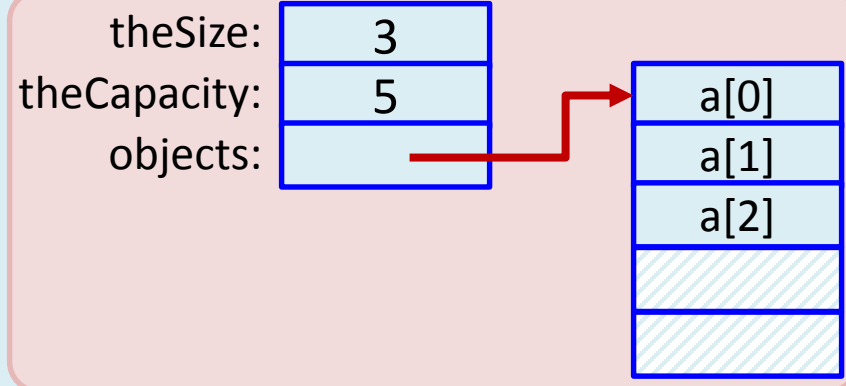
```
Vector(const Vector& rhs) : theSize{rhs.theSize},  
    theCapacity{rhs.theCapacity}, objects{nullptr} {  
    objects = new Object[theCapacity];  
    for (int k = 0; k < theSize; ++k) objects[k] = rhs.object[k];  
}
```

```
// Assignment operator
```

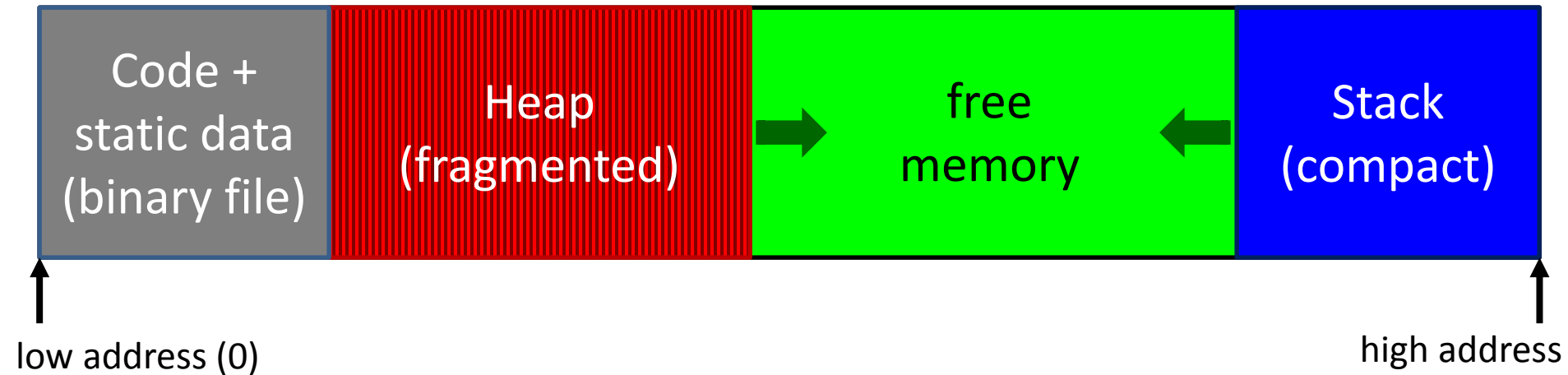
```
Vector& operator=(const Vector& rhs) {  
    if (this != &rhs) { // Avoid unnecessary copy  
        theSize = rhs.theSize;  
        theCapacity = rhs.theCapacity;  
        delete [] objects;  
        objects = new Object[theCapacity];  
        for (int k = 0; k < theSize; ++k) objects[k] = rhs.object[k];  
    }  
    return *this;  
}
```

```
// Destructor
```

```
~Vector() { delete [] objects; }
```



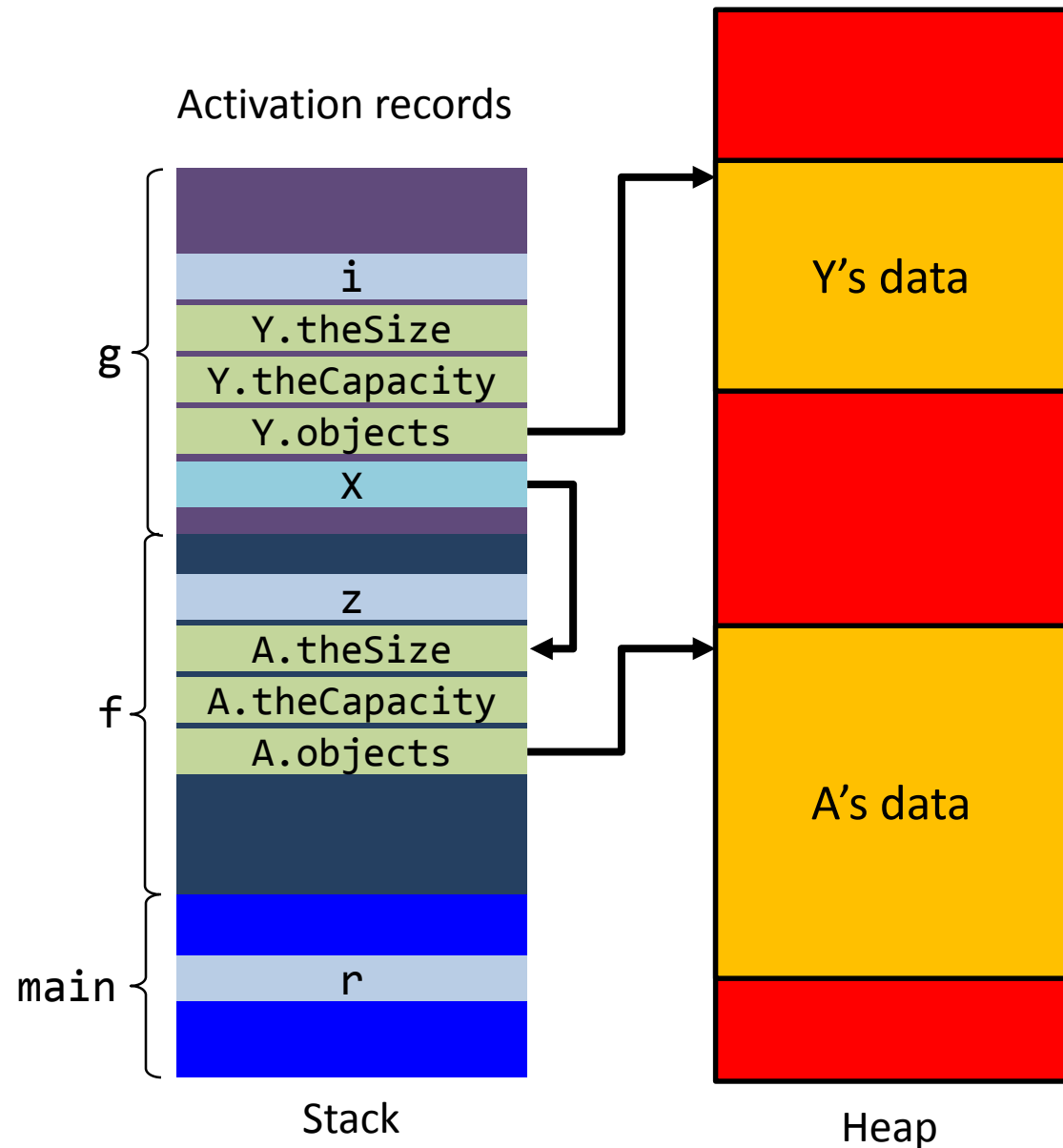
# Memory layout of a program



Region	Type of data	Lifetime
Static	Global data	Lifetime of the program
Stack	Local variables of a function	Lifetime of the function
Heap	Dynamic data	Since created ( <i>new</i> ) until destroyed ( <i>delete</i> )

# Memory layout of a program

```
int g(vector<int>& X) {  
    int i;  
    vector<double> Y;  
    ...  
}  
  
void f() {  
    int z;  
    vector<int> A;  
    ...  
    z = g(A);  
    ...  
}  
  
int main() {  
    double r;  
    ...  
    f();  
    ...  
}
```



# Memory management models

- **Programmer-controlled management:**
  - The programmer decides when to allocate (new) and deallocate (delete) blocks of memory.
  - Example: C++.
  - Pros: efficiency, memory management can be optimized.
  - Cons: error-prone (dangling references and memory leaks)
- **Automatic management:**
  - The program periodically launches a garbage collector that frees all non-referenced blocks of memory.
  - Examples: Java, python, R.
  - Pros: the programmer does not need to worry about memory management.
  - Cons: cannot optimize memory management, less control over runtime.



# Dangling references and memory leaks

```
myClass* A = new myClass;  
myClass* B = new myClass;  
// We have allocated space for two objects  
  
A = B;  
// A and B point at the same object!  
// Possible memory leak (unreferenced memory space)  
  
delete A;  
// Now B is a dangling reference  
// (points at free space)  
  
delete B; // Error
```

# Pointers and memory leaks

```
for (i = 0; i < 1000000; ++i) {  
    myClass* A = new MyClass; // 1Kbyte  
    ...  
    do_something(A, i);  
    ...  
    // forgets to delete A  
}  
// This loop produces a 1-Gbyte memory leak!
```

## Recommendations:

- Do not use pointers, unless you are very desperate.
- If you have to use pointers, hide their usage inside a class and define consistent constructors/destructors.
- Use *valgrind* ([valgrind.org](http://valgrind.org)) to detect memory leaks.
- Remember: no pointers → no memory leaks.

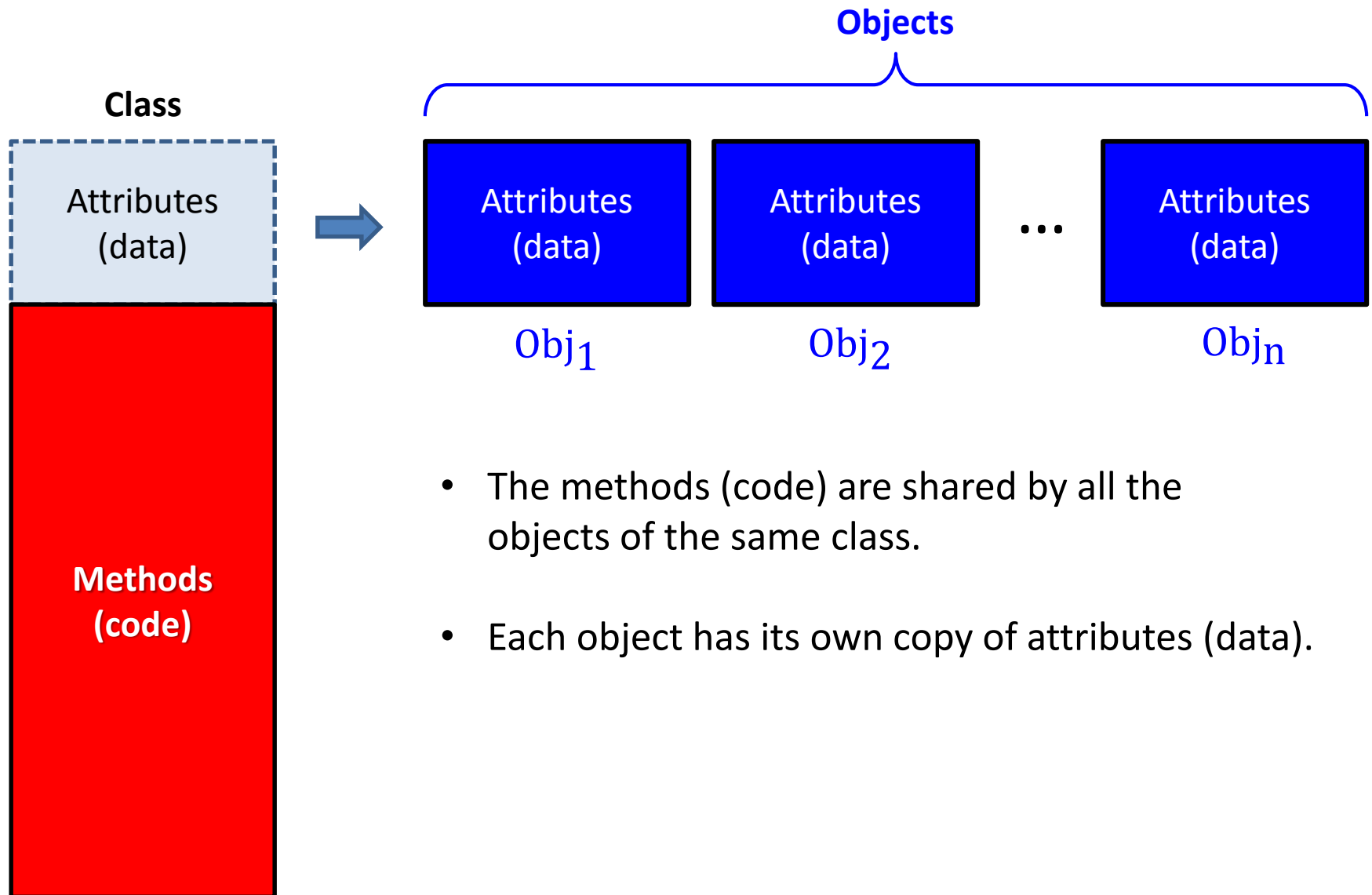
# Pointers and references to dynamic data

```
vector<myClass> a;  
...  
// do some push_back's to a  
...  
const myClass& x = a[0]; // ref to a[0]  
// x contains a memory address pointing at a[0]  
  
a.push_back(something);  
// the address of a[0] might have changed!  
// x might be pointing at garbage data
```

**Recommendation:** don't trust on pointers/references to dynamic data.

**Possible solution:** use indices/keys to access data in containers.

# How much memory space does an object take?



- The methods (code) are shared by all the objects of the same class.
- Each object has its own copy of attributes (data).

# About C (the predecessor of C++)

- Developed by Dennis Ritchie at Bell Labs (1972) and used to re-implement Unix.
- It was designed to be easily mappable to machine instructions and provide low-level memory access.
- Today, it is still necessary to use some C libraries, designed by skilled experts, that have not been rewritten in other languages (the same happens with some FORTRAN libraries).
- Some aspects that must be known to interface with C libraries:
  - No references (only pointers).
  - No object-oriented support (no STL, no vectors, no maps, etc).
  - Vectors must be implemented as ***arrays***.

# C: parameters by reference

```
// a is received by value and b by reference
// (using a pointer)
int f(int a, int* b) {
    *b = *b + a;
    return a*a;
}

int main() {
    int x, y, z;
    ...
    // pointers used to pass by reference
    z = f(x, &y);
    ...
}
```

# C: arrays

```
int a[100];           // an array of 100 int's
double m[30][40];    // a matrix of size 30x40
int c[]; // also int* c: an array of unknown size
char* s; // a string (array of unknown size)
```

```
...
// memory allocation for n integers
c = malloc(n*sizeof(int));
```

```
// memory allocation for m chars
s = malloc(m*sizeof(char));
```

```
...
// Anything that is allocated must be freed
// (or memory leaks will occur)
free(c);
free(s);
```

# C: arrays

```
// Equivalent declaration: double sum(double* v, int n)
// The size of the array must be indicated explicitly.
```

```
double sum(double v[], int n) {
    double s = 0;
    for (int i = 0; i < n; ++i) s += v[i];
    return s;
}
```

```
// Or we can use a sentinel as a terminator of an array.
// Example: strings are usually terminated by a 0.
```

```
int length(char* s) {
    int len;
    for (len = 0; s[len] != 0; ++len);
    return len;
}
```

```
int main() {
    char hello[] = "Hello, world!";
    // arrays are implicitly passed by reference
    int l = length(hello);
    ...
}
```




# C arrays and C++ vectors

```
double sum(double v[], int n) {  
    double s = 0;  
    for (int i = 0; i < n; ++i) s += v[i];  
    return s;  
}
```

// Any C++ vector is an object that contains  
// a private array. This array can be accessed  
// using the 'data()' method (C++11).

```
int main() {  
    vector<double> a;  
    ...  
    double s = sum(a.data(), a.size());  
}
```



The internal array  
of the vector

Functions using the internal array of a vector should NEVER resize the array !

# Summary

---

- Memory is a valuable resource in computing devices. It must be used efficiently.
- Languages are designed to make memory management transparent to the user, but a lot of inefficiencies may arise unintentionally (e.g., copy by value).
- Pointers imply the use of the heap and all the problems associated to memory management (memory leaks, fragmentation).
- Recommendation: do not use pointers unless you have no other choice. Not using pointers will save a lot of debugging time.
- In case of using pointers, try to hide the pointer manipulation and memory management (new/delete) inside the class in such a way that the user of the class does not need to “see” the pointers.

# EXERCISES

# Constructors

Consider two versions of the program below, each one using a different definition of the class Point. Comment on the behavior of the program at compile time and runtime.

```
class Point {  
    int x, y;  
public:  
    Point(const Point &p) {  
        x = p.x; y = p.y;  
    }  
    int getX() { return x; }  
    int getY() { return y; }  
};
```

```
class Point {  
    int x, y;  
public:  
    Point(int i=0, int j=0) {  
        x = i; y = j;  
    }  
    int getX() { return x; }  
    int getY() { return y; }  
};
```

```
int main() {  
    Point p1(10);  
    Point p2 = p1;  
    cout << "x = " << p2.getX() << endl;  
    cout << "y = " << p2.getY() << endl;  
}
```

# Constructors and destructors

```
int c = 0; // Global variable

class A {
    int id;
public:
    A() : id(++c) { cout << "N"; }

    A(const A& x) {
        id = x.id; cout << "C";
    }

    A& operator=(const A& x) {
        id = x.id; cout << "A";
    }

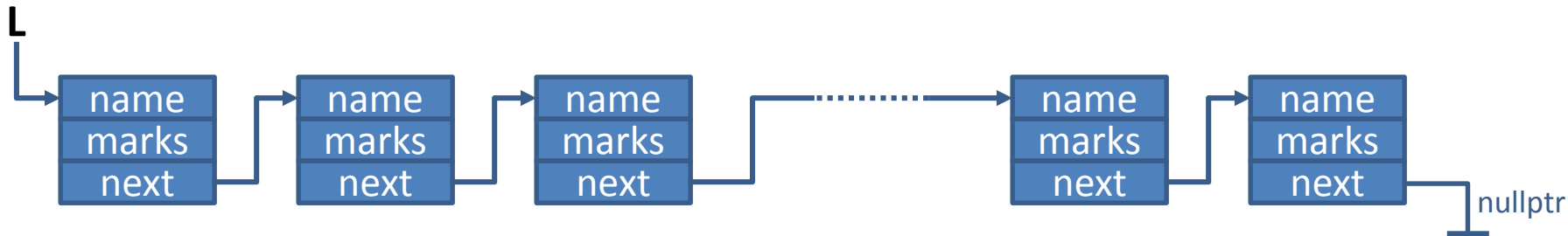
    ~A() { cout << id; }
};
```

What is the output of this program?  
Explain why.

```
void f(const A& x, A y) {
    A z = x;
    A w;
}

int main() {
    A v1, *v2, v3;
    A v4 = v3;
    v2 = new A();
    v1 = *v2;
    f(v1, v4);
    delete v2;
    A v5;
}
```

# List with pointers



```
struct Student {  
    string name;  
    vector<double> marks;  
    Student* next;  
};  
  
string BestStudent(Student* L);
```

Consider the following definition of a list of students organized as shown in the picture.

You can assume that the vector of marks is never empty.

The last student in the list points at “null” (nullptr).

Design the function **BestStudent** with the following specification:

**L** points at the first student of the list. **BestStudent** returns the name of the student with the best average mark. In case no student has an average mark greater than or equal to 5, the function must return the string *“Bad Teacher”*.