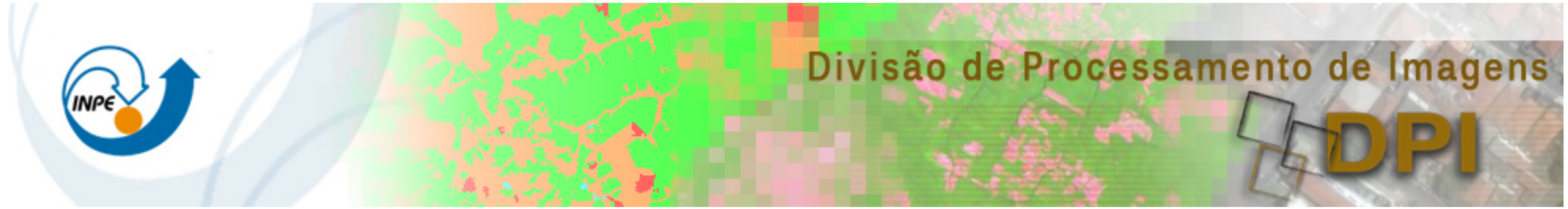
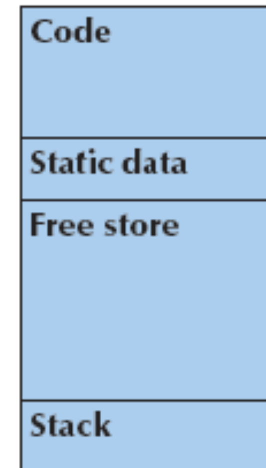



# CAP- 390-1 Fundamentos de Programação Estruturada – Aula 6

Lubia Vinhas

# The computer's memory



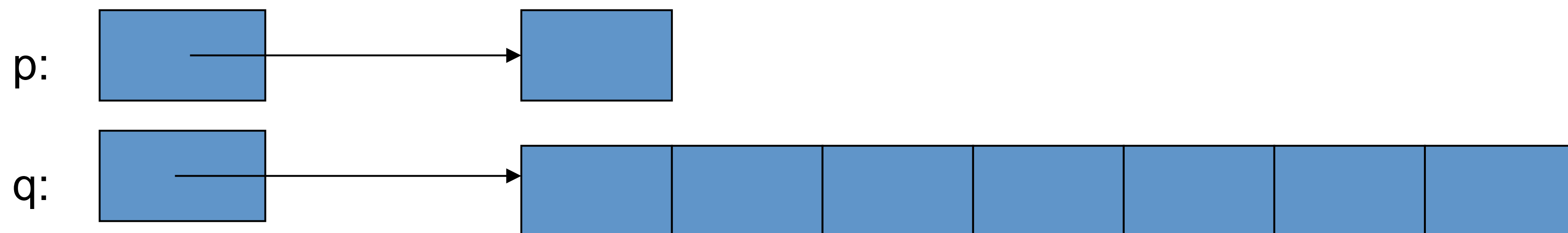

- As a program sees it
  - Local variables "live on the stack"
  - Global variables are "static data"
  - The executable code is in "the code section"

- In C++ the heap is made available through the operator new

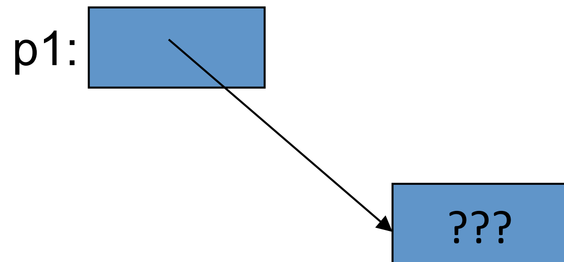
```
int* pi = new int;              // allocates one int
int* qi = new int[4];           // allocates 4 ints
double* pd = new double;
double* qd = new double[4];
```

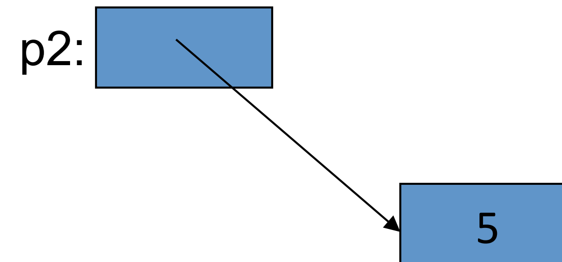
# The free store

- You request memory "to be allocated" "on the free store" by the new operator
  - The new operator returns a pointer to the allocated memory
  - A pointer is the address of the first byte of the memory
  - For example
    - int* p = new int;
    - int* q = new int[7];
  - A pointer points to an object of its specified type
  - A pointer does not know how many elements it points to



# Access




- Individual elements

```
int* p1 = new int;        // get (allocate) a new uninitialized int
int* p2 = new int(5);     // get a new int initialized to 5

int x = *p2;              // get/read the value pointed to by p2
                          // (or "get the contents of what p2 points to")
                          // in this case, the integer 5
int y = *p1;              // undefined: y gets an undefined value; don't do that
```

# Why use free store?

- To allocate objects that have to outlive the function that creates them:

```
double* make(int n)      // allocate n ints
{
    return new double[n];
}
```

# A problem: memory leak

```
double* calc(int result_size, int max)
{
	double* p = new double[max];	// allocate another max doubles
	double* result = new double[result_size];
	// … use p to calculate results to be put in result …
	return result;
}
double* r = calc(200,100);  // oops! We "forgot" to give the memory
                            // allocated for p back to the free store
```

- Lack of de-allocation ("memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

# A problem: memory leak

```
double* calc(int result_size, int max)
{
    int* p = new double[max];       // allocate another max doubles
    double* result = new double[result_size];

    // … use p to calculate results to be put in result …
    delete[ ] p;                    // de-allocate (free) that array
                                    // i.e., give the array back to the free store
    return result;
}

double* r = calc(200,100);
// use r
delete[ ] r;                        // easy to forget
```

# Memory leaks

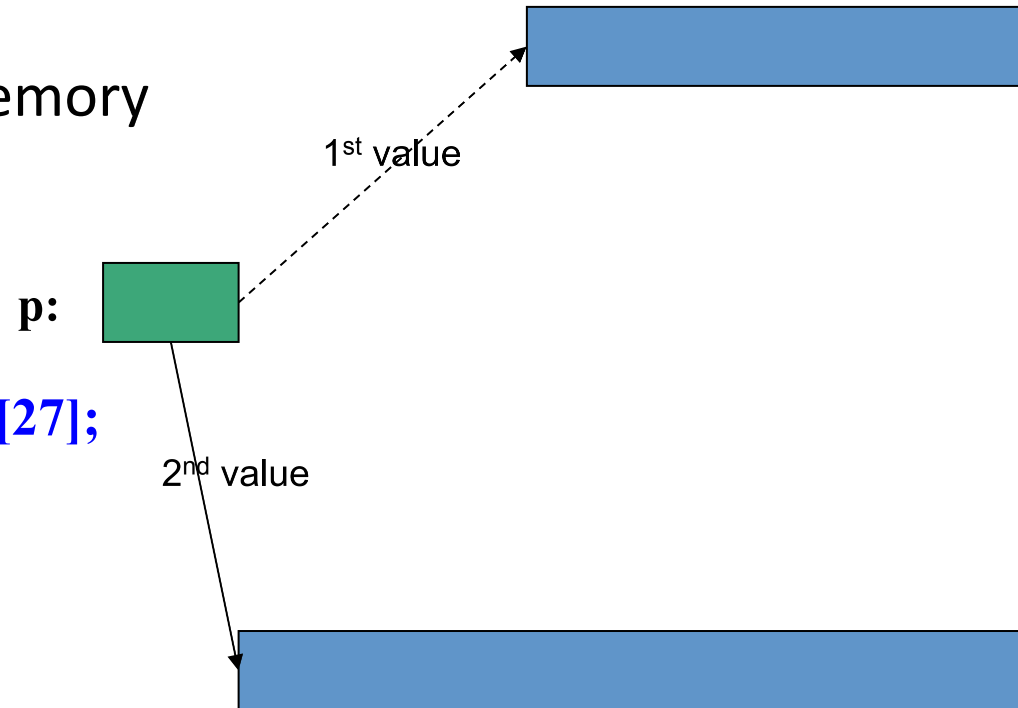
- A program that needs to run “forever” can’t afford any memory leaks
  - An operating system is an example of a program that “runs forever”
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - i.e., memory leaks aren’t “good/bad” but they can be a major problem in specific circumstances

# Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}
```

*// 1st array (of 27 doubles) leaked*



# Free store summary

- Allocate using new: allocates an object on the free store, sometimes initializes it, and returns a pointer to it

```
int* pi = new int;            // default initialization (none for int)
char* pc = new char('a');     // explicit initialization
double* pd = new double[10];  // allocation of (uninitialized) array
```

- Delete and delete[ ] return the memory of an object allocated by new to the free store so that the free store can use it for new allocations

```
delete pi; // deallocate an individual object
delete pc; // deallocate an individual object
delete[ ] pd;    // deallocate an array
```

# void*

- void* means "pointer to some memory that the compiler doesn't know the type of"
- We use void* when we want to transmit an address between pieces of code that really don't know each other's types – so the programmer has to know
  - Example: the arguments of a callback function
- There are no objects of type void

```
void v;     // error
void f();   // f() returns nothing – f() does not return an object of type void
```

- Any pointer to object can be assigned to a void*

```
int* pi = new int;
double* pd = new double[10];
void* pv1 = pi;
void* pv2 = pd;
```

# Pointers and references

- Think of a reference as an automatically dereferenced pointer
  - Or as "an alternative name for an object"
  - A reference must be initialized
  - The value of a reference cannot be changed after initialization

```
int x = 7;
int y = 8;
int* p = &x;   *p = 9;
p = &y;   // ok
int& r = x;      x = 10;
r = &y;   // error (and so is all other attempts to change what r refers to)
```

# Vector

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access
- How is that done?
  - That is, how is vector implemented?

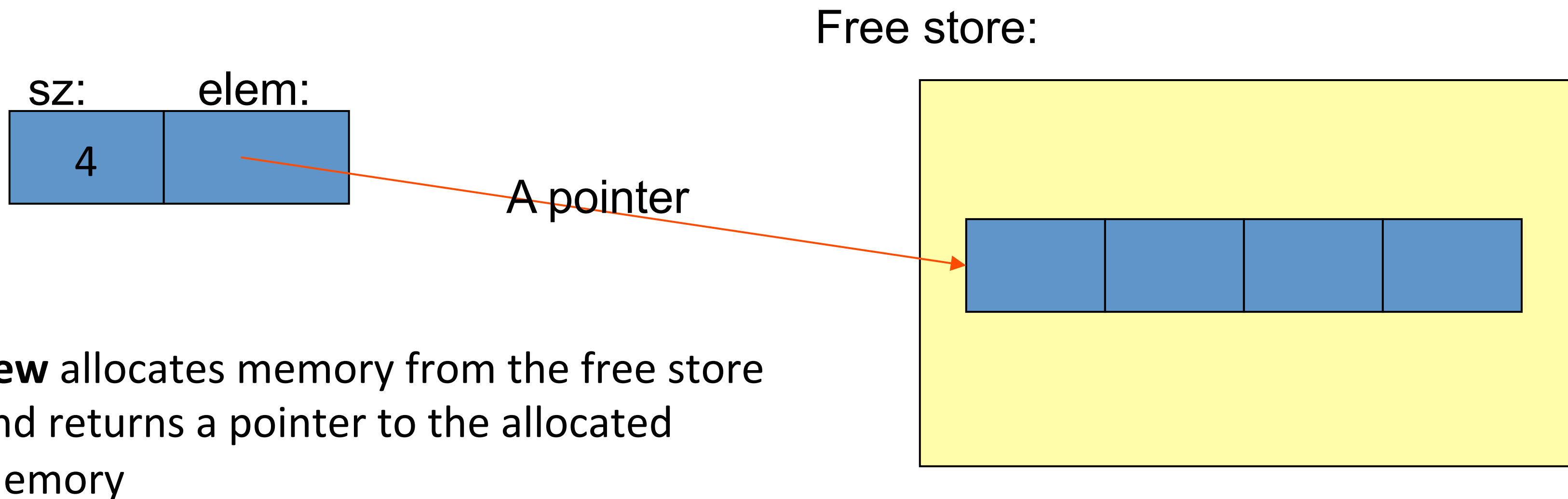
# Vector

```
// a very simplified vector of doubles (like vector<double>):

class vector {
	int sz;			// the number of elements ("the size")
	double* elem;		// pointer to the first element
public:
	vector(int s);		// constructor: allocate s elements,
				// let elem point to them,
				// store s in sz
	int size() const { return sz; }// the current size
};
```

# Vector (constructor)

```
vector::vector(int s)       // vector's constructor
    :sz(s),                 // store the size s in sz
     elem(new double[s])    // allocate s doubles on the free store
                // store a pointer to those doubles in elem
{
}
// Note: new does not initialize elements (but the standard vector does)
```

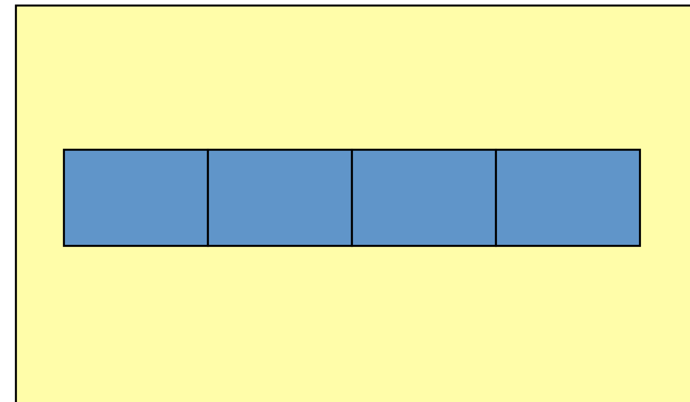



**new** allocates memory from the free store and returns a pointer to the allocated memory

# Vector (destructor)

```
vector::~vector(int s)
{
    delete[] elem;
}
```

Free store:



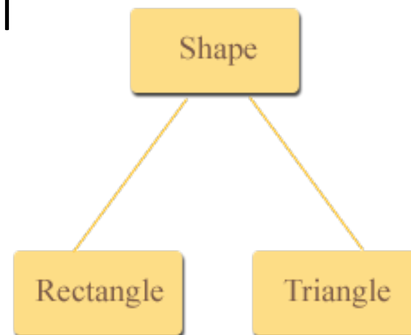
**delete de**allocates memory from the free store and it can be used anymore

# Practice

- Program the vector class in a way that shows the free memory manipulation

# Inheritance

- **Derivation**: a way to build one class form another so that the new class can be used in a place of the original




- **Virtual functions**: the ability to define a function in a base class and have a function of the same name and type in a derived class called when a user calls the basic class function. Run-time **polimorphism**
- Private and protected members: implementation details of our classes private or protect from the direct use. **Encapsulation**
- **Object-oriented programming**

```
class Shape
{
protected:
    float width, height;
public:
    void set_data (float a, float b)
    {
        width = a;
        height = b;
    }
};
```

```
int main ()
{
    Rectangle rect;
    Triangle tri;
    rect.set_data (5,3);
    tri.set_data (2,5);
    cout << rect.area() << endl;
    cout << tri.area() << endl;
    return 0;
}
```

```
class Rectangle: public Shape
{
public:
    float area ()
    {
        return (width * height);
    }
};

class Triangle: public Shape
{
public:
    float area ()
    {
        return (width * height / 2);
    }
};
```

```
class Rectangle
{
private :
        float length;
        float width;
public:
        Rectangle ()
        {
                length = 0;
                width = 0;
        }

        Rectangle (float len, float wid)
        {
                length = len;
                width = wid;
        }

        float area()
        {
                return length * width ;
        }
};
```

```
class Box : public Rectangle
{
private :
        float height;
public:
        Box ()
        {
                height = 0;
        }

        Box (float len, float wid, float ht) :
                Rectangle(len, wid)
        {
                height = ht;
        }

        float volume()
        {
                return area() * height;
        }
};
```

```
int main ()
{
	Box bx;
	Box cx(4,8,5);
	cout << bx.volume() << endl;
	cout << cx.volume() << endl;
	return 0;
}
```

```
class mother
{
public:
        void display ()
        {
                cout << "mother: display function\n";
        }
};

class daughter : public mother
{
public:
        void display ()
        {
                cout << "daughter: display function\n\n";
                mother::display();
        }
};

int main ()
{
        daughter rita;
        rita.display();
        return 0;
}
```

```
class Shape
{
  protected:
    double width, height;
  public:
    void set_data (double a, double b)
    {
        width = a;
        height = b;
    }
    virtual double area()
    { return 0;}
};

class Rectangle: public Shape
{
  public:
    double area ()
    {
        return (width * height);
    }
};
```