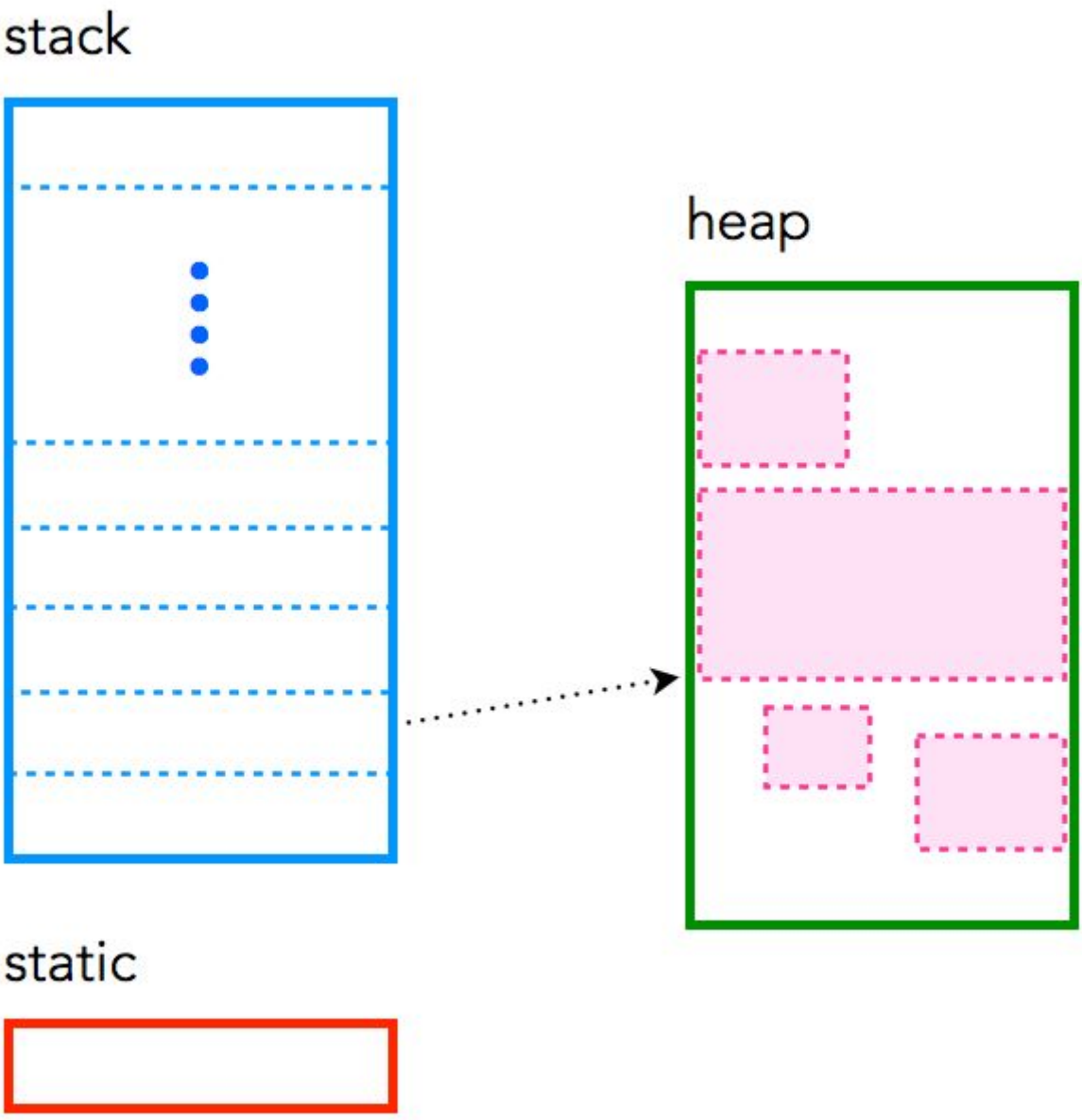
Memory in C++ – the stack, the heap, and static

C++ has three different pools of memory.

- **static**: global variable storage, permanent for the entire run of the program.
- **stack**: local variable storage (automatic, continuous memory).
- **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).

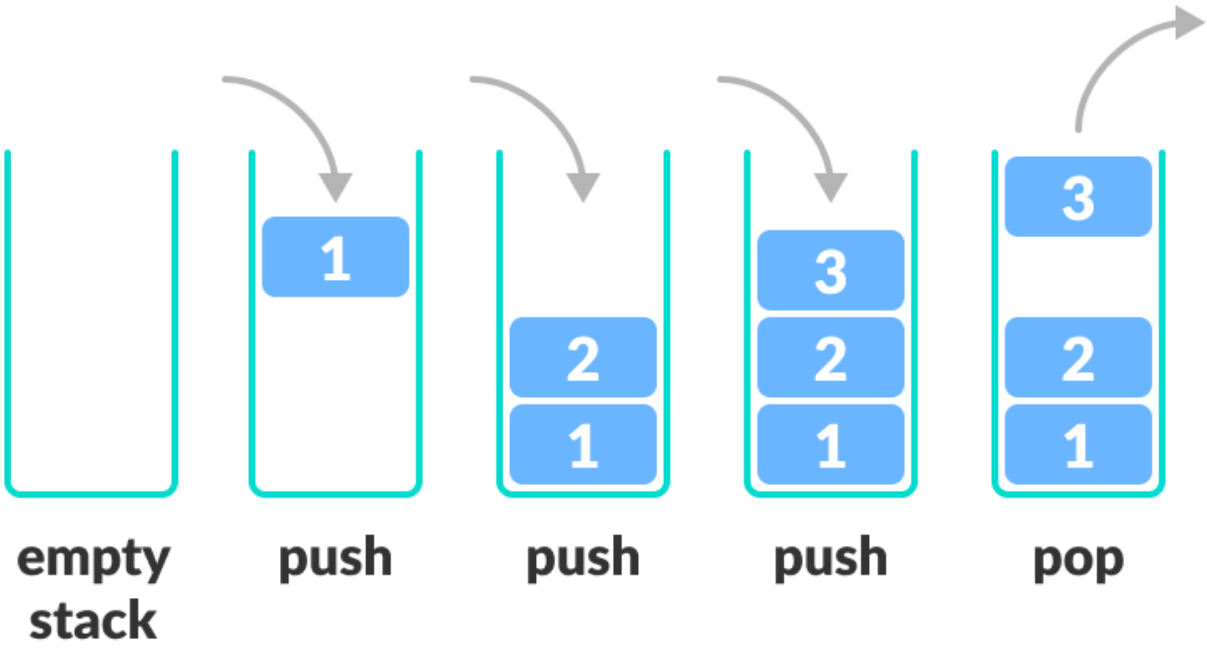


Static memory

Static memory persists throughout the entire life of the program, and is usually used to store things like global variables, or variables created with the `static` clause. Their address is fixed and cannot be changed.

Stack memory

The stack is used to store variables used on the inside of a function (including the `main()`` function). It's a LIFO, "Last-In,-First-Out", structure. Every time a function declares a new variable it is "pushed" onto the stack. Then when a function finishes running, all the variables associated with that function on the stack are deleted, and the memory they use is freed up. This leads to the "local" scope of function variables. The stack is a special region of memory, and automatically managed by the CPU – so you don't have to allocate or deallocate memory. Stack memory is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame.



Note that there is generally a limit on the size of the stack – which can vary with the operating system. If a program tries to put too much information on the stack, **stack overflow** will occur. Stack overflow happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory. Stack overflow also occurs in situations where recursion is incorrectly used or is too deep.

Heap memory

The heap is the diametrical opposite of the stack. The heap is a large pool of memory that can be used dynamically – it is also known as the "free store". This is memory that is not automatically managed – you have to explicitly allocate (`new`), and deallocate (`free`) the memory. Failure to free the memory when you are finished with it will result in what is known as a memory leak – memory that is still "being used", and not available to other processes. This is why you **SHOULD ALWAYS PREFER** using the `std` library to automatically manage memory, that is use managed data structures `std::vector`, `std::map`, `std::set`... and smart pointers `std::shared_ptr`, `std::unique_ptr`. Unlike the stack, there are generally no restrictions on the size of the heap (or the variables it creates), other than the physical size of memory in the machine. Variables created on the heap are accessible anywhere in the program.

An example of memory use

Consider the following example of a program containing all three forms of memory:

```
#include <iostream>
#include <vector>
#include <memory>

struct Counter {
<<<<<<< HEAD
    Counter() { ++count; }
    Counter(const Counter&) { ++count; }
    ~Counter() { --count; }
    static size_t howMany() { return count; } // why you are using static
here? recover!
    static size_t count;
=======
  Counter() { ++count; }
  Counter(const Counter&) { ++count; }
  ~Counter() { --count; }
  static size_t count;
>>>>>>> 5d437a15db6426646f5b3a5d3591c4caa317ba77
};

size_t Counter::count = 0;

void a_function() {
  int z;
  std::cout << "Stack memory (z): " << &z << std::endl;
}

int main() {
    std::cout << "Static memory: " << &Counter::count << std::endl;
    int y;
    int a[3];
    Counter c;

    std::cout << "Stack memory (y): " << &y << std::endl;
    std::cout << "Stack memory (a): " << &a << std::endl;
    std::cout << "Stack memory (c): " << &c << std::endl;
    a_function();

    std::vector<int> v(3);
    auto pt = std::make_shared<Counter>();
    std::cout << "Heap memory (v after init): " << v.data() << std::endl;
    std::cout << "Heap memory (pt): " << pt.get() << std::endl;
    for(int i = 0; i < 1000000; i++)
      v.push_back(0);
    std::cout << "Heap memory (v after push_back): " << v.data() <<
std::endl;
```

```
    return 0;
}
```

The variable `count` is static storage, because of its keyword `static`. Its function is to count how many instances of the class `Counter` have been created. Both `y` and `a` are dynamic stack storage which is deallocated when the program ends. Indeed, notice that arrays which size is knows at compile time are allocated on the stack. Behind the hood, the function `new` is used to allocate the memory for `std::vector` and `std::shared_ptr`, the `delete` is handled by the destructor of these classes.

Exercise 1 - Polymorphism

Assignments

1. Implement an abstract class `Shape`, that has a name and a pure virtual function `getArea` to compute its area
2. Implement the concrete children `Circle` and `Rectangle` that receive as constructor parameters the values needed to compute the area (e.g. radius, basis, height...) and override the pure virtual `getArea`.
3. Instantiate a vector of `Shape`s using `shared_ptr` and print the area of each shape

Comments

The huge advantage of this design pattern is that we can iterate seamlessly through the vector, without knowing which is the actual shape. Indeed, thanks to polymorphism, each shape behaves exactly as it should by resolving the correct method to call at run-time.

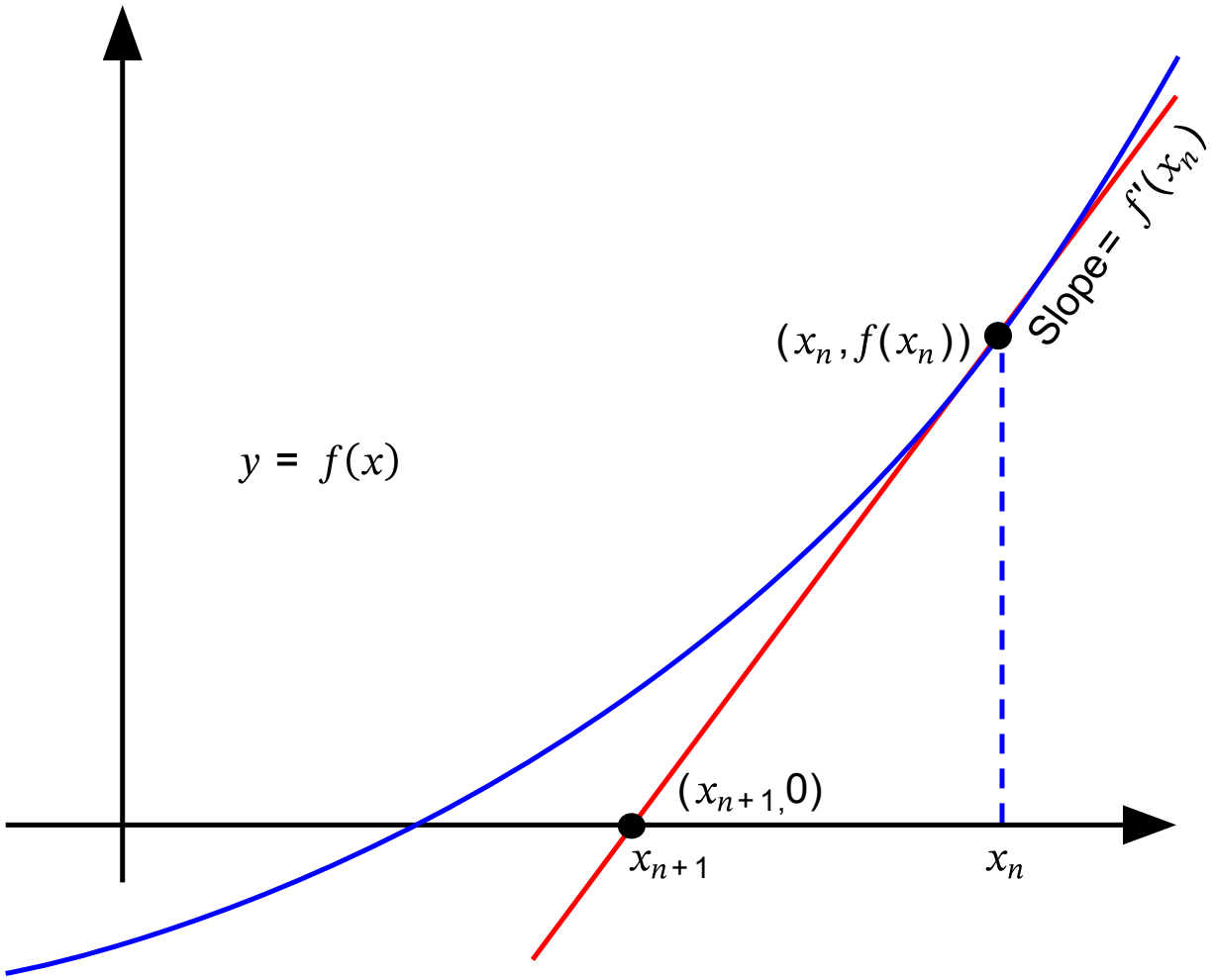
However, this comes with two drawbacks:

1. The run-time cost of resolving the virtual table. That is, the cost of checking which kind of `Shape` an element of the vector is and, by using a table (map), finding the pointer to the correct method.
2. Low spatial locality of memory. That is, each element of the vector is allocated independetly from the others on the heap, this means that elements that are close in the vector may be have large difference in memory addresses (they are physically distant in the RAM). This hurt performances because CPUs like to access memory that is physically in sequence.

In the solution you will find two slightly different implementations of the method `getName`. They are both correct, the difference is that one is more flexible and the other is more memory efficient. Often when coding you face trade-off like this one; depending on the problem at hand you should be able to identify the pros&cons of each approach and choose the most suitable one for your needs.

Exercise 2 - Newton's method

Newton's method is a numerical algorithm used to find roots (zeros) of a function $f(x)$ iteratively (that is find the root α such that $f(\alpha)=0$). The idea is that given an initial guess x_{0}, we approximate $f(x_0)$ with its derivative. We can then find where this line intersects with the x-axis, and this gives us a new estimate x_1.



If the function $f(x)$ satisfies sufficient regularity assumptions and the initial guess x_0 is close enough to the root α, then it can be proved that the iterations defined by

$$x_{n+1}=x_{n}-{\frac {f(x_{n})}{f'(x_{n})}}$$

convege quadratically to α.

How do we stop the iterations?

1. Residual criteria: check that $|f(x_n)|$ is smaller than a given tolerance
2. Step size criteria: check that $|x_n - x_{n-1}|$ is smaller than a given tolerance

Assignments

Write a `NewtonSolver` class that takes as constructor arguments

- the function $f(x)$
- its derivative $f'(x)$
- the maximum number of iterations `max_iter`
- the tolerance on the residual `rtol`
- the tolerance on the step size `stol`

Has a method `solve` that takes as input the starting point x_0, and has the following members (accessible with a getter):

- `iter` number of iterations
- `res` residual
- `xs` the history of the iterations

Apply it to the function $f(x) = x^2 - 2$. Pass it as:

- a function pointer
- a lambda

Homeworks

1. **Easy.** Write a `NewtonParams` strunct that wraps all the values taken as input by the `NewtonSolver` constructor in one place. Make sure that it has reasonable default values for `rtol` and `stol`. Update `NewtonSolver` so that it may be constructed from a `NewtonParams` struct.
2. **Intemidiate.** Write a function `finite_difference` that given as input a function a step size `h` and a point `x`, approximates the derivative of the function in x using the finite difference formula $f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$. Make suitable adjustments to the `NewtonSolver` class so that you can choose either to provide f' as an `std::function` or use the approximation given by `finite_difference`.