C++ objects. Basic usage

Vitalii Drogan

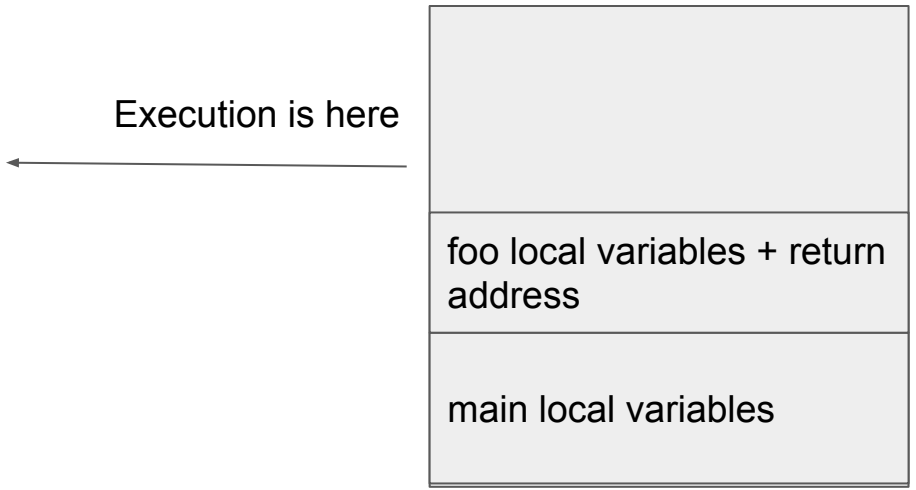
Stack & Heap. Stack.

Allocated an startup of program.(~8mb on linux with glibc)

Used to store function local variables, return address, arguments.

Deallocated when function returns.

```
void foo(){
  int j; // i am allocated on stack.
         //Will be gone after foo returns!
}
int main(){
  int i; // i am allocated on stack
  foo();
  return 0;
}
```



Stack & Heap. Heap.

Bounded by RAM.

Used to store everything big.

Should be deallocated manually.

```
int main(){
  int* p_i = new int; //p_i points to allocated object on heap
  delete p_i; //this call will return memory to OS

  int* array_i = new int[14];
  delete [ ] array_i; //NOTE, delete [] to deallocate arrays.

  return 0;
}
```

Stack & Heap. Bonus.

“Hello world!” - where does our program store this string?

```
std::cout << "Hello world!" << std::endl;
```

In binary itself! (static memory - allocated on program startup)

Proof: in linux run

$ strings a.out

to see all strings stored in binary

Structs. Struct data.

Combine data. Better API. (with no overhead)

Type checking.(only point could go inside Dist function)

```
struct Point{ // SIZE(POINT) = SIZE(DOUBLE)*2
  double x,y;
};

double Distance(const Point& p1, const Point& p2){
  double distX = p1.x - p2.x;
  double distY = p1.y - p2.y;
  return distX*distX + distY*distY;
}
```

Structs. Member functions.

Member function has access to struct data members.

```
struct Vector2{
  double x,y;
  double Length();
};

double Vector2::Length2(){
  return x*x + y*y;
}
```

```
// Usage
Vector2 vec;
//....
double len2 = vec.Length2();
// OR
Vector2 * pVec;
//....
double len2 = pVec->Length2();
```

Structs. Construction.

Struct is “interpretation” of bytes somewhere in memory.

When bytes are allocated(on heap or stack) they are meaningless and should be put into some state to represent our struct.

In previous examples default constructor is generated by compiler

```
struct Point{
  float x,y;
  //Correct
  Point(float x, float y) : x(x), y(y) {
  }
};
```

```
struct Point{
  float x,y;
  //Wrong!
  Point(float X, float Y){
    x = X;
    y = Y;
  }
};
```

Structs. Destruction.

After object is no longer needed and before deallocation of memory we run destruction operation on it. (to clean resources, etc.)

Default destructor is great for simple structs.

```
struct Point{
  float x,y;
  ~Point(){
    //Explicit destructor.
  }
};
```

Structs. Destructor & Constructor.

On stack

```
{ //Inside function, loop, if statement
  Point s(6.0, 66.0); //Constructor

  //...

  //Destructor called here
}
```

On heap

```
Point* s = new Point(6.0, 66.0); //Constructor + allocation

delete s; //Destructor + deallocation
```

RAII

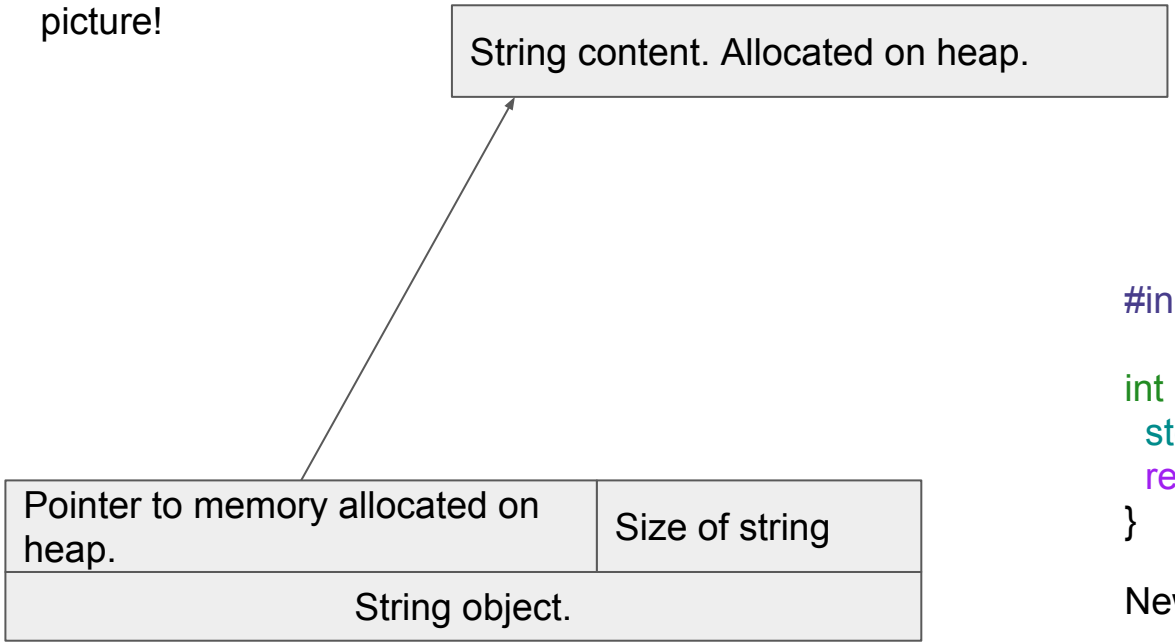
Destructor call is implicit and guaranteed to be executed on exit from { block } (even after exception is thrown.)

How we control resource? (file, net connection, memory on heap etc.)

1) We get it (place into construction of wrapper struct)
2) We use it
3) We free that resource. Don’t forget! (Place into destructor of wrapper struct)

RAII. std::string.

Simplified picture!

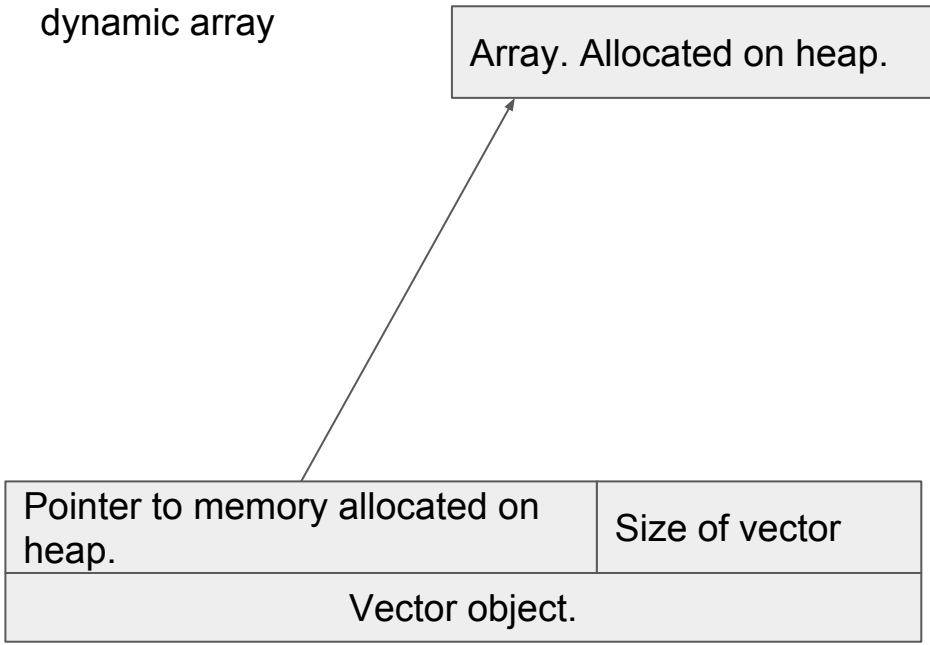



```
#include <string>

int main () {
  std::string str("Hello world");
  return 0;
}
```

New and delete are automatic.

RAII. std::vector

C++ - style
dynamic array




```
#include <vector>

int main () {
  std::vector<int> vec;
  vec.push_back(3);
  vec[0] = 33;
  return 0;
}
```

Vector itself - on stack.
Content of vector - on heap.

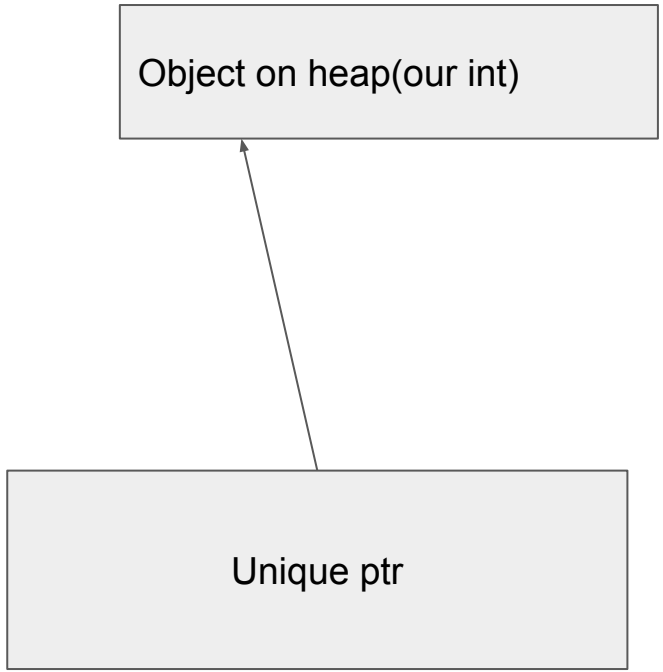
RAII. File stream

```
#include <iostream>
#include <fstream>

int main () {
  std::ofstream myfile("example.txt"); //FILE OPENED HERE
  myfile << "Writing this to a file.\n";
  return 0; //FILE CLOSED HERE
}
```

RAII. Smart pointers. Unique.

Since C++11.

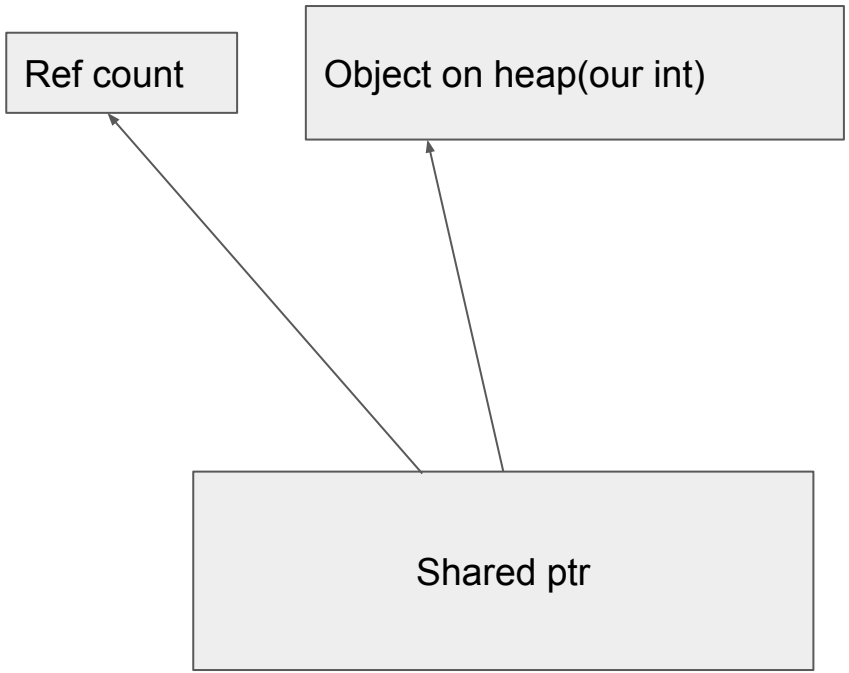
```
#include <memory>

int main () {
  std::unique_ptr<int> intPtr(new int); //Can't copy it!
  *intPtr = 10; //Can be used as usual pointer
  return 0; //Delete is called here in destructor
}
```



RAII. Smart pointers. Shared.

Since C++11.

```
#include <memory>

int main () {
  std::shared_ptr<int> intPtr(new int); //Can copy it!
  *intPtr = 10;
  return 0;
}
```



Class.

Class - is struct with private members by default.

Plays major role in typical big C++ project organization.(for ex. ROOT, Geant)

Class is used for abstraction of some computation.(for ex. string class is used to abstract away from us details of manual string allocation and management + gives us handy operations on string like find)

Member visibility.

Public - visible to everyone. This is “face” of class. Public members will be used by outside code.

Private - visible to class itself. “insides” of object, no one can use them except itself(and no one is interested)

Protected - like private but visible to ancestors.

```
class Foo{
public:
  void DoStuff();
private:
  int x; //details of implementation

protected:
  double y;
};
```

Virtual functions.

Solve problems of deciding which function to call after program compiled.

Example:

We are writing part of program that will take HEP event from user of our program, simulate it and write result to disk.

But user can provide event from different sources: read from disk, generate it with her generator, etc. We don’t care! We just need event.

Suppose event is generated in function GenerateEvent() and read from file in function ReadEventFromFile(). Does our program need to know about this functions? No.

Virtual functions.

Solution - virtual functions.

```
class EventProducer{
public:
  virtual Event* GetNextEvent() = 0;
};

//this class is called ancestor
class CustomEventGenerator : public EventProducer{
public:
  Event* GetNextEvent() override; //C++11 syntax
};

class EventFileReader : public EventProducer{
public:
  Event* GetNextEvent() override; //C++11 syntax
};
```

```
 //Usage
void Simulate(EventProducer* prod){
  //...
  Event* evt = prod->GetNextEvent();
  //...
}
```

Virtual functions

How do they work?

Every class derrived from EventProducer has pointer(vptr) to table of functions(vtable). One table for CustomEventGenerator, other for EventFileReader.

When EventProducer::GetNextEvent() is called we follow pointer and execute correct function from table.

Adds extra overhead. Now size of class is bigger than size of parts(now it contains extra pointer)

Virtual functions

How to use it properly?

```
Base* pBase = new Derrived;
//store pointer to derrived in pointer to base
class Base{
  virtual ~Base(){} //Virtual destructor is needed! Otherwise delete pBase will leak!
};
```