Object Oriented Programming - Polymorphism

OOP Principles, Polymorphism, polymorphic function calls

Object Oriented Programming Principles so far

1. Encapsulation - *hidden for good purpose*

Object Oriented Programming Principles so far

1. Encapsulation - *hidden for good purpose*
2. Abstraction - *the good parts of not caring at all*

Object Oriented Programming Principles so far

1. Encapsulation - *hidden for good purpose*
2. Abstraction - *the good parts of not caring at all*
3. Inheritance - *the only way to get rich in OOP (he-he :D)*

Object Oriented Programming Principles so far

1. Encapsulation - *hidden for good purpose*
2. Abstraction - *the good parts of not caring at all*
3. Inheritance - *the only way to get rich in OOP (he-he :D)*

Object Oriented Programming - Polymorphism

- *The word **polymorphism** meaning “having multiple forms”*
- The OOP **polymorphism principle -** *The ability to **send a message** to an object, **without knowing its type/class** (wtf???)*

Object Oriented Programming - Polymorphism

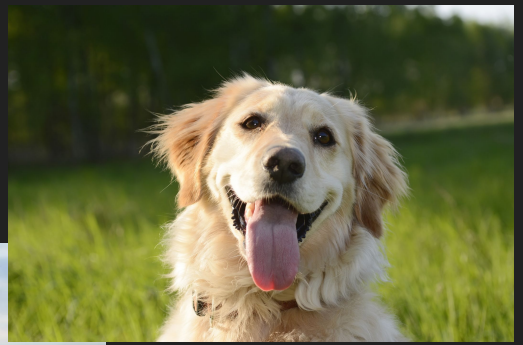
- *The word **polymorphism** meaning “having multiple forms”*
- The OOP **polymorphism principle -** *The ability to **send a message** to an object, **without knowing its type/class** (wtf???)*

Object Oriented Programming - Polymorphism

- *The word **polymorphism** meaning “having multiple forms”*
- The OOP **polymorphism principle -** *The ability to **send a message** to an object, **without knowing its type/class** (wtf???)*

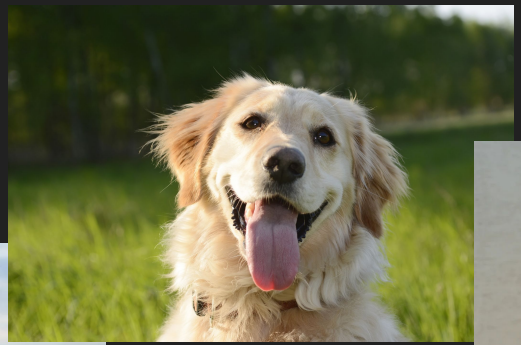
Object Oriented Programming - Polymorphism

- *The word **polymorphism** meaning “having multiple forms”*
- The OOP **polymorphism principle -** *The ability to **send a message** to an object, **without knowing its type/class** (wtf???)*



Object Oriented Programming - Polymorphism

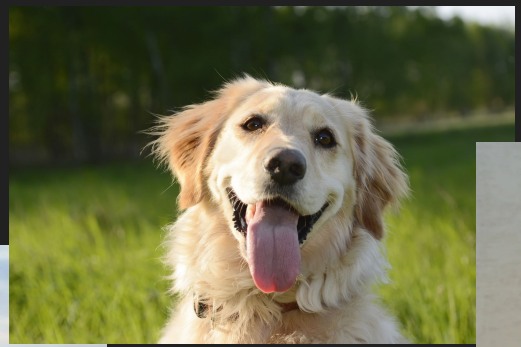
- *The word **polymorphism** meaning “having multiple forms”*
- The OOP **polymorphism principle -** *The ability to **send a message** to an object, **without knowing its type/class** (wtf???)*



Tell me what to do hooman.

“Sit down!”

“Sit down!”



“Sit down!”

“Sit down!”

Polymorphism - How to (C++)

- … without knowing it’s type/class
  - But we have to know something, what is it?
- In c++, “send a message” within the same program usually means calling a method
- To call a method polymorphically in C++ the following is necessary:
  - An instance of some class (i.e. a *Circle object*)
  - Either a ***reference or a pointer*** of some base class (i.e. a *Shape* shape or a Shape& shape*) ***that refer to the concrete object (****the Rectangle object****)***
  - The method is called via that base-class pointer or reference
  - The method **must be present** in the base class … and one more thing ...

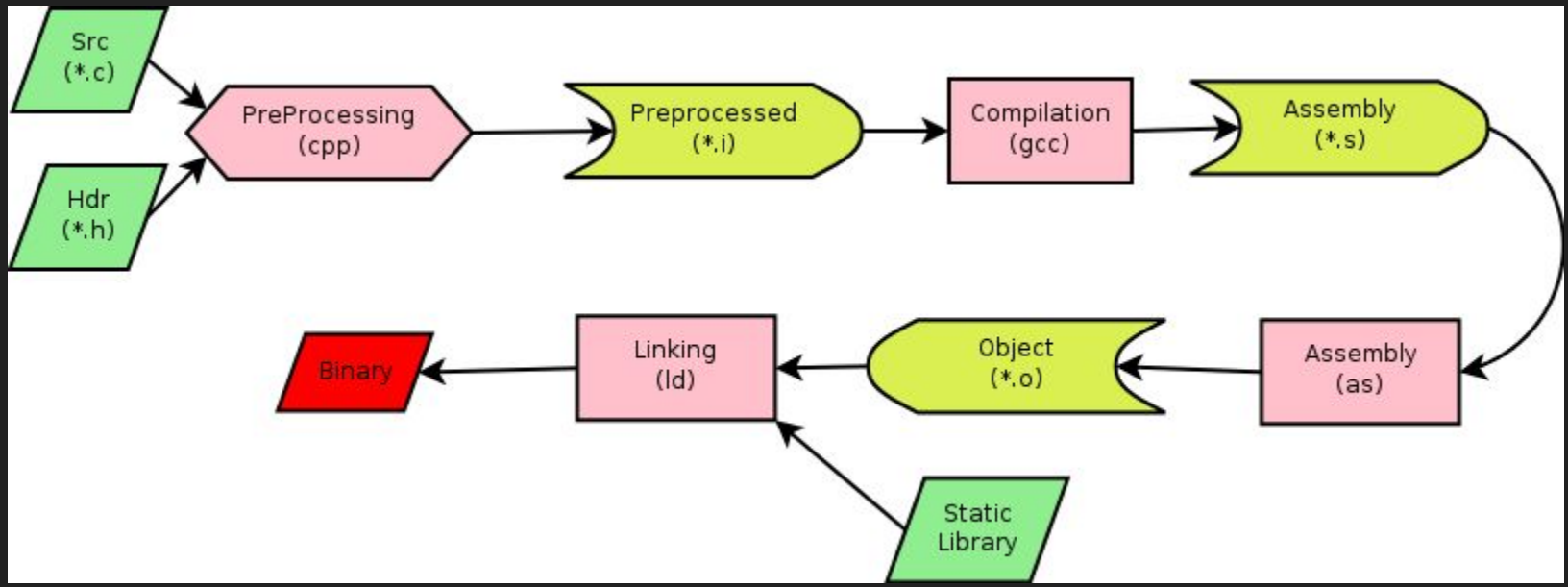
Polymorphism exercise

- *Canvas* is a drawing area that can visualize different *Shapes.* Make a class *Canvas* that ***contains*** *Circles, Rectangles and Squares.* Make sure that *Canvas* ***does not know (depend on)*** any of the concrete shapes. Inside *Canvas* create a method *update(),* that invokes the drawing methods of all *Shapes* in the *Canvas.*
- Implement that scenario using pointers to *Shape* class inside *Canvas* and try to call *draw()* to all objects pointed by the *Shape* pointers.

Compile time binding vs. Runtime binding

- Compile time binding (often called Early binding) is the ordinary way of binding to any methods / operators used so far and it is associated with the **type of object** the method being called.
- That way the compiler (or linker - reference on the next slide) directly associates an address to the function call - it replaces the actual call with machine language instructions
- Runtime binding (often called Late binding) is the way to achieve function calls to object, being **pointed by a pointer, not associated with the type of that pointer.**

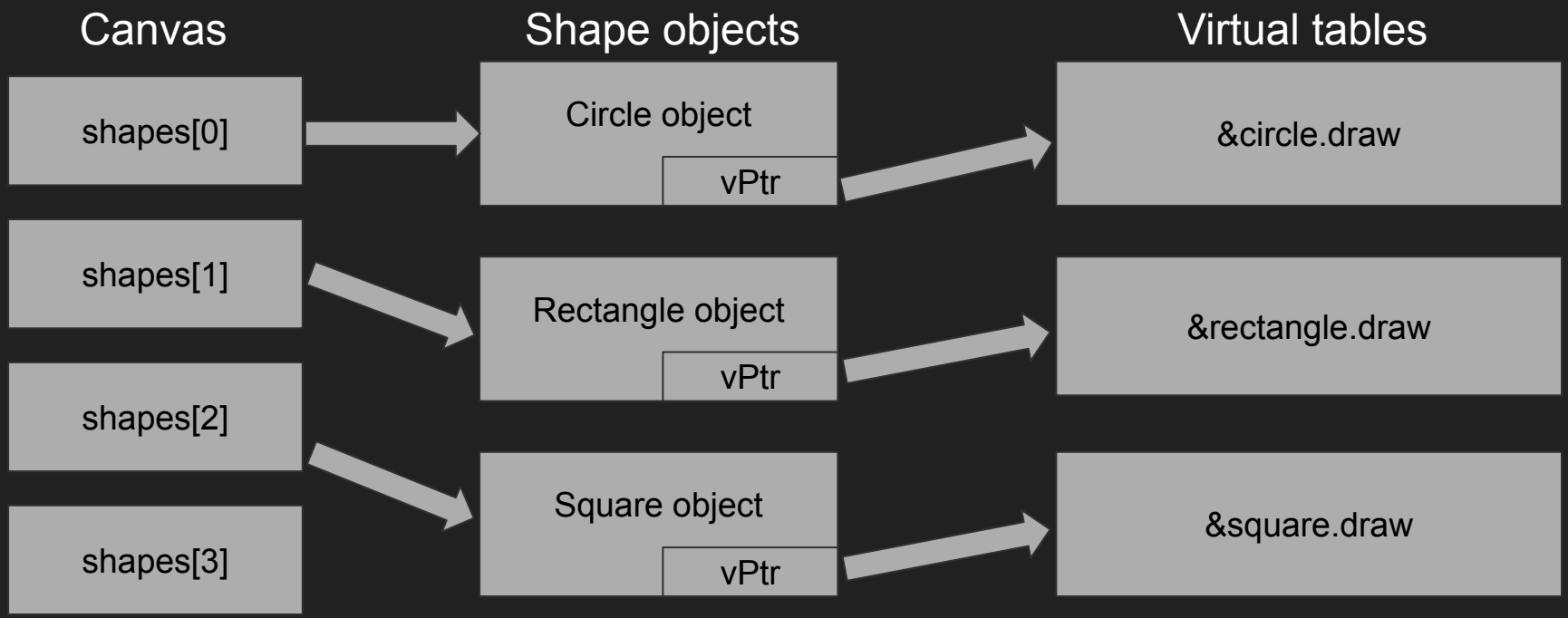
Compilation process



Virtual functions

- *Virtual functions* are called according to the **type** of the object instance **pointed to or referenced, not according to the type of pointer/reference**
- They are resolved late - at runtime (*How ??*)
- The compiler maintains two things to serve this purpose:
  - ***Virtual table*** of function pointers, **generated per class**
  - A ***pointer to virtual table*** **per object instance**
- Whenever an object is being created, a ***pointer to virtual table*** is being inserted to that object, pointing to the virtual table of the class
- Whenever a virtual function is being called, the compiler looks for that ***pointer to virtual table*** and when it is fetched, the virtual table is being accessed and specific function pointer is being called

Virtual tables and pointers to virtual tables



Questions?


IF YOU HAVE
A QUESTION... ASK
YOU WON'T BE
MAULED, PROMISE
imgflip.com