EDAF80 Introduction to Computer Graphics

Introductory Seminar

OpenGL & C++

Pierre Moreau

2019 Slides by Carl Johan Gribel, 2010-13 and Michael Doggett, 2014-17

Today

- Lab info
- OpenGL
- C(++)rash course

Labs overview

- 1 optional + 5 mandatory assignments
  - week 2-7
  - "Lab 0" in week 2: optional attendance, but mandatory preparation work for lab 1
  - book sessions on course homepage
- Work in pairs
  - If looking for a partner, post on forum
- E:**Uranus**
  - Located in the E-huset basement
  - Windows 10, 64 bit, Core i5, 8 GB RAM
  - Visual Studio 2019
  - Geforce GTX 560

Assignment 0

- Brand new: please give us feedback on it; feedback for the other labs is also welcomed
- Goal
  - Setting up and learning the framework
  - Applying some of the C++ knowledge from this seminar
  - Introducing helpers for the next assignment
- Content
  - Create a CelestialBody class, that can spin, be scaled and have rings
  - Extended in lab 1, with orbiting, tilting and parenting capabilities, to help create the Solar System
- Attending the lab sessions in week 2, is optional
  but the work still needs to be done before assignment 1 in week 3

Today

- Lab info ✔
- OpenGL
- C(++)rash course

OpenGL

OpenGL

- Application Programming Interface (API)
  - Set of functions that create a 2D image of a 3D scene
  - 3D scene is made of :
    - Primitives - Triangles
    - Textures - 2D images
    - and much, much more
- Controls a graphics pipeline (graphics hardware)
  - Graphics Processing Unit (GPU)

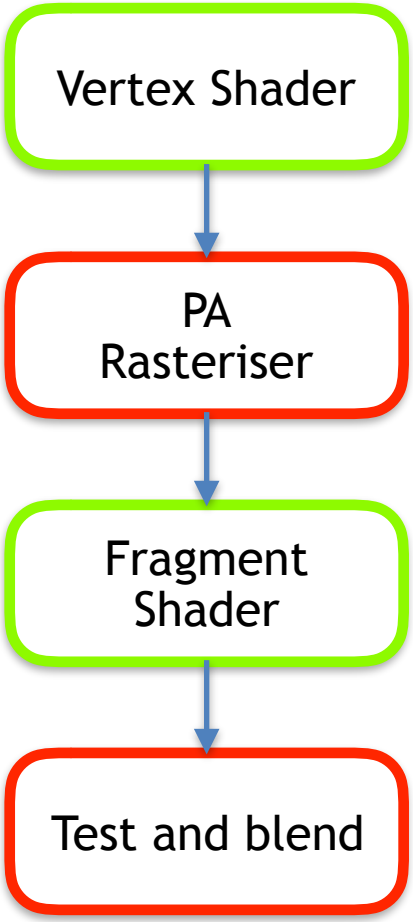
OpenGL

- We will focus on the core profile
  - no fixed function/immediate mode
- OpenGL is a state machine
  - Current state is the “OpenGL context”
  - There are many functions that change the current state
  - OpenGL uses objects that are a part of the state
  - Drawing uses the most recently bound buffers

Application setup

- First, make a window, use GLFW library
- Second, create a while loop i.e. the render loop:
  - grab inputs, render the screen, swap the buffers
- Third, do some rendering in the render loop …

OpenGL Pipeline



- Shaders are programmable, other parts are not
- There are no default Vertex and Fragment Shaders, you must provide them
- Primitive Assembly (PA) puts the vertices into the primitive that is currently specified

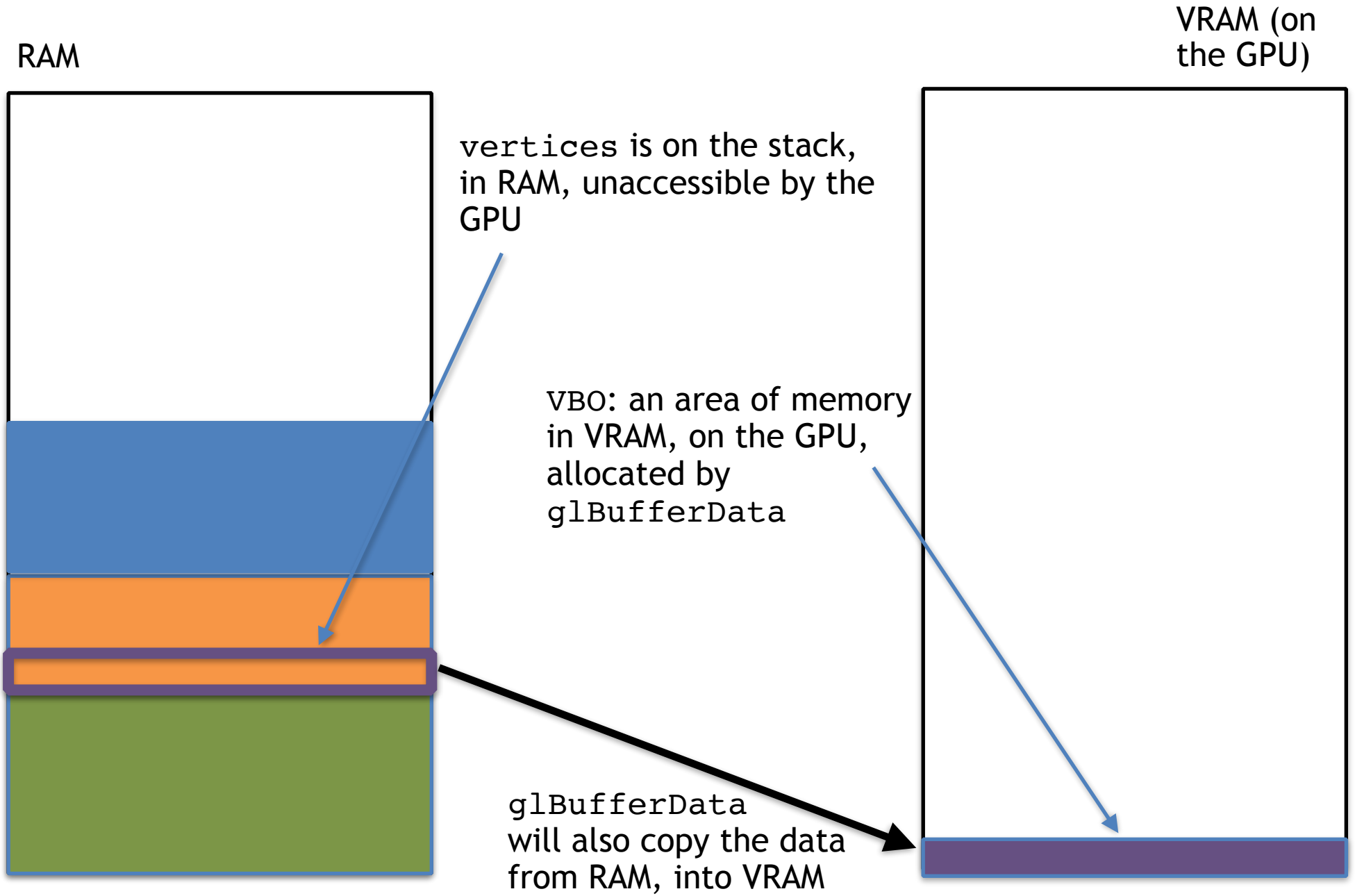
Vertices

- 3 vertices in (x,y,z)
  - Range from [-1,+1]
- Output from VS is in Normalized Device Coordinates (NDC)
  - also [-1,+1]
  - origin is in the middle of the screen
- Put vertices into Vertex Buffer Object (VBO)

```
GLfloat vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

```
GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Where Are Your Vertices?

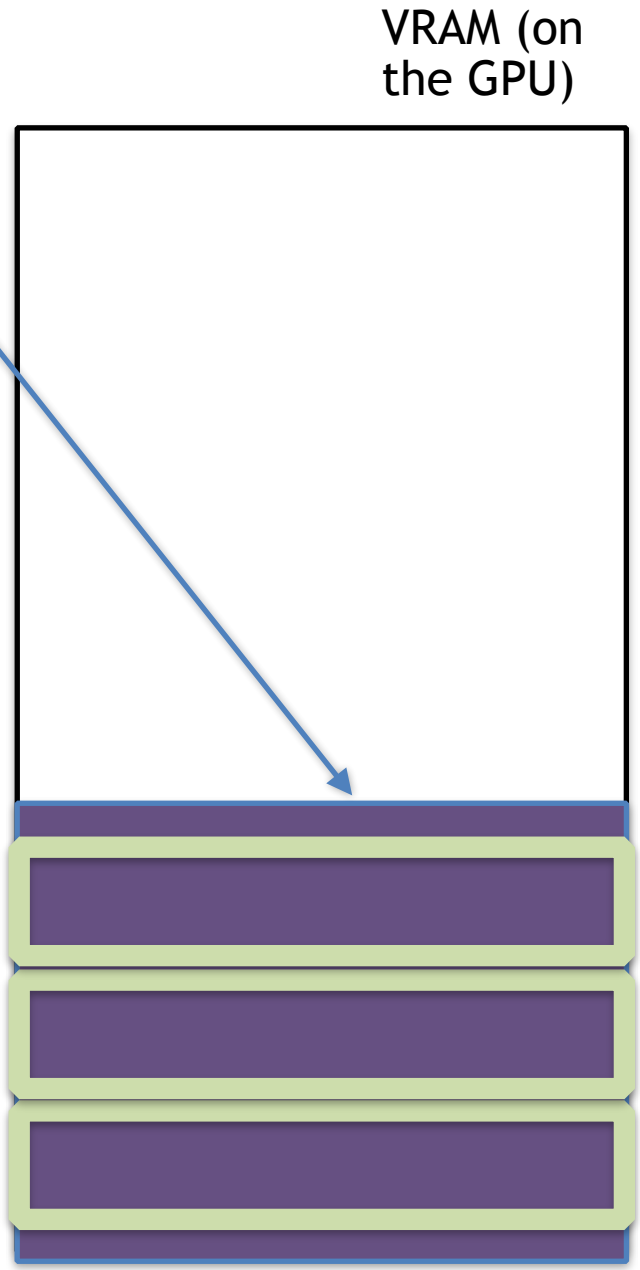


Simple Vertex Shader

- Must set the predefined variable **gl_Position**
- Need to link vertex data to the vertex shader
    - A Vertex Array Object (VAO) is also required

```
#version 400

in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```

How to Access the Vertices



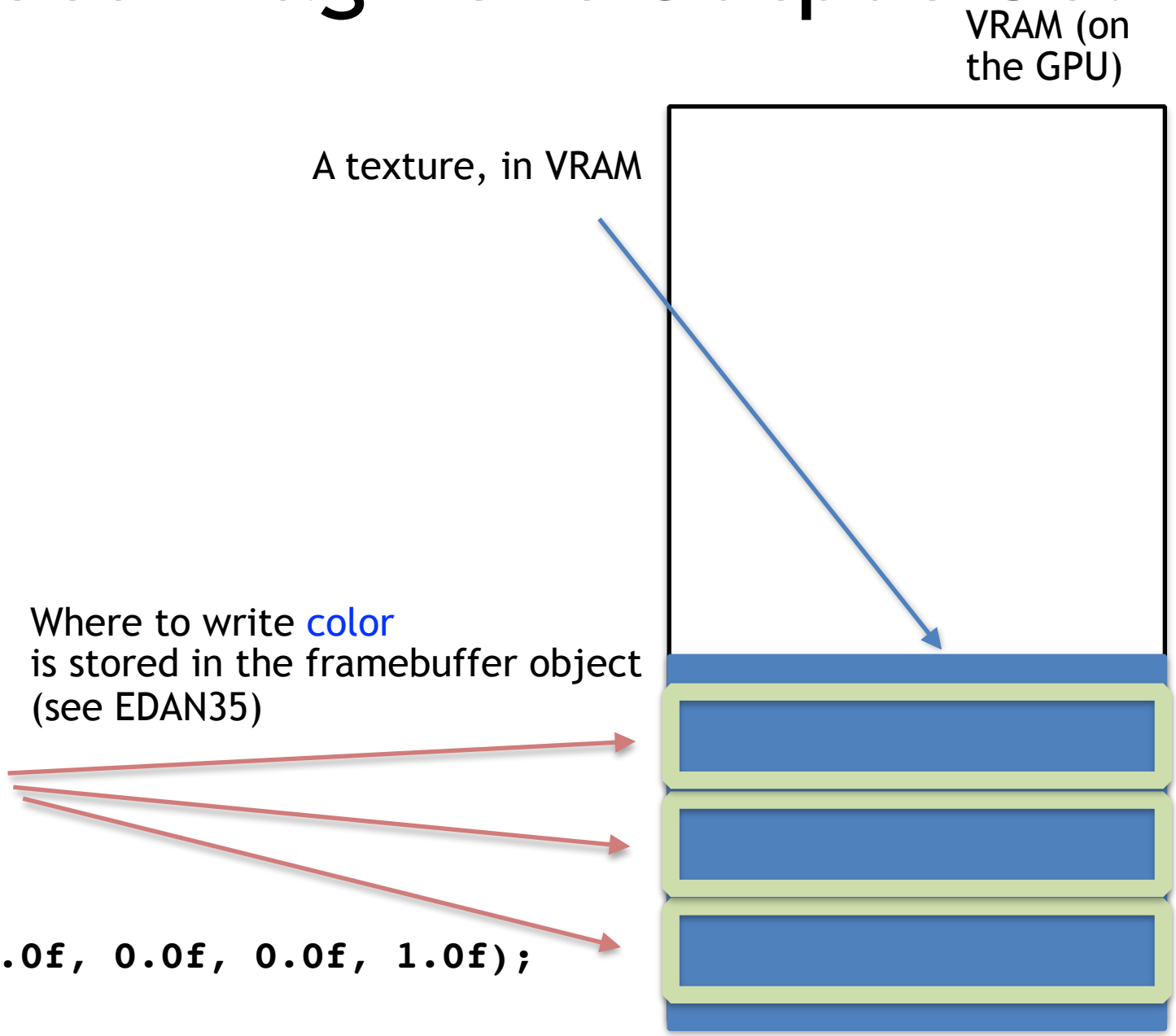

```
#version 400

in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```

Simple Fragment Shader

- Requires one output variable of **`vec4`**, for the color

```
#version 400

out vec4 color;

void main()
{
    color = vec4(1.0f, 0.0f, 0.0f, 1.0f); // set color to red
}
```

Where Does Fragment Output Go?




```
#version 400

out vec4 color;

void main()
{
    color = vec4(1.0f, 0.0f, 0.0f, 1.0f);
}
```

Buffers Versus Textures

- Both reside in VRAM on the GPU
- Both represent a chunk of memory (both can be viewed as *n*-d arrays, with data in cells)

Buffers:

- Supports any data format (even custom)
- Only the cells can be read
- Stored linearly

Textures:

- Only specific data formats allowed
- You can read between cells, and get interpolated results
- Stored in tiles

Compiling Shaders

- Shaders run on the **GPU**, not CPU
- They are written in GLSL, which is C-based
- Like for CPUs, need to compile to machine-specific instructions
- Unlike CPUs, shader compilation is done at runtime by your GPU driver

Compiling Shaders

Done in two steps:

1. Compile each shader individually

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
…
```

Check for possible compile errors after **glCompileShader**

2. Link all shaders into a single shader program

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
…
```

Check for possible compile errors after **glLinkProgram**

OpenGL - Drawing

- Tell OpenGL what to render
  - `glDrawArrays(GL_TRIANGLES, 0, 3);`
  - (what to draw, starting index, number of vertices);

OpenGL - more info

- http://www.learnopengl.com/
- Some first triangles :
  - Easiest : Anton Gerdelan, http://antongerdelan.net/opengl/hellotriangle.html
  - Alexander Overvoorde, https://open.gl/drawing (GL focused)
  - Joey de Vries, http://www.learnopengl.com/#!Getting-started/Hello-Triangle (Part of a complete course)

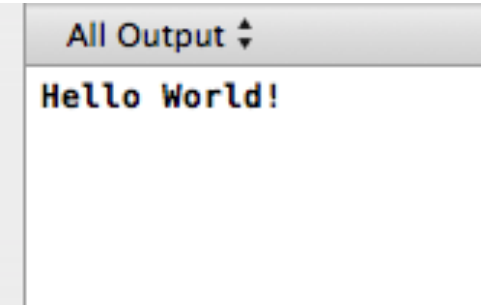
Today

- Lab info ✔
- OpenGL ✔
- C(++)rash course

C++

Hello C++

```
#include <iostream>

int main()
{
  std::cout << "Hello world!\n";
  return 0;
}
```

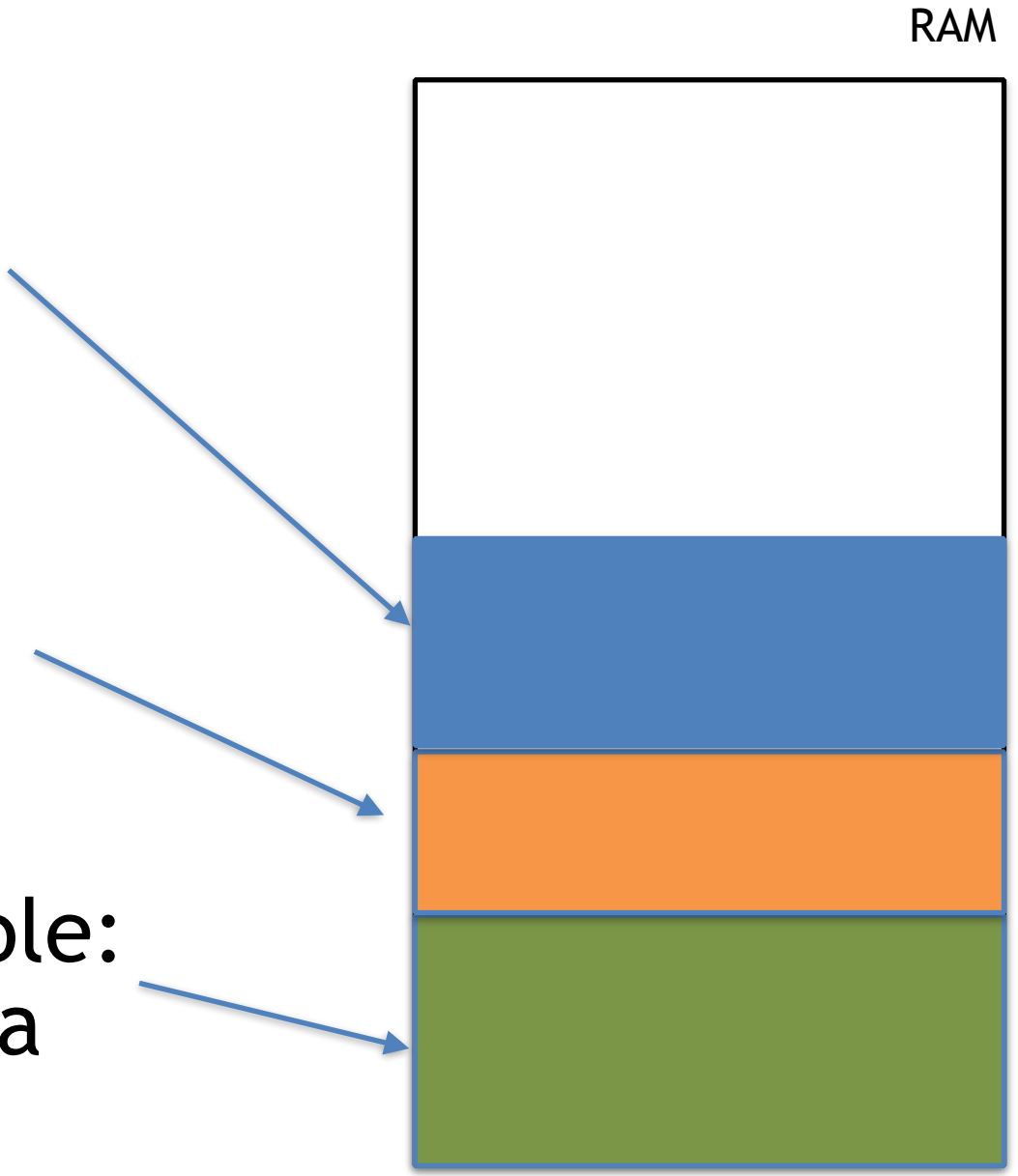


About C++

- Based on C
- Created by Bjarne Stroustrup in 80's 
- Object-oriented (classes and structs)
- Constructors & destructors
- Inheritance & virtual functions
- Operator overloading (+, -, *, / etc)
- Templates
- C++11 began a 3 year cycle of updates

Simplified runtime view

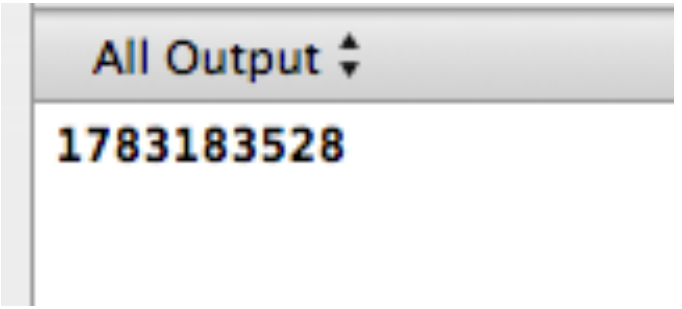
- Heap-space allocated: usage managed by the programmer
- Stack-space allocated: usage managed by the compiler
- Program executable: instructions + data



Stack integer declaration

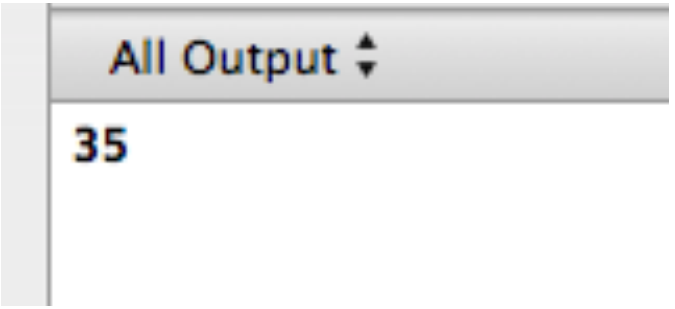
```
int x;
```

```
printf("%i", x);
```



Stack integer declaration & initialization

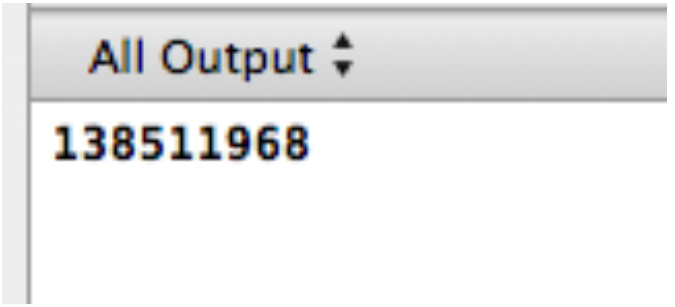
```
int x;
x = 35;

printf("%i", x);
```



Pointer to an integer

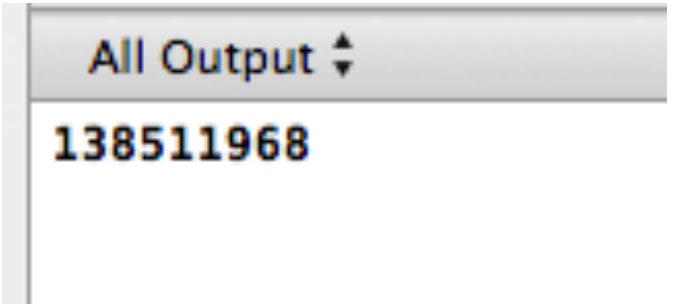
```
int *y;
```

```
printf("%i", y);
```



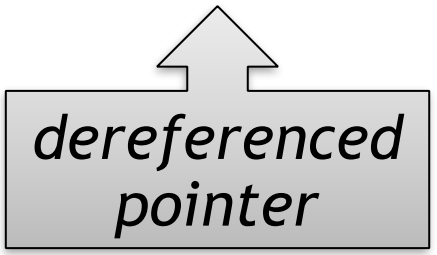
Allocate heap memory

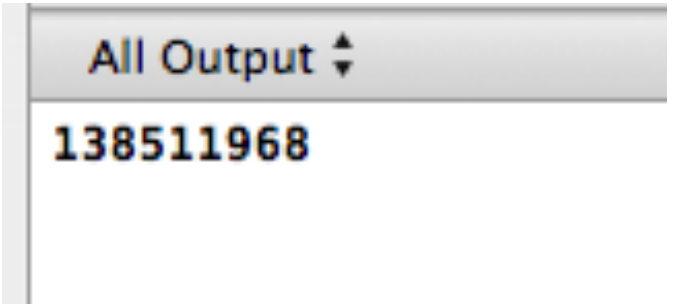
```
int *y;
y = new int(10);

printf("%i", y);
```

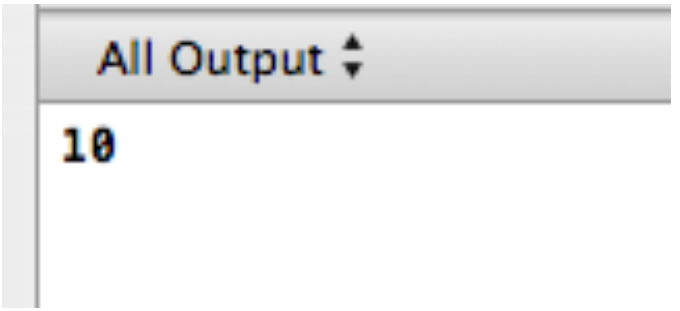


Pointer dereferencing

```
int *y;
y = new int(10);

printf("%i", y);

printf("%i", *y);
```





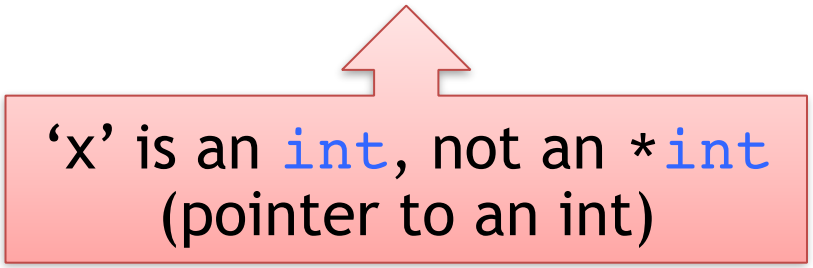


Pointer to stack integer

```
int x = 35;
int *xp;
```

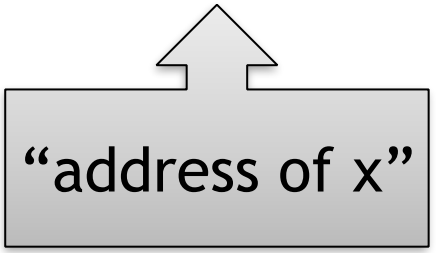
Pointer to stack integer

```
int x = 35;
int *xp = x;
```

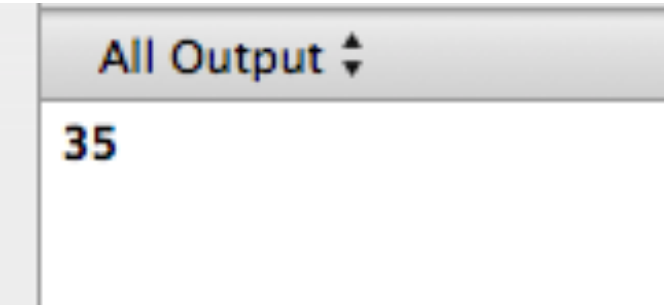


Pointer to stack integer

```
int x = 35;
int *xp = x;
int *xp = &x;
```




```
printf(“%i”, *xp);
```



Heap deallocation

```
int *y = new int(10);
…
delete y;
```

C++ class

```
class MyClass
{



};
```

C++ class: scope

```
class MyClass
{
    class scope
};
```

C++ class: member access

```
class MyClass
{
private:
        access within this class only
                (default)
protected:
        access to this and inherited classes
public:
        access to everyone
};
```

C++ class: constructor

```
class MyClass
{
    float mX;

    MyClass(float x)
    {
        mX = x;
    }
};
```

C++ class: constructor + initialization

```
class MyClass
{
    float mX;

    MyClass(float x) : mX(x)
    {

    }
};
```

C++ class: constructor & destructor

```
class MyClass
{
    float mX;

    MyClass(float x)
    {
        mX = x;
    }

    ~MyClass()
    {

    }
};
```

(mX on stack so automatically deallocated)

C++ class: constructor & destructor

```
class MyClass
{
    float *mXp;

    MyClass(float x)
    {
        mXp = new float(x);
    }

    ~MyClass()
    {
        delete mXp;
    }
};
```

mX on heap,
deallocate manually

C++ class: member method

```
class MyClass
{
    float mX;

    void setX(float x)
    {
        mX = x;
    }
};
```

Class member access

Stack

```
MyClass myclass = MyClass(5);
myclass.setX(2);
```

Heap

```
MyClass *myclassp = new MyClass(5);
myclassp->setX(2);
...
delete myclassp;
```

C++ class: declaration + definition

MyClass.h

```
class MyClass
{
    float mX;

    void setX(float x);
};
```

MyClass.cpp

```
#include "MyClass.h"

void MyClass::setX(float x)
{
    mX = x;
}
```

Array stack & heap allocation

Stack 1

```
float numbers[3];
numbers[0] = 1.0f; ...
```

Stack 2 (direct initialization)

```
float numbers[3] = { 1.0f, 2.0f, 3.0f };
```

Heap

```
float *numbers = new float[3]
numbers[0] = 1.0f; ...
...
delete[] numbers;
```

Parameters: Value, Reference, Pointer

```
MyClass mc0 = MyClass(1);
MyClass mc1 = MyClass(1);
MyClass *mc2 = new MyClass(1);

foo(mc0, mc1, mc2);
```

by copy

by reference (Java-style)

by pointer

```
int foo(MyClass mc0, MyClass &mc1, MyClass *mc2)
{
    mc0.setX(10);

    mc1.setX(10);

    mc2->setX(10);
}
```

edits local copy only

edits original

edits original

C++ Types

```
int                a = -1;                (32 bit)
unsigned int       b = 1u;                (32 bit)
long               c = -2l;               (64 bit)
unsigned long      d = 2lu;               (64 bit)
float              e = 1.0f;              (32 bit)
double             f = 3.14;              (64 bit)
bool               g = true;              (8 bit)
char               h = 'x';               (8 bit)
char               i[] = "abcd";          (5*8 bit)
...
```

Operator overloading

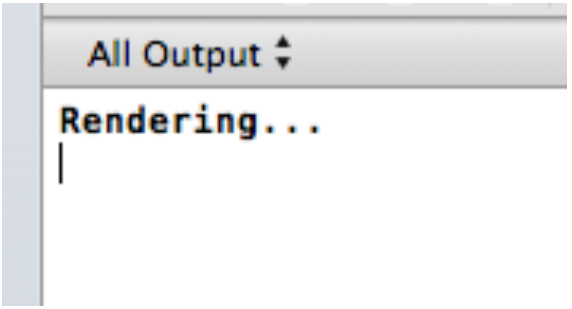
- May customize +, -, *, / and many others
- Very useful for linear algebra, e.g.:

```
glm::mat3 A, B;
glm::vec3 u;
…
glm::mat3 M = A * B;
glm::vec3 v = M * u;
```

Output

Print “Rendering...” to standard output, followed by a new line:

```
std::cout << "Rendering...\n";
```



Or, with the same result:

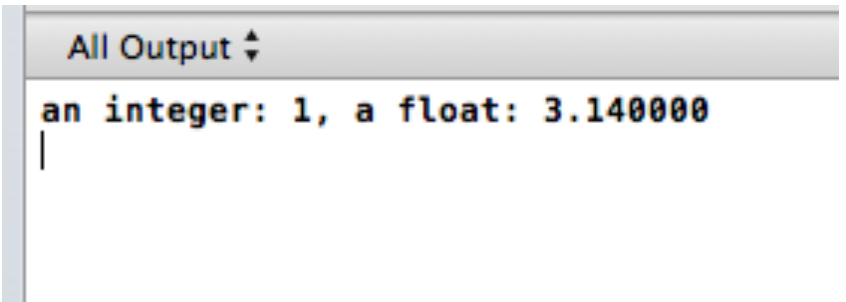
```
printf("Rendering...\n");
```

Inclusion of variables (many formatting options available):

```
std::cout << "an integer: " << 1 << ", a float: "
          << 3.14f << '\n';
```

Or, with the same result:

```
printf("an integer: %d, a float: %f\n", 1, 3.14f);
```



More info

- EDA031 - C++ Programming

  - cplusplus.com

    http://www.cplusplus.com/doc/tutorial/

Today

- Lab info ✔
- OpenGL ✔
- C(++)rash course ✔

End

- Kursombud/course representative
- Check the forum often
- Next week : Assignment 1

- Questions?