Laboratoire d'Informatique Graphique et Géométrique

C++

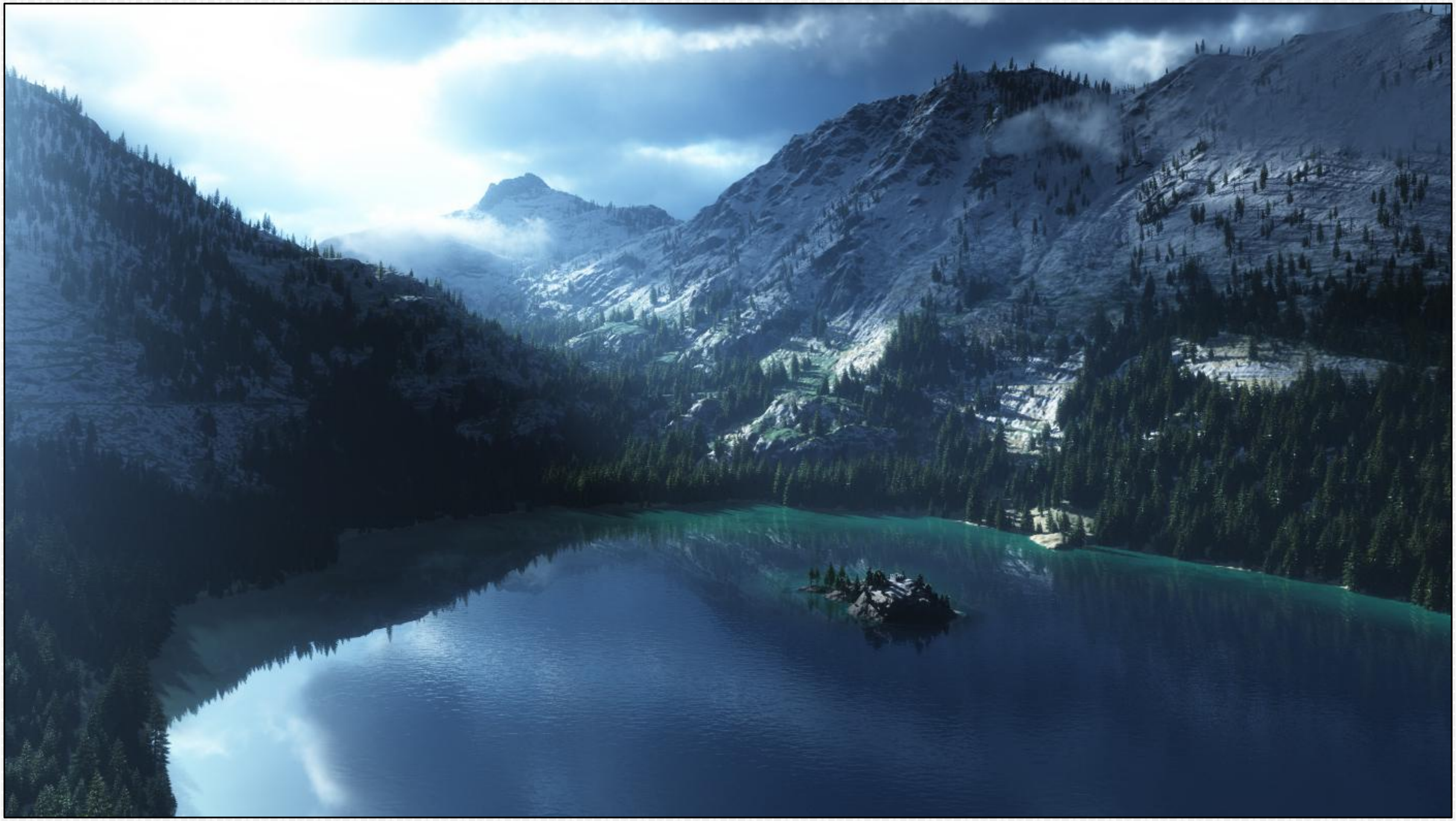
Sofien Bouaziz (BC346) – *sofien.bouaziz@epfl.ch*

TODAY

- C++ Introduction
  - Feel free to ask questions
  - Please participate
  - Take it easy

C++

- Widely used in CG industry…



WHAT YOU MAY KNOW

- Java
  - Object-Oriented Programming
  - Containers (vector, map, …)
  - …
- C
  - Pointers
  - Memory Allocations
  - …

WHAT YOU WILL LEARN TODAY

- 1 – Pointers vs References
- 2 – Memory Allocations
- 3 – Classes
- 4 – Inheritance
- 5 – Templates and STL
- 6 – Debugging Advices

1 – POINTERS VS REFERENCES

1 – POINTERS VS REFERENCES

- References can **basically** be seen as pointers. They are safer but less powerful. Let’s have a look at references in action and how they differ from pointers…

1 – POINTERS VS REFERENCES

Pointer

```
int i = 5;
```

i 5

Reference

```
int i = 5;
```

5 i

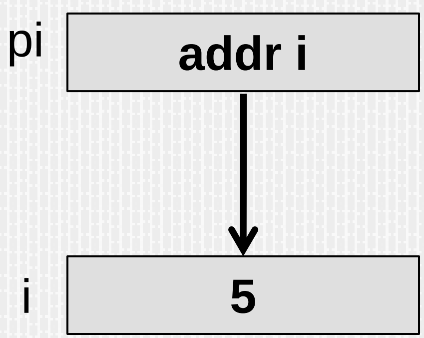
1 – POINTERS VS REFERENCES

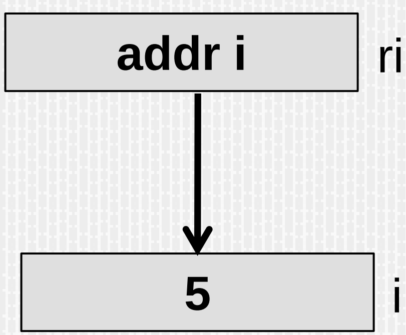
Pointer

```
int i = 5;
int *pi = &i;
```

Reference

```
int i = 5;
int& ri = i;
```

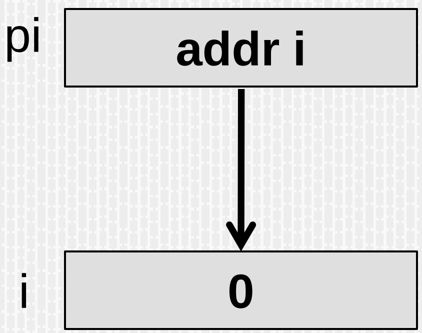




1 – POINTERS VS REFERENCES

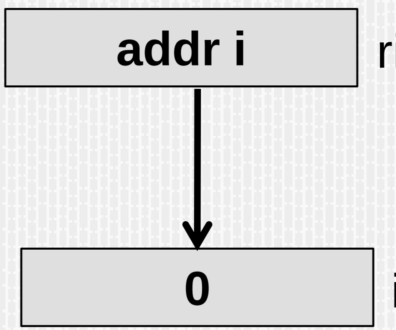
Pointer

```
int i = 5;
int *pi = &i;
*pi = 0;
```



Reference

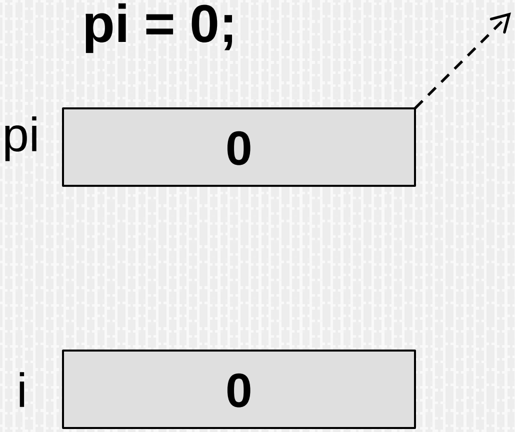
```
int i = 5;
int& ri = i;
ri = 0;
```



1 – POINTERS VS REFERENCES

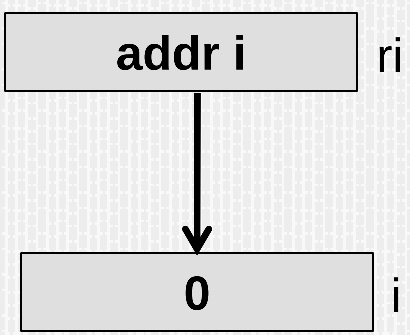
Pointer

```
int i = 5;
int *pi = &i;
*pi = 0;
pi = 0;
```



Reference

```
int i = 5;
int& ri = i;
ri = 0;
-----
```



1 – POINTERS VS REFERENCES

- References have many advantages
  - Safer than pointers (see bellow)
  - User friendly (code comprehension)
- Because they are safer they have some limitations
  - No arithmetic (they are like const-pointers)
  - Must be initialized and cannot be NULL
  - Unlike pointers they cannot be reseated

1 – POINTERS VS REFERENCES

- Let’s try to replace all the pointers of this code with references

Pointer

```
int x = 5;
int y[2] = {10, 11};
int *p1 = &x;
int *p2 = p1;
*p2 = 4;
int *p3 = &(y[0]);
p2 = p3;
p3++;
```

1 – POINTERS VS REFERENCES

- Do you think this is right?

Pointer

```
int x = 5;
int y[2] = {10, 11};
int *p1 = &x;
int *p2 = p1;
*p2 = 4;
int *p3 = &(y[0]);
p2 = p3;
p3++;
```

Reference

```
int x = 5;
int y[2] = {10, 11};
int& r1 = x;
int& r2 = r1;
r2 = 4;
int& r3 = y[0] ;
r2 = r3;
r3++;
```

1 – POINTERS VS REFERENCES

- Sadly it is not possible to convert it. It is compiling but does something different…

Pointer	Reference

`int x = 5;`
`int y[2] = {10, 11};`
`int *p1 = &x;`
`int *p2 = p1;`
`*p2 = 4;`
`int *p3 = &(y[0]);`
`p2 = p3;`
`p3++;`
	`int x = 5;`
	`int y[2] = {10, 11};`
	`int& r1 = x;`
	`int& r2 = r1;`
	`r2 = 4;`
	`int& r3 = y[0] ;`
	~~`r2 = r3;`~~
	~~`r3++;`~~

1 – POINTERS VS REFERENCES

- What is actually happening?

Pointer

```
int x = 5;
int y[2] = {10, 11};
int *p1 = &x;
int *p2 = p1;
*p2 = 4;
int *p3 = &(y[0]);
*p2 = *p3;
(*p3)++;
```

Reference

```
int x = 5;
int y[2] = {10, 11};
int& r1 = x;
int& r2 = r1;
r2 = 4;
int& r3 = y[0] ;
r2 = r3;
r3++;
```

1 – POINTERS VS REFERENCES

- A good reason to use them : pass by reference-to-const avoids copying parameters (more efficient for big objects) and is almost as safe and as easy to use than pass by value

```
void f (Object o)
{
	// do stuff with o
}
```

```
void f (const Object& o)
{
	// do stuff with o
}
```

2 – MEMORY ALLOCATIONS

2 – MEMORY ALLOCATIONS

Static Allocation

```
int value;
int array[10];
```

Dynamic Allocation

```
int *value = new int;
delete value;

int *array = new int[10];
delete [] array;
```

- What are the three main differences?
  - Life time (Scope/Until deleted)
  - Memory Location (Stack/Heap)
  - Compiler Time/Run Time

2 – MEMORY ALLOCATIONS

- Let’s have a look at an example with three of my best friends.

```
void f()
{
     int *mario = new int;
     int pacman;
     {
          int sonic;
     }
    delete mario;
}
```

Stack	Heap

2 – MEMORY ALLOCATIONS

- Let's have a look at an example with three of my best friends.

```
void f()
{
    int *mario = new int;
    int pacman;
    {
        int sonic;
    }
   delete mario;
}
```

Stack

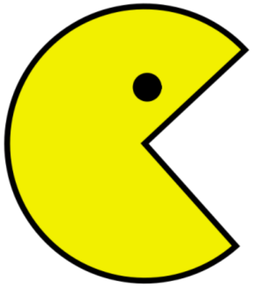
Heap

2 – MEMORY ALLOCATIONS

- Let's have a look at an example with three of my best friends.

```
void f()
{
    int *mario = new int;
    int pacman;
    {
        int sonic;
    }
    delete mario;
}
```

Stack



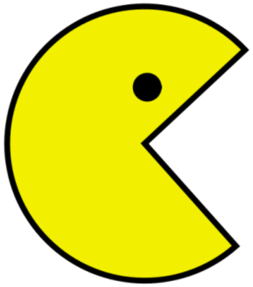
Heap

2 – MEMORY ALLOCATIONS

- Let's have a look at an example with three of my best friends.

```
void f()
{
    int *mario = new int;
    int pacman;
    {
        int sonic;
    }
   delete mario;
}
```

Stack

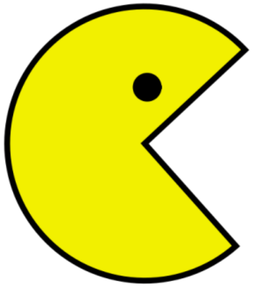


Heap

2 – MEMORY ALLOCATIONS

- Let's have a look at an example with three of my best friends.

```
void f()
{
    int *mario = new int;
    int pacman;
    {
        int sonic;
    }
    delete mario;
}
```

2 – MEMORY ALLOCATIONS

- What is wrong with these cases?

```
void f()
{
    int *a = NULL;
    if(a == NULL)
    {
        int b = 4;
        a = &b;
    }
    *a = 5;
}
```

```
int& f()
{
    int a = 2;
    return a;
}
```

```
int* f()
{
    int a = 2;
    return &a;
}
```

3 – CLASSES

3 – CLASSES

- Create your own data types. The only limit is your imagination. Let’s build a Nanosuit…

3 – CLASSES

```cpp
//Nanosuit.h
class NSuit
{
    public:
        //Constructor
        NSuit();
        //Copy Constructor
        NSuit(const NSuit& _ns);
        //Destructor
        ~NSuit();
        //Operator
       NSuit& operator=(const NSuit& _ns);
        //Member function
        double getMeanHealth() const;
    protected:
        …..
    private:
        //Data member
        double head, body;
};
```

```cpp
//Nanosuit.cpp
#include “Nanosuit.h”
//Constructor
NSuit :: NSuit() : head(1.0), body(1.0) { }

//Copy Constructor
NSuit :: NSuit(const NSuit& _ns) {*this = _ns; }

//Operator
NSuit& NSuit :: operator=(const NSuit& _ns)
{
    if(this == &_ns) return *this;
    head = _ns.head; body = _ns.body;
    return *this;
}

//Destructor
NSuit ::~ NSuit()  {}

//Member function
double NSuit :: getMeanHealth() const
{
    return (head+body)/2.0;
}
```

3 - CLASSES

- Can you tell me which function among the Constructor, Copy constructor, Copy assignment operator and Destructor is called?

```
{
    NSuit n1;
    NSuit n2 = n1;
    NSuit n3(n1);
    NSuit n4;
    n4 = n1;
}
```

3 – CLASSES

- Constructor, Copy constructor, Copy assignment operator and Destructor are automatically generated by the compiler if not provided.

3 - CLASSES

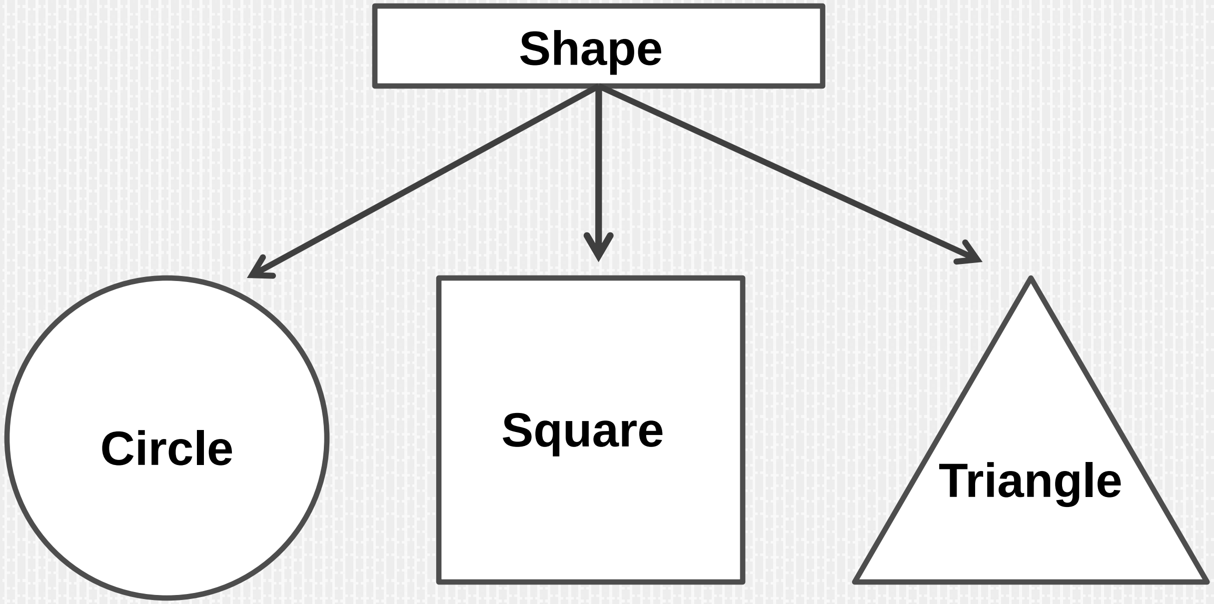
- Can you tell me which function among the Constructor, Copy constructor, Copy assignment operator and Destructor is called?

```
{
    NSuit n1;           // Constructor
    NSuit n2 = n1;      // Copy constructor
    NSuit n3(n1);       // Copy constructor
    NSuit n4;           // Constructor
    n4 = n1;            // Copy assignment operator
}                       // Destructor for n1, n2, n3 and n4
```

4 – INHERITANCE

4 - INHERITANCE

- In CG we like shapes…



4 - INHERITANCE

```
class Shape
{
    public:
        void whoAmI() const
         {
              std::cout << "Shape" << std::endl;
         }
    protected:
        double centerX, centerY, centerZ;
};

class Square : public Shape
{
    public:
        void whoAmI() const
         {
              std::cout << "Square" << std::endl;
         }
};
```

```
class Triangle : public Shape
{
    public:
        void whoAmI() const
        {
            std::cout << "Triangle" << std::endl;
        }
};

class Circle : public Shape
{
    public:
        void whoAmI() const
        {
            std::cout << "Circle" << std::endl;
        }
};
```

4 - INHERITANCE

- What is happening here?

```
{
        Shape *s =  new Circle;
        s->whoAmI();
        delete s;
}
```

4 - INHERITANCE

- What is happening here?

```
{
        Shape *s =  new Circle; // Called Circle and then
                                //  Shape constructor
        s->whoAmI(); // Print «Shape»
        delete s; //Only called Shape destructor (leak!)
}
```

4 - INHERITANCE

```
class Shape
{
    public:
        virtual ~Shape() {}
        virtual void whoAmI() const
        {
            std::cout << "Shape" << std::endl;
        }
    protected:
        double centerX, centerY, centerZ;
};
class Square : public Shape
{
    public:
        virtual ~Square () {}
        virtual void whoAmI() const
        {
            std::cout << "Square" << std::endl;
        }
};
```

```
class Triangle : public Shape
{
    public:
        virtual ~Triangle () {}
        virtual void whoAmI() const
        {
            std::cout << "Triangle" << std::endl;
        }
};

class Circle : public Shape
{
    public:
        virtual ~Circle () {}
        virtual void whoAmI() const
        {
            std::cout << "Circle" << std::endl;
        }
};
```

4 - INHERITANCE

- What is happening here?

```
{
        Shape *s =  new Circle;
        s->whoAmI();
        delete s;
}
```

4 - INHERITANCE

- What is happening here?

```
{
        Shape *s =  new Circle; // Called Shape and then
                                // Circle constructor
        s->whoAmI(); // Print «Circle»
        delete s; // Called Circle and then
                  Shape destructor (this is better!)
}
```

4 - INHERITANCE

```
class Shape
{
    public:
        Shape(double x, double y, double z) :
        centerX(x), centerY(y), centerZ(z) {}
        virtual ~Shape() {}
        virtual void whoAmI() const = 0;
    protected:
        double centerX, centerY, centerZ;
};
class Square : public Shape
{
    public:
        Square(double x, double y, double z) :
        Shape(x, y, z) {}
         virtual ~Square () {}
         virtual void whoAmI() const
         {
              std::cout << "Square" << std::endl;
         }
};
```

```
class Triangle : public Shape
{
    public:
        Triangle(double x, double y, double z) :
        Shape(x, y, z) {}
        virtual ~Triangle () {}
        virtual void whoAmI() const
         {
              std::cout << "Triangle" << std::endl;
         }
};
class Circle : public Shape
{
    public:
        Circle(double x, double y, double z) :
        Shape(x, y, z) {}
         virtual ~Circle () {}
         virtual void whoAmI() const
         {
              std::cout << "Circle" << std::endl;
         }
};
```

5 – TEMPLATES/STL

5 – TEMPLATES/STL

- Quick introduction to a wide subject and a really powerful feature of C++. Templates allow you to have generic type for functions and objects

- STL : Standard template library

5 – TEMPLATES/STL

- A template function

```
template <typename T>
T getMax(T a, T b)
{
	return (a>b) ? a : b;
}
```

```
float a = getMax(1.0f, 2.5f); //Automatically Determined
float a = getMax<float>(1.0f, 2.5f);
```

5 – TEMPLATES/STL

- A template object

```
template <typename T>
class Vector4
{       ….
        private:
                T values[4];
};
```

```
Vector4<int> v;
```

5 – TEMPLATES/STL

- Now you better understand STL
  - I may need to explain you namespaces (std::)

```
std::vector<int> v;
std::map<int, float> m;
….
```

6 – DEBUGGING ADVICES

6 – DEBUGGING ADVICES

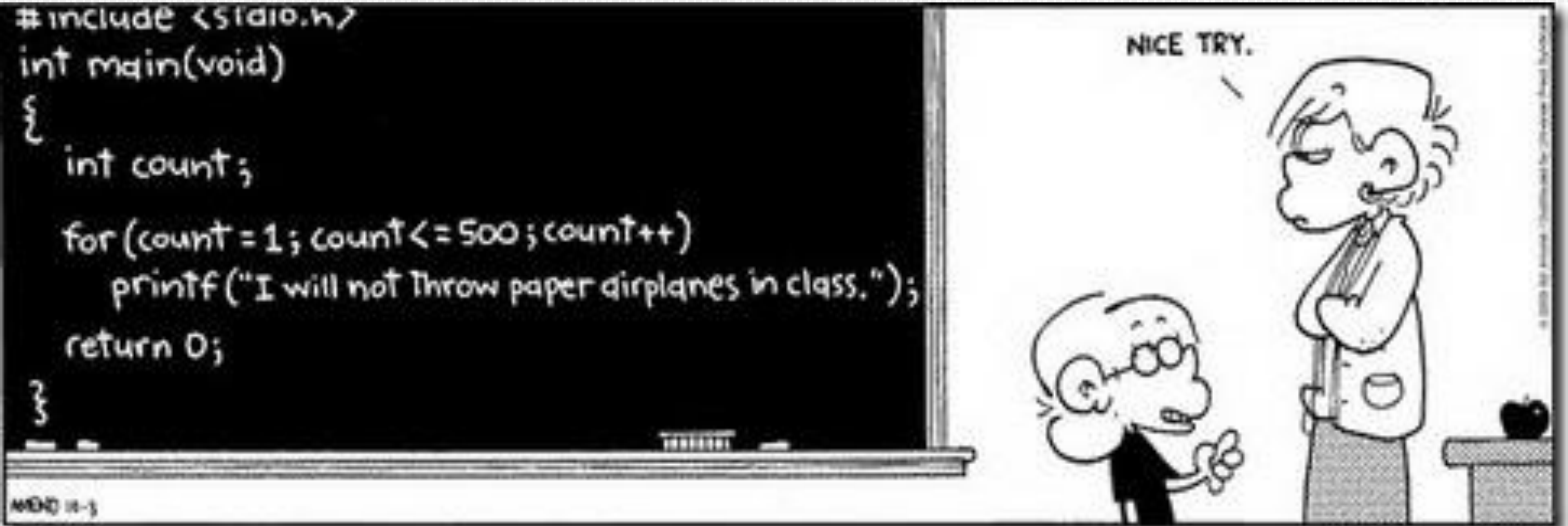
- "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian W. Kernighan

6 – DEBUGGING ADVICES

- Debugging will be easier if
  - Your code is readable ("premature optimization is the root of all evil")
  - Your code has a good design (modular, code factorisation, …)
  - You use assertions (cassert / boost assert)
  - You carefully manage the memory
    - smart pointers (TR1 / boost)
    - vector, string, map, … (STL / TR1 / boost)
    - ….
  - You use Design Patterns
    - Resource Acquisition Is Initialization (RAII)
    - ….

- Don't be afraid to use the debugger!

THE END



#include <stdio.h>
int main(void)
{
int count;
for(count=1;count<=500;count++)
printf("I will not Throw paper airplanes in class.");
return 0;
}
NICE TRY.

WEBSITES

- LGG lecture website: http://lgg.epfl.ch

- C++ (many tutorials available online):
  - http://www.cplusplus.com
  - http://www.learncpp.com
  - http://www.cprogramming.com
- Books
  - **The C++ Programming Language** - Bjarne Stroustrup
  - **Effective C++** - Scott Meyer
  - **Modern C++ Design** - Andrei Alexandrescu