

Simple Class Building

Overview

- Standard C++ Code Structure
- Constructors, Methods, and Friends

References

- Gary J. Bronson: C++ for Engineers and Scientists. 3rd Edition. Thomson (2010)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

A Simple Class

- Consider class A:

```
class A
{
    private:
        int instanceVariable1;
        float instanceVariable2;

    public:
        A();

        void method1( int argument );

        int method2() const;
        float method3() const;

        friend float function( A argument1, A argument2 );
};
```


Where do we implement class A?

- Header file A.h:

```
#pragma once

class A
{
private:
    int instanceVariable1;
    float instanceVariable2;

public:
    A();

    void method1( int argument );

    int method2() const;
    float method3() const;

    friend float function( A argument1, A argument2 );
};
```

- Implementation file A.cpp:

```
#include "A.h"

A::A() { ... }

void A::method1( int argument ) { ... }


int A::method2() const { ... }
float A::method3() const { ... }

float function( A argument1,
               A argument2 ) { ... }
```

Implementing a Member Function

```
class A
{
private:
    int instanceVariable1;
    float instanceVariable2;

public:
    A();
    ...
};
```



The constructor for class A:

```
A::A()
{
    instanceVariable1 = 0;
    instanceVariable2 = 0.0;
}
```


What does `A::A()` mean?

```
A::A()  
{  
    instanceVariable1 = 0;  
    instanceVariable2 = 0.0;  
}
```

- `A::` in `A::A()` is called a **scope name**. It refers here to the class the constructor `A()` belongs to.
- This is the way we implement methods C++.
- It is helpful to remember that in OOP methods of a class have a first implicit argument: **this**. Hence we can think of the scope name `A::` here as telling us that the method we are implementing has an implicit first argument named **this** of type `A*` (pointer to this object of type `A`). The pointer semantics can be confusing at first when coming from C#.

Do not use this approach!

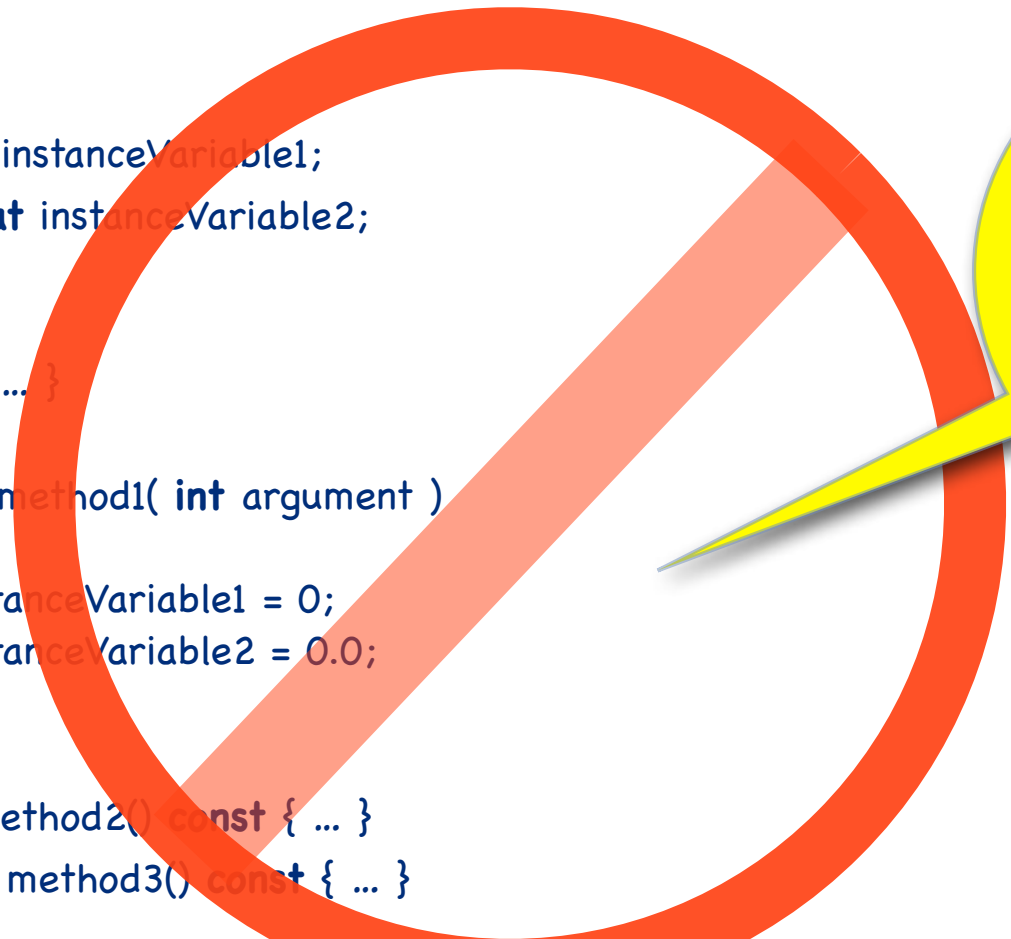
```
class A
{
private:
    int instanceVariable1;
    float instanceVariable2;

public:
    A() { ... }

    void method1( int argument )
    {
        instanceVariable1 = 0;
        instanceVariable2 = 0.0;
    }

    int method2() const { ... }
    float method3() const { ... }

    friend float function( A argument1, A argument2 ) { ... }
};
```



Do not implement methods inside the class declaration as you may have done in C#.

It may lead to problems.

Use the CPP file to implement methods.

```
A::A()
{
    instanceVariable1 = 0;
    instanceVariable2 = 0.0;
}

void A::method1( int argument )
{
    instanceVariable1 += argument;
}

...
```

You can use this to refer to members.

```
float A::method3() const
{
    return this->instanceVariable2;
}
```

return instanceVariable2;
would work here also

- Method3 is a getter. It does not change the object. Hence the method has been decorated with **const** – readonly access.
- The variable **this** is a pointer to the current object. Using deference ***this** yields the object itself. Writing **this->member** means we dereference the this pointer to obtain the current object and select member in one step.

Friends

```
friend float function( A argument1, A argument2 );
```

- Friends are not member of a class.
- We use the friend declaration to say that functions (or classes and operators) have private access rights to objects of our class. This simplifies some coding and avoids the definition of getters and setter to a large extent.
- When implementing friends you can drop the **friend** keyword.

A Friend Implementation

```
float function( A argument1, A argument2 )  
{  
    return argument1.instanceVariable2 +  
           argument2.instanceVariable2;  
}
```

- Function has private access to the fields of the objects argument1 and argument2.
- This feature is what we use when implementing input and output in C++ which relies on overloading operator>> and operator<<.

operator<< – Output

```
ostream& operator<<( ostream& aOStream, const A& aObject )  
{  
    aOStream << aObject.instanceVariable1  
              << " "  
              << aObject.instanceVariable2;  
  
    return aOStream;  
}
```

- The output operator<< requires access to the fields of objects of type A in order to work.
- We could use corresponding getters here, but this would add operational overhead. Hence we revise class A.

Class A with support for output

```
#include <iostream>

class A
{
private:
    int instanceVariable1;
    float instanceVariable2;

public:
    A();

    void method1( int argument );

    int method2() const;
    float method3() const;

    friend float function( A argument1, A argument2 );
    friend std::ostream& operator<<( std::ostream& aOStream, const A& aObject );
};
```


operator<< is a binary operator

```
#include <iostream>
#include "A.h"

using namespace std;

int main()
{
    A obj;

    cout << obj << endl;

    return 0;
}
```

- Here, `cout << obj << endl` means:
 - first run `out << obj` that returns an updated `cout`
 - next run on the updated `out << ends` which further updates `out`

operator<< cannot be a member of a general class.

```
#include <iostream>
```

```
class A
```

```
{
```

```
...
```

```
public:
```

```
    A();
```

```
...
```

```
    std::ostream& operator<<( std::ostream& aOStream, const A& aObject );  
};
```



- This does not work: operator<< now takes three arguments, which is illegal:
 - implicit first argument this of type A*
 - second argument of type reference to ostream
 - third argument of type reference to readonly object of type A