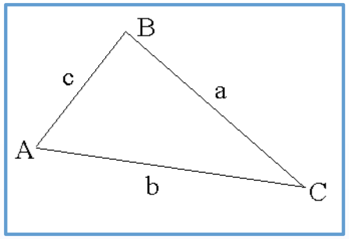
EECS 280 – Lecture 8

Abstract Data Types in C++

2/2/2022

Review: ADTs in C



- Define functions for `Triangle` **behaviors**.

```
struct Triangle {
  double a, b, c;
};

void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in);

double Triangle_perimeter(Triangle const *tri);

void Triangle_scale(Triangle *tri, double s);

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;
}
```

The first parameter is a pointer to the object we're working with.

Respect the interface!

Onward to C++…

- Build the link between an ADT's data and behaviors (functions) into the language itself.
- Protect raw member data, but allow the ADT’s own functions to access.
- Provide a mechanism to ensure ADT objects are ALWAYS initialized.

On to classes!

struct	class
Heterogeneous aggregate data type	Heterogeneous aggregate data type
C style	**C++ style**
Contains only data	**Contains data and functions**
Undefined by default	**Constructors can be used to initialize**
All data is accessible	**Control of data access**

Introducing Classes

- A `class` has both **member data** and **member functions**.

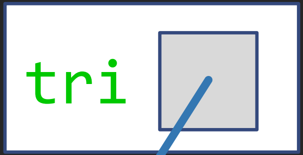
```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in) { ... }

  double perimeter() const { ... }
  void scale(double s) { ... }
};
```

```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter();
}
```

Member Functions

C Style(struct)

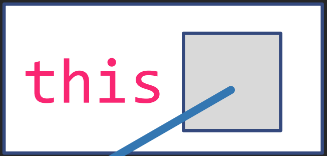
```
void Triangle_scale(
  Triangle *tri, double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}


int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
}
```



We had to pass the address of t1 ourselves.

C++ Style (class)

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }
};
```

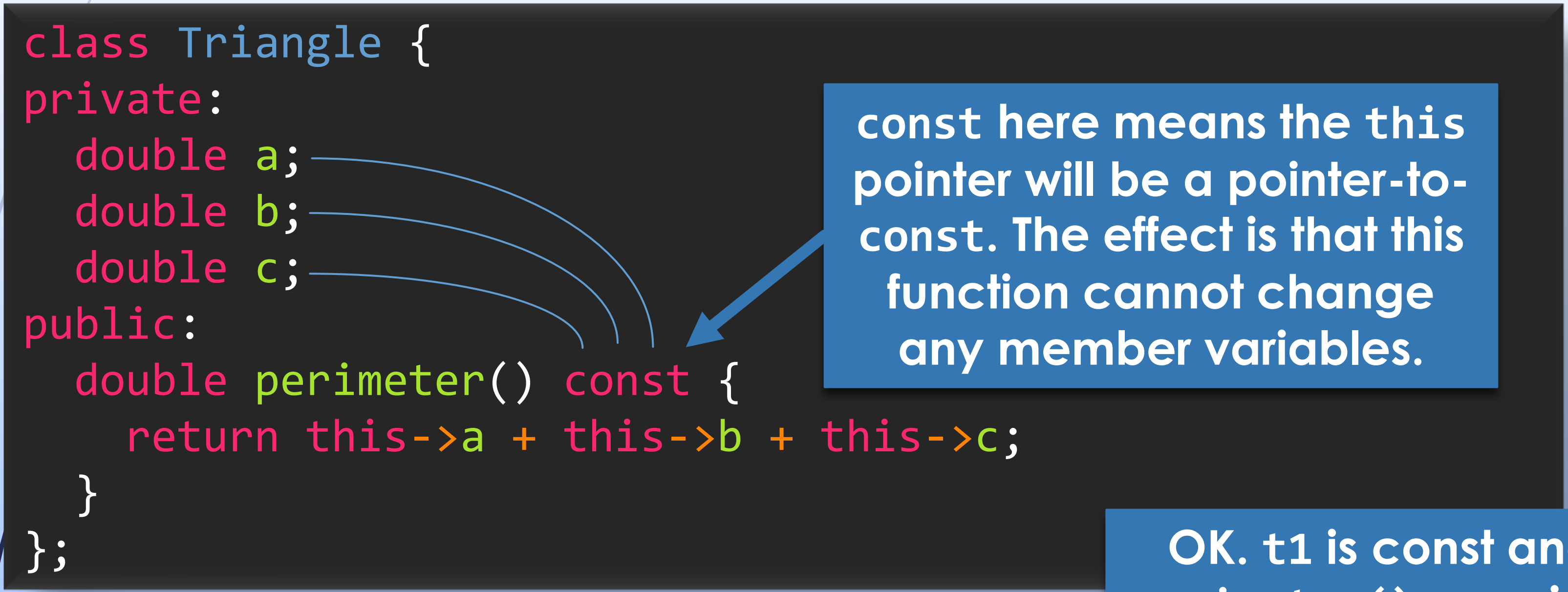


```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
}
```

Compiler does it for us.

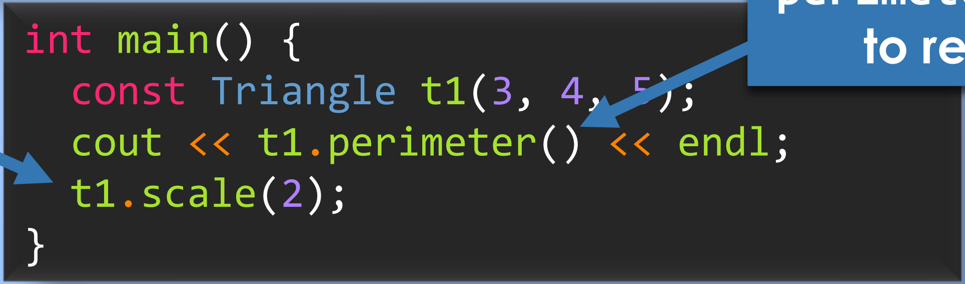
const Member Functions

- The `perimeter` function shouldn't change the `Triangle` (i.e. its member variables).



```
class Triangle {
private:
  double a;
  double b;
  double c;
public:
  double perimeter() const {
    return this->a + this->b + this->c;
  }
};
```

const here means the this pointer will be a pointer-to-const. The effect is that this function cannot change any member variables.



```
int main() {
  const Triangle t1(3, 4, 5);
  cout << t1.perimeter() << endl;
  t1.scale(2);
}
```

OK. t1 is const and perimeter() promises to respect this.

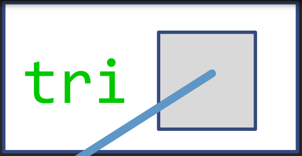
Compile error since t1 is const but scale isn't.

const Member Functions

C Style(struct)

```
double Triangle_perimeter(
  Triangle const *tri) {
  return tri->a +
    tri->b;
    tri->c;
}
```

tri is a pointer to const

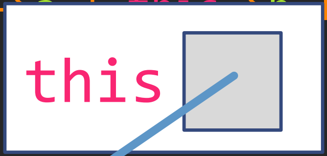


```
int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  cout << Triangle_perimeter(&t1);
}
```

C++ Style (class)

```
class Triangle {
private:
  double a;
  double b;
  double c;
public:
  double perimeter() const {
    return this->a + this->b +
      this->c;
  }
};
```

this is a pointer to const

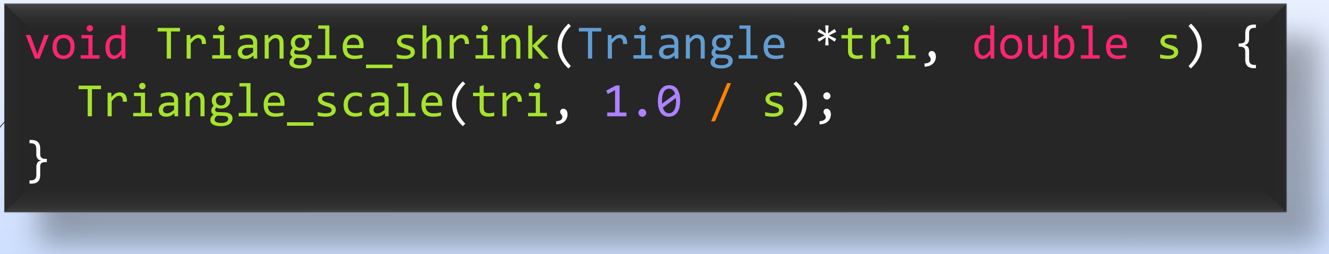


```
int main() {
  Triangle t1(3, 4, 5);
  cout << t1.perimeter();
}
```

Member Functions

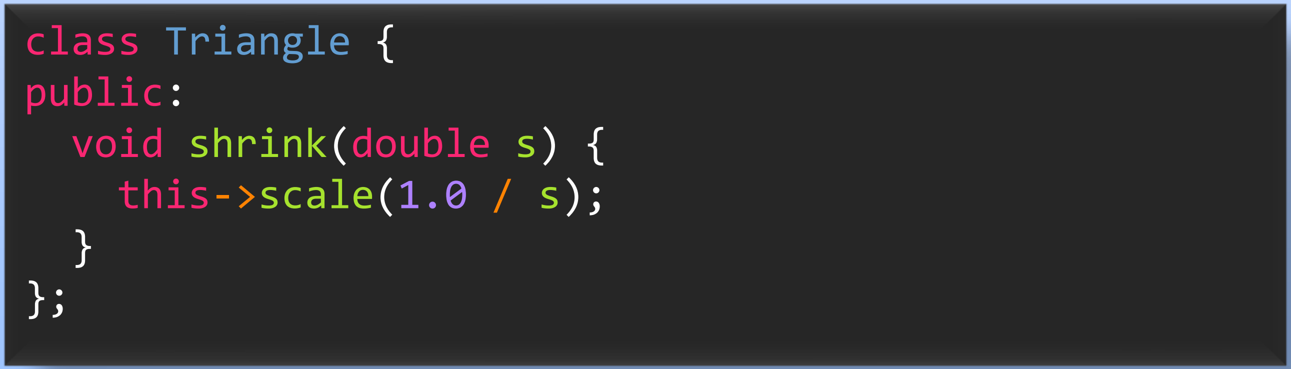
- You should reuse functionality wherever you can.

C Style(struct)



```
void Triangle_shrink(Triangle *tri, double s) {
  Triangle_scale(tri, 1.0 / s);
}
```

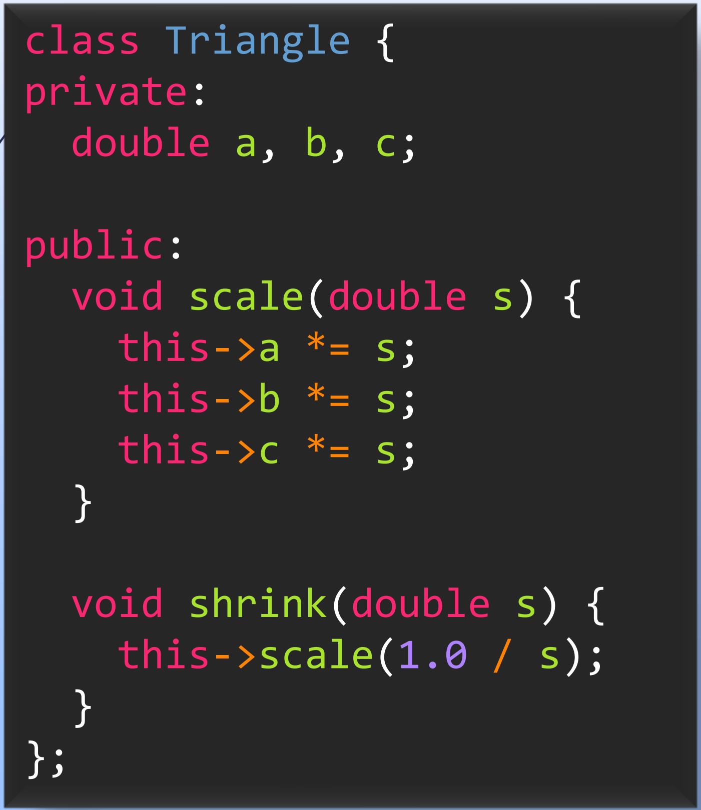
C++ Style (class)



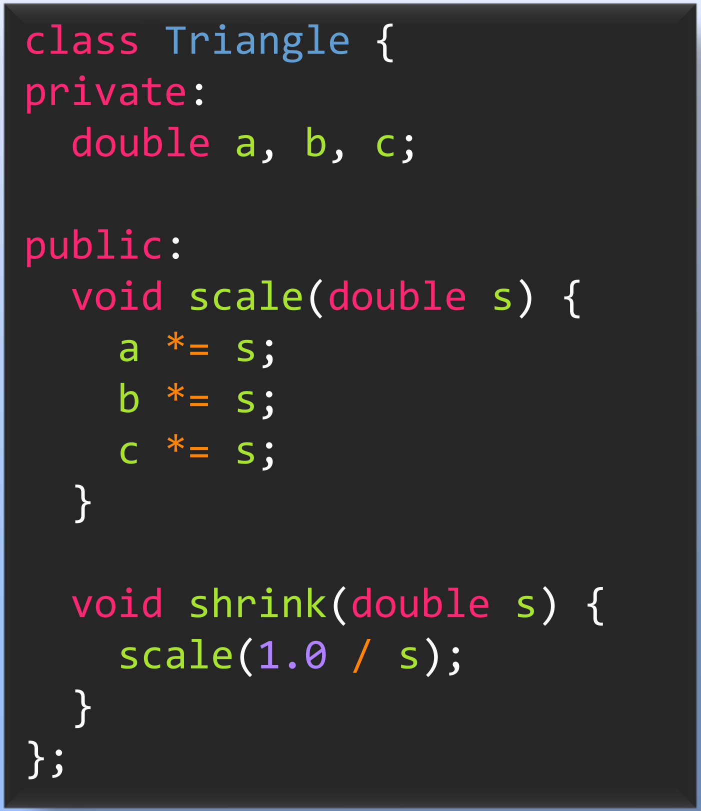
```
class Triangle {
public:
  void shrink(double s) {
    this->scale(1.0 / s);
  }
};
```

Using Members Without `this`

- Members can be referred to directly in a member function.
  - (The compiler inserts `this->` for you.)



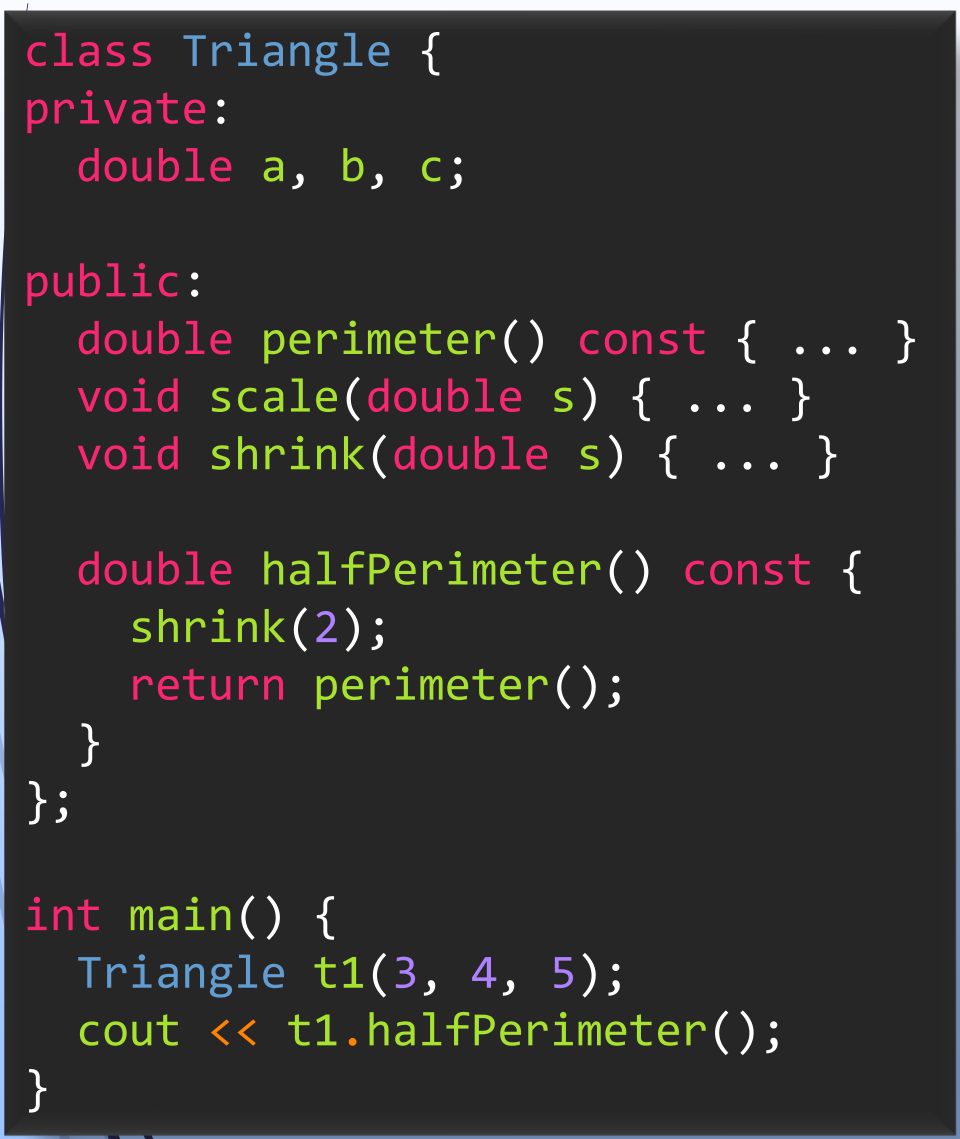
```
class Triangle {
private:
  double a, b, c;

public:
  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }

  void shrink(double s) {
    this->scale(1.0 / s);
  }
};
```



```
class Triangle {
private:
  double a, b, c;

public:
  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }

  void shrink(double s) {
    scale(1.0 / s);
  }
};
```

Exercise

- What's wrong with the function `halfPerimeter`?



```
class Triangle {
private:
  double a, b, c;

public:
  double perimeter() const { ... }
  void scale(double s) { ... }
  void shrink(double s) { ... }

  double halfPerimeter() const {
    shrink(2);
    return perimeter();
  }
};

int main() {
  Triangle t1(3, 4, 5);
  cout << t1.halfPerimeter();
}
```

Question

A) A const is missing on `shrink()`.

B) The `this` keyword is missing somewhere.

C) The call to `shrink(2)` won't compile.

D) It computes the wrong result.

E) The call to `perimeter()` won't compile.

Member Accessibility

- Declare members with an access level.
  - Public: Can be used anywhere.
  - Private: Can only be used in **class scope**.

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};
```

Data members are private. Member functions are public.

Accessing a, b, c here is fine since we're inside Triangle.

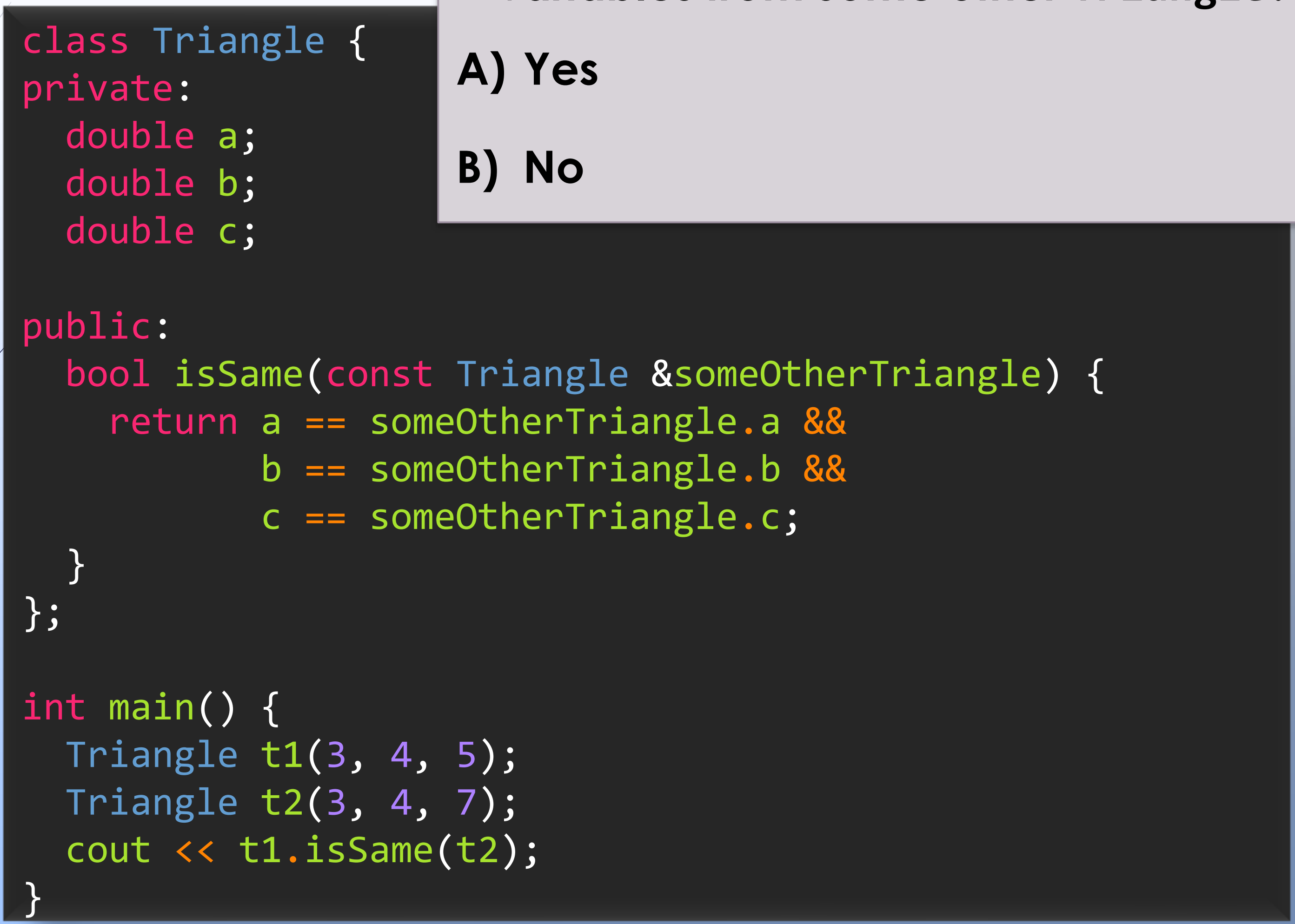
```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter();

  // Die triangle! DIE!
  t1.a = -1;
}
```

Ok. These member functions are public.

Compile error! a is private and not accessible here!

People sometimes call this *visibility* instead of *accessibility*.

Exercise



```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  bool isSame(const Triangle &someOtherTriangle) {
    return a == someOtherTriangle.a &&
           b == someOtherTriangle.b &&
           c == someOtherTriangle.c;
  }
};

int main() {
  Triangle t1(3, 4, 5);
  Triangle t2(3, 4, 7);
  cout << t1.isSame(t2);
}
```

Question

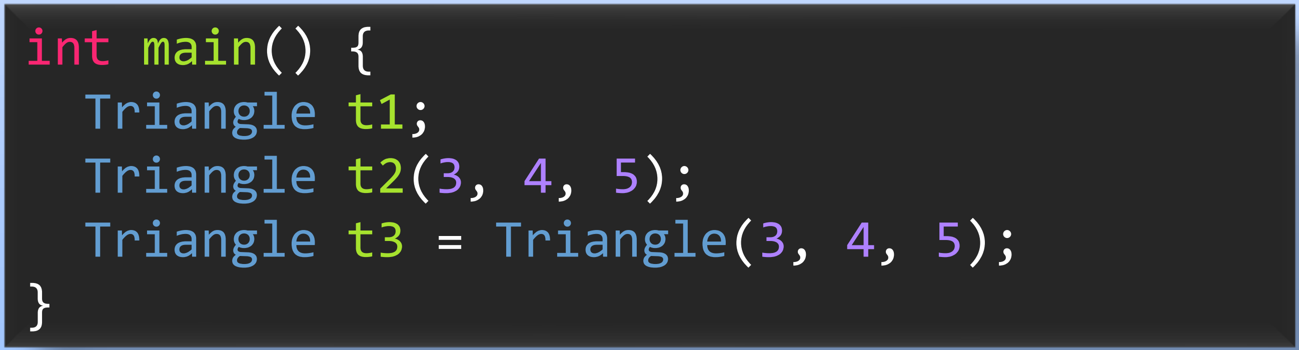
Is it OK for `isSame` to use private member variables from some other `Triangle`?

A) Yes

B) No

Constructors

- Whenever you create an object of class type, a **constructor** for that class is called on the object to initialize it. **Always**[1].
- A constructor is basically a function, but you don't call it yourself – the compiler does it automatically.
- All of these use a `Triangle` constructor:



```
int main() {
  Triangle t1;
  Triangle t2(3, 4, 5);
  Triangle t3 = Triangle(3, 4, 5);
}
```

[1] C-style structs can be initialized with an initializer list, as we've seen before, but C++ style ADTs always use constructors.

Defining Constructors

```
class Triangle {
private:
          le a;
          le b;
  double c;

public:

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {

    // nothing to do in body
  }

};
```

```
int main() {
  Triangle t2(3, 4, 5);
}
```

Same "name" as the class.

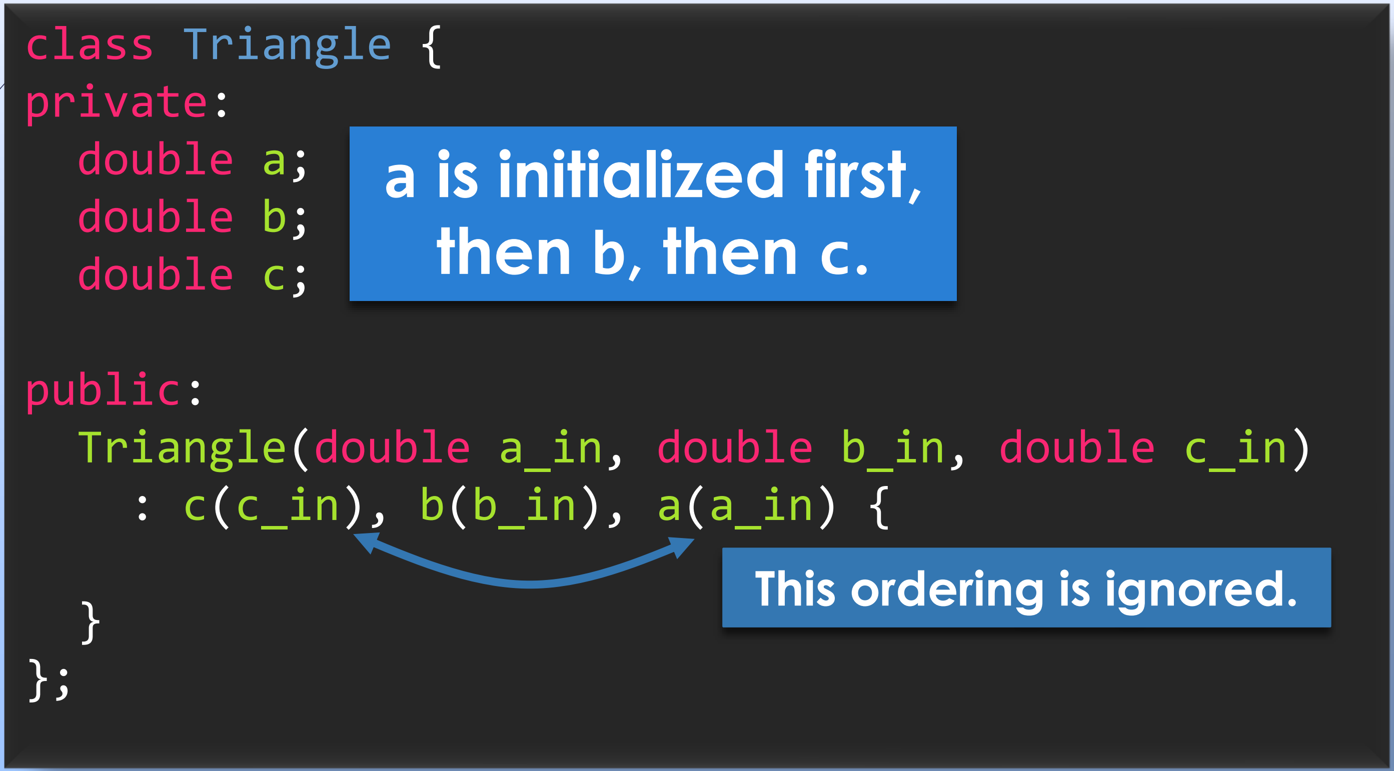
Parameters receive arguments provided to the initializers.

Constructors are usually public.

A *member initializer list* is a special syntax for initializing members in a constructor.

Member Initializer Lists

- Warning! The order of initialization depends on the declaration order.
  - NOT the order of the member initializer list.



```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : c(c_in), b(b_in), a(a_in) {

  }
};
```

a is initialized first, then b, then c.

This ordering is ignored.

Multiple Constructors

- A class may have several different constructors.

```
class Triangle {
private:
  double a, b, c;

public:

  Triangle()
    : a(1), b(1), c(1) { }

  Triangle(double side)
    : a(side), b(side), c(side) { }

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) { }
};
```

A "default" constructor.

```
int main() {
  Triangle t1;
  Triangle t2(10);
  Triangle t3(3, 4, 5);

  Triangle t4(3, 4);
}
```

Error: No matching constructor.

Exercise

```
class Coffee {
private:
  int creams;
  int sugars;
  bool isDecaf;

public:
  // Regular coffee with creams/sugars
  Coffee(int creams, int sugars);

  // This ctor can specify regular/decaf
  Coffee(int creams, int sugars,
         bool isDecaf);

  void addCream();

  void addSugar();

  void print() const;
};
```

Question

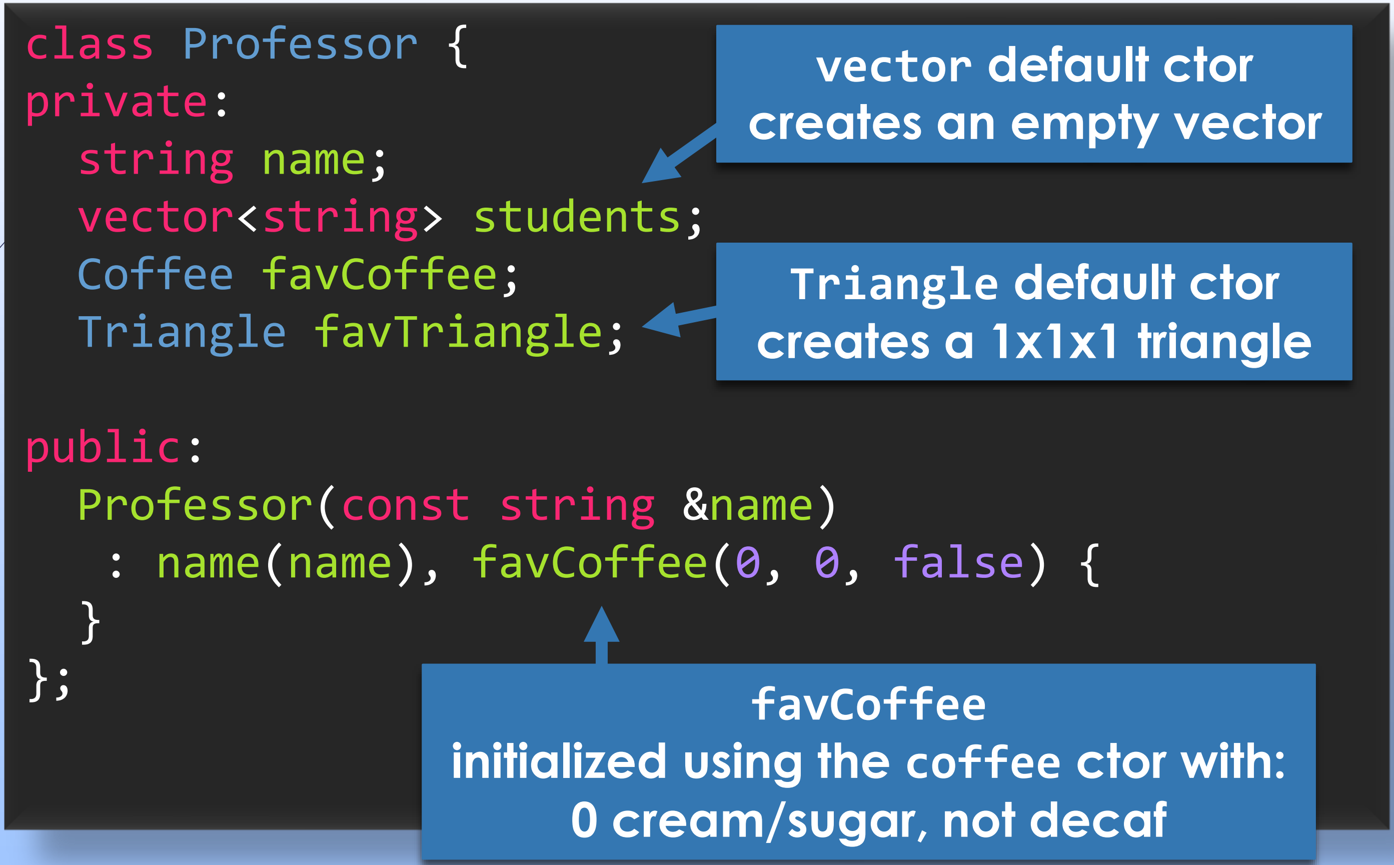
Which snippet does NOT have a compile error?

A B C D E

```
 int main() {

    Coffee c1;
A   c1.addCream();
    c1.print();

    Coffee c2(2, 2);
B   if (c2.isDecaf) {
      c2.print();
    }

    Coffee c3(2, 2, false);
    const Coffee &c3_r = c3;

C   c3.print();
    c3_r.print();

D   c3.addCream();
    c3_r.addCream();

    Coffee c4(true);
E   c4.addSugar();
    c4.print();
 }
```

Classes as Members

- Members are default-initialized if left out of the member-initializer list for a constructor.



```
class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;

public:
  Professor(const string &name)
   : name(name), favCoffee(0, 0, false) {
  }
};
```

vector default ctor creates an empty vector

Triangle default ctor creates a 1x1x1 triangle

favCoffee
initialized using the coffee ctor with:
0 cream/sugar, not decaf

Exercise

Question

How many of these ctors compile successfully?

A) 0 B) 1 C) 2 D) 3 E) 4

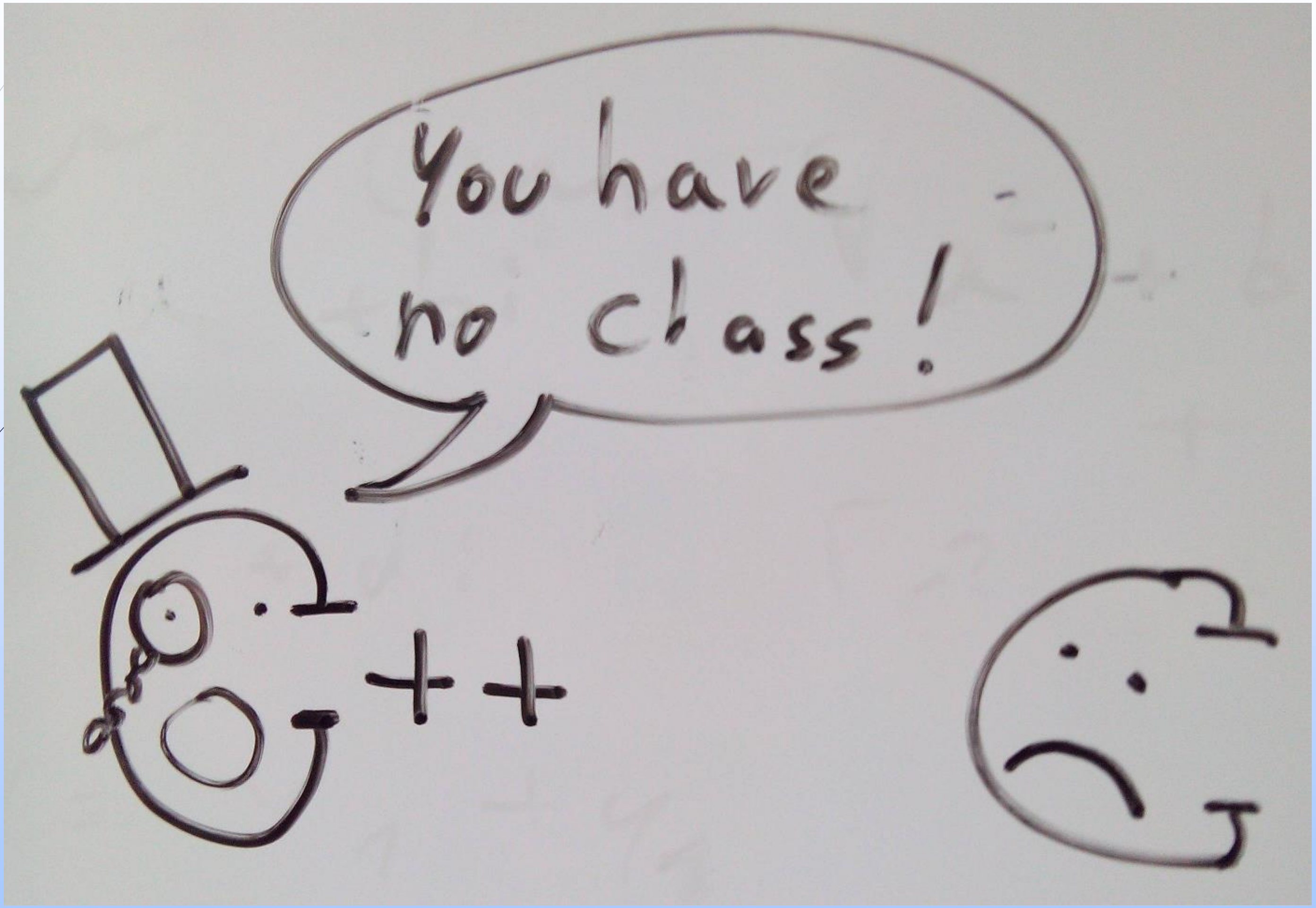
```
class Coffee {
public:
  Coffee(int creams, int sugars);
  Coffee(int creams, int sugars,
         bool isDecaf);
};
```

```
class Triangle {
public:
  Triangle();
  Triangle(double side);
  Triangle(double a_in, double b_in,
           double c_in);
};
```

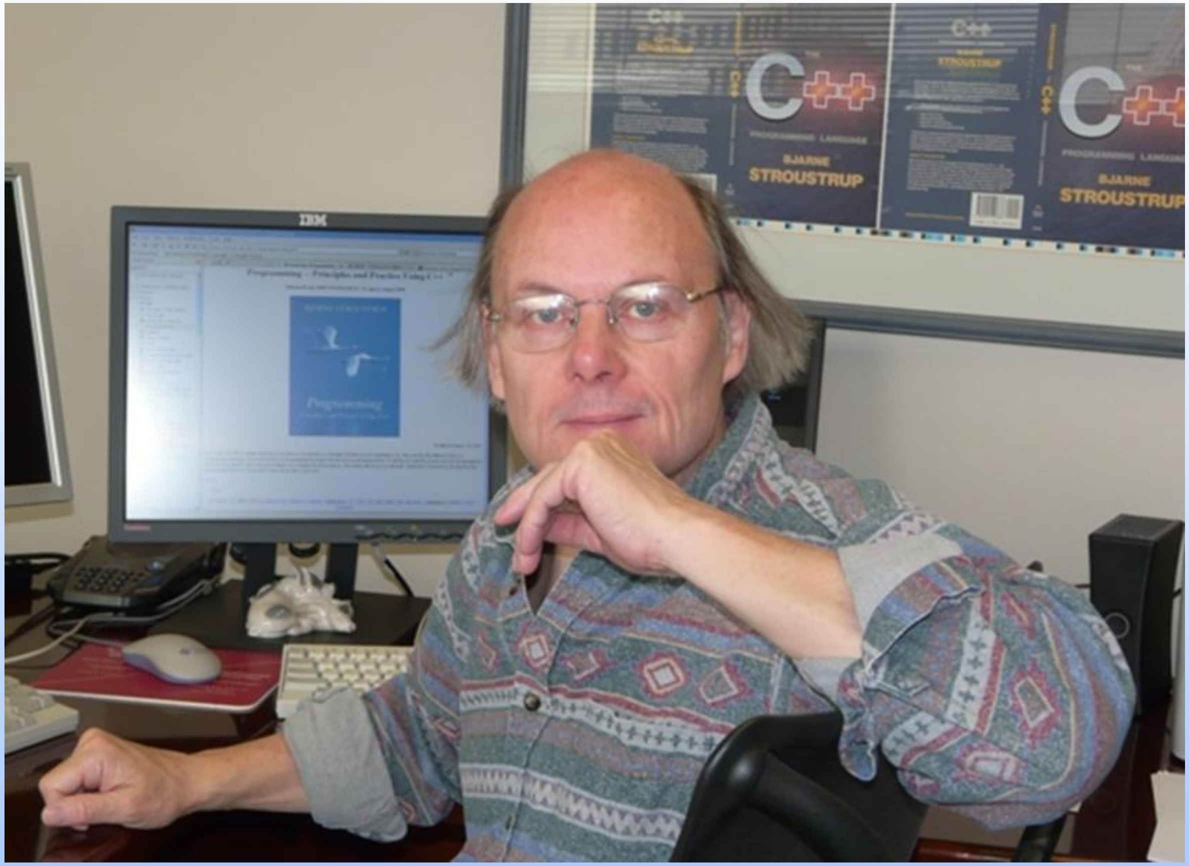
```
class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;
  …
```

```
Professor(const string &name)
 : name(name) { }


Professor(int creams, int sugars)
 : favCoffee(creams, sugars) { }


Professor(const string &name,
          const string &student)
 : name(name) {
  students.push_back(student);
}

Professor(const Coffee &coffee)
 : name("Laura"),
   favCoffee(coffee),
   favTriangle(3, 5) { }
```

Break Time!

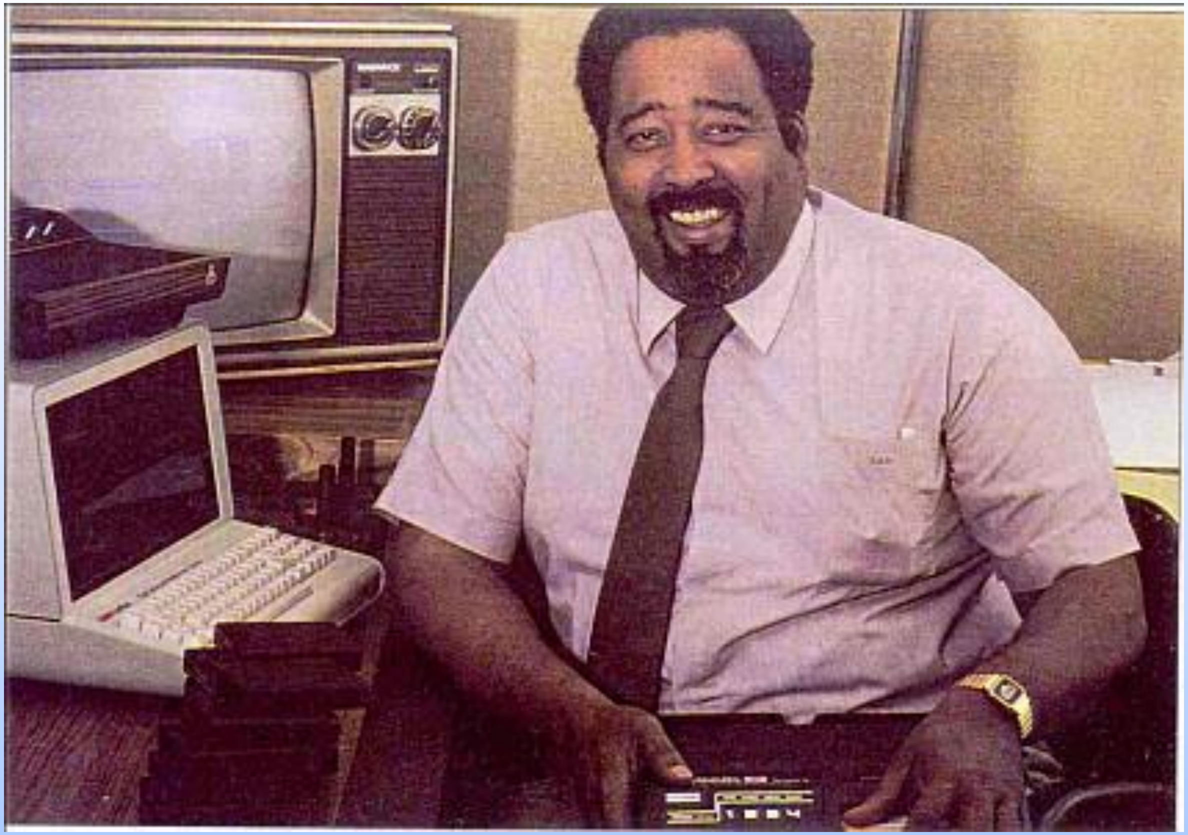


Bjarne Stroustrup



Creator of the C++ Programming Language

Jerry Lawson



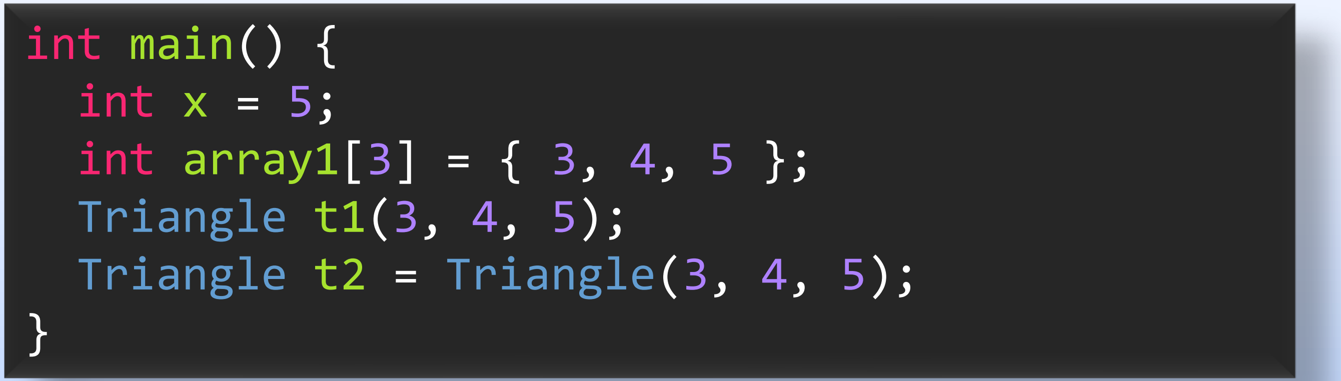
Inventor of the Video Game Cartridge

Break Time

We’ll start again in one minute.

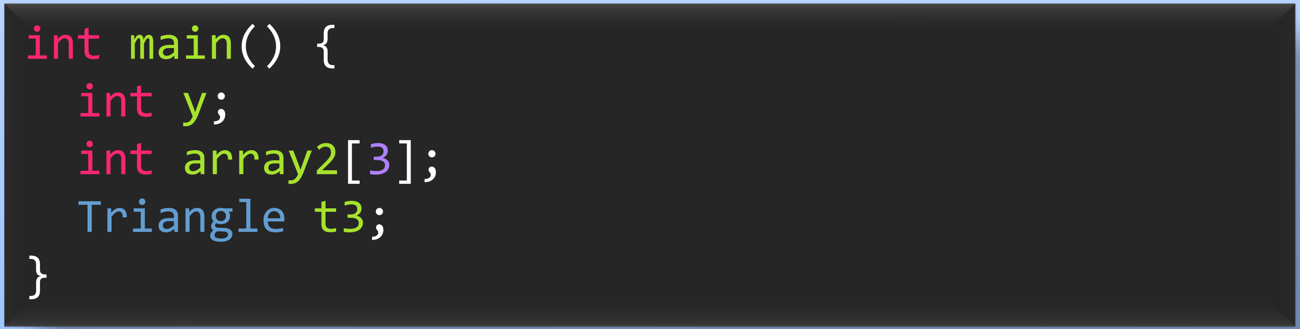
Initialization

- Every object in C++ is initialized upon creation
- Objects can be explicitly initialized



```
int main() {
  int x = 5;
  int array1[3] = { 3, 4, 5 };
  Triangle t1(3, 4, 5);
  Triangle t2 = Triangle(3, 4, 5);
}
```

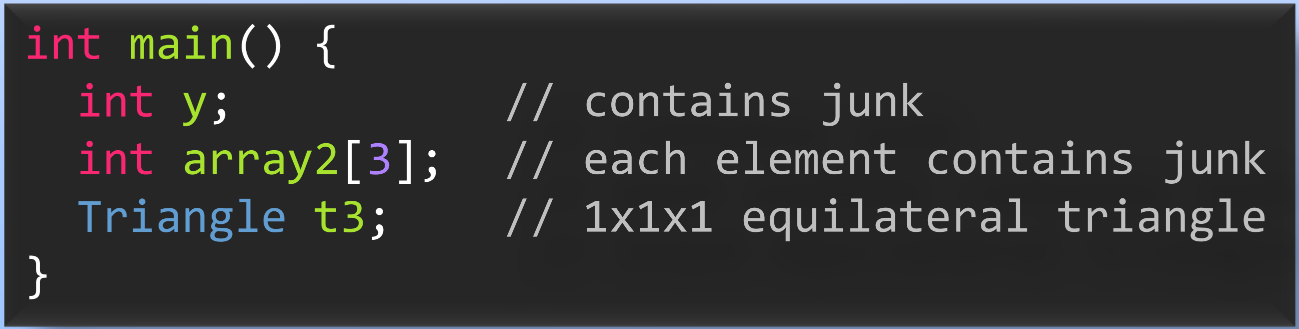
- Objects can also be *default initialized*



```
int main() {
  int y;
  int array2[3];
  Triangle t3;
}
```

Default Initialization

- Objects that are not explicitly initialized are **default initialized**
- Atomic objects (`int`, `double`, `bool`, `char`, pointers) are default initialized by doing nothing
  - They retain whatever value was previously there in memory (junk)
- Array objects are default initialized by default initializing each element
- Compound (i.e. class-type) objects are default initialized by calling the default constructor



```
int main() {
  int y;             // contains junk
  int array2[3];     // each element contains junk
  Triangle t3;       // 1x1x1 equilateral triangle
}
```

Exercise: Default Initialization Syntax

- Line A creates a default-initialized 1x1x1 triangle.
- **What does line B do?**

```
class Triangle {
private:
  double a, b, c;

public:

  Triangle()
    : a(1), b(1), c(1) { }

};
```

```
int main() {
 A Triangle t1;
 B Triangle t2();
}
```

Question

A) It does the same thing as line A.

B) The () syntax can't be used in declarations, so this doesn't compile.

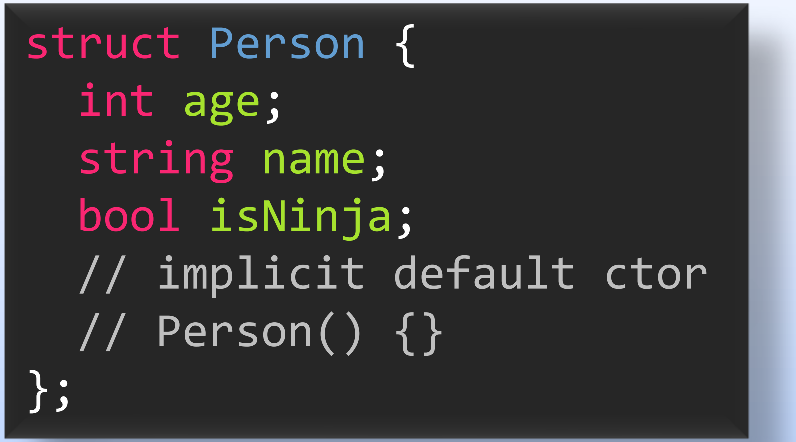
C) t2 is created as a triangle, but initialized with memory junk.

D) t2 is declared as a function that returns a Triangle.

E) It calls the constructor as a function, but doesn't create a Triangle object.

The Implicit Default Constructor

- If you don’t define **any** constructors, the compiler provides a default constructor for you.

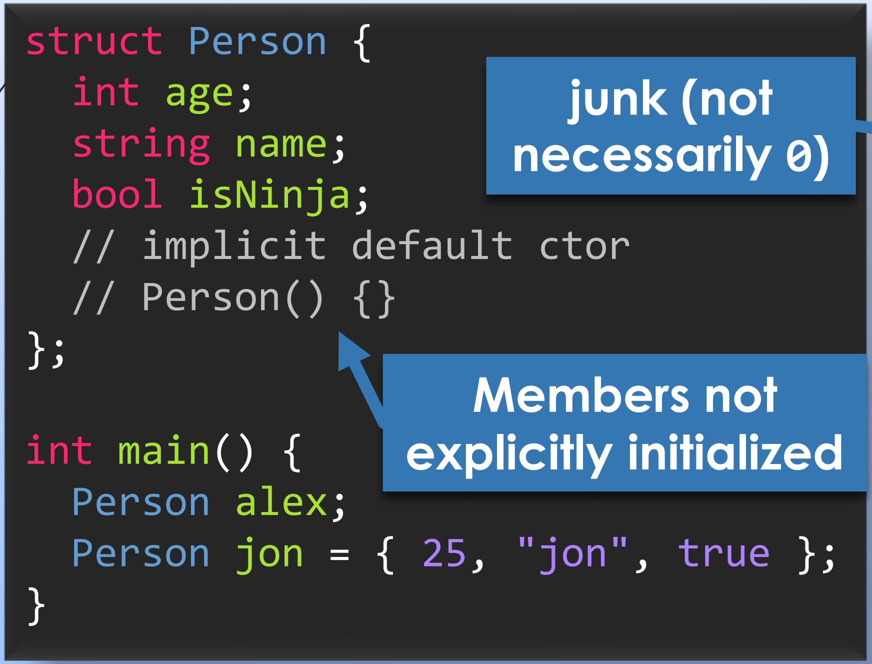


```
struct Person {
  int age;
  string name;
  bool isNinja;
  // implicit default ctor
  // Person() {}
};
```

- If you define **any** constructors, the compiler **doesn’t** give you a default one automatically. (And if you don't write it, there is no default ctor.)

Default Initialization of Members

- Members of compound objects are default initialized if not explicitly initialized

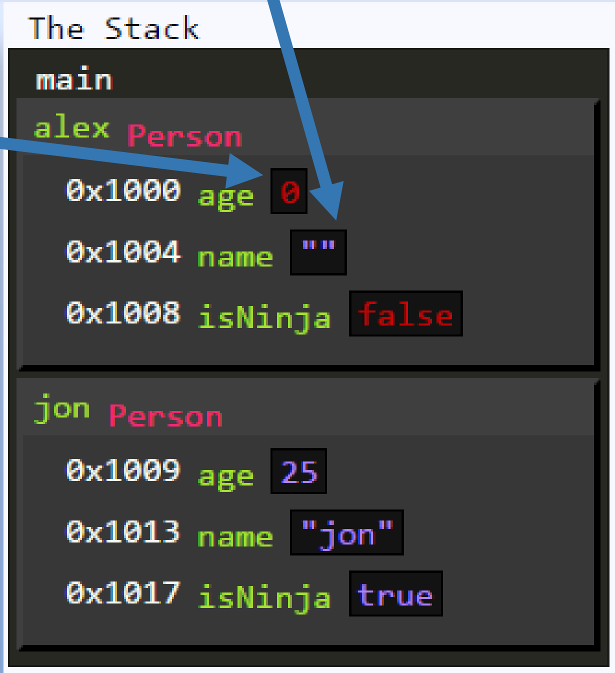


```
struct Person {
  int age;
  string name;
  bool isNinja;
  // implicit default ctor
  // Person() {}
};

int main() {
  Person alex;
  Person jon = { 25, "jon", true };
}
```

junk (not necessarily 0)

Members not explicitly initialized

string default constructor makes it empty



```
The Stack
main
alex Person
  0x1000 age     0
  0x1004 name    ""
  0x1008 isNinja false
jon Person
  0x1009 age     25
  0x1013 name    "jon"
  0x1017 isNinja true
```

struct vs. class

- In the C++ language, the only difference between the `struct` and `class` keywords is the default access level for members.
  - `struct` – `public` by default
  - `class` – `private` by default

```
struct Triangle {
  double a;
  double b;
  double c;

  ...
};
```

a, b, c are `public`

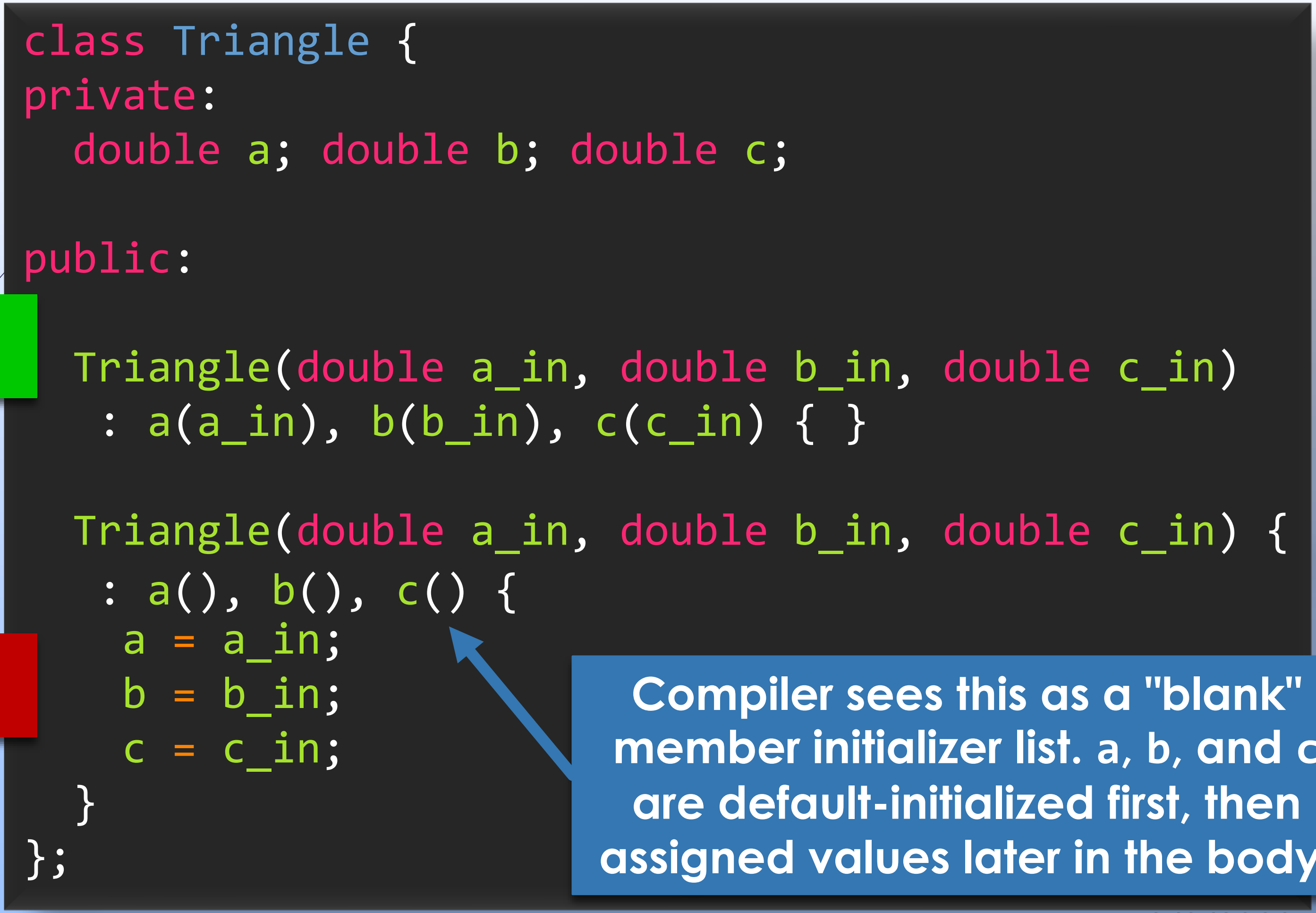
```
class Triangle {
  double a;
  double b;
  double c;

  ...
};
```

a, b, c are `private`

- However, **by convention** we use structs and classes very differently!

Member Initializer Lists

- ALWAYS use a member initializer list if you can.



```
class Triangle {
private:
  double a; double b; double c;

public:

  Triangle(double a_in, double b_in, double c_in)
   : a(a_in), b(b_in), c(c_in) { }

  Triangle(double a_in, double b_in, double c_in) {
   : a(), b(), c() {
    a = a_in;
    b = b_in;
    c = c_in;
  }
};
```

DO

DON'T

Compiler sees this as a "blank" member initializer list. a, b, and c are default-initialized first, then assigned values later in the body.

Member Initializer Lists

- ALWAYS use a member initializer list if you can.

```
class Professor {
private:
  string name;
  vector<string> students;
  Coffee favCoffee;
  Triangle favTriangle;

public:
   Professor(int creams, int sugars)
    : favCoffee(creams, sugars) { }

   Professor(int creams, int sugars) {

     favCoffee = Coffee(creams, sugars);
   }
};
```

DO

DON'T

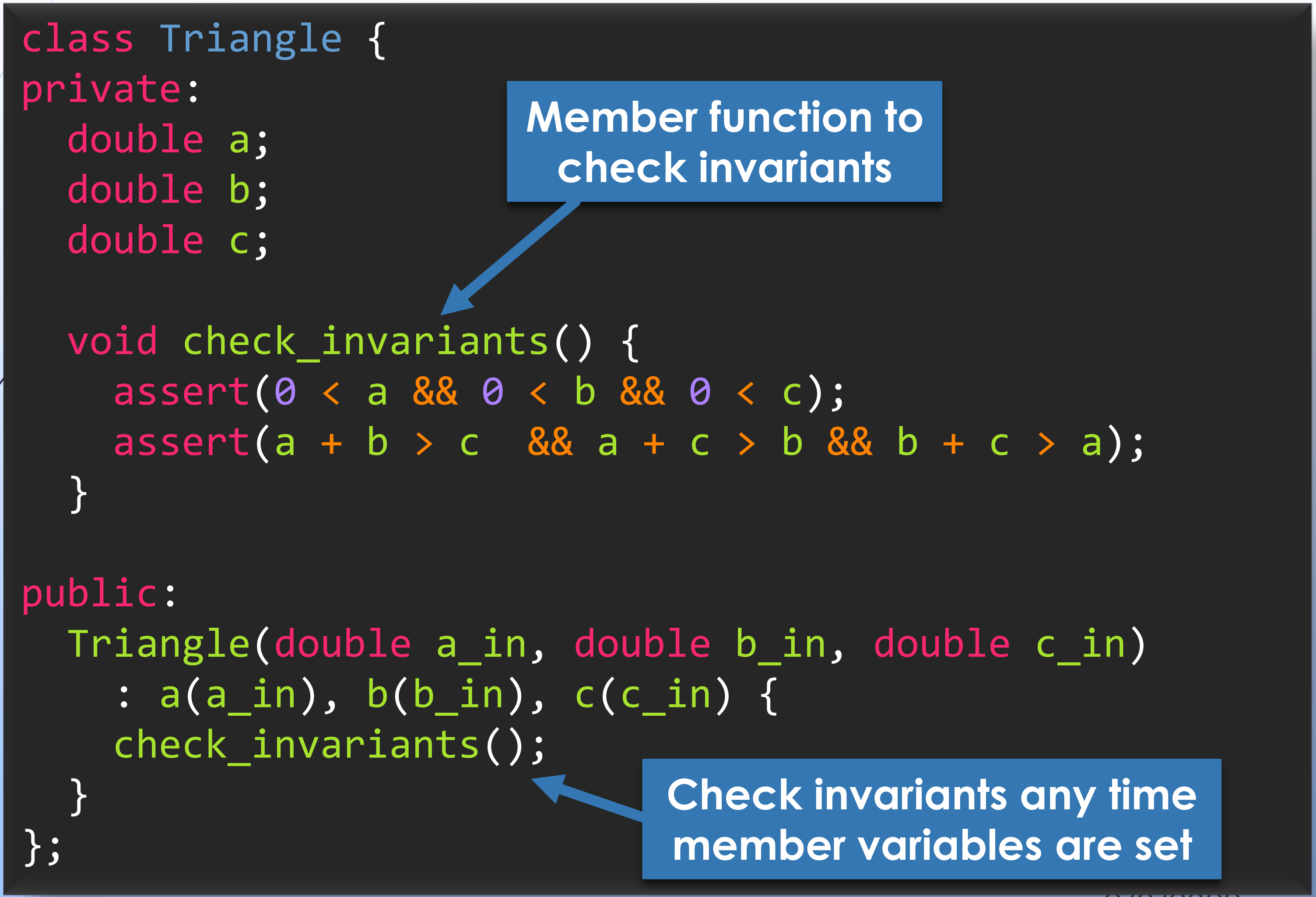
Error: Compiler attempts to default construct favCoffee before the body of the ctor.

Review: Representation Invariants

- A problem for compound types…
  - Some combinations of member values don’t make sense together.
- We use **representation invariants** to express the conditions for a **valid** compound object.
- For Triangle:

Positive Edge Lengths	Triangle Inequality
`0 < a`	`a + b > c`
`0 < b`	`a + c > b`
`0 < c`	`b + c > a`

Check Invariants



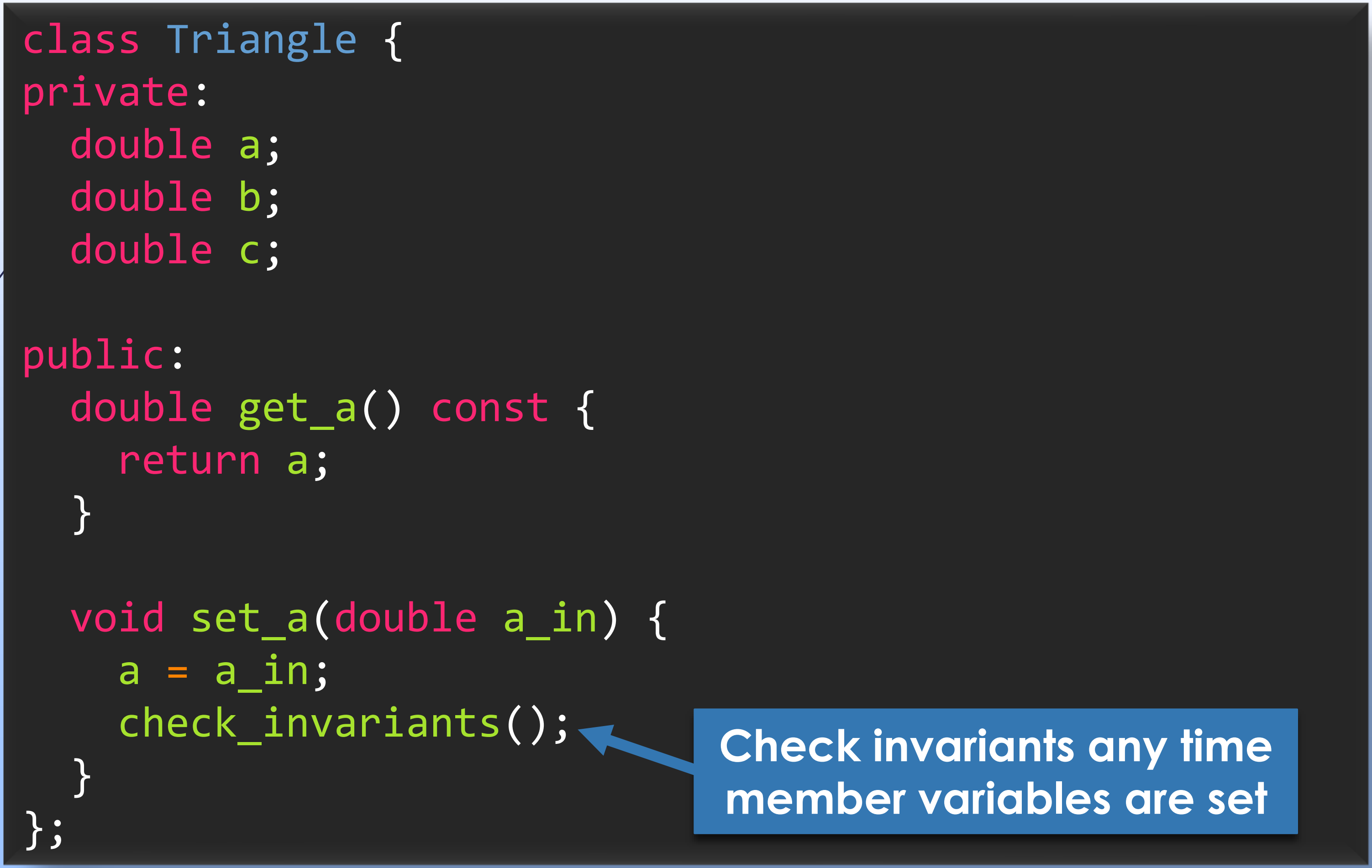
```
class Triangle {
private:
  double a;
  double b;
  double c;

  void check_invariants() {
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  && a + c > b && b + c > a);
  }

public:
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {
    check_invariants();
  }
};
```

Member function to check invariants

Check invariants any time member variables are set

Get and Set Functions

- Some classes provide functions to get and set private member variables



```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  double get_a() const {
    return a;
  }

  void set_a(double a_in) {
    a = a_in;
    check_invariants();
  }
};
```

Check invariants any time member variables are set

Good Abstraction Design

- Encapsulation
  - C++ groups data and behavior together in a class.
  - It gives us mechanisms to protect representation invariants. (access control, constructors)
- Separate **interface** from **implementation.**
  - Work only with the interface, and "hide" away the implementation.
  - Avoid improper dependencies on the implementation.

C-Style Information Hiding

Triangle.h

Interface

What a Triangle does.

```
struct Triangle {
  double a, b, c;
};


double Triangle_perimeter(Triangle const *tri);
Void Triangle_scale(Triangle *tri, double s);
```

Triangle.cpp

Implementation

Details of how it does it.

```
#include "Triangle.h"



double Triangle_perimeter(Triangle const *tri) {
  return tri->a + tri->b + tri->c;
}
void Triangle_scale(Triangle *tri, double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

Information Hiding in C++

Triangle.h
Interface

- What a Triangle does.

```
class Triangle {
public:
  Triangle();
  Triangle(double a_in,
           double b_in,
           double c_in);
  double area() const;
  double perimeter() const;
  void scale(double s);

private:
  double a, b, c;
};
```

Private members are implementation details, so they should be at the bottom.

Triangle.cpp
Implementation

- Details of how it does it.

```
#include "Triangle.h"

Triangle::Triangle(double a_in,
  double b_in, double c_in)
  : a(a_in), b(b_in), c(c_in) { }



void Triangle::scale(double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

The scope resolution operator (::) allows us to refer to the member function from outside.

Testing a C++ ADT

```
#include "Triangle.h"
#include "unit_test_framework.h"

TEST(test_triangle_basic) {
  Triangle t(3, 4, 5);
  ASSERT_EQUAL(t.area(), 6);
  ASSERT_EQUAL(t.get_a(), 3);
  t.set_a(4);
  ASSERT_EQUAL(t.get_a(), 4);
}

TEST_MAIN()
```

C++ forces you to respect the interface