EXAMPLE 8: ~~STOP USING `malloc`/`free`~~

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

WHY? BECAUSE `malloc/free` ONLY ASSIGN MEMORY

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

USING THEM IN C++ WILL HAVE TERRIBLE TERRIBLE CONSEQUENCES

REFRESHER

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME **CERTAIN** THAT NO CONSTRUCTOR IS CALLED.

RULE #1: NEVER USE malloc/ free AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME CERTAIN THAT NO CONSTRUCTOR IS CALLED.

BECAUSE WE HAVE A PRINT STATEMENT INSIDE EACH CONSTRUCTOR..

```
ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart
  {
    cout << "Inside the 2-argument constructor" << endl;
  }
  ComplexNumber(const ComplexNumber& rhs) : realPart(rhs.realPa
  {
    cout << "Inside the copy constructor" << endl;
```

RULE #1: NEVER USE `malloc/ free` AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME CERTAIN THAT NO CONSTRUCTOR IS CALLED.

BECAUSE WE HAVE A PRINT STATEMENT INSIDE EACH CONSTRUCTOR..

```
ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart
  {
    cout << "Inside the 2-argument constructor" << endl;
  }
  ComplexNumber(const ComplexNumber& rhs) : realPart(rhs.realPa
  {
    cout << "Inside the copy constructor" << endl;
```

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME **CERTAIN** THAT NO CONSTRUCTOR IS CALLED.

BECAUSE WE HAVE A PRINT STATEMENT INSIDE EACH CONSTRUCTOR..

```
ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart
  {
    cout << "Inside the 2-argument constructor" << endl;
  }
  ComplexNumber(const ComplexNumber& rhs) : realPart(rhs.realPa
  {
    cout << "Inside the copy constructor" << endl;
```

RULE #1: NEVER USE malloc/ free AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME CERTAIN THAT NO CONSTRUCTOR IS CALLED.

BECAUSE WE HAVE A PRINT STATEMENT INSIDE EACH CONSTRUCTOR..

```
ComplexNumber() : realPart(0.0),complexPart(0.0)
  {
    cout << "No arg-constructor called" << endl;
  }
  ComplexNumber(double c, double r) : realPart(r) , complexPart
  {
    cout << "Inside the 2-argument constructor" << endl;
  }
  ComplexNumber(const ComplexNumber& rhs) : realPart(rhs.realPa
  {
    cout << "Inside the copy constructor" << endl;
```

RULE #1: NEVER USE malloc/ free AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

CREATE A DYNAMICALLY ALLOCATED OBJECT..

```
ComplexNumber * cDynamic = (ComplexNumber*)malloc(sizeof(ComplexNumber));
cout << "Printing out dynamically allocated object" << endl;
cDynamic->print();

free(cDynamic);
cout<<"Okey-dokey! All done!"<<endl;
```

WE RUN THIS CODE, AND BECOME CERTAIN THAT NO CONSTRUCTOR IS CALLED.

```
ComplexNumber() : realPart(0.0),complexPart(0.
  {
    cout << "No arg-constructor called" << end
  }
  ComplexNumber(double c, double r) : realPart
  {
    cout << "Inside the 2-argument constructor
  }
  ComplexNumber(const ComplexNumber& rhs) : re
  {
    cout << "Inside the copy constructor" << e
  }
```

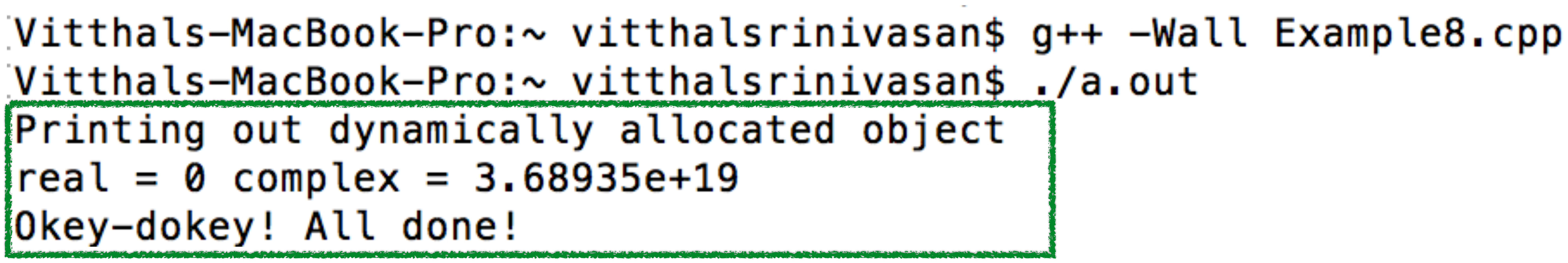
BUT NOTHING IS PRINTED WHEN WE RUN OUR CODE!

BECAUSE WE HAVE A PRINT STATEMENT INSIDE EACH CONSTRUCTOR..

RULE #1: NEVER USE `malloc`/ `free` AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

BUT NOTHING IS PRINTED WHEN WE RUN OUR CODE!



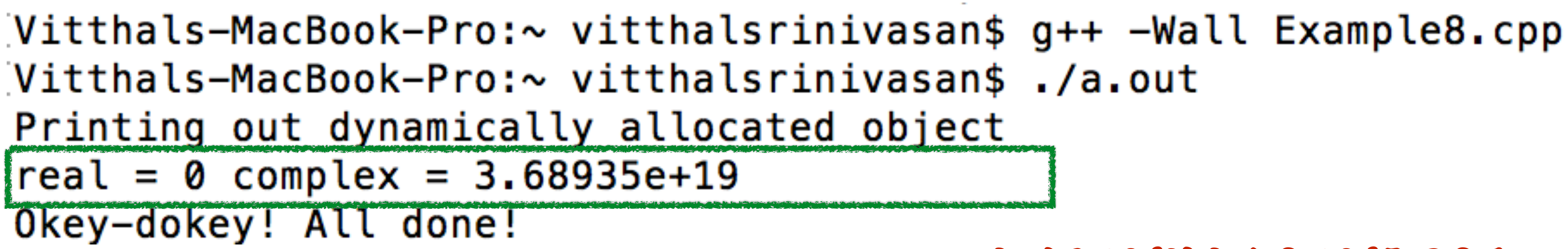
```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example8.cpp
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Printing out dynamically allocated object
real = 0 complex = 3.68935e+19
Okey-dokey! All done!
```

NO MESSAGE FROM CONSTRUCTOR..

RULE #1: NEVER USE `malloc/ free` AGAIN. EVER.

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

BUT NOTHING IS PRINTED WHEN WE RUN OUR CODE!



```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example8.cpp
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Printing out dynamically allocated object
real = 0 complex = 3.68935e+19
Okey-dokey! All done!
```

AND WHAT'S WORSE - A GARBAGE VALUE INSIDE OUR MEMBER VARIABLE!

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

WHY? BECAUSE `malloc`/`free` ONLY ASSIGN MEMORY

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

USING THEM IN C++ WILL HAVE TERRIBLE TERRIBLE CONSEQUENCES

REFRESHER