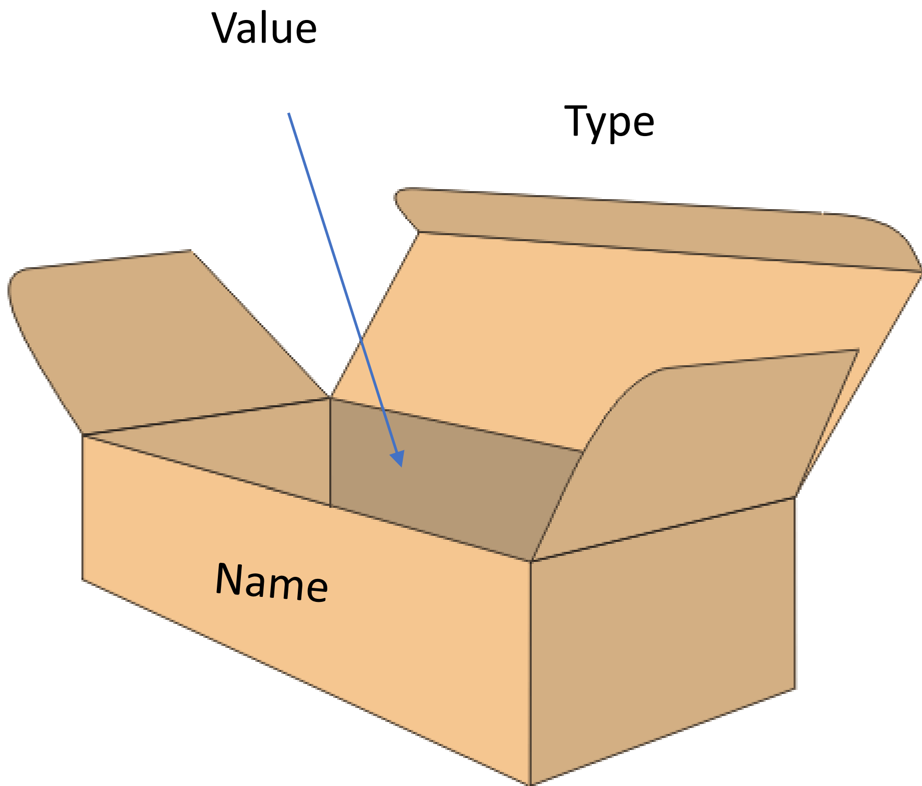
TDDE18 & 726G77

Classes

Variable

- Fundamental (also called built-in types)
  - Stores a value of a fundamental type, nothing more
- Object
  - Stores values tied to an derived type (struct, class)
  - Operations associated to the type are provided
  - More about classes later in the course
- Pointer
  - Stores the address of some other variable
  - More about pointers in the course

Variable



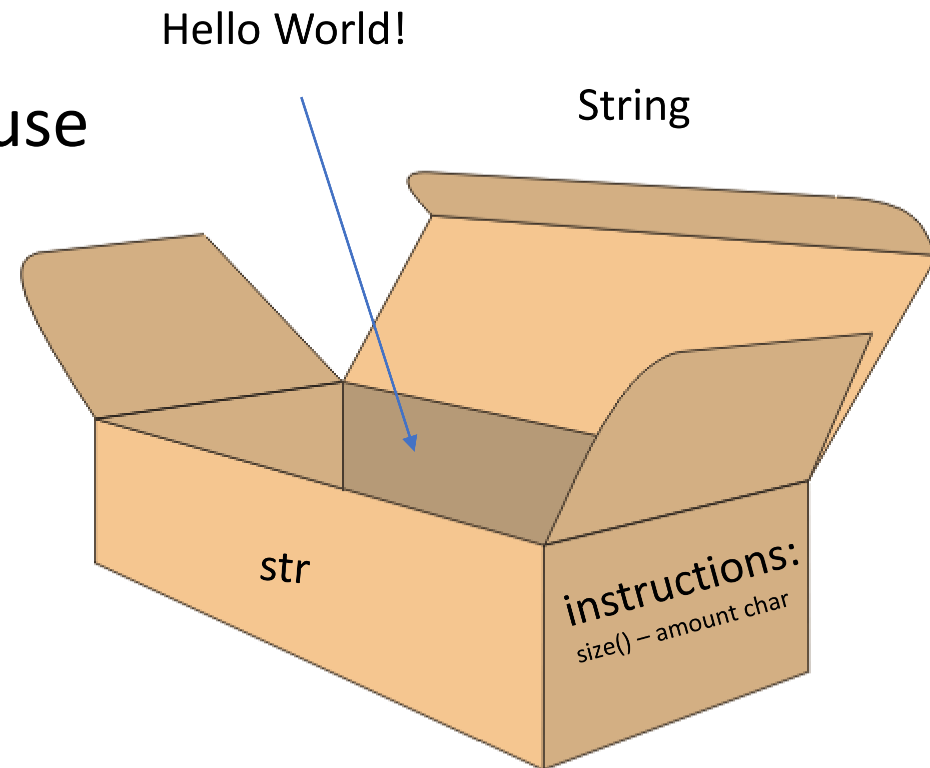
Class

- Data members – store values

```
string str{“Hello World!”};
```

- Member functions – operations available to use

```
str.size();
```



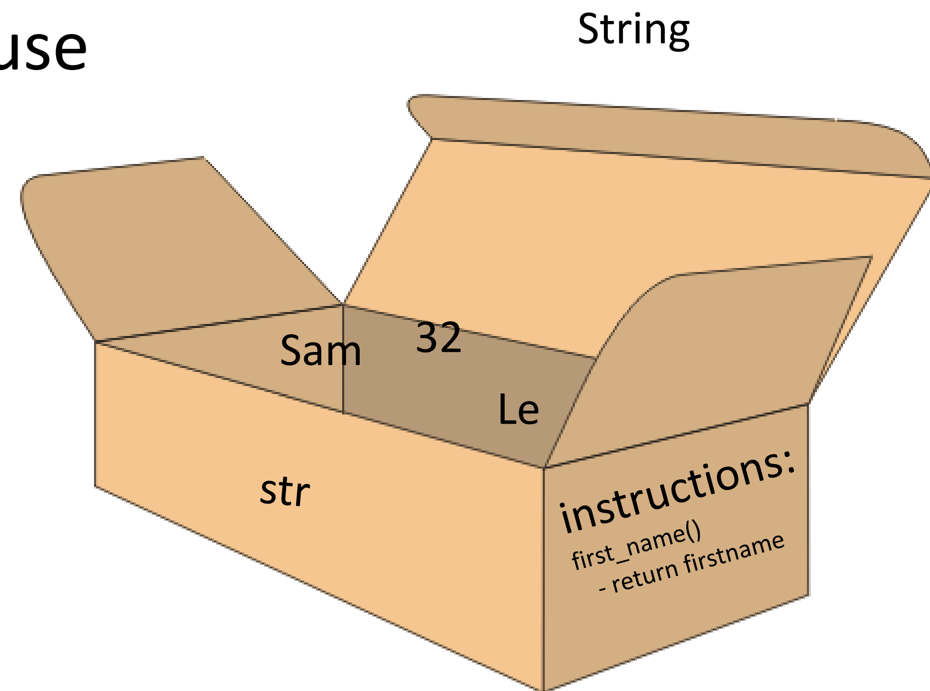
Class – the blueprint of an object

- Data members – store values

```
Person p{“Sam”, “Le”, 32};
```

- Member functions – operations available to use

```
p.first_name();
```



Class syntax – header file

```
// header file guard protect from multiple inclusion
#ifndef _CLASS-NAME_H_
#define _CLASS-NAME_H_
// DO NOT use namespaces here, it may not be
// wanted in programs including this file
// prefix std:: on standard types instead
// names in italic are customizable
class class-name
{
public:
    class-name(); // constructor (Initiator)
    // member functions (methods in Java)
    return-type operation(parameter-list);
private:
    // member variables
    data-type property;
};
#endif
```

Class syntax – implementation file

```
#include "class-name.h"
// Constructor (Initiator)
class-name::class-name()
{
    // implementation
}
// Member function
return-type
class-name::operation(parameter-list)
{
    // implementation
}
```

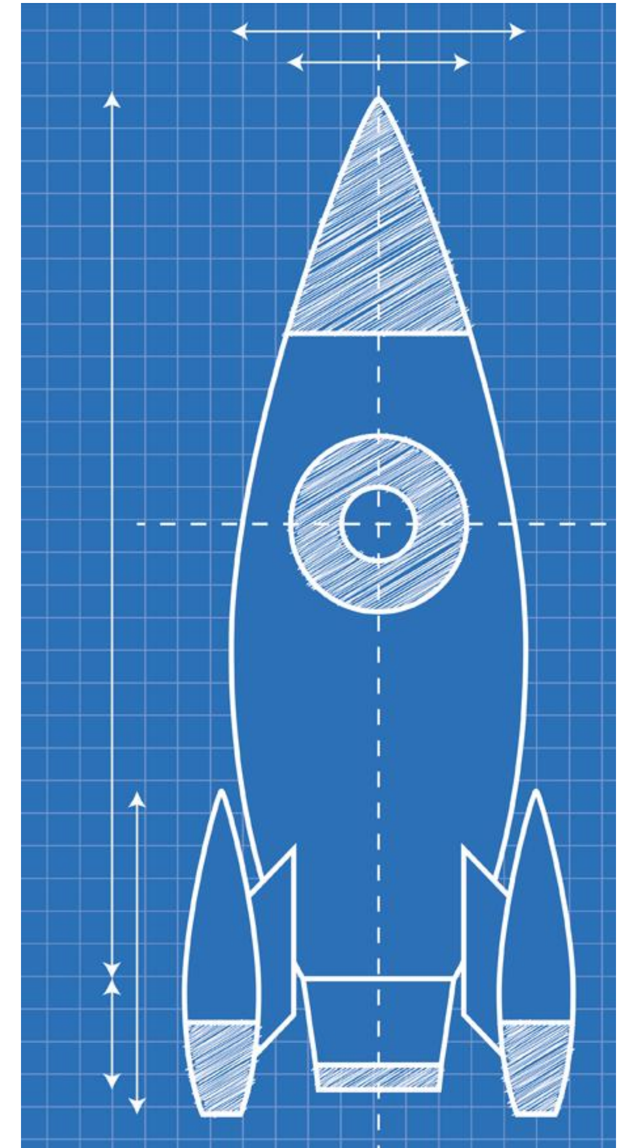
Class

- Provide language support for object orientation
- Having a single purpose, responsibility
- Consist of private member variables and public interface methods
- Can only be manipulated through a well defined interface
- Constructors and interface enables the programmer to depend on always known and correct internal state
- Operators, constructors and destructors allow for easy management

Class vs Instance

- A class only describe the layout. It does not create any data in memory. It’s a description of a data-type with operations ”embedded”.

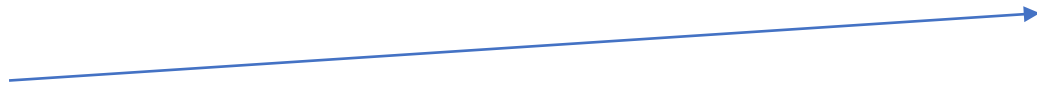
```
class Rocket {
public:
      void fly();
      bool finished;
private:
      int height;
};
```

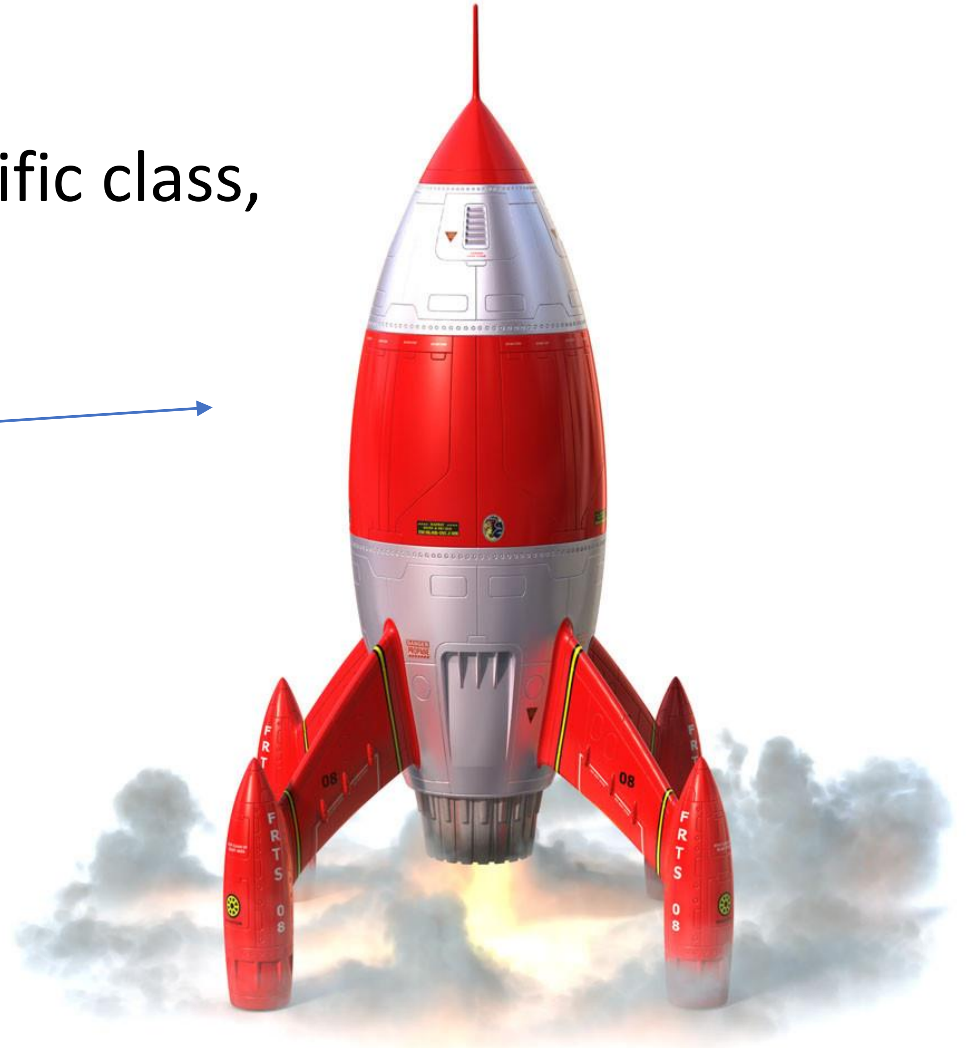


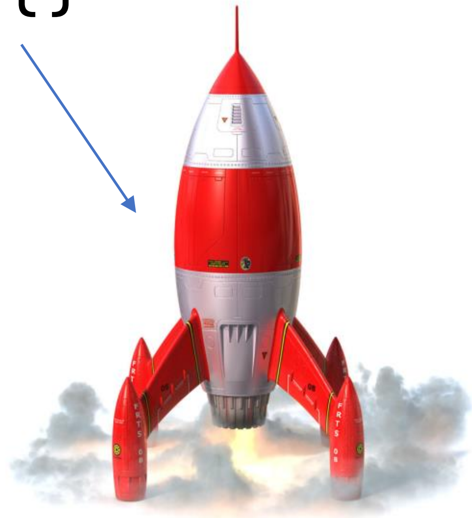
Class vs Instance

- An instance is a variable created of a specific class, an object. You can create many.

```
Rocket r{};
Rocket s{}
```







Class declaration

```
// h-file
class Robot {
public:
  void fly();
  bool finished;
private:
  int height;
};
```

```
// cc-file
void Robot::fly() {
  cout << “I’m flying” << endl;
}
```

Accessing members

- An object variable allow you to access member functions (operations) and member variables of that instance. You use the dot operator

```
// Class definition
class Rocket {
public:
     void fly();
     bool finished;
private:
     int height;
};
```

```
// Access member functions
Rocket r{};
r.finished = true;
r.fly();
```

Accessing members

- Accessing a member inside a class does not require you to tell the compiler which instance you are referring to.

```
// Outside of class
int main() {
   Rocket r{};
   r.finished = true;
}
```

```
// Inside the class
class Rocket {
public:
   void fly() {
      finished = true;
   };
```

The keyword “this”

- Member functions are called “on” an instance and automatically receive that instance to work on, available as the special pointer *this.*

```
void Robot::fly() {
   finished = true;
   cout << ”I’m finished and I can fly” << endl;
}

void Robot::fly() {
   this -> finished = true;
   cout << ”I’m finished and I can fly” << endl;
}
```

Private members

- Private members are only accessible in functions belonging to the same class

```
class Rocket {
public:
   void fly() {
      r.model = “M-3”; //OK
   }
};

int main() {
     Rocket r{};
     r.model = “M-3”; //Error
}
```

Friends

- A class can decide to have friends. Friends can access private members!
- Friends should be avoided at all cost, since it creates high coupling – it makes the two classes highly interdependent.

```
class Rocket {
   ...
   friend bool equals(Rocket r1, Rocket r2);
   ...
};
bool equals(Rocket r1, Rocket r2) {
   return r1.model == r2.model;
}
```

Object lifecycle

- class definition:
  - no object created yet, before birth
- variable definition:
  - object born, memory allocated
  - memory initiated with default values
- variable used...
- variable declaration block ends:
  - memory reclaimed for other variables

Object lifecycle

- class definition:
  - no object created yet, before birth
- variable definition:
  - object born, memory allocated
  - memory initiated with default values
- variable used...
- variable declaration block ends:
  - memory reclaimed for other variables

Constructor

Member functions
Operator functions

Destructor

Lifecycle “hooks”

- Constructor is automatically called when a class variable is defined or allocated
  - have no return value
  - any defined parameters must be specified
- Operators functions are automatically called when variable is used by an operator
  - covered later on
- Destructor is automatically called when a variable goes out of scope or is deleted
  - have neither return value nor parameters

The rocket constructor

```
// h-file
class Rocket {
public:
  Rocket(); // Constructor
  ...
};
```

```
// cc-file
Rocket::Rocket() {
  model = “Unknown model”;
}
```

Using the constructor

- If you define a constructor you must specify all arguments when you create an instance!
- If you do not define a constructor a default constructor that does nothing will be created.
- If you only have private constructors other code can not create instances.

Default constructor

- If you do not define a constructor the compiler will generate a similar default constructor for you.

```
// h-file
class Rocket {
public:
   Rocket(); // Default Constructor
   ...
};
// cc-file
Rocket::Rocket() {
}
```

Constructor Example

```
// h-file
class Rocket {
public:
   Rocket(string m);
   ...
};
// cc-file
Rocket::Rocket(string m) {
   model = m;
}
```

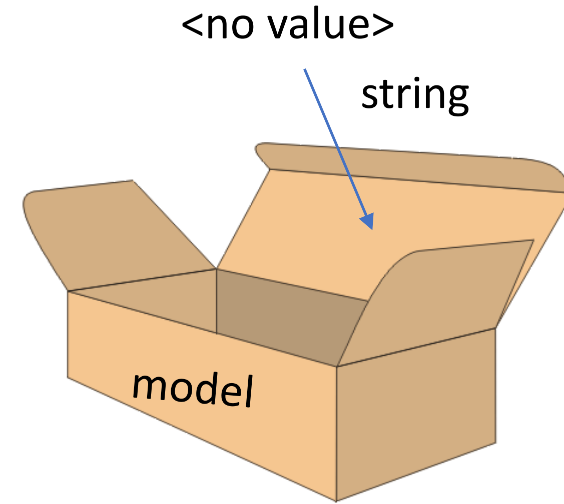
Constructor Example

```
// h-file
class Rocket {
public:
   Rocket(string m);
   ...
};
// cc-file
Rocket::Rocket(string m) {
   model = m;
}
```

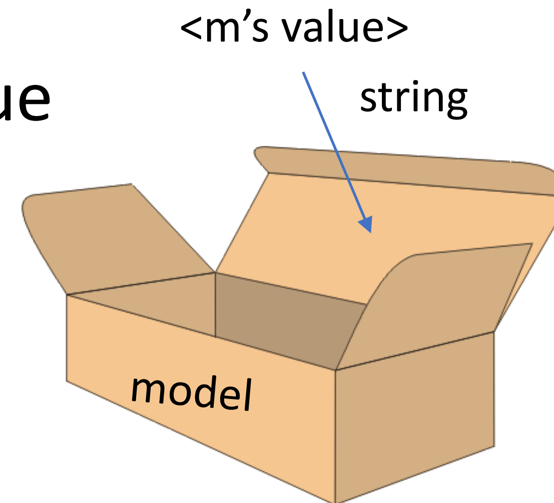
```
// Ok
Rocket r{“M-3”};

// Error no fitting constructor
Rocket s{};
```

Constructor Performance Issue

```
Rocket::Rocket(string m) {
    model = m;
}
```



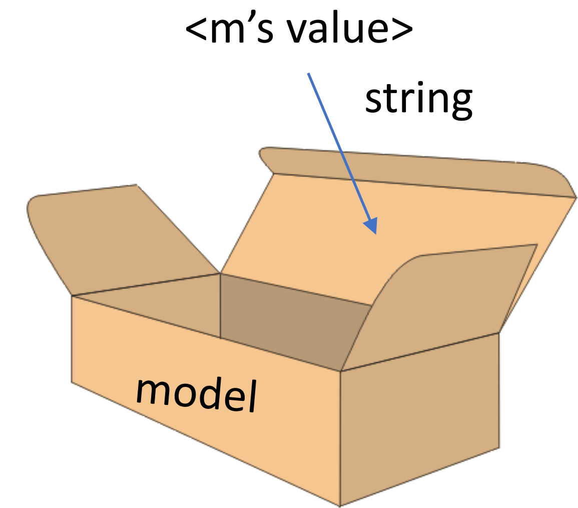
1. Create model variable inside rocket
2. Update model variable with correct value



Constructor Member Initializer List

Robot::Robot(string m) : model{m} {}

Member initializer list specifies the initializers for data members.



Const member variables

- Data members could also be const
- Constant member variable must be initialized in constructor initialization list

```
class Robot {
public:
    ...
    string const model;
};
```

```
Robot::Robot(string m) model{m} {}
```

Reference member variables

- Data members could also be a reference to another variable
- Reference member variables must be initialized in constructor initialization list

```
class Robot {
...
private:
   Person & creator;
};
```

Constructor – Multiple

- Constructor can be overloaded in a similar way as function overloading
- Overloaded constructor have the same name (name of the class) but different number of arguments
- The compiler choose the constructor that fits best with the given input arguments

```
...
   Robot();
   Robot(string m);
   Robot(Person p);
   Robot(Person p, string m);
   etc.
...
```

Constructor delegation

- Many classes have multiple constructors that do similar things
- You could reduce the repetitive code by delegating the work to another constructor

```
Robot::Robot() : Robot{“unknown”} {}
Robot::Robot(string m) : model{m} {}
```

Destructor

- The object calls the destructor when it is about to go out of scope

```
int main() {
  Robot r{};
} // r will call its destructor on this line
```

Destructor

```
// h-file
class Robot {
public:
   ~Robot(); // no return or parameters
   ...
};
// cc-file
Robot::~Robot() { // not useful yet...
   cout << “destructor called” << endl;
}
```

Example class - Money

- Class that represent money
- Have the capacity to hold units (Swedish krona)
- Have the capacity to hold hundreds (Swedish öre)
- Can validate that it have valid (non-negative values) in units and hundreds.

Example class

```
class Money {
public:
   Money();
   Money(int unit);
   Money(int unit, int hundred);
   ~Money();
   void validate();
private:
   int unit;
   int hundred;
};
```

```
Money()
   : Money{0} {}
Money(int unit)
   : Money {0, 0} {}
Money(int unit, int hundred)
   : unit{unit}, hundred{hundred}
{
   validate();
}
void Money::validate() {
  if (unit < 0 || hundred < 0)
...
```