Member Functions

Due this week

- **HW 6**
  - Write solutions in VSCode and paste in **CodeRunner**.
  - Zip your .cpp files and submit on canvas. Check the due date! **No late submissions!!**
- **Quiz 6.** Check the due date! **No late submissions!!**
- **3-2-1 on Friday**
- **Practicum 2 on Monday (all info on Canvas)**

Member Functions

Class

- A class describes a set of objects with the same behavior
- Variables of a class are called objects

- Classes can have:
  - Data members
  - Member functions

Member Functions

Two types:

1. Mutators / setters
2. Accessors / getters

Mutators / Setters

Mutators are member functions that modify the data members

- Increment the value of the counter
- Reset counter value to zero

Accessors / Getters

Accessors are member functions that query a data member(s) of the object, and returns the value(s) to the user

- Get the value (of… value) of the counter

Constructors

- A constructor is a member function that initializes the data members of an object.
- The constructor is automatically called whenever an object is created.

```
Counter my_counter;
```

- (You don’t see the function call nor the definition in the class, it but it’s there.)

Motivation

- By supplying a constructor, by writing our own implementation, you can ensure that all data members are properly set before any member functions act on an object.
- To understand the importance of constructors, consider:

```
Counter my_counter;
my_counter.count();
int cur_val = my_counter.get_value(); // May not be 1
```

- Notice that the programmer forgot to call **reset** before counting.

Constructor Code

- You declare constructor functions in the class definition. There must be **no** return type, not even **void**.
- The name of the constructor must be the same as the class:

```
class Counter
{
public:
   Counter(); // A constructor
...
};
```

- The constructor definition resembles other member functions:

```
Counter::Counter()
{
   value = 0;
}
```

Default Constructors

- If you do not write a constructor for your class, the compiler automatically generates one for you, which does nothing but allocate memory space for the data members.

- The compiler does NOT provide safe initial data values, EXCEPT that `string` members are initialized to “”.

- Default constructors are called when you define an object and do not specify any parameters for the construction.

```
Counter counter1;
```

Parameterized Constructors

- Constructors can have parameters, and can be overloaded :

```
class Counter
{
public:
  // “Default” constructor: Sets value = 0
  Counter();
  // Sets value = initial_count
  Counter(int initial_count);
private:
  int value;
};
```

Overloaded Constructors

- When the same name is used for more than one function, then the functions are called **overloaded**. The compiler determines which to use, based on the parameter list of the call.
- When you construct an object, the compiler chooses the constructor that matches the parameters that you supply:

```
Counter(); // Uses default constructor

Counter(10); // uses parameterized constructor
```


BOA[illegible]OOL?
CODING SCHOOL

Example

We have used the string class, but we didn’t have to deal with how str.substr(6) works, or what str[6] is actually doing.

- We had access to the **public interface** to the string class, and just got to use that
- Protects the class from us accidentally messing it up

A generic class interface

```
class NameOfClass
{
  public:
    // the public interface

  private:
    // the data members
};
```

Designing a class: cash register

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Designing a class: cash register

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

These are function prototypes.
We'll define them later.

What are the data members?

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   int item_count;
   double total_price;
};
```

Always **think carefully** about what the values we might need to access from our class could be!

Member Functions

Two types:

1. Mutators / setters
2. Accessors / getters

Mutators / Setters

Mutators are member functions that modify the data members

- Increment the item count
- Add price to the total bill
- Clear all data members (reset total bill and item count to 0)

Accessors / Getters

Accessors are member functions that query a data member(s) of the object, and returns the value(s) to the user

- Get the total bill
- Get the item count

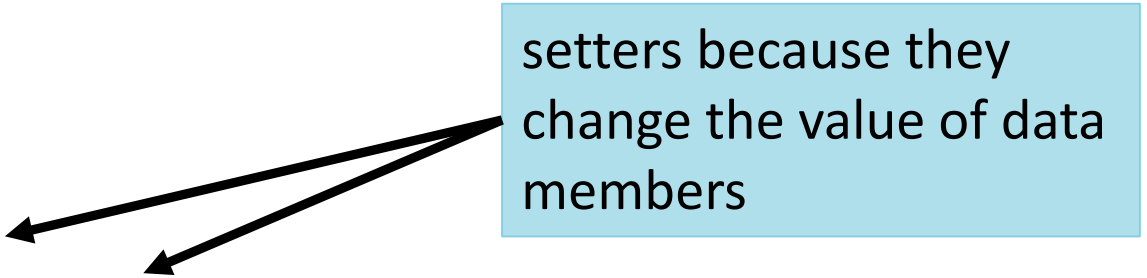
Designing a class: cash register

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Question: Which member functions are getters (accessors) and which are setters (mutators)?

Designing a class: cash register

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```



Question: Which member functions are getters (accessors) and which are setters (mutators)?

Designing a class: cash register

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

setters because they change the value of data members

getters because they simply report the values of data members

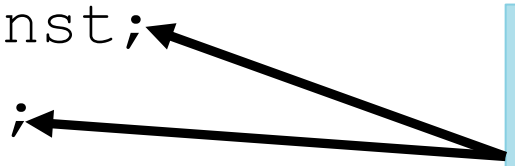
Question: Which member functions are getters (accessors) and which are setters (mutators)?

What is const?

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

What is const?

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```



getters only report the values of data members, and never alter them
→ we declare these functions to be const so they can't mess our stuff up

Dot Notation

You call the member functions by first creating a variable of type **CashRegister** and then using the dot notation:

```
CashRegister register1;
...
register1.clear();
register1.add_item(1.95);
...
int count = register1.get_count();
cout << "Number of items: " << count << endl;
```

Encapsulation

- Every `CashRegister` object has its own copy of these data members

```
CashRegister register1;
CashRegister register2;
… [use setter functions] ...
```

register1
```
item_count = 1
total_price = 1.95
```

register2
```
item_count = 1
total_price = 1.95
```

…..

Encapsulation

The private data members are only accessible via member functions:

- Won't work:

```
CashRegister register1;
… [use setter functions] ...
cout << register1.total_price << endl;
```

Encapsulation

The private data members are only accessible via member functions:

- Won't work:
```
CashRegister register1;
… [use setter functions] ...
cout << register1.total_price << endl;
```

- Will work!
```
CashRegister register1;
… [use setter functions] ...
cout << register1.get_total() << endl;
```

Encapsulation

- You can move data members to the public interface and make it accessible
- DON’T! It is not good practice
  - Will keep things tidier and easy to debug!

Encapsulation

- We might want to change how data members are computed and/or manipulated, but the important details (data members) shouldn't necessarily change.
- Example:
  - We can write the mutator for item_count so it can never be negative
  - On the other hand, if item_count were public, we could just straight up set it to be negative.

The Interface

- The interface should not change even if the details of how they are implemented change.
- A driver switching to an electric car does not need to re-learn how to drive.

Class Implementation

Class Implementation

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   int item_count;
   double total_price;
};
```

Now that we have the interface, we need to actually define the prototypes!
→ start by **implementing the member functions**

Implementing member functions

- Start with the add_item() member function:

```
void add_item( double price ) {
      item_count++; // added an item, so increment item counter
      total_price = total_price + price; // added item price too
}
```

- One more thing to add: as written, there is no connection to the CashRegister class!

Implementing member functions

```
void CashRegister::add_item( double price ) {
      item_count++; // added an item, so increment item counter
      total_price = total_price + price; // added item price too
}
```

- One more thing to add: as written, there is no connection to the CashRegister class!
- so we specify for our member functions:

```
CashRegister::[member function name]
```

Implementing member functions

- We do not need the CashRegister:: declaration when defining the class:

```
class CashRegister {
        public:
        ...
        void add_item( double price );
        …
        private:
        ...
};
void CashRegister::add_item( double price ) {
        item_count++;
        total_price = total_price + price;
}
```

Constructors

- A constructor is a member function that initializes the data members of an object.
- The constructor is automatically called whenever an object is created.

```
CashRegister register1;
```

- (You don’t see the function call nor the definition in the class, it but it’s there.)

Motivation

- By supplying a constructor, by writing our own implementation, you can ensure that all data members are properly set before any member functions act on an object.
- To understand the importance of constructors, consider:

```
CashRegister register1;
register1.add_item(1.95);
int count = register1.get_count(); // May not be 1
```

- Notice that the programmer forgot to call **clear** before adding items.

Constructor Code

- You declare constructor functions in the class definition. There must be **no** return type, not even **void**.
- The name of the constructor must be the same as the class:

```
class CashRegister
{
public:
   CashRegister(); // A constructor
...
};
```

- The constructor definition resembles other member functions:

```
CashRegister::CashRegister()
{
   item_count = 0;
   total_price = 0;
}
```

Default Constructors

- If you do not write a constructor for your class, the compiler automatically generates one for you, which does nothing but allocate memory space for the data members.

- The compiler does NOT provide safe initial data values, EXCEPT that `string` members are initialized to “”.

- Default constructors are called when you define an object and do not specify any parameters for the construction.

```
CashRegister register1;
```

Parameterized Constructors

- Constructors can have parameters, and can be overloaded :

```
class CashRegister
{
public:
   // “Default” constructor: Sets item_count & total_price = 0
   CashRegister();
   // Sets item_count = count and total_price = price
   CashRegister(int count, double price);
private:
   int item_count;
   double total_price;
};
```

Overloaded Constructors

- When the same name is used for more than one function, then the functions are called **overloaded**. The compiler determines which to use, based on the parameter list of the call.
- When you construct an object, the compiler chooses the constructor that matches the parameters that you supply:

```
CashRegister(); // Uses default constructor

CashRegister(10,2.25); // uses parameterized

// constructor  CashRegister(int count, int price)
```

Common Error: Resetting objects

- You cannot call a constructor with dot notation to “reset” an object.

```
CashRegister register1;
...
register1.CashRegister(); // Syntax Error
```

- The correct way to reset an object is to construct a new one and assign it to the old:

```
register1 = CashRegister(); //creates an
// unnamed object, then copies it to register1
```