

COMP 10  
EXPLORING COMPUTER SCIENCE

Lecture 10  
Object-oriented programming

TODAY

Project presentations

Object-oriented programming

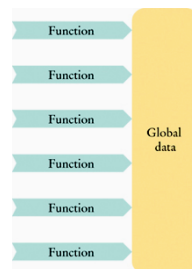
Course evaluations

PROCEDURAL PROGRAMMING

As programs get larger,  
it becomes increasingly difficult  
to maintain a large collection of  
functions

Often becomes necessary to use  
global variables

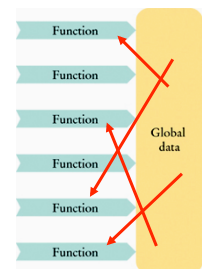
Defined outside of all functions  
All functions have access to them



PROCEDURAL PROGRAMMING

What if some part of the global data  
needs to be changed (to improve  
performance or to add new  
capabilities)?

A large number of functions  
may be affected



OBJECTS TO THE RESCUE

Keep the data and the functions that work with them together

Primitive types and functions: *Procedural programming*

Can use *classes* to create new types: *Object-oriented programming*

CUSTOM TYPES

A *class* is a template or blueprint for objects

May contain variables and functions

An *object* is an instance of a particular class

## OBJECTS TO THE RESCUE

An object's *member variables*:

Set of variables of certain types

Store information or current state of the object

An object's *member functions*:

Let the main program or other objects modify the state

Allow information to be communicated between objects

## CLASS: Muppet

### Variables:

name  
species  
color  
numberOfLegs  
series

### Functions:

speak()  
getSpecies()  
getColor()

## Muppet OBJECTS



### Variables:

name = "Kermit"  
species = "Frog"  
color = "Green"  
numberOfLegs = 2  
series = "Sesame Street"



### Variables:

name = "Rowlf"  
species = "Dog"  
color = "Brown"  
numberOfLegs = 4  
series = "Muppet Show"

## CLASS: MuppetSeries

### Variables:

name  
seriesType  
castofCharacters

### Functions:

listCharacters()  
addNewCharacter()

## MuppetSeries OBJECTS

### Variables:

name = "Sesame Street"  
seriesType = "TV"  
castofCharacters =



### Variables:

name = "Muppet Show"  
seriesType = "TV"  
castofCharacters =



## ENCAPSULATION

The data members are *encapsulated*: Hidden from other parts of the program and accessible only through member functions

When we want to change the way that an object is implemented, only a small number of functions need to be changed

Those member functions are the *interface* to the object

## DEFINING A CLASS

Don't implement a single object; implement a class

A class describes a set of objects with the same behavior

To define a class, specify the behavior by providing implementations for the member functions and by defining the data members for the objects

Implement the member functions to specify the behavior

Define the data members to hold the object's data

## BASIC CLASS FORMAT

```
class ClassName
{
    public:
    // public interface
}
private:
// private data members
};
```

Any part of the program should be able to call the member functions

Only member functions of the class can access data

## Muppet

```
class Muppet
{
    public:
    // public interface

    private:
    // private data members
    string name;
    string species;
    string color;
    int numberOfLegs;
    string series;
};
```

## USING Muppets IN main ()

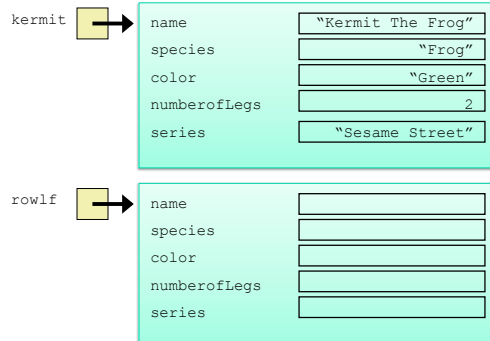
```
class Muppet
{
    public:
    // public interface

    private:
    // private data members
    string name;
    string species;
    string color;
    int numberOfLegs;
    string series;
};

int main()
{
    // Create an object.
    Muppet kermit;
    kermit.name = "Kermit The Frog";
    kermit.species = "Frog";
    kermit.color = "Green";
    kermit.numberOfLegs = 2;
    kermit.series = "Sesame Street";

    // Create another object.
    Muppet rowlf;
    ...
}
```

## USING Muppets IN main ()



## OBSERVATIONS

main () **not** declared in the Muppet class

### Member variables

Declared in a class, outside of any method

Exist as long as the containing object exists

### Local variables

Exist only during the execution of its containing method

### Dot (.) operations

Access a member variable

Call a member function; the object is the *implicit parameter*

## OBJECT-ORIENTED PROGRAMMING

An object encapsulates state and behavior

State ⇔ Variables

Behavior ⇔ Functions

A class is a template or blueprint for objects

An object is an instance of a particular class

Each object is responsible for carrying out a set of related tasks

Objects “talk” to each other to complete tasks

## DESIGNING A CLASS: CASH REGISTER

Need member functions to

- Clear the cash register to start a new sale
- Add the price of an item
- Get the total amount owed and the count of items purchased



## DESIGNING A CLASS: CASH REGISTER

Need member functions to

- Clear the cash register to start a new sale
- Add the price of an item
- Get the total amount owed and the count of items purchased

The public interface supports these activities:

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

private:
    // data members will go here
};
```



## MEMBER FUNCTIONS

There are two kinds of member functions:

- Mutators
- Accessors

A *mutator* modifies the data members of the object

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

private:
    // data members will go here
};
```

Clear the cash register  
to start a new sale

## MEMBER FUNCTIONS

There are two kinds of member functions:

- Mutators
- Accessors

A *mutator* modifies the data members of the object

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

private:
    // data members will go here
};
```

Add the price of an item

## CALLING MEMBER FUNCTIONS

First create a variable of  
type `CashRegister`

```
CashRegister register1;
```

1 Before the member function call.

```
register1 =
```

**CashRegister**

```
item_count = 0
total_price = 0
```

Then use the `dot`  
operator to call the  
member functions:

```
register1.clear();
register1.add_item(1.95);
```

2 After the member function call `register1.add_item(1.95)`.

```
register1 =
```

**CashRegister**

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

## MEMBER FUNCTIONS

There are two kinds of member functions:

- Mutators
- Accessors

An *accessor function* queries a data member of the object

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

Get the total amount owed and the count of items purchased

## THE INTERFACE



Access member variables using the dot operator

This statement will print the current total:

```
cout << register1.get_total() << endl;
```

## MEMBER VARIABLES

Each `CashRegister` object must store the total price and item count of the sale that is currently rung up

We store this data as private member variables

```
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;
};
```

Accessible only by object's member functions

## TWO NOTIONS OF EQUALITY

### Shallow equality

Same actual object data in memory  
Compare only references  
==

### Deep equality

Compare objects' variable values  
Might not have same reference  
Need a separate function for this

## CONTROLLING ACCESS TO OBJECTS

*Access specifiers* define where the variable or function is accessible

`public`: can be used from any class

`private`: can only be used within the class in which it is defined

Protect data from manipulation, corruption

```
class KindlyPerson {
public:
    string name;
    double salary;
};
class EvilDoer {
public:
    KindlyPerson enemy;
    EvilDoer(KindlyPerson e) {
        enemy = e;
    }
    void sabotageEnemy() {
        enemy.salary = 0;
    }
};
int main() {
    KindlyPerson bert;
    EvilDoer ernie;
    ernie.sabotageEnemy();
}
```

## CONTROLLING ACCESS TO OBJECTS

Important to keep variables private, but most functions are public

Private functions are generally only used for helper functions you don't want other classes to have direct access to

Hide implementation details: Lets us change internal representation without changing interface

## IMPLEMENTING THE MEMBER FUNCTIONS

```
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;
};
```

## IMPLEMENTING THE MEMBER FUNCTIONS

```
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;
};

void CashRegister::add_item(double price)
{
    item_count++;
    total_price = total_price + price;
}
```

Specify that a function is a member function in the name

## IMPLEMENTING THE MEMBER FUNCTIONS

```
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;
};

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

## IMPLEMENTING THE MEMBER FUNCTIONS

```
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;
};

double CashRegister::get_total() const
{
    return total_price;
}

int CashRegister::get_count() const
{
    return item_count;
}
```

## THE CASH REGISTER PROGRAM

```
// Display the item count and total price of a cash register.
void display(CashRegister reg)
{
    cout << reg.get_count() << " $" << reg.get_total() << endl;
}

// The main cash register program.
int main()
{
    CashRegister register1;
    register1.clear();
    register1.add_item(1.95);
    display(register1);
    register1.add_item(0.95);
    display(register1);
    register1.add_item(2.50);
    display(register1);
    return 0;
}
```

## CONSTRUCTORS

A constructor is a member function that initializes the data members of an object

The constructor is automatically called when an object is created

```
CashRegister register1;
```

Constructors are written to guarantee that an object is always fully and correctly initialized when it is defined

## WHAT HAPPENS HERE?

```
CashRegister register1;  
register1.add_item(1.95);  
int count = get_count(); // What is the count?
```

## THE CONSTRUCTOR

Guarantees that an object's variables are correctly initialized

Constructor name must match class name

No return type

Invoked automatically by C++ when a new object is declared (never called directly)

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

## THE CONSTRUCTOR

Guarantees that an object's variables are correctly initialized

Constructor name must match class name

No return type

Invoked automatically by C++ when a new object is declared (never called directly)

```
class CashRegister  
{  
    public:  
    CashRegister();  
    ...  
};
```

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

## PLACEMENT OF `main()`

Your program may use multiple classes

Defined each class in a separate file

Each program can have only one `main()`

Choose one class (file) to contain the program's `main()`

## TWO KINDS OF FILES

*Header .h files (which will be #included) contain the interface:*

Definitions of classes

Definitions of constants

Declarations of nonmember functions

*Source .cpp files contain the implementation:*

Definitions of member functions

Definitions of nonmember functions

## TWO KINDS OF FILES

For the `CashRegister` class, create a pair of files:

`cashregister.h`  
the interface – the class definition

`cashregister.cpp`  
the implementation – all the member function definitions

## DEFINING CLASSES

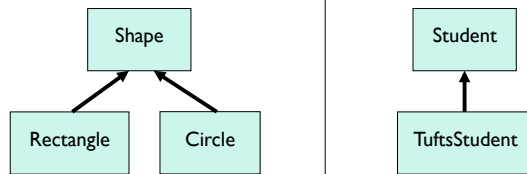
In a problem description...

Nouns correspond to classes, e.g. `paycheck` object

Verbs correspond to member functions, e.g. `compute_taxes()`

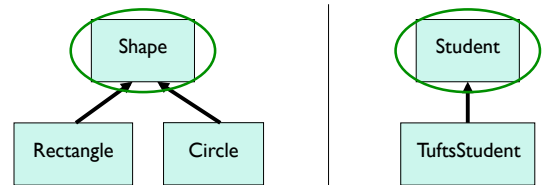
## INHERITANCE

Create new classes that are based on (or extend) existing classes.



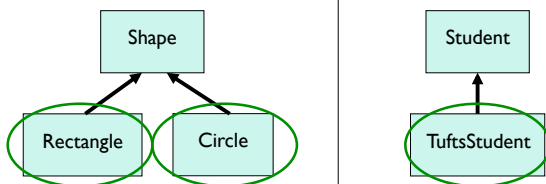
## SUPERCLASS (PARENT OR BASE CLASS)

The more general class from which other classes inherit



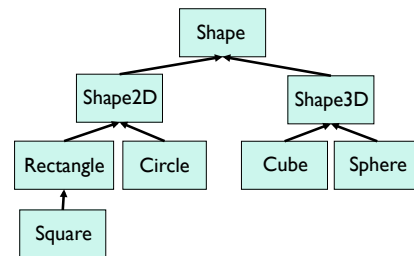
## SUBCLASS (CHILD OR DERIVED CLASS)

The more specific class that modifies the existing superclass by  
adding new fields or methods  
modifying (*overriding*) some methods' behavior



## INHERITANCE HIERARCHY

A superclass can also be a subclass of another class



### INHERITANCE HELPS US...

Factor out common functionality and put it in a superclass

Reuse code

Write code that more accurately reflects objects' relationships

“Is a”: inheritance

“Has a”: composition / aggregation

“Uses a”: dependence

### PROGRAMMING MODELS

Procedural programming

Break problem into tasks to be performed

Write functions to solve simple subtasks and combine them

Object-oriented programming

Break problem into classes or types

Associate functionality with those classes

### WHY CHOOSE ONE OVER THE OTHER?

Depends on the goal of the program

Object-oriented often considered better for

large-scale projects

working within a team

code that will be extended over time

OOP promotes encapsulation, reusability, and allows inheritance