Topic 6

1. Object oriented programming
2. Implementing a simple class
3. Specifying the public interface
4. Designing the data representation
5. Member functions
6. Constructors
7. Problem solving: tracing objects
8. Problem solving: discovering classes
9. Separate compilation
10. Pointers to objects
11. Problem solving: patterns for object data

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.)

Constructors: Motivation

By supplying a constructor, 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 = 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 the empty string.

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

```
CashRegister register1;
```

Notice that you do NOT use an empty set of parentheses when you call the constructor followed by a new identifier to create the object.

Constructors with Parameters

Constructors can have parameters, and can be overloaded :

```
class BankAccount
{
public:
   // "Default" constructor: Sets balance=0
   BankAccount();
   // Sets balance to initial_balance
   BankAccount(double initial_balance);
   // . . . Member functions omitted
private:
   double balance;
};
```

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:

```
BankAccount joes_account;
   // Uses default constructor
BankAccount lisas_account(499.95);
   // Uses BankAccount(double) constructor
```

Common Error: How (NOT) to Use the Constructor to Reset

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
```

Initialization Lists

When you construct an object whose data members are themselves objects, those objects are constructed by their class's default constructor.

However, if a data member belongs to a class *without* a default constructor, you need to invoke the data member's constructor explicitly.

Initialization Lists: Example (1)

A class to represent an item might not have a default constructor:

```
class Item:
public:
   Item(string item_descript, double item_price);
   // No other constructors
   ...
};
```

The Order class has an Item object as data:

```
class Order
{
public:
   Order(string customer_name, string
  item_descript, double item_price);
...
private:
   Item article;
   string customer;
};
```

Initialization Lists Example (2)

The Order constructor must call the Item constructor.

This is done in the ***initializer list***.

The initializer list goes after a colon, and before the opening brace of the constructor by putting the name of the data member followed by their construction arguments:

```
Order::Order(string customer_name,
   string item_description, double item_price)
         : article(item_description, item_price)
 {
    customer = customer_name;
}
```

Initialization Lists Example (3)

Any other data members can also be initialized in the initializer list by putting their initial values in parentheses after their name, just like the class type data members.
These must be separated by commas:

```
Order::Order(string customer_name,
             string item_description,
             double item_price)
      : article(item_description, item_price),
        customer(customer_name)
{
}
```

Notice there's nothing to do in the body of the constructor now.

C++ 11 Adds Uniform Initialization Syntax

There are several syntactic variations to initialize variables:

```
double price = 19.25;
int squares[] = { 1, 4, 9, 16 };
BankAccount lisas_account(499.95);
```

C++ 11 introduces a uniform syntax, using braces and no equal sign, like this:

```
double price { 19.25 };
int squares[] { 1, 4, 9, 16 };
BankAccount lisas_account { 499.95 };
```

Use empty braces for default initialization:

```
double balance {}; // Initialized with zero
BankAccount joes_account {};
// Uses default constructor
```

Don't expect to see this change adopted by most C++ programmers soon.