# CS 101: Computer Programming and Utilization

Shivaram Kalyanakrishnan
(Abhiram Ranade's slides, borrowed and edited)
Lecture 23

# Today's Lecture

- Global variables
- Namespaces
- Constructors, initialisation lists in structures

# Namespaces: High level ideas

- Suppose many people cooperatively develop a single program.
  - Possible that several people may define function with the same name.
  - Creates conflict/ambiguity.
  - Can be avoided using namespaces.
  - Namespace = catalog of names.
  - The “full name” of a function `f` defined in a namespace `N` is `N::f`
  - Suppose `f` is defined in two namespaces `N` and `P`.
    - We can specify which we mean by writing `N::f` or `P::f`.

# Defining a namespace

```
namespace N{
  declarations/definition of names

}
```

- This creates a namespace with name `N`, and also defines/ declares names inside it.
- You can add more names to a namespace `N` simply by writing `namespace N{ }` again.
- A name `g` defined without putting it inside a namespace is said to belong to the global namespace. Its fullname is `::g`.

# Example

```
namespace N{
   int gcd(int m, int n){ … }
   int lcm(int m, int n){ … }
}
int main(){
   cout << N::lcm(36,24) << endl;
}
```

# The `using` directive

- Suppose you refer to names defined in some namespace `N` very frequently.
  - You may find it tedious to write `N::` all the time.
  - Put the following line at the top of your program

```
using namespace N;
```

  - Then you will be allowed to use any name from `N` without having to write `N::` before it.

# Today's Lecture

- Global variables
- Namespaces
- Constructors, initialisation lists in structures

# “Packaged Software components”

- Things that you buy from the market are packaged, and made safe to use.
  - Fridge, television : no danger of getting an electric shock.
  - A “control panel” is provided on the device.  A user does not have to change capacitor values to change the channel on a television.
- Analogous idea for software:
  - Make functionality associated with a `struct` available to the user only through member functions (“control panel”)
  - Do not allow the user to directly access the data members inside a `struct`.  (Just as a user cannot touch the circuitry)  The user does not need to know what goes on inside.
- If you build a better fridge, keep control panel same as the previous model, the user does not need to relearn how to use the new fridge.
  - If you build a better version of the `struct`, but keep the member function signatures the same, the programs that use the `struct` need not change.

# The modern version of a struct

- Can behave like a packaged component.
- Designer of the struct provides member functions.
- Designer of the struct decides what happens during execution of "standard" operations such as:
  - Creation of the object.
  - Assignment
  - Passing the object to a function
  - Returning the object from a function
  - Destroying the object when it is not needed.

  How to do this: discussed next.
- Once structs are designed in this manner, using them becomes convenient and less error-prone.
- Structs endowed with above features are more commonly called "objects".

# Constructor: Motivational example- the Queue struct in taxi dispatch

```
const int N=100;
struct queue{
 int elements[N],
     nwaiting,front;
 void initialize(){
…
}
 bool insert(int v){
  …
  }
 book remove(int &v){
  …
 }
};
```

```
int main(){
 Queue q;
 q.initialize();
 ...
}
```

- A programmer may forget to call `initialize`!
- The designer can ensure that `q.nWaiting` and `q.front` will become 0 even so!
  - Next

# Constructor example

- In C++, the programmer may define a special member function called a constructor
- The constructor is called whenever an instance of the struct is created.
- A constructor has the same name as the struct, and no return type.
- The code inside the constructor can perform initializations of members.
- When q is created in the main program, the constructor is called automatically.

```
struct Queue{
  int elements[N],
front,
      nWaiting;
  Queue(){ //
constructor
    nWaiting = front =
0;
  }
  ...
};
int main(){
  Queue q;
  ...
}
```

# Constructors in general

```
struct X{
   …
   X(parameters){
   …
   }
};

int main(){
   X x(arguments);
}
```

- Constructor can take arguments.
- The creation of the object `x` in main can be thought of as happenning in two steps.
  - Memory is allocated for `x` in main.
  - The constructor is called on `x` with the given arguments.
- You can have many constructors, provided they have different signatures.

# Another example: Constructors for V3

```
struct V3{
   double x,y,z;
   V3(){
      x = y = z = 0;
   }
   V3(double a){
      x = y = z = a;
   }
};
int main();
   V3 v1(5), v2;
}
```

- When defining v1, an argument is given. So the constructor taking a single argument is called. Thus each component of v1 is set to 5.
- When defining v2, no argument is given. So the constructor taking no arguments gets called. Thus each component of v2 is set to 0.

# Predefined constructors

If you do not define a constructor, C++ defines one for you

- Constructor takes no arguments
- Does nothing if data members are fundamental types

If you define any constructor, your constructors gets used.

Note: For nested structures, first constructors of members are called, then constructor of outer object.

# Copy constructor

When you pass a structure to a function (by value), a copy needs to be made.

- This is done by a function called the copy constructor.

C++ provides a predefined copy constructor

- It copies member by member

You can override

- Graphics object passed to function: make copy + show on screen.
- More examples next week.