

Basics of Classes

5.1 Data Members and Member Functions

An object is a particular instance of a class and there may be multiple instances of a class. An object has (i) data, called attributes, fields or *data members*, and (ii) functions, called methods or *member functions*. A member function is called with the `.` notation. For example:

```
object.method();
```

Every member function can access every data member. But, usually all data and some member functions are labeled **private**; user methods are labeled **public**. (There are other possibilities.) Private means that the external user of this class cannot see or use it.

5.2 More on Member Functions

Most objects have *accessor* functions: these allow the user to *get* data from the object. *Mutator* functions allow the user to *set* data in the object. One can indicate to the user that a function does not alter its arguments and/or the object by using the `const` modifier.

A variable has a *scope* (where it is accessible from) and a *lifetime* (when it exists). The variables defined in a function are called *local variables* and are accessible only within the function and exist only while the function is being executed. An exception is *static* variables whose lifetime is the program's execution: they are always available, and there is only one copy per class. If public, a static variable can be accessed by using the *scope resolution* operator: prefixing it with the classname followed by `::`.

A function should normally check its arguments. It notifies the caller of a problem by using an *exception* (discussed later) or a special return value. The programmer should try to avoid exceptions: consider error recovery and avoidance.

5.3 Constructors

A *constructor* is a special function that initializes the state of the object; it has the same name as the class, but does not have a return type. There can be more than one constructor. Note that the compiler will provide a default constructor if none is coded.

When a constructor is called, any member data is initialized before any of the commands in the body of the constructor. In particular, any member that is a class

has its constructor called automatically. (This occurs in the order that the member data is listed when defined.) So one should use specific *initializers*; this is a list after the header before the body.

```
class Foo {
public:
    Foo() : Bar(1) , ging('d')    // default constructor
    {
    }
private:
    Iso Bar;
    char ging;
};
```

5.4 Strings

There is a `string` class in the `string` library. These can be compared, cin-ed and cout-ed, assigned C-string, appended, and many other things.

5.5 Sample program: Citizen.cpp

The output is

```
The drinking age is 21
GI Joe can drink
Barbie can't drink
```

```
// Citizen.cpp - wdg - 2009
#include <iostream>
#include <string>
using namespace std;

class Citizen
{
public:
    static const int DRINKING_AGE=21;

    // constructors
    Citizen() : name("UNKNOWN"), age(0)
    {}
```

```

Citizen(string nam, int ag) : name(nam), age(ag)
{}

//mutators
void setName(string nam)
{
    name = nam;
}

void setAge(int ag)
{
    age = ag;
}

//accessors
string getName() const
{
    return name;
}

int getAge() const
{
    return age;
}

bool canDrink() const
{
    return (age>=DRINKING_AGE);
}

private:
    string name;
    int age;
};

int main( )
{
    Citizen one;

```

```
one.setName("GI Joe");
one.setAge(50);
Citizen two("Barbie",13);

cout << "The drinking age is " << Citizen::DRINKING_AGE << endl;
cout << one.getName();
if ( one.canDrink() )
    cout << " can drink\n";
else
    cout << " can't drink\n";

cout << two.getName();
if ( two.canDrink() )
    cout << " can drink\n";
else
    cout << " can't drink\n";

return 0;
}
```