

## II.7. OOP Concepts in C++

### Declaring a class

The first step in designing a new class is to create the class declaration. For the Cat example, the declaration is:

```
class Cat
{
    unsigned int age;
    float weight;
    void Meow();
};
```

Some people like to decide on variables first, while others like to decide on functions first. As with function declarations, class declarations must appear before the class is used.

In the class declaration, member functions and variables can be given different access restrictions:

- public: access from any function,
- protected: access from this class's and sub-class functions only,
- private: access from this class's functions only,

For example:

```
class Cat
{
    public:
    unsigned int age;
    float weight;
    void Meow();
};
```

### Defining class functions

Class functions are defined the same as any other functions, except that the function name must be prefixed with the class name and two colons. E.g.:

```
void Cat::Meow()
{
    cout << "Meow." << endl;
}
```

### Creating an object

An instance of a class (an object) can be declared the same way as fundamental data types. E.g.:

```
Cat Frisky;
```

As with fundamental data types, the data will be erased at the end of the function. To control the creation and deletion of objects, use object pointers, and the keywords “new” and “delete”. E.g., use:

```
Cat *pFrisky = new Cat;
```

to create the object and assign it to the pointer pFrisky. Use:

```
delete pFrisky;
```

to delete the object (and free the memory). Every call to “new” must have a corresponding call to “delete” somewhere in the program. Memory leaks will occur if you assign a new value to pFrisky before the previous pFrisky is deleted. Memory leaks are memory spaces that can no longer be accessed, but that the system thinks are still in use.\*

## Accessing an object

To access data or functions of an object use the period. E.g.:

```
Frisky.age = 5;
```

If you have a pointer to an object, use ‘->’ instead. E.g.:

```
pFrisky->age = 5;
```

## Constructor/destructors

What is the initial value of `Frisky.age`, before the value 5 is assigned? It could be anything. “Constructors” are functions that are automatically called when an object is created. They are used to initialize objects, including setting the initial values of variables. “Destructors” are functions that are automatically called when an object is deleted. Constructors and destructors have the same name as the class, and destructors have an additional ‘~’ in front. They never have a return type. E.g.:

```
class Cat
{
    public:
        Cat();
        ~Cat();
        ...
}
```

---

\* Note that Java and C++ “Smart pointers” automated the deletion of used objects.

```
};
```

*Create a constructor to initialize the values of the cat's age and weight. Remove the initialization code in main().*

## Access Functions

It is highly recommended that data members NOT be made public, and instead be made accessible through access functions to get and set their values. This way they can not be inadvertently changed by others. As well, it allows you to easily change how the class works, without requiring other people's code to be changed. It also makes debugging easier, since you can then easily trace whenever a value has been changed. One convention for access functions is shown here:

```
class Cat
{
    public:
        unsigned int age() { return m_age; }
        void age(unsigned int aage) { m_age = aage; }
        float weight()      { return m_weight; }
        void weight(float aweight)  { m_weight = aweight; }
        void Meow();
    protected:
        unsigned int m_age;
        float m_weight;
};
```

Instead of using “*m\_variable\_name*” for object variable names, some conventions simply use an underscore: “*\_variable\_name*.” Another convention is to leave the variable name without a prefix and instead prefix the access functions with “Get” and “Set”. E.g., GetAge(), SetAge(), GetWeight(), SetWeight().

## Inheritance Syntax

To inherit a class, list it after the class name in the class declaration as in this example:

```
class SiameseCat : public Cat
{
};
```

The class “SiameseCat”, known as the “derived,” “child,” or “sub-class” now has all of the data and functions from Cat, the “parent” or “super-class”.

Interface classes should be inherited as “virtual” functions in C++. The data in virtually inherited classes is not accessible in the derived class.

```
class SiameseCat_sound : public SiameseCat, virtual public
    Animal_sounds
{
};
```

### Over-riding a function

To over-ride a function from the parent class,

- (i) declare the function as virtual in the parent class,
- (ii) re-declare it in the child class declaration, and
- (iii) re-define it for the child class. E.g.:

```
class Cat
{
    public:
        virtual void Meow();
    ...
};

class SiameseCat : public Cat
{
    public:
        void Meow();
};

void SiameseCat::Meow()
{
    cout << "Purr..meow." << endl;
}
```

Now edit the main program to add Spot, the SiameseCat. The variable Spot will also be of type Cat to demonstrate polymorphism. Since polymorphism only works with pointers, Frisky and Spot will both be changed to pointers.

```
int main()
{
    Cat *pFrisky=0, *pSpot=0; // pointers should always point
    // to a valid object or should be null
    pFrisky = new Cat;
    pSpot = new SiameseCat;
    pSpot->age(4);
    pSpot->weight(6.);
}
```

```

pFrisky->Meow();
pSpot->Meow();
cout << "Frisky is a cat who is ";
cout << pFrisky->age << " years old." << endl;
cout << "Spot is a cat who is ";
cout << pSpot->age << " years old." << endl;
pFrisky->Meow();
pSpot->Meow();
return 0;
}

```

## Over-loading a function

While “over-riding” a function redefines a function in a sub-class, “over-loading” a function refers to defining a new function with the same name as an existing function. This new function must have a different return type or different parameters. E.g.:

```
void Meow(float tone, float loudness);
```

## Over-riding operators

Operators are not just for the fundamental data types. Operators can be defined for any kind of object. E.g.

```

Cat mommy_cat, daddy_cat;
Cat *pkitten = daddy_cat + mommy_cat;

```

is a valid expression if the ‘+’ operator has been implemented for Cat. In this example, the ‘+’ operator has been implemented to create a new Cat object. The syntax is the same as for over-riding a function, except that “operator<symbol>” is used as the function name. E.g., in the Cat class declaration, use:

```
Cat *operator+(Cat mom);
```

Only one parameter is required, the object behind the ‘+’ symbol. The object in front of the ‘+’ symbol is the one for which the function is called. The operator function definition is:

```

// This operator evaluates: this + mom
// and returns: Cat *kitten
Cat *Cat::operator+(Cat mom)
{
    Cat *pkitten = new Cat;
    pkitten->age = 0;
    pkitten->weight = 0.05 * (weight + mom.weight)/2.;
    return pkitten;
}

```

Note that “weight” by itself refers to “this” object’s weight. It is daddy\_cat’s weight, since daddy\_cat is the operand to the left of the ‘+’ operator.

## The “this” keyword

The “this” keyword can be used in a class function to yield a pointer to the object for which the function is running. E.g. we can use:

```
kitten->weight = 0.05 * (this->weight + mom.weight)/2.;
```

The result will be exactly the same as above.

## dynamic casting

If a sub-class has member function you need to access, you can use dynamic casting. The dynamic cast will return 0 if the variable is not of the indicated type. E.g.:

```
class SiameseCat : public Cat
{
public:
    void Meow();
    void SiameseMessage() { cout << "I'm Siamese!" << endl; }
};

Cat *pFrisky=0, *pSpot=0;
pFrisky = new Cat;
pSpot = new SiameseCat;
SiameseCat *ptest=0;
ptest = dynamic_cast<SiameseCat*>(pFrisky);
if (ptest)
{
    cout << "Frisky says: ";
    ptest->SiameseMessage();
}
ptest = dynamic_cast<SiameseCat*>(pSpot);
if (ptest)
{
    cout << "Spot says: ";
    ptest->SiameseMessage();
}
}
```

The dynamic\_cast keyword requires the compiler flag “-features=rtti” to be set to enable run-time type identification. E.g. compile using:

```
>CC -features=rtti cat.cc
```

Similarly, in Visual C++, Run-Time Type Identification needs to be enabled in the compiler settings.

## Working with Pointers and References

Pointer variables are declared by using an asterisk. For example,

```
float *pX;
```

defines the variable `pX` to be a pointer to a memory location holding a floating point value.

Use the `&` (“reference”) operator to refer to the memory location, rather than the value of a variable. Use the `*` (“indirection” or “dereference”) operator to refer to the value at the memory location indicated by the pointer. For example:

```
float my_age, *pAge1, *pAge2;
my_age = 5.;
pAge1 = &my_age;
pAge2 = pAge1;
cout << "The age is " << *pAge2 << endl;
```

*Draw diagram of memory and values in memory and pointers for above program.*

Try this program. See if you can predict what the output will be.

```
#include <iostream>
using namespace std;

int main()
{
    unsigned short my_age = 5, your_age = 6;
    cout << "my_age: value " << my_age << " is in memory location "
         << &my_age << endl;
    cout << "your_age: value " << your_age << " is in memory "
         << location << &your_age << endl;

    unsigned short *p_age = &my_age;

    cout << "p_age: value " << p_age << " is in memory location "
         << &p_age << endl;
    cout << "*p_age: value " << *p_age << " is in memory location "
         << p_age << endl;
    return 0;
}
```

In this program `p_age` can be used to refer to either of the ages, `my_age` or `your_age`.

Practice safe computing. Pointers should ALWAYS be pointing at a valid address, or else should be assigned a value of NULL (0). As well, for every “new” operation, there must always be one pointer that “owns” the object (or memory) until it is deleted. Failure to do so will result in “memory leaks” (memory that is reserved, but can never be used.)

### Dynamic allocation of arrays

To dynamically reserve an array of memory locations, use the `new` keyword.  
E.g.:

```
Cat *family = new Cat[10];
```

This can be done with objects as well as the fundamental data types. To delete the array use:

```
delete []family;
```