Chapter 12: Object Oriented Programming

Object oriented programming is a different approach to writing code. There are two concepts that need to be addressed: **Classes** and **Objects**. A class can be viewed as a blueprint from which many objects are created. Imagine the blueprint (class) for a house. Many houses (objects) can be constructed from these plans, each one containing its own attributes / locational data.

The class approach builds from the **struct** approach covered in a previous chapter. However there are benefits to using classes that structs do not possess. These are techniques such as inheritance and polymorphism, which will reduce the amount of code you are required to write, whilst also making for a stronger code base.

To declare a new class, simply use the keyword **class** followed by the name you wish to call your class. The following example creates a class called Cat.

class Cat

{

}

To control what can be accessed and changed in your class you will need to use accessors. These restrict access to internal functions / variables according to table 15.1

|  |  |
| --- | --- |
| **Accessor** | **Description** |
| public | Anything falling under public can be accessed from anywhere. |
| protected | Access is only given to the current class and child classes. |
| private | Access is only available to the current class.  This is the default access level if one is not set. |

Table 15.1: Accessors

Next we will add some functions and variables to our Cat class. These will be set to differing accessor levels.

class Cat

{

public:

Cat( int initialAge );

~Cat();

int GetAge();

void SetAge( int age );

private:

int itsAge;

}

As you can see from the above code snippet there are two functions with the same name as the class, ie Cat.

The following is a **constructor**. A constructor is the function that will be called when an object of this class type is created. A constructor does not have a return type like standard functions do, but may take whatever values as parameters as are required for setup.

Cat( int initialAge );

The second function with the same name as the class is the **destructor**. A destructor does not have a return type and cannot take any parameters. The function name it is always preceeded with a tilde ~ character to distinguish it from the constructor.

The destructor will always be called when an object goes out of memory when created on the stack or is deleted from memory when created on the heap.

~Cat();

When writing the definitions for functions, you need to include the header class at the top of your source file. Each function prototype requires the name of the class the function is referring to. See the GetAge() function below.

In the header:

int GetAge();

In the source:

int Cat::GetAge()

{

return itsAge;

}

If a function definition contains minimal code as in the above example it is suitable to be implemented as an inline function. That is to say the implementation is contained within the header. The above code snippets modified into an inline function would look like this:

int GetAge() { return itsAge; }

**Program 44: Creating a Class**

1. To begin, start Visual Studio.
2. Create a new project via File -> New -> Project or Ctrl+Shift+N Name it “Chapter15\_CatClass”
3. Click **Next** and you should be greeted with the following screen. Make sure to have **Empty Project** ticked and click **Finish**.
4. Add a new header file called “Cat.h” and a source file and name it “Cat.cpp” and a source file called “Main.cpp”
5. Replicate program listings 44.

// Cat.h

#ifndef CAT\_H

#define CAT\_H

class Cat

{

public:

Cat( int initialAge );

~Cat();

int GetAge();

void SetAge( int age );

private:

int itsAge;

};

#endif //CAT\_H

Program Cat Header Listing 44

// Cat.cpp

#include "Cat.h"

Cat::Cat( int initialAge )

{

itsAge = initialAge;

}

Cat::~Cat()

{

}

int Cat::GetAge()

{

return itsAge;

}

void Cat::SetAge(int age )

{

itsAge = age;

}

Program Cat Source Listing 44

// Main.cpp

#include "Cat.h"

#include <iostream>

using namespace::std;

int main()

{

// Create on the stack – Has parameters so the brackets are required along with the starting value.

Cat button(5);

// Create on the heap.

Cat\* pDaisy = new Cat(10);

cout << "Button is " << button.GetAge() << " years old." << endl;

cout << "Daisy is " << pDaisy->GetAge() << " years old." << endl;

cin.get();

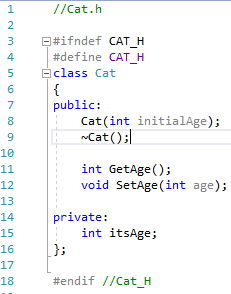
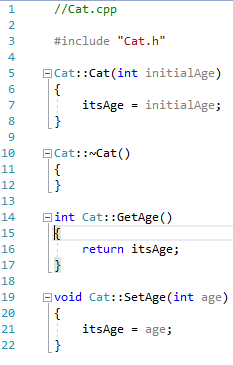
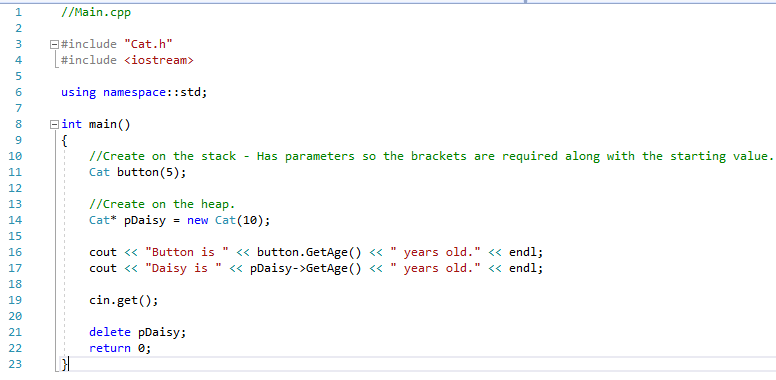
delete pDaisy;

return 0;

}

Program Main Source Listing 44

Note\* In the header is a pre-processor which determines whether the header has already been created.





**Inheritance**

Inheritance is a key benefit to using an objected oriented approach. When you write a class you may find that it contains all the information another class requires. Instead of duplicating this code you can instead inherit all this behaviour for free.

As an example, imagine a game character. This character has a texture for drawing its image on screen and a minimum of a positional data. We need to create a Mario class and a Luigi class. Instead of repeating this generic code we can write a base class for this data and then have both Mario and Luigi inherit this behaviour.

To inherit from another class use the following in the class declaration header:

class childClass : parentClass

or in terms of our Mario example:

class Mario : baseCharacter

and

class Luigi : baseCharacter

Within the source declaration file of the child class similar syntax is required to call the parent class’ constructor.

Mario::Mario() : baseCharacter()

{

// Mario Constructor.

}

An important point to note is that the order in which classes are constructed and destructed. The following example program will demonstrate this.

**Program 45: Creation Order**

1. To begin, start Visual Studio.
2. Create a new project via File -> New -> Project or Ctrl+Shift+N Name it “Chapter15\_ClassCreationOrder”
3. Click **Next** and you should be greeted with the following screen. Make sure to have **Empty Project** ticked and click **Finish**.
4. Add a headers and source files for ‘BaseClass’ and ‘ChildClass.’ Also create a source file called “Main.cpp”
5. Replicate program listings 45.

// BaseClass.h

#ifndef BASECLASS\_H

#define BASECLASS\_H

class BaseClass

{

public:

BaseClass();

~BaseClass();

};

#endif // BASECLASS\_H

Program BaseClass Header Listing 45

// BaseClass.cpp

#include "BaseClass.h"

#include <iostream>

using namespace::std;

BaseClass::BaseClass()

{

cout << "BaseClass Constructor" << endl;

}

BaseClass::~BaseClass()

{

cout << "BaseClass Destructor" << endl;

}

Program BaseClass Source Listing 45

// ChildClass.h

#ifndef CHILDCLASS\_H

#define CHILDCLASS \_H

#include "BaseClass.h"

class ChildClass : BaseClass

{

public:

ChildClass();

~ ChildClass ();

};

#endif // CHILDCLASS\_H

Program ChildClass Header Listing 45

// ChildClass.cpp

#include "ChildClass.h"

#include "BaseClass.h"

#include <iostream>

using namespace::std;

ChildClass::ChildClass() : BaseClass()

{

cout << "ChildClass Constructor" << endl;

}

ChildClass::~ChildClass()

{

cout << "ChildClass Destructor" << endl;

}

Program ChildClass Source Listing 45

// Main.cpp

#include "ChildClass.h"

#include <iostream>

using namespace::std;

int main()

{

cout << "CREATING CHILD CLASS ON HEAP" << endl;

// Create on the heap.

ChildClass\* pChild = new ChildClass();

cout << "DELETING CHILD CLASS FROM HEAP" << endl;

delete pChild;

cout << "CREATING CHILD CLASS ON STACK" << endl;

// Create on the stack – No parameters, so no brackets required.

ChildClass child;

// Pause for input.

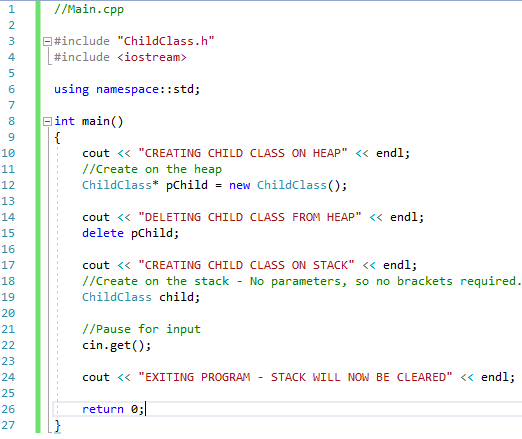
cin.get();

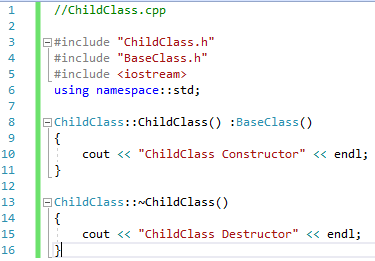
cout << "EXITING PROGRAM – STACK WILL NOW BE CLEARED" << endl;

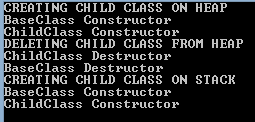
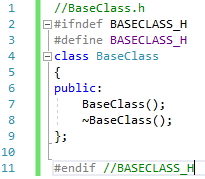
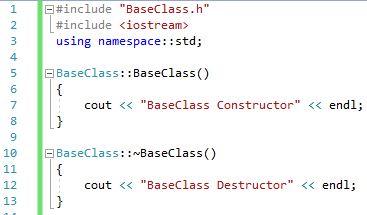
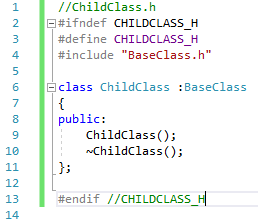
return 0;

}

Program Main Source Listing 45







**Polymorphism**

Polymorphism is another technique which will extend your programming skills. This technique allows you to call functions on a class even when it is contained within a list of its parent type. Going with the Mario example again, we could have an array of BaseCharacter type, which could hold all the characters in the level. Obviously Mario is of Mario type and only inherits from BaseCharacter, but because of the lineage the array can hold Mario.

It will make sense when you have worked through the example program below. Before we get in to that though there is one more keyword you need to be aware of. This is the **virtual** keyword. In a base class if you use this term before your functions it will enable the correct function to be called regardless of the type it was called from. This is only stated in a class you wish it to be overridden from.

**Program 46: Polymorphic Behaviour**

1. To begin, start Visual Studio.
2. Create a new project via File -> New -> Project or Ctrl+Shift+N Name it “Chapter15\_polymorphicBehaviour”
3. Click **Next** and you should be greeted with the following screen. Make sure to have **Empty Project** ticked and click **Finish**.
4. Add a header called ‘Mammals.h’ and a source file called “Main.cpp”
5. Replicate program listings 46.

// Mammals.h

#ifndef MAMMALS\_H

#define MAMMALS \_H

#include <iostream>

using namespace::std;

// Mammal class – Our base class.

class Mammal

{

public:

Mammal() : itsAge(1) {}

~Mammal() {}

virtual void Speak() const { cout << "Mammal speak!" << endl; }

protected:

int itsAge;

};

// Dog class.

class Dog : public Mammal

{

public:

void Speak() const { cout << "Woof!" << endl; }

};

// Cat class.

class Cat : public Mammal

{

public:

void Speak() const { cout << "Meow!" << endl; }

};

// Horse class.

class Horse : public Mammal

{

public:

void Speak() const { cout << "Neigh!" << endl; }

};

// Pig class.

class Pig : public Mammal

{

public:

void Speak() const { cout << "Oink!" << endl; }

};

#endif // MAMMALS \_H

Program Mammals Header Listing 46

//Main.cpp

#include "Mammals.h"

#include <iostream>

using namespace::std;

int main()

{

Mammal\* mammalPtr;

int choice;

cout << "1.Dog 2.Cat 3.Horse 4.Pig" << endl << "Enter choice : ";

cin >> choice;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

// Create the chosen mammal – stored within a mammal pointer.

switch (choice)

{

case 1:

mammalPtr = new Dog();

break;

case 2:

mammalPtr = new Cat();

break;

case 3:

mammalPtr = new Horse();

break;

case 4:

mammalPtr = new Pig();

break;

default:

mammalPtr = new Mammal();

break;

}

// Make this mammal speak.

mammalPtr->Speak();

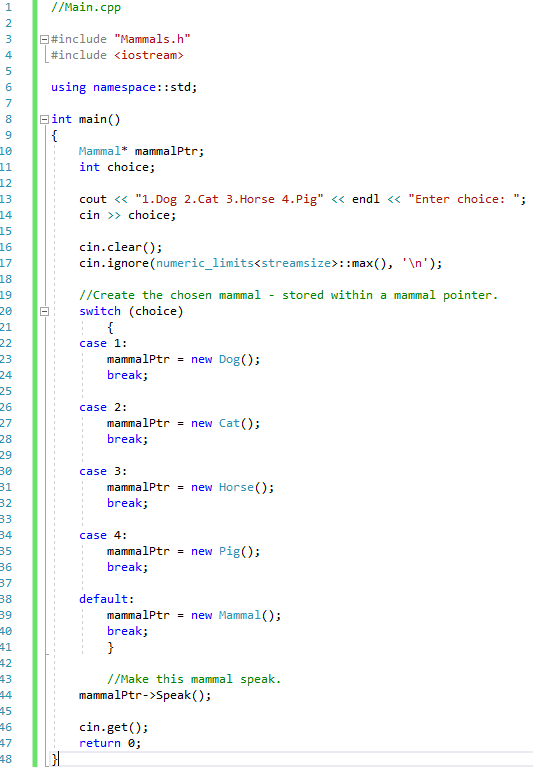
cin.get();

return 0;

}

Program Main Source Listing 46









**Program 47: Bank Account**

Write a program, which presents the user with the following menu:

1. Open a new account
2. View an account
3. Close an account
4. Exit program

Upon entering ‘1’ this second account menu is presented:

1. General account
2. Junior account
3. Savings account
4. Return to main menu

See below for the actions to be carried out when selecting from this menu.

Upon entering a ‘2’ the user is asked to enter an account number and if the account exists the following details are presented: Account number, Surname, Forename, Address, Balance, Interest rate and Account type. The user is then returned to the main menu.

Upon entering a ‘3’ the user is asked to enter an account number and if the account exists the account is closed. Closing an account removes it from the save file.

Upon entering a ‘4’ the program closes.

Within the account menu if the user enters a ‘4’ they should be returned to the main menu. On any other selection they are taken through a process of entering their details. These are to include: Surname, Forename & Address. They should be asked how much money they wish to place into the account to open it – this must be greater than 0 otherwise the process is cancelled. A unique account number is provided for the user. When the account has been opened the user is returned to the main menu.

All details must be stored on file so that they can be accessed between executions of the program. Ensure that adding a new account does not corrupt the already saved data. Likewise when closing an account.

To complete this task you should use the object-oriented approach described in this chapter. There should be a base account class with a junior account and savings account which both inherit from it. The differences in accounts will be the interest rates and account type.

Account type should be an enum consisting of GENERAL, JUNIOR and SAVINGS.

Provide screenshots of all menus and actions, along with the save file. The data shown in the file must correspond to the screenshots demonstrating it being used.

Program BaseAccount.h Source Listing 47

Program BaseAccount.cpp Source Listing 47

Program JuniorAccount.h Source Listing 47

Program JuniorAccount .cppSource Listing 47

Program SavingsAccount.h Source Listing 47

Program SavingsAccount .cppSource Listing 47

Program Main Source Listing 47

Program Screenshot 47

Account Save File 47