The Class Concept

Overview

- Some Differences Between C and C++
- Classes in C++
- Class Members and Special Members
- The C++ Header File
- Access Specifiers
- The Source File
- Using the Class

2

Input and Output (1/2)

- printf() is obselete J
- printf() uses a format specifier

```
printf("Value %d Text %s", 10, "hallo");
```

- C++ uses cout, no format specifier

```
cout << "Value" << 10 << "Text" << "hallo";
```

3

INPUT AND OUTPUT

In C you normally use printf() to display values on the screen. When using the printf function you specify how the output should be formatted followed by optional values to be inserted into the format string. This type of output can still be done in C++. However the output mechanism of C++ is a bit different than that of C. In C++ you do not supply a format specifier with values but you stream the things you want to print. You always start with cout. This object also called "kout' or "console out" is the representation of your screen. By using the operator<< you can print to the screen without specifying a format. The technique used is called operator overloading which we will describe later. When you use cout you supply one value at a time separated with the operator <<. In order to use cout you have to use the header file iostream.h. This header file is changed in the new ANSI C++ standard. In this new standard you have to include iostream (no extension). If you want to use this new standard you have to prefix cout and endl with std. It will be std::cout and std::endl. This technique is called namespaces which will not be discussed in this course. If you use the new standard you must keep in mind that you cannot mix the new and old standard in one project, they use different run-time libraries.

Input and Output (2/2)

- Newline in C++ is done with endl

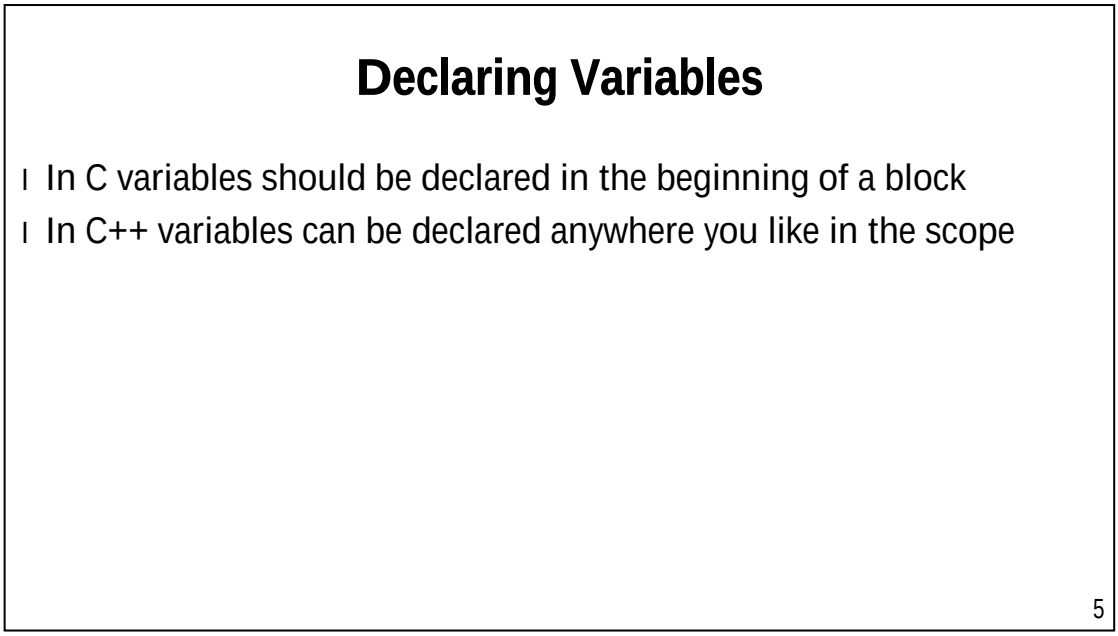
```
cout << "A one liner" << endl;
```

- Header file *iostream* is needed
- STL needed

4

INPUT AND OUTPUT (2)

In C you can use all kinds of escape characters like newline '\n'. These characters can still be used. However with the idea of streaming output C++ has an object that represents a newline this object is called endl.



Declaring Variables

In C you should declare a variable in the beginning of a new block. Whenever you start a new clock you can declare variables and use them in expressions following these declarations. It is not possible to mix the declarations and expressions. In C++ it is possible and sometimes good practice. There are situations where you need two different objects according to some settings. Declaring these objects from the beginning and using only one can be a waste of memory and time, especially if such an object has a lot of data inside. We do not say that you should not declare variables in the beginning of a new block but we want to make clear that it is not mandatory in C++.

Scope rules With for Loop

- In old standard i is global

```
for (int i = 0; i < 10; i++) {}
for (i; i < 20; i++) {}
```

- In new standard i is local

```
for (int i = 0; i < 10; i++) {}
for (int i = 0; i < 20; i++) {}
```

6

SCOPE RULES WITH FOR LOOP

In C you should declare a looping variable in the beginning of a block if you want to use it in certain for loop. In C++ this is not mandatory, as you should now after the previous sheets. A number of C++ programmers have been using this technique since the beginning of C++. In the first example on the sheet the first loop declares the variable i. In the second loop the variable still exists and should not be declared again. The declaration is not bound to the for loop but to the scope in which the for loop is declared. In the new C++ standard this has been changed. The declaration of the variable is local to the for loop. In the second example the variable i is declared again because it does not exist. When you start a new project or extend an existing project you should examine which of the two options you should use, because they are incompatible.

Classes in C++

- Class code separated into
  - Description of type,
    header file (*.h, *.hpp, *.hxx)
  - Implementation of member functions,
    source file (*.cpp, *.cxx)

7

CLASSES IN C++

Having discovered a class and the corresponding messages that can be sent to instances of that class it is possible to implement the class in C++.

When writing classes we split the code into two parts; first, the declaration of the class together with its structure and member functions and secondly, the body of the member functions making up the class. These two parts are to be found in the header and code files, respectively. There are a number of standard file extension names in use; for the header file we use extension 'hxx' or 'hpp' or 'h' and for the code file we use the extension 'cxx' or 'cpp'.

In the new ANSI C++ standard the extension of the header files is left out. The system header files do not have an extension. All the C header files are prefixed with a c for example cmath and cstdio instead of math.h and stdio.h.

Class Members

- Data members
  - Correspond to state/structure
  - Standard C types
  - Can be user-defined types
- Member functions
  - Act on the state of an object

8

CLASS MEMBERS

Class member data correspond to the structure relating to instances of the class. In some cases we can speak of the attributes of the class. Usually member data should be hidden from clients (because it can and does change) and thus most data will appear in the private part. For example, we can represent a point in two dimensions by two doubles (Cartesian coordinate system). At some later stage it may be more advantageous to change this representation to polar form (a double and an angle). The clients of the point class should be spared having to change their code to accommodate the change; placing the representation in the private part ensures this.

The essentials of member functions are:

- Name; each member function should have a suitably chosen name.
- Signature; this is the so-called return type of the function and its arguments.
- Arguments; these are the objects which function as input to the function. In the formal declaration we use so called function prototyping by which is meant that the type of the input object is given in the argument list.

Example:
Consider the function:

```
double calculate(int i, double d, Number iu);
```

In this case the return type is double; the name of the function is 'calculate' and the function has three arguments, one of which is a user-defined type (Number).

More on Member Functions

- Like normal C functions
- Messages for the object
- Special Member functions
  - Constructors
  - Destructors

9

MORE ON MEMBER FUNCTIONS

Member functions are declared the same as a C function consisting of three parts:

- return type
- function name
- function arguments

The member functions of an object are the messages the object responds to. These functions specify its behaviour.

The member functions are declared in the declaration of the class. This declaration is placed in a header file. The functions are implemented in the implementation file of the class that is called the source file or code file.

Constructor and Destructor

- Constructor gets called once for each object on creation
- Destructor called when object is destroyed
- Constructor and destructor are called by the system
- Cannot call these function explicitly
- Have NO return type (not even `void`)
- Same name as class

10

CONSTRUCTORS AND DESTRUCTOR

There are special member functions that are called the constructors and destructor. These member functions have the same name as the class and have no return type. It is possible to have any number of constructors in a class provided the argument list for a given constructor is not the same as that for another constructor. A constructor initialises the state of an object and may not return anything.

There is at most one destructor for a given class; its main purpose is to deallocate memory that has been created by a given constructor. We note that, besides freeing memory, a destructor can call other member functions. This might be necessary in certain applications. The name of a destructor is the same as that of its corresponding class with a preceding tilde '~' character. It is not possible to directly call a destructor from another function. It is not possible for a destructor to take arguments and, (like a constructor) it has no return type. Destructors may however call other member functions.

Destructors are used for the following purposes:

- Program termination for static objects.
- When auto or temporary objects go out of scope.
- Through the use of the delete operator or by using the destructors fully qualified name.

Some rules/restrictions relating to constructors and destructors
These will be discussed in more detail in later units.

- Constructors may not be declared virtual.
- Constructors are not inherited.
- A constructor turns raw memory into an object.
- Default constructors, destructors and copy constructors are generated by the compiler (when not defined by the user) where needed and these constructors are placed in the public area.
- A destructor reverses the effect of a constructor.
- 'Normal' destructors are not inherited.
- A destructor may be declared virtual.

Constructor and Destructor

```
class Point
{
public:
  Point();     // Default constructor, no return type
  ~Point();    // Destructor, no return type
};
```

```
void main()
{
  Point pnt;      // Constructor is called
  pnt.Point();    // Impossible, object already created
  pnt.~Point();   // DO NOT DO THIS,
                  // the destructor is called twice
} // Scope ends destructor is called of object pnt
```

11

```
class Point
{
private:
      // Data members

public:

      // Constructors and Destuctor
      Point();
      ~Point();
      // Member functions
      // Accessible to clients
      //Selectors
};
```

```
// smptst.cpp
// Simple test program for the point class.

#include <iostream.h>
#include "point.hpp"

void main()
{
      Point pt1;        // Constructor is called for pt1
      Point pt2;        // Constructor is called for pt2

}     // Scope for pt1 and pt2 ends, their destructor is called
```

The C++ Header File

- Contains
  - Description and purpose of class
  - Class description
  - Strategic comments
  - Tactical comments
  - Does not contain implementation

12

THE C++ HEADER FILE

Header files should be created in a structured and consistent fashion. This is important for other programmers who will use your classes. Choosing a consistent "look and feel" means that it is relatively easy to read, understand and use your classes. The header file should contain the following information:

- The (disk) name of the file representing the class.
- A description of what the class is intended to do.
- The state of the class (implemented in the private part).
- The class member functions divided into categories.
- Using tactical and strategic comments to document member functions and groups of member functions.

Header File Model

Point.hpp

```
// point.hpp
// Description of class

class Point
{

}; // DON'T FORGET THE SEMICOLON !!!
```

Class Syntax

- Description of data members
- Prototype for member functions
- Access specifiers describe accessibility to members for client code
- Semi colon (;) at end of class !

13

CLASS SYNTAX

The header file contains the description of the user defined type. This description consists of the declaration of the data members and the prototype of the member functions. The class description starts with the '{' and ends with the '};'. All declarations of variables and functions between these brackets are respectively data members and member functions.

Point.hpp

```
class Point
{
      // Data members
      double x;

      // Member functions
      double ret_x();
};
```

Access Specifiers

- Specifies how that member may be used and accessed
- public:
  - Accessible to clients (member functions)
- private:
  - Accessible only to own member functions (data members)

14

ACCESS SPECIFIERS

The information hiding principle and encapsulation are both supported in C++. C++ implements these by the keywords 'private' and 'public'. Class members which are declared in the private area are visible only to the other instances and members of the class. All other objects ('the world') have no access. Class members which are declared in the public area can be accessed by any object or function. The public area is the implementation of the services that a given class offers to the world.

The private area is used mainly for state information about an object. It contains declarations of the data types that represent the attributes of an object. This area may also contain the declaration of member functions (private member functions) that may not be used by clients.

Point.hpp

```
class Point
{
private:
  // Data members, not accessible to clients
  double x; // X value of point
  double y; // Y value of point

public:
  // Member functions, accessible to clients

  // Constructors & destructor
  Point();
  ~Point();

  //Selectors
  double ret_x(); // Access the x value
  double ret_y(); // Access the y value

  // Modifiers
  void set_x(double newxval); // Set the x value
  void set_y(double newyval); // Set the y value
};
```

Excluding Multiple Inclusion

- Header file can be included multiple times in the same source file
- Prevention using pre-processor directives

```
#ifndef POINT_HPP
#define POINT_HPP

class Point
{
};

#endif
```

15

```
// point.hpp
// Simple two dimensional point consisting of a x and y
// value

#ifndef POINT_HPP
#define POINT_HPP

class Point
{
private:
      double x;         // X value of point
      double y;         // Y value of point
public:
      // Constructors and destructor
      Point();          // Default constructor
      ~Point();         // Destructor

      //Selectors
      double ret_x();         // Access the x value
      double ret_y();         // Access the y value

      // Modifiers
      void set_x(double newxval);   // Set the x value
      void set_y(double newyval);   // Set the y value
};

#endif
```

The C++ Source File

- Contains
  - Description of the class
  - Modification dates
  - Inclusion of the class header file
  - The body of member functions

16

```
// point.cpp
// Simple two dimensional point consisting of a x and y
// value.
//
// Modification dates:
// 01-01-2011 AM Kick off simple example
//
// (C) Datasim BV 2011

#include "point.hpp"

// Member functions
```

The Source File and Member Functions

- Specify the class name with each member function by using the scope resolution

```
double Point::GetX()
{
  return m_x;
}
```

17

THE SOURCE FILE AND MEMBER FUNCTIONS

Each member function that is declared in the header file must have its implementation in the class code file. Furthermore, the code file is the place where the program modification details should be documented.
The fully qualified name of the function must be used and this is achieved by the use of the scope resolution operator '::'. It is also important to provide ample commentary to show how difficult and more obscure parts of the code work.

Point.cpp

```
// point.cpp
// Simple two dimensional point consisting of a x and y
// value.
// Modification dates:
// 01-01-2011 AM Kick off simple example
//
// (C) Datasim BV 2011

#include "point.hpp"

//Selectors

// Access the x value
double Point::ret_x()
{
      return x;
}

// Access the y value
double Point::ret_y()
{
      return y;
}
```

Point.cpp

```
// Modifiers

// Set the x value
void Point::set_x(double newxval)
{
      x = newxval;
}

// Set the y value
void Point::set_y(double newyval)
{
      y = newyval;
}
```

Using the class

- Include the header file where you use the class

```
#include "point.hpp"
```

- Create objects by declaring variables of the new type

```
Point pt;
```

- Call member functions using a period (.)

```
pt.SetX(10.0);
```



```
// smptst.cpp
// Simple test program for the point class.
//
#include <iostream.h>
#include "point.hpp"

void main()
{
      Point pt1;        // First test point
      Point pt2;        // Second point

      pt1.set_x(10.0);
      pt2.set_x(30.0);

      cout << "X value p1 : " << pt1.ret_x() << endl;
      cout << "X value p2 : " << pt2.ret_x() << endl;
}
```

Tips for Class Declaration

- Divide the member functions into categories

```
class SomeClass
{
  // Constructors and destructor

  // Accessing functions

  // Modifiers

};
```

19

TIPS FOR CLASS DECLARATION

Objects have a state and an interface; this interface consists of the messages that can be sent to objects of a class. When writing code for a new class we advise on creating categories of member functions. This has the advantage that users of the code can read the class header file more easily than if member functions are declared in an arbitrary and unstructured manner.

We have chosen for the (somewhat arbitrary) categories:

- Constructors; these are special member functions which create instances of the given class.
- Destructor; there can be at most only one destructor per class. Destructors are only necessary for objects that have allocated memory on the heap (free store) but may be used for any class.
- Selectors; these are functions which do not change the state of an object.
- Modifiers; these functions change the state of an object.

The above categories can be placed in either the private or public areas. Where you place your functions will depend on whether you wish clients to access them or not.

Tips for Member Functions

- Keep member functions short
- Create member functions in discrete blocks
  - Preconditions
  - Function code
  - Postconditions

20

TIPS FOR MEMBER FUNCTIONS

When you create member function there are a few things to watch for:

- Do not create to many member functions. When a class has more than 40 member functions there could be something wrong with the design. Often these kinds of classes, also called fat classes, do more than they need to do.
- When creating a member function try to divide it into three separate sections. The pre-conditions to check the input values. The function code to use these values. And finally the post conditions to check the values calculated or return these.