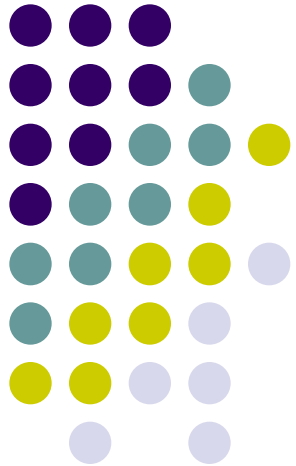
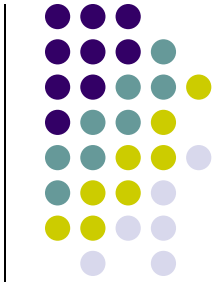
Object Oriented Programming and C++

Xiaoqiang Wang

Florida State University

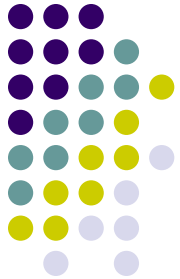


Introduction to C++



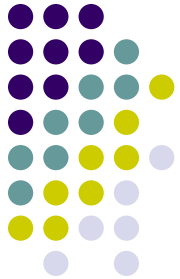
- **Objectives**
- To discuss some C++ language features.
-
- **Topics**
- Introduction to C++
- Comments
- I/O streams
- Program scope
- Scope resolution operator
- Constants
- Reference arguments
- **new** and **delete** operators

Why OO?



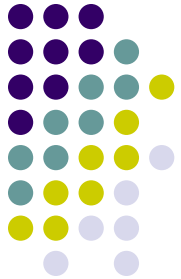
- OO programming models the real world with groups of interacting objects
- Allows code reuse through client/supplier and inheritance relationships, class libraries, templates.
- Allows faster software development
- Allows development of software that is more error free, and reliable
- Allows lower production costs

Why C++?



- C++ is a superset of C
- C++ is the major commercial OO language
- C programs should compile under C++
- C++ is not a pure OO language like Smalltalk and Eiffel
- C++ allows efficient implementation of OO systems
- Existing non OO C systems can have OO C++ parts added
- C++ has all the features required of an OO language

Some OO features of C++



- *Classes*

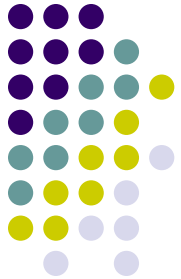
  Classes can contain both data and operations

- *Encapsulation*

  Data and functions in classes can be hidden

- *Composition*

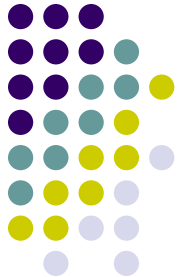
  A class can be composed of other classes



- *Inheritance (class derivation)*
  New classes can be obtained from other classes

- *Polymorphism*
  C++ allows all forms of polymorphism:
  overloading
  pure polymorphism
  parametric polymorphism: templates

Specific language features



- **Comments**
- Both C style comments ***/* ..... */*** are allowed as well as comments starting with ***//***

- Can use ***/* ...*/*** for multiple line comments
- ***//*** can be used as in line comments

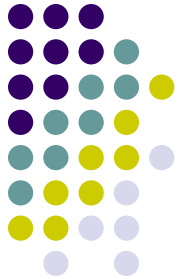
  long numTribbles; // will increase!

- ***Programming recommendation***

  Always use // comments

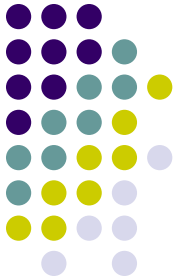
  This allows sections of code to be commented out with /* ... */

Introduction to iostream



- Input and output are not part of the C++ language.
- They are supported by the *iostream library*.
- This predefines three streams for I/O:
  - **cin** - standard input (terminal)
  - **cout** - standard output (screen)
  - **cerr** - standard error
- The << and >> (put to or get from) operators direct output to a stream or retrieve data from a stream.
- A stream is a sequence of characters.

cout



- The following statement displays **I'm here** on the screen

  cout << "I'm here" << endl ;

- Flow of information from right to left
- **endl = "\n"** = new line
- Multiple **<<** operators can be concatenated.
- The operator **<<** knows how to display values - no need for a string to specify the data types.

  Compare to

  printf ("%s\n", "I'm here");

cin

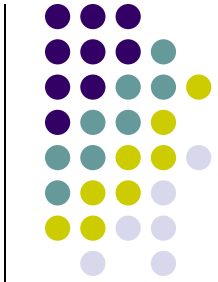
cin >> name;

- Multiple >> operators can be concatenated
  cin >> num1 >> num2;
- Flow of information is from left to right

- Example demonstrates both input and output :

```
#include <iostream.h>
int main (void)
{
    int num1, num2;
    cout<<"Enter two numbers:";
    cin >> num1 >> num2;
    cout << " The sum is " << num1 + num2 << endl;
    return (0);
}
```

Program Scope



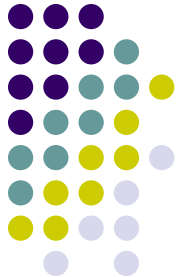
- The scope of an identifier is that part of a program where the indentifier can be used.
- C++ supports three kinds of scope:

  1. *File scope*
     - File scope is that part of a program that is not contained in a function or a class.

  2. *Local scope*
     - Local scope is that part of a program that is within the definition of a function.
     - This includes the argument list.
     - In a function, each compound statement maintains its own associated local scope.

  3. *Class scope*
     - Each class represents a distinct class scope.

Local Scope



- In C++, variables can be declared where they are needed, and do not have to be declared at the beginning of a function.
- However, they can only be used after they have been declared.
- A local variable goes out of scope when the function terminates or a compound statement finishes execution.

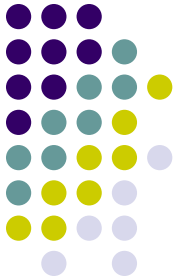
```
// The declaration of j below is not local to the
// for loop, but the declaration of k is local
void Demo (void)
{
    for (int j = 0; j < 10; j++)
    {
        ...
        int k; // k is local to the for loop
        ...
    } // End for
    ...
    int k; // This is a different k
} //End Demo
```

Class Scope

```
int height;
class Bar
{
    public:
        Bar (void);
        Draw (void);

    private:
        short height;
};
```

- Here **height** is in scope within the whole class body.
- It is as if **height** had been declared at the top of the class.
- Member functions are also within the class scope.
- How do we refer to class members outside the class declaration?
- We use the *scope resolution operator*.

The scope resolution operator

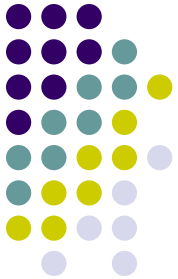


- This has the symbol **::**.
- A member function is referred to outside a class declaration by using the class name and the scope resolution operator.
- For the **Bar** example above, the definition of **Draw** is written

  Bar::Draw (void) { ... }
- An identifier prefixed with the **::** operator will access the global (file scope) identifier

```
int max = 1000; // global declaration
void fun (void)
{
    int max; // local declaration
    max = 100; // accesses local max
    if (count > ::max) {...} // accesses global max
}
```

Constants



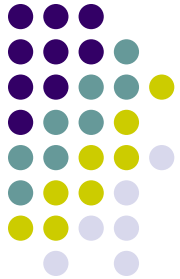
- In C++ a constant can be defined with the **const** reserved word.
- Examples

  const int kBufferSize = 256;

  const int kMaxValues = 100;

  const float kEpsilum = 1.0e-08;
- You can not change constants**.** Put a const in front of the variable to make sure its value will never get changed.

Reference arguments

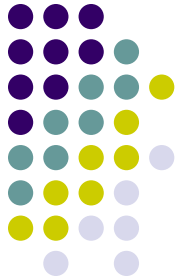


- Reference arguments act in a similar manner to **var** parameters in Pascal.
- They can be used in C++ where normally you would pass a pointer to a variable in order to modify it.
- A reference argument is always a local name (or synonym) for its actual argument (whatever that argument may be).
- The output for the following code is:

  Initial Value of j 10
  After DoubleValue 10
  After DoublePointer 20
  After DoubleReference 40

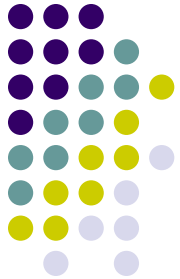
  - **void DoubleValue(int i) {i = 2 * i;}**
  - **void DoublePointer(int *i) {*i = 2 * (*i);}**
  - **void DoubleReference(int &i) { i = 2 * i;}**

Reference arguments



- Reference arguments override the normal pass by value semantics used in C.
- Use reference arguments when
  1. It is necessary to change the values of the actual arguments.
  2. Large objects are being passed, because no local copy of the actual argument is made.
- Note the syntax when using reference variables.
- No address of operator (**&**) is required in front of the formal arguments when the function is called.
- No de-referencing operator (*) is required when referring to the formal arguments inside the function.

The new and delete operators



- In C, memory allocation typically involves a call to **malloc ()**, which is paired with **free ()** to deallocate the memory.
- In C++ **new** and **delete** perform similar operations
- (**new** == **malloc** + constructor calls, see later)

  For example, if we have defined a class **Sphere**

  Sphere* spherePtr;

  the statement

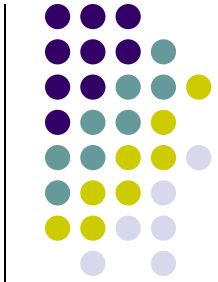
  spherePtr = new Sphere;

  allocates a block of memory to hold a **Sphere** object.

  To free up the memory just allocated use

  delete spherePtr;

Some points about new and delete

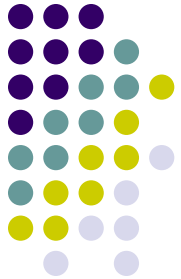


- Always call **delete** when you are finished with a pointer whose memory was allocated with **new**. The memory is not automatically deallocated when the pointer goes out of scope.
- **new** and **delete** can be used with classes as well as the inbuilt types of C++.
- Use **new** and **delete** for pointers to variables (variables themselves have automatic memory allocation and deallocation).
- Only use **new** and **delete**

  Reasons

  1. **malloc/delete** and **new/free** combinations are undefined.
  2. **malloc** and **free** do not know anything about constructors and destructors. (See later).

C++ Class Basics



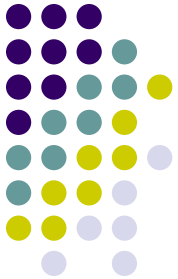
Objectives

- To discuss the basics of writing classes and creating and deleting objects in C++.

Topics

- What are classes
- Class relationships
- Class components
- Writing class functions
- Accessing member functions
- Constructors and destructors
- File structure for C++ programs
- The Rectangles program

What is a class?



A C++ class has four attributes

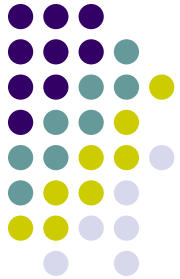
1. A collection of **data members**
   This is the *implementation* of the class

2. A collection of **member functions**
   The set of operations that can be applied to objects of the class
   These are the class *interface*.

3. Levels of program access
   Members may be specified as **private**, **protected** or **public**
   These control access to the members from within the program

4. A class **tag name**
   This serves as a *type specifier*

Encapsulation



- Usually, the implementation of a class is private, and the operations (the interface) are public.

- A private implementation is said to be ***encapsulated***

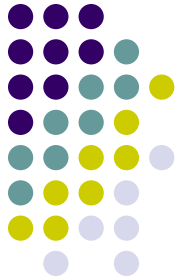
- Another term for a private implementation is

  information hiding,

  or

  data hiding.

Abstract Data Types (ADT's)

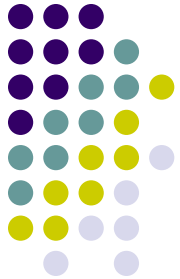


- A class with a private implementation and a public set of operations is known as an

  abstract data type

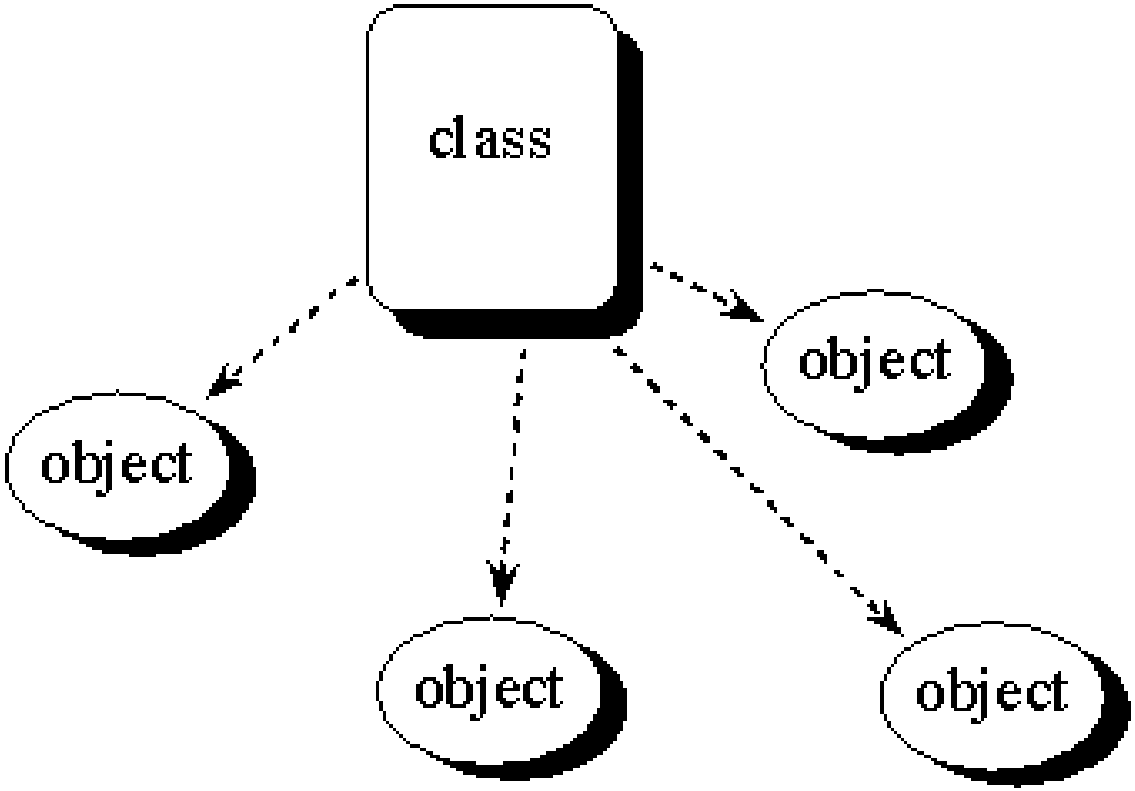
- The user of a class does not have to know how it is implemented, eg a Triangle class.
- The writer of a class can change the implementation without this affecting the user.

Objects



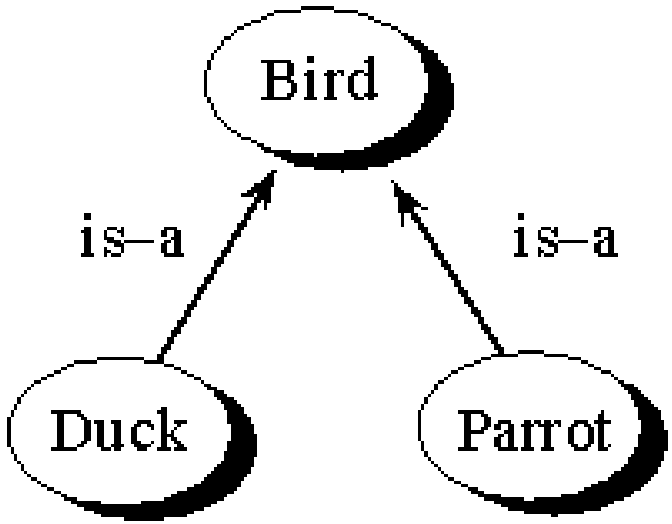
- Running OO programs consist of a number of interacting objects.
- Where do the objects come from?
- The objects are *instantiations* of the classes.
- The class defines the behaviour of the objects through its public interface, and is used to create the objects.
- As an OO program executes, a single class can be used to create and destroy many objects.

A class acts as a *blueprint* for new objects

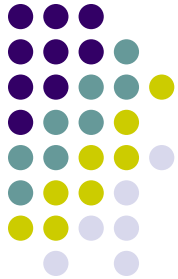


Inheritance (class derivation)

- This implements the ***is-a*** relationship between classes
- A derived (child) class aquires data members and member functions from its base (parent) class.
- It can use these as is, modify them, or add new members of its own.

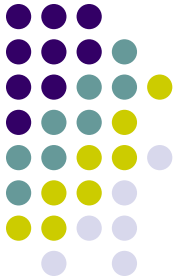


How do we write classes in C++?



- Classes are usually written in two parts: *declaration* and *definition*.
- The first part is the class *declaration*, which lists the following information about the class
  - The class name
  - The names and types of all the data members
  - The names, argument types, and return types of all the member functions
  - The level of access for each data member and member function
- The second part is the class *definition*, where the code is given for all the member functions.
- These two parts are normally written in two separate files

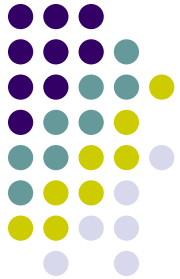
Example of a class declaration



```
class Rectangle
{
    public:
        Rectangle (int w, int h); // The constructor
        int Area(void); // The area function
        ~Rectangle (void); // The destructor
    private:
        int width;
        int height;
};
```

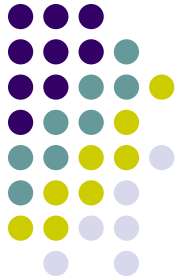
- Note the ; at the end, which must be present.

Constructors



- The constructors have the same name as the class itself.
- They build objects by initialising the data members to anything you want them initialised to.
- When an object is created, one of the constructors will be called automatically.
- Never call a constructor directly.

Definition code



- Here is the *definition* (code) for the constructor

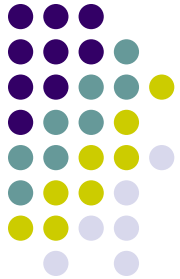
```
Rectangle::Rectangle (int w, int h)
{
    width = w;
    height = h;
}
```

- The constructor takes two arguments which are used to initialise the **width** and **height**.
- There is no return type, as constructors do not have a return type.
- Here is the *definition* (code) for **Area**

```
int Rectangle::Area (void)
{
    return (width * height);
}
```

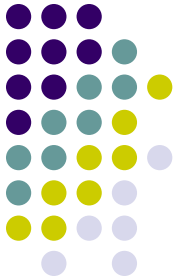
- A member function automatically has access to all data members and member functions of the class it belongs to.

Destructors



- Each class should have one destructor ***even if the destructor is empty***.
- The destructor is called *automatically* when an object is destroyed.
- The destructor has the same name as the class itself, with a tilda (**~**) placed before it.
- There is no return type, as destructors can not return anything.
- The destructor destroys the object by freeing up memory, closing files, etc.

Destructors

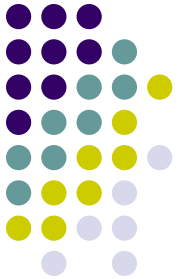


- The destructor is called automatically when
  - An automatic object goes out of scope
  - A object created dynamically with **new** is **deleted**.
- Never call a destructor directly
- The destructor defintion is

```
Rectangle::~Rectangle (void)
{
// This destructor is empty
}
```

- This destructor is empty because the data members width and height are automatic variables, which are deleted automatically when the object is destroyed.
- If Rectangle contained data members which were pointers, these would not be deleted automatically, and the destructor would have to delete them.

Creating an object of type



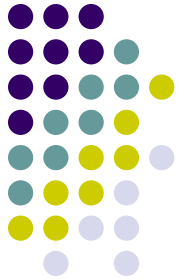
2. Define a pointer to the object
   Rectangle* rectPtr;

   and then create the object by calling **new**

   rectPtr = new Rectangle (5,7);

- The operator **new** allocates memory for the **Rectangle** object and returns a pointer to the object

Accessing an Object's Member functions



- For an object itself, use the dot (.) notation.

  rectangle1.Area ();

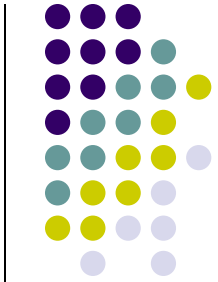
- For a pointer to an object, use the **->** operator

  rectPtr->Area ();

  which is shorthand for

  (*rectPtr).Area ();

Deleting an object



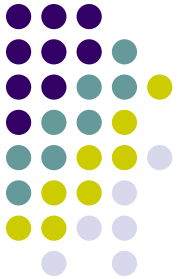
- An object created automatically:

  Rectangle rectangle1 (5,7);

  is automatically deleted when the object variable goes out of scope. For example, if the object is declared in a function, and the function terminates, the object is automatically deleted.

- The same applies to the variables of the built in types: **int**, **char**, etc.

Deleting an object



- An object created with **new**:

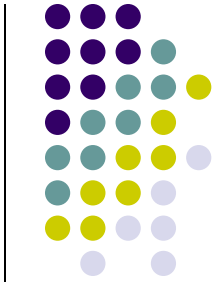
  Rectangle* rectPtr (5,7);

  must be explicitly deleted with the **delete** operator to release the memory.

  delete rectPtr;

- If this is not executed, the memory is not released when the object goes out of scope.
- Calling **delete** automatically calls the destructor.

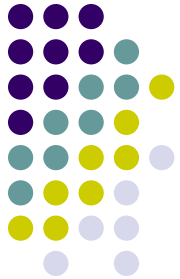
Member functions that all classes should have



- All classes you write should have
  - A *default constructor*
  - A *copy constructor*
  - An *assignment operator*
  - A *destructor*
- A default constructor has no arguments.
- Classes with these four member functions are known as ***concrete data types***. More on these later

- If you don't write your own constructor and destructor, The C++ system will write a constructor and a destructor for you, if required. These may not be the constructor and destructor you want.

File structure for C++ programs



Put the code for each class you write in two files

1. A header file that contains the class *declaration* and any #includes required for using the class.

   Call this file <ClassName>.h

   eg Rectangle.h

   The header file for a class must contain all the information a user requires to use the class.

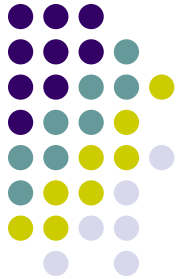
2. A file that contains the *definition* of all the member functions of the class and a #include for the class header file.

   Call this file <ClassName>.cc

   eg Rectangle.cc

   The user should not have to refer to the .cc file to use the class.

File structure for C++ programs



- In addition, each C++ program must have a file that contains

  the main function

  required #includes

  object declarations for running the program.

- The rectangles program discussed below consists of three files:
  - Rectangle.h
  - Rectangle.cc
  - main.cc

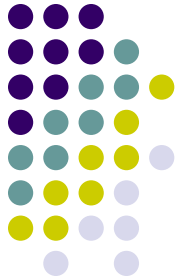
Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
    public:
        Rectangle (int w, int h);  // Constructor
        int Area (void); // The area
        ~Rectangle (void); // Destructor
    private:
        int width;
        int height;
};
#endif
```

Note:

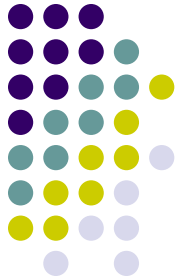
The use of pre-compiler directive: This is used to prevent multiple inclusions of the definition in a given file. You must use a mechanism like this with each of your header files.

Rectangle.cc



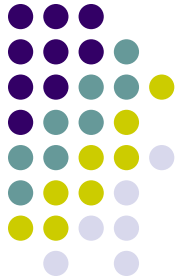
```
#include "Rectangle.h"
Rectangle::Rectangle (int w, int h)
{
   width = w;
   height = h;
}
int Rectangle::Area (void)
{
   return (width * height);
}
Rectangle::~Rectangle (void) {}
```

main.cc



```
#include <iostream.h>
#include "Rectangle.h"
int main (void)
{
    Rectangle small (5,7);
    // This calls the constructor
    Rectangle medium (8, 8);
    // This calls the constructor
    Rectangle large (15, 20);
    // This calls the constructor
    cout << "The small rectangle area is " << small.Area () << endl;
    cout << "The medium rectangle area is " << medium.Area () << endl;
    cout << "The large rectangle area is " << large.Area () << endl;
    return (0);
}
```

Designing Concrete Data Types



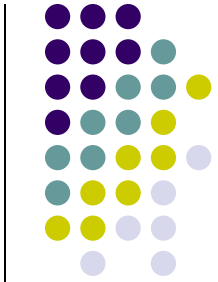
Objectives

- To discuss the design of concrete data types

Topics

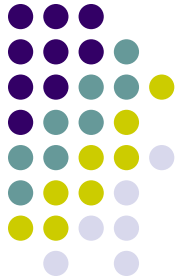
- Requirements for concrete data types
- The **String** class
- Copy constructors
- The assignment operator

Concrete Data Types



- The C++ inbuilt types such as **int** and **double** have the following properties:
  - They are well behaved. For example, there are no memory leaks
  - They can be passed as parameters to functions (using pass by value or pass by reference).
  - Their values can be returned by functions
  - They can be assigned to each other using chaining

Concrete Data Types

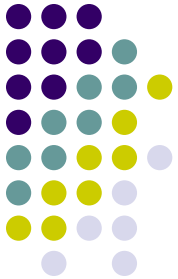


- The data types you define should have the same properties. In this case they are called

concrete data types

- Concrete data types can be built out of the inbuilt types and other concrete data types.
- They can have their own operators defined on them.
- Only by creating concrete data types can you create correct, safe, and efficient C++ programs of arbitrary complexity.
- If your objects don't behave properly, your programs can't either.

Concrete Data Types



- Concrete data types have the following member functions
  - A default constructor
  - A copy constructor
  - An assignment operator
  - A destructor
- If you don't supply these, the compiler will supply them for you, if required. **You may not like the results.**

The String class

```
class String
{
    public:
        String (void);
        String (const char* value);
        ~String (void);
    private:
        char* data;
};
```

- **This String class has**
  - **A default constructor**
  - **A second constructor that can take a C string as an argument**
  - **A destructor**

Hence, it is not a concrete data type

The default constructor

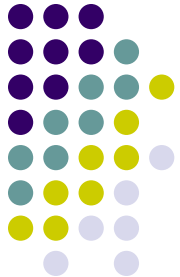
```
// The default constructor
String::String(void)
{
   data = new char[1];
   *data = '\0';
}
```

- This allows a default string object to be created with the definition

  String a;

  The string a will contain only the null character.

The second constructor

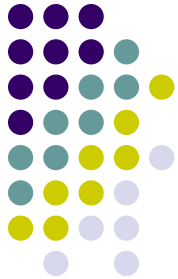


- The second constructor allows a string object to be initialised with an ordinary C string

  String b("Hello");
- The string **b** will contain the characters **Hello\0**

```
// The second constructor
String::String(const char* value)
{
    if (value) {
        data = new char[strlen(value) + 1];
        strcpy(data, value);
    } else
    {
        data = new char[1];
        *data = '\0';
    }
}
```

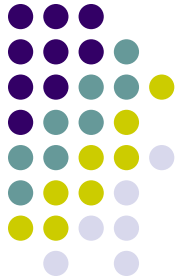
The destructor



- The destructor is responsible for freeing the memory accupied by the ordinary C string that stores the characters.
- The **String** class **must** have a destructor because **data** was allocated dynamically with **new** in the constructor.
- This memory is not automatically deallocated when the string object goes out of scope.

```
// The destructor
String::~String(void)
{
    delete [ ] data;                    //with new [ ]
}
```

Assignment problem?

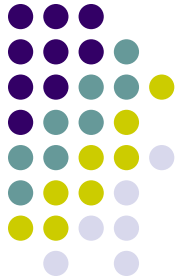


- Consider:
  String a(“I’m");
  String b(" here");

- **Now consider the following assignment**
  b = a;

- **There is no assignment operator defined for String, so the compiler generates a simple default assignment operator and calls it.**
- **This assignment operator does a *bitwise copy* only.**
- **This means it only copies the *address* of any (pointer) data members allocated dynamically with new.**

What is wrong with this?



1. The memory that **b** pointed to was not deleted.

   This is a *memory leak*.

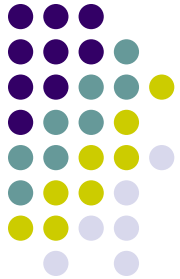
2. Both **a** and **b** contain pointers to the same character string.

   When one goes out of scope, its destructor will delete the memory pointed to by the other.

Example

```
String a (“I’m"); // Declare and create a

{ // Open new scope
    String b (“ here"); // Declare and create b
    ...
    b = a; // Execute default operator=
    // lose b's memory

} // Close scope,
// call b's destructor

String c = a; // c.data is undefined
// a.data is already deleted
```

Note



In the statement

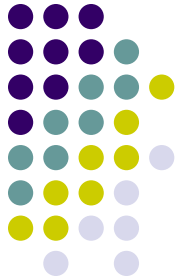
String c = a; // Definition, calls copy constructor

c is *initialised* with **a**.

This is different from

String c ; // Definition, calls default constructor
c = a; // Calls assignment operator

Copy constructors



A copy constructor is used to initialise an object with another object of the same type.

Copy constructors are invoked in the following three situations

(1) Explicit copy during initialisation

For automatic variables

String b = a ;

Here **b** is initialised with **a**.

For object pointers

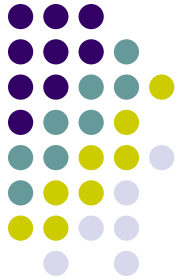
String* b ;

b = new String (a) ;

(2) Pass by value for function arguments

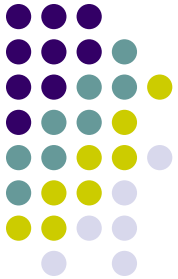
(3) Return by value for functions

Example



```
class X
{
	public:
		X (void) { cout << "Default constructor" << endl; }
		X (const X& x) { cout << "Copy constructor" << endl; }
};

X ReturnX (X b) // Pass by value
{
	X c = b; // Explicit copy
	return (c); // Return by value
}

int main (void)
{
	X a;
	cout << "Calling ReturnX" << endl;
	X d = ReturnX (a);
	cout << "Back in main" << endl;
	return (0);
}
```

Output



Default constructor // main's a

Calling ReturnX

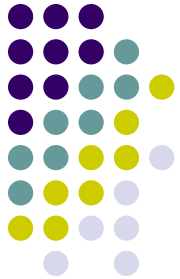
Copy constructor // ReturnX's b

Copy constructor // ReturnX's c

Copy constructor // main's d

Back in main

Writing Copy Constructors



A copy constructor for the **String** class has the following declaration

String (const String& s) ;

The parameter **s** is a reference to a string.

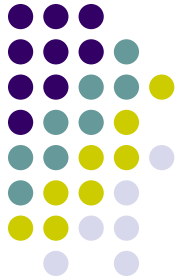
This means the copy constructor has the actual string **s** to work with, but can't alter it because of the **const** declaration.

const – s will not be changed.

The copy constructor should make an exact copy of the string **s**.

It should copy each data member of **s** into the corresponding data member of the new string.

Writing Copy Constructors

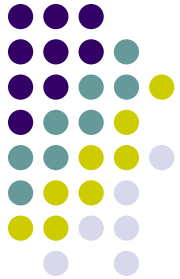


```
String::String (const String& s)
{
    data = new char [strlen (s.data) + 1];
    strcpy (data, s.data);
}
```

- **The following two fragments of code will now work correctly**

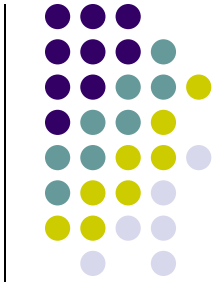
```
(1)
String a (“I’m here") ;
String c = a ;

(2)
String a (“I’m here") ;
String* stringPtr ;
stringPtr = new String (a) ;
delete stringPtr ;
```

The implicit ‘this’ pointer



- Each member function of a class maintains a hidden pointer to the object that the function was invoked on.
- The hidden pointer is called **this**, and is of the same type as the object.
- For example, for the **Rectangle** class **this** is of type pointer to **Rectangle**. (**Rectangle* this;**)
- Programmers can refer to **this** explicitely.
- For example, within a member function of **Rectangle** we can use the notation
- **this->width**, which is the same as **width**.
- The implicit **this** pointer has some important uses. See later.

Assignment operator



With the inbuilt data types, we can make assignments like

int a = 0 ;

int b ;

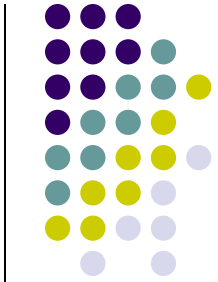
b = a ;

and *chain* assignments together

int a, b, c, d ;

a = b = c = d = 0 ;

Assignment operator



We should be able to perform similar operations with user-defined types too:

(1)

String a ("Hello") ;

String b ;

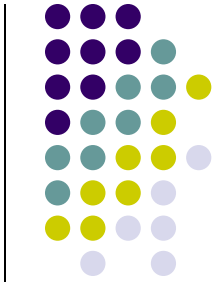
b = a ;

(2)

String a, b, c, d ;

a = b = c = d = "Hello" ;

Assignment operator



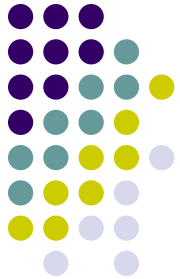
This involves *overloading* the **=** operator for the **String** class.

To perform (1) we use the default version of assignment operator, which has the declaration

String& operator= (const String& rhs);

This **=** operator takes a reference to a **String** as the argument, and returns a reference to a **String**.

Notes



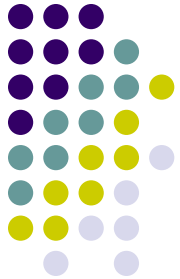
(1) The assignment operator needs to check for assignment to self (see below).

(2) The assignment operator must delete any memory allocated with **new**, before allocating new memory.

(3) In general, make sure you assign to all data members in the assignment operator.

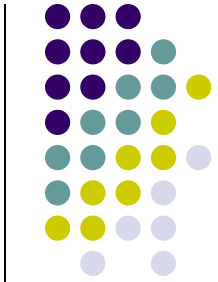
(4) The assignment operator must return a reference to ***this** to allow chaining of assignments.

Code for the assignment operator



```
String&
String::operator= (const String& rhs)
{
    // Check assignment to self
    if (this == &rhs)
        return (*this);

    delete [ ] data; // delete old memory
    data = new char [strlen (rhs.data) + 1];
    strcpy (data, rhs.data);

    return (*this);
}
```

Checking for assignment to self



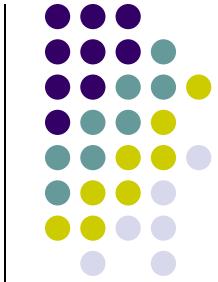
Example

a = a; or **a = b;** where **b** is another name for **a**.

Failure to check for this situation results in disaster.

Calling strlen on rhs.data is then undefined

A Question



The assignment operator must check that the two strings are the same, but what does this mean?

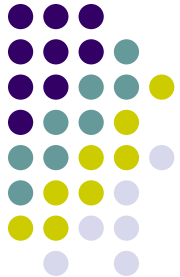
This could mean that the contents are the same or the addresses are the same.

The standard way assignment operators are written is to use the addresses

```
if (this == &rhs)
    return (*this)
```

This is simple to test for (for arbitrary objects, not just strings), and usually safe, unless multiple inheritance is used.

Assignment operator



The assignment operator needs to return a reference to the object on the lhs of the assignment expression.

In the function this is ***this**.

We do this to allow *chaining* of assignments

a = b = c;

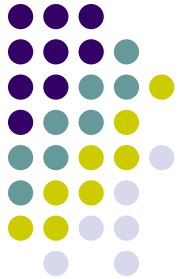
Here we need the return type to be the same as the argument.

Since the assignment operator is right associative, we can write the above assignment statement in functional form as

a.operator = (b.operator = (c));

Working from the right, the value returned from each operator becomes the input to the next operator.

Automatic type conversion



For objects of type **String** we also need to make assignments of the form

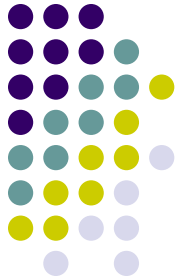
a = "Hello";

Here we are assigning *an ordinary C string* to a **String** object.

As it turns out, the assignment operator on page 34 will handle this case too, through *automatic type conversion*.

In this case compiler first calls the second constructor, which takes a C string as an argument, and converts it to a **String**, before calling the assignment operator.

Summary



- Always write your classes as concrete data types, which have the following member functions.
  - A default constructor
  - A copy constructor
  - An assignment operator
  - A destructor

Acknowledgement

- Most Slides from Kevin Suffern.