

C++ Refresher/Overview (1)

- **Classes**
 - Data members
 - Member functions
- **Access control**
- **Constructor**
- **Destructor**
- **Overloading**
 - Functions
 - Operators
- **I/O streams**
 - Operators:
 - <<
 - >>

C++ Refresher/Overview (2)

- **Static**
 - Data members
 - Member functions
- **References**
- **Copy constructor**
- **Dynamic objects**
 - new
 - delete
- **Inheritance**
- **Polymorphism**
- **Templates**

Other C++ Features

- **Exception handling**
- **New casts**
- **Namespaces**
- **Miscellaneous**
 - Inline functions, RTTI, conversion operators, etc.

Classes

- **Similar to C struct**
 - Data members
 - Used to implement attributes
- **Add member functions**
 - Used to implement behavior
 - Group data & procedures together

Stack ADT Definition

```

class Stack
{
    public:
        void Push (int val);
        int Pop ();
        bool IsEmpty ();
    private:
        int elems [100]; // ??
        unsigned int count;
};

```

Member functions

Data members

Stack ADT Usage

```

Stack stack1;
Stack stack2;
stack1.Push (3);
stack1.Push (-5);

if (stack2.IsEmpty ())
    cout << "Stack 1 is empty."
        << endl;

for (ii = 1; ii <= 6; ii++)
    stack1.Push (ii);

```

Implementation

```

int Stack::Pop ()  Name scoping
{
  assert (count > 0);
  return elems [--count];
}

bool Stack::IsEmpty ()
{
  return (count == 0);
}
Direct member access

```

Object Pointer: this

```

stack1.Push (ii);
Hidden pointer
argument passed
to member
function

void Stack::Push (int element)
{...}

```

C++ Access Control

- **Public**
 - Member visible outside class
- **Private (default)**
 - Visible only inside member functions
 - Friends
- **Protected (more later - inheritance)**

C++ Access Control

- **Multiple access blocks**
 - Each preceded by keyword
 - No required order, duplicate blocks ok
- **Default access**
 - class - private
 - struct - public

Only difference between class and struct!!

Initialization

- **Our C++ stack ADT**
 - Declared like a normal variable
 - But how did "count" get init'ed?
- **Automatic initialization?**
 - Compiler knows when created

Constructor

Also known as "ctor"

- **Special member function**
 - Must still be declared (in most cases)
- **Name same as class name**
- **Automatically called**
 - When instance (object) is created
- **Destructor (for object cleanup)**

Constructor & Destructor

```

Stack::Stack ()
{
    count = 0;
}

Stack::~Stack ()
{
    // Do cleanup      Destructor
    ...
}

```

Function Overloading

- **Functions are operations**
 - Especially member functions
- **Operation variants exist**
 - Print data item (int, float, char, etc.)
 - Multiple functions, same name
 - Different “signatures”

Overload Example

```

class Stack
{
    ...           Function “signature”
                (pattern of argument types)
public:
    ...
    void Push (int element);
    void Push (int elem1, int elem2);
    void Push (double elem);
}

```

Overloaded Constructor Use

```
Complex c0; // Default ctor
Complex c1 = Complex (1.0);
Complex c2 = Complex (1.0,2.0);

Complex c1a = 2.2;
Complex c1b (4.4);
Complex c2a (2.2, 3.3);
```

Operator Overloading

- **Operator is just a function**
 - Syntactic convenience ("sugar")
- **Can overload functions**
 - By argument types (signature)
- **So, can overload operators**
 - At least one user-defined operand

What We Want

```
Complex c1;
Complex c2;
Complex c3;

c3 = c1 + c2; // binary op
c2 = -c1;     // unary op
c1 = c3;     // assignment
c2 += c1;
```

Operator Functions

- **Special function name**
 - Keyword operator plus symbol
 - E.g., operator+, operator<<
- **Operator function syntax**
 - Unary vs binary operator
 - Global function vs member function

Member Binary +

```
class Complex
{
public:
...
  const Complex operator+
    (const Complex& c) const;
private:
  double re;
  double im;
};
```

Member Binary +

```
const Complex Complex::operator+
  (const Complex& c) const
{
  return Complex (re+c.re,
                 im+c.im);
}
```

I/O Operators

- **Insertion - << (like printf, ...)**
 - Formats data types into bytes
 - Puts bytes into stream
- **Extraction - >> (like scanf, ...)**
 - Takes bytes from stream
 - Converts representation to data types

I/O Example

```
int main (void)
{
  cout << "Hello there\n";
  cout << "Enter two numbers: ";

  int n1, n2;

  cin >> n1;
  cin >> n2;
  ...
}
```

I/O Example

```
...
cout << "The sum of ";
cout << n1;
cout << " and ";
cout << n2;
cout << " is ";
cout << (n1 + n2);
cout << '\n';

return (0); /* Return success */
}
```

I/O Example

```
Hello there
Enter two numbers: 5 18
The sum of 5 and 18 is 23
```

```
Hello there
Enter two numbers: 14.5 6.8
The sum of 14 and 2695 is 2709
```

Why?

Manipulators

- **endl**
 - Inserts '\n'
 - Flushes output
- **flush**
- **Base control**
 - oct, dec, hex
- **With args**
 - setw()
 - Field width
 - setprecision()
 - setfill()
 - Fill character
- **More ...**

static Keyword

- **Static storage** (same as C)
 - Static versus automatic variable
- **Linkage (visibility)** (same as C)
 - Confined to file scope
- **Static class members**
 - New in C++

Static Members

- **Associated with entire class**
 - Not with individual object (instance)
- **Data member**
 - A “global” variable in class scope
- **Function member**
 - No this pointer argument

Static Data Example

```
class Bunny
{...
private:
    static int how_many; ...
};
int Bunny::how_many = 0;
```

Maintain count
in constructor
& destructor

```
Bunny::Bunny()
{ how_many++; ... }

Bunny::~Bunny()
{ how_many--; ... }
```

Static Function

```
class Bunny
{
public:
    static int HowMany ();
    ...
};
```

Invoke function
with or without
object

```
Bunny bunny1;
int n;
n = bunny1.HowMany();
n = Bunny::HowMany();
```

References

- **Like a constant pointer**
 - Automatically dereferences
- **Acts as alias for an object**
 - Works for built-in types as well
- **Often used with functions**
 - Argument or return value

Pointer vs Reference

```
// pointer
int x;
int* yp = &x;

*yp = 10;
cout << x;
```

```
// reference
int x;
int& y = x;

y = 10;
cout << x;
```

Function Arguments

```
// pointer
void f(int* px)
{
  *px = 25;
}

int y;
f (&y);
```

```
// reference
void f(int& x)
{
  x = 25;
}

int y;
f (y);
```

const Reference

Efficiency of "pass by pointer";
safety of "pass by value"

```
void g(const Complex& c)
{
  cout << c.Real();
}
...
Complex c1; // in "main"
g (c1);
```

Copy Constructor

- **A special form of constructor**
 - Overloaded constructor function
 - Signature: one arg, a class reference
- **Automatically used**
 - Call generated by compiler
 - When passing or returning by value

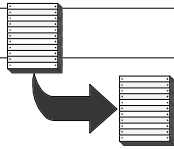
Passing by Value

```
float f (int x)
{
  float y = x + 0.5;
  return (y);
}
```

Compiler must
copy values on
call and return

```
int aa = 2;
float bb;
bb = f(aa)-3;
```

How to Copy?



- **Simple bit copy**
 - Like memcpy
 - Works fine for simple non-ptr types
 - Warning: may be the default!
- **Copy constructor (CC)**
 - Explicit constructor to make copy

“Default” CC

- **Compiler-generated**
 - If you don't declare one
- **Memberwise copy**
 - For built-in types, bit copy
 - For user-defined types, calls member's copy constructor
 - Or compiler-generated “default”

C++ Dynamic Objects

- **Need heap operations**
 - Must be part of language
 - So compiler can invoke constructor and destructor
 - Can't be just library routines
- **Operator new - allocate/construct**
- **Operator delete - destroy/free**

new & delete

```

int main()
{
  Complex* pc1; // pointers!!
  Complex* pc2;
  pc1 = new Complex (1.0, 2.0);
  pc2 = new Complex;
  ...
  delete pc1;
  delete pc2;
}

```

Building Up Classes 

• **Composition**

- Incorporate existing class objects
- Objects become members of new class

• **Inheritance**

- Extend an existing class (“is a”)
- New class based on existing class

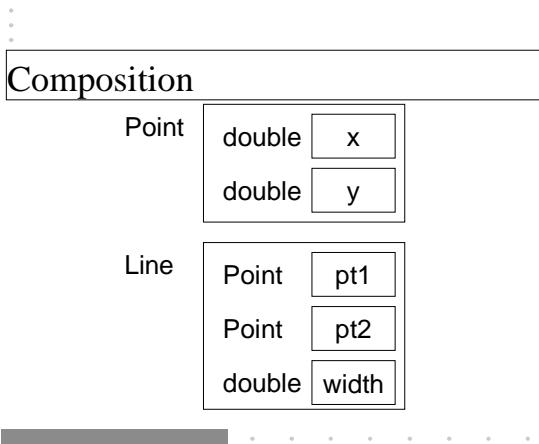
Composition

• **New class has members**

- Built-in types
- User-defined types (classes)

• **Object members**

- Any number of objects
 - Same or different types (even array)



```

class Point
{
public:
    Point (double xval = 0.0,
           double yval = 0.0);
    double GetX() const;
    double GetY() const;
private:
    double x;   Class to describe a
    double y;   point, as specified in
    ...};      Cartesian coordinates.
  
```

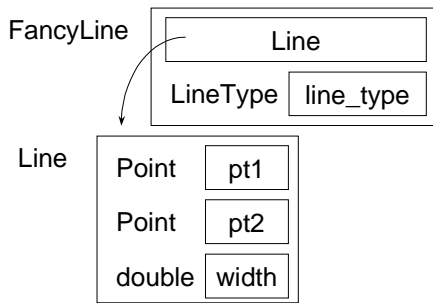
```

class Line
{
public:
    Line ();
    const Point& GetP1() const;
    void SetWidth (double w); ...
private:
    Point pt1;
    Point pt2;   Class to describe a
    };          line, as determined by
               its endpoints.
  
```

Inheritance

- **New class specifies base class**
- **Base is subobject of derived**
 - All base class members are included
- **Derived “is a” base**
 - Base behavior inherited by derived

Inheritance



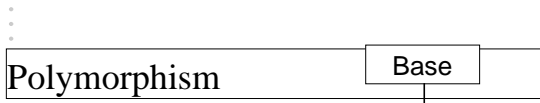
Inheritance

```

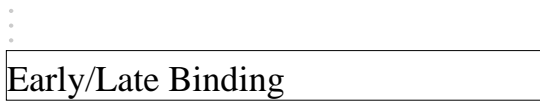
class FancyLine : public Line
{
public:
    enum LineType { SOLID, DASHED };
    void SetWidth (double w);
    ...
private:
    LineType line_type;
    ...
};
  
```



- **Derived “is a” base**
- **So derived can be used**
 - When a base is called for
 - Automatically done by compiler



- **Inheritance**
 - Derived object “is a” base object
- **Common function**
 - Present in both base and derived
- **Which function to invoke?**
 - If derived accessed as a base (upcast)



- **The problem - early binding**
 - Function associated with class
 - Function call bound at compile time
- **The solution - late binding**
 - Choose function at run time
 - Actual object type: even if base pointer

Virtual Functions

- **Function declared in base**
 - With virtual keyword
- **Override function in derived**
 - No virtual needed (but ok)
- **Function call**
 - Invokes most-derived version

Virtual Functions

```
class Sensor
{ public:
  virtual void Calibrate ();
};

class TempSensor : public Sensor
{ public:
  void Calibrate ();
};
```

Templates

- **Parameterized types**
- **Blueprint for class/function**
 - Common generic description
 - Made specific by parameters
- **Instantiated as needed**
 - When compiler detects use

Template (Class)

```

template<class T>
class array
{
    enum { size=100 };
    T A[size];

public:
    T& operator[](unsigned int idx);
};

```

Declare template class with parameter(s)
