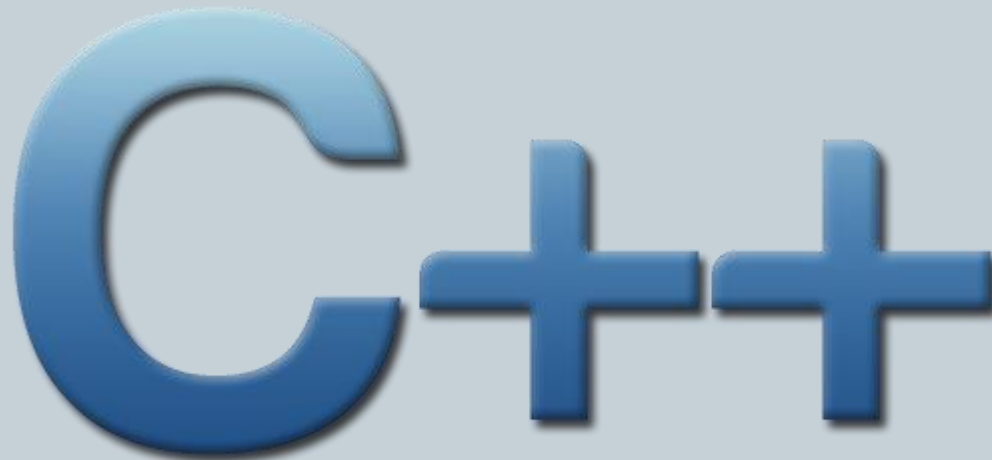


# And Even More and More C++

A large, blue, 3D-rendered logo of the C++ programming language. The 'C' is a large, rounded letter, and the two '+' signs are smaller, positioned to the right of the 'C'. The logo has a slight shadow and a gradient, giving it a three-dimensional appearance.

# Outline

---

- C++ Classes
- Special Members
- Friendship

# Classes

- Classes are an expanded version of data structures (structs)
  - Like structs, they hold data members
  - They also hold functions as members
  - Can specify access permissions also
    - ✦ (Sounding a lot like Java, right?)
- Defined with the keyword “class”:

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

# Classes

- Access Specifiers:

- private

- ✦ Only accessible from within members of the same class or from “friends”

- protected

- ✦ Private access plus members of derived classes can have access

- public

- ✦ Accessible anywhere the object is visible

- Default access is private

# Classes

- An example:

```
class Rectangle {  
    int width, height;  
public:  
    void set_values (int,int);  
    int area (void);  
} rect;
```

- To access data and function members:

```
rect.set_values (3,4);  
myarea = rect.area();
```

# Classes

- The full Rectangle example:

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
    width = x;
    height = y;
}

int main () {
    Rectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

```
area: 12
```

# Classes

- The Scope Operator:

```
class Rectangle {  
    int width, height;  
public:  
    void set_values (int,int);  
    int area() {return width*height;}  
};  
  
void Rectangle::set_values (int x, int y) {  
    width = x;  
    height = y;  
}
```

- Used to define a member function of a class outside of the class definition
  - ✦ area() is defined within the class
  - ✦ set\_values() is defined outside the class
- Scope operator (::) specifies the class to which the function belongs

# Classes

---

- **Classes (just like structs) define a data type**
  - As always, we can have many objects of the class type
  - And each object will have its own member variables and functions that operate on those variables



# Classes

- **Constructors**

- Used to create a new object of the class data type
- Initializes any member variables that need to be initialized
- May do other work if needed
  
- Just like in Java, the constructor function name is the same as the class name
- They cannot be called like regular member functions
  - ✦ They are only executed once – when the object is instantiated
- They have no return values – not even void

# Classes

- Constructor Example

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    Rectangle rect (3,4);
    Rectangle rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

```
rect area: 12
rectb area: 30
```

# Classes

- Overloading constructors
  - Classes can have more than one constructor
    - ✦ All named the same, since they are constructors
    - ✦ But with different parameter lists
      - Remember the method signature in Java?
  - The default constructor is called when an object is declared but a constructor is not specified
    - ✦ This is different than a constructor with no parameters
    - ✦ An example would be appropriate here...

# Classes

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
    width = 5;
    height = 5;
}

Rectangle::Rectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    Rectangle rect (3,4);
    Rectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

```
rect area: 12
rectb area: 25
```

```
Rectangle rectb;    // ok, default constructor called
Rectangle rectc();  // oops, default constructor NOT called
```

# Classes

- Member initialization

- C++ offers a cleaner way of initializing member variables than does Java

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

# Classes

- A subtle constructor example

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
    Cylinder foo (10,20);

    cout << "foo's volume: " << foo.volume() << '\n';
    return 0;
}
```

```
foo's volume: 6283.19
```

# Pointers to Classes

```
// pointer to classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle(int x, int y) : width(x), height(y) {}
    int area(void) { return width * height; }
};
```

```
int main() {
    Rectangle obj (3, 4);
    Rectangle * foo, * bar, * baz;
    foo = &obj;
    bar = new Rectangle (5, 6);
    baz = new Rectangle[2] { {2,5}, {3,6} };
    cout << "obj's area: " << obj.area() << '\n';
    cout << "*foo's area: " << foo->area() << '\n';
    cout << "*bar's area: " << bar->area() << '\n';
    cout << "baz[0]'s area:" << baz[0].area() << '\n';
    cout << "baz[1]'s area:" << baz[1].area() << '\n';
    delete bar;
    delete[] baz;
    return 0;
}
```

expression	can be read as
*x	pointed to by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed to by x
(*x).y	member y of object pointed to by x (equivalent to the previous one)
x[0]	first object pointed to by x
x[1]	second object pointed to by x
x[n]	(n+1)th object pointed to by x

# Classes

- When you define a class, you are defining a new data type, just like in Java
  - This includes the member data and the operators on that data
- Unlike Java, you can overload symbolic operators

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

- These are defined with regular functions with a special name: operator

```
type operator sign (parameters) { /*... body ...*/ }
```



# Classes

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
    CVector temp;
    temp.x = x + param.x;
    temp.y = y + param.y;
    return temp;
}

int main () {
    CVector foo (3,1);
    CVector bar (1,2);
    CVector result;
    result = foo + bar;
    cout << result.x << ',' << result.y << '\n';
    return 0;
}
```

4,3

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

# Classes

- Overloaded operators

- The parameter(s) expected for an overloaded operator are shown below (the “@” sign is just a place holder for the actual operator)

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@ (A)
a@	++ --	A::operator@(int)	operator@ (A, int)
a@b	+ - * / % ^ &   < > == != <= >= << >> &&    ,	A::operator@ (B)	operator@ (A, B)
a@b	= += -= *= /= %= ^= &=  = <<= >>= []	A::operator@ (B)	-
a (b, c...)	()	A::operator () (B, C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE ()	-

# Classes

- Overloaded operators can be member functions – or not!

```
// non-member operator overloads
#include <iostream>
using namespace std;

class CVector {
public:
    int x,y;
    CVector () {}
    CVector (int a, int b) : x(a), y(b) {}
};

CVector operator+ (const CVector& lhs, const CVector& rhs) {
    CVector temp;
    temp.x = lhs.x + rhs.x;
    temp.y = lhs.y + rhs.y;
    return temp;
}

int main () {
    CVector foo (3,1);
    CVector bar (1,2);
    CVector result;
    result = foo + bar;
    cout << result.x << ',' << result.y << '\n';
    return 0;
}
```

4,3

# Classes

- **this**

- Just like Java, C++ uses the keyword “this” to refer to itself
  - ✦ Difference is, “this” is a pointer

```
// example on this
#include <iostream>
using namespace std;

class Dummy {
public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
    if (&param == this) return true;
    else return false;
}

int main () {
    Dummy a;
    Dummy* b = &a;
    if ( b->isitme(a) )
        cout << "yes, &a is b\n";
    return 0;
}
```

```
yes, &a is b
```

# Classes

- Static members

- Just like Java, a static member means there is only one common variable for all objects of that class

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
public:
    static int n;
    Dummy () { n++; };
};

int Dummy::n=0;

int main () {
    Dummy a;
    Dummy b[5];
    cout << a.n << '\n';
    Dummy * c = new Dummy;
    cout << Dummy::n << '\n';
    delete c;
    return 0;
}
```

6  
7

# Classes

- Constant member functions (const)
  - When an object is instantiated by const, the access to its data members outside the class is read only
  - A constructor is still called to initialize variables – but can only be called once

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
    const MyClass foo(10);
    // foo.x = 20;           // not valid: x cannot be modified
    cout << foo.x << '\n';  // ok: data member x can be read
    return 0;
}
```

10

# Classes

- **Const**
  - Member functions specified as const cannot modify non-static data members
  - They cannot call other non-const member functions
- **Member functions can be overloaded based on const or not**
  - You can have two functions of the same name, with one const, and one not (example next slide)

# Classes

- Const overloading

```
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
    MyClass foo (10);
    const MyClass bar (20);
    foo.get() = 15;           // ok: get() returns int&
    // bar.get() = 25;        // not valid: get() returns const int&
    cout << foo.get() << '\n';
    cout << bar.get() << '\n';

    return 0;
}
```

```
15
20
```



# Classes

- Template Classes

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first, T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

```
mypair<int> myobject (115, 36);
```

```
mypair<double> myfloats (3.0, 2.18);
```

# Classes

- To use template classes

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () {
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
```

100

# Classes

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
        if ((element>='a') && (element<='z'))
            element+= 'A' - 'a';
        return element;
    }
};

int main () {
    mycontainer<int> myint (7);
    mycontainer<char> mychar ('j');
    cout << myint.increase() << endl;
    cout << mychar.uppercase() << endl;
    return 0;
}
```

8  
J

# Classes

- Special Member Functions

Member function	typical form for class C:
Default constructor	<code>C::C();</code>
Destructor	<code>C::~~C();</code>
Copy constructor	<code>C::C (const C&amp;);</code>
Copy assignment	<code>C&amp; operator= (const C&amp;);</code>
Move constructor	<code>C::C (C&amp;&amp;);</code>
Move assignment	<code>C&amp; operator= (C&amp;&amp;);</code>

# Classes

---

- **Default Constructor**

- The constructor called when objects are declared but not initialized with any parameters
- This is supplied automatically
- But just like Java, if a constructor of any kind is defined, the default constructor is no longer valid

# Classes

- Default Constructor

```
class Example {  
    public:  
        int total;  
        void accumulate (int x) { total += x; }  
};
```

```
Example ex;
```

```
class Example2 {  
    public:  
        int total;  
        Example2 (int initial_value) : total(initial_value) { };  
        void accumulate (int x) { total += x; };  
};
```

```
Example2 ex (100);    // ok: calls constructor
```

```
Example2 ex;          // not valid: no default constructor
```

# Classes

- Destructors

- The opposite of a constructor
- They clean up any memory used by the object before deleting it
- Same name as class/constructor, but starts with a “~”

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
    Example4 foo;
    Example4 bar ("Example");

    cout << "bar's content: " << bar.content() << '\n';
    return 0;
}
```

```
bar's content: Example
```

# Classes

- Copy Constructor

- Default copy constructor does a “shallow copy”
- Deep copy is done when pointer contents, not just addresses, are also copied

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
    Example5 foo ("Example");
    Example5 bar = foo;

    cout << "bar's content: " << bar.content() << '\n';
    return 0;
}
```

bar's content: Example



# Classes

- Copy Assignment

- Gives you the ability to copy objects not just on construction, but at any time
- Really overloading the “=” operator

```
Example5& operator= (const Example5& x) {  
    delete ptr;                // delete currently pointed string  
    ptr = new string (x.content()); // allocate space for new string, and copy  
    return *this;  
}
```

# Summary

- C++ Classes
- Special Members
- Friendship

