C++ Classes

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/classes/

Structs vs. Classes

In C, structs are used to create your own complex/custom data types.

structs can also be used in C++, however most developers use classes instead.

A class is a data type that combines state (variables with different data types) and behavior (functions, or methods that operate on the state).

Classes and Objects

Classes describe how to create Objects.

Think of a class like a blueprint, and the object as what the blueprint makes.

Declaring Classes

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

class_name is the name of the class.

access_specifiers can be: private, protected or public.

members are names of the variables belonging to the class.

object_names are optional, and allow you to declare variables of that type.

Example Class Declaration

```
class TreeNode {
  private:
    TreeNode *left;
    TreeNode *right;
  public:
    int value;
};
```

This is how you would declare a node for a tree class. Note that you do not need to use a typedef to use the class type within itself.

Members declared `private` can only be accessed other instances of the same class, the class itself or friends of the class.

Members declared `protected` can be accessed by the same class, its friends and all derived classes (more on that when we talk about inheritance).

Members declared `public` can be accessed by any function or class.

Members coming before a access modifier are by default private.

Example Class Declaration

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
    int getValue() {
       return value;
    }
};
```

Classes may also contain methods (ie., functions that belong to a class).

Note that by making the value member private, and providing a getValue function, we can provide access to value without allowing other classes to modify it.

Design decisions like this allow for extensible and less error prone code when working in teams.

Example Class Declaration

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
    int getValue();
};

int TreeNode::getValue() {
  return value;
}
```

The actual methods don’t necessarily need to be written within the class. This can be helpful when the class is described in a header file (the .hxx or .hpp file) and the actual source file defines the messages (the .cxx or .cpp file).

You can define a classes methods outside the class using class_name::method_name.

Example Class Declaration

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
    int getValue();
    void setValue(int newValue);
};

int TreeNode::getValue() {
  return value;
}

void TreeNode::setValue(int newValue) {
  value = newValue;
}
```

Of course, it is easy to break this encapsulation if you're not careful. See the setValue method.

Constructors and Destructors

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(int);   //constructor
   ~TreeNode();     //destructor
};

TreeNode::TreeNode(int v) {
  value = v;
  left = NULL;
  right = NULL;
}

TreeNode::~TreeNode() {
  left = NULL;
  right = NULL;
}
```

In your previous lab, you had to make functions to create and destroy your LinkedLists and their Nodes.

C++ classes have syntax for constructing and destructing objects.

Constructors and Destructors

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(int);    //constructor
   ~TreeNode();      //destructor
};

TreeNode::TreeNode(int v) {
  value = v;
  left = NULL;
  right = NULL;
}

TreeNode::~TreeNode() {
  left = NULL;
  right = NULL;
}
```

```
int main (int argc, char** argv) {
   //Just like structs, you can declare them as
   //regular variables, or pointers.
  TreeNode node(5);
  TreeNode *node_pointer = new TreeNode(5);

  //This will delete the node pointer
  delete node_pointer;

  //Non-pointers are automatically destructed
at the end of their scope, so the destructor
for node will be called here.
}
```

Constructors are called when the Object is initialized — either as a pointer with the new keyword or when the non-pointer variable is named.

Non-pointers are automatically destructed at the end of their scope. Pointers need to have delete called on them (or else there will be memory leaks).

Constructors and Destructors

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(int);  //constructor
   ~TreeNode();    //destructor
};

TreeNode::TreeNode(int v) {
  value = v;
  left = NULL;
  right = NULL;
}

TreeNode::~TreeNode() {
  cout << “Destroying TreeNode with
value “ << value << endl;
  left = NULL;
  right = NULL;
}
```

```
int main (int argc, char** argv) {
   //Just like structs, you can declare them as
   //regular variables, or pointers.
  TreeNode node(5);
  TreeNode *node_pointer = new TreeNode(5);

  {
     TreeNode node(6);
  }
  //This will delete the node pointer
  delete node_pointer;

  //Non-pointers are automatically destructed at the
end of their scope, so the destructor for node will
be called here.
}
```

Non-pointers are automatically destructed at the end of their scope. Pointers need to have delete called on them (or else there will be memory leaks).

Lets test this.

Overloading Constructors

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
      //constructors
      TreeNode();
      TreeNode(int);
      TreeNode(int, TreeNode*, TreeNode*);

      ~TreeNode();    //destructor
};

TreeNode::TreeNode() {
      value = 0;
      left = NULL;
      right = NULL;
}

TreeNode::TreeNode(int v) {
      value = v;
      left = NULL;
      right = NULL;
}

TreeNode::TreeNode(int v, TreeNode *l, TreeNode *r) {
      value = v;
      left = l;
      right = r;
}
```

It is possible to overload constructors (as well as methods) in objects. They can have the same name, but different argument types and the compiler is (usually) smart enough to determine which is which.

A constructor without any arguments is the default constructor; and would be called in the following situation:

TreeNode my_node;

Improving Constructor Performance

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(int);   //
constructor
   ~TreeNode();     //destructor
};

TreeNode::TreeNode(int v) {
  value = v;
  left = NULL;
  right = NULL;
}
```

This code is not as efficient as possible. When the TreeNode is created, the "value" variable will be created with it's default value (0).

Then it will be assigned the v value.

Improving Constructor Performance

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
    //constructor
   TreeNode(int v);
    //destructor
  ~TreeNode();
};
```

This code is not as efficient as possible. When the TreeNode is created, the "value" variable will be created with it's default value (0).

Then it will be assigned the v value.

This inefficiency can be fixed by extending the constructor definition.

```
TreeNode::TreeNode(int v) : value(v),
                     left(NULL), right(NULL) {
}
```

Copy Constructors, Passing Classes

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(TreeNode
&other); // copy constructor
   ~TreeNode();    //destructor
};

TreeNode::TreeNode(TreeNode
&other) {
  value = other.value;
  left = other.left;
  right = other.right;
}
```

Remember pass-by-value and pass-by-reference?

You can control how a class is copied using copy constructors (see left).

Overloading Operators

Overloadable operators

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

It is possible to specify operator functions, which overload operators for certain operands. The format is:

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

Expression	Operator	Member function	Global 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->x	->	A::operator->()	-

Overloading Operators

For classes, the operator function belongs to the class.

Using the operator + is shorthand for .operator+.

```
class ComplexNumber {
  private:
    double real;
    double imaginary;
  public:
    ComplexNumber(double r, double i) :
        real(r),
        imaginary(i) {};
    ComplexNumber operator+ (const ComplexNumber &);
};

ComplexNumber ComplexNumber::operator+(const ComplexNumber &other) {
  return ComplexNumber(this->real + other.real, this->imaginary + other.imaginary);
}

int main() {
  ComplexNumber c1(10, -3), c2(-2.3, 5);
  ComplexNumber c3 = c1 + c2;
  CompelxNumber c4 = c1.operator+(c2);  //This is the same as the previous line
}
```

Overloading Operators

It can also be very useful to overload the << operator to the ostream class (cout is an instance of ostream).

This will let you write your objects to cout easily. Note that this is a friend function (more on those later).

```
class ComplexNumber {
  private:
    double real;
    double imaginary;
  public:
    ComplexNumber(double r, double i) :
        real(r),
        imaginary(i) {};
    ComplexNumber operator+ (const ComplexNumber &);
    friend ostream& operator<<(ostream&, const ComplexNumber&);
};

ComplexNumber ComplexNumber::operator+(const ComplexNumber &other) {
  return ComplexNumber(this->real + other.real, this->imaginary + other.imaginary);
}

ostream& operator<<(ostream &os, const ComplexNumber &cn) {
  os << “ComplexNumber[r: “ << cn.real << “, i: “ << cn.imaginary << “]”;
  return os;
}

int main() {
  ComplexNumber c1(10, -3), c2(-2.3, 5);
  cout << c1 << “ + “ << c2 << “ = “ << (c1 + c2) << endl;
}
```

The ‘this’ Keyword

If you use the ‘this’ keyword, it returns a pointer to the object that the method is being invoked on. See the operator+ method.

```
class ComplexNumber {
  private:
    double real;
    double imaginary;
  public:
    ComplexNumber(double r, double i) :
        real(r),
        imaginary(i) {};
    ComplexNumber operator+ (const ComplexNumber &);
    friend ostream& operator<<(ostream&, const ComplexNumber&);
};

ComplexNumber ComplexNumber::operator+(const ComplexNumber &other) {
  return ComplexNumber(this->real + other.real, this->imaginary + other.imaginary);
}

ostream& operator<<(ostream &os, const ComplexNumber &cn) {
  os << “ComplexNumber[r: “ << cn.real << “, i: “ << cn.imaginary << “]”;
}

int main() {
  ComplexNumber c1(10, -3), c2(-2.3, 5);
  cout << c1 << “ + “ << c2 << “ = “ << (c1 + c2) << endl;
}
```

The ‘this’ Keyword

The ‘this’ is commonly used in the operator=, as an example;

```
class ComplexNumber {
  private:
    double real;
    double imaginary;
  public:
    ComplexNumber(double r, double i) :
        real(r),
        imaginary(i) {};
    ComplexNumber& operator= (const ComplexNumber &);
};

ComplexNumber& ComplexNumber::operator=(const ComplexNumber &other) {
  real = other.real;
  imaginary = other.imaginary;
  return *this;
}
```

Static Members

```
class Tree {
  …
  public:
    static int number_of_trees;
    Tree();
    ~Tree();
};

Tree::Tree() {
  number_of_trees++;
  …
}

Tree::~Tree() {
  number_of_trees–;
  …
}

int main() {
  Tree *tree1 = new Tree();
  Tree *tree2 = new Tree();
  Tree tree3();
  Tree tree4();

  //static members can be accessed using ::
  cout << "The number of trees is: " <<
Tree::number_of_trees;
}
```

Classes can contain *static* members. Both functions and variables can be static.

You can think of static members as belonging to the class instead of the Object.

In this way, there is only one one of the static member, which all the instances share.

Static members are very similar to (if not essentially) global variables.

Friend Functions

```
class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
    TreeNode(int); // copy constructor
    ~TreeNode();     //destructor
    friend TreeNode* duplicate(TreeNode*);
};

TreeNode::TreeNode(int v) {
  value = v;
  left = null;
  right = null;
}

TreeNode* duplicate(TreeNode *current) {
  if (current == null) return null;
  TreeNode *temp = new TreeNode(current.value);
  temp->left = duplicate(current->left);
  temp->right = duplicate(current->right);
}
```

Sometimes you may want a function to have access to private the members of a class.

You can declare functions as “friend” which will let them allow the private members.

Friend Classes

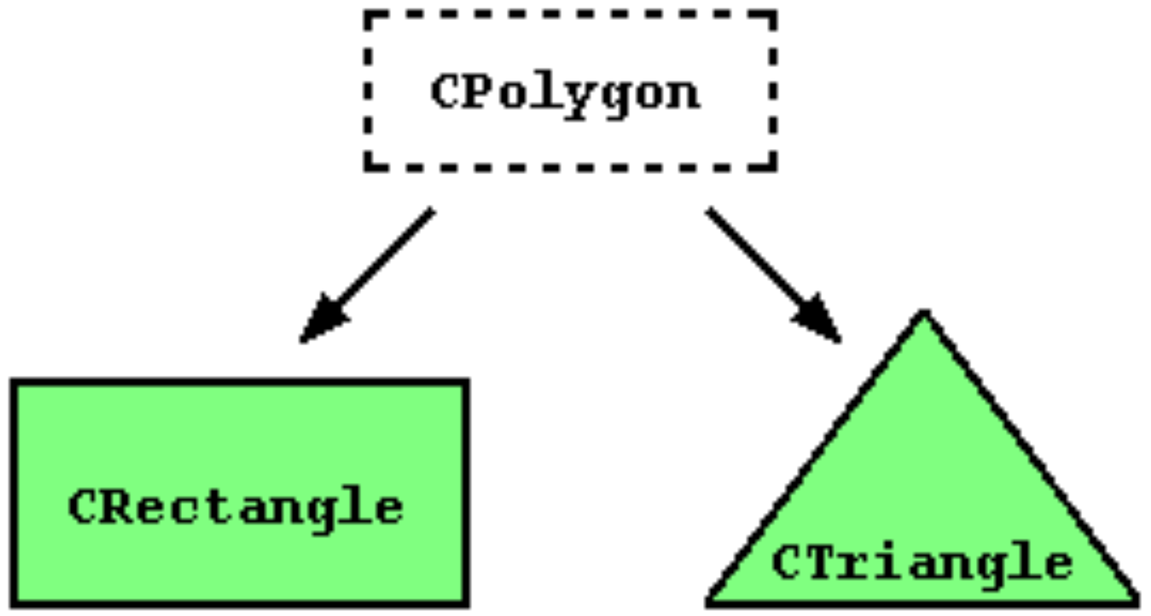
```
class Tree;

class TreeNode {
  private:
    int value;
    TreeNode *left;
    TreeNode *right;
  public:
   TreeNode(int); // copy constructor
   ~TreeNode();    //destructor
    friend class Tree;
};

class Tree {
  private:
    TreeNode *root;
  public:
    Tree();
}
```

You can also declare another class as a friend. This will allow that class to access private members of the class that friended it.

Inheritance



Often, you may want to have multiple classes use the same fields and methods. Cut-and-pasting fields and methods between classes not only is a waste of time, but it also makes code harder to debug and maintain.

Inheritance allows classes to ‘inherit’ fields and methods from their parent classes. This enables easy code reuse.

Inheritance allows specialization and/or extension of other classes.

Inheritance

The format for inheriting (or *deriving*) a class is:

```
class derived_class_name : access_specifier base_class_name {
    /*…*/
};
```

The access specifier can be private, protected or public (as before).

Inheriting a class gives the child class access to all protected and public (but not private) fields and methods of the parent class.

The access specifier determines can be used to restrict the parent members have when accessed from the child class.

For example, if the access specifier is public, the parents protected members will still be protected. However, if the access specifier is private, than the parents protected members will be private (when accessed from the child).

Inheritance

Inheriting a class gives the child class access to all protected and public (but not private) fields and methods of the parent class.

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case Polygon) will have from the derived class (in this case Rectangle). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

With protected, all public members of the base class are inherited as protected in the derived class. Conversely, if the most restricting access level is specified (private), all the base class members are inherited as private.
For example, if the access specifier is public, the parents protected members will still be protected. However, if the access specifier is private, than the parents protected members will be private (when accessed from the child):

Access	**public**	**protected**	**private**
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Inheritance - Example

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle : public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle : public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Inheritance

Apart from the previous rules, derived classes (or child classes) inherit every member of their parent class except:

the parent’s constructor and destructors
the parent’s operator=() members
the parent’s friends

Note that a parent’s default constructor (the constructor with no arguments) is called when the child constructor is called automatically. A parent’s default destructor is also automatically called when the child destructor is called.

You can specify another constructor of the parent to be called instead by adding it after a ‘:’ after the constructor declaration, e.g.:

```
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
```

Inheritance

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);

  return 0;
}
```

Multiple Inheritance

```cpp
// multiple inheritance
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class COutput {
  public:
    void output (int i);
};

void COutput::output (int i) {
  cout << i << endl;
}

class CRectangle : public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height); }
};

class CTriangle : public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values(4,5);
  trgl.set_values(4,5);
  rect.output(rect.area());
  trgl.output(trgl.area());
  return 0;
}
```

It is also possible to inherit from multiple classes in C++ (unlike other languages, for example, Java).

The child class inherit members in the same way from all parents.

This is done by adding multiple class names (separated by commas) after the : in the class declaration.

Polymorphism

```
// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
};

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  CRectangle rect;
  CTriangle trgl;

  CPolygon *ppoly1 = &rect;
  CPolygon *ppoly2 = &trgl;

  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;

  return 0;
}
```

You can refer to child classes by the type of their parent class. For example, the code to the left assigns references to rectangle and triangle to pointers of type polygon.

This is possible as the child classes inherit all the members of the parents classes.

Note, that ppoly1 and ppoly2 are only known by the compiler as type CPolygon, so you cannot call the area() method on them.

Polymorphism

```
// pointers to base class
#include <iostream>
using namespace std;

class TreeNode {
  public:
    int value;
    TreeNode *left, *right;
};

class RBTreeNode : public TreeNode {
  public:
    int area ()
      { return (width * height); }
};

int height(TreeNode *n) {
  if (n == null) return 0;
  else return max(height(n->left), height(n->right));
}

int main () {
  Tree tree = …;
  RBTree rb_tree = …;

  cout << “The height of tree is: “ << height(tree->root) << endl;
  cout << “The height of rb_tree is: “ << height(rb_tree->root) << endl;
  return 0;
}
```

This becomes even more useful when passing classes to methods. For example, you can write a height function which takes either a TreeNode or a RBTreeNode.

Virtual Functions

```cpp
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return (0); }
};

class CRectangle : public CPolygon {
  public:
    int area ()
      { return (width * height); }
};

class CTriangle : public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

You can specify that a classes member function can be overridden by subclasses with the keyword ‘virtual’.

In this case, if the function is invoked on a reference with the type of the parent class, the child classes version will be invoked.

Abstract Classes / Pure Virtual Functions

```
// abstract class CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

It is possible to force all child classes to implement a particular function, but retain the declaration of the function in the parent class.

Classes do this with pure virtual functions, and are called abstract classes (they are abstract because they cannot be instantiated because all their behavior is not defined).

A function is virtual by adding '=0' after it's declaration.

Abstract Classes / Pure Virtual Functions

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

This can be extremely useful — it allows you to specify that other classes will implement some functionality that a particular class does not have enough information.

This allows the development of more generic code, which makes code re-use and debugging much easier.