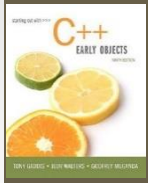
COP2334

Introduction to Object Oriented Programming with C++

D. Singletary

Module 11

Ch. 15 Polymorphism and Virtual Functions

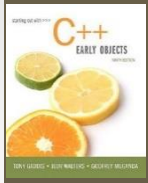


FLORIDA STATE COLLEGE at Jacksonville

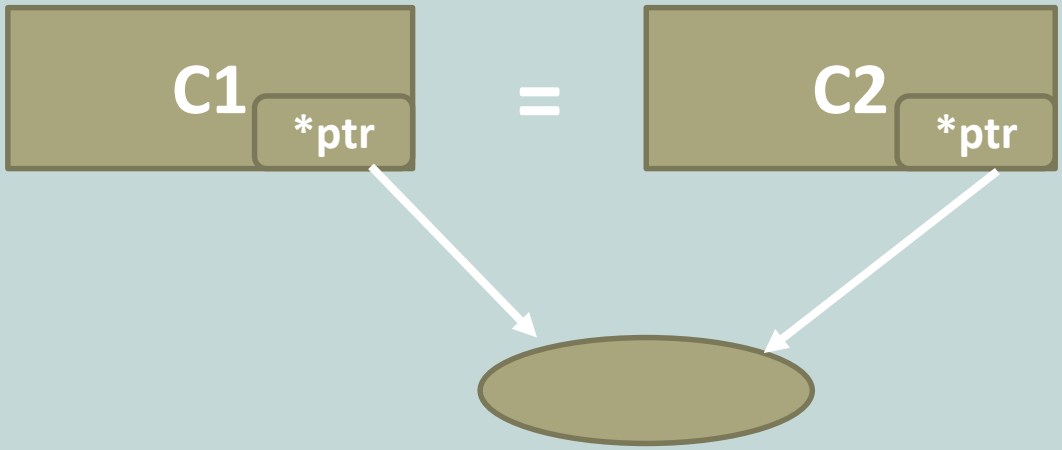
Memberwise Assignment

- The = operator can be used to copy one object to another

```
// declare and print a Triceratops
Triceratops *trike = new Triceratops("T. Horridus", 5.0);
trike->print();

// create a new object (not just a pointer reference)
Triceratops trike2 = *trike;
cout << "trike2:" << endl;
trike2.print();
```



- Assigning an object to another object calls an implicit default copy constructor (not the standard default constructor) which does a member-wise assignment (copies all members from one object to the other).
  - If a class contains members that are pointers, you must write a custom copy constructor; otherwise the copy operation only copies the address of the object pointed to

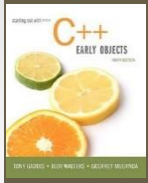


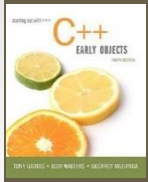
Defining a Copy Constructor

- A copy constructor takes a single parameter that is a reference to the same class it is a constructor for

Triceratops::Triceratops(const Triceratops &obj)

- The copy constructor can allocate memory for the pointer member and perform the copy
- The parameter is const since the source object is not changed
- We can add a call to a base class constructor if this is a derived class (see next slide)





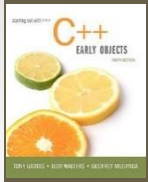
- To demonstrate a copy constructor, add a pointer member to the Triceratops class

```
class Triceratops : public Dinosaur {
   private:
      double hornSize; // inches
      int *numSiblings;
```

- Here's the copy constructor (including a base class constructor call):

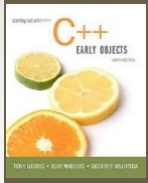
```
Triceratops(const Triceratops &obj) : Dinosaur(obj.getName(), false)
{
        this->hornSize = obj.hornSize;
        this->numSiblings = new int;
       *this->numSiblings = *obj.numSiblings;
}
```

```
// 3-arg constructor (name, horn size, number of siblings)
Triceratops *trike = new Triceratops("T. Horridus", 5.0, 4);
cout << "trike:" << endl;
trike->print();
// copy constructor
Triceratops trike2 = *trike;
cout << "trike2:" << endl;
trike2.print();
// copy constructor
Triceratops *trike3 = new Triceratops(*trike);
cout << "trike3:" << endl;
trike3->print();
// default (0-arg) constructor
Triceratops *trike4 = new Triceratops();
cout << "trike4:" << endl;
trike4->print();
```



```
// 3-arg constructor (name, horn size, number of siblings)
Triceratops *trike = new Triceratops("T. Horridus", 5.0, 4);
cout << "trike:" << endl;
trike->print();
// copy constructor
Triceratops trike2 = *trike;
cout << "trike2:" << endl;
trike2.print();
// copy constructor
Triceratops *trike3 = new Triceratops(*trike);
cout << "trike3:" << endl;
trike3->print();
// default (0-arg) constructor
Triceratops *trike4 = new Triceratops();
cout << "trike4:" << endl;
trike4->print();
```

```
trike:
Triceratops: T. Horridus is not a carnivore, siblings = 0, horn size = 5.00
trike2:
Triceratops: T. Horridus is not a carnivore, siblings = 0, horn size = 5.00
trike3:
Triceratops: T. Horridus is not a carnivore, siblings = 0, horn size = 5.00
trike4:
Triceratops: noname is not a carnivore, siblings = 0, horn size = 0.00
```



Invocation of Copy Constructors

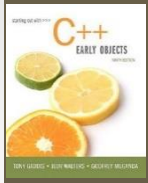
- Copy constructors are automatically called by the system when an object is created by initializing it (copying it) with another object of the same class
- Copy constructors are also called when a function call receives a pass-by-value parameter of the class type

```
void function(Rectangle rect)
```

- Copy constructors are called when a function returns an object of a class by value

```
Rectangle makeRectangle() {
    Rectangle r(12,3);
    return r;
}
```

- Simple "=" assignments do not invoke the copy constructor (see next slide)



Operator Overloading

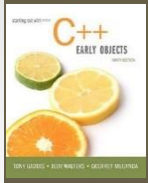
- C++ allows you to redefine standard operator behavior when used with class objects. Consider

```
Triceratops t1("T. Horridus", 5.0, 4);
Triceratops t2("T. Prorsus", 8.0, 2);
```

- We can do the following:

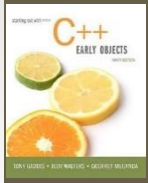
```
t1 = t2;
```

- A member-wise assignment occurs here (but ***not*** through the copy constructor; this is an assignment, not initialization)
- This results in a similar problem as the copy construction; the *address* of the "numSiblings" member from t2 is copied into t1, we want an entirely new address with the copied value (consider what would happen if t2 went out of scope after copying into t1 with the original address)



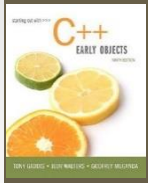
- We can define an operator overload for the "=" operator:

```
Triceratops::Triceratops& operator=(const Triceratops &obj)
{
    if (this != &obj)  // only copy if it's a different object
    {
        this->setName(obj.getName());    // added mutator
        this->setCarnivore(false);
        this->hornSize = obj.hornSize;

        if (this->numSiblings != nullptr)
            delete this->numSiblings;
        this->numSiblings = new int;
        *this->numSiblings = *obj.numSiblings;
    }
    return *this;
}
```



```
Triceratops t1("T. Horridus", 5.0, 4);
Triceratops t2("T. Prorsus", 8.0, 2);

cout << "t1:" << endl;
t1.print();
cout << "t2:" << endl;
t2.print();

t1 = t2;  // overloaded "=" called here
cout << "t1':" << endl;
t1.print();
```

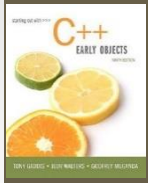
```
t1:
Triceratops: T. Horridus is not a carnivore, siblings = 4, horn size = 5.00
t2:
Triceratops: T. Prorsus is not a carnivore, siblings = 2, horn size = 8.00
t1':
Triceratops: T. Prorsus is not a carnivore, siblings = 2, horn size = 8.00
```



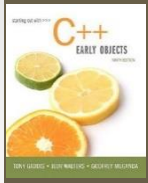
Overloading Other Operators

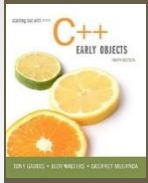
Consider a Date class:

```
Date d1 = new Date("10/31/2016");

d += 5;      // overload += to add 5 days


Date d2 = new Date("11/30/2016");

if (d2 > d1) ...   // overload > to compare
```



```
bool Date::Date::operator< (Date a, Date b)
{
        bool result = false;

        if (a.year < b.year)
           result = true;
        else if (a.year == b.year) {
                if (a.month < b.month)
                        result = true;
                else if (a.month == b.month)
                {
                        if (a.day < b.day)
                                result = true;
                }
        }
        return result;
}
```





Operators which can be overloaded:

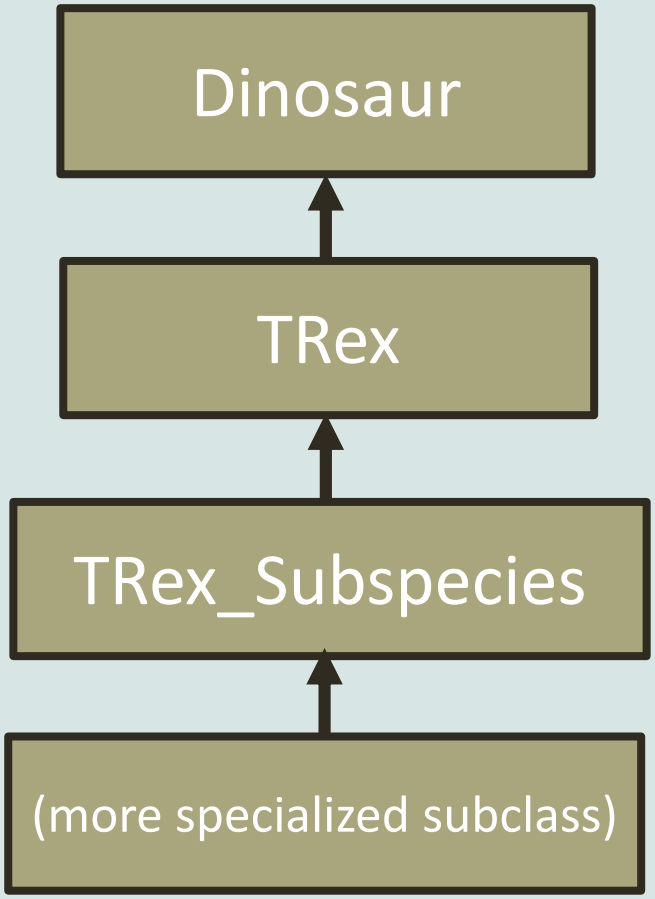
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operators which cannot be overloaded:

::	.*	.	?:

Polymorphism and Virtual Functions

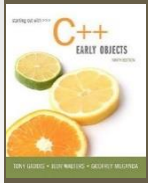
- The is-a relationship represented by inheritance results in an inheritance hierarchy:
  - Arrows are used between classes in a UML diagram to indicate inheritance

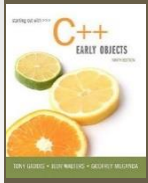


Polymorphism

- Objects of a derived class can be used wherever objects of a base class are expected. This is a form of polymorphism ("occurring in different forms").
  - A derived class pointer can always be assigned to a base class pointer
  - A type cast is required to assign a base class pointer to a derived class pointer

```
TRex *tr1 = new TRex(...
Dinosaur *d = tr1;
TRex *tx = d; // error!
```





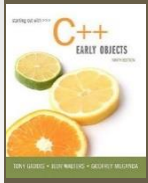

```
TRex *tr1 = new TRex(...
Dinosaur *d = tr1;
TRex *tx = d; // error!
```

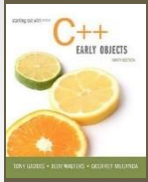
error: invalid conversion from 'Dinosaur*' to 'TRex*'

- A TRex is always a dinosaur, so we can safely assign the TRex pointer "tr1" to the dinosaur pointer "d"
- A dinosaur is not always a TRex, it could be another type of dinosaur, so we cannot assign the dinosaur pointer "d" to the TRex pointer "tx" without a cast (program will not build)

```
TRex *tr1 = new TRex(...
Dinosaur *d = tr1;
TRex *tx = static_cast<TRex*>(d); // this is OK
```

- By casting the dinosaur pointer to a TRex pointer, we are telling the compiler that we know this particular value of d is actually a TRex, so it is acceptable to do this assignment





- Assume getToothSize() is an accessor belonging to the derived Trex class (and not the Dinosaur class)

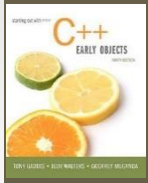
```
TRex *tr1 = new TRex(...
Dinosaur *d = tr1;
cout << "dinosaur's tooth size:" << d->getToothSize();
```

- Why doesn't this work?

```
TRex *tr1 = new TRex(...
Dinosaur *d = tr1;
cout << "dinosaur's tooth size:" << d->getToothSize();
```

- Although we have assigned a TRex to the dinosaur pointer, the Dinosaur class does not know about the getToothSize() accessor, which occurs only in the derived TRex class.
- We can cast "d" to get the correct result:

```
cout << "dinosaur's tooth size = " <<
        static_cast<TRex*>(d)->getToothSize() << endl;
```



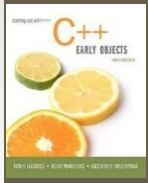
- We can pass a pointer to a subclass to a function which accepts a base class pointer, but we can only access information available to the base class if we don't do a cast:

```
int main() {
        Triceratops *trike1 = new Triceratops("T. horridus",
                                        /* horn size */ 16.0);
        TRex *tr2 = new TRex("Tyrannosaurus Asia",
                                         /* tooth size */ 3.0);
        printDinosaur(trike1);
        printDinosaur(tr2);
        ...
}

void printDinosaur(Dinosaur *dParam)
{
```

```
void printDinosaur(Dinosaur *dParam)
{
    string isCarn = dParam->isCarnivore() ? " is " : " is not ";
    cout << "Dinosaur: " << dParam->getName() <<
        isCarn << "a carnivore." << endl;
}
```

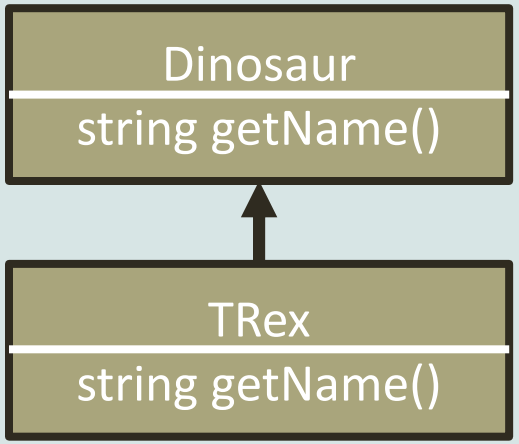
```
Dinosaur: T. horridus is not a carnivore.
Dinosaur: Tyrannosaurus Asia is a carnivore.
```

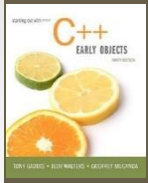
- If dParam->getName() is declared in the base class we can call it in printDinosaur()



Virtual Member Functions

- Virtual functions support polymorphism by allowing the most specific version of a member function in an inheritance hierarchy to be selected for execution.
- Define getName() in both the base class and subclass:





- Let's create more specific getName() output from the subclass's inherited getName() accessor
  - In other words, instead of just

    Tyrannosaurus Asia

  - we want to see a prefix of the specific dinosaur's type which is known only to the subclass:

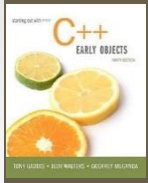
    TRex: Tyrannosaurus Asia

```
string Dinosaur::getName() {
        return this->name;
}
```

```
const string TRex::NAME_PREFIX = "TRex: ";
string TRex::getName() {
        // Dinosaur::getName() calls the base class accessor
        return NAME_PREFIX + Dinosaur::getName();
}
```

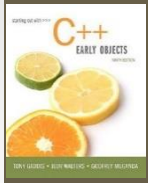
- If we use a Dinosaur object pointer to print the name, we are calling the base class's getName() accessor:

```
TRex *tr1 = new TRex("Tyrannosaurus Laramidia", /* tooth size */ 2.0);
Triceratops *trike1 = new Triceratops("T. horridus", /* horn size */ 16.0);
printDinosaur(trike1);
printDinosaur(tr1);
...
void printDinosaur(Dinosaur *dParam)
{
      string isCarn = dParam->isCarnivore() ? " is " : " is not ";
      cout << "Dinosaur: " << dParam->getName() <<
            isCarn << "a carnivore." << endl;
}
```

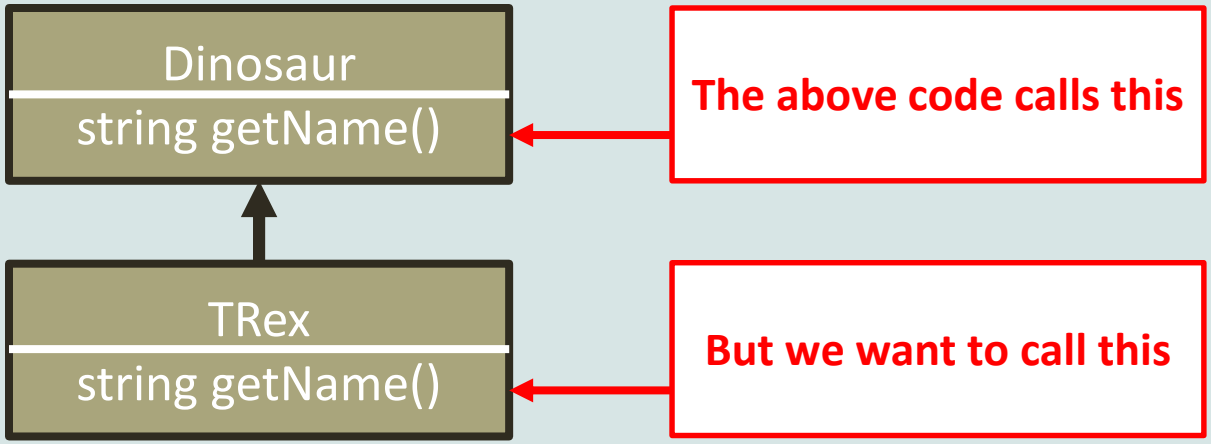


Dynamic and Static Binding

- Static binding occurs when the compiler binds a function name at compile time to the code that should be executed. This occurs if a function is not virtual (polymorphic).
- Dynamic binding occurs at run time for virtual functions which may be associated with different types. The compiler cannot choose beforehand which code should be executed.



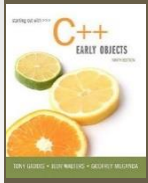
```
void printDinosaur(Dinosaur *dParam)
{
    string isCarn = dParam->isCarnivore() ? " is " : " is not ";
    cout << "Dinosaur: " << dParam->getName() <<
        isCarn << "a carnivore." << endl;
}
```

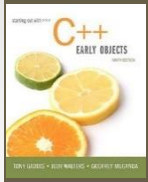


- How do we access the more specific TRex getName() using a Dinosaur pointer?
  - Declare the base class's accessor as virtual:

```
virtual string Dinosaur::getName() {
        return this->name;
}
```

- Now dParam->getName() will use the most specific getName() based on the class type



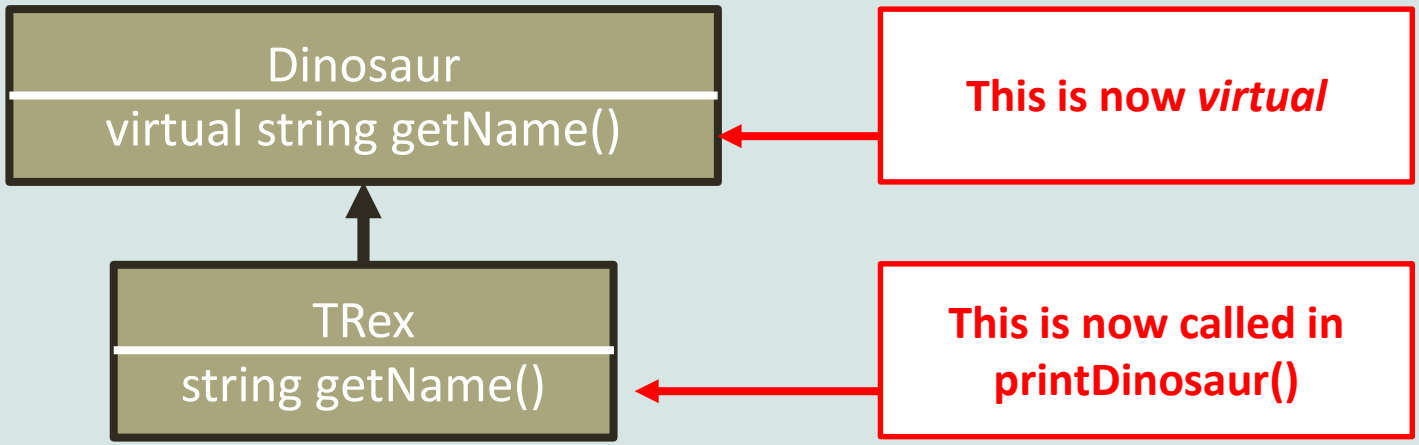


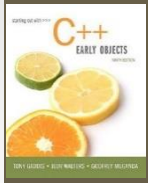
- If we do not define the function inline, only the prototype should be declared virtual:

```
virtual string getName();
...
string Dinosaur::getName()  {  // no virtual needed here
        return this->name;
}
```

```
void printDinosaur(Dinosaur *dParam)
{
    string isCarn = dParam->isCarnivore() ? " is " : " is not ";
    cout << "Dinosaur: " << dParam->getName() <<
        isCarn << "a carnivore." << endl;
}
```

```
Dinosaur: Triceratops: T. horridus is not a carnivore.
Dinosaur: TRex: Tyrannosaurus Laramidia is a carnivore.
```

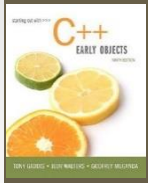




- Our Dinosaur class hierarchy is now polymorphic; we get different behavior for different (but related) types of objects
- The virtual characteristic is inherited throughout the hierarchy, which means that the getName() in the subclass (e.g. TRex) is also now virtual, as are all getName() functions in any classes derived from the subclass.

- By convention, we explicitly label all inherited virtual functions as virtual even though it is not required.

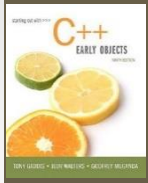
```
class Dinosaur
{
        …
        virtual string getName();

class TRex
{
        …
        virtual string getName();
```



Pure Virtual Functions

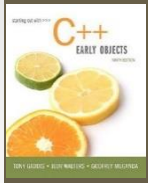
- A pure virtual function has no body; we declare it in a base class in order to force subclasses to implement it (pure virtual functions enforce behavior)
- To declare a pure virtual function add "= 0" after its parameter list, in front of the semi-colon:

```
virtual string getName() = 0;
// this means we can no longer call
// Dinosaur::getName() from the derived
// classes anymore
```

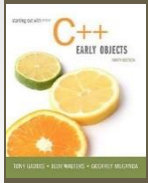


Abstract Base Classes

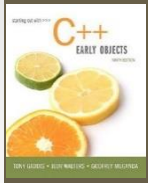
- Pure virtual functions are also known as abstract functions.
- A class containing a pure virtual function is known as an abstract class.
- An abstract class cannot be instantiated.
- Attempts to create an instance of an abstract class result in a build error:
  - "invalid new-expression of abstract class type“
  - We can still call the non-abstract methods from the derived classes, however



```
class Fruit {
    private:
        string type;
    protected: // restrict access to within the subclass
        void setType(string type) {
                this->type = type;
        }
    public:
        Fruit() {
                this->type = "unknown";
        }
        string getType() {
                return this->type;
        }
        virtual void setFruitType(string type) = 0;
};
```

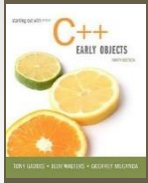


```
class Apple : public Fruit {
    private:
        string color;
    public:
        string getColor() {
            return this->color;
        }
        void setColor(string color) {
            this->color = color;
        }
        void setFruitType(string type)
        {
            Fruit::setType(type);
        }
};
```

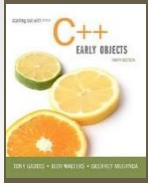


```
int main()
{
        Apple *a = new Apple();
        a->setFruitType("granny smith");
        a->setColor("green");
        cout << "apple color = " << a->getColor() <<
                        " and type is " << a->getType() << endl;
        return 0;
}
```

apple color = green and type is granny smith

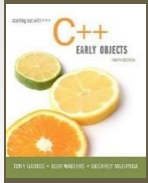


```
int main()
{
        Apple *a = new Apple();
        a->setType("granny smith"); // Error! (protected)
        a->setColor("green");
        cout << "apple color = " << a->getColor() <<
                        " and type is " << a->getType() << endl;
        return 0;
}
```



```
int main()
{
        Fruit *f = new Fruit(); // Error! Fruit is abstract
}
```

```
int main()
{
        Fruit *f = new Apple();
        Apple *a  = new Fruit(); // Error
        f->setType("granny smith"); // Error! (protected)
}
```



```
int main()
{
        Fruit *f = new Apple();
        f->setFruitType("granny smith");
        cout << "fruit type is " << f->getType() << endl;
}
```

```
fruit type is granny smith
```