

Inheritance

16.1 Inheritance and Subclasses

In C++ you can make one class an *extension* of another. This is called *inheritance*. The classes form an *is-a* hierarchy. The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendibility.

Inheritance allows one to derive classes from a base class without disturbing the implementation of the base class.

```
class Derived : public Base
{
    additional instance variables;
    new constructors;
    //inherited members and functions;
    overriding methods; // replacing those in Base
    additional methods;
}
```

To create a class `Rectangle` which extends a class `Shape`, one starts with

```
class Rectangle : public Shape {
```

The class `Rectangle` then automatically has all the methods of class `Shape`, and you can add some of your own.

The methods and the instance variables of the superclass can be accessed (if not declared as `private`)— they may be declared as `protected` to allow direct access only to extensions. The default constructor for `Rectangle` is automatically called at the start of the constructor for `Shape`, unless you specify otherwise.

The derived class (*subclass*) is a new class that has some *type compatibility* in that it can be substituted for the base class (*superclass*). A pointer has a *static type* determined by the compiler. An object has a *dynamic type* which is fixed at run-time creation. A pointer reference is *polymorphic* since it can reference objects of different dynamic type. A pointer may reference objects of its declared type or any subtype of its declared type; subtype objects may be used whenever supertype objects are expected. There are times an object may need to be *cast* back to its original type. Note that the actual dynamic type of an object is forever fixed.

Suppose class `Rectangle` extends class `Shape`. Then we could do the following assignments:

```

Shape *X;
Rectangle *Y;
Y = new Rectangle();
X = Y; // okay
X = new Rectangle(); // okay
Y = static_cast<Rectangle*>(X); // cast needed

```

16.2 Overriding Functions

An important facet of inheritance is that the derived class can replace a generic function with a tailored function. *Overriding* is providing a function with the same signature as the superclass but different body.

But then a key question is which version of the function gets used. There are two options: if we declare a function is *virtual*, then which version of the function is used is determined at run-time by the object's actual dynamic type. (In Java, all functions are implicitly virtual.) If we declare a function as *nonvirtual*, then which version is determined by the compiler based on the static type of the reference. Usually a *nonvirtual* function is not intended to be overridden. Note that a function declared as virtual in the base class is automatically virtual in the derived class.

You can access in the **Derived** class the **Base** version of an overridden function by using the scope resolution operator: `Base::fooBar()`.

16.3 Interfaces and Abstract Base Classes

In Java, an interface specifies the methods which a class should contain. A class can then implement an interface (actually it can implement more than one). In doing so, it must implement every method in the interface (it can have more). This is just a special case of inheritance: the base class specifies functions but they are not implemented.

C++ uses abstract base classes. An *abstract base class* is one where only some of the functions are implemented. A function is labeled as abstract by setting it equal to 0; this tells the compiler that there will be no implementation of this function. It is called a *pure* function. An object with one or more pure functions cannot be instantiated.

Example: the abstract base class (interface)

```

class Number {
public:
    virtual void increment()=0;
    virtual void add(Number &other)=0;
    ETC
};

```

the implementation:

```
class MyInteger : public Number {
private:
    int x;
public:
    virtual void increment(){x++;}
    ETC
}
```

the calling program (but then one can only execute `Number`'s methods on `ticket`).

```
Number *ticket = new MyInteger();
ticket->increment();
```

16.4 Sample Code: RedBlackTree

Notice that `RNode` extends `BSTNode` and that `RedBlackTree` extends `BinarySearchTree`. The actual code looks ugly in places...

```
// RNode.h
// provides node suitable for red-black tree
// wdg 2008
#ifndef RBNODE_H
#define RBNODE_H

#include <iostream>
using namespace std;
#include "BSTNode.h"

enum Color { RED, BLACK, NONE }; // creates a data type and three constants

class RNode : public BSTNode
{
public:
    Color color;

    RNode (ItemType e) : BSTNode(e,NULL) , color(NONE)
    {
    }
    void dump( )
```

```

        {
            cout << element << "(" << color << ")";
        }
};

```

```

#endif

```

```

// RedBlackTree.h
// does not allow duplicate items
// wdg 2008
#include "RBNode.h"
#include "BinarySearchTree.h"

class RedBlackTree : public BinarySearchTree
{
public:
    RedBlackTree();
    void insertItem(ItemType it);

private:
    void leftRotate(RBNode *x);
    void rightRotate(RBNode *x);
    RBNode *grand(RBNode *v);
    RBNode *uncle(RBNode *v);
    bool isLeftChild(BSTNode *v);
}; // end of class

```

```

/* extract from RedBlackTree.cpp */
....

// returns the grandparent of a node
RBNode *RedBlackTree::grand(RBNode *v)
{
    if( v==root || v->parent==root)
        return NULL;
    else
        return static_cast<RBNode*> (v->parent->parent);
}

...

```

```

void RedBlackTree::insertItem(ItemType it)
{
    RBNode *x = new RBNode(it);
    if( !BinarySearchTree::insertItem(x) )
        return;
    x->color = RED;
    while
        ...
}

```

16.5 Sample Code: BTree

A primitive implementation of a B-tree. We omit the implementations of `BTreeInternal` and `BTreeLeaf`.

```

// BTreeNode.h wdg 2008
#ifndef BTREENODE_H
#define BTREENODE_H
#include <iostream>
using namespace std;

class BTreeNode
{
public:
    static const int MAX=3;
    virtual BTreeNode *insert(int item, int &newKey) = 0;
    virtual void dump(int depth) = 0;
    static void indent(int depth)
    {
        for(int i=0; i<depth; i++)
            cout << "...";
    }
};

class BTreeLeaf : public BTreeNode
{
private:

```

```

    int value[MAX+1]; // need to go temporarily over
    int elements;

public:
    BTreeNode();
    BTreeNode *insert(int item, int &newKey);
    void dump(int depth);
};

class BTreeNode : public BTreeNode
{

private:
    BTreeNode * child[MAX+1];
    int marker[MAX+1];
    int currChildren;
    // note: marker[0] is not used
    // child[0] - marker[1] - child[1] - marker[2] - child[2] etc

    friend class BTree;

public:
    BTreeNode();
    BTreeNode *insert(int item, int &newKey);
    void dump(int depth);
};

class BTree
{
private:
    BTreeNode *root;

public:
    BTree( );
    void insert(int item);
    void dump();
};
#endif

```

```

#include <iostream>
using namespace std;
#include "BTreeNode.h"

BTree::BTree( ) : root(NULL)
{
}

void BTree::insert(int item)
{
    if( root==NULL )
        root = new BTreeLeaf();
    int newKey;
    BTreeNode *split = root->insert(item, newKey);
    if(split!=NULL) {
        cout << "Splitting root " << endl;
        BTreeInternal *newRoot = new BTreeInternal();
        newRoot->child[0] = root;
        newRoot->child[1] = split;
        newRoot->marker[1] = newKey;
        newRoot->currChildren = 2;
        root = newRoot;
    }
}

void BTree::dump()
{
    root->dump(0);
}

```