C++ Advanced Course

Zoltán Porkoláb
gsd@zolix.hu

2001-08-20

1 Preface

This is a tutorial for the Advanced Course on the C++ Language. It's main purpose is to give a background material for the 24 hours Advanced C++ course. It it supposed that You are already familiar with the basic C++ language elements like:

- Overview of the basic elements
- Differences between C and C++
- Lexical rules
- Scope rules
- Life rules
- References
- Namespaces
- Function declarations
- Default arguments
- Parameter passing by value and by reference
- Overloading
- Template functions
- Classes
- Visibility (public, protected, private)
- This
- Const memberfunction, const and static members
- Operator overloading
- Copy semantics (copy constructor, assignment)
- Class templates

- Inheritance
- Polymorphism (virtual functions)
- Exception handling

This course try to answer the question *when* and *how* to use the language features above, rather than learn newer and newer language elements. (However we learn a few one, like *pointer to member*, etc...) We have an outlook on design questions too. We will analyze already written (and working example programs: try to understand them and find it's problematic parts. Then we improve them.

We also take a tour on the standard library. We get to know the basic concept on STL (*Standard Template Library*) and the I/O mechanism. We also practice the usage of `std::string`.

2 C++ Design Issues

In this section we discuss some of the fundamental design issues of the C++ language. We overlook the role of a class, and the steps how we can specify the classes in a project. We encounter the different connection types between classes. These relationships may describe various connections between the objects of those classes: connections in objects life, usage of interface, etc.

2.1 Never looking for a single class

Consider designing a single class. Tipically, this is **not a good idea**. Concepts do not exist in isolation; rather, a concept is defined in the context of other concepts. Similarly, a class does not exist in isoolation but is defined together with logically related classes. Typically, on e works on a set of related classes. Such a set is often called a class library or a component. Sometimes all the classes in a component constitue a single class hierarchy, sometimes they are members of a single namespace, and sometimes they are a more ad-hoc collection of declarations.

The set of classes in a component is united by some logical criteria, often by a common style and often by a reliance on common services. A component is thus the unit of design, documentation, ownership, and often reuse. This does not mean that if you use one class from a component, you must understand and use all the classes from the component or maybe get the code for every class in the component loaded into your program. On the contrary, we typically strive to ensure that a class from a componentcan be used with only minimal overhead in machine resources and human effort.

2.2 How to design components

As usual, there is no one right way of doing this. However the main steps are regularly the followings:

- **1.** Find the concepts/classes and their most fundamental relationships.
- **2.** Refine classes by specifying the sets of operations on them. Classify these operations. In particular, consider the needs for construction, copying, and destruction. Consider minimalism, completeness, convience.

- **3.** Refine classes by specifying their dependencies. Consider parametrization, inheritance and use dependencies
- **4.** Specify the interfaces. Separate functions into public and protected operations. Specify the exact type of the operations on the classes.

These steps are in an iterative process. Typically, several loops through this sequence are needed to produce a design one can confortably use for an initial implementation or reimplementation.

2.3 Find Classes

Find the concepts/classes and their most fundamental relationships . The key to a good design is to model some aspect of reality directly – that is, capture the concepts of an application as classes, represent the relationships between classwell-defined ways such as inheritance, and do this repeatedly at different levelabstraction. But how do we go about finding those concepts? What is a practical approach to deciding which classes we need?

The best place to start looking is in the application itself, as opposed to looking in the computer scientist's bag of abstractions and concepts. Listen to someone who will become an expert user of the system once it has been built and to someone who is a somewhat dissatisfied user of the system being replaced. Note the vocabulary they use.

It is often said that the nouns will correspond to the classes and objects needed in the program; often that is indeed the case. However, that is by no means the end of the story. Verbs may denote operations on objects, traditional (global) functions that produce new values based on the value of their arguments, or even classes. As examples of the latter, note the function objects and manipulators. Verbs such as iterate or commit can be represented by an iterator object and an object representing a database commit operation, respectively. Even adjectives can often usefully be represented by classes. Consider the adjectives storable, concurrent, registered, and bounded. These may be classes intended to allow a designer or programmer to pick and choose among desirable attributes for later-designed classes by specifying virtual base classes.

Not all classes correspond to application-level concepts. For example, some represent system resources and implementation-level abstractions. It is also important to avoid modeling an old system too closely. For example, we don't want a system that is centered around a database to faithfaspects of a manual system that exist only to allow individuals to manage the physical shuffling of pieces of paper.

2.4 Specify operations

Refine classes by specifying the sets of operations on them . Naturally, it is not possible to separate finding the classes from figuring out what operations are needed on them. However, there is a practical difference in that finding the classes is focusses on the key concepts and deliberately deemphasizes the computational aspects of the classes, whereas specifying the operations focusses on finding a complete and usable set of operations.

The strategy in considering what functions are to be provided may be the following:

- Consider how an object of the class is to be constructed, copied (if at all) and destroyed.
- Define the minimal set of operations required by the concept the class is representing. Typically these operations became the member functions.
- Consider wich operations could be added for notational convience. Include only a few really important ones. Often these operations became nonmember functions.
- Consider wich operations are to be virtual, that is, operations for which the class can act as an interface for an implementation supplied by a derived class.
- Consider what commonality of naming and functionality can be achieved across all the classes of the component.

Motto: *Try to minimalize your interface*. You can allwais extend your interface, but almost never narrow it. Note that minimalism requires more work from the designer, rather than less. When choosing operations, it is also important to focus on what to be done rather than how it is to be done.

Classification of operations:

- *Foundation operators:* constructors, destructors and copy operators.
- *Inspectors:* operations that do not modify the state of an object.
- *Modifiers:* operations that do modify the state of an object.
- *Conversions:* operations that produce an object of another type based on the value (state) of the object to which they are applied.
- *Iterators:* operations that allow aaccess to or use of a sequence of contained objects.

2.5 Specify Dependencies

Refine classes by specifying their dependencies. In C++ we consider the following relationships between classes (types):

- *Inheritance relationship*
- *Containment relationship*
- *Use relationship*
- *Programmed-in relationship*
- *Relationships within a class*

2.5.1 Inheritance relationship

Inheritance is the higher level relationship that can be represented directly in C++ and the one that figures largest in the early stages of a design. In practice inheritace is appearing in two forms.

- With inheritance we may collect the common set of attributes (data members) and behaviour (member functions) of related classes into a common base class. That procedure is called *generalization*.
- Otherwise, we may distinguish difference in data structure or in behaviour between objects of a certain classes: collecting them into different derived classes. That procedure is called *specialization*.

We express inheritance relationship in C++ in the following way:

```
class base1 { /* ... */ };
class base2 { /* ... */ };
class base3 { /* ... */ };

class derived : public base1, protected base2, private base3 { /* ... */ };
```

Of cource the derived class depends on its base classes. It is less often appreciated that if a class has a virtual function, the class depends on the derived classes. Similarly, if a class uses a protected member, that is again a dependency from the derived classes. Try to minimalize such dependencies.

2.5.2 Containment relationship

Where containment is uesd, there are two major alternatives for representing an object of a class X:

- Declare a member of type **X**.
- Declare a member of type **X** or type **X&**.

The following example demonstrates the basic usage of the different solutions:

```
class X
{
public:
    X(int);
    // ...
};

class C
{
    X a;
    X  *p;
    X  &r;
public:
    C( int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) { /* ... */ }
   ~C() { delete p; delete &r; }
};
```

Member of type **X** is preferred, because of efficiency in space, time, and human effort. The pointer solution is useful when the *contained* object may vary in the life of of the container.

The inheritance hierarchy also expresses some kind of containtment. There is a (not easy) tradeoff to decide between membership versus inheritance.

2.5.3 Use relationship

There is a number of use relationships represented in C++. Each of them introduce some dependency between the client code (which uses the other) and the code which was used.

- Using a name. This is the smallest dependency, not even necessery to use the declaration.
- Calling a member function
- Reading a member
- Writing a member
- Creating an object
- Taking the size of an object. This needs to know the declaration but doesn't depend on the constructor.

It is always a good idea to minimalize the dependencies between the different parts of the code.

2.5.4 Programmed-in relationship

Some relationship cannot be described explicitelly in the programming languages (even not in C++). In such cases the semantic should be implemented by the user code.

In C++ for example there is no language concept to describe *inheritance with delegation*. In such a case we use other relationship(s), and we program the semantic.

```
class B
{
    void f();
    void g();
    void h();
};

class A
{
    B *p;
    // ...
    void f();
    void ff();
    void g() { p->g(); }    // delegate g()
    void h() { p->h(); }    // delegate h()
};
```

A one-to-one mapping between the design concepts and the language concepts should be used wherever possible. This ensures simplicity and minimalize the possible errors.

2.5.5 Relationships within a class

Decrementing dependencies inside a class helps to code, test and maintain the code. You can use the following ideas to minimalize to specify clear relationships inside a class:

- Define class invariants
- Use assertations
- Use preconditions and postconditions for mamber functions
- Increase encapsulation

2.6 Specify Interfaces

Private functions are don't usually need to be considered at the design stage. What implementation issues must be considered in the design stage are best dealt as part of the consideration of dependencies. Even more: there is a strong rule is used by Mr. Stroustrup as a rule of thumb: *that unless at least two significantly different implementations of a class are possible, then there is probably something wrong with the class*. That it, it is simply an implementation in disguise and not a representations of a proper concept. Motto: *Ask, whether the interface to this class sufficiently implementation independent* .

Public bases and friends are part of the public interface of a class.

This is the step, where exact types of arguments are considered and specified. The ideal is to have as many interfaces as possible statically typed with application-level types.

2.7 Class in C++

The key notion of C++ is class. A C++ class is a type. Together with namespaces, classes are also a primary mechanism for information hiding. Programs can be specified in terms of user-defined types and hierarchies of such user-defined types. Both built-in and user-defined types obey statically checked type rules. Virtual functions provide a mechanism for run-time binding without breaking the static type rules. Templates support the design of parameterized types. Exceptions provide a way of making error handling more regular. These C++ features can be used without incurring overhead compared to C programs. These are the first-order properties of C++ that must be understood and considered by a designer.

In addition, generally available major libraries (such as matrix libraries, database interfaces, graphical user interface libraries, and concurrency support libraries) can strongly affect design choices.

Fear of novelty sometimes leads to sub-optimal use of C++. So does misapplication of lessons from other languages, systems, and application areas. Poor design tools can also warp designs.

Five ways designers fail to take advantage of language features and fail to respect limitations are worth mentioning:

- Ignore classes and express the design in a way that constrains implementers to use the C subset only.
- Ignore derived classes and virtual functions and use only the data abstraction subset.
- Ignore the static type checking and express the design in such a way that implementers are constrained to simulate dynamic type checking.
- Ignore programming and express systems in a way that aims to eliminate programmers.
- Ignore everything except class hierarchies.

These variants are typical for designers with

- a C, traditional CASE, or structured design background,
- an Ada83, Visual Basic, or data abstraction background,
- a Smalltalk or Lisp background,
- a nontechnical or very specialized background,
- a background with heavy emphasis on pure object-oriented programming,

Try to avoid the traps above. Use procedural part of C++, classes, inheritance, generic programming (eg. template functions and template classes), polymorphism (eg. use of virtual functions) and static typechecking in a ballance.

3 Advanced examples

In this section we present three examples. In all the examples you should implement a class on the base of user specification. This specification is given in the form of existing client code. In other words: there is a `main()` function already using your class. You should implement the class which makes the client code valid.

You may do this in iterative steps. In the client code there are sections between two to five (in correspondig the difficulty of the given problem). You can uncomment a given section, when you have implemented the corresponding part of the class. You shouldn't do any other changes in the *main* files.

Do practicing and enjoy C++!

3.1 Big integer

The purpose of that class is to store an arbitrary number of digits in a user defined type, and make some basic operation on that. This is an easy starting - no templates, no exceptions.

```
/*
 *  main.cc   - the client code for class Bigint
 *  (C) Porkolab Zoltan, ELTE, Budapest, Hungary
 *  (C) 1998
 */

#include <iostream>
#include "bigint"

class BigintCnt : public Bigint
{
public:
    BigintCnt( const char *s) : Bigint(s) { ++cnt; }
    BigintCnt( const Bigint &rhs) : Bigint(rhs) { ++cnt; }
    ~BigintCnt() { --cnt; }
    static int get_cnt() { return cnt; }
    int mark_five( int arg = 5 )  { return arg - 1; }
private:
    static int cnt;
};
int BigintCnt::cnt = 0;

int main()
{
    int iMark(1);

/* 2-es
    Bigint b1("1999" "1999" "1999" "1999" "1999");
    iMark = (++b1)[4];     // jobbrol 4-ik szamjegy int erteke
*/

/* 3-as
    Bigint b2("2000");
    Bigint b3 = b2;
    iMark += static_cast<int>(b3 - 1999);   // b3 erteke
*/

/* 4-es
    Bigint *bip = new BigintCnt("9999");
    iMark += BigintCnt::get_cnt();   // hany BigintCnt objektum el
    delete bip;
    iMark -= BigintCnt::get_cnt();   // hany BigintCnt objektum el
*/

/* 5-os
    bip = new BigintCnt("5555");
    if ( bip->Bigint::mark_five() != 5 )
    {
iMark = bip->mark_five();
```

```
    }
*/
    cout << "Your mark is " << iMark << endl;
    return 0;
}
```

3.2 Bag

A *bag* type is similar to a *set*, except that a bag has multiplicity, and you can ask it. That means, eg. put the same value twice into a bag has multiplicity 2.

This is a bit more complicated example, because *bag* is a template, and you should even throw templated exceptions. But never mind, just do everything as you did in the Basic Course, and what you haven't done there may be trivially deduced from the general rules of C++. Remember, there are almost no exceptions from language rules in C++.

```
/*
 *  main.cc   - the client code for class bag
 *  (C) Porkolab Zoltan, ELTE, Budapest, Hungary
 *  (C) 2001
 */

#include <iostream>
#include "bag.h"

using namespace std;

int main()
{
    int yourMark(1);

/* 2-es
    bag<long>  your_bag;
    your_bag.put(50);
    your_bag.put(100);
    your_bag.put(50);
    your_bag.put(400);
    yourMark = your_bag.mul(50);
*/

/* 3-as
    your_bag.put(50);
    const bag<long>  copy_of_your_bag = your_bag;
    your_bag.remove(50);
    yourMark = copy_of_your_bag.mul(50);
*/

/* 4-es
    your_bag = copy_of_your_bag;
```

```
    your_bag.put(50);
    if ( your_bag.mul(50) != copy_of_your_bag.mul(50) )
        yourMark = your_bag.mul(50);
*/

/* 5-os
    try
    {
        your_bag.remove_all(50);
        yourMark = your_bag[50];
    }
    catch( bad_value<long> bi )
    {
        std::cerr << "A(z) " << bi.value() << " ertek nem szerepel\n";
        yourMark = 5;
    }
*/
    std::cout << "Your mark is " << yourMark << endl;
    return 0;
}
```

3.3 Dictionary

Here you should implement a dictionary – or a *hash table* in Java phrase. The dictionary is an *associative array*, like the `map` class in the standard C++ library. It is a template type with *two* type parameters, the *key type*, which is used to index the associative array, and the *value type*, which is the value associated to a certain key. Otherwise this example is a natural derivative of the *bag* example.

```
/*
 *  main.cc   - the client code for class dict
 *  (C) Porkolab Zoltan, ELTE, Budapest, Hungary
 *  (C) 2001
 */

#include <iostream>
#include <string>
#include "dict.h"

int main()
{
    int yourMark(1);

/* 2-es
    dict<std::string,int>  your_index;
    your_index.put("programozási módszertan", 4);
    your_index.put("analízis", 2); // :-)
    your_index.put("programozási nyelvek I", 5);
    your_index.put("programozási nyelvek II", 2);
```

```
    yourMark = your_index.get("programozási nyelvek II");
*/

/* 3-as
    your_index.put("programozási nyelvek II", 3);
    const dict<std::string,int>  copy_of_your_index = your_index;
    yourMark = copy_of_your_index.get("programozási nyelvek II");
*/

/* 4-es
    if ( your_index["programozási nyelvek II"] ==
                  copy_of_your_index["programozási nyelvek II"] )
        ++your_index["programozási nyelvek II"];
    yourMark = your_index["programozási nyelvek II"];
*/

/* 5-os
    try
    {
        your_index["Java programozási nyelv"] = 5;
    }
    catch( bad_index<std::string> bi )
    {
        std::cerr << "A " << bi.index() << " targyat meg nem tanultuk\n";
        yourMark = your_index["programozási nyelvek II"]+1;
    }
*/
    std::cout << "Your mark is " << yourMark << endl;
    return 0;
}
```

4 Style examples

Most likely you won't only meet with C++ when you create a new program from scratch. It is highly possible you will inherit C++ code from others, and you should understand, analyze, fix and/or modify these sources.

Therefore it is important for you being able to understand others way of thinking – presented in C++ lines. You should decide which elements of that code is correct, and which elements are overcomplicated or simple mistakes. At the end you should correct the program.

Here we present three examples of this kind. Each of them is "correct" in the meaning that you can compile them and they produce the expected output. But they are not perfect at all. Several kind of mistakes may lay in these programs, from design issues to technicalities.

I have put 3 independent projects under style-1, style-2 and style-3 directories. In each directories You can read the goal of the programs in the version with suffix 'a.cpp' (eg. style-1a.cpp).

Your job is to analyze the examples, and try to improve the solutions. In the other files (eg. style-1b.cpp) you can find improved versions to get a hint,

what to do. Please, look at those files only in "case of emergency".

4.1 Hardware Store

The practice of abstraction is central to the creation of software. The single most important abstraction mechanism in C++ is the class. A class captures the common properties of the objects instantiated from it; a class characterizes the common behaviour of all objects that are its instances. Identifying appropriate abstractions is a critical part of programming in C++. To find good abstractions, the programmer must understand the underlying properties of the objects manipulated by the program.

To study the class as an abstraction mechanism, we examine a sample program and evaluate its strength and weaknesses, particulary with respect to the choice of classes. ALternative ways of writing the program with different classes and different class relationships appear. General rules of programming style, which will improve other programs, then emerge from rethinking the design and rewriting the code of the sample program.

4.1.1 Original version

A hardware store

The program determines the price of various configurations of a computer. Two selections must be made when configuring a computer: one for an expansion card and an other for monitor. The slot for an expansion card must contain one of the three options:

- CDROM drive
- Tape drive
- Networld card

The monitor must be either:

- Monochrom
- Color

Questions:

- Are the classes providing the right abstractions?
- Are there ways to eliminate complexity from the program by changing its abstractions?

In reading the program, think about how it may be simplified. Write your own version, compile, and run it. What you gain and what you loose with the new version?

```
#include <iostream>
using namespace std;

enum CARD       {CDROM, TAPE, NETWORK };
```

```
enum MONITOR    { MONO, COLOR };

class Card
{
public:
    virtual int     price() = 0;
    virtual char   *name() = 0;
    virtual int     rebate();
};

class Network : public Card
{
public:
    int     price();
    char   *name();
};

class Tape : public Card
{
public:
    int     price();
    char   *name();
};

class CDRom : public Card
{
public:
    int     price();
    char   *name();
    int     rebate();
};

class Monitor
{
public:
    virtual int     price() = 0;
    virtual char   *name() = 0;
};

class Color : public Monitor
{
public:
    int     price();
    char   *name();
};

class Monochrome : public Monitor
{
public:
    int     price();
```

```
    char   *name();
};

int   Card::rebate()  { return 45; }

int   Network::price()    { return 600; }
char *Network::name()     { return "Network"; }

int   CDRom::price()      { return 1500; }
char *CDRom::name()       { return "CDRom"; }
int   CDRom::rebate()     { return 135; }

int   Tape::price()       { return 1000; }
char *Tape::name()        { return "Tape"; }

int   Color::price()      { return 1500; }
char *Color::name()       { return "Color"; }

int   Monochrome::price() { return 500; }
char *Monochrome::name()  { return "Mono"; }

class Computer
{
public:
            Computer( CARD, MONITOR );
           ~Computer();
    int     netPrice();
    void    print();
private:
    Card    *card;
    Monitor *mon;
};

Computer::Computer( CARD c, MONITOR m )
{
    switch( c )
    {
        case NETWORK:   card = new Network;     break;
        case CDROM:     card = new CDRom;       break;
        case TAPE:      card = new Tape;        break;
    }
    switch( m )
    {
        case MONO:      mon = new Monochrome;   break;
        case COLOR:     mon = new Color;        break;
    }
}

Computer::~Computer()
{
```

```
    delete card;
    delete mon;
}

int   Computer::netPrice()
{
    return mon->price() + card->price() - card->rebate();
}

void    Computer::print()
{
    cout << "Configuration = " << card->name()
         << ", "               << mon->name()
         << ", net price = "   << netPrice()
         << endl;
}

int main()
{
    Computer nm( NETWORK, MONO );
    Computer cm( CDROM,   MONO );
    Computer tm( TAPE,    MONO );
    Computer nc( NETWORK, COLOR );
    Computer cc( CDROM,   COLOR );
    Computer tc( TAPE,    COLOR );

    nm.print();
    cm.print();
    tm.print();
    nc.print();
    cc.print();
    tc.print();

    return 0;
}
```

4.1.2 2nd version (Finding a Common Abstraction)

Motto: Concentrate common abstractions in a base class.

The class interfaces of Card and Monitor are similar: Both have price() and name() as pure virtual functions. The difference is that class Card has an additional virtual function: rebate(). Studying the similarities and differences between these classes will clarify their relationships and lead to a better program.

Card and Monitor are similar, but they are not formally related in the inheritance hierarchy. Both Card and Monitor have price() and name() memberfunctions. They are both computer components. It makes mor esense to formalize that common abstraction in a further base class called: Component. This makes the semantic of a computer explicitelly defined with the help of C++ language

tools.

```
#include <iostream>
using namespace std;

enum CARD       {CDROM, TAPE, NETWORK };
enum MONITOR    { MONO, COLOR };

class Component
{
public:
    virtual int     price() = 0;
    virtual char   *name() = 0;
    virtual int     rebate();
            int     netPrice();
};
```

Although Component adds another class to the program, it unifies Card and Monitor by identifying a common base abstraction. With the addition of Component, the program better models the program domain.

As you can read from the first implementation a card may have a rebate but a Monitor object may not. Is the distintion intentional, or merely accidental? It is better to chose a general approximation here, therefore we suppose that a Component may have a rebate. We express this to declare rebate() as a member of Component.

```
int Component::rebate()
{
    return 0;
}
```

Although price() and name() in Component are pure virtual functions, Component::rebate() is not. There is no meaningful default default implementation of price() and name() that the base class might use – that information must be specified by a derived class before an object may be instatiated. In contrast, a rebate of zero is a sound default implementation to use in a base class.

```
class Card : public Component
{
public:
    virtual int     price() = 0;
    virtual char   *name() = 0;
    virtual int     rebate();
};

class Monitor : public Component
{
public:
    virtual int     price() = 0;
    virtual char   *name() = 0;
```

```
};

class Network : public Card
{
public:
    int     price();
    char   *name();
};

class Tape : public Card
{
public:
    int     price();
    char   *name();
};

class CDRom : public Card
{
public:
    int     price();
    char   *name();
    int     rebate();
};

class Color : public Monitor
{
public:
    int     price();
    char   *name();
};

class Monochrome : public Monitor
{
public:
    int     price();
    char   *name();
};

int   Card::rebate()  { return 45; }

int   Network::price()    { return 600; }
char *Network::name()     { return "Network"; }

int   CDRom::price()      { return 1500; }
char *CDRom::name()       { return "CDRom"; }
int   CDRom::rebate()     { return 135; }

int   Tape::price()       { return 1000; }
char *Tape::name()        { return "Tape"; }
```

```
int   Color::price()      { return 1500; }
char *Color::name()       { return "Color"; }

int   Monochrome::price() { return 500; }
char *Monochrome::name()  { return "Mono"; }

class Computer
{
public:
            Computer( CARD, MONITOR );
           ~Computer();
    int     netPrice();
    void    print();
private:
    Card    *card;
    Monitor *mon;
};

Computer::Computer( CARD c, MONITOR m )
{
    switch( c )
    {
        case NETWORK:   card = new Network;     break;
        case CDROM:     card = new CDRom;       break;
        case TAPE:      card = new Tape;        break;
    }
    switch( m )
    {
        case MONO:      mon = new Monochrome;   break;
        case COLOR:     mon = new Color;        break;
    }
}

Computer::~Computer()
{
    delete card;
    delete mon;
}
```

Unification of the base abstractions (creating the common base class Component permits a simplification of the pricing. Both Card and Monitor have a rebate, so we can implement netPrice() in Component:

```
int   Component::netPrice()
{
    return price() - rebate();
}
```

As a consequece we can drastically simplify the netPrice() of Computer. It is the sum of netPrice()-es of the Components.

```
int   Computer::netPrice()
{
    //  was: return mon->price() + card->price() - card->rebate();
    return mon->netPrice() + card->netPrice();
}

void    Computer::print()
{
    cout << "Configuration = " << card->name()
         << ", "               << mon->name()
         << ", net price = "   << netPrice()
         << endl;
}

int main()
{
    Computer nm( NETWORK, MONO );
    Computer cm( CDROM,   MONO );
    Computer tm( TAPE,    MONO );
    Computer nc( NETWORK, COLOR );
    Computer cc( CDROM,   COLOR );
    Computer tc( TAPE,    COLOR );

    nm.print();
    cm.print();
    tm.print();
    nc.print();
    cc.print();
    tc.print();

    return 0;
}
```

4.1.3 3rd version (Differences between classes)

Motto: a class should describe a set of objects

We were encountered the similarities between the classes, it is time to answer the question: what makes these classes different? What is the difference between CDRom and Network? The difference is found in the derived class declarations. Neither class adds new members; they have no additional state or supplementary behaviour. The difference is in their definitions of the virtual functions price(), name() and rebate().

These virtual functions do not vary the behaviour of the objects of the different derived classes. The only difference between a Network object and a CDRom object is the values returned by their virtual functions.

In general, the behaviour of an object is the way it responds to each stimulus it can receive. Viewing an object in terms of its response to stimuli emphasizes

the independence of objects - each object is an autonomous component of a program in execution.

The most common form of stimulus is a member function call. The object responds by executing its member function, either to perform a side effect or to return a value or both. Polymorphism - virtual functions in C++ - permits the objects of different types (classes) to respond in different ways to the same stimulus. In this program, the virtual functions do not produce variation in behaviour between objects.

The difference between the derived classes can be seen from another perspective. All information in an object of one of these leaf classes is incorporated in its type. A Tape object, for example has no data members, every Tape object is equivalent to every other. There is no reason to instantiate more than one Tape object, because they must all behave in the very same way. Such a type is called Singleton, and sometimes is an essential part of a program. But a program containing only singletons - that is very suspicious.

Generality is an essential property of a program. Code fragments addressed to solve general problems are more usefull than those that are restricted to specific problems. In practice, a program cannot afford to define a different class for every object that it creates. Rather, each class should characterize a set of objects.

```
#include <iostream>
using namespace std;

enum CARD       {CDROM, TAPE, NETWORK };
enum MONITOR    { MONO, COLOR };

class Component
{
public:
        Component( int p, char *n, int r = 0);
    int     netPrice();
    int     price()  { return m_price; }
    char   *name()   { return m_name; }
    int     rebate() { return m_rebate; }
private:
    int     m_price;
    char   *m_name;
    int     m_rebate;
};

Component::Component( int p, char *n, int r)
                : m_price(p), m_name(n), m_rebate(r)  { }

int   Component::netPrice()
{
    return price() - rebate();
}
```

The 1st version of the program was drawn into the common trap of thinking

that inheritance and virtual functions are the only ways to program in C++. The excessive use of inheritance resulted in class declarations in which some classes are so specialized that each describes just one object. Inheritance and polymorphism are powerful tools when behavious varies between objects of different classes. However, in this program the variation is in values, not behaviour.

Simple data members and non-virtual functions are sufficient to represent the differences between componenct objects.

The difference between components is now a difference in value, not a difference in type. The different values are established by arguments to the constructor.

Use data members for variation in value, reserve virtual functions for variation in behaviour.

```
Component Network(600, "Network", 45);
Component CDRom(1500, "CDRom", 135);
Component Tape(1000, "Tape", 45);
Component Color(1500, "Color");
Component Monochrome(500, "Mono");
```

The objects above are automatic ones, in a real program perhaps we create them on user request from the heap.

```
class Computer
{
public:
            Computer( CARD, MONITOR );
    int     netPrice();
    void    print();
private:
    Component   *card;
    Component   *mon;
};
```

The Computer has changed accordingly the changes above. It does not create new Components and destroys them. Therefore no destructor is needed anymore.

```
Computer::Computer( CARD c, MONITOR m )
{
    switch( c )
    {
        case NETWORK:   card = & Network;     break;
        case CDROM:     card = & CDRom;       break;
        case TAPE:      card = & Tape;        break;
    }
    switch( m )
    {
        case MONO:      mon = & Monochrome;   break;
        case COLOR:     mon = & Color;        break;
    }
}
```

Unification of the base abstractions (creating the common base class Component permits a simplification of the pricing. Both Card and Monitor have a rebate, so we can implement netPrice() in Component:

As a consequece we can drastically simplify the netPrice() of Computer. It is the sum of netPrice()-es of the Components.

```
int   Computer::netPrice()
{
    return mon->netPrice() + card->netPrice();
}

void    Computer::print()
{
    cout << "Configuration = " << card->name()
         << ", "               << mon->name()
         << ", net price = "   << netPrice()
         << endl;
}
```

See the client code: no modification was needed here.

```
int main()
{
    Computer nm( NETWORK, MONO );
    Computer cm( CDROM,   MONO );
    Computer tm( TAPE,    MONO );
    Computer nc( NETWORK, COLOR );
    Computer cc( CDROM,   COLOR );
    Computer tc( TAPE,    COLOR );

    nm.print();
    cm.print();
    tm.print();
    nc.print();
    cc.print();
    tc.print();

    return 0;
}
```

4.1.4 Final version (The Role of a class)

Motto: a public derived class should be a specialization of its base class

In one respect, the program has lost ground in the transformation. The original program specified only once, in Card::rebate(), that the default rebate for cards i s45. With the current definition of Component, the values of all non-zero rebates must be specified in the object declarations. The program has no place to record a default rebate specifically for cards. The program does

need to distinguish cards from monitors. Inheritance can provide appropriate specialization with distinct constructors for cards and monitors.

We must reintroduce the classes Card and Monitor to provide constructors with the appropriate rebate defaults.

The specialization in the derived classes is limited to their constructors. The default rebates are specified as default argument values to the constructors for Card and Monitor.

```
#include <iostream>
using namespace std;

enum CARD       {CDROM, TAPE, NETWORK };
enum MONITOR    { MONO, COLOR };

class Component
{
public:
        Component( int p, char *n, int r = 0);
    int     netPrice();
    int     price()  { return m_price; }
    char   *name()   { return m_name; }
    int     rebate() { return m_rebate; }
private:
    int     m_price;
    char   *m_name;
    int     m_rebate;
};

Component::Component( int p, char *n, int r)
                : m_price(p), m_name(n), m_rebate(r)  { }

int   Component::netPrice()
{
    return price() - rebate();
}

class Card : public Component
{
public:
    Card( int p, char *n, int r = 45)
: Component( p, n, r)  {   }
};

class Monitor : public Component
{
public:
    Monitor( int p, char *n, int r = 0)
: Component( p, n, r)  {   }
};
```

Generally, public inheritance is used when the derived class is a specialization of the base class, that is, when the classes exhibit the "is a kind of" relationship. A Card is a kind of Component; a Monitor is a kind of Component. In this case, the specialization applies only during construction - once constructed, all Component objects behave uniformly. Limiting the variation in the derived classes to initialization - constructor specialization - is a legitimate use of inheritance.

It is interesting to note, that Card and Monitor now differ in a "default value", implemented as a constructor default argument value. In the original version of the program, Card and Monitor differed in a "default behaviour", implemented as a virtual function.

```
Card    Network(600, "Network");
Card    CDRom(1500, "CDRom", 135);
Card    Tape(1000, "Tape");
Monitor Color(1500, "Color");
Monitor Monochrome(500, "Mono");
```

In place of five Component objects, there are now three Card objects and two Monitor objects. Only one rebate value need be specified; only the CDRom object deviates from its default.

```
class Computer
{
public:
            Computer( Card *c, Monitor *m );
    int     netPrice();
    void    print();
private:
    Card    *card;
    Monitor *mon;
};
```

The reintroduction of Card and Monitor can simplify the program in other way. The arguments to Computer::Computer specify one CARD and one MONITOR value. The enumerations place a level of indirection between the constructor and the information it needs, without adding any flexibility. The constructor really needs the Component objects and has to map each enumeration value to an object.

Maintaining this mapping is complicated. To add another Monitor for example GreyScale, the programmer has to

- add GREY_SCALE to the MONITOR enumeration type,
- declare a GreyScale object, and
- add a GREY_SCALE case to the constructor's switch statement.

Card and Monitor are distinct types. If Computer::Computer takes pointers to objects as its arguments, the enumerations and the switch statements can be removed.

```
Computer::Computer( Card *c, Monitor *m )
{
    card = c;
    mon  = m;
}

int   Computer::netPrice()
{
    return mon->netPrice() + card->netPrice();
}

void    Computer::print()
{
    cout << "Configuration = " << card->name()
         << ", "               << mon->name()
         << ", net price = "   << netPrice()
         << endl;
}
```

Here you must modify the client code: you construct Computer with the addresses of the objects rather than the enumerators

```
int main()
{
    Computer nm( &Network, &Monochrome );
    Computer cm( &CDRom,   &Monochrome );
    Computer tm( &Tape,    &Monochrome );
    Computer nc( &Network, &Color );
    Computer cc( &CDRom,   &Color );
    Computer tc( &Tape,    &Color );

    nm.print();
    cm.print();
    tm.print();
    nc.print();
    cc.print();
    tc.print();

    return 0;
}
```

4.2 Vehicles and Garages

In the hardware store example, we examined the class relationships, and the difference between variability in behaviour and value. In that example, not all the inheritance relationships was verified.

In the following example inheritance relationships are "natural", and public inheritance is appropriate. However, the decision to use public inheritance raises other questions with respect to the detailed distribution of the and function members of the classes in the inheritance hierarchy.

4.2.1 Original version

The program manipulates vehicles, recording their entry and exit from a parking garage. Classes Car and Truck are derived from a common base class, Vehicle.

Vehicles identify themselves by printing a message with the vehicle's type (car or truck) and license plate number.

The main function exercises the Garage class by inserting and removing Truck and Car objects from a Garage called Park. The public interface to class Garage is defined in terms of pointers to Vehicle objects.

Note that Garage is more than just a bounded collection of pointers to Vehicle. Each vehicle registered in the garage has an associated bay number, the only key by which it may be referred to within the Garage.

Questions:

- Which members should be public, private or protected?
- Which functions should be virtual?
- There is a serious bug in the interaction between the base class and the derived class. Find it and fix it.

Read the program, and make suggestions to create better coupling between classes.

```
#include <stdio.h>
#include <string.h>

class Vehicle
{
public:
    Vehicle() { plate = 0; }
    Vehicle(char *p)
    {
        plate = new char[strlen(p)+1];
        strcpy(plate, p);
    }
   ~Vehicle() { delete [] plate; }
    virtual void identify() { printf("generic vehicle\n"); }

protected:
    char   *plate;
};

class Car : public Vehicle
{
public:
    Car() : Vehicle() { }
    Car(char *p) : Vehicle(p) { };
    void identify() { printf("car with plate %s\n", plate); }
};
```

```
class Truck : public Vehicle
{
public:
    Truck() : Vehicle() { }
    Truck(char *p) : Vehicle(p) { };
    void identify() { printf("truck with plate %s\n", plate); }
};

class Garage
{
public:
    Garage(int max);
   ~Garage();

    int       accept(Vehicle*);
    Vehicle  *release(int bay);
    void      listVehicles();
private:
    int       maxVehicles;
    Vehicle **parked;
};

Garage::Garage(int max)
{
    maxVehicles = max;
    parked = new Vehicle*[maxVehicles];
    for( int bay = 0; bay < maxVehicles; ++bay )
    {
        parked[bay] = 0;
    }
}

Garage::~Garage()
{
    delete [] parked;
}

int Garage::accept(Vehicle *veh)
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( !parked[bay] )
{
            parked[bay] = veh;
    return( bay );
}
    return( -1 ); // No free bay
}

Vehicle *Garage::release(int bay)
{
```

```
    if( bay < 0 || bay > maxVehicles )
        return 0;
    Vehicle *veh = parked[bay];
    parked[bay] = 0;
    return( veh );
}

void Garage::listVehicles()
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( parked[bay] )
{
            printf("Vehicle in bay %d is: ", bay);
    parked[bay]->identify();
}
}

Car c1("AAA100");
Car c2("BBB200");
Car c3("CCC300");

Truck t1("TTT999");
Truck t2("SSS888");
Truck t3("UUU777");


int main( )
{
    Garage Park(14);

    Park.accept(&c1);
    int t2bay = Park.accept(&t2);
    Park.accept(&c3);
    Park.accept(&t2);

    Park.release(t2bay);

    Park.listVehicles();

    return 0;
}
```

The output of the program is the following:

```
Vehicle in bay 0 is: car with plate AAA100
Vehicle in bay 2 is: car with plate CCC300
Vehicle in bay 3 is: truck with plate SSS888
```

4.2.2 2nd version (Consistency)

Consistency is fundamental in class design and implementation. We can speak about external consistency (in the interface) and internal consistency (in the implementation of each objects state.

When inheritance is used, there is a further issue of consistency to consider: consistency in the interface between a base class and its derived classes. In addition to the public interface, a base class may have a protected interface consisting of the protected members accessible from derived classes. By declaring protected members, a base class provides special access for use by its derived classes. The derived classes must use this access consistently with the implementation of the base class.

Car and Truck make inconsistent use of the base protected member plate. The default constructor of Vehicle set plate to 0 pointer. However identify() function in Car and Truck pass plate to printf vithout checking for a null value. If the parameter for null value, the behaviour of printf is not defined.

The following code has an undefined result:

```
Truck t;
t.identify();
```

We should fix this inconsistency.

```
#include <stdio.h>
#include <string.h>

class Vehicle
{
public:
    Vehicle() { plate = 0; }
    Vehicle(char *p)
    {
        plate = new char[strlen(p)+1];
        strcpy(plate, p);
    }
   ~Vehicle() { delete [] plate; }
    virtual void identify() { printf("generic vehicle\n"); }

protected:
    char   *plate;
};
```

The root of the problem is in the identify() member function in the derived classes. They behave inconsistently with respect to the protected interface of their base class. The base class represents a missing license plate number with a null pointer, but the derived identify() functions assume that the pointer is always non-null.

A derived class cannot be coded before its base class. A derived class should conform consistently to the conventions established by its base class. In our example, the derived classes do not account for all of the legitimate states of the base part of their objects.

To correct the bug, Car::identify() and Truck::identify() should test for null pointer and take a reasonable action to handle that case.

```
class Car : public Vehicle
{
public:
    Car() : Vehicle() { }
    Car(char *p) : Vehicle(p) { };
    void identify()
{
char *p = plate ? plate : "<node>";
printf("car with plate %s\n", p);
}
};

class Truck : public Vehicle
{
public:
    Truck() : Vehicle() { }
    Truck(char *p) : Vehicle(p) { };
    void identify()
{
char *p = plate ? plate : "<node>";
printf("truck with plate %s\n", p);
}
};
```

An other important point to see is, that Vehicle has a non-virtual destructor. We know, that for a polymorphic type a virtual destructor is very important. However, if the derived classes have no destructor defined, the non-virtual base destructor causes no problem.As long as the derived classes do not need destructors, the base class may remain as it is.

(This rule does not adequately address multiply inheritance.)

```
class Garage
{
public:
    Garage(int max);
   ~Garage();

    int       accept(Vehicle*);
    Vehicle  *release(int bay);
    void      listVehicles();
private:
    int       maxVehicles;
    Vehicle **parked;
};

Garage::Garage(int max)
{
    maxVehicles = max;
```

```
    parked = new Vehicle*[maxVehicles];
    for( int bay = 0; bay < maxVehicles; ++bay )
    {
        parked[bay] = 0;
    }
}

Garage::~Garage()
{
    delete [] parked;
}

int Garage::accept(Vehicle *veh)
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( !parked[bay] )
{
        parked[bay] = veh;
    return( bay );
}
    return( -1 ); // No free bay
}

Vehicle *Garage::release(int bay)
{
    if( bay < 0 || bay > maxVehicles )
        return 0;
    Vehicle *veh = parked[bay];
    parked[bay] = 0;
    return( veh );
}

void Garage::listVehicles()
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( parked[bay] )
{
        printf("Vehicle in bay %d is: ", bay);
    parked[bay]->identify();
}
}

Car c1("AAA100");
Car c2("BBB200");
Car c3("CCC300");

Truck t1("TTT999");
Truck t2("SSS888");
Truck t3("UUU777");
```

```
int main( )
{
    Garage Park(14);

    Park.accept(&c1);
    int t2bay = Park.accept(&t2);
    Park.accept(&c3);
    Park.accept(&t2);

    Park.release(t2bay);

    Park.listVehicles();

    return 0;
}
```

The output of the program is the following:

```
Vehicle in bay 0 is: car with plate AAA100
Vehicle in bay 2 is: car with plate CCC300
Vehicle in bay 3 is: truck with plate SSS888
```

4.2.3 3rd version (Coupling)

The inheritance hierarcy in this program is sound. Both Car and Truck are specialization of the base class Vehicle. Class Garage manages cars and truck uniformly with respect to the base class interface.

The public inheritance establishes valid "is a kind of" relationships between the base class and the derived classes. Other part of the program can deal with Car and Truck objects knowing only that they are at least Vehicle objects and they react to the interface established by the public members of Vehicle.

Although the inheritance is valid, the detailed distribution of code between the base and derived classes can be improved. To make a better version from the example, try to answer the following questions:

- What is common among classes of an inheritance hierarcy
- What are the differences between the derived classes

The common properties of all vehicles are a license plate number and the ability to identify themselves. These properties are captured in the plate base class data member, and virtual member fuction identify().

Vhat is the difference between a Car and a Truck? The only difference is the string "car" or "truck" printed by identify(). In other words, truck and car differ only in value not in their behaviour.

The similarity between Car::identify() and Truck::identify() is telling. In the following, we migrate common behaviour - represented by the identify() function - to the base class.

```
#include <stdio.h>
```

```
#include <string.h>

class Vehicle
{
public:
    Vehicle() { plate = 0; }
    Vehicle(char *p)
    {
        plate = new char[strlen(p)+1];
        strcpy(plate, p);
    }
    virtual ~Vehicle() { delete [] plate; }
    void identify()
    {
char *p = plate ? plate : "none";
printf( "%s with plate %s\n", group(), p);
    }
protected:
virtual char *group() = 0;

private:
    char   *plate;
};
```

The virtual behaviour identify() has been replaced by the virtual behaviour of group(). In fact, group() itselfmay be replaced with a data member. Plate member has been moved from the protected area to private.

Car and Vehicle has been also modified accordingly to the changes at Vehicle.

```
class Car : public Vehicle
{
public:
    Car() : Vehicle() { }
    Car(char *p) : Vehicle(p) { };

private:
char *group()  { return "car"; }
};

class Truck : public Vehicle
{
public:
    Truck() : Vehicle() { }
    Truck(char *p) : Vehicle(p) { };

private:
char *group()  { return "truck"; }
};

class Garage
```

```
{
public:
    Garage(int max);
   ~Garage();

    int       accept(Vehicle*);
    Vehicle  *release(int bay);
    void      listVehicles();
private:
    int       maxVehicles;
    Vehicle **parked;
};

Garage::Garage(int max)
{
    maxVehicles = max;
    parked = new Vehicle*[maxVehicles];
    for( int bay = 0; bay < maxVehicles; ++bay )
    {
        parked[bay] = 0;
    }
}

Garage::~Garage()
{
    delete [] parked;
}

int Garage::accept(Vehicle *veh)
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( !parked[bay] )
{
    parked[bay] = veh;
    return( bay );
}
    return( -1 ); // No free bay
}

Vehicle *Garage::release(int bay)
{
    if( bay < 0 || bay > maxVehicles )
        return 0;
    Vehicle *veh = parked[bay];
    parked[bay] = 0;
    return( veh );
}

void Garage::listVehicles()
{
```

```
    for( int bay = 0; bay < maxVehicles; ++bay )
if( parked[bay] )
{
    printf("Vehicle in bay %d is: ", bay);
    parked[bay]->identify();
}
}

Car c1("AAA100");
Car c2("BBB200");
Car c3("CCC300");

Truck t1("TTT999");
Truck t2("SSS888");
Truck t3("UUU777");


int main( )
{
    Garage Park(14);

    Park.accept(&c1);
    int t2bay = Park.accept(&t2);
    Park.accept(&c3);
    Park.accept(&t2);

    Park.release(t2bay);

    Park.listVehicles();

    return 0;
}
```

4.2.4 Final version (Value versus behaviour)

Vehicle::identify() was a virtual function in the original version of the program, and following that style group() is virtual in the revised version. Is a virtual function is the right way to capture the difference between the characteristic strings "car" and "truck"? The difference between the derived classes is one of value and not behaviour. Objects ofthe derived classes do not behave differently nor use different algorithms. A difference in value can be recorded more naturally in a data member than in a virtual function. A data member in Vehicle would be sufficient:

```
#include <stdio.h>
#include <string.h>

class Vehicle
{
```

```
public:
    // Vehicle(char *g) { group = g; plate = 0; }
    Vehicle(char *g, char *p)
    {
group = g;
        plate = new char[strlen(p)+1];
        strcpy(plate, p);
    }
    virtual ~Vehicle() = 0;  // pure virtual
    void identify()
    {
char *p = plate ? plate : "none";
printf( "%s with plate %s\n", group, p);
    }

private:
    char   *plate;
    char   *group;
};
```

The data member can receive its value from an additional argument to each Vehicle constructor, supplied by the corresponding derived constructor. The base and the derived classes interact only during initialization, further reducing coupling.

```
Vehicle::~Vehicle()
{
    delete [] plate;
}

class Car : public Vehicle
{
public:
    // Car() : Vehicle("car") { }
    Car(char *p = 0) : Vehicle("car", p) { };
};

class Truck : public Vehicle
{
public:
    // Truck() : Vehicle("truck") { }
    Truck(char *p = 0) : Vehicle("truck", p) { };
};
```

We can further reduce the code by using default argument in the constructors of the derived classes and so eliminating the second cosntructors both in derived and the base classes.

In the same time we must mentioned that when we eliminated a virtual function we increased the size of the object. So the data versus function tradeoff has a size versus runtime parallel. Here we should recognize, that eliminating the virtual function also eliminated the vptr, se reduced size.

```
class Garage
{
public:
    Garage(int max);
   ~Garage();

    int       accept(Vehicle*);
    Vehicle  *release(int bay);
    void      listVehicles();
private:
    int       maxVehicles;
    Vehicle **parked;
};

Garage::Garage(int max)
{
    maxVehicles = max;
    parked = new Vehicle*[maxVehicles];
    for( int bay = 0; bay < maxVehicles; ++bay )
    {
        parked[bay] = 0;
    }
}

Garage::~Garage()
{
    delete [] parked;
}

int Garage::accept(Vehicle *veh)
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( !parked[bay] )
{
    parked[bay] = veh;
    return( bay );
}
    return( -1 ); // No free bay
}

Vehicle *Garage::release(int bay)
{
    if( bay < 0 || bay > maxVehicles )
        return 0;
    Vehicle *veh = parked[bay];
    parked[bay] = 0;
    return( veh );
}

void Garage::listVehicles()
```

```
{
    for( int bay = 0; bay < maxVehicles; ++bay )
if( parked[bay] )
{
    printf("Vehicle in bay %d is: ", bay);
    parked[bay]->identify();
}
}

Car c1("AAA100");
Car c2("BBB200");
Car c3("CCC300");

Truck t1("TTT999");
Truck t2("SSS888");
Truck t3("UUU777");


int main( )
{
    Garage Park(14);

    Park.accept(&c1);
    int t2bay = Park.accept(&t2);
    Park.accept(&c3);
    Park.accept(&t2);

    Park.release(t2bay);

    Park.listVehicles();

    return 0;
}
```

4.3 Big Integer

In this project we created a class which represents a big integer value. "Big" means, it can hold arbitrary number of digits. We defined the basic operations on that class (operator= and operator+). The class seems to work correctly.

However we have serious efficiency problems. It seems, that operations in main() are terrible slow, and even worse: execution time is non-linear with the number of digits.

Try to detect the bottlenecks, and fix they (in iterative steps).

```
#include <stdio.h>
#include <string.h>
#include <time.h>

class BigInt
```

```
{
    friend class DigitStream;
public:
BigInt(const char*);
BigInt(unsigned n = 0);
BigInt(const BigInt&);
       ~BigInt()  { delete [] digits; }
    void operator=(const BigInt&);
    BigInt operator+(const BigInt&) const;
    void print(FILE* f = stdout) const;
private:
    char* digits;
    unsigned ndigits;
BigInt(char* d, unsigned n)  { digits = d; ndigits = n; }
};
class DigitStream
{
public:
    DigitStream(const BigInt& n)  { dp = n.digits; nd = n.ndigits; }
    unsigned operator++()  { if ( nd == 0 ) return 0;
 else           {nd--; return *dp++; } }
private:
    char* dp;
    unsigned nd;
};
void BigInt::print(FILE* f) const
{
    for (int i = ndigits-1; i>=0; i--)   fprintf(f, "%c", digits[i]+'0');
}
void BigInt::operator=(const BigInt& n)
{
    if (this == &n)  return;
    delete [] digits;
    unsigned i = n.ndigits;
    digits = new char[ndigits = i];
    char* p = digits;
    char* q = n.digits;
    while(i--) *p++ = *q++;
}
BigInt BigInt::operator+(const BigInt& n) const
{
    unsigned maxDigits = (ndigits>n.ndigits ? ndigits : n.ndigits)+1;
    char* sumPtr = new char[maxDigits];
    BigInt sum(sumPtr, maxDigits);
    DigitStream a(*this);
    DigitStream b(n);
    unsigned i = maxDigits;
    unsigned carry = 0;
    while (i--)
    {
```

```
*sumPtr = (++a) + (++b) + carry;
if ( *sumPtr >= 10 )
{
    carry = 1;
    *sumPtr -= 10;
}
else carry = 0;

sumPtr++;
    }
    return sum;
}
BigInt::BigInt(unsigned n)
{
    char d[3*sizeof(unsigned)+1];
    char* dp = d;
    ndigits = 0;

    do
    {
*dp++ = n%10;
n /= 10;
     ndigits++;
    }
    while (n > 0);

    digits = new char[ndigits];
    for (register i = 0; i < ndigits; i++) digits[i] = d[i];
}
BigInt::BigInt(const BigInt& n)
{
    unsigned i = n.ndigits;
    digits = new char[ndigits = i];
    char* p = digits;
    char* q = n.digits;
    while (i--)  *p++ = *q++;
}
BigInt::BigInt(const char* digitString)
{
    unsigned n = strlen(digitString);
    if (n != 0)
    {
digits = new char[ndigits = n];
     char* p = digits;
     const char* q = &digitString[n];
     while (n--) *p++ = *--q - '0';
    }
    else
    {
digits = new char[ndigits = 1];
```

```
digits[0] = 0;
    }
}

int main()
{
    BigInt b = 1;
    time_t t1 = time(NULL);

    for (int i = 1; i <= 50000; ++i)  b = b + 1;
    time_t t2 = time(NULL);
    printf("Elapsed time: %ld\n", t2 - t1);
    //  1000 rec =   1 sec
    //  2000 rec =   7 sec
    //  5000 rec =  38 sec
    // 10000 rec = 150 sec
    //   ... on my old machine :-)
}
```