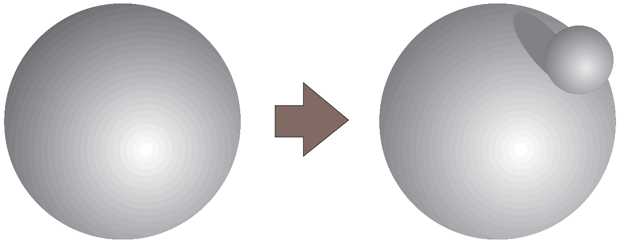
C++ Programming for Scientists

Lecture # 5

Inheritance and Polymorphism

C++ Inheritance

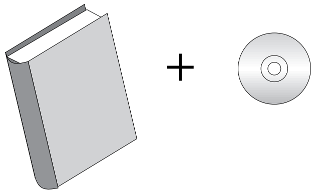


"It is easier to modify than to reinvent."

Common situations with code development

Consider our Book data structure example:

```
class Book
{
    private:
        char *title;
        char *author;
        char *publisher;
        float price;
        int num_pages;
    public:
void set_price(double);
void set_title(const char*);
        /* ... */
};
```



"Many computer books now come with software..."

Let's assume we have a database of several thousand books comprising our collection. As this collection grows, we realize that there are extra fields we want to add. For example, many books (particularly computer programming texts) come bundled with software. We'll need another field, say

```
        char *software;
```

taking on values such as "floppy", "CD-ROM", or "voucher", and so on.

How best to do this?

Solutions

Several options:

1. **Change all the structures in our database:** will have to do this every time we make a change, and as our database gets larger and larger, it becomes more and more unwieldy.
2. **Include the old data structure:** define a new data structure (say, Bookware) which contains a Book and adds the proper field:

```
class Bookware
{
    private:
        Book b;
        char *software;

    /* ... */
}
```

; but we now have to redefine all of the functions like

```
    void Bookware::set_price(double cost)
    {
        b.set_price(cost);
    }
```

which do nothing more than just call the old Book functions. A lot of needless work!

Solutions (cont'd.)

3. **Define a totally separate data structure**. Make a new copy of source code, and use an editor to create the new class:

```
class Bookware         // change from "Book" with editor
{
    private:
        char *title;
        char *author;
        char *publisher;
        float price;
        int num_pages;
        char *software;        // our contribution

    /* .. */
};
```

This is a recipe for disaster! Now the classes `Book` and `Bookware` have no relationship. Modifications to one have to be reflected in the other –a maintenance nightmare.

what's a better solution?

C+ Inheritance

Derive a new class from an old one:

```
class Bookware : public Book
{
    private:
        char *software;

    public:
        void set_software(char *);
        char *get_software();
};
```

`Bookware` automatically *inherits* all *public* functions of `Book`, e.g., one can call

```
    Bookware B;

    B.set_title("Dr. Linux");       // inherits Book::set_title()
    B.set_price(49.95);             // inherits Book::set_price()

    B.set_software("CD-ROM");       // this is the new part
```

Improvements over previous solutions:

- the C++ compiler is aware that the classes `Book` and `Bookware` are closely related. If the `Book` structure is further modified, this will be automatically reflected in `Bookware`.
- you only need to specify the new additions in the derived class; all of the other functions of `Book` are automatically included.
- (***) Application code that worked for `Book` will still work correctly with `Bookware` classes!

Application examples

```
void too_expensive(const Book &b)
{
    // print if over $300

    if (b.get_price() > 300.0)
    cout << b.get_title() << " is too expensive!\n";
}

int main()
{
    Book     B;
    Bookware Bw;

    B.set_title("The Firm");          // works just as before
    B.set_price(6.99);
    /* ... */

    Bw.set_title("Oxford Dictionary");
    Bw.set_price(799.00);
    Bw.set_software("CD-ROM");

    too_expensive(B);       // does nothing, under $300

    too_expensive(Bw);      // prints that it's too expensive.
}
```

Note that previous functions like `too_expensive()` *still* work! That is,

- we need no *modification* to existing application code for it to work with derived classes

C++ Inheritance: derived classes

- Terminology:
  - we say that `Bookware` is a *derived* class of the *base* class `Book`,
  - furthermore `Bookware` *inherits* methods and data from class `Book`.
- one can derive further from already derived classes, e.g.,

```
class B : public A
{
    /* ... */
};
```

- one can also inherit from more than one base class, e.g.

```
class D : public E, public F
{
    /* ... */
};
```

- General format for declarations:

```
class DerivedClassName :  AccessMethod BaseClassName
{
    /* ... */

}
```

where *AccessMethod* is one of the following

- `public` : inherited public members in the base class are visible from application code.
- `private` : inherited public methods in the base class are *not* callable from application code.
- `protected` : inherited methods and data in the base class are visible only to derived classes.

C++ Inheritance: Constructors & Destructors

- Arguments to constructors are passed to base classes via the ’:’ notation, e.g.

```
class D : public A, public B, public C { /* ... */ };

D::D() : A(), B(), C()
{
    /* ... */
}
```

where A(), B(), C() can be substituted with the appropriate constructor.

C++ Inheritance: Constructors & Destructors

- constructors for the base classes are always called first, destructors for base classes are always called last, e.g.

```
class base
{ public:
    base()   { cout << "constructing base.\n";}
    ~base()  { cout << "destructing base.\n";}
};

class derived : public base
{   public:
    derived()   { cout << "constructing derived.\n";}
    ~derived()  { cout << "destructing derived.\n";}
};

int main()
{
    derived A;
    return 0;
}
```

will print out

```
constructing base.
constructing derived.
destructing derived.
destructing base.
```

Modifying a member in a derived class

The member function `display_info()` is slightly different for `Bookware`; it displays the price with the string ”(software included)”.

```
void Book::display_info(void)
{
    cout << "Price:  $" << get_price() << "\n";
}

void Bookware::display_info(void)
{
    cout << "Price:  $" << get_price() << " (software included) \n";
}
```

Notice that the new `Bookware::display_price()` function overrides the old definition.

```
    Book *pB, B;
    Bookware *pBw, Bw;

    /* ... */

    B.display_info();          // calls Book::display_info()
    pB->display_info();        // calls Book::display_info()

    Bw.display_info();         // calls Bookware::display_info()
    pBw->display_info();       // calls Bookware::display_info()

    pB = &Bw;                  //  Book* = &Bookware (legal!)
    pB->display_info();        //    but calls Book::display_info(), not
                               //    not Bookware::display_info() !
```

How to integrate derived and base classes?

Notice, however, we have a problem:

how we do mix `Books` and `Bookware` items together?

In other words, let’s say we have a list of books and we want to display their prices:

```
void display_list_of_prices(Book *L[], int N)
{
    int i;

    for (i=0; i<N; i++)
    {
        L[i]->display_info();
    }
}
```

Unfortunately, this won’t do the “right” thing:

```
    Book B;
    Bookware C;
    Book *L[] = {&B, &C};

    B.set_price(6.99);
    C.set_price(29.95);


    diplay_list_of_prices(L, 2);
```

Prints out

```
Price: $6.99
Price: $25.95
```

C++: Virtual functions

Virtual functions are a mechanism for getting at the methods of derived classes through pointers of a *base* class.

```
#include <iostream.h>

class base
{ public:
    void virtual print() { cout << "calling base.print().\n";}
};

class derived : public base
{ public:
    void print() { cout << "calling derived.print().\n";}
};

int main()
{
    base A;
    derived B;
    base *pb;

    A.print();          // calls base::print()
    B.print();          // calls derived::print()

    pb = &B;

    pb->print();        // what does this call?

}
```

Output looks like

```
calling base.print().
calling derived.print().
calling derived.print().
```

Case statement considered harmful...

Anytime you are trying to deal with a collection of related types and see code like this

```
    switch (x->type)
    {
        case TYPE1 :  ((type1 *)x.p)->f();  // call type1::f()
                      break;

        case TYPE2 :  ((type2 *)x.p)->f();  // call type2::f()
                      break;

        case TYPE3 :  ((type3 *)x.p)->f();  // call type3::f()
                      break;

        default:  /* ???? */
    }
```

replace it with inheritance and let the compiler manage it instead!

Inheritance means no case statements...

```
    class base
    {
        public:
            virtual void f();
        /* ... */
    };

    class type1 : public base { /* ... */ };    // new definitions of f()
    class type2 : public base { /* ... */ };
    class type3 : public base { /* ... */ };

int main()
{
    base *x;

    /* ... */

    x->f();        // essentially replaces previous case statement!

    /* ... */
}
```

Why is this better?

- if you decide to add another type, e.g. `TYPE4`, you have to modify *every* subroutine that relies on this `case` statement, including **application codes!**
- responsibility is on the **developer** to use the predefined macros `TYPE1`, `TYPE2`, etc. correctly. (This is a mundane task best suited for a compiler.)

OO Programming: Inheritance + Polymorphism

So what is Object Oriented programming, anyway?

- while there are several interpretations of the "details", (exactly what constitutes an object oriented language, and so forth) most experts agree that it should provide at least
  - data encapsulation
  - inheritance
  - polymorphism
- How does C++ provide these?
  - classes, together with `private` and `public` sections provide a useful form of **data encapsulation**
  - deriving new classes from base classes provides a form of class **inheritance**
  - function and operator overloading, together with C++ templates provide forms of compile-time **polymorphism**.
  - virtual functions provide a form of run-time **polymorphism**.