南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 12, class inheritance & polymorphism

廖琪梅，王大兴，于仕琪，王薇

Topic

- **class inheritance & polymorphism in C++**
  - Class inheritance
  - Polymorphism (virtual function)
  - Inheritance and dynamic memory allocation

- **class inheritance & polymorphism in Python**
  - Public inheritance, explicit call, dynamic resource management
  - Duck Type

Class inheritance

Inheritance syntax:

```
class derived_class_name : access_mode   base_class_name
{
    // body of subclass
};
```

Subclass,Derived class, Child class

public, protected, private

Base class, Super class, Parent class

Public inheritance represents an ***is-a*** relationship, it means every derived-class object ***is an*** object of its base class.

If you do not provide a copy constructor or an assignment operator for the base class and the derived class, the compiler will generate a copy constructor and assignment operator for both of them respectively.

Default copy constructor and assignment operator

If no copy constructor or an assignment operator are designed for the class, the compiler will generate a copy constructor and assignment operator for the class.

```
class Parent{
private:
    int id;
    string name;
public:
    Parent():id(1),name("null"){
        cout<<"calling default constructor Parent()\n";
    }
    Parent(int i, string n):id(i),name(n){
        cout<<"calling default constructor Parent(int,string)\n";
    }
    friend ostream& operator<<(ostream& os, const Parent& p){
        return os<<"Parent:"<<p.id<<","<<p.name<<endl;
    }
};
```

```
int main(){

    Parent p1(101,"Liming");
    cout<<"values in p1:\n"<<p1<<endl;

    Parent p2(p1);
    cout<<"values in p2:\n"<<p2<<endl;

    Parent p3;
    cout<<"before assignment, values in p3:\n"<<p3<<endl;

    p3=p2;
    cout<<"values in p3:\n"<<p3<<endl;

    return 0;

}
```

```
calling default constructor Parent(int,string)
values in p1:
Parent:101,Liming

values in p2:
Parent:101,Liming

calling default constructor Parent()
before assignment, values in p3:
Parent:1,null

values in p3:
Parent:101,Liming
```

The behavior of the default copy constructor and overloaded assignment operator generated by the compiler is:
No extra information printed, assign the attribute values in the rhs object to the target object's attributes.

If **no copy constructor or assignment operator provided for the base class and the derived class,**

the **compiler will generate** a copy constructor and assignment operator for both
of them respectively.

```
class Parent
{
private:
    int id;
    string name;

public:
    Parent():id(1),name("null")
    {
        cout << "calling default constructor Parent()\n";
    }
    Parent(int i,string n) :id(i),name(n)
    {
        cout << "calling Parent constructor Parent(int,string)\n";
    }

    friend ostream& operator<<(ostream& os, const Parent& p)
    {
        return os << "Parent:" << p.id << "," << p.name << endl;
    }
};
```

```
class Child :public Parent
{
private:
    int age;

public:
    Child():age(0)
    {
        cout << "call Child default constructor Child()\n";
    }
    Child(int age) :age(age)
    {
        cout << "calling Child constructor Child(int)\n";
    }
    Child(const Parent& p, int age) :Parent(p), age(age)
    {
        cout << "calling Child constructor Child(Parent,int)\n";
    }
    friend ostream& operator<<(ostream& os, const Child& c)
    {
        return os << (Parent&)c << "Child:" << c.age << endl;
    }
};
```

The derived class will call the default constructor of the base class to initialize the data members.

Call Parent copy constructor

```
int main()
{
    Parent p(101, "Liming");
    Child c1(19);
    cout << "values in c1:\n" << c1 << endl;

    Child c2(p, 20);
    cout << "values in c2:\n" << c2 << endl;

    Child c3 = c2;
    cout << "values in c3:\n" << c3 << endl;

    Child c4;
    cout << "Before assignment,values in c4:\n" << c4 << endl;

    c4 = c2;
    cout << "values in c4:\n" << c4 << endl;

    return 0;
}
```

Call Child copy constructor

Call Child assignment operator

```
calling Parent constructor Parent(int,string)
calling default constructor Parent()
calling Child constructor Child(int)
values in c1:
Parent:1,null
Child:19

calling Child constructor Child(Parent,int)
values in c2:
Parent:101,Liming
Child:20

values in c3:
Parent:101,Liming
Child:20

calling default constructor Parent()
call Child default constructor Child()
Before assignment,values in c4:
Parent:1,null
Child:0

values in c4:
Parent:101,Liming
Child:20
```

Copy Constructor of Child(1): without initializing the base class component.

Define copy constructor of Child without initializing the base class component.

```
Child(const Child& c) :age(c.age)
{
    cout << "calling Child copy constructor Child(const Child&)\n";
}
```

```
int main(){
    Parent p(101,"Liming");
    Child c1(19);
    cout<<"values in c1:\n"<<c1<<endl;

    Child c2(p,20);
    cout<<"values in c2:\n"<<c2<<endl;

    Child c3(c2);
    cout<<"values in c3:\n"<<c3<<endl;

    Child c4;
    cout<<"Before assignment, values in c4:\n"<<c4<<endl;

    c4=c2;
    cout<<"values in c4:\n"<<c4<<endl;

    return 0;
}
```

without initializing the base class component

```
calling default constructor Parent(int,string)
calling default constructor Parent()
calling default constructor Child(int)
values in c1:
Parent:1,null
Child:19

calling default constructor Child(Parent,int)
values in c2:
Parent:101,Liming
Child:20

calling default constructor Parent()
calling Child copy constructor Child(const Child& c)
values in c3:
Parent:1,null
Child:20

calling default constructor Parent()
calling default constructor Child()
Before assignment, values in c4:
Parent:1,null
Child:0

values in c4:
Parent:101,Liming
Child:20
```

Copy Constructor of Child(2) : with initializing the base class component.

Define copy constructor of Child with initializing the base class component.

```
Child(const Child& c):Parent(c),age(c.age){
        cout<<"calling Child copy constructor Child(const
Child& c) with Parent initialed\n";
}
```

```
int main(){
    Parent p(101,"Liming");
    Child c1(19);
    cout<<"values in c1:\n"<<c1<<endl;

    Child c2(p,20);
    cout<<"values in c2:\n"<<c2<<endl;

    Child c3(c2);
    cout<<"values in c3:\n"<<c3<<endl;

    Child c4;
    cout<<"Before assignment, values in c4:\n"<<c4<<endl;

    c4=c2;
    cout<<"values in c4:\n"<<c4<<endl;

   return 0;
}
```

initializing the base class component

```
calling default constructor Parent(int,string)
calling default constructor Parent()
calling default constructor Child(int)
values in c1:
Parent:1,null
Child:19

calling default constructor Child(Parent,int)
values in c2:
Parent:101,Liming
Child:20

calling Child copy constructor Child(const Child& c) with Parent initialed
values in c3:
Parent:101,Liming
Child:20

calling default constructor Parent()
calling default constructor Child()
Before assignment, values in c4:
Parent:1,null
Child:0

values in c4:
Parent:101,Liming
Child:20
```

Note:

When creating an object of a derived class, a program first calls the base-class constructor and then calls the derived-class constructor. The base-class constructor is responsible for initializing the inherited data member. The derived-class constructor is responsible for initializing any added data members. A derived-class constructor always calls a base-class constructor.

To destroy an derived class object, the program first calls the derived-class destructor and then calls the base-class destructor. That is, **destroying an object occurs in the opposite order used to constructor an object**.

The table below shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

In a base class definition, if a member declared as **protected** can be directly accessed by the **derived classes** but cannot be directly accessed by the general program.

Polymorphism

Polymorphism is one of the most important feature of object-oriented programming. **Polymorphism** works on object **pointers** and **references** using so-called **dynamic binding** at run-time. It does not work on regular objects, which uses static binding during the compile-time.

There are **two key mechanisms for implementing polymorphic public inheritance:**

1. **Redefining base-class methods in a derived class**
2. **Using virtual methods**

Polymorphism - demo

```
#include<iostream>
#include<string>
using namespace std;
class Employee{
private:
    string name;
    string ssn;
public:
    Employee(const string& n, const string& s): name(n),ssn(s){
        cout<<"The base class constructor is invoked."<<endl;
    }
    ~Employee(){
        cout<<"The base class destructor is invoked."<<endl;
    }
    string getName() const { return name;}
    string getSSN() const { return ssn; }
    void setName(const string& n){ name=n; }
    void setSSN(const string& s) { ssn=s; }
    double earning(){ return 0; }
    virtual void show(){
        cout<<"Name is:"<<name<<", SSN number is: "<<ssn<<endl;
    }
};
```

base class

If the access specifier is **protected**, the derived class can access the data

If the keyword **virtual** used, the program choose a method based on the type of object the reference or pointer refers to rather than based on the reference type or pointer type.

derived class

base class

```
class SalariedEmployee : public Employee{
private:
    double salary;
public:
    SalariedEmployee(const string& name, const string& ssn,
double s):
    Employee(name,ssn),salary(s){
        cout<<"The derived calss constructor is
invoked."<<endl;
    }
    ~SalariedEmployee(){
        cout<<"The derived class destructor is invoked."<<endl;
    }
    SalariedEmployee(const Employee& e, double s):Employee(e),
salary(s){ }
    double getSalary()const { return salary; }
    void setSalary(doubles) { salary=s; }
    double earning(){ return getSalary();}
    void show(){
        cout<<"Name is:"<<getName()<<", SSN number is: "
            <<getSSN()<<", salary is:"<<salary<<endl;
    }
};
```

invoke base class method in derived class to get the name and snn

override the function **show()** in SalariedEmployee

```
int main()
{
    Employee e("Liming", "1000");
    SalariedEmployee se("Wangfang", "1001", 2000);

    Employee* pe = &e;
    pe->show();

    pe = &se;
    pe->show();

    return 0;
}
```

```
Name is:Liming,SSN number is: 1000
Name is:Wangfang,SSN number is: 1001,Salary is:2000
```

The pointer type of **pe** is Employee, it points to a different object respectively, and invokes different objects' **show()** functions. This is polymorphism.

Destructors: not virtual vs virtual

```
int main()
{
    Employee* pe = new SalariedEmployee("Wangfang", "1001", 2000);

    pe->show();

    delete pe;

    return 0;
}
```

```
class Employee{
private:
    string name;
    string ssn;
public:
    ~Employee(){
        cout<<"The base class
destructor is invoked."<<endl;
//
    }
```

```
The base class constructor is invoked.
The derived class constructor is invoked.
Name is:Wangfang,SSN number is: 1001,Salary is:2000
The base class destructor is invoked.
```

If the destructors is **not virtual**, the delete statement invokes the ***~Employee()*** destructor.
This frees memory pointed to by the ***Employee*** component of the ***SalariedEmployee*** object not memory pointed to by **SalariedEmployee** component.

If the destructor is **virtual**, the same code invokes the ***~SalariedEmployee()*** destructor, which frees memory pointed to by the **SalariedEmployee** component, and then calls the ***~Employee()*** destructor to free memory pointed to by the ***Employee*** component.

```
class Employee{
private:
    string name;
    string ssn;
public:
    virtual ~Employee(){
        cout<<"The base class
destructor is invoked."<<endl;
//
    }
```

```
The base class constructor is invoked.
The derived class constructor is invoked.
Name is:Wangfang,SSN number is: 1001,Salary is:2000
The derived class destructor is invoked.
The base class destructor is invoked.
```

Inheritance and Dynamic Memory Allocation

If a **base class** uses dynamic memory allocation and redefines assignment operator and a copy constructor, how does that affect the implementation of the **derived class**? The answer depends on the nature of the derived class.

If the **derived class does not itself use dynamic memory allocation**, you needn't take any special steps.

If the **derived class does use new to allocate memory**, you do have to define an explicit destructor, copy constructor, and assignment operator for the derived class.

```
class Parent
{
private:
    int id;
    char* name;

public:
    Parent(int i = 0, const char* n = "null");
    Parent(const Parent& p);
    virtual ~Parent();
    Parent& operator=(const Parent& prhs);

    friend ostream& operator<<(ostream& os, const Parent& p);
};
```

base class

```
Parent::Parent(int i, const char* n)
{
    cout << "calling Parent default constructor Parent()\n";
    id = i;
    name = new char[std::strlen(n) + 1];
    strcpy_s(name, std::strlen(n) + 1, n);
}
```

base class constructor

derived class constructor

```
Child::Child(int i, const char* n, const char* s, int a) : Parent(i, n)
{
    cout << "call Child default constructor Child()\n";
    style = new char[std::strlen(s) + 1];
    strcpy_s(style, std::strlen(s) + 1, s);
    age = a;
}
```

derived class

```
class Child :public Parent
{
private:
    char* style;
    int age;

public:
    Child(int i = 0, const char* n = "null", const char* s = "null", int a = 0);
    Child(const Child& c);
    ~Child();
    Child& operator=(const Child& crhs);

    friend ostream& operator<<(ostream& os, const Child& c);
};
```

The data fields both in the base class and the derived class hold pointers, which indicate they would use dynamic memory allocation.

```
Parent:: ~Parent()
{
    cout << "Call Parent destructor.\n";
    delete[] name;
}

Child::~Child()
{
    cout << "call Child destructor.\n";
    delete[] style;
}
```

A derived class destructor automatically calls the base-class destructor, so its own responsibility is to clean up after what the derived-class destructors do.

Consider copy constructor:

the base-class copy constructor

```
Parent::Parent(const Parent& p)
{
    cout << "calling Parent copy constructor Parent(const Parent&)\n";
    id = p.id;
    name = new char[std::strlen(p.name) + 1];
    strcpy_s(name, std::strlen(p.name) + 1, p.name);
}
```

The derived class copy constructor can only accesses to its own data, so it must invoke the **base-class** copy constructor to handle the **base-class** share of the data.

```
Child::Child(const Child& c) : Parent(c)
{
    cout << "calling Child copy constructor Child(const Child&)\n";
    style = new char[strlen(c.style) + 1];
    strcpy_s(style, std::strlen(c.style) + 1, c.style);
    age = c.age;
}
```

The member initialization list passes a **Child** reference to a **Parent** constructor.
The **Parent** copy constructor has a **Parent** reference parameter, and a base class reference can refer to a derived type. Thus, the **Parent** copy constructor uses the **Parent** portion of the **Child** argument to constructor the **Parent** portion of the new object.

Consider assignment operators:

```
Parent& Parent::operator=(const Parent& prhs)
{
    cout << "call Parent assignment operator:\n";
    if (this == &prhs)
        return *this;

    delete[] name;
    this->id = prhs.id;
    name = new char[std::strlen(prhs.name) + 1];
    strcpy_s(name, std::strlen(prhs.name) + 1, prhs.name);

    return *this;
}
```

the **Parent** assignment operator

```
Child& Child::operator=(const Child& crhs)
{
    cout << "call Child assignment operator:\n";
    if (this == &crhs)
        return *this;
    Parent::operator=(crhs);

    delete[] style;
    style = new char[std::strlen(crhs.style) + 1];
    strcpy_s(style, std::strlen(crhs.style) + 1, crhs.style);
    age = crhs.age;

    return *this;
}
```

Because **Child** uses dynamic memory allocation, it needs an explicit assignment operator. Being a **Child** method, it only has direct access to its own data.

An explicit assignment operator for a derived class also has to take care of assignment for the inherited base class **Parent** object. You can accomplish this by explicitly calling the base class assignment operator.

```cpp
int main()
{
    Parent p1;
    cout << "values in p1\n" << p1 << endl;
    Parent p2(101, "Liming");
    cout << "values in p2\n" << p2 << endl;

    Parent p3(p1);
    cout << "values in p3\n" << p3 << endl;
    p1 = p2;
    cout << "values in p1\n" << p1 << endl;

    Child c1;
    cout << "values in c1\n" << c1 << endl;
    Child c2(201, "Wuhong", "teenager", 15);
    cout << "values in c2\n" << c2 << endl;
    Child c3(c1);
    cout << "values in c3\n" << c3 << endl;
    c1 = c2;
    cout << "values in c1\n" << c1 << endl;

    return 0;
}
```

```
calling Parent default constructor Parent()
values in p1
Parent:0,null

calling Parent default constructor Parent()
values in p2
Parent:101,Liming

calling Parent copy constructor Parent(const Parent&)
values in p3
Parent:0,null

call Parent assignment operator:
values in p1
Parent:101,Liming

calling Parent default constructor Parent()
call Child default constructor Child()
values in c1
Parent:0,null
Child:null,0

calling Parent default constructor Parent()
call Child default constructor Child()
values in c2
Parent:201,Wuhong
Child:teenager,15

calling Parent copy constructor Parent(const Parent&)
calling Child copy constructor Child(const Child&)
values in c3
Parent:0,null
Child:null,0

call Child assignment operator:
call Parent assignment operator:
values in c1
Parent:201,Wuhong
Child:teenager,15

call Child destructor.
Call Parent destructor.
call Child destructor.
Call Parent destructor.
call Child destructor.
Call Parent destructor.
Call Parent destructor.
Call Parent destructor.
Call Parent destructor.
```

Three child objects

Three parent objects

class inheritance in Python(1)

1. Inheritance in Python is **public inheritance**, providing flexible data access
2. The inheritance mechanism in Python is **explicit** and **requires calling the constructor of the parent class during initialization**, which can be implemented through **super**()
3. Assignment in Python is usually a reference, and deep copy requires the **deepcopy** method of the copy module
4. Python **does not allow overloading operator**, using method instead(**assign**, **__str__** here)

```python
import copy
class Parent:
    def __init__(self, i=0, n="null"):
        print("calling Parent default constructor Parent()")
        self.id=i
        self.name=n

    def __deepcopy__(self, memo):
        print("calling Parent copy constructor Parent(const Parent&)")
        new_obj=self.__class__(self.id, self.name)
        memo[id(self)] =new_obj
        return new_obj

    def assign(self, other):
        print("call Parent assignment operator:")
        if self is other:
            return self
        self.id=other.id
        self.name=other.name
        return self

    def __del__(self):
        print("call Parent destructor.")

    def __str__(self):
        return f"Parent:{self.id}, {self.name}\n"
```

```python
class Child(Parent):
    def __init__(self, i=0, n="null", s="null", a=0):
        super().__init__(i, n)
        print("call Child default constructor Child()")
        self.style=s
        self.age=a

    def __deepcopy__(self, memo):
        print("calling Child copy constructor Child(const Child&)")
        new_obj=self.__class__(self.id, self.name, self.style, self.age)
        memo[id(self)] =new_obj
        return new_obj

    def assign(self, other):
        print("call Child assignment operator:")
        if self is other:
            return self
        super().assign(other)
        self.style=other.style
        self.age=other.age
        return self

    def __del__(self):
        super().__del__()
        print("call Child destructor.")
    def __str__(self):
        parent_str=super().__str__()
        return f"{parent_str}Child:{self.style}, {self.age}\n"
```

class inheritance in Python(2)

5. Destructors in Python rely on garbage collection mechanisms.

6. Rewriting __del__ requires caution and attention to potential memory risks.

```
if__name__=="__main__":

    p1=Parent()
    print("value in p1\n", p1)

    p2=Parent(101, "Liming")
    print("value in p2\n", p2)

    p3=copy.deepcopy(p1)
    print("value in p3\n", p3)

    p1.assign(p2)
    print("value in p1\n", p1)


    c1=Child()
    print("value in c1\n", c1)

    c2=Child(201, "Wuhong", "teenager", 15)
    print("value in c2\n", c2)

    c3=copy.deepcopy(c1)
    print("value in c3\n", c3)

    c1.assign(c2)
    print("value in c1\n", c1)

    del c3, c2, c1, p3, p2, p1
```

```
calling Parent default constructor Parent()
value in p1
 Parent:0, null

calling Parent default constructor Parent()
value in p2
 Parent:101, Liming

calling Parent copy constructor Parent(const Parent&)
calling Parent default constructor Parent()
value in p3
 Parent:0, null

call Parent assignment operator:
value in p1
 Parent:101, Liming

calling Parent default constructor Parent()
call Child default constructor Child()
value in c1
 Parent:0, null
Child:null, 0

calling Parent default constructor Parent()
call Child default constructor Child()
value in c2
 Parent:201, Wuhong
Child:teenager, 15

calling Child copy constructor Child(const Child&)
calling Parent default constructor Parent()
call Child default constructor Child()
value in c3
 Parent:0, null
Child:null, 0

call Child assignment operator:
call Parent assignment operator:
value in c1
 Parent:201, Wuhong
Child:teenager, 15
```

Three child objects

```
call Parent destructor.
call Child destructor.
call Parent destructor.
call Child destructor.
call Parent destructor.
call Child destructor.
call Parent destructor.
call Parent destructor.
call Parent destructor.
```

Three parent objects

Python garbage collection timing uncertain, manually triggering decomposition here.

Duck type in Python

- **Polymorphism**: allowing different objects of the same class to exhibit different behaviors towards the same method name.
- **Duck type** in Python: focuses on the behavior of objects (whether methods or properties exist), rather than their specific types or inheritance relationships.
  - For example, as long as an object has a speak() method, it can be called uniformly, regardless of whether it inherits from the same parent class or not.

```
class Animal:
        def __init__(self, name):
            self.name=name
class Duck(Animal):
        def __init__(self, name):
            super(Duck, self).__init__(name)
        def Speak(self):
            print(self.name, ' Quack')
class Dog(Animal):
        def __init__(self, name):
            super(Dog, self).__init__(name)
        def Speak(self):
            print(self.name, ' Quack')
class Robot:
    def __init__(self, name):
       self.name=name
    def Speak(self):
        print(self.name, 'Beep')
```

```
def testSpeak(duck):
    duck.Speak()

if __name__ == "__main__":
    duck=Duck('Tommy')
    dog=Dog('Fox')
    robot=Robot('Wall-E')

    testSpeak(duck)
    testSpeak(dog)
    testSpeak(robot)
```

```
Tommy  Quack
Fox  Quack
Wall-E Beep
```

Exercise:

1. Please find the errors of the following code and explain why to SA.

```
class Base
{
   private:
      int x;
   protected:
      int y;
   public:
      int z;
   void funBase (Base& b)
   {
      ++x;
      ++y;
      ++z;
      ++b.x;
      ++b.y;
      ++b.z;
    }
};
```

```
class Derived:public Base
{
   public:
      void funDerived (Base& b, Derived& d)
      {
         ++x;
         ++y;
         ++z;
         ++b.x;
         ++b.y;
         ++b.z;
         ++d.x;
         ++d.y;
         ++d.z;
      }
};
```

```
void fun(Base& b, Derived& d)
{
   ++x;
   ++y;
   ++z;
   ++b.x;
   ++b.y;
   ++b.z;
   ++d.x;
   ++d.y;
   ++d.z;
}
```

Exercise:

2. Run the following program, and explain the result to SA.

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

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

```
class Rectangle: public Polygon
{
  public:
    int area()
      { return width * height; }
};

class Triangle: public Polygon
{
  public:
    int area()
      { return width*height/2; }
};
```

```
int main ()
{
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (2,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

Exercise:

3. Run the following program, and explain the result to SA. Is there any problem in the program?

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

class Polygon
{
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};
```

```
class Rectangle: public Polygon
{
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon
{
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};
```

```
int main ()
{
  Polygon * ppoly = new Rectangle (4,5);
  ppoly->printarea();
  ppoly = new Triangle (2,5);
  ppoly->printarea();

  return 0;
}
```

Exercise:

- 4. Write a simple Rust “Hello world” program, compile and run it.