Inheritance, Polymorphism and the Object Memory Model

- how objects are stored in memory at runtime?
- **compiler** - operations such as access to a member of an object are compiled
- **runtime** - implementation of operations such as new and delete

Object-class in memory

- object= instance of a class
  - class defines characteristics of instances: data members (state)/member functions (methods).
- object is implemented at runtime as a region of storage (a contiguous block of memory)
- class defines the memory layout of **all** the objects that belong to that class

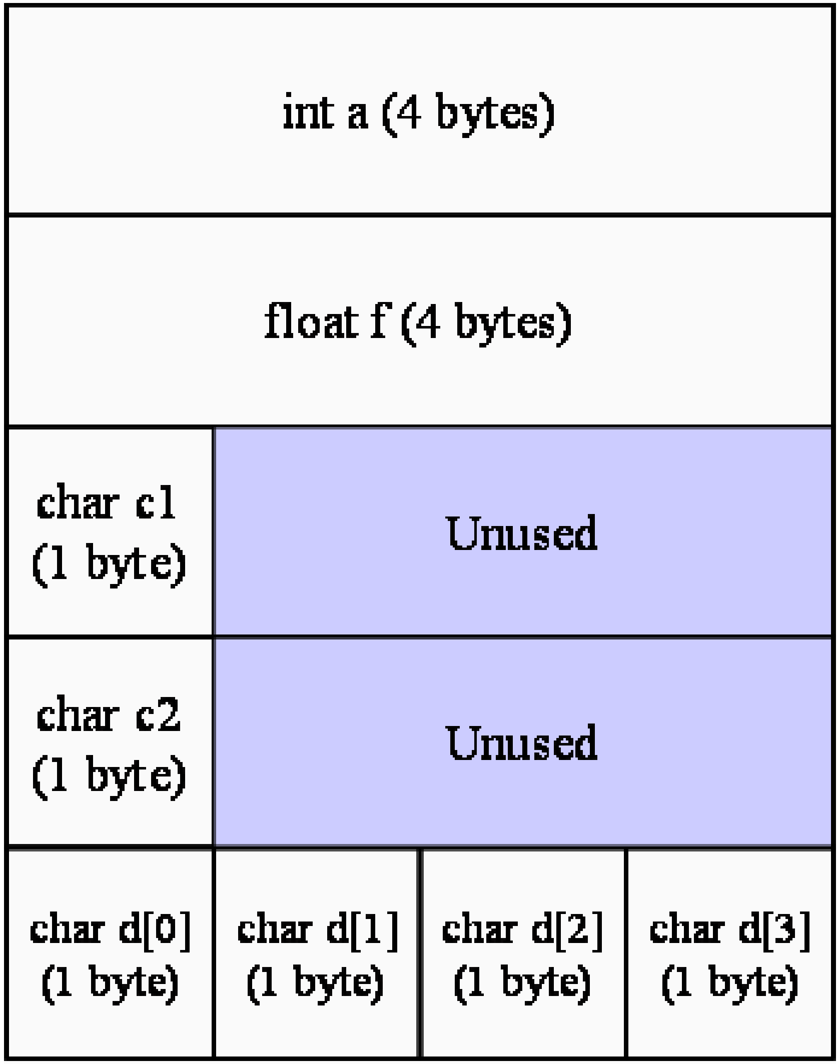
Object-class in memory

- object of class is allocated a copy of all class *data members*
  - static members allocated once
- objects of class share *member functions* (methods)
  - code for functions is stored only once in memory for each class.

object values / object references

- object references is as a pointer to an object value
- object values are implemented as a contiguous block of memory, where each field (data member) is stored in sequence

```
class A {
   int a;
   float f;
   char c1;
   char c2;
   char d[4]; // An array of 4 char values
};
```



int a (4 bytes)			
float f (4 bytes)			
char c1 (1 byte)	Unused		
char c2 (1 byte)	Unused		
char d[0] (1 byte)	char d[1] (1 byte)	char d[2] (1 byte)	char d[3] (1 byte)

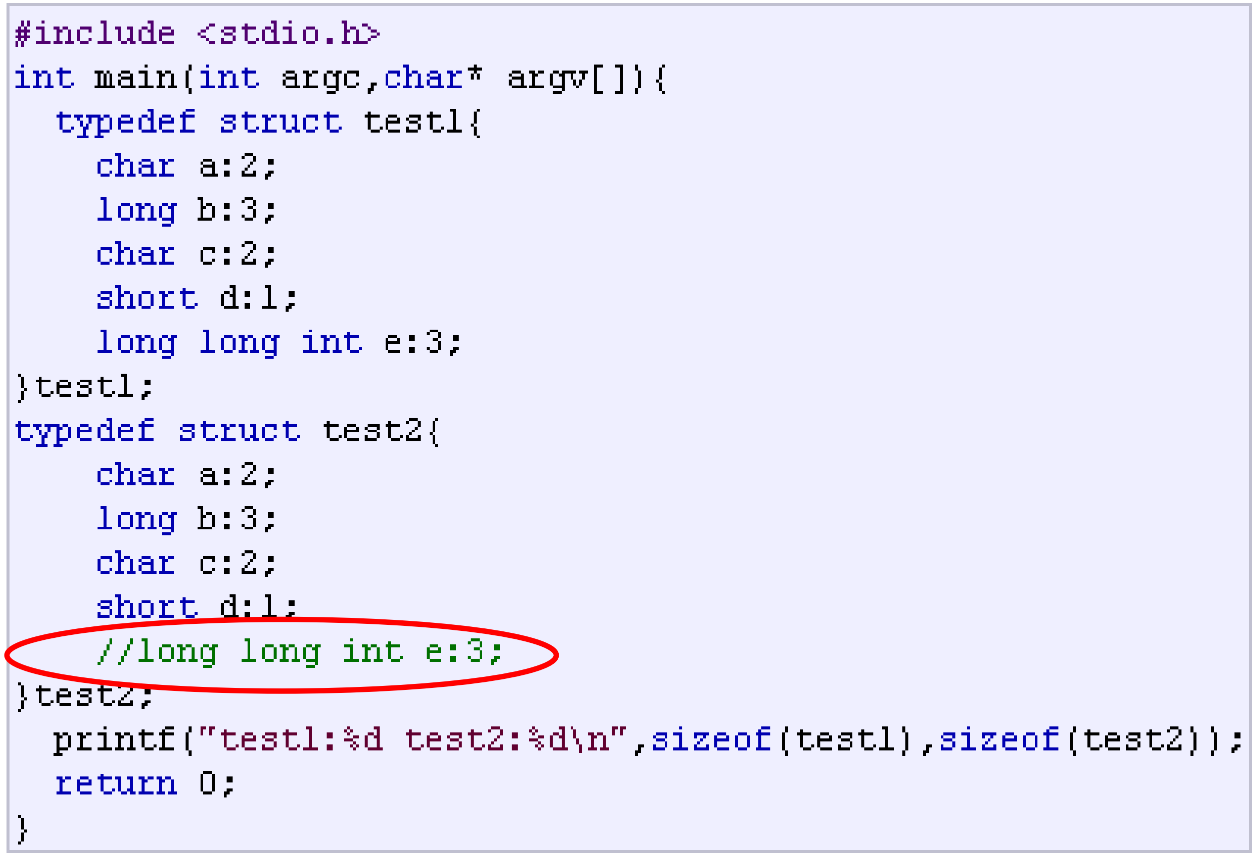
sizeof()

- primitive type is encoded in a fixed amount of memory.
  - int 4 bytes, char 1 byte, double 8 bytes… etc.
- sizeof() - size used by a given type.
  - computed at compile-time
  - a compiler operator
  - can return size allocated for object data-types
  - sizeof(*A*) = 20 (5 words of 4 bytes).

Field Alignment

- fields c1 and c2 are "word aligned" within the block of memory of the object:
  - fields start on a word boundary (word=4b)
  - memory left "wasted"
  - compiler flag not to align fields
- aligning fields - easy data accessing

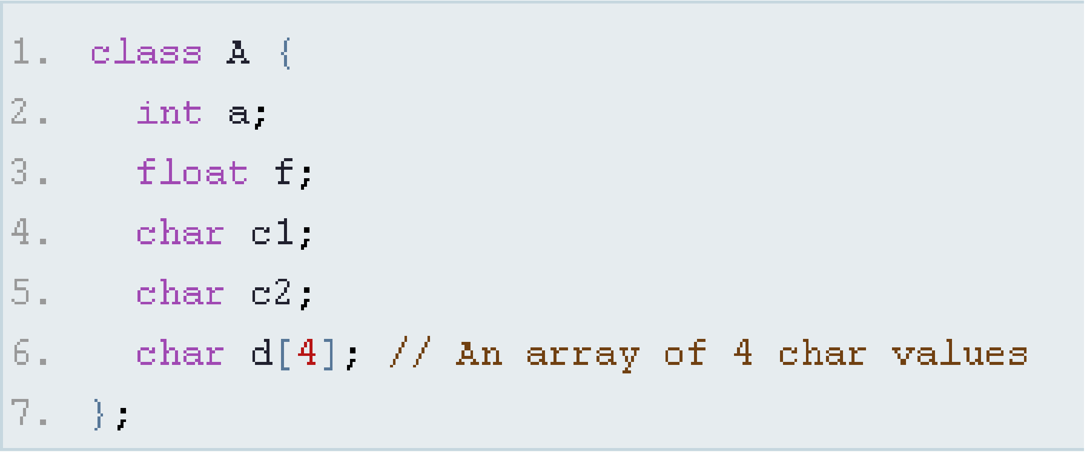
Bitfields should be avoided



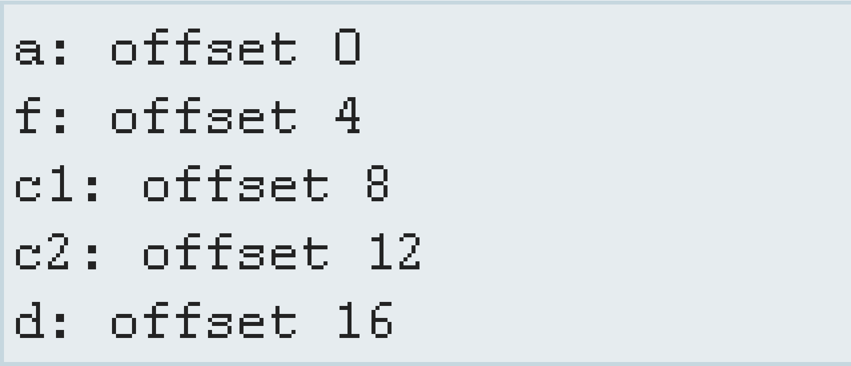
```
#include <stdio.h>
int main(int argc,char* argv[]){
  typedef struct test1{
    char a:2;
    long b:3;
    char c:2;
    short d:1;
    long long int e:3;
}test1;
typedef struct test2{
    char a:2;
    long b:3;
    char c:2;
    short d:1;
    //long long int e:3;
}test2;
  printf("test1:%d test2:%d\n",sizeof(test1),sizeof(test2));
  return 0;
}
```

result:

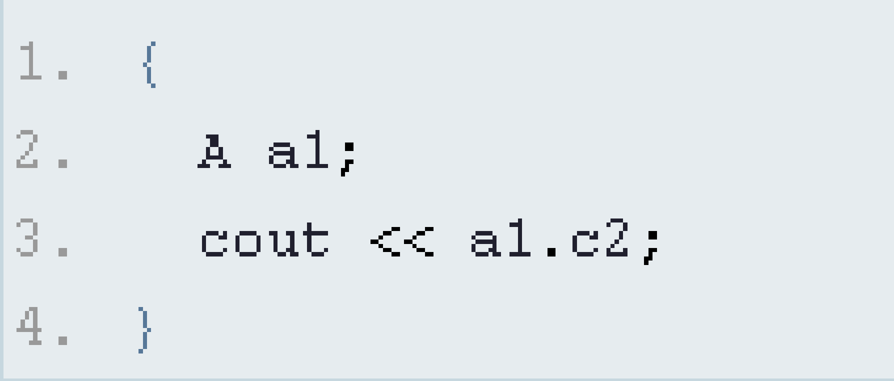
```
t1:4 test2:4
```



```
class A {
  int a;
  float f;
  char c1;
  char c2;
  char d[4]; // An array of 4 char values
};
```



```
a: offset 0
f: offset 4
c1: offset 8
c2: offset 12
d: offset 16
```



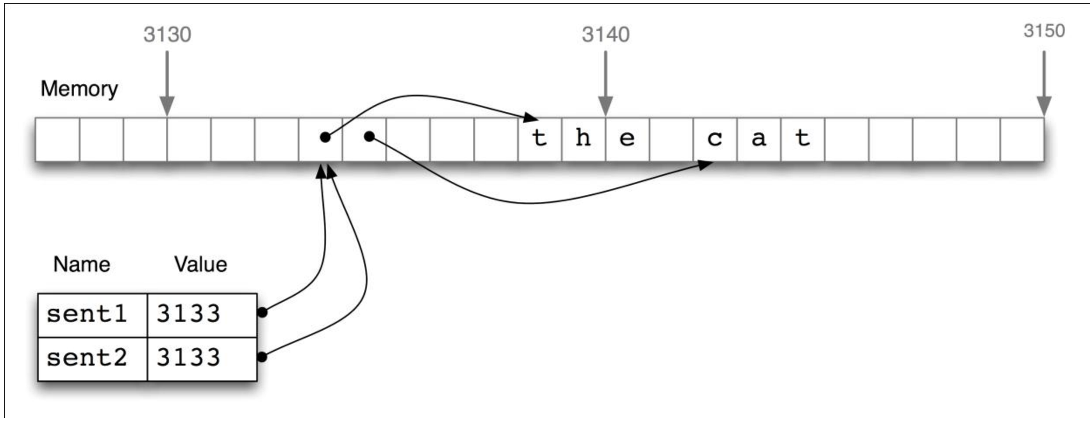
```
{
   A a1;
   cout << a1.c2;
}
```

- reference to a field - compiler uses offset of field within the object
- a1.c2 is translated to:
  - push activation frame for new block with one variable of 20 bytes (for a1)
  - invoke constructor of A on the address of register S (top of stack)
  - READ [S]+12, B - address [S]+12 into register B

Memory Layout of Arrays

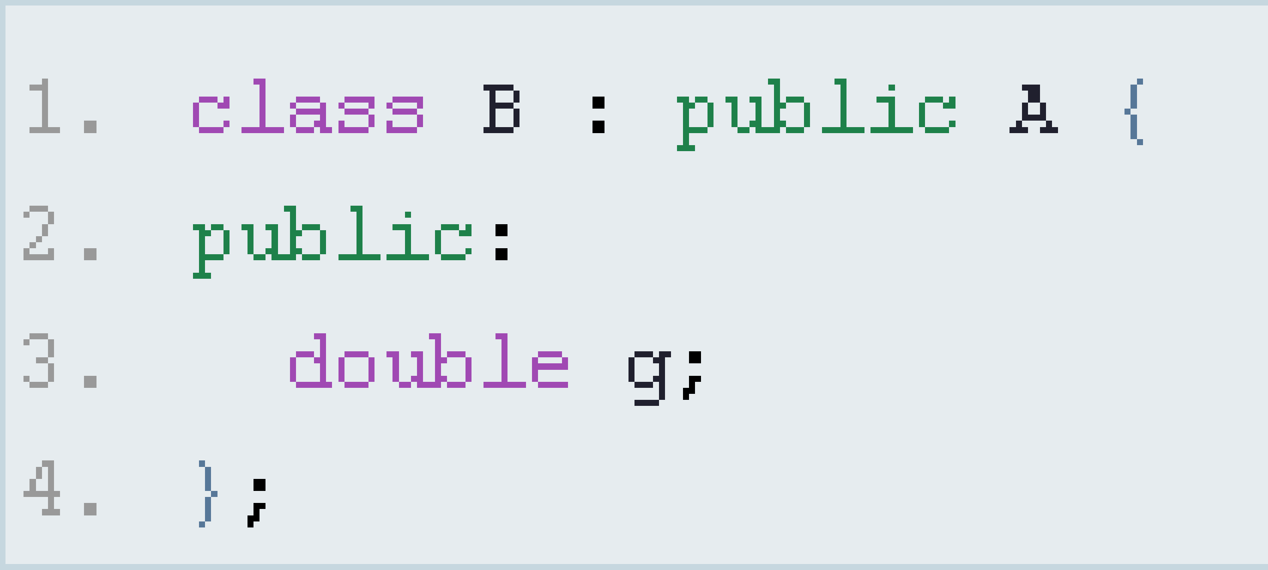
- field is aligned on a word boundary
- arrays are generally "packed": elements of array are *one after the other*
  - no holes

char *str = “the cat”;



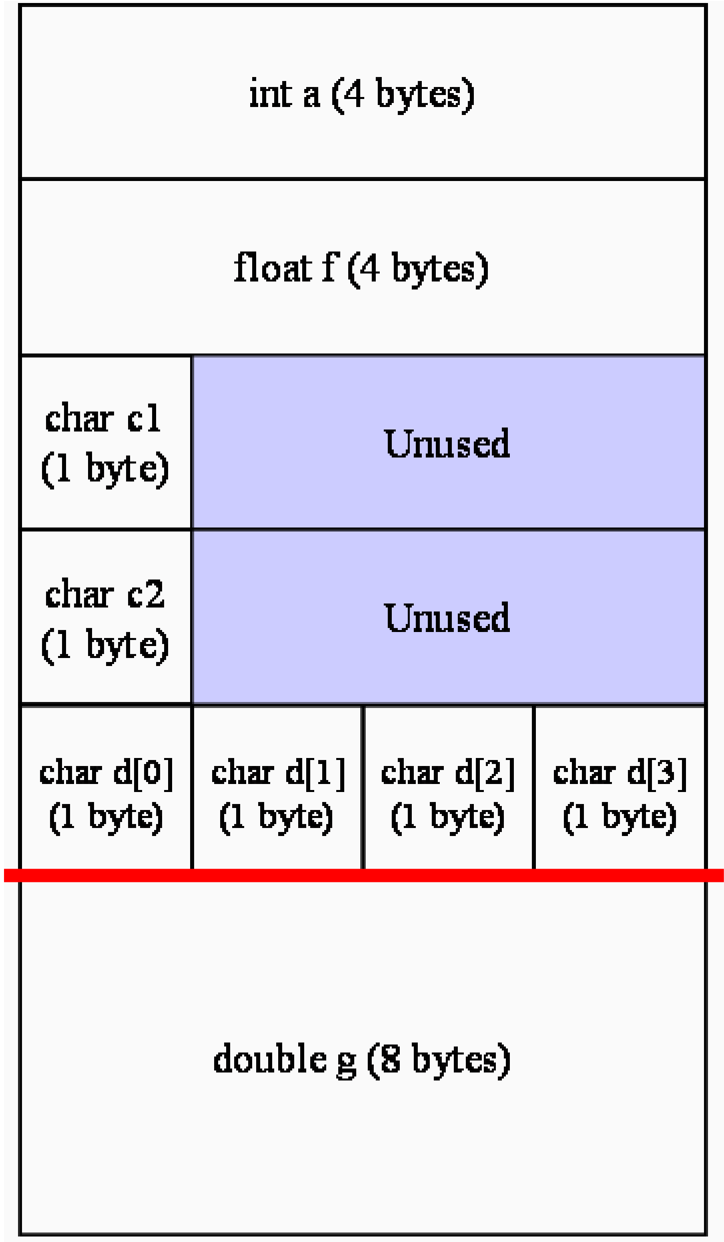
Memory Layout and Inheritance

- class B extends class A
  - fields defined in A exist in B
  - new fields for objects of type B.
- block memory for objects of class B is larger than that of objects of class A.



```
class B : public A {
public:
    double g;
};
```

- first 20 bytes = structure of type A.
- "look at a B value" ***as if*** an "A value":
  - take first part of B and "cut" to sizeof(A).



C++ and memory

- code for the methods of a class is stored only once for each class
- picture of the memory allocated to a process covers 3 distinct areas:
  - **heap**: values allocated using the new operator
  - **stack**: automatic values in activation frames
  - **code segment**: code of all the classes used in the program executed by the process

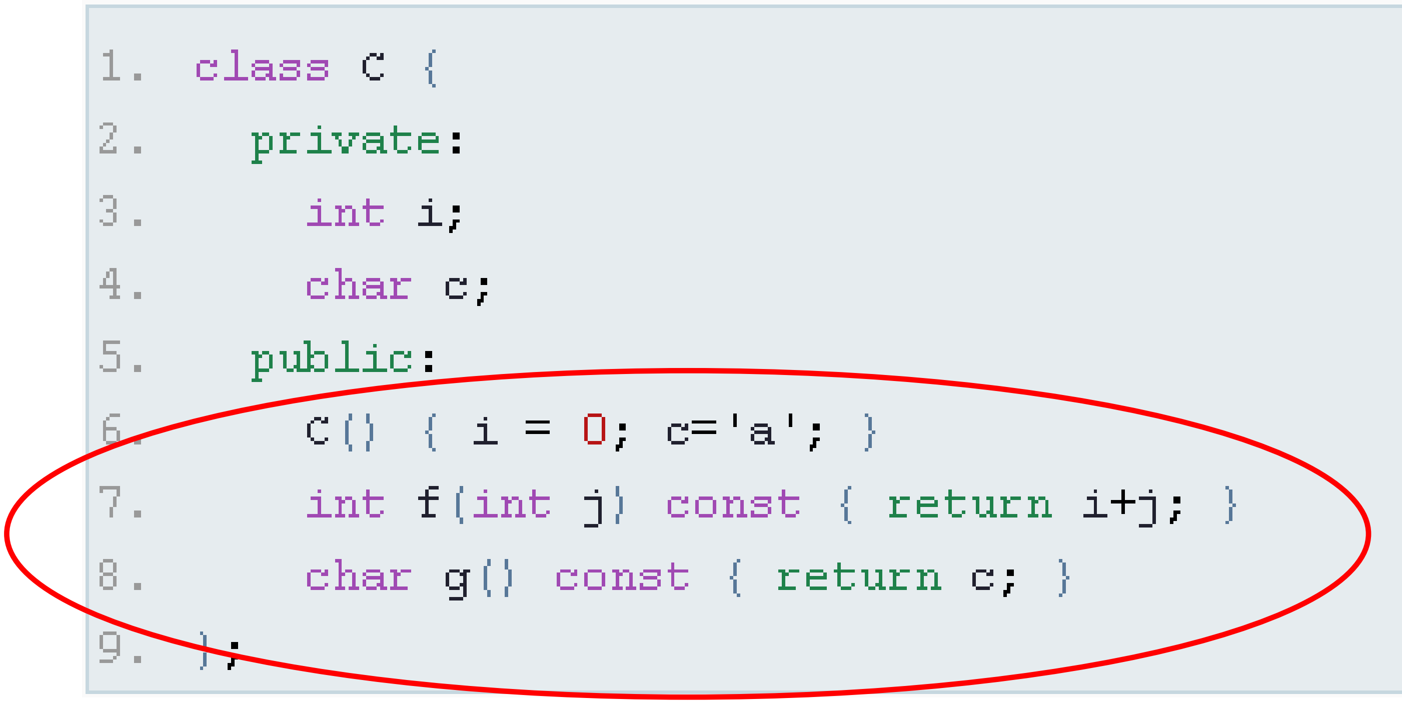
abstract objects & memory model

- an abstract object is characterized by the following elements:
  - identity
  - state
  - set of objects it knows
  - interface (set of messages to which the object can react)

abstract objects & memory model

- **identity** - address of object data in memory
- **state of object** - encoded in associated memory block (fields values)
- **interface of object** - known by the compiler, based on type of object
  - methods for objects to react (defined by class)

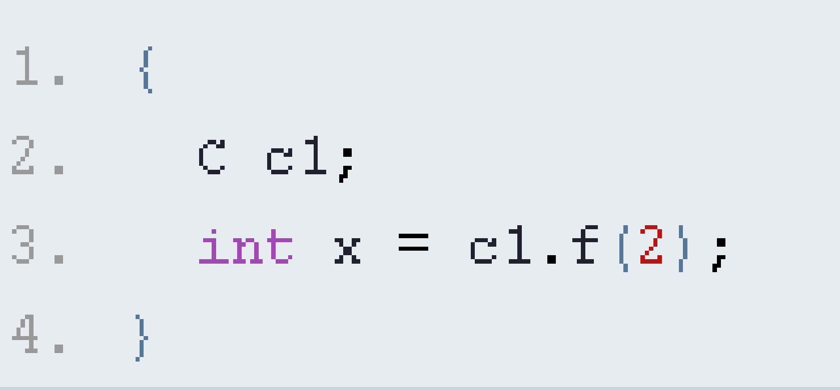
- interface: C::C(), C::~C(), C::f(), C::g()



```
class C {
  private:
    int i;
    char c;
  public:
    C() { i = 0; c='a'; }
    int f(int j) const { return i+j; }
    char g() const { return c; }
};
```

code region

- method is stored in **code region** allocated to process in which the class is used
- method is encoded as **sequence** of processor instructions
- method is known to compiler by **start address** in memory
- invocation of method = sequence of instructions:
  - parameters pushed on stack
  - method invoked by using CALL instruction of the processor
  - passed the address of the first instruction of the method that is invoked.



```
{
   C c1;
   int x = c1.f(2);
}
```

- push new activation frame on stack - c1= 8B + x=4B
- invoke C::C() on the address [S]
- push [S] -- push the address of c1 on the stack
- push $2 -- push the constant 2 on the stack
- push [S] -- push the address of c1 on the stack
- call [C::f] -- invoke c1.f(2)
- write ReturnRegister [S-4] -- copy the value returned by f into variable x which is below c1 in the stack
- pop [S] – pop the address of c1 from stack
- call [C::~C] -- invoke the destructor of c1

- compiler maintains internal table where it keeps track of the address of each of the methods of the class
- compiler invokes a method of a class
- method has access to internal state of object, wherever it may be. How?

implicit "this" parameter

- How method knows where are fields of object?
- Solution: compiler always passes a "hidden" parameter to method call: **address of the object-this**
- this of type *C** (for class *C*): address of block organized according to structure of class *C*.

Static method

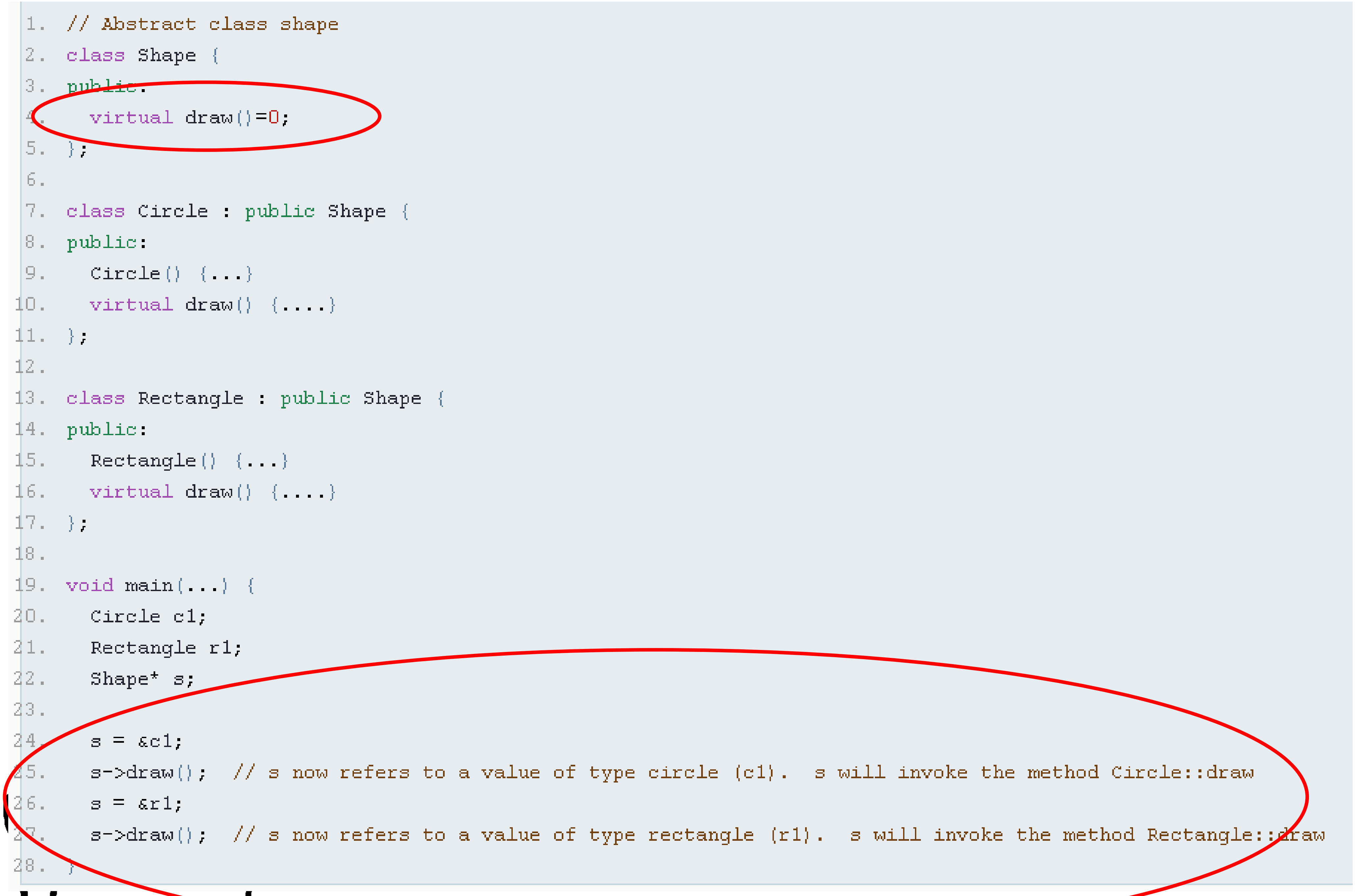
- static methods do not have access to this - can be invoked independently
  - C::static_method(x)

Polymorphism

- ability to use an operator or function in different ways.
- Different meanings to the **operators** or **functions** (poly = many / morph = shape)
  - 6+5
  - “a”+”bc”
  - 3.2+4.75

Late Binding

- Polymorphism = essential property of OO languages
- Refers to the possibility to decide which method to invoke at ***runtime*** and not at *compile* time.



```
// Abstract class shape
class Shape {
public:
  virtual draw()=0;
};

class Circle : public Shape {
public:
  Circle() {...}
  virtual draw() {....}
};

class Rectangle : public Shape {
public:
  Rectangle() {...}
  virtual draw() {....}
};

void main(...) {
  Circle c1;
  Rectangle r1;
  Shape* s;

  s = &c1;
  s->draw();  // s now refers to a value of type circle (c1).  s will invoke the method Circle::draw
  s = &r1;
  s->draw();  // s now refers to a value of type rectangle (r1).  s will invoke the method Rectangle::draw
}
```

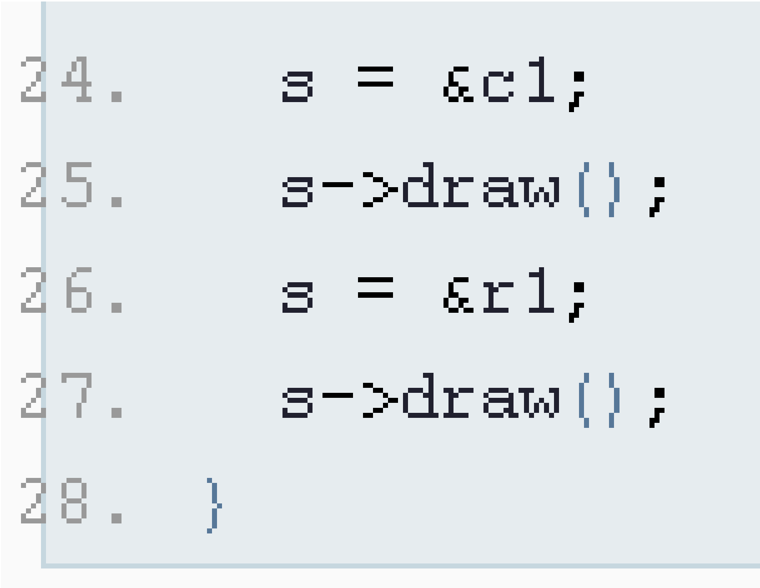
s->draw()

- compiler does not know the address of the function to invoke
- same C++ instruction will sometimes execute:
  - "call [Circle::draw]"
  - "call [Rectangle::draw]"
- How does the compiler manage to produce the right code?

Vtable Mechanism

- compiler ***delaying*** the decision of method to invoke to ***runtime***, instead of compile time.
- method is marked as virtual
- actual method invoked depends on the type of the value of the object at runtime
  - not on the type of the value at compile time

- s is a variable of type Shape
- invoke s->draw():
  - Call draw() of Rectangle or Circle by value to which s is ***bound*** at time of invocation



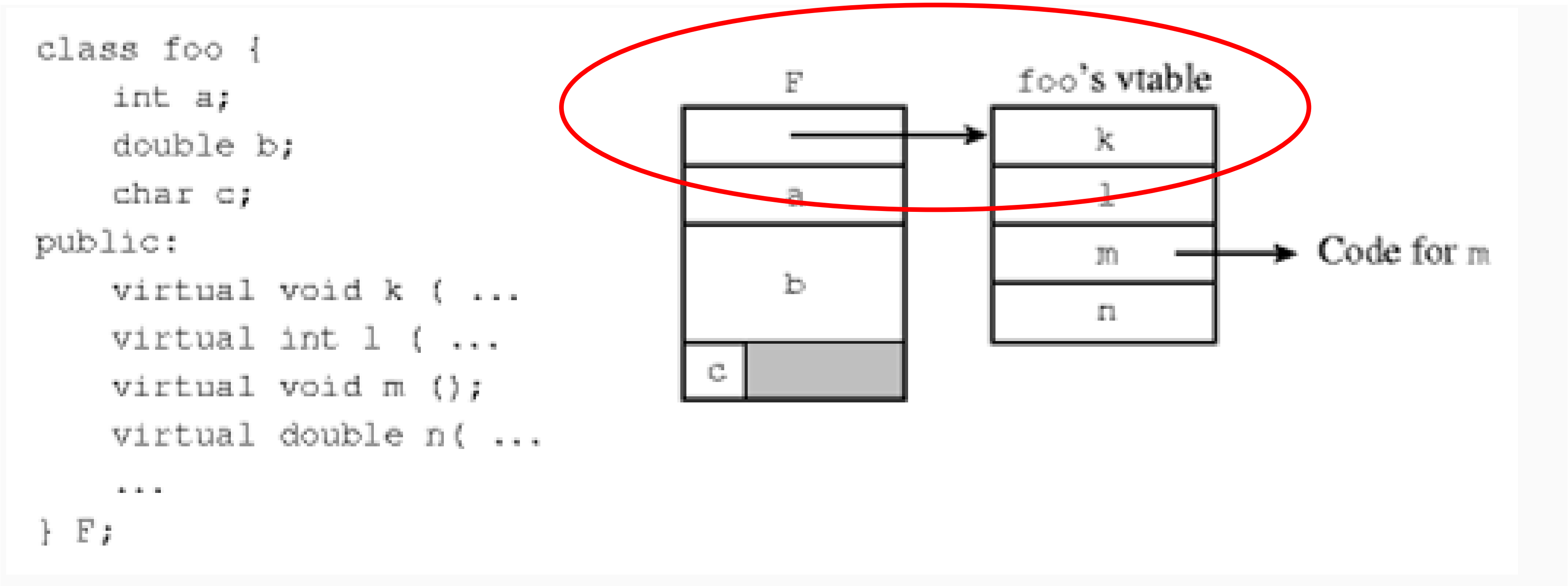
```
    s = &c1;
    s->draw();
    s = &r1;
    s->draw();
}
```

virtual-table (vtable)

how an object decides which code to invoke when it receives a message?

- **message** = invocation of a method through a pointer to an object.
- value of object in memory is **extended** by a pointer to a table with ***function address***
- table is stored explicitly in process memory (code region).
- table for each class that contains virtual methods.

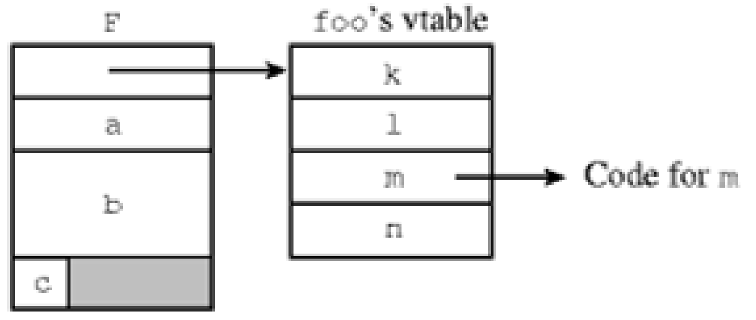
vtable for class foo



```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```

Invoking a virtual method

- Suppose d is of type foo *.
- call to object reference: d->m():
  - dereferencing d's vpointer,
  - looking up the m entry in the vtable,
  - dereferencing that pointer to call the correct method code.

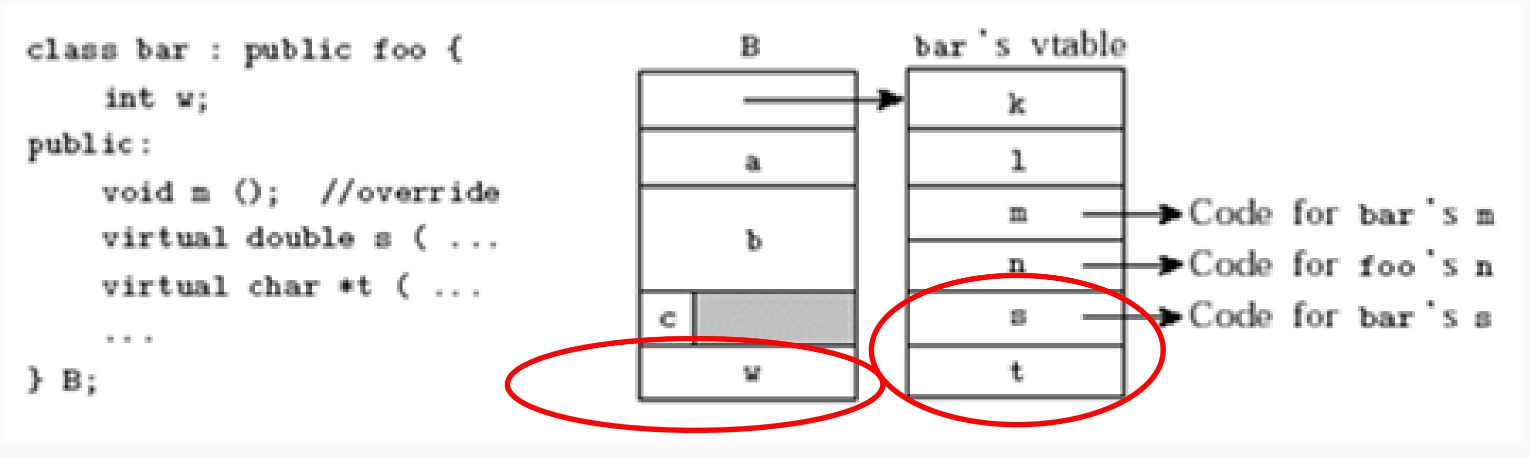


Example: *((*d)[2])(d);

- Assume vpointer is always the first element in d:
  - d is the address of the beginning of the block of memory which stores the foo value bound to d
  - *d is the content of the first word in the block of memory: it contains the address of the vtable
  - (*d)[2] is the address of the 3rd element in the vtable (the address of method m)
  - *((*d)[2])(d) - invoke function located at third slot in vtable of foo and pass to it the address of the value

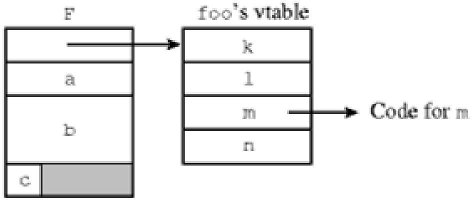
Inheritance and vtable

- When a class extends another class, how is the vtable managed?



```
class bar : public foo {
     int w;
public:
     void m ();   //override
     virtual double s ( ...
     virtual char *t ( ...
     ...
} B;
```

- bar extends foo. bar overrides m, and introduces 2 new methods s and t.



- compiler generates a new distinct vtable for class bar. vtable elements point:
  - to same addresses as parent when method is not overridden
  - overridden methods or to the new methods otherwise
- vtable of inherited class is an extension of the vtable of the parent table:
  - shared methods appear in the same order
  - new methods in the child class appear at the *end* of the vtable.

Vtable and Multiple Inheritance

- multiple inheritance: a class can extend more than one base class

Multiple Inheritance

- Class student inherits both from class person and from class gp_list_node
- vtable layout becomes more complex in such a situation.

Multiple Inheritance

- object of type student has in its state 3 types of fields (inherited from person, gp_list_node) and declared in class student

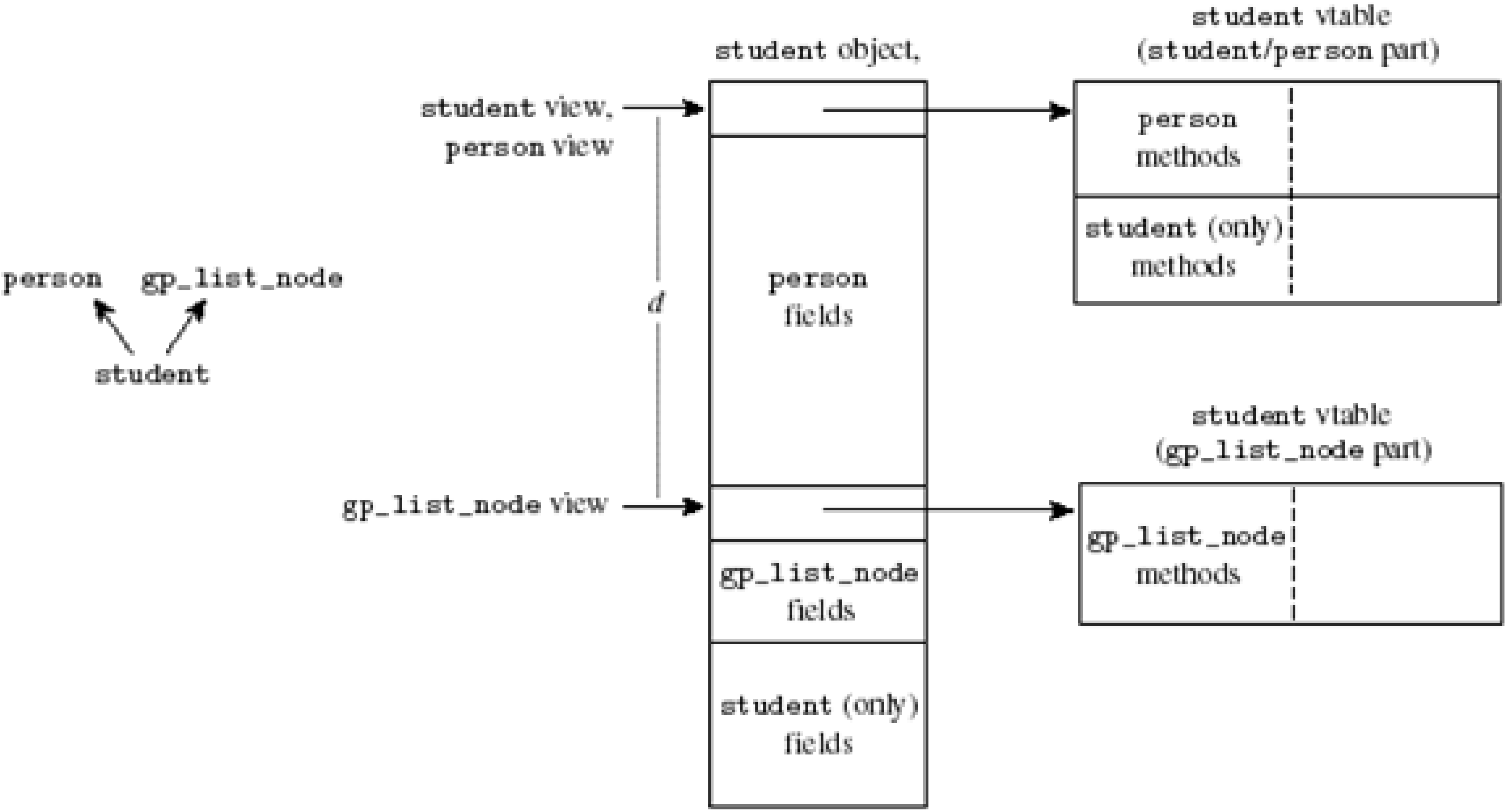
- 3 types of methods (inherited from person, gp_list_node) and defined in class student

Vtable

- vtable points to student specific vtable
- vtable first contains person methods, next methods that appear in class student
- vtable is then followed by the person fields. (look at a student value as a person value - just ignore the bottom part of the block)

we cannot store the gp_list_node vtable at the beginning of the block.

- So where can we store gp_list_node vtable?
- Store fields right after the person fields.
- Store gp_list_node data members after this vtable
- Finally we store the student specific data members at the end of the data block



person gp_list_node

student

student view,
person view

gp_list_node view

d

student object,

person
fields

gp_list_node
fields

student (only)
fields

student vtable
(student/person part)

person
methods

student (only)
methods

student vtable
(gp_list_node part)

gp_list_node
methods

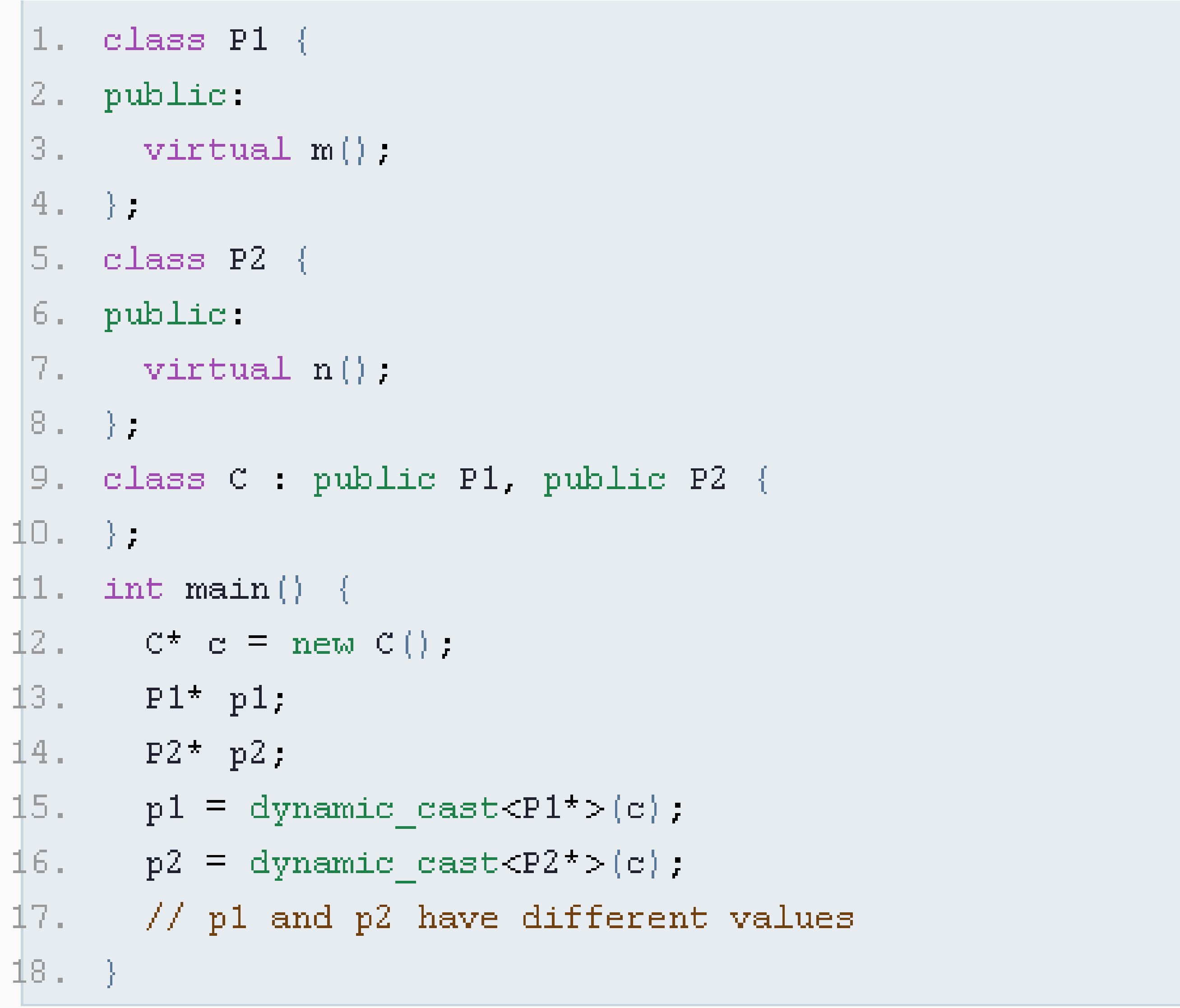
- how does the compiler find the appropriate vtable?
- how to pass valid *this* pointer to a method of gp_list_node that is not overridden?
    - code cannot know that what it receives as a this pointer is not a real gp_list_node.
    - accesses the fields of the value it has assuming it is a gp_list_node value.

pointer fixup

- compiler knows what type of method is invoked - either inherited from person, gp_list_node or specific to student.
- If inherited from gp_list_node: pass "corrected address" (this+d, d=sizeof(person)).
- ***look down*** from this address, block memory looks as a valid gp_list_node value
- vtable to which we point is also a valid gp_list_node vtable

Casting and Addresses

- when we cast an object to a different class, we may end up with a different address in memory
- casting is not only to tell the compiler "trust me I know what I do"; it also can end up generating code to fix the pointers in memory.



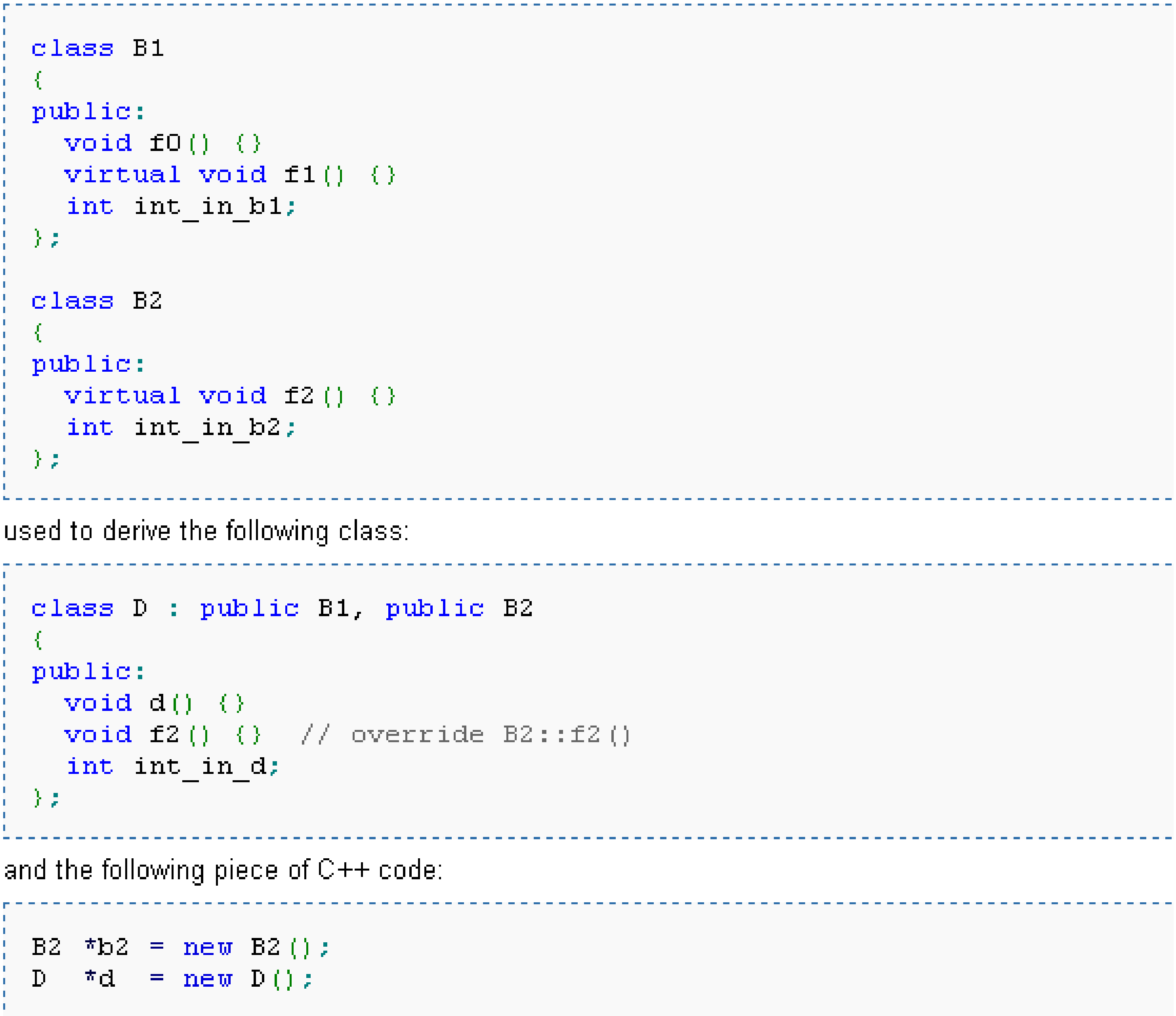
```
class P1 {
public:
  virtual m();
};
class P2 {
public:
  virtual n();
};
class C : public P1, public P2 {
};
int main() {
  C* c = new C();
  P1* p1;
  P2* p2;
  p1 = dynamic_cast<P1*>(c);
  p2 = dynamic_cast<P2*>(c);
  // p1 and p2 have different values
}
```

- **Implicit conversion:**
  - short a=2000; int b; b=a;
  - automatically performed when a value is copied to a compatible type
- **Explicit conversion**
  - short a=2000; int b; b = (int) a;
  - explicit type-casting allows to convert any pointer into any other pointer type

- **Dynamic_cast:**
  - can be used only with pointers and references
  - ensure that the result of the type conversion is a valid complete object
  - always successful when we cast a class to one of its base classes
  - **Compatibility:** dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types

g++ fdump -class-hierarchy option

- look at the exact structure of the vtables the compiler generates for us
- g++ has an option that allows us to get this information in a readable manner:
  - g++ c.cpp -fdump-class-hierarchy -o c
  - generates a text file called c.cpp.t01.class which
  - gives the details of the memory layout and vtable of the classes defined in the file.



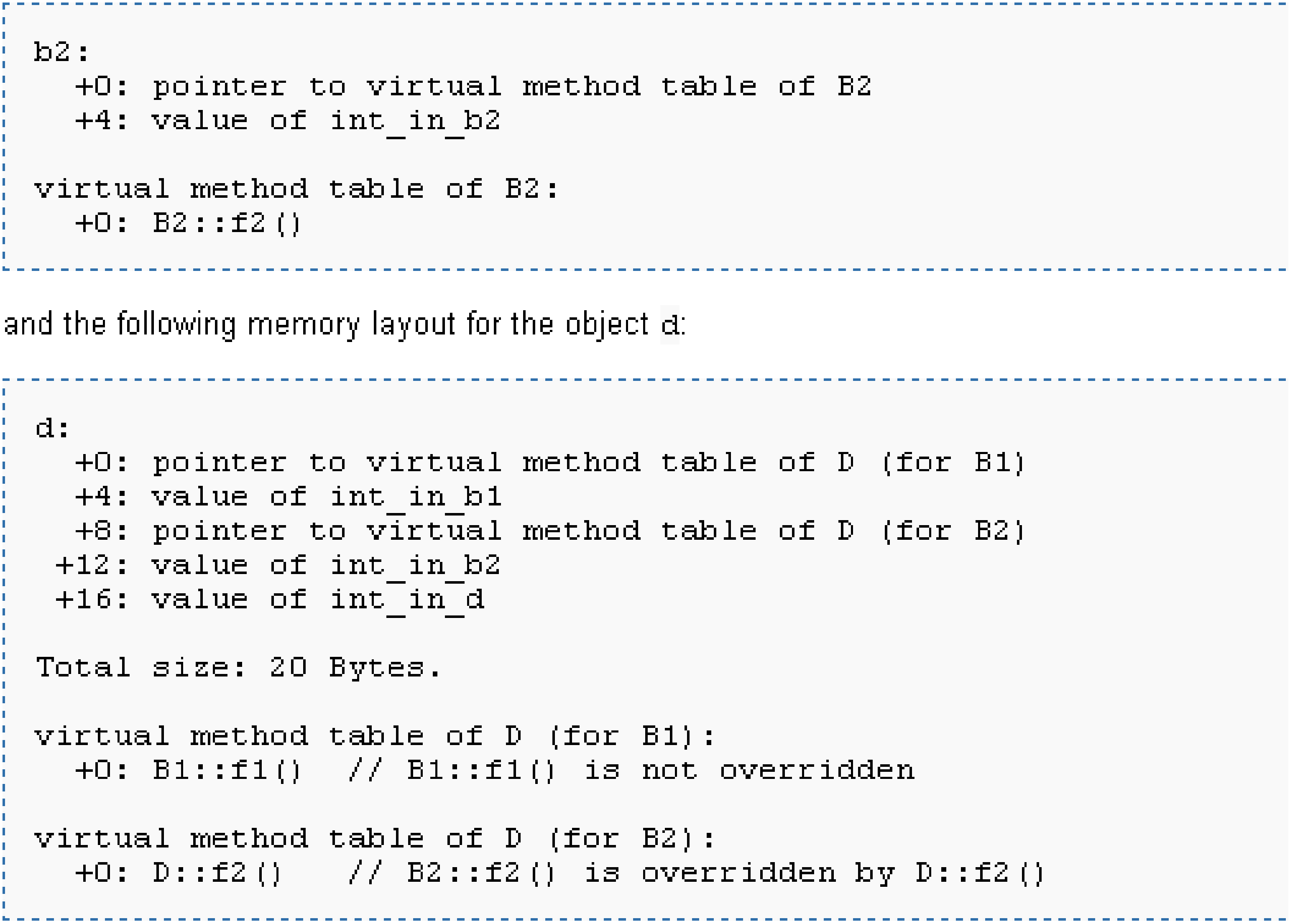
```
class B1
{
public:
  void f0() {}
  virtual void f1() {}
  int int_in_b1;
};

class B2
{
public:
  virtual void f2() {}
  int int_in_b2;
};
```

used to derive the following class:

```
class D : public B1, public B2
{
public:
  void d() {}
  void f2() {}  // override B2::f2()
  int int_in_d;
};
```

and the following piece of C++ code:

```
B2 *b2 = new B2();
D  *d  = new D();
```



```
b2:
  +0: pointer to virtual method table of B2
  +4: value of int_in_b2

virtual method table of B2:
  +0: B2::f2()
```

and the following memory layout for the object `d`:

```
d:
  +0: pointer to virtual method table of D (for B1)
  +4: value of int_in_b1
  +8: pointer to virtual method table of D (for B2)
 +12: value of int_in_b2
 +16: value of int_in_d

Total size: 20 Bytes.

virtual method table of D (for B1):
  +0: B1::f1()  // B1::f1() is not overridden

virtual method table of D (for B2):
  +0: D::f2()   // B2::f2() is overridden by D::f2()
```

Virtual methods: performance issues

- invoking a virtual method is more expensive at runtime than invoking a regular function.
    - 2 operations: get the location of the function's code from vtable, and invoke the function.
- 2 other costs to the vtable mechanism:
    1. Object values are extended by one word for each vtable to which they refer.
    2. Virtual methods cannot be compiled "inline"
    - (Inline: avoids calling a function - pushing arguments on the stack, popping them out at end - copying code of the function at invocation)

- 3 costs combined can have a strong effect on performance of program.
- in C++, methods are not virtual by default. If a class does not have virtual method, then does not include a vtable.
- in Java, methods are always virtual.

Implementing Interfaces in C++

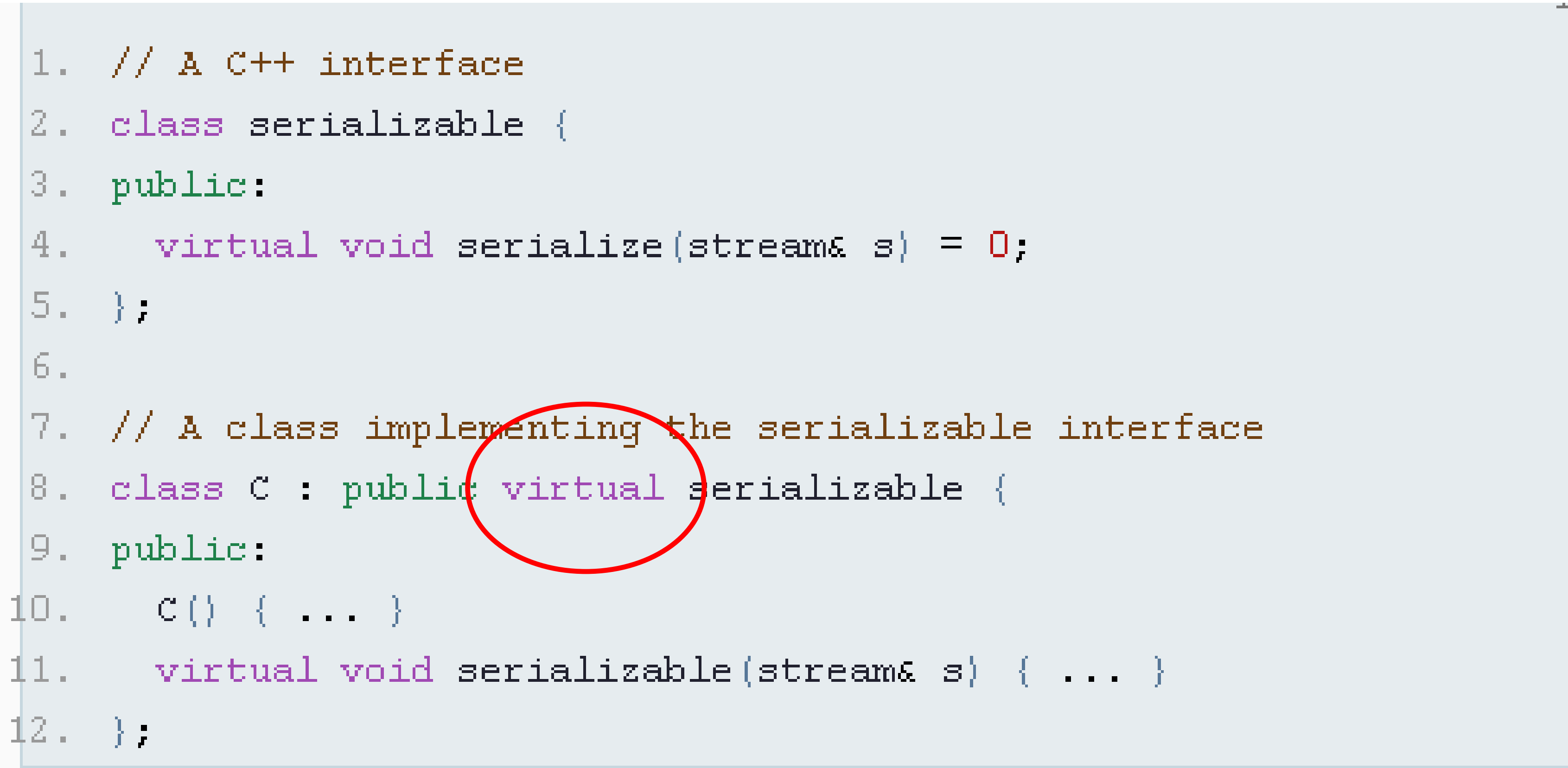
- Java avoids the complexity of multiple inheritance
  - restricts programmers to single inheritance and the mechanism of interfaces.
- Interfaces = restricted method of multiple inheritance.
- interfaces in C++: pure virtual abstract class.

pure virtual abstract class

- does not define any data members.
- All of its methods are virtual.
- All of its methods are abstract (marked in C++ as virtual m() = 0;)

"virtual inheritance"= "implement an interface"

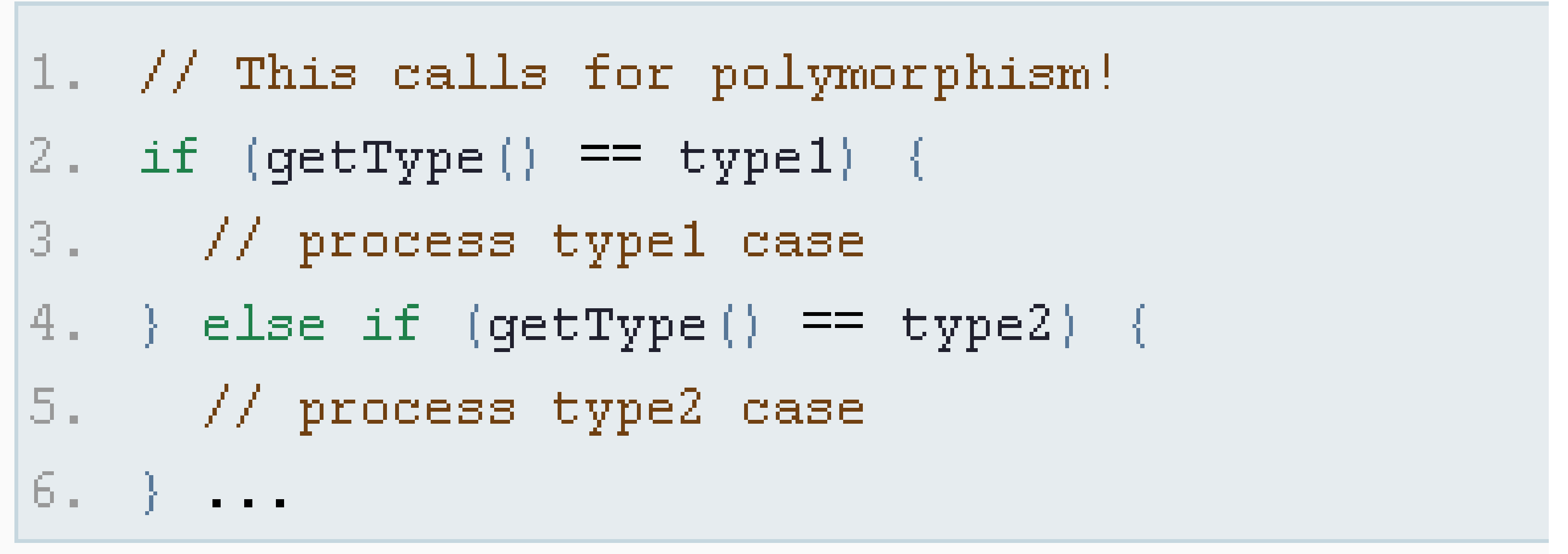
- avoid the problem of ambiguous hierarchy composition - diamond problem
- inheritance=arranging classes in memory



```
// A C++ interface
class serializable {
public:
  virtual void serialize(stream& s) = 0;
};

// A class implementing the serializable interface
class C : public virtual serializable {
public:
  C() { ... }
  virtual void serializable(stream& s) { ... }
};
```

The Visitor Pattern

- when you want to decide at runtime which piece of logic to execute=polymorphism
- In such cases in your code -refactor - introduce polymorphism

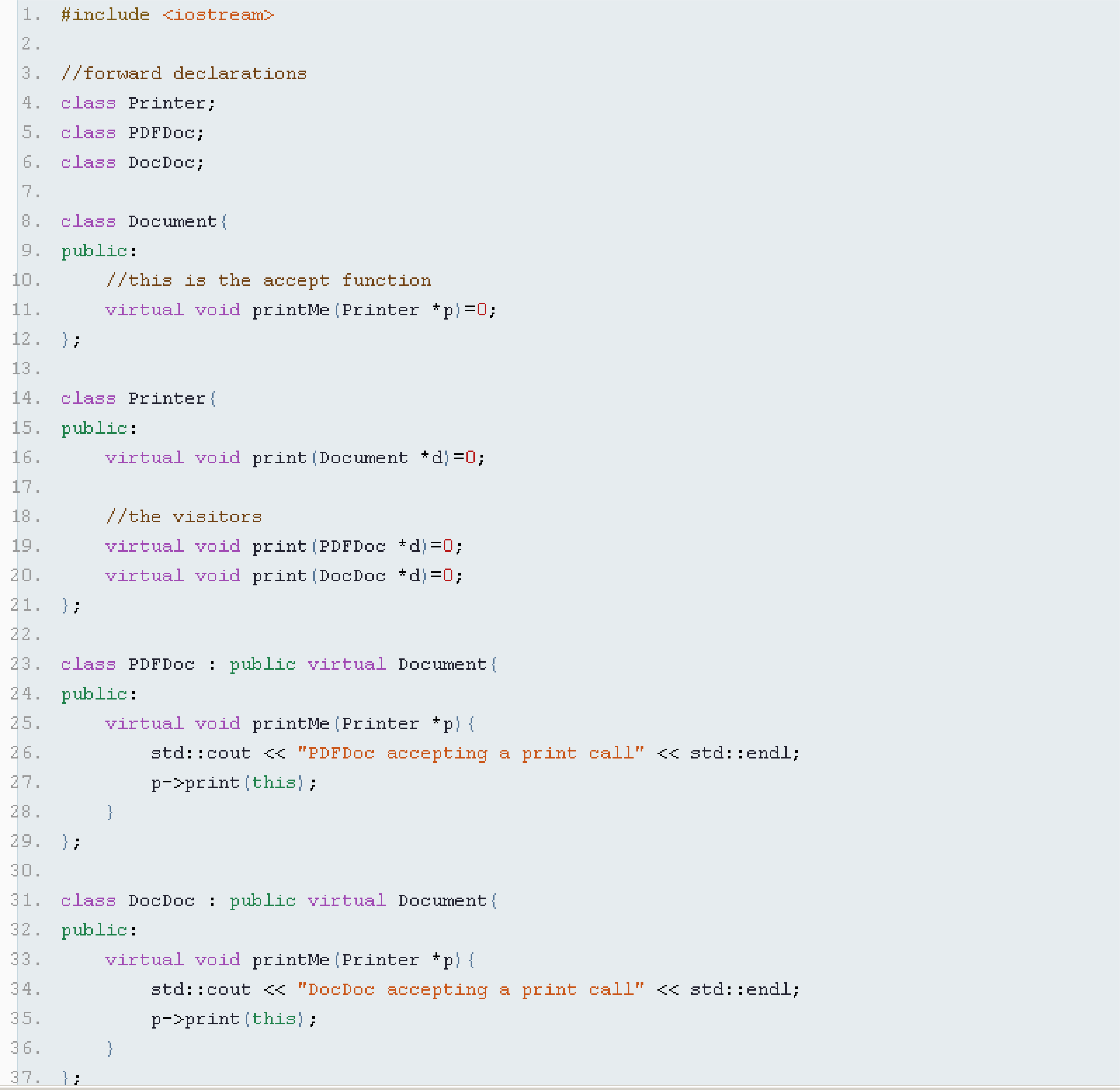


```
// This calls for polymorphism!
if (getType() == type1) {
  // process type1 case
} else if (getType() == type2) {
  // process type2 case
} ...
```

- Visitor pattern - achieve re-organization of your code
- a Printer object can print Document objects.
  - code to print documents types is different
  - code for each type of printer is different.
  - “double dispatch”

- use the virtual table dispatch mechanism to send the print message to the right method.
- a printer object receives the print message with a document object:
  - first dispatch happens when we select the appropriate printer object
  - second dispatch (based on the type of document) is achieved by sending the printMe message to the document.
  - e.

- do not want each document class to know about each printer class –very bad coupling.
- document object just invokes the specific printer method on a specific document type method.
- each document type for each printer type has a separate method handling the specific code



```cpp
#include <iostream>

//forward declarations
class Printer;
class PDFDoc;
class DocDoc;

class Document{
public:
    //this is the accept function
    virtual void printMe(Printer *p)=0;
};

class Printer{
public:
    virtual void print(Document *d)=0;

    //the visitors
    virtual void print(PDFDoc *d)=0;
    virtual void print(DocDoc *d)=0;
};

class PDFDoc : public virtual Document{
public:
    virtual void printMe(Printer *p){
        std::cout << "PDFDoc accepting a print call" << std::endl;
        p->print(this);
    }
};

class DocDoc : public virtual Document{
public:
    virtual void printMe(Printer *p){
        std::cout << "DocDoc accepting a print call" << std::endl;
        p->print(this);
    }
};
```

```cpp
class MyPrinter : public virtual Printer{
public:
    void badPrint(Document *d){
        if (dynamic_cast<PDFDoc*>(d)){
            print(dynamic_cast<PDFDoc*>(d));
        } else if (dynamic_cast<DocDoc*>(d)){
            print(dynamic_cast<DocDoc*>(d));
        } else {
            std::cout << "what to do???" << std::endl;
        }
    }

    virtual void print(Document *d){
        std::cout << "dispatching function <print> called" << std::endl;
        d->printMe(this);
    }
    virtual void print(PDFDoc *d){
        std::cout << "printing a PDF doc" << std::endl;
    }
    virtual void print(DocDoc *d){
        std::cout << "printing a Doc doc" << std::endl;
    }
};

int main(){
    MyPrinter p;
    Document *docA = new PDFDoc();
    Document *docB = new DocDoc();

    p.print(docA);
    p.print(docB);
    std::cout << "using badPrint" << std::endl;
    p.badPrint(docA);
    p.badPrint(docB);
    delete docA;
    delete docB;
    return 0;
```