ECE326

PROGRAMMING LANGUAGES

Lecture 9 : Inheritance in C++

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

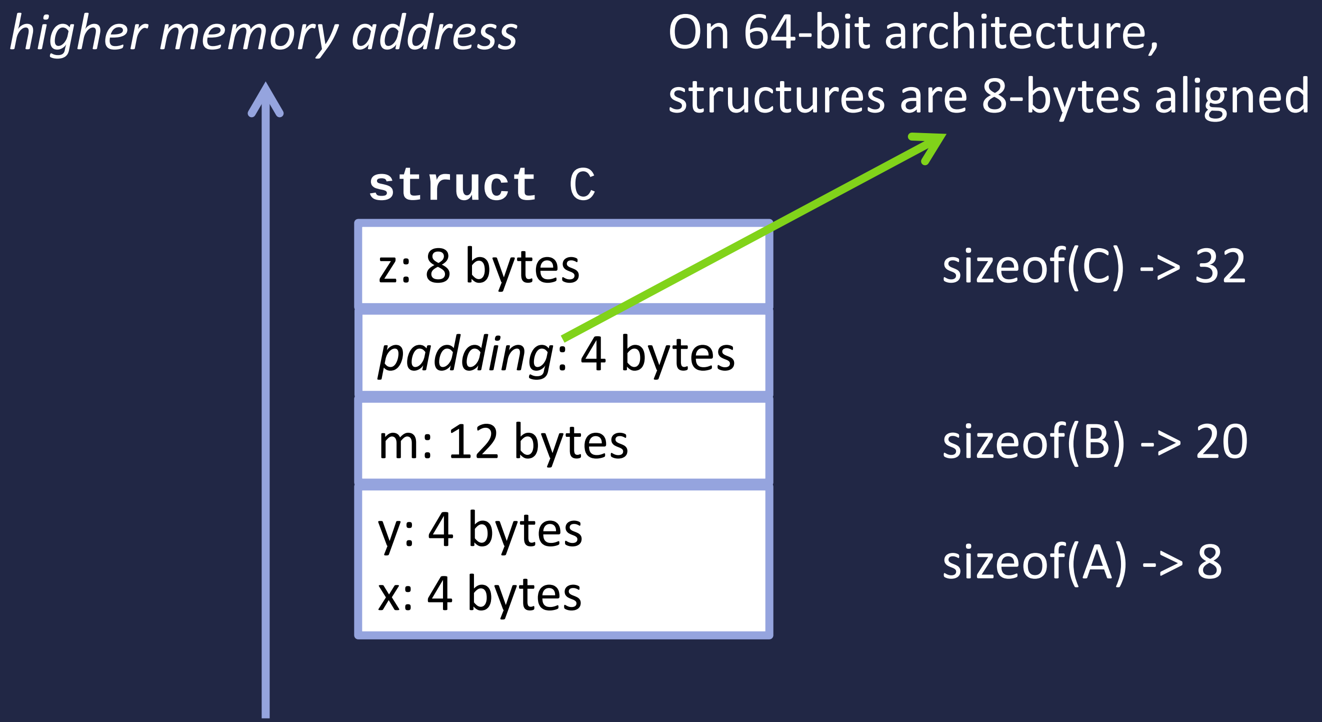
Single Inheritance

- All object-oriented languages supports it
- Derived class can only inherit from one base class
- Java *only* supports single inheritance
  - Simplifies compiler implementation

Object Layout

- Used by all compiled languages that support inheritance

```
struct A {
    int x;
    int y;
};

struct B : public A {
    char m[12];
};

struct C : public B {
    long z;
};
```



Alignment

- Some architectures have data alignment requirements
  - E.g. A 64-bit integer must be 8 bytes aligned
    - 0xFFFFCC00 is 8-byte aligned but 0xFFFFCC02 is not
  - Unaligned data requires extra instructions to re-align
- Padding
  - An unnamed structure member to align subsequent fields
  - Note: C/C++ does not allow reordering
    - Fields must be placed in order of appearance in structure definition
  - Disable padding
    - Add `__attribute__((packed))` to structure definition

Packed

```
struct Loose {
    int x;
    /* 4-bytes padding */
    long y;
    char z[10];
    /* 6 bytes padding */
    long w;
};

sizeof(Loose) -> 40
```

```
struct Tight {
    int x;
    long y;
    char z[10];
    long w;
} __attribute__((packed));

sizeof(Tight) -> 30
```

Abstract Base Class

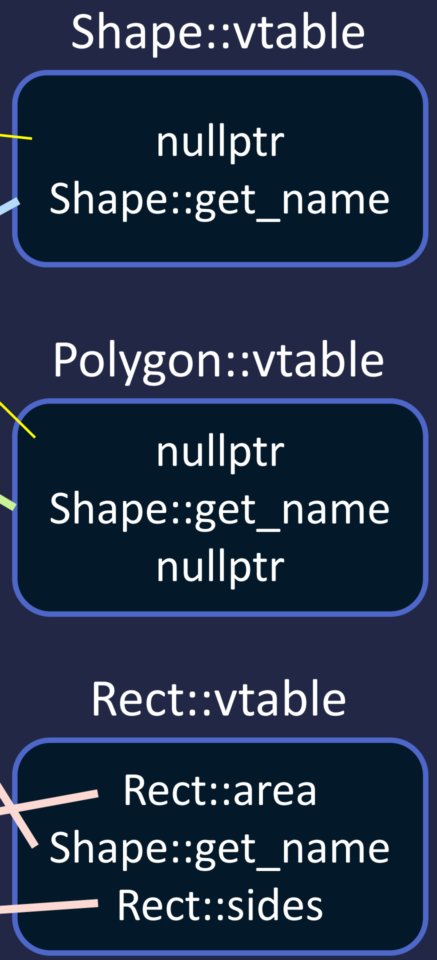
- Contains one or more *pure virtual function(s)*
- Pure virtual function
  - Declared, but not defined (implemented)
- Cannot be instantiated

```
struct Shape {
    /* pure virtual function */
    virtual double area() const=0;
    /* normal virtual function */
    virtual const char * get_name() const {
        return “Shape”;
    }
};
```

Virtual Table

Cannot have nullptr in vtable, ∴
cannot instantiate these classes

```
struct Shape {
    virtual double area() const=0;
    virtual const char * get_name() const;
;

struct Polygon : public Shape {
    virtual int sides() const=0;
};

struct Rect : public Polygon {
    double w, h;
    virtual double area() const { return w*h };
    virtual int sides() const { return 4; }
};
```



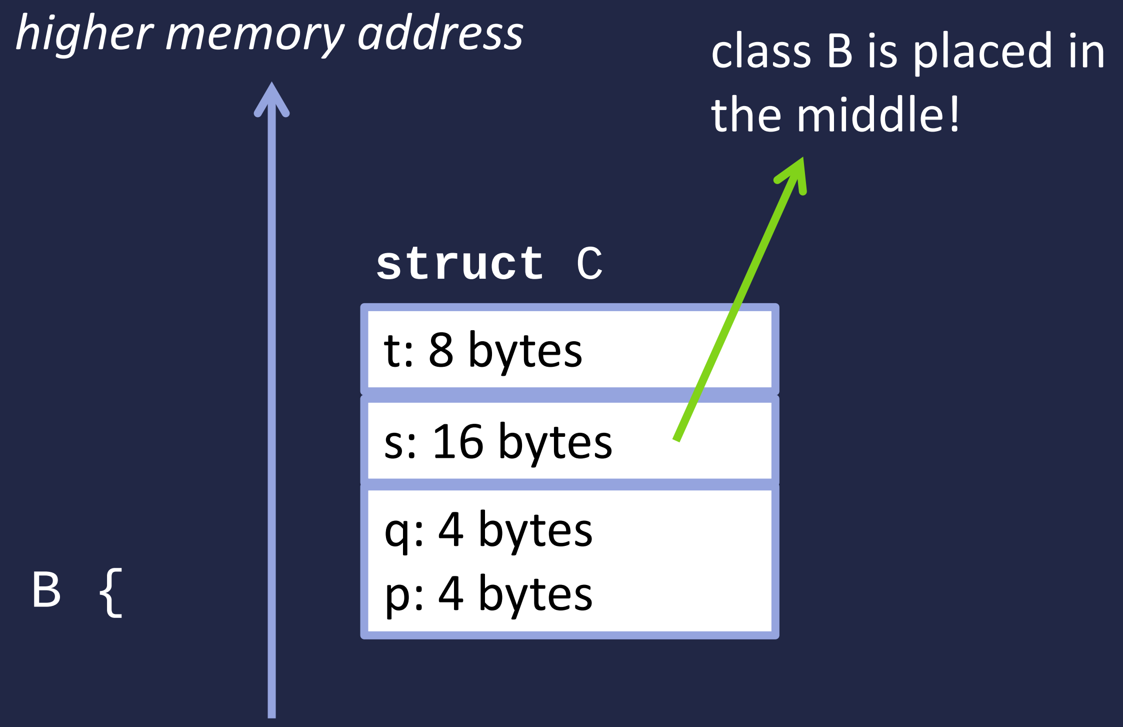
Multiple Inheritance

- Derived class has two or more base classes
- Use Cases:
  - Support for multiple interfaces
    - E.g. Amphibian class is both a Terrestrial and a Swimmer
  - Implementation inheritance
    - Base class inherited for its implementation (code reuse)
    - E.g. Actor class is a Person, and borrows implementation from Singer
- Introduces possibility of ambiguity
  - E.g. both Cowboy and Painter have a `draw` function
    - Special NPC character Joe is both a Cowboy and a Painter

Object Layout

- Base classes are stacked by order of appearance

```
struct A {
    int p;
    int q;
};

struct B {
    char s[16];
};

struct C : public A, public B {
    long t;
};
```



Resolving Ambiguity

- When accessing members of same name from different base classes, must specify which base class
  - Does *not* check function signature

```
struct Cowboy {
    void draw(Target *);
};

struct Painter {
    void draw(Canvas *);
};

struct Joe : public
 Cowboy, public Painter {
    …
};
```

```
Joe joe = Joe();

// error: request for member
// ‘draw’ is ambiguous
joe.draw(canvas);

// ok – base class specified
joe.Painter::draw(canvas);

// ok – base class specified
joe.Cowboy::draw(victim);
```

Upcasting

- Casting a more specific type to a more generic type
  - i.e. from a subclass to a super class

```
struct A {
    int x;
    int y;
};

struct B : public A {
    char m[16];
};

struct C : public B {
    long z;
};
```

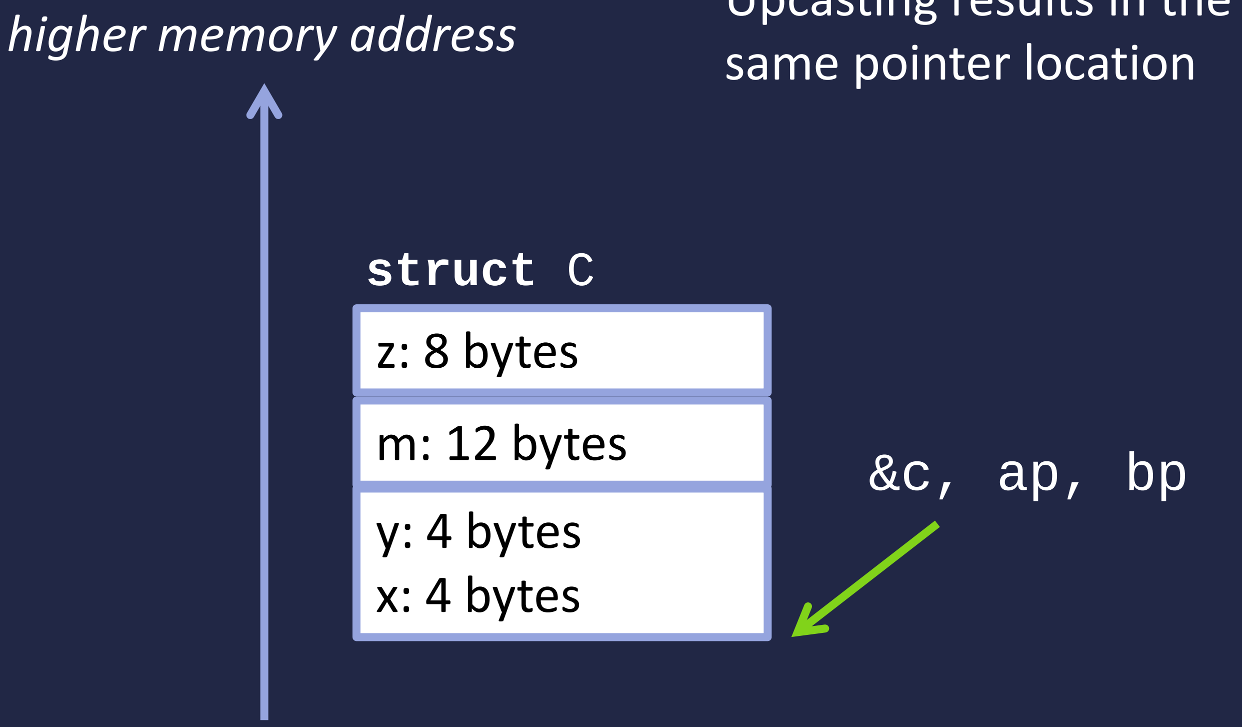
```
/* single inheritance */

  /* &c == 0xffffcbf0 */
  C c = C();

  /* ap == 0xffffcbf0 */
  A * ap = &c;

  /* bp == 0xffffcbf0 */
  B * bp = &c;
```

Upcasting results in the same pointer location

Upcasting

- Casting a more specific type to a more generic type
  - i.e. from a subclass to a super class

```
struct A {
    int x;
    int y;
};

struct B : public A {
    char m[16];
};

struct C : public B {
    long z;
};
```



Upcasting results in the same pointer location

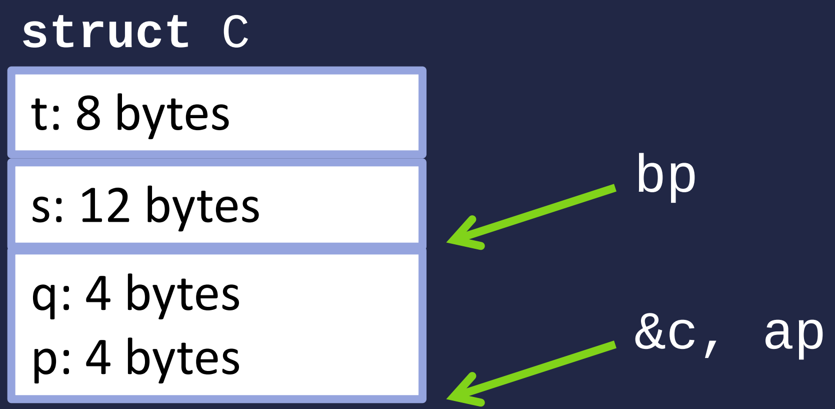
Upcasting

- For multiple inheritance, upcasting may require shifting of memory address

```
struct A {
    int p;
    int q;
};

struct B {
    char s[16];
};

struct C : public A, public B {
    long t;
};
```

Upcasting results in the *different* memory address

```
/* multiple inheritance */

  C c = C();
  A * ap = &c;
  B * bp = &c;
```



Downcasting

- Casting a generic type to a specific type
  - i.e. from a super class to a subclass
- Can be potentially dangerous (type unsafe) if coerced
  - Type coercion: forcefully cast one type to another
    - Requires special cast operator: e.g. `reinterpret_cast`
- Single inheritance
  - Safe as long as type is correct
- Multiple inheritance
  - Requires pointer offset

Runtime Type Information

- Exposes information about type of object at runtime
- Adds runtime overhead, can be turned off
- Allows for type-safe downcasting
- `dynamic_cast`
  - Attempts to cast, return nullptr if not type safe
  - Offsets pointer correctly for multiple inheritance

```
Penguin p = Penguin();    Animal * ap = &p;

// success, pp is a valid pointer
Penguin * pp = dynamic_cast<Penguin *>(ap);
// fail, tp is nullptr
Turkey * tp = dynamic_cast<Turkey *>(ap);
```

Repeated Base Class

- Appears more than once during inheritances

```
struct Person {
    const char * name;
};

struct Student :
    public Person {
    int student_id;
};
```

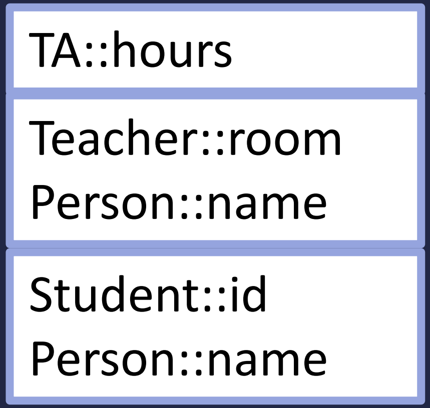
```
struct Teacher :
    public Person {
    int class_id;
};
```

```
struct TA : public Student, public Teacher {
    int hours;
};
```

Repeated Base Class

- By default, multiples copies of base class is made

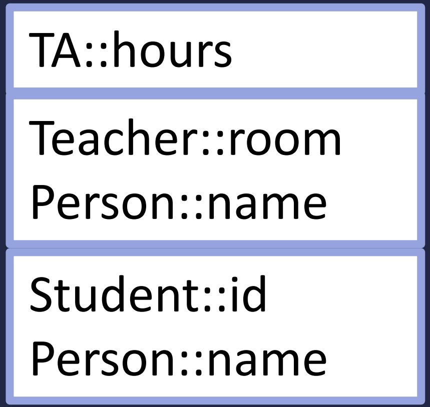
```
struct Person {
    const char * name;
};

struct Student : public Person {
    int id;
};

struct Teacher : public Person {
    int room;
};

struct TA : public Student, public Teacher {
    int hours;
};
```



Ambiguous Base

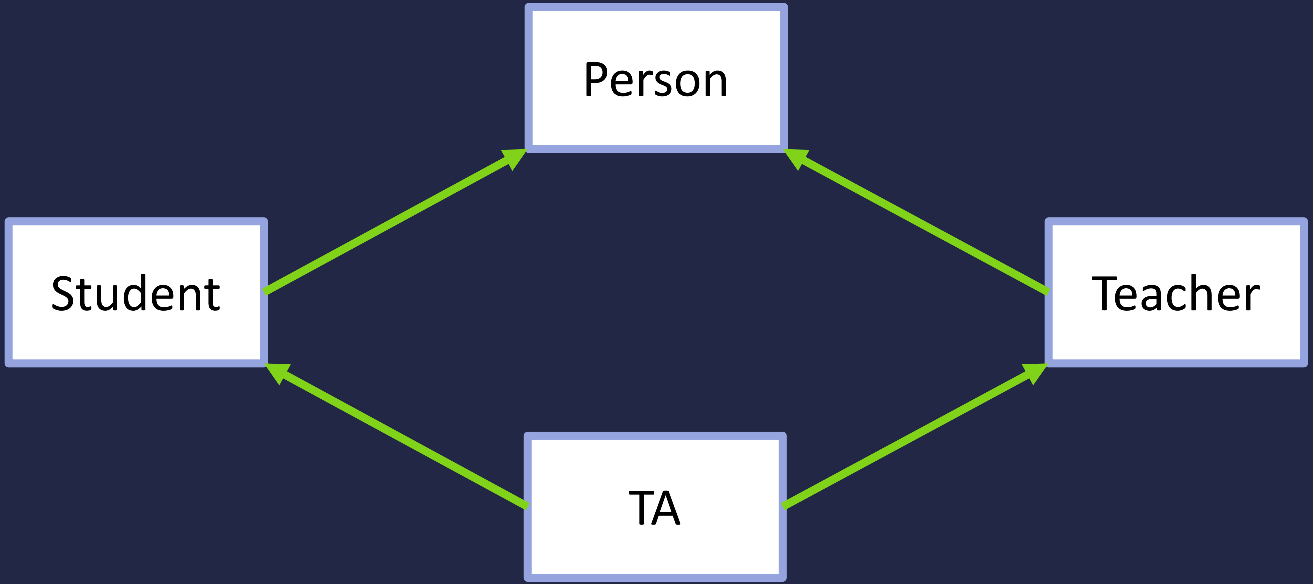
- Occurs when trying to access members of repeated base class – requires disambiguation

```
TA ta = TA();

// error: ‘Person’ is an ambiguous
// base of ‘TA’
Person * pp = &ta;

// following two lines are fine
Person * teacher = (Teacher *)&ta;
Person * student = (Student *)&ta;

teacher.name = “Jack”;  // Both teacher and student
student.name = “Bob”;   // have their own copy of name
```



Diamond Problem

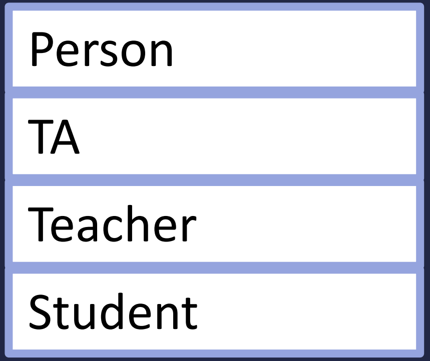
- When repeated base classes are undesirable
  - Each parent class has its own copy of common base class
    - Causes ambiguity, even after disambiguation!
- What we want is *shared* common base class



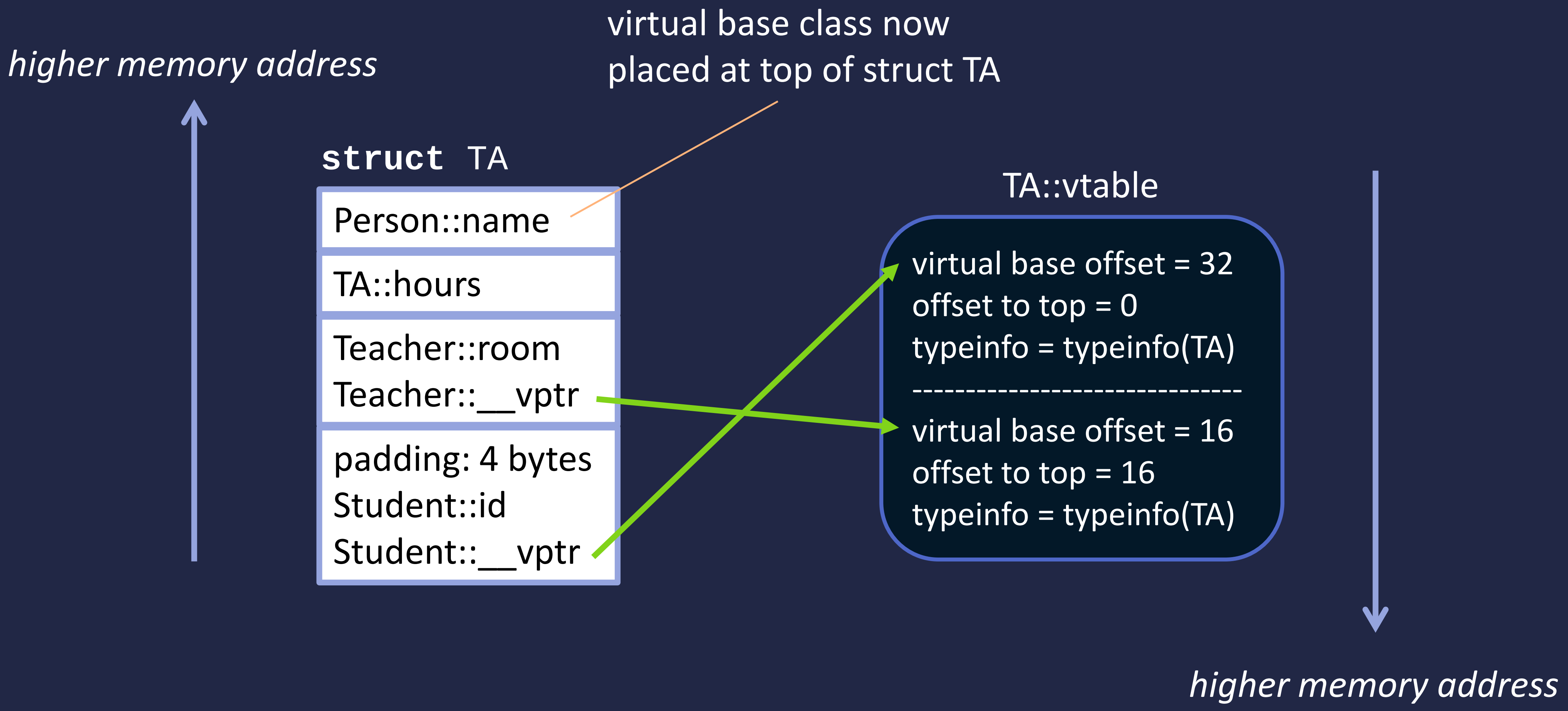
Virtual Base Class

- Allows common base class to be shared

```
struct Person {
    const char * name;
};
struct Student : virtual public Person {
    int id;
};
struct Teacher : virtual public Person {
    int room;
};
struct TA : public Student, public Teacher {
    int hours;
};
```



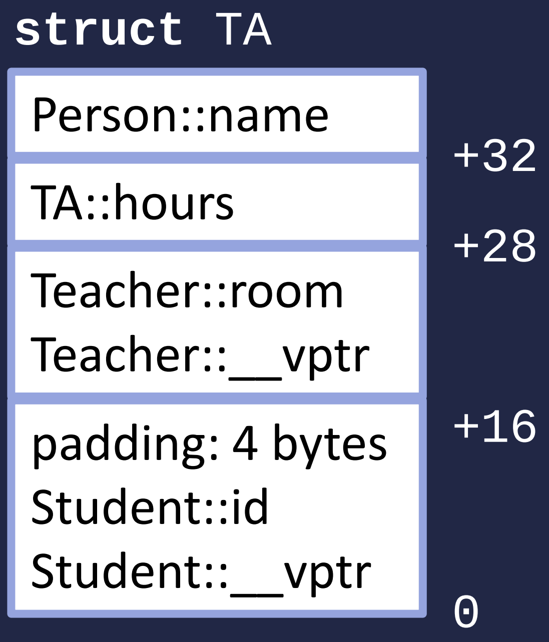
Object Layout



Upcasting

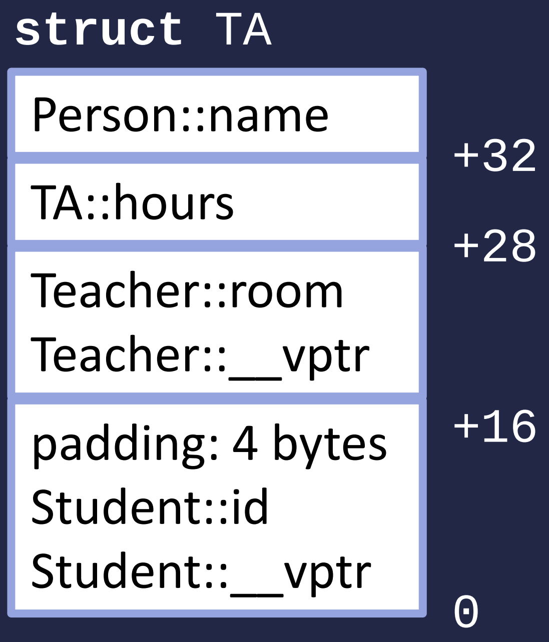
- To access virtual base class from one of the parent classes, consult virtual base offset in table

```
1. TA ta = TA();
2. Student * student = &ta;
3. student->name = “Jack”;

// locate student object (offset = 0)
2. student = &ta + vtable.top_offset;
// locate shared person object (offset = 32)
3. _person = student + vtable.vbase_offset;
// locate name field in person object
// (offset 0, first field)
   __name = _person + 0;
// set name field
   *__name = “Jack”;
```



Upcasting

```
1. TA ta = TA();
2. Teacher * teacher = &ta;
3. teacher->name = “Jack”;

// locate teacher object (offset = 16)
2. teacher = &ta + vtable.top_offset;
// locate shared person object (offset = 16)
3. _person = teacher + vtable.vbase_offset;
// locate name field in person object
// (offset 0, first field)
   __name = _person + 0;
// set name field
   *__name = “Jack”;
```



Downcasting

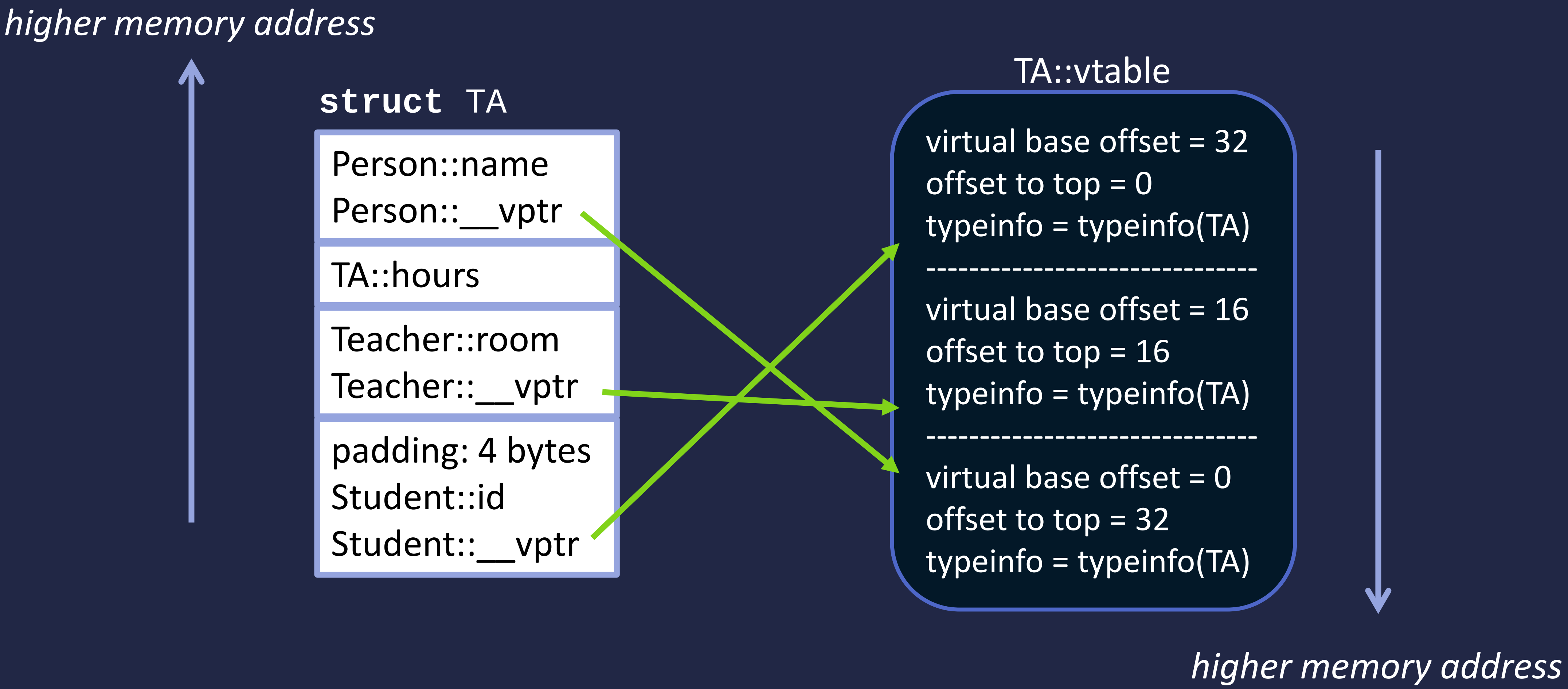
- Downcasting in multiple inheritance requires vtable
  - If base class is not virtual, cannot downcast

```
Person * person = new TA();
// error: source type ‘person’ is not polymorphic
Student * student = dynamic_cast<Student *>(person);
```

- Force vtable by adding a virtual destructor

```
struct Person {
    const char * name;
    virtual ~Person() {}
};
```

Object Layout



higher memory address
struct TA
Person::name
Person::__vptr
TA::hours
Teacher::room
Teacher::__vptr
padding: 4 bytes
Student::id
Student::__vptr
TA::vtable
virtual base offset = 32
offset to top = 0
typeinfo = typeinfo(TA)
----
virtual base offset = 16
offset to top = 16
typeinfo = typeinfo(TA)
----
virtual base offset = 0
offset to top = 32
typeinfo = typeinfo(TA)
higher memory address