```
class Text: public Book {
        string topic;
        public:
        Text (string title, string author, int numPages, string topic):
        title{title, author {autor{, numPages {numpages}, topic {topic{ {}
};
```

Above constructor is **wrong:**

1) Fields like title etc, NOT ACCESSIBLE IN Text
2) When an object is created:
    a. space is allocated
    b. superclass part is constructed
    c. fields constructed in declaration order
    d. constructor body runs

HOWEVER, step 2b won't work because book has no default constructor.
Solution: invoke Book's constructor in MIL.

```
public:
        Text(string title, string author, int numPages, string topic):
                Book{title, author, numPages}, topic {topic} {}
};
```

If superclass has no default constructor, subclass must invoke a superclass constructor in MIL.

There is good reason to keep superclass fields ***private*** from subclasses.

Protected access: To give subclasses access to certain members.

```
class Book {
        protected:
          string title, author;
          int numPages;
        public:
          Book(____);
};
```

Better practice: make fields **private** and provide **protected** accessors/mutators.

```
class Book {
        string title, author;
        int numPages;
        protected:
          string getTitle() const;
          void setAuthor (string auth);
        public:
          Book(_____);
        bool isHeavy() const;
};
```

Relationship among Text, Comic, Book is “is a”.

- In UML, open arrow to parent.

Let’s consider isHeavy().

- Ordinary books: >200 pages is heavy for ordinary books
- Textbooks: >500 pages
- Comics: >30 pages

```
class Book {
        …
        public:
        …
        bool isHeavy() const {return numPages > 200;}
};
class Comic: public Book {
        …
        public:
        …
          bool isHeavy() const {return numpages > 30;}
};

Book b {“small book”, “author”, 50};
Comic c{“big comic”, “author”, 40, “spiderman”};

b.isHeavy()     // false
c.isHeavy()     // true
```

Book b = Comic{“big comic”, “author”, 40, “spiderman”};
b.isHeavy() // false
Why?

Book b = Comic{“big comic”, “author”, 40, “spiderman”};

- tries to fit a comic when only enough space is reserve for a Book
- hero field chopped off, comic is now a book
- **NO LONGER A COMIC**

When accessing objects through **pointers or references**, slicing is unnecessary and doesn’t happen.

Book *pb = &c;
Comc *pc = &c;
pc→isHeavy() // heavy
pb→isHeavy() // not heavy
but slicing didn’t happen?!

Same object behaves differently dependingon what kind of pointer is pointing at it.
Compiler uses the **type of the pointer** (or ref) to pick the method.

Solution:
Declare the method virtual.

```
class Book {
        protected:
          int numPages;
        public:
          Book(___);
        virtual bool isHeavy() const {return numPages > 200;}
};

class Comic: public Book {
        …
        public:
          bool isHeavy() const override {return numPage s> 30;}
};

Comic c {_, _, 40, )];
Book *pb = &c;
Book &rb = c;
Comic *pc = &c;

pc→isHeavy()  // true
```

```
rb.isHeavy()     // true
pc→isHeavy()  // true
```

Virtual methods: choose which class's methods to run based on the actual type of the object at runtime.

A book collection:

```
Book *mybooks[20];
// fill mybooks
for (int i = 0; i < 20; ++i) {
        cout << myBooks[i]→isHeavy() << endl; // isHeavy() called based on type of book
}
```

Polymorphism: *"many forms"* accommodating multiple types under one abstraction.