CONSTRUCTORS

Lecturer：陈笑沙

TABLE OF CONTENTS

- 4.1 Classes and Objects
- 4.2 The Necessity of Constructors
- 4.3 Using Constructors
- 4.4 Destructors
- 4.5 Constructors with Parameters
- 4.6 Overloading Constructors
- 4.7 Default Constructors
- 4.8 Class Member Initialization
- 4.9 Order of Object Construction

4.1 CLASSES AND OBJECTS

Classes are custom data types in C++—describing data organization and operations

```
class Point {
    int x, y; // data organization
public:
    void set(int ix, int iy) { // operations
        x = ix;
        y = iy;
    }
};
```

4.1 CLASSES AND OBJECTS

Objects are entities of classes

Variables are entities of basic types

```
int a = 3;  // basic type variable
struct Date { // structure variable
    int year, month, day;
};
Point w;
w.set(3, 5);
```

4.1 CLASSES AND OBJECTS

C++ IS A HYBRID PROGRAMMING LANGUAGE

4.1 CLASSES AND OBJECTS

C++ IS A HYBRID PROGRAMMING LANGUAGE

- Simple data structures, pure compound control statements, function (module) design, forming procedural programming

4.1 CLASSES AND OBJECTS

C++ IS A HYBRID PROGRAMMING LANGUAGE

- Simple data structures, pure compound control statements, function (module) design, forming procedural programming
- Based on classes and objects, with procedural programming as the framework, forming object-oriented programming

4.1 CLASSES AND OBJECTS

C++ IS A HYBRID PROGRAMMING LANGUAGE

- Simple data structures, pure compound control statements, function (module) design, forming procedural programming
- Based on classes and objects, with procedural programming as the framework, forming object-oriented programming
- Based on class hierarchy polymorphism as the main object of data processing, with object-oriented design as the framework, forming object-oriented programming

4.2 THE NECESSITY OF CONSTRUCTORS

All data entities have an initialization requirement

```
int a = 3;     // Integer variable initialization
int a; a = 3;  // Integer variable assignment
double t[] = {1.3, 2.5}; // Array initialization
struct Date {int year, month, day; };
Date d = {1998, 5, 23}; // Structure variable initialization
```

4.2 THE NECESSITY OF CONSTRUCTORS

For objects, initialization is a complex problem

```
class Point {
    int x, y;
public:
    void set(int ix, int iy) {
        x = ix;
        y = iy;
    }
};
int main() {
    // Error, directly assigning to private variables
    Point t = {3, 4};
    Point d; // Produces an uninitialized object
    d.set(3, 4); // Assign values, not initialization
}
```

4.2 THE NECESSITY OF CONSTRUCTORS

If all variables are public, you can do this:

```cpp
class Point {
public:
    int x, y;
};

int main() {
    Point t = {3, 4};
    return 0;
}
```

4.2 THE NECESSITY OF CONSTRUCTORS

If all variables are public, you can do this:

```cpp
class Point {
public:
    int x, y;
};

int main() {
    Point t = {3, 4};
    return 0;
}
```

4.2 THE NECESSITY OF CONSTRUCTORS

If all variables are public, you can do this:

```cpp
class Point {
public:
    int x, y;
};

int main() {
    Point t = {3, 4};
    return 0;
}
```

4.3 USING CONSTRUCTORS

The design of object initialization

Should be completed during object construction, regardless of access permissions

```
class A {
    int a, b;
};
A x = {2, 3}; // OK
A y{2, 3}; // OK
A z(2, 3); // OK
```

4.3 USING CONSTRUCTORS

Constructor syntax:

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even `void`)

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even `void`)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily `public`

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even `void`)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily `public`
- A class can have multiple constructors

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily **`public`**
- A class can have multiple constructors

```
class A {
    int a, b;
public:
    void A(int ia, int ib); // Wrong
    construct(int ia, int ib); // Wrong
    A(int ia, int ib); // Right
};
```

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily **`public`**
- A class can have multiple constructors

```
class A {
    int a, b;
public:
    void A(int ia, int ib); // Wrong
    construct(int ia, int ib); // Wrong
    A(int ia, int ib); // Right
};
```

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily **`public`**
- A class can have multiple constructors

```
class A {
    int a, b;
public:
    void A(int ia, int ib); // Wrong
    construct(int ia, int ib); // Wrong
    A(int ia, int ib); // Right
};
```

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily **`public`**
- A class can have multiple constructors

```
class A {
    int a, b;
public:
    void A(int ia, int ib); // Wrong
    construct(int ia, int ib); // Wrong
    A(int ia, int ib); // Right
};
```

4.3 USING CONSTRUCTORS

Constructor syntax:

- No return type (not even **`void`**)
- Name matches the class name (case-sensitive)
- Can be any access privilege, not necessarily **`public`**
- A class can have multiple constructors

```
class A {
    int a, b;
public:
    void A(int ia, int ib); // Wrong
    construct(int ia, int ib); // Wrong
    A(int ia, int ib); // Right
};
```

4.3 USING CONSTRUCTORS

Example: Adding a constructor to the `Clock` class

```
class Clock {
    int hour, minute, second;
public:
    Clock(int h, int m, int s) {
        ...
    }
};
```

4.3 USING CONSTRUCTORS

Implementing Constructors: Initializing Member Variables

```
Clock(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
}
```

4.3 USING CONSTRUCTORS

Implementing Constructors: Initializing Member Variables

```
Clock(int h, int m, int s) : hour(h), minute(m), second(s) {}
```

4.3 USING CONSTRUCTORS

Constructor Call:

```cpp
class Desk {
public:
    Desk();
private:
    int height, width;
};

Desk::Desk() {
    cout << "Constructor of Desk." << endl;
    height = 3;
    width = 2;
}

Desk desk; // call the constructor
```

4.3 USING CONSTRUCTORS

Constructor Call:

```cpp
class Desk {
public:
     Desk();
private:
     int height, width;
};

Desk::Desk() {
     cout << "Constructor of Desk." << endl;
     height = 3;
     width = 2;
}

Desk desk; // call the constructor
```

4.3 USING CONSTRUCTORS

Constructor Call:

```
class Desk {
public:
    Desk();
private:
    int height, width;
};

Desk::Desk() {
    cout << "Constructor of Desk." << endl;
    height = 3;
    width = 2;
}

Desk desk; // call the constructor
```

4.3 USING CONSTRUCTORS

Constructor Call:

```
class Desk {
public:
    Desk();
private:
    int height, width;
};

Desk::Desk() {
    cout << "Constructor of Desk." << endl;
    height = 3;
    width = 2;
}

Desk desk; // call the constructor
```

4.3 USING CONSTRUCTORS

Constructor Call:

```cpp
class Desk {
public:
    Desk();
private:
    int height, width;
};

Desk::Desk() {
    cout << "Constructor of Desk." << endl;
    height = 3;
    width = 2;
}

Desk desk; // call the constructor
```

4.4 DESTRUCTORS

In class design, if there is a pointer member, the constructor will allocate heap memory and assign it to the pointer, allowing member functions to share the resource

```
class A {
    int* aa;
    int num;
public:
    A(int n) {
        num = n;
        aa = new int[n];
        // delete[] aa; ???
    }
};
```

4.4 DESTRUCTORS

4.4 DESTRUCTORS

- Sometimes when we release a class object, we may need to do some cleanup work, but we may also encounter problems due to forgetting to call these cleanup functions. How can we solve this?

4.4 DESTRUCTORS

- Sometimes when we release a class object, we may need to do some cleanup work, but we may also encounter problems due to forgetting to call these cleanup functions. How can we solve this?
- C++ also considers this for us. Opposite to the constructor, it provides a destructor specifically for handling cleanup work when an object is destroyed.

4.4 DESTRUCTORS

- Destructors have no return type, no parameters, and the function name is the class name prefixed with "~". Destructors will be automatically called when the object's lifespan ends.
- Destructors can also be private, although this is uncommon.

4.4 DESTRUCTORS

```cpp
class A {
    int* aa;
    int num;
public:
    A(int n) {
        num = n;
        aa = new int[n];
    }
    ~A() { // Cannot have parameters
        delete[] aa;
    }
};
```

4.4 DESTRUCTORS

```cpp
class A {
    int* aa;
    int num;
public:
    A(int n) {
        num = n;
        aa = new int[n];
    }
    ~A() { // Cannot have parameters
        delete[] aa;
    }
};
```

4.4 DESTRUCTORS

```cpp
class A {
    int* aa;
    int num;
public:
    A(int n) {
        num = n;
        aa = new int[n];
    }
    ~A() { // Cannot have parameters
        delete[] aa;
    }
};
```

4.5 CONSTRUCTORS WITH PARAMETERS

By passing parameters to the constructor, you can specify the values of member variables.

```cpp
class Student {
    std::string name;
    char gender;
    double gpa;
public:
    Student(const std::string& name, char gender, double gpa):
    name(name), gender(gender), gpa(gpa) {}
};

Student stu("Eric", 'M', 4.5);
```

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.5 CONSTRUCTORS WITH PARAMETERS

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

To abandon this feature, you can do this:

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

To abandon this feature, you can do this:

```
class A{
    int a;
public:
    explicit A(int n):a(n){} // 注意新的关键字
};

int main() {
    A a = 1; // Wrong
    return 0;
}
```

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

To abandon this feature, you can do this:

```
class A{
    int a;
public:
    explicit A(int n):a(n){} // 注意新的关键字
};

int main() {
    A a = 1; // Wrong
    return 0;
}
```

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

To abandon this feature, you can do this:

```
class A{
    int a;
public:
    explicit A(int n):a(n){} // 注意新的关键字
};

int main() {
    A a = 1;  // Wrong
    return 0;
}
```

4.5 CONSTRUCTORS WITH PARAMETERS

If a constructor only has one parameter, you can also initilize the object like this:

```
className obj = parameter;
```

To abandon this feature, you can do this:

```
class A{
    int a;
public:
    explicit A(int n):a(n){} // 注意新的关键字
};

int main() {
    A a = 1; // Wrong
    return 0;
}
```

4.5 CONSTRUCTORS WITH PARAMETERS

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
  - `const char*`

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
    - `const char*`
    - `const std::string&`

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
  - `const char*`
  - `const std::string&`
- Generally speaking, `const char*` is more performant but less safe.

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
  - `const char*`
  - `const std::string&`
- Generally speaking, `const char*` is more performant but less safe.
- Most of the time, it's unnecessary to consider performance to this extreme.

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
  - `const char*`
  - `const std::string&`
- Generally speaking, `const char*` is more performant but less safe.
- Most of the time, it's unnecessary to consider performance to this extreme.
- In most cases, choose `const std::string&`.

4.5 CONSTRUCTORS WITH PARAMETERS

- Aside: What type should string parameters of functions be?
  - `const char*`
  - `const std::string&`
- Generally speaking, `const char*` is more performant but less safe.
- Most of the time, it's unnecessary to consider performance to this extreme.
- In most cases, choose `const std::string&`.
- After `c++17`, you can use `std::string_view`.

4.5 CONSTRUCTORS WITH PARAMETERS

4.5 CONSTRUCTORS WITH PARAMETERS

- Sometimes, when the constructor parameters have the same name as the member variables, you can:

4.5 CONSTRUCTORS WITH PARAMETERS

- Sometimes, when the constructor parameters have the same name as the member variables, you can:
  - Use the `this` pointer

4.5 CONSTRUCTORS WITH PARAMETERS

- Sometimes, when the constructor parameters have the same name as the member variables, you can:
  - Use the `this` pointer
  - Use the member initialization list

4.5 CONSTRUCTORS WITH PARAMETERS

- Sometimes, when the constructor parameters have the same name as the member variables, you can:
  - Use the `this` pointer
  - Use the member initialization list
- The `this` pointer is an implicit pointer in class member functions that points to the current object instance, used to access the object's members.

4.5 CONSTRUCTORS WITH PARAMETERS

```cpp
class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}
};
```

4.5 CONSTRUCTORS WITH PARAMETERS

```
class Point {
    int x, y;
public:
    Point(int x, int y){
        this->x = x;
        this->y = y;
    }
};
```

4.5 CONSTRUCTORS WITH PARAMETERS

The following two ways to call constructors are equivalent:

```cpp
Point p(3, 4); // Traditional way
// If the constructor is explicit, compilation will fail
Point p = {3, 4};
// Recommended way, after C++11,
// distinguishes from function calls
Point p{3, 4};
```

4.6 OVERLOADING CONSTRUCTORS

4.6 OVERLOADING CONSTRUCTORS

- C++ member functions can be overloaded, and constructors can also be overloaded

4.6 OVERLOADING CONSTRUCTORS

- C++ member functions can be overloaded, and constructors can also be overloaded
- A class can provide multiple constructors, i.e., constructor overloading.

4.6 OVERLOADING CONSTRUCTORS

- C++ member functions can be overloaded, and constructors can also be overloaded
- A class can provide multiple constructors, i.e., constructor overloading.
- The purpose of overloading is to meet different initialization needs.

4.6 OVERLOADING CONSTRUCTORS

```
class Clock
{
private:
    int hour, minute, second;
public:
    Clock(int h, int m, int s);
    Clock();
    Clock(const std::string& timestr);
};

int main( )
{
    Clock clock1{23, 12, 0};
    Clock clock2{};
    Clock clock3{“14:45:32”};
```

4.6 OVERLOADING CONSTRUCTORS

```cpp
class Clock
{
private:
    int hour, minute, second;
public:
    Clock(int h, int m, int s);
    Clock();
    Clock(const std::string& timestr);
};

int main( )
{
    Clock clock1{23, 12, 0};
    Clock clock2{};
    Clock clock3{“14:45:32”};
```

4.6 OVERLOADING CONSTRUCTORS

```
class Clock
{
private:
      int hour, minute, second;
public:
      Clock(int h, int m, int s);
      Clock();
      Clock(const std::string& timestr);
};

int main( )
{
      Clock clock1{23, 12, 0};
      Clock clock2{};
      Clock clock3{“14:45:32”};
```

4.6 OVERLOADING CONSTRUCTORS

How to implement a class with multiple construction methods?

```
int main()
{
    Date date1{2000, 3, 4};
    Date date2{2000, 3};
    Date date3{2000};
    Date date4{};
    return 0;
}
```

4.6 OVERLOADING CONSTRUCTORS

SCHEME 1 (OVERLOADING)

```
class Date{
  int year, month, day;   // default private
public:
  // The following four overloaded functions,
  // each corresponding to a way to build an object
  Date();
  Date(int d);
  Date(int m, int d);
  Date(int y, int m, int d);
};
Date::Date(){ year=1900; month=1; day=1; }
Date::Date(int y){ month=4; day=d; year=1996; }
Date::Date(int y, int m){ month=m; day=1; year=1900; }
Date::Date(int y, int m, int d){ month=m; day=d; year=y; }
```

4.6 OVERLOADING CONSTRUCTORS

SCHEME 2 (C++11)

```
class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
      : year(year), month(month), day(day) {}
    Date(int year, int month) : Date(year, month, 1) {}
    Date(int year) : Date(year, 1, 1) {}
    Date() : Date(1900, 1, 1) {}
};
```

4.6 OVERLOADING CONSTRUCTORS

SCHEME 3 (DEFAULT PARAMETERS, RECOMMENDED)

```cpp
class Date
{
    int year, month, day;
public:
    Date(int year = 1900, int month = 1, int day = 1)
        : year(year), month(month), day(day)
    {
    }
};
```

4.7 DEFAULT CONSTRUCTORS

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters
- Creating an object definitely requires a constructor

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters
- Creating an object definitely requires a constructor
- Default constructor: If no constructor is defined in the class, the system will default to a parameterless constructor to fulfill the mission of creating an object

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters
- Creating an object definitely requires a constructor
- Default constructor: If no constructor is defined in the class, the system will default to a parameterless constructor to fulfill the mission of creating an object
- As long as the programmer defines a constructor (regardless of how many), the system will no longer provides a default constructor

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters
- Creating an object definitely requires a constructor
- Default constructor: If no constructor is defined in the class, the system will default to a parameterless constructor to fulfill the mission of creating an object
- As long as the programmer defines a constructor (regardless of how many), the system will no longer provides a default constructor
- The default constructor must be a parameterless constructor, and the parameterless constructor can be customized

4.7 DEFAULT CONSTRUCTORS

- Default constructor is not a constructor with default parameters
- Creating an object definitely requires a constructor
- Default constructor: If no constructor is defined in the class, the system will default to a parameterless constructor to fulfill the mission of creating an object
- As long as the programmer defines a constructor (regardless of how many), the system will no longer provides a default constructor
- The default constructor must be a parameterless constructor, and the parameterless constructor can be customized
- In fact, there is also a corresponding default destructor that exists

4.7 DEFAULT CONSTRUCTORS

```cpp
class Date
{
};

int main()
{
    Date date{}; // ok
    Date date2; // ok
    Date date3(); // ok
    return 0;
}
```

4.7 DEFAULT CONSTRUCTORS

```cpp
class Date
{
};

int main()
{
    Date date{}; // ok
    Date date2; // ok
    Date date3(); // ok
    return 0;
}
```

4.7 DEFAULT CONSTRUCTORS

```cpp
class Date
{
};

int main()
{
    Date date{}; // ok
    Date date2; // ok
    Date date3(); // ok
    return 0;
}
```

4.7 DEFAULT CONSTRUCTORS

```cpp
class Date
{
};

int main()
{
    Date date{}; // ok
    Date date2; // ok
    Date date3(); // ok
    return 0;
}
```

4.7 DEFAULT CONSTRUCTORS

```cpp
class Date
{
};

int main()
{
    Date date{}; // ok
    Date date2; // ok
    Date date3(); // ok
    return 0;
}
```

4.8 CLASS MEMBER INITIALIZATION

4.8 CLASS MEMBER INITIALIZATION

- Constructors always create object space first, then execute constructor body statements

4.8 CLASS MEMBER INITIALIZATION

- Constructors always create object space first, then execute constructor body statements
- If the class contains object members, when the constructor creates object space, it calls the default constructor of the object member, then executes the constructor body statements

4.8 CLASS MEMBER INITIALIZATION

- Constructors always create object space first, then execute constructor body statements
- If the class contains object members, when the constructor creates object space, it calls the default constructor of the object member, then executes the constructor body statements
- If object members do not have a default constructor, use the member initialization list method

4.8 CLASS MEMBER INITIALIZATION

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.8 CLASS MEMBER INITIALIZATION

```
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.8 CLASS MEMBER INITIALIZATION

```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.8 CLASS MEMBER INITIALIZATION

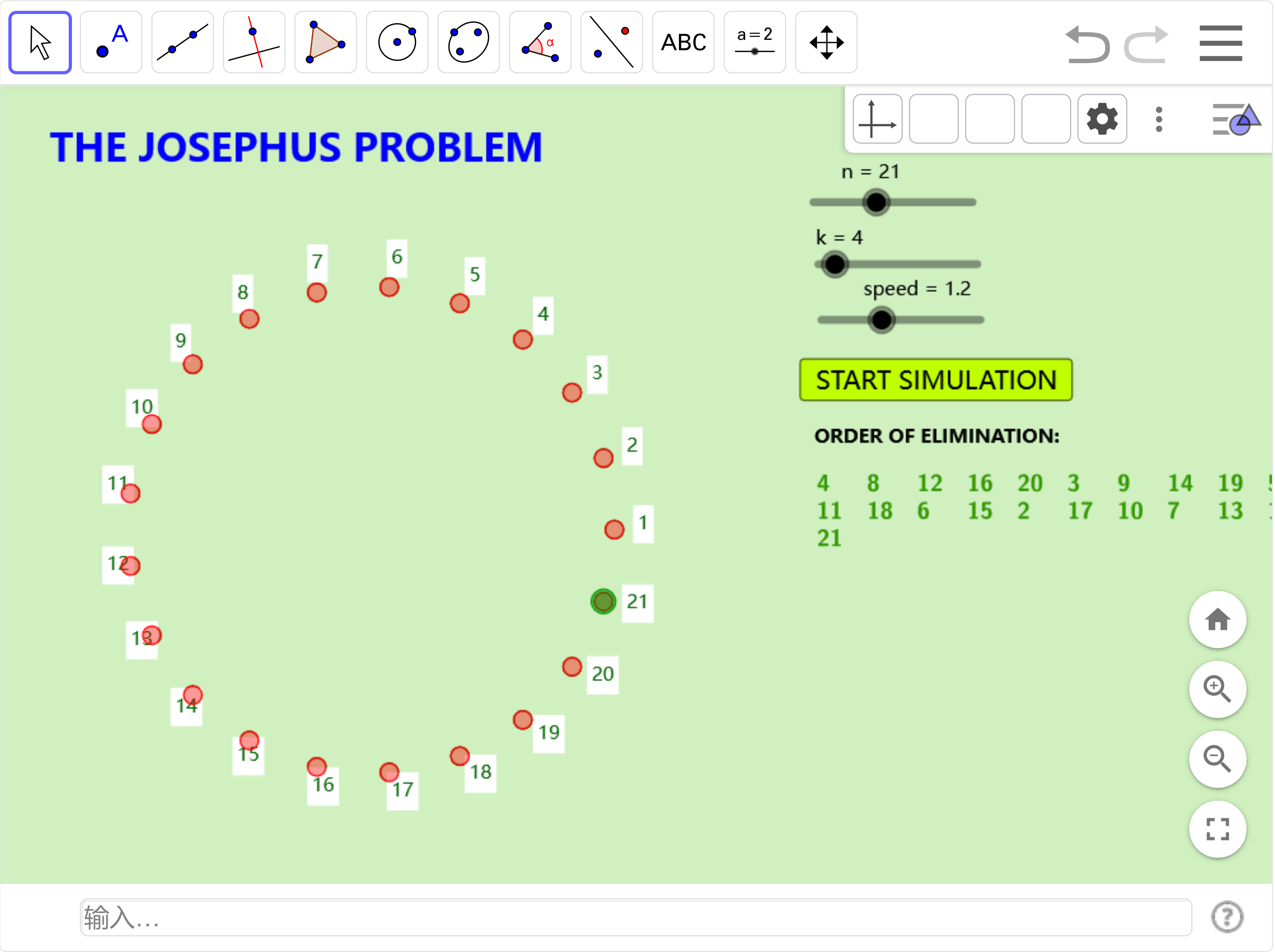
```cpp
class Teacher {
    std::string name;
    uint8_t age;
public:
    Teacher(const std::string& name, uint8_t age) {
        this->name = name;
        this->age = age;
    }
};

class Student {
    std::string name;
    char gender;
    double gpa;
    Teacher t;
```

4.9 ORDER OF OBJECT CONSTRUCTION

Members are constructed in the order they are declared in the class

```
class A {
    int num, age; // First construct num, then construct age
public:
    // The order here does not matter
    A(int n) : age(n), num(age + 1) {}
};
```

EXAMPLE: JOSEPHUS PROBLEM



THE JOSEPHUS PROBLEM

n = 21

k = 4

speed = 1.2

START SIMULATION

ORDER OF ELIMINATION:

```
4    8    12   16   20   3    9    14   19   5
11   18   6    15   2    17   10   7    13
21
```