SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220209<A|D>

Replace <A|D> with this section's letter

Pointers

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Background
- Addresses and pointers
- Pointers and objects
- More on nested types
- In-class problem

Background

Memory addresses

- programs without memory access not very useful
    - instructions stored in memory while program running
    - values need to be stored for later use
- memory access largely invisible in some languages (e.g. Python)

```
char char1 = 'b';
char char2 = 'L';
char char3 = 'x';
```

variable	value
char1	b
char2	L
char3	x

Memory addresses

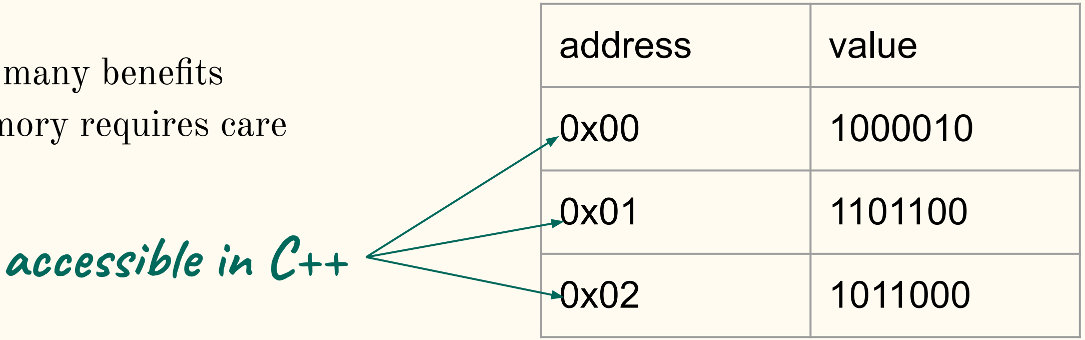
- programs without memory access not very useful
    - instructions stored in memory while program running
    - values need to be stored for later use
- memory access largely invisible in some languages (e.g. Python)

```
char char1 = 'b';
char char2 = 'L';
char char3 = 'x';
```

address	value
0x00	1000010
0x01	1101100
0x02	1011000

Memory addresses

- programs without memory access not very useful
  - instructions stored in memory while program running
  - values need to be stored for later use
- memory access largely invisible in some languages (e.g. Python)
- C/C++ enable direct access to computer memory
  - powerful feature with many benefits
  - directly accessing memory requires care



accessible in C++

address	value
0x00	1000010
0x01	1101100
0x02	1011000

What is possible?

1) Determine where a value is located (its address)
2) Store an address to keep track of where a value is located
3) Access the value stored at a particular location/address

Addresses and pointers

Obtaining an address

- preceding variable with & evaluates to variable's address
- **& - address-of operator**

```
int day = 17;
cout << day << endl;
cout << &day << endl;
```

```
17
0x7ffee69088cc
```

address when code executed on my computer

What's happening here?

- 17 stored at memory address (0x7ff...)
- memory address referred to as `day`

Storing an address

```
int day = 17;
cout << day << endl;
cout << &day << endl;

addr = &day;
cout << addr << endl;
```

compilation error!!

Storing an address

```
int day = 17;
cout << day << endl;
cout << &day << endl;

?? addr = &day;
cout << addr << endl;
```

compilation error!!

Storing an address

```
int day = 17;
cout << day << endl;
cout << &day << endl;

int* addr = &day;
cout << addr << endl;
```

~~compilation error!!~~

```
17
0x7ffee69088cc
0x7ffee69088cc
```

- `*` after a type defines the variable as a *pointer* of that type
- pointer variables store memory addresses
- a pointer can be declared for any type
    - Vorlon*
    - double*
    - char*
    - etc
- pointers are all of the same size

Pointer declarations

```
int* addr = &day;
```

addr - pointer to int (int "star")

```
int *addr2 = &week;
```

addr2 - pointer to int

```
int * addr3 = &year;
```

addr3 - pointer to int

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220209<A|D>

Replace <A|D> with this section's letter

What is the type of `other`?

```
int* addr, other;
```

Pointer declarations

```
int* addr = &day;
```

addr - pointer to int"(int "star")

```
int *addr2 = &week;
```

addr2 - pointer to int

```
int * addr3 = &year;
```

addr3 - pointer to int

```
int* addr, other;
```

asterisk binds to
nearest variable name

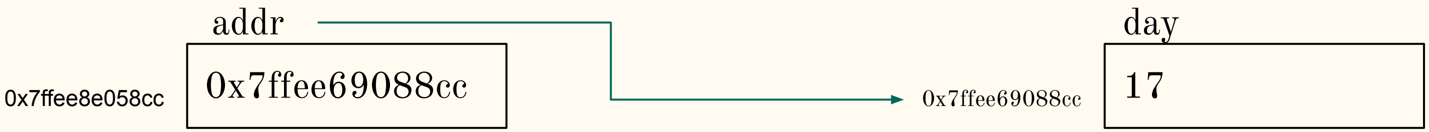
regular `int`

Accessing values

```
int day = 17;
cout << day << endl;
cout << &day << endl;
int* addr = &day;
cout << addr << endl;
cout << *addr << endl;
```

```
17
0x7ffee69088cc
0x7ffee69088cc
17
```

- outside of declaration, asterisk (*) used as **dereference operator**
  - only works with pointer types
- expression `*ptr` evaluates to value *stored* at address



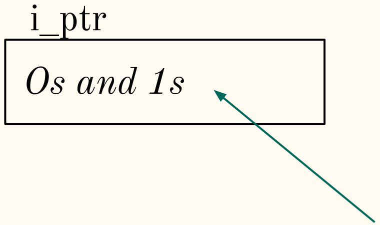
`*addr <==> day`

Initializing pointers

- initializing variables always a good idea
  - `int day = -1;`
  - `double salary = 0;`
- `nullptr` can be assigned to any pointer variable
  - represents an address that *cannot* be used

```
int* i_ptr;

…

*i_ptr = 23;
```

previous value at address may be important



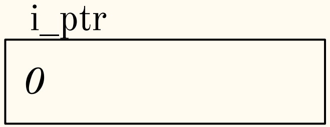
value interpreted as an address

Which address??

Initializing pointers

- initializing variables always a good idea
  - `int day = -1;`
  - `double salary = 0;`
- `nullptr` can be assigned to any pointer variable
  - represents an address that *cannot* be used

```
int* i_ptr = nullptr;
…
*i_ptr = 23;
```



i_ptr

0

nullptr evaluated as 0

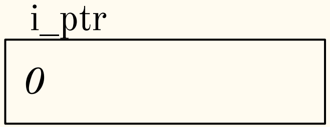
program crash!!

GOOD - only YOUR program will crash

Initializing pointers

- initializing variables always a good idea
  - `int day = -1;`
  - `double salary = 0;`
- `nullptr` can be assigned to any pointer variable
  - represents an address that *cannot* be used

```
int* i_ptr = NULL;
```



alternative

(use nullptr in your code)

Pointers and objects

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;

};

int main() {
    Person george("George");
    Person* ptr = &george;
    // display george using ptr
}
```

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;

};

int main() {
    Person george("George");
    Person* ptr = &george;

    // display george using ptr
    ---
}
```

Accessing object members via pointers

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;
};

int main() {
    Person george("George");
    Person* ptr = &george;

    // display george using ptr
    _1_
}
```

How do we invoke the `display()` method on the person object `george` (replacing blank #1)?

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;
};

int main() {
    Person george("George");
    Person* ptr = &george;

    // display george using ptr
    _1_
}
```

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;

};

int main() {
    Person george("George");
    Person* ptr = &george;

    // display george using ptr
    *ptr.display();
}
```

compilation error!!

dot (.) higher precedence than dereference (*)

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;

};

int main() {
    Person george("George");
    Person* ptr = &george;

    // display george using ptr
    (*ptr).display();
}
```

~~compilation error!!~~

dot (.) higher precedence than dereference (*)

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;
};

int main() {
    Person george("George");
    Person* ptr = &george;

    (*ptr).display();
}
```

works but cumbersome syntax

Accessing object members via pointers

```
class Person {

public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
private:
    string name;

};

int main() {
    Person george("George");
    Person* ptr = &george;

    ptr->display(); // equivalent to (*ptr).display() but looks better
}
```

-> known as "arrow" operator

The `this` pointer

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }

private:
    string name;
};
```

The `this` pointer

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
    void set_name(const string& the_name) { name = the_name; }
private:
    string name;
};
```

function parameter

member variable

The `this` pointer

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
    void set_name(const string& name) { name = the_name; }
private:
    string name;
};
```

function parameter

member variable

The `this` pointer

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
    void set_name(const string& name) { name = name; }
private:
    string name;
};
```

function parameter ?

compilation error!!

member variable ?

The `this` pointer

```
class Person {
public:
    Person(const string& name) : name(name) {}
    void display() const {
        cout << "Name: " << name << endl;
    }
    void set_name(const string& name) { this->name = name; }
private:
    string name;
};
```

function parameter ✓

~~compilation error!!~~

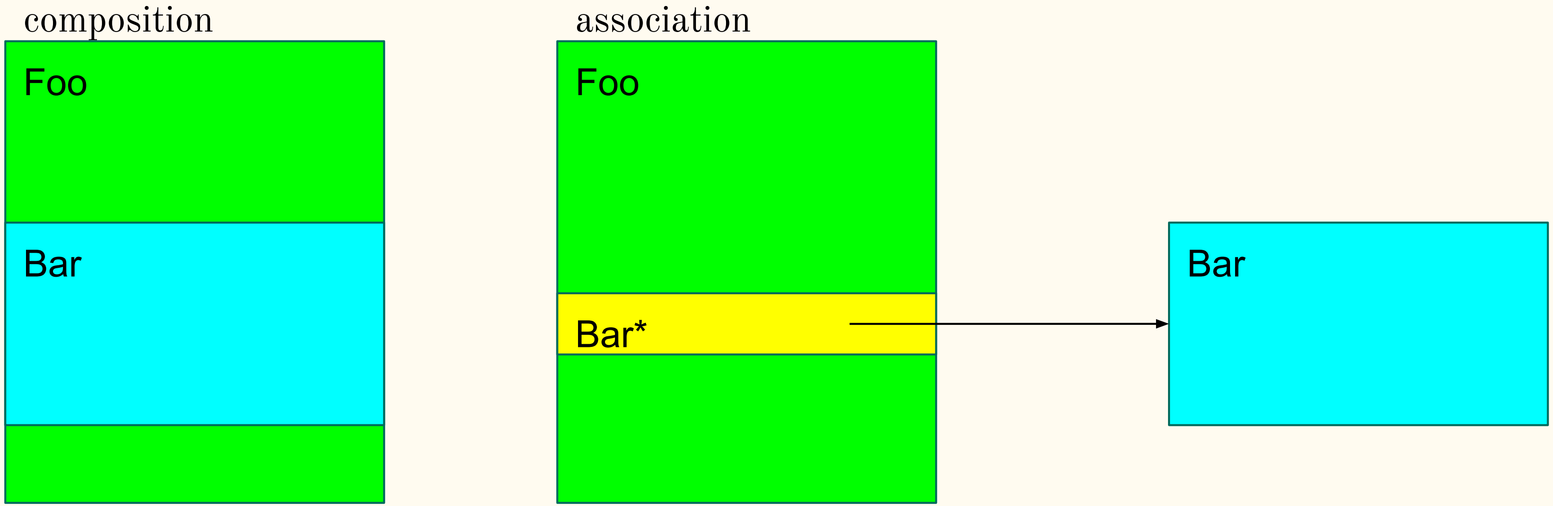
member variable ✓

- `this` - access to current object from within a method
    - pointer to current object (stores address of current object)
    - use `->` for member access
        - `this->name`
        - `this->dob`
        - etc

Nested types (again)

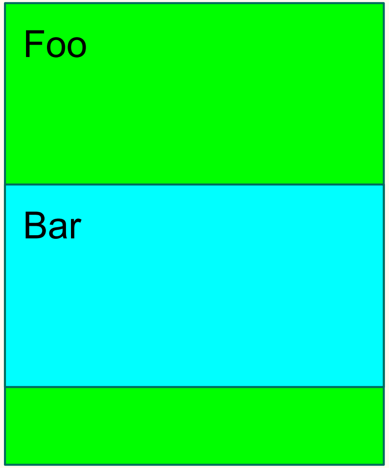
Nested types: composition vs association

Goal: bind objects together



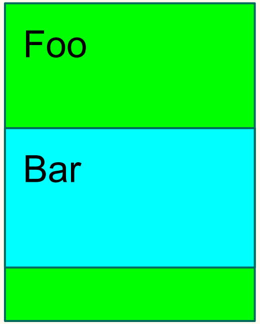
recall Person/Date relationship

Composition



```
class Bar {}; // Not interested Bar details for now

class Foo {
   public:
       ... // ignoring details
   private:
       Bar my_bar;  // Every Foo has a Bar
       int some_other_data;
};
```

Composition

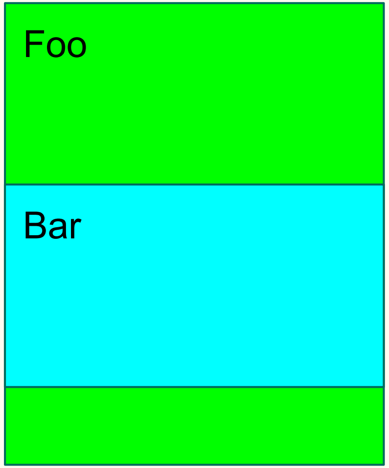


```
class Bar {}; // Not interested Bar details for now

class Foo {
    public:
        ... // ignoring details
    private:
        Bar my_bar;   // Every Foo has a Bar
        int some_other_data;
};
```

- creation of `Foo` creates `Bar`
- destroying `Foo` destroys `Bar`
- `Foo` stuck with `Bar` once created
- `Bar` cannot be shared
- `Foo` size impacted by `Bar` size

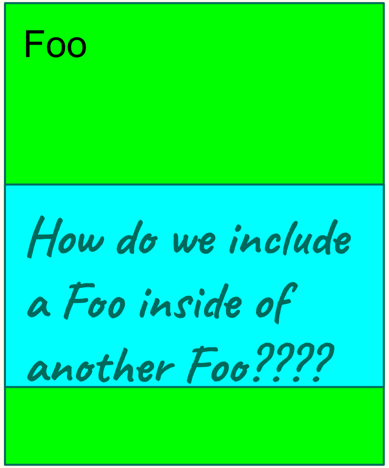
consequences of design choice

Composition



```
~~class Bar {};~~ // Not interested Bar details for now

class Foo {
   public:
       ... // ignoring details
   private:
       ~~Bar my_bar;~~ // Every Foo has a Bar
       int some_other_data;
};
```

Composition

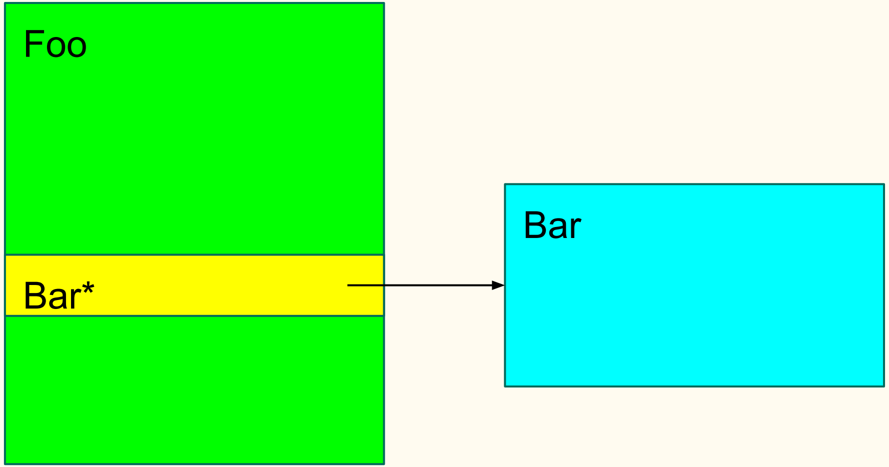


Foo

How do we include a Foo inside of another Foo????

```
class Foo {
   public:
       ... // ignoring details
   private:
       Foo my_foo;  // Every Foo has a Foo
       int some_other_data;
};
```

Association



```
class Bar {};

class Foo {
   public:
       ... // ignoring details
   private:
       Bar* my_bar; // pointer to Bar
       int some_other_data;
};
```

- associate Bar with Foo on-demand
- destroy Bar independently
- Foo not stuck with same Bar
- multiple Foos can share Bar
- Foo size not impacted by Bar size

- Foo can exist without Bar association
    - my_bar = nullptr;

Defining a nested class (elaboration)

```
class Vorlon {
    class Date {
        public:
            Date(int month, int day, int year)
                : month(month), day(day), year(year) {}

            void display() const {
                cout << month << '/' << day << '/' << year;
            }
        private:
            int month, day, year;
    };

public:
    Vorlon(const string& a_name, int b_month, int b_day, int b_year)
    : my_name(a_name), bday(b_month, b_day, b_year) {}

    void display() {
        cout << "Displaying a Vorlon named " << my_name << endl;
    }
private:
    const string my_name;
    Date bday;
};
```

`Date` is only accessible through `Vorlon` class

- `Date` is private class of `Vorlon`
- must use `Vorlon::Date`

scope resolution operator

Defining a nested class (elaboration)

```
class Vorlon {
    class Date {
        public:
            Date(int month, int day, int year)
                : month(month), day(day), year(year) {}

            void display() const {
                cout << month << '/' << day << '/' << year;
            }
        private:
            int month, day, year;
    };

public:
    Vorlon(const string& a_name, int b_month, int b_day, int b_year)
    : my_name(a_name), bday(b_month, b_day, b_year) {}

    void display() {
        cout << "Displaying a Vorlon named " << my_name << endl;
        cout << "Born on " << bday.month << '/' << bday.day << '/' << bday.year;
    }
private:
    const string my_name;
    Date bday;
};
```

compilation error!!

Defining a nested class (elaboration)

```
class Vorlon {
    class Date {

        friend Vorlon;
        public:
            Date(int month, int day, int year)
                : month(month), day(day), year(year) {}

            void display() const {
                cout << month << '/' << day << '/' << year;
            }
        private:
            int month, day, year;
    };

public:
    Vorlon(const string& a_name, int b_month, int b_day, int b_year)
    : my_name(a_name), bday(b_month, b_day, b_year) {}

    void display() {
        cout << "Displaying a Vorlon named " << my_name << endl;
        cout << "Born on " << bday.month << '/' << bday.day << '/' << bday.year;
    }
private:
    const string my_name;
    Date bday;
};
```

~~compilation error!!~~

In-class problem

Person relationships

Enabling Person objects to marry!

as long as neither already married

Person class (so far)

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Person: name = " << rhs.name ~~<< ", dob = " << rhs.dob~~;
        return os;
    }

public:
    Person(const string& the_name~~, int b_month, int b_day, int b_year~~)
        : name(the_name)~~, dob(b_month, b_day, b_year)~~ {}
    ~~void eat() const { cout << name << " eating\n"; }~~
    ~~void set_name(const string& the_name) { name = the_name; }~~

private:
    string name;
    ~~Date dob;~~
};
```

Person class (so far)

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Person: name = " << rhs.name;
        return os;
    }

public:
    Person(const string& the_name) : name(the_name) {}

private:
    string name;
};
```

Person class

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Person: name = " << rhs.name;
        return os;
    }

public:
    Person(const string& the_name) : name(the_name) {}

private:
    string name;
};
```

In which section of the `Person` class would a `spouse` variable be declared?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Person: name = " << rhs.name;
        return os;
    }

public:
    Person(const string& the_name) : name(the_name) {}

private:
    string name;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name) {}

private:
    string name;
    ___;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name) {}

private:
    string name;
    ___ spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name) {}

private:
    string name;
    _1_ spouse;
};
```

Which *type* replaces blank #1 to allow one `Person` to be *associated* with another `Person` using the name `spouse`?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name) {}

private:
    string name;
    _1_ spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name) {}

private:
    string name;
    Person* spouse;
};
```

`Person*` allows `Person` object to be unmarried

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(___) {}

private:
    string name;
    Person* spouse;
};
```

`Person*` allows `Person` object to be unmarried

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(_2_) {}

private:
    string name;
    Person* spouse;
};
```

`Person*` allows `Person` object to be unmarried

In order for a newly created Person to be considered unmarried/single, which value replaces blank #2?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(_2_) {}

private:
    string name;
    Person* spouse;
};
```

`Person*` allows `Person` object to be unmarried

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

private:
    string name;
    Person* spouse;
};
```

`Person*` allows `Person` object to be unmarried

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry() {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry(___) {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry(___ fiance) {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry(_3_ fiance) {
    }

private:
    string name;
    Person* spouse;
};
```

Which *type* replaces blank #3 for the `fiance` parameter if we do not want to allow modifications to the object?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry(_3_ fiance) {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    // enable one Person to marry another
    ___ marry(const Person& fiance) {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
        ___
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
        spouse = ___;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
        spouse = _4_;
    }

private:
    string name;
    Person* spouse;
};
```

Which expression replaces blank #4 indicating that the current Person object's spouse "points to" the Person named fiance?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
        spouse = _4_;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        // marry fiance
        spouse = &fiance;      // this->spouse = &fiance;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        ___
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        fiance.spouse = ___;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        fiance.spouse = _5_;
    }

private:
    string name;
    Person* spouse;
};
```

Which expression replaces blank #5 to assign the *current* `Person` object as the `fiance`'s `spouse`?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(const Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        fiance.spouse = _5_;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(~~const~~ Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

~~compilation error!!~~

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(Person& fiance) {
        this->spouse = &fiance;
        // fiance marry this Person
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Persons (people)
now married

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Persons (people)
now married

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    _6_ marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Given the current definition of `marry`, which return type replaces blank #6?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    _6_ marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name;
        // share marriage status
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";
        // share marriage status
        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == ___) {
            os << "Single";
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == _7_) {
            os << "Single";
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Which value replaces blank #7 so that `"Single"` will be output when the current `Person` object is not married?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == _7_) {
            os << "Single";
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```cpp
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == nullptr) {
            os << "Single";
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        // share marriage status
        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

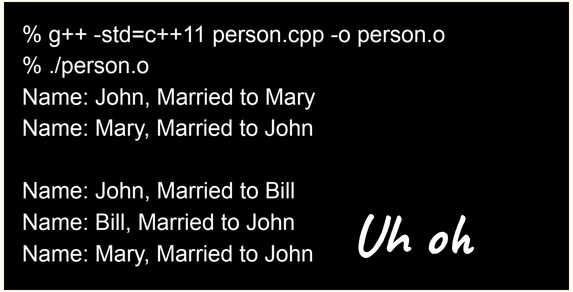
```
int main() {
    Person john("John");
    Person mary("Mary");

    john.marry(mary);
    cout << john << '\n' << mary << '\n';

}
```

```
% g++ -std=c++11 person.cpp -o person.o
% ./person.o
Name: John, Married to Mary
Name: Mary, Married to John
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

```
int main() {
    Person john("John");
    Person mary("Mary");

    john.marry(mary);
    cout << john << '\n' << mary << '\n';

    Person bill("Bill");

    john.marry(bill);
    cout << '\n' << john << '\n';
    cout << bill << '\n' << mary << '\n';
}
```



```
% g++ -std=c++11 person.cpp -o person.o
% ./person.o
Name: John, Married to Mary
Name: Mary, Married to John

Name: John, Married to Bill
Name: Bill, Married to John
Name: Mary, Married to John
```

Uh oh

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Enabling Person objects to marry!

as long as neither already married

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        this->spouse = &fiance;
        fiance.spouse = this;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == ___ ___ this->spouse == ___) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == _8_ ___ this->spouse == _8_) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Which value replaces blank #8 to evaluate whether the `Person` objects are single (able to marry)?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == _8_ ___ this->spouse == _8_) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == nullptr ___ this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == nullptr _9_ this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Which operator replaces blank #9 to complete the condition evaluating whether the `Person` objects can marry?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == nullptr _9_ this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        // check if Person objects already married
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    void marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    ___ marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    _10_ marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Which *type* do we return (replacing blank #10) to indicate whether the marriage operation was successful or not?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    _10_ marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```cpp
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
            // indicate that marriage was successful
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
            // indicate that marriage was successful
            ___
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
            // indicate that marriage was successful
            _11_
        }
    }

private:
    string name;
    Person* spouse;
};
```

Which statement replaces blank #11 to indicate that the marriage was successful?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
            // indicate that marriage was successful
            _11_
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;
            // indicate that marriage was successful
            return true;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }
        // indicate that marriage was not successful
        ___
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }
        // indicate that marriage was not successful
        _12_
    }

private:
    string name;
    Person* spouse;
};
```

Which statement replaces blank #12 to indicate that the marriage was **not** successful?

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }
        // indicate that marriage was not successful
        _12_
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```cpp
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }
        // indicate that marriage was not successful
        return false;
    }

private:
    string name;
    Person* spouse;
};
```

Person class marriage

```
class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", ";

        if (rhs.spouse == nullptr) {
            os << "Single";
        } else {
            os << "Married to ";
            os << rhs.spouse->name;
        }

        return os;
    }

public:
    Person(const string& name) : name(name), spouse(nullptr) {}

    bool marry(Person& fiance) {
        if (fiance.spouse == nullptr && this->spouse == nullptr) {
            this->spouse = &fiance;
            fiance.spouse = this;

            return true;
        }

        return false;
    }

private:
    string name;
    Person* spouse;
};
```

```
int main() {
    Person john("John");
    Person mary("Mary");

    john.marry(mary);
    cout << john << '\n' << mary << '\n';


    Person bill("Bill");

    john.marry(bill);
    cout << '\n' << john << '\n';
    cout << bill << '\n' << mary << '\n';
}
```

```
% g++ -std=c++11 person.cpp -o person.o
% ./person.o
Name: John, Married to Mary
Name: Mary, Married to John

Name: John, Married to Mary
Name: Bill, Single
Name: Mary, Married to John
```