Lec-7: C++ Classes Continue

Bernard Nongpoh

explicit Constructors

```
class Date{
  int d, m, y;
  public:
     Date(int dd=0, int mm=0, int yy=0);
};

void my_date(Date d);

// These all compile but are CONFUSING
Date d{15};  // OK: {15, 0,0} - clear intent
my_date(15); // WHAT? 15 converts to Date?
d=15; // WHAT? Assigning int to Date? Confusing
```

There's no logical connection between 15 and a Date. This implicit conversion is misleading!

explicit Constructors

```
class Date{
  int d, m, y;
  public:
      explicit Date(int dd=0, int mm=0, int yy=0);
};

void my_date(Date d);

// These all compile but are CONFUSING
Date d{15};  // OK: {15, 0,0} - clear intent
my_date(15); // WHAT? 15 converts to Date?
d=15; // WHAT? Assigning int to Date? Confusing
```

The Solution: explicit Keyword

What `explicit` Prevents

```
ALLOWED (Explicit initialization)

Date d1{15};  // OK: Direct initialization
Date d2 = Date(15); // OK: Explicit conversion

BLOCKED (Implicit conversion):
Date d3={15}; // ERROR
Date d4=15; // ERROR
my_date(15); // ERROR
my_date({15}); // ERROR
```

In-Class Initializers (1)

```
class Date{

 int d, m, y;

 public:
    Date(int dd, int mm, int yy):d{dd}, m{mm}, y{yy}{}
    Date(int dd, int mm):d{dd},m{mm}, y{2026}{} // Repeat y init
    Date(int dd): d{dd}, m{2}, y{2026}{} // Repeat m, y
    Date():d{3},m{2},y{2026}{} // All
};
```

PROBLEM: `d{3},m{2},y{2026}` repeated many times

- Easy to make mistakes
- Hard to change defaults

In-Class Initializers (2)

```
class Date{

 int d{3}, m{2}, y{2026};

 public:
    Date(int dd, int mm, int yy):d{dd}, m{mm}, y{yy}{}
    Date(int dd, int mm):d{dd},m{mm}{} // y uses default
    Date(int dd): d{dd}{} // m, y use defaults
    Date(){} // all use defaults

};
```

In-Class Initializer Precedence (1)

```
class Date{

 int d{3}, m{2}, y{2026};

 public:
    Date(int dd): d{dd}{} // m, y use defaults

};
```

In-Class Initializer (backup)

Constructor initializer (overrides)

In-Class Initializer Precedence (2)

```
class Date{

 int d{3}, m{2}, y{2026};

 public:
    Date(int dd): d{dd}{} // m, y use defaults

};


Date d{15};
// d=15
//m=2;
//d=3
```

In-Class Initializer (backup)

Constructor initializer (overrides)

RULE: Constructor initializer **WINS** over in-class initializer

`const` Member Functions: Protecting Object State (1)

WHY WE NEED **const** MEMBER FUNCTIONS?

```
class Date{

 int d, m, y;

 public:
    int day(){return d;} // Just reads .. or does it?
    void add_year(int n);

};

void f(const Date& cd){
    int j=cd.day(); // Is this safe?
// compiler doesn't know if day() modifies the object!
}
```

Without **const**, the compiler can't verify that a function doesn't modify the object.

`const` Member Functions: Protecting Object State (2)

The solution: const Member Functions

```
class Date{

 int d, m, y;

 public:
    int day() const {return d;} // Promise: won't modify
    int month() const {return m;} // Promise: won't modify
    int year() const {return y;} // Promise: won't modify

    void add_year(int n);  // May modify (not const)

};
```

`const` Member Functions: Protecting Object State (3)

COMPILER ENFORCES THE PROMISE:

```
int Date::year() const{
    return ++y; // ERROR! Can't modify y in const function
}
int Date::year(){ // Missing const
    return y; // ERROR const missing in definition
}
```

`const` Member Functions: Protecting Object State (4)

COMPILER ENFORCES THE PROMISE:

```
int Date::year() const{
    return ++y; // ERROR! Can't modify y in const function
}
int Date::year(){ // Missing const
    return y; // ERROR const missing in definition
}
```

const is PART OF THE FUNCTION'S TYPE

```
int day() → type: int Date::day()
int day() const → type Date::day() const
```

These are DIFFERENT function types

const vs Non-const Objects

WHAT CAN BE CALLED ON WHAT

	const member function	Non-const member function
No-const object	OK	OK
const object	OK	ERROR

```
void f(Date&d, const Date& cd){

    int i=d.year(); // OK (const fn on non-const obj)
    d.add_year(1); // OK (non-const fn on non-const

    int j = cd.year(); // OK (const fn on const obj)
    cd.add_year(1); // ERROR (non-=const on const)
}
```

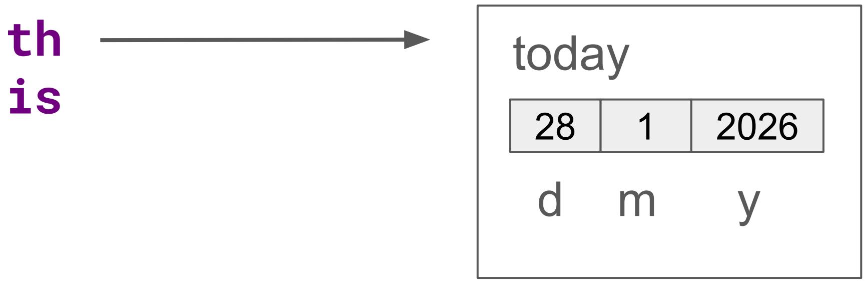
The this Pointer

When you call: `today.add_year(5);`

The compiler sees: `add_year(&today, 5);`

Hidden **this** parameter

Inside `add_year()`, this points to today



Implicit vs Explicit this

```
Date& Date::add_year(int n){

// These are equivalent
y+=n;  // Implicit: compiler adds “this->”
this->y+=n; // Explicit
Return *this; // Return the object, not the pointer
}
```

Using this for Method Chaining

```
// We want to write
d.add_day(1).add_month(1).add_year(1);

// How it works:

d.add_day(1) → return *this (reference to d)


            .add_month(1) → return *this (reference to d)

                        .add_year(1) →*this
```

this: in const vs non-const functions

```
class X{
 void f(); // this has type: X*
  void g() const; // this has type const X*
};
```

```
In non-const function f():
 -  this  → X*
 -  Can modify members through this
 -  this->m=5; // OK
 -
```

```
In const function f():
 -  this  → const X*
 -  Cannot modify members through this
 -  this->m=5; // ERROR
 -
```

`this pointer`

- You cannot assign to this
  - `this=something`, this is an **rvalue**

- You cannot take the address of this
  - `&this` is not allowed because this is an **rvalue**

Static Members (1)

The **GLOBAL** Variable Problem

```
class Date{
int d, m, y;
public:
Date(int dd=0,int mm=0, int yy=0);
};
Date today;
Date::Date(int dd,int mm, int yy){
 d=dd?dd:today.d; // depends on global!
  m = mm? mm:today.m;
  y=yy?yy:today.y;
}
int main(){
    today={1,1,2026};
   Date d;
}
```

Static Members (2)

The **GLOBAL** Variable Problem

```
class Date{
int d, m, y;
public:
Date(int dd=0,int mm=0, int yy=0);
};
Date today;
Date::Date(int dd,int mm, int yy){
 d=dd?dd:today.d; // depends on global!
  m = mm? mm:today.m;
  y=yy?yy:today.y;
}
int main(){
    today={1,1,2026};
   Date d;
}
```

- Anyone can modify today
- Hard to maintain across multiple files
- Create hidden dependencies

Static Members (3)

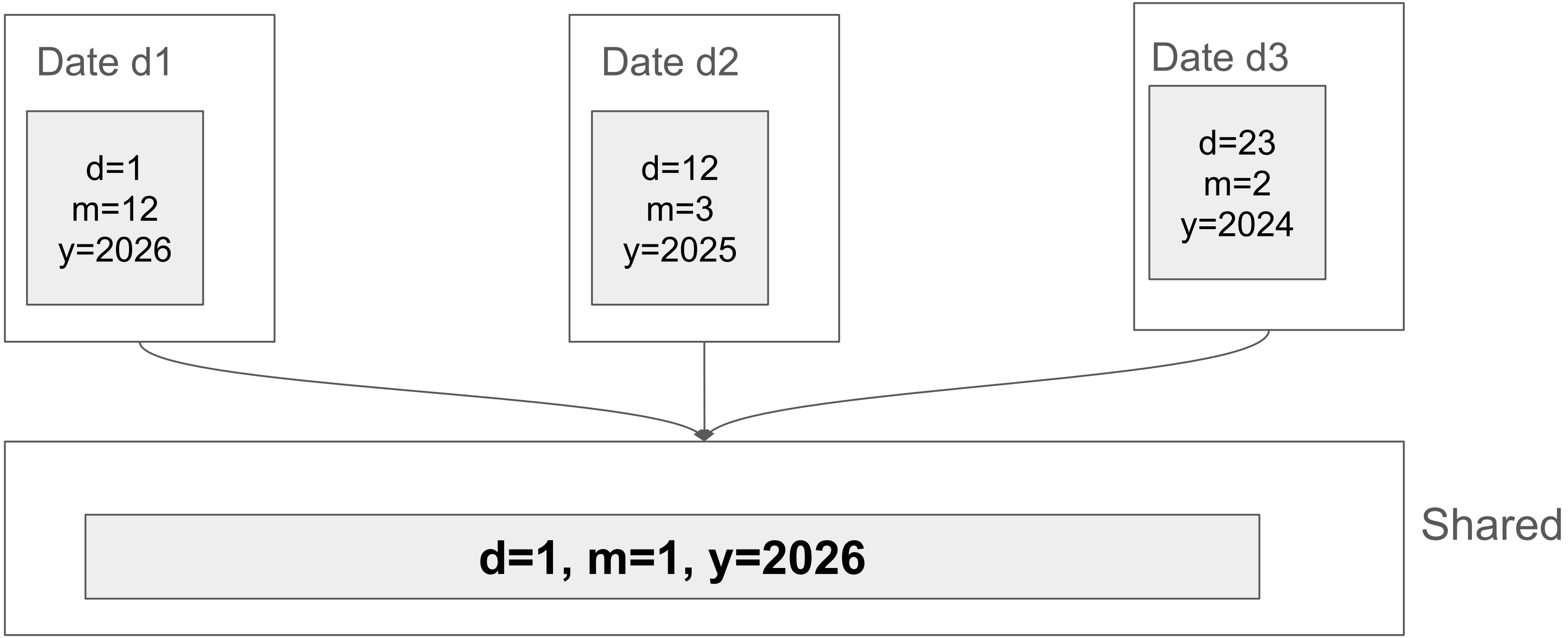
The GLOBAL Variable Problem

```
class Date{
int d, m, y;


Public:
Static Date default_date; // Shared by ALL Date objects
Date(int dd=0,int mm=0, int yy=0);
Static void set_default(int dd, int mm, int yy);
};
Date Date::default_date(1,1,2026);
```

Static vs Non-Static Members

NON-STATIC: One copy **PER OBJECT**



STATIC: ONE copy for the **WHOLE CLASS**

Accessing Static Members

```cpp
// Method 1: Through an object (works but not recommended)
Date d;
d.set_default(1, 1, 2000);

// Method 2: Through the class name (recommended)
Date::set_default(1, 1, 2000);  // Clear that it's static

// Using static member in constructor:
Date::Date(int dd, int mm, int yy) {
    d = dd ? dd : default_date.d;  // Access static member
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;
}
```

Static vs Non-Static Member Functions

Regular Member Functions	**Static Member Functions**
• Has this pointer • Called on an object • Can access all members	• No this pointer • Called on the class • Can only access static members

`void Date::add_year(int n){`
`this->y+=n; //OK`
`}`
	`static void Date::set_default(int d, int m, int y){`
	`   this->y // ERROR`
	`   default_date={..}; // OK`
	`}`

Usage:
`Date d;`
`d.add_year(5);`
	Usage:
	`Date::set_default(1,1,2026);`
	`// No Object needed`

Important Rules for static Members

Declaration (in class):

- `static Date default_date; // just declares`

Definition (outside class in .cpp):

- `Date Date::default_date{1,1,2026}; // Allocates storage`
- Note no `static` keyword here

Initialization:

- Before `main()` starts
- Order between files is undefined (be careful)

Thread safety

- Static members are shared across threads, use mutex or other synchronization

Class Design Pattern

1. Constructors
- Initialize objects
- Establish invariants
- Throw if invalid

2. Accessors (const member functions)
- day(), month(), year()
- Don’t modify state
- Allow reading data

3. Mutators (non-const member functions)
- add_day(), add_month(), add_year()
- Modify state
- Maintain invariants

4. Helper functions (non-members)
- is_leapyear(), operator==, operator+
- Don’t need private access
- Keep class interface minimal

Class Design Checklist

- Represent concept as classes
  - One **class** = one concept
- Separate interface from implementation
  - **public** = interface, **private** = implementation
- Use struct only from plain data
  - No **invariants**? Use struct
- Need protection? Use **class**
- Define **constructors** for initialization
  - Nevel leave objects **uninitialized**

Class Design Checklist

- Make single argument constructors **explicit**
  - Prevent surprising implicit conversions
- Mark non-modifying functions const
  - If it doesn’t change state
- Make functions members only if necessary
  - Need private access? →member
- Don’t need it? → non-member helpers