Lec-6: C++ Structures & Classes

Bernard Nongpoh

Real-World Anatomy

Blueprint (class)

creates

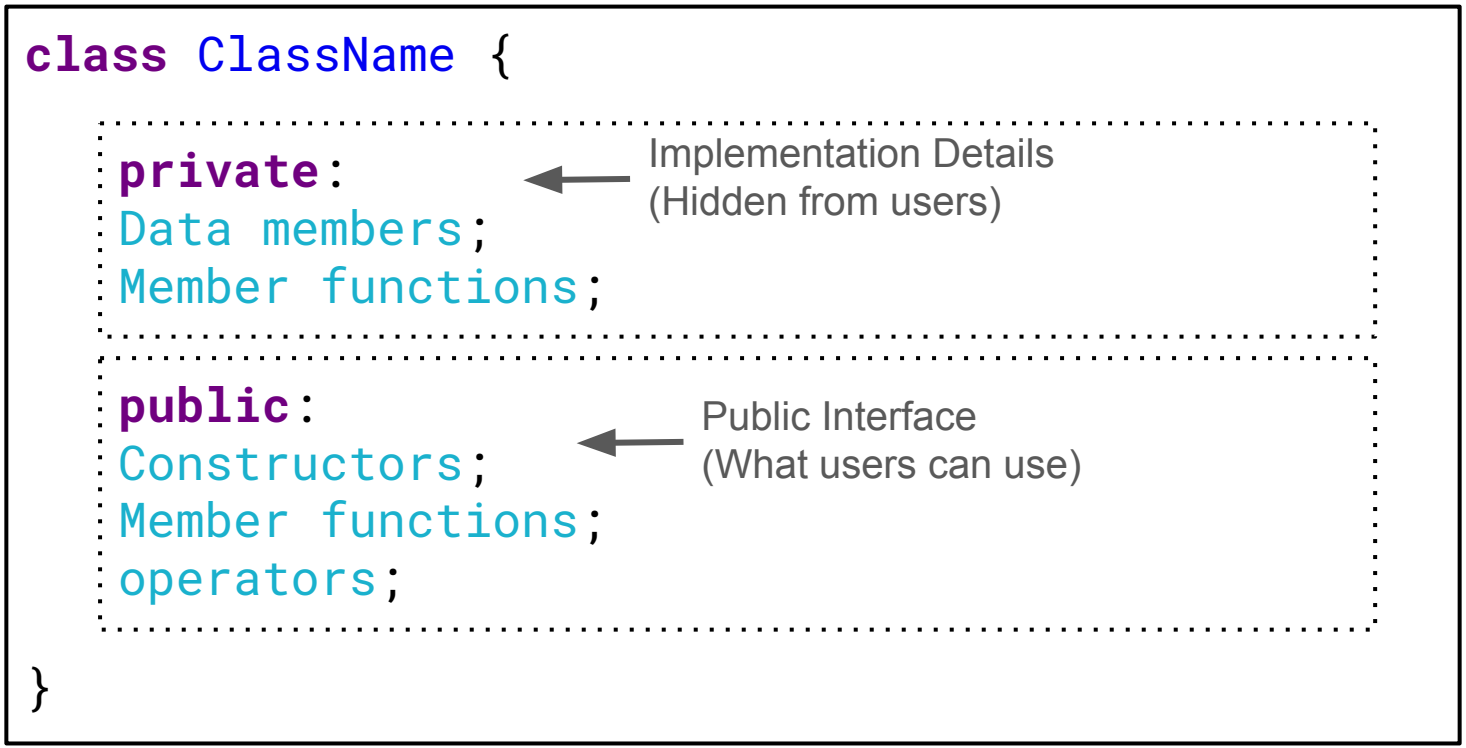
House (object)

Defines structure and behavior

Actual instance with real data

source

Class Anatomy



Class Anatomy

- **Data Members:** Store the object's state
- **Member Functions:** Define the object's behaviour
- **Access Specifiers:** Control who can access what
  - **private** : Only class members
  - **public**: Anyone
- **Constructors**: Initialize objects

Implementing the Concept of a Date

```
struct Date{
  int d, m, y;
};

void init_date(Date& d, int d, int m, int y); // initialize d
void add_year(Date& d, int n); // add n years to d
void add_month(Date& d, int n); // add n months to d
void add_day(Date& d, int n);   // add n days to d
```

There is no explicit connection between the data type `Date`, and these functions

Implementing the Concept of a Date

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

  void init_date(int dd, int mm, int yy); // initialize d
  void add_year(int n); // add n years to d
  void add_month(int n); // add n months to d
  void add_day(int n); // add n days to d
};
```

Function declared within a **`class/struct`** definition are called **member functions**

Implementing the Concept of a Date

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

  void init_date(int dd, int mm, int yy); // initialize d
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};

Date my_birthday;
Date today;

today.init_date(28,1,2026);
my_birthday(10,1,2015);

Date tomorrow=today;
tomorrow.add_day(1);
```

Implementing the Concept of a Date

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

  void init_date(int dd, int mm, int yy); // initialize d
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};

Date my_birthday;
Date today;

today.init_date(28,1,2026);
my_birthday(10,1,2015);

Date tomorrow=today;
tomorrow.add_day(29);
```

🤔 Implementation?

Implementing the Concept of a Date

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

  void init_date(int dd, int mm, int yy);
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};

void Date::init_date(int dd, int mm, int yy){
    dd=d;
    mm=m;
    yy=y;

}
```

```
Date my_birthday;
Date today;

today.init_date(28,1,2026);
my_birthday(10,1,2015);

Date tomorrow=today;
tomorrow.add_day(29);
```

Evolution of Design

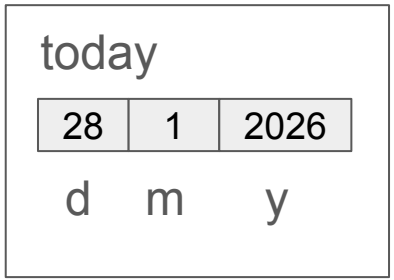
C-Style	C++ Classes

`struct Date{`
`  int d, m, y;`
`};`
 	
`void init_date(Date& d, int d, int m, int y); // initialize d`
`void add_year(Date& d, int n);`
`void add_month(Date& d, int n);`
`void add_day(Date& d, int n);`
	`struct Date{`
	`  int d, m, y;`
	
	`void init_date(int dd, int mm, int yy); // initialize d`
	`  void add_year(int n);`
	`  void add_month(int n);`
	`  void add_day(int n);`
	`};`

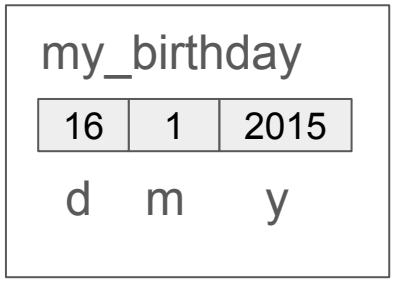
Data and functions are SEPARATE (weak connection)	**Data and functions are TOGETHER (strong connection)**

How Member Functions Know Their Object

```
Date today;
today.init_date(28,1,2026);
```




```
Date my_birthday;
my_birthday.init_date(16,1,2015);
```



Defining Member Functions Outside the Class

Date.h

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

void init_date(int dd, int mm, int yy); //
initialize d
  void add_year(int n);
  void add_month(int n);
  void add_day(int n);
};
```

Date.cpp

```
#include “Date.h”

// Definition (outside class)
void Date::init_date(int dd, int mm, int yy)
{  // Use ClassName::
    dd = d;
    mm = m;
    yy = y;
}
```

The Scope Resolution Operator ::

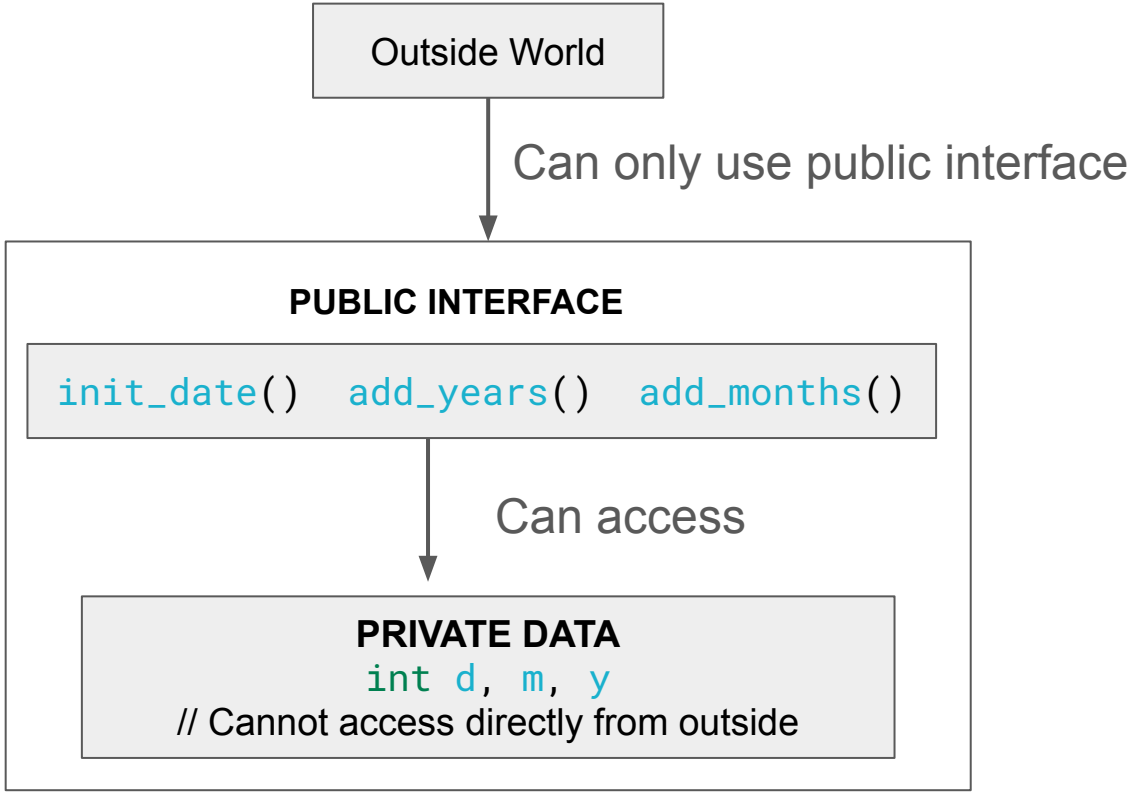
```
void Date::init_date(...)
```

This `init_date` belongs to class Date

Without it: `void init_date(....)` → Global function
With it: void `Date::init_date(...)` → Date's member

Access Control (public vs private)

Encapsulation



Why Access Control

- **Bug localization**
  - If `Date` has illegal value (Dec 36, 2026):
  - Without private: Bug could be **ANYWHERE** in code
  - With private: Bug **MUST** be in member functions
- **Easy modification**
  - Change representation from `{d,m,y}` to `{days_since_1970}`
  - Without private: Change all code using `Date`
  - With private: Change on private member functions
- **Clear interface**
  - Users only need to learn public functions
  - Internal details are hidden and irrelevant

Access Control: Code example

```
class Date {
    int d, m, y;  // private
public:
    void init_date(int dd, int mm, int yy);
    void add_year(int n);
};

// This works:
Date dx;
dx.init_date(25, 3, 2011);  // OK - using public interface

// This fails:
Date dx;
dx.m = 3;               // ERROR: m is private
```

Struct vs Class

struct	class

`struct S{`
`  int x; // public`
`  int y; // public`
`};`
 	
`EQUIVALENT TO:`
 	
`struct S{`
 	
`public:`
`  int x;`
`  int y;`
`};`
	`class C{`
	`  int x; // private`
	`  int y; // private`
	`};`
	
	`EQUIVALENT TO:`
	
	`private C{`
	
	`private:`
	`  int x;`
	`int y;`
	`};`

For simple data with no **invariants**	For types with **invariants** that need protection

An invariant is a logical condition or property that must always be true for an object to be considered in a valid, consistent state

Struct vs Class

struct	class

`struct S{`
`  int x; // public`
`  int y; // public`
`};`
 	
EQUIVALENT
 	
`struct S{`
 	
`public:`
`  int x;`
`  int y;`
`};`
	`class C{`
	`  int x; // private`
	`  int y; // private`
	`};`
	
	`};`

For simple data with no **invariants**	For types with **invariants** that need protection

Invariant:
A condition that should always be true for an object. For **Date**, an invariant might be "**month is always 1-12**" or "**day is valid for the month.**"

An invariant is a logical condition or property that must always be true for an object to be considered in a valid, consistent state

Struct vs Class

struct	class

`struct S{`
`  int x; // public`
`  int y; // public`
`};`
	`class C{`
	`  int x; // private`
	`  int y; // private`
	`};`

`EQUIVALENT`
 	
`struct S{`
 	
`public:`
`  int x;`
`  int y;`
`};`
	`int y;`
	`};`

For simple data with no **invariants**	For types with **invariants** that need protection

In C++, invariants are primarily enforced through encapsulation (making data members `private` or `protected`). This ensures that the only way to modify an object's state is through its member functions, which are responsible for maintaining the invariant.

An invariant is a logical condition or property that must always be true for an object to be considered in a valid, consistent state

Constructors

WITHOUT Constructor	WITH Constructor

`Date d; // d contains garbage`
`d.init_date(28,1,2026);`
`// Programmer might forget or call init_date(..) twice`
	`Date d{28,6,2026}; // d is properly initialized`
	`// No separate init needed`

Uninitialized, Error-prone & Unreliable	Always initialized, safe and Consistent

Constructors

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

public:
  Date(int dd, int mm, int yy); // Constructor Declaration
//…

};
```

Constructor has:

- Same name as the class
- No return type (not even void)
- Can have parameters

Object Creation with Constructors

```
Date today = Date(28,1,2026); // Explicit call
Date xmas(25,12,2026); // Abbreviated () form
Date today = Date{28,1,2026}; // modern {} style
Date xmas{25,12,2026}; //Recommend {} style
Date my_birthday; // ERROR: initializer missing
Date release_1(10,12); // ERROR: third arg missing
```

Multiple Constructors (Overloading)

```
class Date {
      int d, m, y;
 public:
      Date(int, int, int);      // day, month, year
      Date(int, int);           // day, month, today's year
      Date(int);                // day, today's month & year
      Date();                   // default: today
      Date(const char*);        // from string "July 4, 1983"
  };

  USAGE:

  Date today{4};                // → {4, today.m, today.y}
  Date july4{"July 4, 1983"};   // → {4, 7, 1983}
  Date guy{5, 11};              // → {5, 11, today.y}
  Date now;                     // → today's date
  Date start{};                 // → today's date
```

Constructor with Default Arguments

```
class Date {
    int d, m, y;
public:
    Date(int dd = 1, int mm = 1, int yy = 2026);  // One constructor!
};

Date::Date(int dd, int mm, int yy) {
 // Check validity...
    d = dd;
    m = mm;
    y = yy;
   }

// Now all these work:
Date d1{15, 3, 2020};  // All specified
Date d2{15, 3};        // yy = 2026
Date d3{15};           // mm = 1, yy = 2026
Date d4{};
```

Constructor Execution Flow