Balaguruswamy + Topics + Spice

C++ OOP Concepts and Language Fundamentals – Refined Notes

1. Object Oriented Programming Concepts

1.1. Overview: POP vs OOP

Procedure Oriented Programming (POP)

- **Definition:** Program is divided into functions; focus is on procedures and the sequence of tasks.
- **Features:**
  - Top-down approach.
  - Data moves freely between functions; global data is common.
  - No access specifiers.
  - Difficult to add new data/functions.
  - No data hiding; less secure.
  - Overloading not possible.
  - Examples: C, FORTRAN, Pascal.
- **Limitations:**
  - Poor data security.
  - Difficult to manage large codebases.
  - Lacks real-world modeling.

Object Oriented Programming (OOP)

- **Definition:** Program is divided into objects; focus is on data and methods that operate on data.
- **Features:**
  - Bottom-up approach.
  - Data is hidden and protected via access specifiers (public, private, protected).
  - Objects communicate via member functions.
  - Easy to add new data/functions.
  - Data hiding and security.
  - Supports overloading (function/operator).
  - Examples: C++, Java, C#, Python.
- **Advantages:**
  - Modularity, reusability, security, maintainability, and real-world modeling.

1.2. Principles (Features) of OOP

1. **Encapsulation:**

   Wrapping data and functions together as a single unit (class).

   - Data is not directly accessible from outside; access via member functions.
   - Provides *data hiding*.

2. **Data Abstraction:**

   Representing only essential features, hiding implementation details.

   - Classes use abstraction; only relevant attributes and methods are exposed.
   - Classes are Abstract Data Types (ADT).

3. **Inheritance:**

   Mechanism by which one class (derived) acquires properties of another (base).

   - Promotes code reuse and extensibility.
   - Derived class = inherited part (from base) + incremental part (new code).

4. **Polymorphism:**

   Ability to take multiple forms.

   - Same operation behaves differently on different classes.
   - Achieved via function overloading, operator overloading, virtual functions.

5. **Dynamic Binding (Late Binding):**

   The code to be executed in response to a function call is determined at runtime.

   - Enables runtime polymorphism (virtual functions).

6. **Message Passing:**

   Objects communicate by sending messages (calling member functions).

1.3. Benefits of OOP

- **Reusability:** Code/modules/classes can be reused in other programs.
- **Inheritance:** Eliminates redundant code, extends existing code.
- **Data Hiding:** Secure programs by hiding data and functions.
- **Reduced Complexity:** Problem is broken into objects, each handling specific tasks.
- **Easy Maintenance:** Changes in one class do not affect others.
- **Modifiability:** Easy to update data/procedures without affecting the rest of the code.
- **Message Passing:** Simplifies interfaces and communication.

---

2. C++ Language Fundamentals

2.1. Overview of C and C++

C Language

- Structure/procedure-oriented.
- Top-down approach.
- All C code can be executed in C++ (not vice versa).
- No function/operator overloading; no inheritance, encapsulation, or polymorphism.

- Local variables only at block start.
- Data abstraction not supported; data is open to all functions.

C++ Language

- Incremental version of C with OOP features.
- Supports both procedure-oriented and object-oriented paradigms.
- File extension: `.cpp`
- Function/operator overloading possible.
- Variables can be declared anywhere.
- Emphasis on data rather than procedures.
- Supports encapsulation, inheritance, polymorphism, data abstraction, dynamic binding.
- Data access is controlled via access specifiers.

2.2. Structure of a C++ Program

Sections:

1. **Documentation Section:** Comments about the program.
2. **Linking Section:** Header files (`#include <iostream>`).
3. **Definition Section:** Constants (`#define PI 3.14`).
4. **Global Declaration Section:** Global variables/classes.
5. **Class Declarations & Member Function Definitions:** Class blueprints and functions.
6. **Main Function Section:** Entry point (`int main() { ... }`).
    - **Declaration Section:** Variable declarations.
    - **Executable Section:** Statements to perform tasks.

Sample Program:

```
#include <iostream>
using namespace std;

void display() {
    cout << "C++ is better than C";
}

int main() {
    display();
    return 0;
}
```

2.3. Namespace

- Used to define a scope for identifiers.
- `std` is the standard namespace for C++ library.
- **Syntax:**

```
namespace sample {
    int m;
    void display(int n) { cout << n; }
}
using namespace sample;
```

2.4. C++ Tokens

- **Definition:** Smallest individual units in a program.
- **Types:**
  - Keywords (`int` , `if` , `class`)
  - Identifiers (user-defined names)
  - Constants (10, 3.14, 'A', "Hello")
  - Strings
  - Operators (`+` , `-` , `*` , `/`)
  - Punctuators (`;` , `{}` , `()` , `,`)

2.5. Identifiers, Variables, Constants

- **Identifiers:** Names for variables, functions, classes; must start with letter/underscore.
- **Variables:** Storage locations with a name and type.
- **Constants:** Fixed values; declared using `const` or `#define`.

2.6. Data Types

1. Primary (Fundamental):

- `int` , `char` , `float` , `double` , `bool` , `void`
- Example:

```
int a = 10;
char c = 'A';
float f = 3.14;
bool flag = true;
```

2. Derived:

- Arrays, pointers, references, functions.
- Example:

```
int arr[10];
int *p;
int &ref = a;
```

3. User-Defined:

- `struct` , `class` , `union` , `enum`
- Example:

```
struct Point { int x, y; };
enum Color { RED, GREEN, BLUE };
```

2.7. Operators in C++

- **Arithmetic:** `+`, `-`, `*`, `/`, `%`
- **Relational:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical:** `&&`, `||`, `!`
- **Assignment:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- **Increment/Decrement:** `++`, `--`
- **Bitwise:** `&`, `|`, `^`, `~`, `<<`, `>>`
- **Conditional:** `?:`
- **Comma:** `,`
- **Sizeof:** `sizeof(var)`
- **Scope Resolution:** `::`
- **Member Access:** `.` (object), `->` (pointer)
- **Pointer Operators:** `*`, `&`
- **Memory Management:** `new`, `delete`
- **Type Cast:** `(type)`, `static_cast<type>(expr)`

2.8. Control Structures & Loops

- **Selection:** `if`, `if-else`, `switch`
- **Iteration:** `for`, `while`, `do-while`
- **Jump:** `break`, `continue`, `goto`, `return`

---

3. C++ Language Details

3.1. Reference Variables

- **Definition:** Alias for another variable.
- **Syntax:** `int x = 10; int &y = x;`
- **Use:** Pass by reference in functions.

3.2. Scope Resolution Operator (`::`)

- Used to access global variables/functions when local variable of same name exists.
- Used to define member functions outside class.

3.3. Member Dereferencing Operators

- `.` : Access members of object.
- `->` : Access members of object through pointer.

3.4. Memory Management Operators

- `new` : Allocates memory dynamically.

  ```
  int *p = new int;
  ```

- `delete` : Deallocates memory.

  ```
  delete p;
  ```

3.5. Manipulators

- Used to format output.
- **Common Manipulators:**
  - `endl` : New line.
  - `setw(n)` : Set width.
  - `setprecision(n)` : Set decimal precision.

-
  ```
  cout << setw(10) << 123;
  ```

3.6. Pointers and Constants

Type	Syntax	Meaning
Pointer to Constant	`const int *ptr;`	Value pointed to cannot be changed via pointer
Constant Pointer	`int *const ptr;`	Pointer cannot point to another address
Constant Pointer to Const	`const int *const ptr;`	Neither pointer nor value can be changed

3.7. Type Cast Operator

- **C-style:** `(type)variable`
- **C++ style:** `static_cast<type>(variable)`

3.8. Expressions and Their Types

- **Arithmetic:** `a + b - c`
- **Relational:** `a > b`
- **Logical:** `a && b`
- **Assignment:** `a = b`

3.9. Special Assignment Expressions

- **Chained Assignment:** `a = b = c = 0;`

3.10. Implicit Conversions

- Automatic type conversion by compiler (e.g., `int` to `float`).

3.11. Operator Precedence

- Determines order of evaluation in expressions.
- Example: `*` and `/` have higher precedence than `+` and `-`.

---

4. Functions in C++

4.1. The Main Function

- **Syntax:** `int main() { ... }`
- **Returns:** Integer value (usually `0` on success).

4.2. Function Prototyping

- **Declaration:** `int add(int, int);`
- **Purpose:** Informs compiler about function name, return type, and parameters.

4.3. Call by Reference

- Function receives reference to argument, can modify original variable.

```
vvoid swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

4.4. Return by Reference

- Function returns a reference, allowing direct modification of original variable.

```
int& max(int &a, int &b) {
    return (a > b) ? a : b;
}
```

4.5. Inline Functions

- Suggests compiler to replace function call with function code.

```
inline int square(int x) { return x * x; }
```

4.6. Default Arguments

- Parameters can have default values.

```
void display(int x, int y = 10);
```

4.7. Const Arguments

- Prevents modification of argument inside function.

```
void show(const int x);
```

4.8. Function Overloading

- Multiple functions with same name but different parameter lists.

```
int sum(int a, int b);
float sum(float a, float b);
```

---

5. Classes and Objects

5.1. Specifying a Class

- **Syntax:**

```
class Box {
    int length;
public:
    void setLength(int l) { length = l; }
    int getLength() { return length; }
};
```

5.2. Defining Member Functions

- **Inside Class:** Implicitly inline.
- **Outside Class:** Use scope resolution.

```
void Box::setLength(int l) { length = l; }
```

5.3. Making an Outside Function Inline

- Use `inline` keyword before definition outside class.

5.4. Nesting of Member Functions

- One member function calls another within the same class.

5.5. Private Member Functions

- Only accessible by other member functions.

5.6. Arrays within a Class

- Data members can be arrays.

```
class Sample {
    int arr[10];
};
```

5.7. Memory Allocation for Objects

- **Static:** At compile time.
- **Dynamic:** Using `new` operator.

5.8. Static Data Members

- Shared by all objects of the class.
- Declared with `static`, defined outside class.

5.9. Static Member Functions

- Can access only static data members.

5.10. Arrays of Objects

- Declare like normal arrays.

```
Student s[10];
```

5.11. Objects as Function Arguments

- Pass by value or reference.

5.12. Friendly Functions

- Declared with `friend` keyword.
- Can access private/protected data.

5.13. Returning Objects

- Function can return an object.

```
Box add(Box b1, Box b2) { ... }
```

5.14. Const Member Functions

- Cannot modify data members.

```
void display() const;
```

5.15. Pointers to Members

- Pointer to data member: `int Class::*ptr = &Class::member;`
- Pointer to member function: `void (Class::*fptr)() = &Class::func;`

---

6. Constructors and Destructors

6.1. Constructors

- Special member functions with same name as class.
- No return type.
- Called automatically when object is created.
- Used to initialize objects.

6.2. Parameterized Constructors

- Constructors with parameters.

```
Box(int l, int b, int h) { ... }
```

6.3. Multiple Constructors (Constructor Overloading)

- More than one constructor with different parameter lists.

6.4. Constructors with Default Arguments

- Parameters can have default values.

6.5. Copy Constructor

- Initializes object using another object of same class.

```
Box(const Box &b) { ... }
```

6.6. Const Objects

- Declared as `const`.
- Can only call `const` member functions.

6.7. Destructors

- Special member function with `~` prefix.
- Called automatically when object is destroyed.
- Used for cleanup.

Summary Table: OOP Pillars (For Quick Revision)

Pillar	What it Means in C++	Example/Code
Encapsulation	Data + functions in class, access control	`private`, `public`
Abstraction	Hide details, show essentials	Public interface only
Inheritance	Reuse via base/derived classes	`class B : public A {}`

Polymorphism	Many forms: overloading, overriding	`virtual` functions