Access Specifiers

Determines how base class members are inherited:

- `public` **inheritance**: (Most common)
  - Base `public` → Derived `public`
  - Base `protected` → Derived `protected`
  - Base `private` → Inaccessible
- `protected` **inheritance**:
  - Base `public` → Derived `protected`
  - Base `protected` → Derived `protected`
- `private` **inheritance**:
  - Base `public` → Derived `private`
  - Base `protected` → Derived `private`

Constructor / Destructor Order

- **Construction**: Base class constructor runs *first*, then derived class constructor.
- **Destruction**: Derived class destructor runs *first*, then base class destructor (reverse order).

Calling Base Constructor

Use the member initializer list to pass arguments to the base constructor.

```
class Base {
public:
    Base(int x) { /* ... */ }
};

class Derived : public Base {
public:
    Derived(int x, int y) : Base(x) {
        // y is used by Derived
    }
};
```

Virtual Functions

A function in the base class declared with `virtual`. The dynamic type (the actual object) determines which function version is called at runtime, not the static type (the pointer).

```
class Animal {
public:
    virtual void speak() {
        std::cout << "???";
    }
    // ALWAYS make base class destructors virtual!
    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    // 'override' is optional but recommended:
    compiler checks that it actually overrides a base
    virtual function.
    virtual void speak() override {
        std::cout << "Woof!";
    }
};
```

Usage:

```
Animal* p = new Dog();
p->speak(); // Calls Dog::speak()
delete p;   // Calls Dog::~Dog(), then
            // Animal::~Animal()
```

Abstract Classes & Pure Virtual

A class with one or more **pure virtual functions** (`= 0`) is an **abstract class** and cannot be instantiated. It *must* be subclassed.

```
class Shape { // Abstract Base Class
public:
    // Pure virtual function
    virtual double getArea() = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double r;
public:
    // Must implement all pure
    // virtual functions
    virtual double getArea() override {
        return 3.14 * r * r;
    }
};
```

GDB (GNU Debugger)

Compile with the `-g` flag to include debug symbols.

Starting

- `gdb ./my_program` : Start GDB session.
- `gdb -tui ./my_program` : Start with text-based GUI

Essential Commands

- `run (r)`: Start program (with optional args, e.g., `r arg1`).
- `break (b) <loc>`: Set breakpoint.
- `continue (c)`: Continue execution to next breakpoint.
- `next (n)`: Step over (executes line, doesn't enter funcs).
- `step (s)`: Step into (enters function calls).
- `finish`: Step out (run until current function returns).
- `print (p) <var>`: Print value of a variable.
- `backtrace (bt)`: Show the call stack (func call history).
- `list (l)`: Show source code around current line.
- `info break`: List all breakpoints.
- `delete <num>`: Delete breakpoint by number.
- `quit (q)`: Exit GDB.

Valgrind (Memcheck)

A tool for detecting memory errors (leaks, invalid access, etc.). Compile with `-g` for useful line numbers.

The default tool is **Memcheck**.

```
valgrind [options] ./my_program
```

Most Common Flags

- `--leak-check=full`: (Essential) Shows details for
- `--log-file="report.txt"`: Write output to a file.

Interpreting Output

Look for:

- **Invalid read / Invalid write**: Accessing memory you don't own (e.g., array out of bounds, using a dangling pointer).
- **Conditional jump... depends on uninitialised value(s)**: Using a variable (e.g., in an `if` statement) before it was given a value.
- **Mismatched free() / delete / delete[]**: Using `delete` on memory from `malloc`, or `delete[]` on `new`, or `delete` on `new[]`.
- **HEAP SUMMARY**:
  - `in use at exit`: Should be 0 bytes.
  - `definitely lost`: True memory leaks. You lost all pointers to this memory. **This is the one to fix!**