OOP Cheat Sheet: Upcasting, Downcasting, Binary File Handling

Quick Revision Guide for Exam Preparation, May 2025

1 Upcasting

1.1 Concept

Upcasting converts a derived class pointer or reference to a base class pointer or reference. It is implicit and safe in C++ because a derived class inherits all base class properties, enabling polymorphic behavior through virtual functions.

1.2 Syntax

```
class Base {
public:
  virtual void func() { /* Base implementation */ }
  virtual ~Base() {} // Virtual destructor
};
class Derived : public Base {
public:
  void func() override { /* Derived implementation */ }
};
// Upcasting
Derived d;
Base* b = &d; // Implicit upcast
Base& br = d; // Reference upcast
```

1.3 Tips

- Use `virtual` functions for polymorphism.
- Always include a virtual destructor in the base class.
- Upcasting is implicit; no casting operator needed.
- Ideal for passing derived objects to functions expecting base types.

1.4 Example

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

class Shape {
public:
  virtual double area() { return 0.0; }
  virtual ~Shape() {}
};
class Circle : public Shape {
  double radius;
public:
  Circle(double r) : radius(r) {}
  double area() override { return 3.14 * radius * radius; }
};

int main() {
  Circle c(5.0);
  Shape* s = &c; // Upcasting
  cout << "Area: " << s->area() << endl; // Calls Circle::area()
  return 0;
}
```

Output: `Area: 78.5`

2 Downcasting

2.1 Concept

Downcasting converts a base class pointer or reference to a derived class pointer or reference. It is not implicit and can be unsafe unless the base pointer points to a derived object. Use `dynamic_cast` for safe downcasting with runtime type checking or `static_cast` when the type is certain.

2.2 Syntax

```
class Base {
public:
  virtual ~Base() {} // Required for dynamic_cast
};
class Derived : public Base {
public:
  void specificFunc() { /* Derived-specific method */ }
};
// Downcasting
Base* b = new Derived;
Derived* d = dynamic_cast<Derived*>(b); // Safe downcast
if (d) d->specificFunc();
```

2.3 Tips

- Use `dynamic_cast` for polymorphic classes (requires virtual functions).
- Always check `dynamic_cast` result to avoid null pointers.
- Use `static_cast` only when type is guaranteed (faster but risky).
- Prefer virtual functions over downcasting when possible.

2.4 Example

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

class Animal {
public:
  virtual void speak() { cout << "Generic sound\n"; }
  virtual ~Animal() {}
};
class Dog : public Animal {
public:
  void speak() override { cout << "Woof\n"; }
  void fetch() { cout << "Fetching ball\n"; }
};

int main() {
  Animal* a = new Dog; // Upcasting
  a->speak(); // Output: Woof
  Dog* d = dynamic_cast<Dog*>(a); // Downcasting
  if (d) d->fetch(); // Output: Fetching ball
  return 0;
}
```

Output: `Woof`
`Fetching ball`

3 Binary File Handling

3.1 Concept

Binary file handling involves reading and writing raw bytes to files using `fstream`, `ofstream`, or `ifstream` with the `ios::binary` flag. Data (objects, arrays) is converted to `char*` using `reinterpret_cast` for precise memory representation. This is efficient for storing complex data like arrays and supports random access with `seekg`/`seekp`.

3.2 Syntax

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

// Writing object/array to binary file
ofstream out("file.bin", ios::binary);
out.write(reinterpret_cast<char*>(&object), sizeof(object));

// Reading object/array from binary file
ifstream in("file.bin", ios::binary);
in.read(reinterpret_cast<char*>(&object), sizeof(object));

// Random access
in.seekg(position); // Move read pointer
out.seekp(position); // Move write pointer
```

3.3 Tips

- Always use `ios::binary` to avoid text formatting issues (e.g., newline conversions).
- Use `reinterpret_cast<char*>` to convert objects/arrays to `char*` for binary I/O.
- For arrays, write/read the entire array using `sizeof(array)` or loop for dynamic sizes.
- Check file status with `is_open()` before operations.
- Close files with `close()` to free resources.
- For random access, calculate positions as `index * sizeof(type)`.
- Avoid pointers in objects written to files; use fixed-size data (e.g., arrays, structs).
- Store array size in the file to read dynamic arrays correctly.

3.4 Examples

Example 1: Writing and Reading an Array

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

int main() {
  // Write array to binary file
  int arr[] = {10, 20, 30, 40};
  int size = 4;
  ofstream out("array.bin", ios::binary);
  out.write(reinterpret_cast<char*>(&size), sizeof(size)); // Write size
  out.write(reinterpret_cast<char*>(arr), size * sizeof(int));
  out.close();

  // Read array from binary file
  int readSize;
  ifstream in("array.bin", ios::binary);
  in.read(reinterpret_cast<char*>(&readSize), sizeof(readSize));
  int* readArr = new int[readSize];
  in.read(reinterpret_cast<char*>(readArr), readSize * sizeof(int));
  for (int i = 0; i < readSize; ++i) {
    cout << readArr[i] << " "; // Output: 10 20 30 40
```

```
  }
  delete[] readArr;
  in.close();
  return 0;
}
```

Example 2: Random Access for Array Elements

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

struct Record {
  int id;
  Record(int i = 0) : id(i) {}
};

int main() {
  // Write array of structs
  Record records[] = {Record(1), Record(2), Record(3)};
  int size = 3;
  ofstream out("records.bin", ios::binary);
  out.write(reinterpret_cast<char*>(&size), sizeof(size));
  out.write(reinterpret_cast<char*>(records), size * sizeof(Record));
  out.close();

  // Read specific record (e.g., second record)
  ifstream in("records.bin", ios::binary);
  int readSize;
  in.read(reinterpret_cast<char*>(&readSize), sizeof(readSize));
  in.seekg(sizeof(int) + sizeof(Record)); // Skip size + first record
  Record r;
  in.read(reinterpret_cast<char*>(&r), sizeof(Record));
  cout << "ID: " << r.id << endl; // Output: ID: 2
  in.close();
  return 0;
}
```