Unit-04

UTA018 • Polymorphism • Operator Overloading • Type Conversion

FUNCTION OVERLOADING

Use when: same operation/name, different argument types or counts.

```
int f(int x){ return x; }
double f(double x){ return x; }
int f(int x, int y){ return x+y; }
```

Approach: same name → vary parameter list → call with matching arguments → check compiler can choose one.

Trap: return type alone cannot distinguish overloads.

OVERRIDING

Use when: derived class replaces inherited behavior.

```
class Base { public: virtual void show() {} };
class Child : public Base {
 public: void show() override {}
};
```

Approach: base method → inherit → same signature in child → use override.

Remember: overloading = different parameters; overriding = matching inherited signature.

VIRTUAL / RUNTIME POLYMORPHISM

```
class Animal { public: virtual void
sound(){cout<<"Animal";}
 virtual ~Animal()=default; };
class Dog: public Animal { public: void sound() override
{cout<<"Dog";} };
Dog d; Animal* p=&d; p->sound(); // Dog
```

Approach: virtual in base → override in child → call through base pointer/reference → actual object type decides.

Clue: question asks which version runs at runtime? Check virtual.

4 ABSTRACT CLASS / PURE VIRTUAL

```
class Shape { public: virtual void draw()=0;
 virtual ~Shape()=default; };
class Square: public Shape { public: void draw() override
{cout<<"Square";} };
```

Approach: =0 → base is abstract → implement all pure virtual methods → instantiate concrete child.

Trap: Shape s; is invalid; Shape* p=□ is valid.

5 OPERATOR OVERLOADING

Use when: existing operator should work with objects.

```
Number operator+(const Number& rhs) const {
 return Number(value + rhs.value);
}
```

Member binary operator: left operand is object; one explicit right operand. **Unary member:** no explicit operand.

Friend/non-member: useful for 2 + obj; binary function receives both operands.

Restrictions: cannot invent operators or change precedence, associativity, operand count. Cannot overload ::, ., .*, ?:, sizeof.

7 TYPE CONVERSION: BUILT-IN → CLASS

Approach: converting constructor in destination class.

```
class Distance { public:
 Distance(int m) : meters(m) {}
 private: int meters; };
Distance d = 5;
```

Steps: identify source built-in type → constructor accepts it → initialize object.

Tip: use explicit to prevent unwanted implicit conversions.

8 TYPE CONVERSION: CLASS to BUILT-IN

```
operator int() const { return meters; }
```

Approach: conversion function is a class member, takes no parameters, and has no declared return type.

```
Distance d(25); int x = d;
```

9 TYPE CONVERSION: CLASS to CLASS

Approach: use a constructor in the destination class OR a conversion function in the source class.

```
class Destination {
 public: Destination(const Source& s) { /* convert */ }
};
Destination d = source;
```

Steps: identify source/destination → choose where conversion belongs → implement → verify value.

10 OVERLOAD AMBIGUITY

Cause: more than one overload is equally viable after conversions/default arguments.

Approach: list candidates → match argument types → check conversions/defaults → make call's type explicit or revise overloads.

```
show(static_cast<int>(value));
```

11 NAME HIDING

Cause: a same-named function declared in derived class may hide base overloads.

```
class Derived : public Base {
 public: using Base::show;
 void show() {}
};
```

Approach: check derived declarations → add using Base::function; if base overloads should remain visible.

12 DEFAULT ARGUMENTS

```
int add(int a, int b=10) {return a+b;}
add(5);     // 15
add(5,3);   // 8
```

Approach: defaults go on trailing parameters → test calls with/without optional arguments → check overload conflicts.

13 EXAM-TIME DEBUGGING CHECKLIST

1. Identify required behavior and select concept.
2. Check function signature and access specifiers.
3. Implement all pure virtual functions.

6 SUBSCRIPT OPERATOR []

```
int& operator[](int i) { return data[i]; }
```

Approach: member function → accept index → return selected element. Use a reference (T&) if obj[i] must be assignable.

Check: index validity / bounds.

4. Use override where appropriate.
5. Trace exact call, object type and output.
6. Check conversions, ambiguity and boundary cases.

Quick picker: same name/different parameters → overloading; child replaces behavior → overriding; base pointer chooses child method → virtual; object + operator → operator overloading; built-in/class mismatch → conversion.

Use this card as a prompt for reasoning; write and test the complete solution yourself.