Ch 6: Compile-Time Polymorphism

CSCI 330

Overview

- Functionality is determined at compile time, not runtime
- Key tools:
  - Function overloading
  - Operator overloading
  - Templates
  - CRTP (Curiously Recurring Template Pattern

Polymorphism

“The ability to hide different implementations behind a common interface, simplifying the communications among objects”
D. Taylor, 1998

Function Overloading

- Same function name, different parameter types
- Chosen by compiler based on argument types

- void draw(int);
- void draw(double);

Operator Overloading

- Customize the meaning of built=in operators
- Enables intuitive syntax with user-defined types

```
struct Point {
  int x, y;
  Point operator+(const Point& other) const {
    return {x + other.x, y + other.y};
  }
};
```

Function Templates

- Generates a version of the function for each type used
- strongly typed, efficient

```
template<typename T>
T max(T a, T b) {
  return a > b ? a : b;
}
```

Class Templates

- Like generics in Java/C#, but resolved at compile time
- Allows type-safe containers, algorithms, and abstractions

```
template<typename T>
class Container {
  T value;
public:
  Container(T v) : value(v) {}
  T get() const { return value; }
};
```

Compile-Time vs Runtime Polymorphism

Feature	Compile-Time	Runtime
Flexibility	Limited to known types	Supports unknown types
Performance	Optimized	Slight dispatch cost
Type Safety	Strong	Weaker (base interface only)
Inheritance needed?	No	Yes
Use Case	Algorithms, utilities	Interfaces, plugins

CRTP – Advanced Template

```
template<typename Derived>
struct Base {
  void interface() {
    static_cast<Derived*>(this)->implementation();
  }
};

struct Derived : Base<Derived> {
  void implementation() {
    std::puts("Derived implementation");
  }
};
```

Template Specialization

- Allows for custom behavior for specific types

```
template<typename T>
void print_type();

template<>
void print_type<int>() {
  std::puts("int");
}
```

Type traits and constexpr

- Use templates + metaprogramming to query or manipulate types

```
template<typename T>
void describe() {
  if constexpr (std::is_integral_v<T>) {
    std::puts("Integral type");
  } else {
    std::puts("Non-integral type");
  }
}
```