OOP51-CPP. Do not slice derived objects

An object deriving from a base class typically contains additional member variables that extend the base class. When by-value assigning or copying an object of the derived type to an object of the base type, those additional member variables are not copied because the base class contains insufficient space in which to store them. This action is commonly called *slicing* the object because the additional members are "sliced off" the resulting object.

Do not initialize an object of base class type with an object of derived class type, except through references, pointers, or pointer-like abstractions (such as `std::unique_ptr, or std::shared_ptr`).

Noncompliant Code Example

In this noncompliant code example, an object of the derived `Manager` type is passed by value to a function accepting a base `Employee` type. Consequently, the `Manager` objects are sliced, resulting in information loss and unexpected behavior when the `print()` function is called.

```
#include <iostream>
#include <string>

class Employee {
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<(std::ostream &os, const Employee &e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  Employee assistant;

protected:
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t" << get_assistant() << std::
endl;
  }

public:
  Manager(const std::string &name, const Employee &assistant) : Employee
(name), assistant(assistant) {}
  const Employee &get_assistant() const { return assistant; }
};

void f(Employee e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

When `f()` is called with the `designer` argument, the formal parameter in `f()` is sliced and information is lost. When the object `e` is printed, `Employee::print()` is called instead of `Manager::print()`, resulting in the following output:

```
Employee: Jane Doe
```

Compliant Solution (Pointers)

Using the same class definitions as the noncompliant code example, this compliant solution modifies the definition of `f()` to require raw pointers to the object, removing the slicing problem.

```
// Remainder of code unchanged...

void f(const Employee *e) {
  if (e) {
    std::cout << *e;
  }
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(&coder);
  f(&typist);
  f(&designer);
}
```

This compliant solution also complies with EXP34-C. Do not dereference null pointers in the implementation of `f()`. With this definition, the program correctly outputs the following.

```
Employee: Joe Smith
Employee: Bill Jones
Manager: Jane Doe
Assistant:
        Employee: Bill Jones
```

Compliant Solution (References)

An improved compliant solution, which does not require guarding against null pointers within `f()`, uses references instead of pointers.

```
// ... Remainder of code unchanged ...

void f(const Employee &e) {
  std::cout << e;
}

int main() {
```

```
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

Compliant Solution (Noncopyable)

Both previous compliant solutions depend on consumers of the `Employee` and `Manager` types to be declared in a compliant manner with the expected usage of the class hierarchy. This compliant solution ensures that consumers are unable to accidentally slice objects by removing the ability to copy-initialize an object that derives from `Noncopyable`. If copy-initialization is attempted, as in the original definition of `f()`, the program is ill-formed and a diagnostic will be emitted. However, such a solution also restricts the `Manager` object from attempting to copy-initialize its `Employee` object, which subtly changes the semantics of the class hierarchy.

```
#include <iostream>
#include <string>

class Noncopyable {
  Noncopyable(const Noncopyable &) = delete;
  void operator=(const Noncopyable &) = delete;

protected:
  Noncopyable() = default;
};

class Employee : Noncopyable {
  // Remainder of the definition is unchanged.
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<(std::ostream &os, const Employee &e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  const Employee &assistant; // Note: The definition of Employee has been
modified.

  // Remainder of the definition is unchanged.
protected:
```

```
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t" << get_assistant() << std::
endl;
  }

public:
  Manager(const std::string &name, const Employee &assistant) : Employee
(name), assistant(assistant) {}
  const Employee &get_assistant() const { return assistant; }
};

// If f() were declared as accepting an Employee, the program would be
// ill-formed because Employee cannot be copy-initialized.
void f(const Employee &e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

Noncompliant Code Example

This noncompliant code example uses the same class definitions of `Employee` and `Manager` as in the previous noncompliant code example and attempts to store `Employee` objects in a `std::vector`. However, because `std::vector` requires a homogeneous list of elements, slicing occurs.

```
#include <iostream>
#include <string>
#include <vector>

void f(const std::vector<Employee> &v) {
  for (const auto &e : v) {
    std::cout << e;
  }
}

int main() {
  Employee typist("Joe Smith");
  std::vector<Employee> v{typist, Employee("Bill Jones"), Manager("Jane
Doe", typist)};
  f(v);
}
```

Compliant Solution

This compliant solution uses a vector of `std::unique_ptr` objects, which eliminates the slicing problem.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

void f(const std::vector<std::unique_ptr<Employee>> &v) {
  for (const auto &e : v) {
    std::cout << *e;
  }
}

int main() {
  std::vector<std::unique_ptr<Employee>> v;

  v.emplace_back(new Employee("Joe Smith"));
  v.emplace_back(new Employee("Bill Jones"));
  v.emplace_back(new Manager("Jane Doe", *v.front()));

  f(v);
}
```

Risk Assessment

Slicing results in information loss, which could lead to abnormal program execution or denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP51-CPP	Low	Probable	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.3	**OOP-02, JSF-117_a**	
PRQA QA-C++	4.1	**3072**	

Related Vulnerabilities

Search for other vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	ERR61-CPP. Catch exceptions by lvalue reference CTR56-CPP. Do not use pointer arithmetic on polymorphic objects
SEI CERT C Coding Standard	EXP34-C. Do not dereference null pointers

Bibliography

[Dewhurst 2002]	Gotcha #38, "Slicing"
[ISO/IEC 14882-2014]	Subclause 12.8, "Copying and Moving Class Objects"
[Sutter 2000]	Item 40, "Object Lifetimes—Part I"