Comp151

Polymorphism: Virtual Functions

Example: Global print Function By Reference

- Because of the substitution principle, you may want to write a global print function for Person and its derived classes as follows:

```
void print_by_ref(const Person& person) { person.print(); }
int main()
{
   Person mouse("Mickey", "Disney World", arts);
   Teacher einstein("Albert Einstein", "USA", physics, professor);
   Student plato("Plato", "Greece", philosophy);
   plato.enroll_course(COMP151);
   print_by_ref(mouse);
   print_by_ref(einstein);
   print_by_ref(plato);
}
```

Example: Global print Function By Pointer

- Or by a pointer argument:

```
void print_by_ptr(const Person* person) { person->print(); }
int main()
{
   Person mouse("Mickey", "Disney World", arts);
   Teacher einstein("Albert Einstein", "USA", physics, professor);
   Student plato("Plato", "Greece", philosophy);
   plato.enroll_course(COMP151);
   print_by_ptr(&mouse);
   print_by_ptr(&einstein);
   print_by_ptr(&plato);
}
```

Example: Output

```
------ Person details ------
Name: Mickey
Addr: Disney World
Dept: 0
------ Person details ------
Name: Albert Einstein
Addr: USA
Dept: 1
------ Person details ------
Name: Plato
Addr: Greece
Dept: 2
```

Oops!

Example: This Is What We Want

- (assume: enum Department { arts, physics, philosophy, ...})

----- Person details -----
Name: Mickey
Addr: Disney World
Dept: 0

----- Teacher details -----
Name: Albert Einstein
Addr: USA
Dept: 1
Rank: Full Professor

----- Student details -----
Name: Plato
Addr: Greece
Dept: 2
Enrolled in:
COMP151

Static or Early Binding

We want

```
void print_by_ref(const Person& person) { person.print(); }
```

to call Student::print() when a Student object is passed as argument, to call Teacher::print() when a Teacher object is passed as argument, and so forth.

- However, when the compiler generates the function call in the line, it looks at the static type of person which is const Person&, so the method Person::print() is called.
- Static binding: the binding (association) of a function name to the appropriate method is done by a static analysis of the code at compile time based on the static (or declared)type of the object used in the call.

The fact that the pointer can point to, (or the reference is actually a reference of) an object of a derived class is not considered.

Dynamic or Late Binding

- By default, C++ uses static binding. (Same as C, Pascal, and FORTRAN.)
- In C++, another type of binding called dynamic binding is supported through virtual functions. (Same as Java, Smalltalk, Lisp/Scheme.)
- When dynamic binding is used, the method to be called is selected using the actual type of the object in the call, i.e., print_by_ref(mouse) (a Person object) would call Person::print(), print_by_ref(plato) (a Student object) would call Student::print(), and print_by_ref(einstein) (a Teacher object) would call Teacher::print().

Note that the possible object types do not need to be known at the time that the function call is being compiled!

Virtual Functions

- A virtual function is a method that is declared using the **virtual** keyword in the class definition (but not in the method implementation, if it is outside the class).

```
class Person {
    string name;
    string address;
    Department dept;
public:
    virtual void print() const;
…
};

void Person::print() const {
    cout << "--- Person details ---" << endl;
    cout << "Name: " << name << endl;
    cout << "Addr: " << address << endl;
    cout << "Dept: " << dept << endl;
}
```

Virtual Functions…

- Once a method is declared virtual in the base class, it is automatically virtual in all directly or indirectly derived classes.
- Even though it is not necessary to use the virtual keyword in the derived class, it is good style to do so, because it improves the readability of header files.

```
class Student: public Person {
    Course* enrolled;
    int num_courses;
public:
    virtual void print() const;
…};
```

- Calls to virtual functions are a little bit slower than normal function calls. The difference is extremely small, and is worth worrying only if you're writing speed-critical code.
- Objects with one or more virtual functions take a bit more space than normal objects. The difference is typically 4 bytes per object, and is worth worrying about if you're writing memory-intensive code.

Polymorphism

poly = multiple *morphos* = shape

- Polymorphism in C++ means that we can work with objects without knowing their precise type at compile time:
- In: `void print_by_ptr(const Person* person) { person->print(); }` the type of the object pointed to by person is not known to the programmer writing this code, nor to the compiler. We say that person exhibits polymorphism, because the object can take on multiple shapes.
- Polymorphism allows us to write programs that behave correctly even when used with objects of derived classes.
- A pointer or reference must be used to have polymorphism. If call-by-value is used, no polymorphism can happen (WHY?).

```
void print_by_value(Person person) { person.print(); }
/* wrong use */
```

Example: Virtual Function

```
#include "people.hpp"
// class Person { ... virtual void print() const; }
// class Teacher: public Person { ... virtual void print() const; }
// class Student: public Person { ... virtual void print() const; }
// class PG Student: public Student { ... virtual void print() const; }
void print_by_ref(const Person& person) { person.print(); }
void print_by_ptr(const Person* person) { person->print(); }
int main()
{
    const int N = 4;
    Person p("Mickey", "Disney World", arts);
    Teacher t("Albert Einstein", "USA", physics, professor);
    Student s("Plato", "Greece", philosophy);
    s.enroll_course("COMP151");
    PG Student g(‘John", "HK", computer sci, "AI");
    g.enroll_course("COMP527");
    Person* x[N]; x[0] = &p; x[1] = &t; x[2] = &s; x[3] = &};
    for (int j = 0; j < N; ++j) // by pointer to the base class
                x[j]->print();
    cout << endl;
    for (int j = 0; j < N; ++j) // by pointer to the base class
                print_by_ptr(x[j]);
    cout << endl;
    for (int j = 0; j < N; ++j) // by reference to the base class
                print_by_ref(*(x[j]));
}
```

Another Example: Non Virtual Function

```
// NonVirtual.cpp
# include <iostream>
using namespace std;

class Base {
   public: void Display() const; } ;
class X: public Base {
   public: void Display() const; };
class Y: public Base {
   public: void Display() const; };
void Base::Display() const { cout << "Inside Base::Display\n"; }
void X::Display() const { cout << "Inside X::Display\n"; }
void Y::Display() const { cout << "Inside Y::Display\n"; }
void Display_by_ref(const Base& b) { b.Display(); }
void Display_by_pntr(Base* b) { b->Display(); }
```

```
main()
{
  Base base; X x; Y y;
  Base* A[3]; A[0] = &base; A[1] = &x; A[2]= &y;
  for (int j = 0; j<3; j++)  // Pointers
    A[j]->Display();
  cout << endl;
  for (int j = 0; j<3; j++)  // Passing references
    Display_by_ref(*(A[j]));
  cout << endl;
  for (int j = 0; j<3; j++)  // Passing pointers
    Display_by_pntr(A[j]);
}
```

OUTPUT

```
------
Inside Base::Display
Inside Base::Display
Inside Base::Display

Inside Base::Display
Inside Base::Display
Inside Base::Display

Inside Base::Display
Inside Base::Display
Inside Base::Display
```

Another Example: Virtual Function

```
// Virtual.cpp
# include <iostream>
using namespace std;

class Base {
   public: virtual void Display() const; } ;
class X: public Base {
   public: virtual void Display() const; };
class Y: public Base {
   public: virtual void Display() const; };
void Base::Display() const { cout << "Inside Base::Display\n"; }
void X::Display() const { cout << "Inside X::Display\n"; }
void Y::Display() const { cout << "Inside Y::Display\n"; }
void Display_by_val(Base b) { b.Display(); }
void Display_by_ref(const Base& b) { b.Display(); }
void Display_by_pntr(Base* b) { b->Display(); }
```

```
main()
{
  Base base; X x; Y y;
  Base* A[3]; A[0] = &base; A[1] = &x; A[2]= &y;
  for (int j = 0; j<3; j++)
    A[j]->Display();              // Pointers
  cout << endl;
  for (int j = 0; j<3; j++)
    Display_by_val(*(A[j]));    // Not using virtual functions
  cout << endl;
  for (int j = 0; j<3; j++)
    Display_by_ref(*(A[j]));    // Passing references
  cout << endl;
  for (int j = 0; j<3; j++)
    Display_by_pntr(A[j]);      // Passing pointers
}
```

Output

Inside Base::Display
Inside X::Display
Inside Y::Display

Inside Base::Display
Inside Base::Display
Inside Base::Display

Inside Base::Display
Inside X::Display
Inside Y::Display

Inside Base::Display
Inside X::Display
Inside Y::Display