Programming Languages

Spring 1994
1 hour

Write clear but brief answers to the following questions in the space provided.

1. *Implementation of parameter passing.*
   In the original version of Pascal, the formal parameters of a function had to be given types. For example, a procedure with an integer parameter would generally be written

```
procedure P(x : integer);
   begin
      ...
   end
```

and a procedure with a record (struct) parameter might look like

```
type entry = record
       name : string
       age  : integer
end;
procedure Q(x : entry);
   begin
      ...
   end
```

However, for function parameters, the type of a function had the form `function` ⟨*name*⟩ : ⟨*return type*⟩ in the original Pascal report. For example, a procedure accepting a function parameter might begin

```
procedure R(function f:real);
```

This does not tell the argument type of the function parameter. This is not as descriptive as the form of function type used in later versions of Pascal, or the form of function type used in other programming languages. Pascal allows both pass-by-value and pass-by-reference parameters.

(a) (5 points) Explain how the type of a parameter would be used in code generation, not type checking, for each of the two parameter-passing mechanisms, pass-by-value and pass-by-reference.

(b) (3 points) Can either or both of these two parameter-passing mechanisms be implemented for integer and record parameters?

(c) (3 point) Why is it possible to implement pass-by-reference for function parameters with only the type information given?

2. *Design of C++.*
In *An Overview of C++*, Stroustrop says that C++ achieves its aims by "abandoning claims to perfection." For each of the following features, explain why C++ could be considered "imperfect" but practical.

   (a) Absence of garbage collection.

   (b) Allocation of objects on the run-time stack.

   (c) Distinction between virtual and non-virtual members. (In contrast, Smalltalk makes no distinction between methods that can be redefined in subclasses and methods that cannot.)

3. *Subtyping and inheritance.*

Definition: Type A2 is a *subtype* of type A1 if, in any context in which elements of type A1 can appear in a type-safe way, elements of type A2 are also type-safe.

For example, suppose we are given class types B (Base) and D (Derived) such that D is publicly derived from B. Then given the following declarations:

```
B* f1();
D* f2();
```

The type of `f2` is a subtype of the type of `f1`, because anything that can accept a B* can also accept a D*. Such subtypes are called *function subtypes.*

(a) As shown in the example above, we can create function subtypes by specializing (making more restrictive) the return type. Can we also create function subtypes by changing the argument types? If not, why not? If so, why, and give an example.

Suppose we are given the following two class declarations:

```
class c1 {
public:
        int x, y;
        virtual B* f(short, long);
        virtual long g(short);
};

class c2 {
public:
        int x, y;
        virtual B* f(short, long);
        virtual long g(short);
        virtual short h();
};
```

(b) According to the above definition, `c2` is a subtype of `c1`. Why?

(c) In C++, the type-checker does not consider `c2` a subtype of `c1`. Why not?

(d) Modify `c2` so that it is recognized as a subtype of `c1` in C++.

(e) Given the original definitions of `c1` and `c2` above, objects of type `c2` can safely be used in place of objects of type `c1` in C++ by using type casts. Explain.

(f) Without adding any new methods or instance variables to `c1` or `c2`, and without changing the declarations of any of the individual methods or instance variables in `c1` and `c2`, change the declarations of `c1` and `c2` such that objects of type `c1` can no longer safely be used in place of objects of type `c2`, even using type casts, and explain why not.

(g) In general, what must be true about two classes d1 and d2 for a type cast to allow objects of class d2 to be used safely as objects of class d1?

(h) Suppose you define the above classes c1 and c2 in Smalltalk. In this case, c2 is always a subtype of c1. Explain why this is true for Smalltalk and not for C++.

(i) Suppose that c2 is changed so that one of its methods now belongs to a subtype of the corresponding method in c1, rather than having the exact same type, as in the following example:

```
class c1 {
public:
        int x, y;
        virtual B* f(short, long);
        virtual long g(short);
};

class c2 {
public:
        int x, y;
        virtual D* f(short, long);
        virtual long g(short);
        virtual short h();
};
```

Is c2 still a subtype of c1? Why or why not?

4. *Polymorphism and representation of data.*

(a) Both C++ and ML support functions with parameterized types. C++'s template mechanism, however, yields a separate copy of the code for each template instantiation, while ML maintains only a single copy of the code. What property of ML allows it to avoid code replication, and why might the designers of C++ have chosen not to support this property?

(b) Besides templates, what features of C++ support polymorphism, and in what way do these features support it?

(c) Smalltalk, C++, and ML all allow programmers to write functions that operate on multiple kinds of objects. Compare and contrast Smalltalk's approach to the approach taken by ML and C++.