The Semantics of the C++ Programming Language

Charles Wallace*

August 31, 1995

1 Introduction

In this chapter we extend the evolving algebra presented in [GH] to give formal operational semantics for the C++ programming language. The evolving algebra of [GH] is a specification for the C programming language; here we propose modifications to it to accommodate the features of C++. We refer the reader to [GH] for a description of the C algebra, and to [Gu] for an introduction to evolving algebras. [KR] describes the ANSI standard for the C language, on which the algebra of [GH] is based. We assume the informal specification for C++ in [ES] as guidelines for our semantics. Knowledge of C++ will ease comprehension but is not necessary, as we shall explain the new features of C++ and illustrate their use with examples as we proceed.

The C++ programming language is designed to be an extension of C, retaining all of C's language facilities and adding new ones. On a syntactic level, the differences between C and C++ consist entirely of language constructs allowable in C++ but not in C. Our algebra for C++ extends the rules for C in [GH] to capture the new features of C++. Many C++ features do not require any changes at all to the rules. Such features affect only static information about the program, determined at compile time and never changed during the running of the program. In addition to proposing changes to the rules, we shall discuss the C++ features which do not require any rule changes and explain how we can handle them.

1.1 Outline

The new features of C++ support the *object-oriented* programming paradigm. The term *object* can be defined simply as the instantiation of a type. This approach to programming is a synthesis of several principles. An object is accessed and modified through a set of *operations* specified within the definition of the object's type. New types can be defined in terms of preexisting types through *inheritance*. Finally, access to an object's data is localized through *encapsulation*.

The features required to implement encapsulation and inheritance are presented in section 2, while those required to combine type and operation definitions are presented in section 3. Section 4 deals with the features supporting creation and destruction of objects. In sections 5 and 6 we discuss extensions that are not object-oriented in nature; section 5 concerns overloading and parameterized type definitions, and section 6 covers the remaining extensions.

For the sake of readability, we define a set of macros for commonly used rule expressions. The definitions of these macros appear in A.

1.2 Acknowledgments

I would like to thank Yuri Gurevich for inspiring me to write this chapter and guiding me during its development. I would also like to thank Jim Huggins, Solomon Foster and Jon Rossie for their helpful comments.

*Partially supported by NSF grant #029862 and ONR grant #028355. The author's address is: EECS Department, University of Michigan, Ann Arbor, MI 48109-2122. Electronic mail address: wallace@eecs.umich.edu.

2 Class structure and encapsulation

The central notion of the object-oriented programming paradigm is the encapsulation of data types and operations associated with them. Encapsulation ensures that the data stored within an object is accessed only by the operations associated with the object's type. This localization of access is conducive both to data security and to good programming style. Encapsulation is achieved in C++ through the `class` construct, which defines aggregate types similar to `struct` types in C. A `class` type combines the components of a programmer-defined data type with the functions and operators to manipulate it. The notion of a class is introduced in section 2.1. The level of localization of access is achieved by specifying access status for the components of the type; access status is discussed in section 2.4.

In object-oriented programming, redundant code may be eliminated by allowing one type to *inherit* the data structure and operations of another. The inherited structure and operations may then be modified or extended to suit the new type. Inheritance is called *derivation* in C++; the features supporting class derivation are discussed in sections 2.2 and 2.3.

2.1 Classes

C++ introduces a new keyword `class` which indicates the definition of a new type. This is almost identical in functionality to the C keyword `struct`. Both define types whose instantiations are contiguous sequences of ordered fields (*members*) in memory; in C++, both may have operations (*member functions*) associated with them. The only difference between the two is in the default *access status* assigned to their fields.[1] As we shall see in section 2.4, access status is itself a C++ extension of a purely syntactic nature; thus we can treat `class` types in the same way we treat `struct` types in the C algebra, with no rule modifications necessary. We hereby adopt C++ terminology: we shall use the term *class* to refer to both `struct` and `class` types and the term *member* as a synonym for *field*. In addition, we shall use the term *object* to refer to a contiguous area of memory serving as an instantiation of a particular type; in particular, the term *class object* refers to an area allocated as an instantiation of a `class` or `struct` type.

2.2 Derived classes

In addition to the members included explicitly in its definition, a class may inherit the members of a set of other classes. A class that inherits members is said to be *derived*; the classes whose members it inherits are its *base* classes. The base classes of a derived class are specified in its definition. For example, if we define the class `person` containing the members `name` and `age`:[2]

```
// definition of (nonderived) class "person"
class person                                                    // (no members inherited)
{ char name[25];                                                // person's name (string)
  int age;                                                      // person's age (integer)
  void printPerson(); };                                // function to print person's name and age
```

we can add the derived classes `professor` and `student`:

```
// definition of derived class "professor"
class professor:  person                                // (inherits members of class "person")
{ int salary;                                                   // professor's salary (integer)
  void printProfessor(); };                               // function to print professor's info

// definition of derived class "student"
```

[1] The default access status is *public* for a `struct` and *private* for a `class` type. The default status is assigned to a field if no status is specified.

[2] In C++, a pair of slash characters (//) indicates the beginning of a one-line comment. Our C++ examples include comments for clarificational purposes; the text of these comments should not be confused with C++ code.

```
class student:  person                                    // (inherits members of class "person")
{ int year;                                                        // student's year (integer)
  float GPA;                                                        // student's GPA (decimal)
  void printStudent(); };                                  // function to print student's info
```

Professors and students in the real world are individuals with names and ages, as well as attributes specific to professors and students. The classes `professor` and `student` represent this by inheriting the members of the class `person`. Both derived classes contain `name`, `age` and `printPerson`, the members of their base class; in addition, the class `professor` contains the members `salary` and `printProfessor`, while the class `student` contains `year`, `GPA` and `printStudent`.

From these two derived classes we can create another derived class, `teachingAssistant`:

```
// definition of derived class "teachingAssistant"
class teachingAssistant:  professor, student
// (inherits members of classes "professor" and "student")
{ professor* worksFor;                    // professor that TA works for (ptr to "professor" object)
int section;                                                  // section that TA teaches (integer)
void printTA(); };                                                 // function to print TA's info
```

In the real world, a teaching assistant is a single individual with attributes of both a professor and a student. We represent this via *multiple inheritance*: class `teachingAssistant` contains the members of both `professor` and `student`, as well as the members `worksFor`, `section` and `printTA`.

An object of a nonderived class consists of a sequence of members arranged contiguously in memory; the members are arranged according to the order in which they appear in the class definition. An object of a derived class also consists of a sequence of members; some members are inherited from the base classes, and some are defined in the derived class itself. Unlike nonderived classes, there is no way of determining the ordering of a derived class' members from the class definition. In particular, the relative order of base- and derived-class members is implementation-dependent; base members may precede derived members or *vice versa*.

Nevertheless, the order of members in a derived class is fixed for all objects of the class at compile time. Hence we do not need to distinguish between derived and nonderived classes in our semantics; a base class and a class derived from it are distinct elements in the *types* universe. As with objects of nonderived classes, accessing a member of a derived-class object simply involves a constant offset from the memory location of the object.

While the introduction of derived classes does not require any rule changes, we must point out some issues regarding pointer casting. Given a class `D` derived from a class `B`, a pointer to `D` can be cast to a pointer to `B`; the intended result is a reference to the `B` portion of the `D` object. This requires some pointer arithmetic if the `B` portion is not the first portion of the `D` object. The `B` portion lies at some fixed offset from the initial address of the `D` object, so this must be added to the original address before casting. For example, assume that the following pointer variables are defined:[3]

```
profPtr = new professor;
personPtr = (person*) profPtr;
```

Assume that `professor` objects are arranged with their `salary` members first. Then for `ProfPtr` to be cast to a pointer to `person`, in the second definition, its value must be incremented by the size of the `salary` member.

Casting in the opposite direction is also possible; a pointer to `B` can be cast to a pointer to `D`. The intended result is a reference to a `D` object in which the `B` object is embedded. In this case, the offset from the `D` object to the `B` object is decremented from the pointer value before casting.

This implicit pointer arithmetic is not performed if the value of the pointer before casting is `0`. Thus both pointers would be initialized to `0` in the following definitions:

[3] Casting from a derived-class pointer to a base-class pointer is done implicitly; for clarity, we perform the casting explicitly.

```
profPtr = 0;
personPtr = (person*) profPtr;
```

To account for these special cases, we make the following assumptions. For any casting task from a base-class to a derived-class pointer or *vice versa*, we add a conditional expression task which tests whether the pointer value is `0`. If the result is positive, control passes to a task which simply returns `0`. Otherwise, control passes to a pointer addition or pointer subtraction task which adds the appropriate constant offset to the pointer value.

2.3 Virtual base classes

A base class may be designated as *virtual*; this affects the way in which its members are inherited. Consider our class `teachingAssistant`, which inherits members from both `professor` and `student` classes. Since both `professor` and `student` classes in turn inherit members from the class `person`, `teachingAssistant` inherits `person`'s members from two bases. For the class `teachingAssistant` as it is currently defined, this means that the class contains two disjoint sets of `person` members. Each object of this class will have two `name` members and two `age` members, conceivably with different values. For certain applications this is desirable,[4] but if we wish to give objects of type `teachingAssistant` a single `name` and `age` value, our definition of `teachingAssistant` as it stands is unsatisfactory.

To remedy this problem, we define class `person` as a virtual class. Defining a base class `B` as virtual within the definition of `D` ensures that any class derived from `D` will contain only one set of `B`'s members. We modify our definitions of `professor` and `student`:

```
// modified definitions of classes "professor" and "student", making base class "person" virtual
class professor:  virtual person
{ int salary;
  void printProfessor(); };

class student:  virtual person
{ int year;
  float GPA;
  void printStudent(); };
```

With `person` defined as a virtual base class, the class `teachingAssistant` still inherits the members of `person`, but objects of class `teachingAssistant` will contain only a single `name` member and a single `age` member.

Virtual base classes do not require any changes to the algebra. The virtual status of a base class affects only static information about a class subsequently derived from it: namely, the sequence of members it contains. As the number and types of a class' members are determined at compile time, and the effect of a class' virtual status extends only to this static information, the introduction of virtual base classes does not require any changes to the algebra rules.

[4] For an example in which duplication of inherited members is desirable, consider a derived class representing a research project between a professor and a student:

```
// definition of class "researchProject"
class researchProject:  professor, student
{ char topic[25];                                                    // research topic
  int funding; };                                                 // amount of funding for research
```

This class contains information representing a professor and a student, two distinct people. Here it is preferable to have separate copies of the `person` members; the `name` and `age` members corresponding to the professor and student will have distinct values. Thus the class `person` should be defined as nonvirtual in this case.

2.4 Access control

Class members may be restricted in terms of their *accessibility*, *i.e.*, the set of functions which may access them. A *private* member may only be accessed by a member function of the class in which it is defined; a function that is not a member of any class or is the member of a different class, even a derived class which inherits the private member, may not refer to it. A *protected* member is less restricted: it may be accessed by a member function of any class in which it is defined or inherited. A *public* member may be accessed by any function, regardless of the function's class membership. For example, let us assign private status to the `name` and `age` members of class `person`:

```
// modified definition of class "person," specifying access status
virtual class person
{ private:
  char name[25];
  int age;
  public:
  void printPerson(); };
```

Since `name`, `age` and `printPerson` are defined in the class `person`, the private status of `name` and `age` does not prevent `printPerson` from accessing these members:[5]

```
// definition of "printPerson" function for class "person"
person::printPerson()
// print the "name" and "age" members of the "person" object
{ output("name:", name);                                    // note reference to "name"
  output("age:", age); }                                     // note reference to "age"
```

On the other hand, `printProfessor` is not defined in the same class as `name` or `age`, so it cannot access these members. If we had assigned `name` and `age` protected status, `printProfessor` would have been able to access them, as `printProfessor`'s class `professor` is a derived class containing `name` and `age` members.

Access to a class' private members may be granted to nonmember functions by giving them *friend* status within the class definition. For example, we can define a global version of our `printPerson` function that is not a member of the `person` class:

```
// definition of global function "globalPrintPerson"
void globalPrintPerson(person p)
{ output("name:", p.name);                                  // reference to "name"
  output("age:", p.age); }                                   // reference to "age"
```

For the nonmember function `globalPrintPerson` to access the private members `name` and `age`, we must declare it as a friend to `person` within the class definition:

```
// modified definition of class "person,"
// allowing function "globalPrintPerson" to access private members
virtual class person
{ private:
  char name[25];
  int age;
  public:
  void printPerson();                                       // member function
  friend void globalPrintPerson(person); };                 // global function
```

[5] In this and following examples, we assume that the function `output` simply takes a sequence of arguments, of any number, and sends their values to an output device. We do not define the function explicitly.

Access status is a purely syntactic feature; since each member's status is assigned in the definition of the class and cannot be changed, access restrictions can be enforced at compile time and need not be enforced later. A member's status has no further effect on either the member itself or the functions which access it, so no algebra rules need to be changed to accommodate this feature.

2.5 Scope resolution operator

In both C and C++, names may differ in their scope. In C, a name may be either global or local; in C++, the situation is more complex, as a nonglobal name may have scope over any of a number of nested classes. The possibility of overlapping scopes leads to potential ambiguity. For example, let us add a member `course` to the class `professor`; `course` will itself be a class, containing the members `name`, `studentsEnrolled` and `print`:

```
// modified definition of class "professor," with member "course"
class professor:  private person
{ private:
  int salary;
  class course                                              // course that professor teaches
  { private:
    char name[25];                                               // name of course (string)
    int studentsEnrolled;                                     // number of students (integer)
    public:
      void print(); };                                   // function to print info about course
    public:
    void nonvirtualPrint();
    void virtualPrint(); };
```

The class `professor` contains two instances of the member name `name`: one inherited from the base class `person`, and one nested inside the class `course`. Both instances have scope over the nested class `course`. If we introduce a global variable `name`,

```
// definition of global variable "name"
char name[25];                                                          // name of university
```

we now have three identical names with scope over the class `course`.[6] The member function `print` of class `course` refers to `name`:

```
// definition of "print" function for class "course"
void professor::course::print()
{ output("course name:", name);                                         // reference to "name"
  output("students enrolled:", studentsEnrolled); }
```

The identifier `name` here could conceivably refer to two possible variables: the member `name` defined inside `course` or the global variable `name`. In such a case, the more local referent of `name`, *i.e.*, the member of `course`, is selected. The global variable `name` is said to be *hidden*. Note that the member `name` defined inside `professor` is not a possible referent; within a member function body, only members of the function's class may be referred to by a simple identifier. Given the set of C++ features we have considered so far, there is no way to refer to either the global variable or the member of `professor` from within the class `course`.

The scope resolution operator `::` in its unary form allows for references to hidden global variables; the single operand is the name of a global variable, and the expression refers to the global variable of that name. Its binary form allows for references to members of enclosing classes. The left-hand operand is the

[6] Of course, the confusion here could be eliminated by choosing more descriptive labels for the three `name` variables.

name of an enclosing class, and the right-hand operand is the name of a member of the enclosing class; the expression refers to the member of the given name within the enclosing class of the given name. For example, the function `print` within `course` can refer to the global variable `name` using the unary form of the scope resolution operator, and to the `name` member of `person` via the binary form:

```
// modified definition of "print" function for class "course," using scope resolution operator
professor::course::print()
{ output("university:", ::name);                                          // global variable
  output("professor:", professor::name);                                // "professor" member
  output("course name:", name);                                           // "course" member
  output("students enrolled:", studentsEnrolled); }
```

Neither form of the scope resolution operator requires changes to the algebra. An expression consisting of an variable name preceded by the unary operator refers to the global variable of that name; such an expression corresponds to a simple identifier task. An expression involving the binary form of the operator corresponds to a *data-member* task, which we discuss in sections 3.1 and 3.2. We treat such expressions as class-member tasks, referring to a member within an object and involving a statically determined offset to the member. The assumptions made in section 3.2 to handle data-member tasks will also handle binary scope resolution expressions.

3 Programmer-defined class operations

In object-oriented programming, the operations that access a given data type are included as part of the definition of the type. As C does not allow functions to be included as part of a type definition, C++ introduces this possibility for class types. This is discussed in sections 3.1 and 3.2. Functions associated with a class may be definition as *virtual*. If a function is so defined, references to it rely on dynamic rather than static type resolution. The features supporting virtual functions are presented in sections 3.3 and 3.4.

3.1 Member functions

The first extension requiring a change in the algebra is the ability of classes to have functions as members. This extension involves changes to the algebra because the way in which member functions are accessed does not parallel the way in which other members are accessed. Unlike members of other types, referred to as *data members*, a member function does not occupy a memory area at some predetermined offset from the starting location of its class; thus our rule for class members as it stands is incapable of handling a reference to a member function.

The value returned by a member function expression must be the starting address of the function. For nonvirtual functions,[7] this address is statically determined and cannot be changed. Thus a nonvirtual member function expression corresponds to a particular, unchangeable function address. We define a partial function $FunLoc : tasks \rightarrow addresses$ which maps a member function expression to the corresponding starting address of the member function. To distinguish between member functions and data members, we add the values *data* and *function* to the *tags* universe, and a function $MemberStatus : tasks \rightarrow tags$ to determine whether a given member expression is a member function or a data member. We change the task-type tag *struct-reference* to *class-member*, in keeping with our new terminology. Our new rule is shown in Fig. 1.

Inside the body of a member function, a reference to a member of the function's class may be made simply by an expression consisting of the member's name; no class name is necessary. Consider our example in section 2.4: within the body of the member function `printPerson`, the class' `name` and `age` members are

[7] The situation is somewhat more complicated for *virtual* functions, which we discuss in section 3.4.

```
if TaskType (CurTask) = class-member then
    if ValueMode (CurTask) = lvalue then
        ReportValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask))
    elseif ValueMode (CurTask) = rvalue then
        if MemberStatus (CurTask) = data then
            ReportValue (ObjValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask)))
        elseif MemberStatus (CurTask) = function then
            ReportValue (FunLoc (CurTask))
        endif
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 1: Transition rule for class member expressions.

referred to simply as "name" and "age." This is possible because `name`, `age` and `printPerson` are members of the same class.

To handle such expressions within member functions, we shall treat them in the same way as explicit class-member expressions; an expression of this form will correspond to a class-member task. As the left operand is missing from these implicit class-member expressions, the question arises as to what the left-operand value as defined by $OnlyValue$ should be. This is addressed in the next section.

3.2 Implicit `this` parameters

Every member function has a hidden argument that is not included explicitly in either the function's list of parameter declarations or the list of argument expressions in a call to the function. This hidden argument's value is always the address of the class object of which the function is a member. Thus in our example in section 2.4, the function `person::printPerson` is explicitly defined as a nullary function, and no arguments are supplied when it is called; nevertheless, it is in fact a one-place function whose sole argument is a pointer to the class object.

To support this in our algebra, we assume that each member function has an implicit class-pointer parameter declaration in addition to those explicitly defined by the programmer. We wish to be able to distinguish this declaration as that of the implicit class-pointer parameter; we do this by adding the partial function $IsImplicitParm : tasks \rightarrow \{true, false\}$. We also add the partial function $ImplicitParm : stack \rightarrow tasks$, which returns the declaration task of the implicit class pointer for a given level of the stack. When the implicit class-pointer parameter declaration is encountered, we change the $ImplicitParm$ function. Our new rule for parameter declarations is shown in Fig. 2.

```
if TaskType (CurTask) = parameter-declaration then
    DoAssign (NewMemory (CurTask), ParamValue (CurTask, StackTop), ValueType (CurTask))
    OnlyValue (CurTask, StackTop) := NewMemory (CurTask)
    if IsImplicitParm (CurTask) = true then
        ImplicitParm (StackTop) := CurTask
ENDIF
```

Figure 2: Transition rule for parameter declarations.

We also assume that each call to a member function has an expression task returning the address of

the function's class.[8] We introduce the macro *ThisPtr* to express the value of the implicit class-pointer parameter. Within a member function, the value of the implicit parameter can be accessed via an expression consisting of the keyword `this`. To handle this new type of expression, we introduce a tag, *this*, and a corresponding rule. The definition of *ThisPtr* and the rule for `this` expressions are shown in Fig. 3.

```
macro ThisPtr: MemoryValue (OnlyValue (ImplicitParm (StackTop), StackTop),
                            ValueType (ImplicitParm(StackTop)))

if TaskType (CurTask) = this then
    ReportValue (ThisPtr)
    Moveto (NextTask (CurTask))
endif
```

Figure 3: Macro *ThisPtr*; transition rule for `this` expressions.

With the *ThisPtr* macro returning the implicit class pointer parameter value, we are now able to handle a member function's references to members of its own class. For a nonfunction member, the value to return for such a reference is the memory address value of the implicit argument offset by some value determined by *ConstVal*. For a reference to a function member, the value to return is the function's memory location, determined via *FunLoc*. Assuming that we treat class-member expressions within a member function as class-member tasks, we simply define the task's left-operand value to be the value *ThisPtr*. Thus the implicit class-member expression `name` in our example in section 2.4 will be equivalent to the explicit class-member expression `this->name`.

3.3 Object type

In C++, each class object has a particular type associated with it. If the object has been allocated as the memory location for a variable of a certain class type, the object's type will simply be the predetermined *static type* of the variable. A class object's type may also correspond to the static type of a pointer variable pointing to it, but this is not necessarily the case. For example, assume the following variables are defined using our `person`, `professor` and `student` classes:

```
person personObject;
professor profObject;
student studentObject;
```

Assume that `personPtr` is defined as a pointer to an object of class `person`:

```
person* personPtr;
```

The object types of `personObject`, `profObject` and `studentObject` are fixed at the time of their definition: the object type of `personObject` is `person`, the object type of `profObject` is `professor`, and the object type of `studentObject` is `student`. The value of `personPtr` can be changed to point to any of the objects defined by `personObject`, `profObject` or `studentObject`:

```
personPtr = (person*) &personObject;                    // object type is "person"
personPtr = (person*) &profObject;                     // object type is "professor"
personPtr = (person*) &studentObject;                   // object type is "student"
```

[8] If a base-class function is called with a pointer to a derived class as its `this` pointer, an offset must be added to the pointer to point to the embedded base-class object, and the result must be cast to a base-class pointer. We assume that the appropriate pointer-addition and casting tasks are included.

A class object's type is information stored in the object itself and determined at the time of initialization of the object. Thus while a variable can point to objects of different types, the object type of a class object cannot change. To keep track of a class object's type, we simply associate a type with the object's location in memory. We define a partial function $ObjType : addresses \rightarrow types$, which returns the type of the object at the given memory location. When a new class object is initialized, either by a variable declaration or by use of the `new` operator,[9] the $ObjType$ function is changed to reflect the type of the object at the new address.[10]

We add a new task type, *object-setup*; tasks of this type assign object type to newly created objects. After any class-object creation via a declaration or `new`-expression task, *object-setup* tasks, arranged in the order described above, assign object types to the object and its class subobjects. The offset at which to assign the object type is determined by $ConstVal$, and the type to assign is determined by $ValueType$. The rule for *object-setup* tasks is shown in Fig. 4.

```
if TaskType (CurTask) = object-setup then
    if PointerType (Decl (CurTask)) = true then
        ObjectType (MemoryValue (OnlyValue (Decl (CurTask)), ValueType (CurTask))
                + ConstVal (CurTask)) := ValueType (CurTask)
    else
        ObjectType (OnlyValue (Decl (CurTask)) + ConstVal (CurTask)) := ValueType (CurTask)
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 4: Transition rule for object type assignment tasks.

3.4 Virtual functions

The importance of object type is manifested in its interaction with *virtual functions*. A member function may be labeled virtual by placing the keyword `virtual` before the definition of the function. A function defined as virtual for a base class is also virtual for all classes derived from the base class, even if the function is redefined in a derived class. In the case where a member function is originally defined in a base class and redefined in a derived class, an access of the member name may refer to either the base-class version or the derived-class version of the function; the difference between a virtual and a nonvirtual function is in the way in which the correct version is chosen. A nonvirtual function is determined at compile time, based on the static type associated with the function's class; for a virtual function, the choice is based on the type associated with the class object.

As an example, let us add member functions to the classes `person` and `professor`. In place of the functions `printPerson`, `globalPrintPerson` and `printProfessor`, we add the functions `virtualPrint` and `nonvirtualPrint` to `person` and the function `virtualPrint` to `professor`:

```
// modified definition of class "person," with new member functions
class person
{ private:
```

[9] The `new` operator is discussed in section 4.1.

[10] As a class object may itself contain class objects, each of which must be assigned a type, a single variable declaration or `new` expression may require several object-type assignments. These assignments proceed in a specific order: first the base-class subobjects are initialized in declaration order, then the member subobjects are initialized in declaration order, and finally the enclosing object itself is initialized. A class object and one of its class subobjects may have the same starting address; in this case, the object type assigned at this address is a combination of both classes. We assume that any such combinations of types needed are included in the *types* universe.

```
  char name[25];
  int age;
  public:
  virtual void virtualPrint();                                    // virtual print function
  void nonvirtualPrint(); };                                    // nonvirtual print function

// modified definition of class "professor," with new member function
class professor:  virtual person
{ private:
  int salary;
  public:
  virtual void virtualPrint(); };                                 // virtual print function
```

Each function simply prints the member values of its class:

```
// definition of nonvirtual print function for class "person"
void person::nonvirtualPrint()
{ output("Nonvirtual print function for 'person' object");
  output("name:", name);                                          // print "name" member
  output("age:", age); }                                           // print "age" member

// definition of virtual print function for class "person"
void person::virtualPrint()
{ output("Virtual print function for 'person' object");
  output("name:", name);                                          // print "name" member
  output("age:", age); }                                           // print "age" member

// definition of virtual print function for class "professor"
void professor::virtualPrint()
{ output("Virtual print function for 'professor' object");
  person::nonvirtualPrint();                                // print function for class "person"
  output("salary:", salary); }                                      // print "salary" member
```

We define a variable `profObject`, of type `professor`; this initializes an object of type `professor`. A variable of type `person*` may then be assigned to point to this object:

```
// definition of variable "personPtr"
person* personPtr = &profObject;                                 // points to "profObject"
```

According to the static type of the variable `personPtr`, the type of the object it points to is `person`, but the object type of `personPtr`'s dereferencing is `professor`. The function calls `personPtr->nonvirtualPrint()` and `personPtr->virtualPrint()` will now exhibit different behaviors. Since `nonvirtualPrint` is a nonvirtual function, `personPtr->nonvirtualPrint()` calls the version of `nonvirtualPrint` according to the static type of `personPtr`'s dereferencing, namely `person`. The resulting output will be

```
Nonvirtual print function for "person" object
name:  Hazel Motes    age:  45
```

On the other hand, `virtualPrint` is a virtual function, so `personPtr->virtualPrint()` calls the version of `virtualPrint` as defined by the object type of `personPtr`'s dereferencing, namely `professor`. The output will be

```
Virtual print function for "professor" object
name:  Hazel Motes    age:  45    salary:  50000
```

Virtual functions require a change to the rule for class member expressions. To determine the correct address of a virtual function, we redefine $FunLoc$ as a binary function: $tasks \times types \rightarrow addresses$, which determines the address for a given member function identifier and object type. In the case of a nonvirtual function, $FunLoc$ always returns the same function starting address, regardless of the type argument; in the case of a virtual function, the starting address varies depending on the object type supplied. Our new rule is shown in Fig. 5.

```
if TaskType (CurTask) = class-member then
    if ValueMode (CurTask) = lvalue then
        ReportValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask))
    elseif ValueMode (CurTask) = rvalue then
        if MemberStatus (CurTask) = data then
            ReportValue (ObjValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask)))
        elseif MemberStatus (CurTask) = function then
            ReportValue (FunLoc (CurTask, ObjType (OnlyValue (CurTask, StackTop))))
        endif
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 5: Transition rule for class member expressions.

4 Object creation and destruction

C++ introduces convenient mechanisms for creating and destroying objects. The operator `new` allocates memory for a new object, while the operator `delete` deallocates memory associated with an object. These operators are covered in sections 4.1 and 4.2 respectively. In addition, the programmer may define functions to be invoked implicitly when an object is created or destroyed. Discussion of these *constructor* and *destructor* functions appears in sections 4.3 and 4.4 respectively.

4.1 The `new` operator

C++ introduces an operator `new` for dynamic object creation. This operator takes a type name as an operand, allocates a region of memory whose size corresponds to that of the indicated type, and returns the memory location of this newly allocated memory. In our object terminology, it creates an object of a given type and returns a pointer value to it. For example, the expression `new person` allocates enough space for the `name` and `age` members of a `person` object and returns a pointer value to this newly allocated space. Memory allocation is accomplished by a call to the global function `operator new`; if the object being allocated contains a member function `operator new`, the member function is called instead.

The `new` operator introduces a new task type tag, *new-object*, to the universe of tags. As with variable declarations, an initializing expression, if one exists, is evaluated; the evaluation task for this initializer is determined by the function *Initializer*. We introduce a partial function $Allocator : tasks \rightarrow tasks$, which maps each expression involving the `new` operator to a task which calls the appropriate version of `operator new`. This function call changes the $OnlyValue$ function, setting the `new` expression's value to a new memory location. Once a location has been established for the new object, it is returned as the value of the `new` expression; if an initializer is provided, the object is assigned its value. Our new rule is shown in Fig. 6.

```
if TaskType (CurTask) = new-object then
    if Defined (Initializer (CurTask)) and Undefined (RightValue (CurTask, StackTop)) then
        Moveto (Initializer (CurTask))
    elseif Undefined (OnlyValue (CurTask, StackTop)) then
        Moveto (Allocator (CurTask))
    else
        ReportValue (OnlyValue (CurTask, StackTop))
        if Defined (Initializer (CurTask)) then
            DoAssign (OnlyValue (CurTask), RightValue (CurTask, StackTop), PointsToType (CurTask))
        else
            Moveto (NextTask (CurTask))
ENDIF
```

Figure 6: Transition rule for `new`-operator expressions.

4.2 The `delete` operator

The `delete` operator reverses the effects of the `new` operator: given the address of an object as an operand, it deallocates the memory allocated for the object so that it can be reused. For example, given the definition

```
person* personPtr = new person;
```

the expression `delete person` deallocates the memory allocated by the `new` operator; the pointer `personPtr` no longer points to an object. Memory deallocation is accomplished by calling the global function `operator delete`; if the deallocated object contains a member function of this name, its member function will be called. An expression with the `delete` operator returns a value of type `void`. This feature can be incorporated without changing the algebra; for every `delete` expression task, we assume that a function invocation task calling the appropriate version of `operator delete` is added.

4.3 Constructors

The programmer may define special member functions called *constructors* to be invoked when an object of a class is created. Constructors are commonly used to initialize newly created objects with default values. It is important to note that a constructor function does not actually create a new object in the sense of allocating new memory to be used as a class object. The name of a constructor function member within a class is simply the name of the class itself; like other function names, it may be overloaded. To illustrate, we add two constructor functions to our class `person`:

```
// modified definition of class "person" with constructor functions
class person
{ private:
  char name[25];
  int age;
  public:
  person();                                    // "default" constructor:  requires no argument
  person(const char*, int);                              // constructor taking string and int
  virtual void virtualPrint();
  void nonvirtualPrint(); };
```

The first constructor function takes no arguments; an invocation of this function fills in the `name` and `age` members with default values:[11]

[11] The `strcpy` function, found in the `<string.h>` library, takes two string pointers as arguments and copies the first argument's string to that of the second argument.

```
// definition of default constructor for class "person"
person::person()
{ strcpy(name, "");                                          // set "name" member to null string
  age = -1; }                                               // set "age" member to invalid value
```

The second constructor function takes two arguments and fills in the `name` and `age` members with these argument values:

```
// definition of binary constructor function for class "person"
person::person(const char* n, int a)
{ strcpy(name, n);                                  // copies contents of string "n" to "name" member
  age = a; }
```

A constructor may be invoked when an automatic or static variable is declared; in the case of a static variable, it is invoked only the first time the declaration is encountered. For example, the definitions

```
person p1(), p2("Lily Hawks", 30);
```

initialize variable `p1` with the first constructor function and variable `p2` with the second constructor function. When a constructor function with no arguments, a so-called *default constructor*, is defined, it may be invoked without the use of argument parentheses; thus the definition of `p1` above is equivalent to

```
person p1;
```

Alternatively, the declaration of a variable may initialize the variable's new object in the standard C fashion, via direct assignment; in this case, the constructor function is not called.

A constructor function may also be called when a new object is allocated using the `new` operator. For example, the expression `new person("Hoover Shoats", 68)` allocates memory for a new object of class `person` and initializes it by calling the binary constructor function for `person`. Finally, a constructor may be called explicitly in the standard form for functions. Here, the result is that a temporary object is created and the constructor function called for this object.

Constructor functions do not require changes to our rules. We simply assume that function invocation tasks calling the appropriate constructor functions follow a declaration or `new`-expression task, in the order specified in section 3.3. There is no order specified between object setup and constructor invocation tasks. In the case of an explicit contructor function call, we assume that a declaration task precedes the constructor function invocation to create a temporary object.

4.4 Destructors

Just as constructor functions can be defined to handle initialization of new class objects, special member functions may also be defined to perform certain actions when a class object is destroyed. These functions, called *destructor functions*, are invoked implicitly by a variable going out of scope or by use of the `delete` operator, or explicitly by a function call. As an example, let us add a destructor function to the class `person`:

```
// modified definition of class "person," with destructor function
class person
{ private:
  char name[25];
  int age;
  public:
  person();
  person(const char*, int);
  ~person();                                                    // destructor function
```

```
  virtual void virtualPrint();
  void nonvirtualPrint(); };
```

We also add a global variable `totalPeople` to keep track of the number of allocated `person` objects:

```
int totalPeople = 0;
```

This global variable can be incremented and decremented in the class' constructor and destructor functions; then once the class is defined, the programmer need not perform any explicit incrementing or decrementing outside the class. We modify our constructor functions, adding a statement incrementing `totalPeople`:

```
// modified definition of default constructor for class "person,"
// including increment of global object counter
person::person()
{ strcpy(name, "");
  age = -1;
  totalPeople++; }                                                    // counter incremented

// modified definition of binary constructor function
// for class "person," including increment of global object counter
person::person(const char* n, int a)
{ strcpy(name, n);
  age = a;
  totalPeople++; }                                                    // counter incremented
```

Now each time a new object of class `person` is created, the counter `totalPeople` is incremented. The opposite action is performed by the destructor function: when an object is destroyed, the counter is decremented:

```
// definition of destructor function for class "person"
person::~person() { totalPeople--; }                                  // counter decremented
```

As mentioned above, the destructor function is called implicitly when a variable goes out of scope. For a local automatic variable, this is the point at which the function in which it is declared ends. For a static or global automatic variable, it is the end of the program. The destructor is also called implicitly when the `delete` operator is used to deallocate a class object. Finally, the programmer may call the destructor member function explicitly: the function name is simply the class name preceded by a tilde. Thus the expression `p1.~person()` will call the destructor function and decrement the global counter.

Explicit calls to destructor functions are handled by the existing rule for function invocations. However, as not all destructor-function calls are explicit in the program code, we must make them so in the representation of the program. We simply add a destructor function call task at each point where a class variable with a destructor function defined goes out of scope: this will be either at the end of a function or the end of the program, depending on the variable type. We shall refer to the sequence of implicit destructor function calls followed by a `return` task at the end of a function or program as a *return sequence*. We also add a destructor function invocation task before each call to `operator delete`.

5 Overloading and parameterized types

C++ allows function names and operators to be *overloaded*. Overloading is a loosening of the restrictions on associating names with functions. While in C a name may refer to at most one function, in C++ a

name may refer to a family of functions. The particular function referred to by an instance of a name is determined by the types of the arguments and return type associated with the name instance. Overloading allows the programmer to refer to conceptually similar functions with the same name. We discuss overloading in sections 5.1 and 5.2.

Similarly, the template mechanism allows the programmer to define a family of conceptually similar types through a parameterized type definition. An instantiation of a type from the family is attained by supplying values for the parameters. As with overloading of functions and operators, this allows the programmer to refer to similarly defined types with a single name. Template definitions are discussed in section 5.3.

5.1 Function overloading and default arguments

An overloaded function name refers to more than one function declaration within the same scope. When an overloaded name is used in an expression, it refers to a particular function; the function it refers to is determined by matching the actual arguments of the function reference with the formal arguments of a function declaration.[12] As an example, let us add an overloaded function `monthlySalary` as a friend to the `professor` class defined in section 2.2. Within the class definition we define two functions, both named `monthlySalary`: the first `monthlySalary` function takes a single `int` argument, while the second `monthlySalary` function takes two `int` arguments. The first `monthlySalary` function calculates a monthly salary for the `professor` object by dividing its `yearlySalary` argument by `12`:

```
// definition of unary "monthlySalary" function
int monthlySalary(int yearlySalary) { return yearlySalary / 12; }
```

The second `monthlySalary` function divides its integer `yearlySalary` argument by its integer `months` argument:

```
// definition of binary "monthlySalary" function
int monthlySalary(int yearlySalary, months)
{ return yearlySalary / months; }
```

A subsequent function call using the name `monthlySalary` is disambiguated by considering the arguments supplied in the function call. An expression `monthlySalary(40000)` is a call to the first function declaration, as its arguments match the formal arguments of the first declaration exactly. Likewise, an expression `monthlySalary(30000, 9)` is a call to the function defined in the second declaration.

Function overloading does not require any changes to the algebra because the mapping between function references and function declarations is static. When an overloaded function name is used, the function it refers to is determined by the types of its arguments; these types are determined at compile time, so the referent of the overloaded name is as well. For any expression task T consisting of an overloaded function name, we determine the best match for the function reference and assign $Decl(T)$ the declaration task of this best match.

Another C++ addition, related to function overloading, is the ability to supply default values for the formal arguments of a function. A function with a default value specified for one of its arguments may be called either with or without a value for that argument; if no actual argument is supplied, the default value is used. For instance, rather than defining separate unary and binary `monthlySalary` functions, we can define the function once as a binary function and give the `months` argument a default value of `12`:

```
// modified definition of binary "monthlySalary" function,
// with default value for "months" argument
```

[12] An exact match between actual and formal arguments is not necessary; there are rules used at compile time to determine the best match when no exact match exists. For the sake of simplicity, we shall only consider examples where formal and actual arguments match exactly.

```
int monthlySalary(int yearlySalary, months = 12)
{ return yearlySalary / months; }
```

The result is identical to that of defining unary and binary `monthlySalary` functions. The function may be called with two arguments, in which case the formal argument `months` receives the value of the second argument; it may also be called with one argument, in which case `months` receives the default value `12`.

There is no standard method for implementing default argument values; however, none of the different possible approaches require changes to the algebra. Functions with default argument values can be thought of as special cases of function overloading. Thus the above definition of `monthlySalary` would be equivalent to the following definitions:

```
// binary "monthlySalary" function
int monthlySalary(int yearlySalary, months)
{ return yearlySalary / months; }

// unary "monthlySalary" function
int monthlySalary(int yearlySalary)
{ int months = 12;
  return yearlySalary / months; }
```

An alternate approach to implementing default arguments is to define a single function and modify calls to the function, supplying default values as actual arguments if need be. For instance, our definition of `monthlySalary` above would instantiate a single binary function, and a unary function call like `monthlySalary(40000)` would be changed to `monthlySalary(40000, 12)`.

The overloaded-function approach requires no changes to the algebra, as we have seen that function overloading is a purely syntactic feature. The single-function approach does not even require function overloading; it simply involves calls to a nonoverloaded function. Thus our algebra as it stands is able to handle default argument values, regardless of their implementation.

5.2 Operator overloading

Operators may also be overloaded. The user may define an *operator function* for a particular operator, taking at least one class object as an argument. When an operator is used with no class objects as operands, the result is the standard action for the operator as defined in C; when used with a class object as one of its operands, the result is a call to the operator function defined for that class. Operator functions taking different argument types may be defined for the same operator. As with overloaded functions, the operator function for a given occurrence of an operator is determined by matching the actual operands with the formal arguments of the operator functions.

As an example, we shall overload the relational operator `>` to accommodate our class `student`. Within the class definition, we declare two friend functions, both denoted by `operator>`:

```
// modified definition of class "student,"
// allowing operator functions to access private members
class student:  private person
{ private:
  int year;
  float GPA;
  public:
  void printStudent();
  friend int operator>(student, student);                    // operator >
  friend int operator>(student, int); };                     // operator >
```

Both operator functions take a `student` class object as a left operand; the first declaration defines a function taking a `student` object as a right operand, while the second defines a function taking an `int` object as a right operand. We define the first version of `operator>` so as to return a "true" value if the `year` member of the left operand is greater than that of the right operand:[13]

```
// definition of operator function > for (student, student) operands
int operator>(student s1, s2) { return s1.year > s2.year; }
```

We define the second version of the operator function so as to return a "true" value if the `year` member of the left operand is greater than the second operand:

```
// definition of operator function > for (student, int) operands
int operator>(student s, int p) { return s.year > p; }
```

When the operator `>` is used with a `student` object as its left operand, the appropriate version of the operator function is chosen based on the type of the right operand. Thus given the `student` variables `s1` and `s2`, the expression `s1 > s2` will result in a call to the first function, since the type of the actual argument `s2` matches that of the first function's formal argument. The expression `s1 > 5` will result in a call to the second member function, for similar reasons.

Any programmer-defined operator function can also be invoked by an explicit function call. In our example, we defined two operator functions with the name `operator>`; a function call using this name is equivalent to using the operator `>`. Thus the function calls `operator>(s1, s2)` and `operator>(s1, 5)` are equivalent to the two operator expressions above.

Operator overloading, like function overloading, does not require any changes to the algebra rules as it uses only statically determined information. The use of a given operator requires one or more operand expressions of a given type: this static type information is all that is needed to determine the correct meaning of the operator. Using a programmer-defined operator function corresponds to a function-invocation task, with the function to invoke determined statically by the types of the actual arguments. For each task T involving an overloaded operator function name, we determine the best match for the function reference and assign to $Decl(T)$ the result of this best match.

5.3 Templates

The template mechanism in C++ allows the programmer to define *container classes*, classes containing members whose types are specified outside the class definition. A container class defines a family of classes differing in the types of some of their members but sharing common structure. *Abstract data types* such as stacks and queues can be represented as container classes. For example, the notion of a list defines the way in which list items, or nodes, are linked to one another and methods of manipulating the items but leaves undefined the type of information stored in a node. A family of list types can be defined simply by specifying different values for this type information. A template separates the structure common to all members of the family from the type information specifying a particular member of the family. A list of type arguments is supplied first, followed by a declaration; the specific type information is supplied as arguments, while the common structure is defined in the declaration.

A common example of a container class is the *singly-linked list*. This abstract data type consists of two data items and a set of manipulation functions. The data items are the information contained in a node of the list and a pointer to the next node in the list, and typical manipulation functions include a print function and a node-addition function. The term *singly-linked list* denotes a family of data types all sharing

[13] Of course, the values of the `year` members is not the only possible basis for a "greater-than" ordering of `student` objects; they could just as easily be ordered by the values of their `age` members, for instance. It may not be clear to a programmer using the `operator>` function what the basis for the ordering is. This is a common problem with defining operators for classes; one way of avoiding this confusion would be to define a function rather than an overloaded operator.

the above characteristics but differing in the type of information stored in each node. We define the common characteristics within the template's definition:

```
// definition of template "listNode" for singly-linked list node
template <class T>
class listNode
{ private:
  T data;                                                    // data contained in node
  listNode* next;                                  // ptr to node following this node in list
  public:
  void print();                                          // print data for all nodes in list
  void addNode(); };                                                  // add node to list
```

The only information not specified in the definition, the type of `T`, is supplied as an argument whenever the template is used. For example, to create a singly-linked list of `person` objects, we define a variable using the `listNode` template:

```
listNode<person>* personList;
```

Nodes in this list have `data` members of type `person`. A linked list of nodes with data members of type `int` can be created in a similar way:

```
listNode<int>* intList;
```

A family of functions can also be defined by a function definition within a template. For instance, we can create a function template `max` which, given two objects of the same type as arguments, returns the greater of the two. The function definition within the template specifies everything except the return type of the function and the type of its arguments:

```
// definition of function template "max"
template<class T>
T max (T a, b) { return a > b ?  a :  b; }
```

The type information is supplied in a particular invocation of the template; for instance, the function call `max<int>(1, 2)` will compare the two `int` objects and return the `int` value `2`. The function call `max<student>(s1, s2)` will compare two `student` objects, using our definition of `>` in section 5.2, and return a `student` value.

The template feature is another language facility that affects only the static information associated with a program. As stated earlier, a template defines a family of types; defining a template is equivalent to defining each type in the family separately. Thus we may treat types like `listNode<int>` and `listNode<person>` as distinct types and functions like `max<int>` and `max<student>` as distinct functions; the fact that they are generated by the same template has no effect on the dynamics of the program. As templates affect only the way in which a program's static information is determined, we do not need to alter our algebra to accommodate them.

6 Other extensions

C++ introduces several language features which do not fit well into any category. The extensions discussed here round out the set of C++ extensions.

6.1 Constant objects

When an object is created, it may be specified as *constant*. An object so specified may be given an initial value, but this value may not be subsequently modified. Constant status is assigned by prefixing the keyword `const` to the object's type. For example, once a constant `person` object has been created from the definition of variable `p1`,

```
const person p1("Leora Watts", 26);
```

an expression that simply accesses a member of the object, such as `p1.age`, is valid, but an expression that would modify the value of a member, such as `p1.age++`, results in an error at compile time.

Apart from special considerations during compile time, constant objects are treated no differently from nonconstant objects. Expressions and statements that would alter the value of a constant object are simply rejected during compilation; once a program is compiled, constant and nonconstant objects are equivalent. Thus our algebra does not need to be changed to accommodate constant objects.

6.2 Inline functions

A function may be defined as *inline* by prefixing the keyword `inline` to its declaration; this indicates to the compiler that it should try to handle calls to this function without using the standard function-call mechanism. Rather than creating a single memory location for the function and subsequently passing control to this location each time it is called, the compiler will replace each call to the function with the sequence of code contained within the function. The function name then acts much like a macro. For example, let us define our `monthlySalary` function as inline:

```
// modified definition of function "monthlySalary" as "inline"
inline int monthlySalary(int yearlySalary, months = 12)
{ return yearlySalary / months; }
```

The compiler will transform an expression like `monthlySalary(30000, 9)` into an expression not involving a function call. Replacing the formal arguments of the inline function with actual arguments results in the expression `30000 / 9`, which may then be simplified to `3333`.

The effects of inlining are limited to compile time; it has no effect on the code once it is compiled. Our algebra applies only to the compiled version of a program, in which all changes to the code made by inlining are incorporated. Hence this feature does not require any changes to the algebra.

6.3 References

C++ introduces the *reference* as a means of attaching a name to an object. A reference is declared in a way similar to the declaration of a variable: a declaration contains a name for the reference and a type specification followed by the symbol `&`. A reference declaration must also contain an initializing expression determining the object that the reference refers to. For instance, given the definition of `personObject` in section 3.3, we may subsequently define a reference `personReference`:

```
// definition of reference "personReference"
person& personReference = personObject;
```

This creates a reference which returns the value of the object referred to by `personObject` each time it is used. Note that the definition does not create a new object of type `person`; `personReference` simply refers to the existing object `personObject`. Thus a modification to `personReference`'s object is a modification to `personObject`'s object; after the assignment

```
personReference.age = 22;
```

the expressions `personObject.age` and `personReference.age` will both return the value 22.

Our existing rules for declarations can accommodate references, with one additional stipulation. In a reference declaration, the reference is assigned the address or *lvalue* of an object specified in the required initializer expression. We stipulate that the initializer-expression task associated with a reference via the function *Initializer* returns the lvalue of its expression. In addition, we must add rules for reference identifiers and members. The use of a reference should return the lvalue or rvalue of the reference's object. This is determined indirectly by the address stored when the reference is declared. Thus an lvalue access returns the address stored in the reference, while an rvalue access returns the value of the object stored at this address. We add the task types *reference-identifier* and *reference-class-member* to the *tasks* universe, and the corresponding rules shown in Fig. 7 and Fig. 8.

```
if TaskType (CurTask) = reference-identifier then
    if ValueMode (CurTask) = lvalue then
        if GlobalVar (CurTask) = true then
            ReportValue (ObjValue (GlobalVarLoc))
        elseif GlobalVar (CurTask) = false then
            ReportValue (ObjValue (LocalVarLoc))
        endif
    elseif ValueMode (CurTask) = rvalue then
        if GlobalVar (CurTask) = true then
            ReportValue (Deref (ObjValue (GlobalVarLoc)))
        elseif GlobalVar (CurTask) = false then
            ReportValue (Deref (ObjValue (LocalVarLoc)))
        endif
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 7: Transition rule for reference identifier expressions.

```
if TaskType (CurTask) = reference-class-member then
    if ValueMode (CurTask) = lvalue then
        ReportValue (ObjValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask)))
    elseif ValueMode (CurTask) = rvalue then
        ReportValue (Deref (ObjValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask))))
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 8: Transition rule for reference class member expressions.

6.4 Exception handling

C++ adds exception handling as a means of recovering from run-time errors. A set of *catchers* may be associated with a block of code, a *try block*. These catchers are themselves blocks of code, intended to be used as a means of recovering smoothly from run-time errors occurring within the try block. A catcher is

invoked by *throwing an exception*; this results in control being passed to an exception catcher associated with an enclosing `try` block. An *exception* is an object, and different exception catchers are associated with different object types; thus the catcher invoked for a given exception is determined by matching the exception object's type and the type associated with a catcher.

If an exception is thrown within a function, control passes to the appropriate catcher defined within that function, if one exists. Otherwise the function stack is "unwound": the function invocation is popped off the stack, the return sequence of destructor functions is performed for any objects local to the function,[14] control returns to the next function invocation on the stack, and a catcher is searched for there. Unwinding continues until a catcher with the proper type is found.

To illustrate, we add exception handling to the member functions of our `person` class. As this class contains a string member `name`, there is the possibility of string overflow. Consider adding a member function `inputName` which accepts a name value from the user and sets the object's `name` member to this value. A user could enter a string longer than 25 characters, exceeding the bounds of the name member. In this case, we would like to issue a warning to the user and truncate the string to the 25-character limit. We first add a new member `nameTooLong` to serve as an *exception class*; this is the type of object to be thrown when a string overflow exception is encountered:

```
// modified definition of class "person,"
// with exception class "nameTooLong"
class person
{ private:
  char name[25];
  int age;
  public:
  class nameTooLong { } ;                                  // exception class for string overflow
  person();
  person(const char*, int);
  ~person();
  void inputPerson();                                          // new member function "inputName"
  virtual void virtualPrint();
  void nonvirtualPrint(); };
```

Next we define `setName`, which throws a `nameTooLong` exception if it encounters a string of length greater than 25:

```
// definition of function "setName", with exception "nameTooLong"
person::setName(const char* n)
{ if (strlen(n) > 25) throw nameTooLong;
  strcpy(name, n); }
```

Finally, we add the `inputName` function, enclosing the call to the constructor function within a `try` block and adding a catcher for a `nameTooLong` exception:[15]

```
// definition of function "inputName," with catcher for exception
person::inputName()
{ char n[80];
  try {
    input(n);
    setName(n);}
```

[14] See section 4.4 for a discussion of return sequences.

[15] We assume that the function `input` reads input from a device and sets the value of its argument to this input value. As with the function `output`, we do not define the function explicitly.

```
catch(nameTooLong)
{ output("Warning:  truncating name to 25 characters");
  n[24] = \0;                                                    // set end-of-string marker
  setName(n); } }                                  // call "setName" again with truncated string
```

Exception handling is now in place for the `inputPerson` function. If the function is invoked and the user enters an overly long string, the `nameTooLong` exception will be thrown when the `setName` function is invoked. At this point, memory is allocated for a temporary `nameTooLong` exception object. As the `setName` function has no `nameTooLong` exception catcher, the function terminates and control returns to `inputPerson`. This function does contain a `nameTooLong` catcher, so control passes directly to the catcher; the warning is displayed, and the name member is truncated and copied into the object.

We add a new rule to be executed when a program is unwinding. $Unwinding : \{true, false\}$ determines whether the stack is being unwound. $ExceptionTask : tasks$ returns the `throw`-statement task that has been executed. $Catcher : tasks \times types \rightarrow tasks$ maps each task to the catcher associated with it for the given exception-object type. If no such catcher is defined for a given task, the value of $Catcher$ is $undef$ for that task and type. $Return : tasks \rightarrow types$ maps each task to the first task of the return sequence. Finally, $InReturnSeq : Tasks \rightarrow \{true, false\}$ determines whether the given task is part of a return sequence. Our rule for the unwinding state, shown in Fig. 9, fires if unwinding is underway and a return sequence is not being executed. It passes control to an exception catcher if one of the appropriate type is defined for the current task and passes control to the return sequence at the end of the function otherwise.

---

if Defined (CurTask) and Unwinding = true and not InReturnSeq (CurTask) then
  if Defined (Catcher (CurTask, ValueType (ExceptionTask))) then
    Unwinding := false
    Moveto (Catcher (CurTask, ValueType (ExceptionTask)))
  else
    Moveto (Return (CurTask))
ENDIF

Figure 9: Transition rule for unwinding state.

We place an extra constraint on all our other rules so that they do not conflict with the unwinding rule. For each rule of the form "*if* G *then[rule body]*" where G is a truth-functional guard condition, we change the rule to: "***if (Unwinding = false or InReturnSeq (CurTask)) and*** G *then[rule body]*."

We add a new task type to handle `throw` statements and a corresponding tag name *throw*. In the rule for such statements, shown in Fig. 10, we check to see whether memory has been allocated for the exception object; if none has been allocated, we move to a task calling the `operator new` function. Once memory has been allocated, we set $ExceptionTask$ to the current task and copy the exception object's value. After the copying, control passes back to the throw task, at which point $Unwinding$ is set to $true$.

We also alter our rules to handle expressions involving exception objects. Within a catcher, the exception thrown can be referred to by an identifier. The memory location of the exception object is determined by the throw task that threw the exception. We add a partial function $IsException : \{true, false\}$ which determines whether a given identifier task refers to an exception object. The modified rule for nonreference identifiers is shown in Fig. 11; a similar modification is needed for the rule for reference identifiers.

Finally, as the exception object is eliminated when the catcher terminates, the destructor function for this object must be called. Thus at the end of the catcher we add a function-invocation task which calls the destructor of the exception object.

```
if (Unwinding = false or InReturnSeq (CurTask)) and TaskType (CurTask) = throw then
    if Undefined (OnlyValue (CurTask, StackRoot)) then
        ExceptionTask := undef
        Moveto (Allocator (CurTask))
    elseif Undefined (ExceptionTask) then
        ExceptionTask := CurTask
        DoAssign (OnlyValue (CurTask, StackRoot), RightValue (CurTask, StackTop),
                  ValueType (CurTask))
    else
        Unwinding := true
ENDIF
```

Figure 10: Transition rule for `throw` statements.

```
if TaskType (CurTask) = identifier then
    if ValueMode (CurTask) = lvalue then
        if GlobalVar (CurTask) = true then
            if IsException (CurTask) = true then
                ReportValue (ExceptionLoc)
            else
                ReportValue (GlobalVarLoc)
            endif
        elseif GlobalVar (CurTask) = false then
            ReportValue (LocalVarLoc)
        endif
    elseif ValueMode (CurTask) = rvalue then
        if GlobalVar (CurTask) = true then
            if IsException (CurTask) = true then
                ReportValue (ObjValue (ExceptionLoc))
            else
                ReportValue (ObjValue (GlobalVarLoc))
            endif
        elseif GlobalVar (CurTask) = false then
            ReportValue (ObjValue (LocalVarLoc))
        endif
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 11: Transition rule for identifier expressions.

7 Conclusion

Our algebra as it stands represents all the features of C++ as described in [ES]. Unfortunately, we cannot claim that our specification constitutes a standard version of C++, as no standard has been established for the language. While [ES] is the closest thing to a specification currently in print, it is primarily a guide for language implementors; as such, it mixes "tips" for implementation into the language specification. In some parts of the text, it is not easy to distinguish the implementation tips from the specification. Nevertheless, we have done our best to remove implementation-specific details from our specification and present the language in its full generality. As [ES] has been chosen as a "starting point" for an ANSI standard, it seems likely that our specification will closely approximate any eventual standard.

A Macro definitions

We assume the macro definitions shown in Fig. 12. $Defined$ and $Undefined$ test whether a given value is defined or undefined. The macros $GlobalVarLoc$, $LocalVarLoc$ and $ExceptionLoc$ are used to determine the memory locations of global and local variables and exception objects. $ObjValue$ returns the value of an object, given the object's memory location. Finally, $Deref$ takes a pointer value and returns the value of the object pointed to by the pointer.

---

macro Defined(Value): Value ≠ undef
macro Undefined(Value): Value = undef
macro GlobalVarLoc: OnlyValue (Decl (CurTask), StackRoot)
macro LocalVarLoc: OnlyValue (Decl (CurTask), StackTop)
macro ExceptionLoc: OnlyValue (ExceptionTask, StackRoot)
macro ObjValue(MemLoc): MemoryValue (MemLoc, ValueType (CurTask))
macro Deref(Value): MemoryValue (Value, PointsToType (CurTask))

Figure 12: Initial macro definitions.

References

[ES] Ellis, M. and B. Stroustrup. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

[Gu] Gurevich, Y. (1992). "Evolving Algebras: An Attempt to Discover Semantics," in *Current Trends in Theoretical Computer Science* (ed. G. Rozenberg and A. Salomaa), World Scientific, 266-292.

[GH] Gurevich, Y. and J. Huggins. (1993). "The Semantics of the C Programming Language," in *Lecture Notes in Computer Science*, 702 (ed. E. Börger *et al.*), Springer-Verlag, 274-308.

[KR] Kernighan, B. and D. Ritchie. (1988). *The C Programming Language*. Prentice-Hall.