CS143 Summer 2012 Handout 18 July 16th, 2012

Semantic Analysis

What Is Semantic Analysis?

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now we'll move forward to *semantic analysis*, where we delve even deeper to check whether they form a sensible set of instructions in the programming language. Whereas any old noun phrase followed by some verb phrase makes a syntactically correct English sentence, a semantically correct one has subject-verb agreement, proper use of gender, and the components go together to express an idea that makes sense. For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth. Semantic analysis is the front end's penultimate phase and the compiler's last chance to weed out incorrect programs. We need to ensure the program is sound enough to carry on to code generation.

A large part of semantic analysis consists of tracking variable/function/type declarations and type checking. In many languages, identifiers have to be declared before they're used. As the compiler encounters a new declaration, it records the type information assigned to that identifier. Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed. For example, the type of the right side expression of an assignment statement should match the type of the left side, and the left side needs to be a properly declared and assignable identifier. The parameters of a function should match the arguments of a function call in both number and type. The language may require that identifiers be unique, thereby forbidding two global declarations from sharing the same name. Arithmetic operands will need to be of numeric—perhaps even the exact same type (no automatic **`int`**-to-**`double`** conversion, for instance). These are examples of the things checked in the semantic analysis phase.

Some semantic analysis might be done right in the middle of parsing. As a particular construct is recognized, say an addition expression, the parser action could check the two operands and verify they are of numeric type and compatible for this operation. In fact, in a one-pass compiler, the code is generated right then and there as well. In a compiler that runs in more than one pass (such as the one we are building for Decaf), the first pass digests the syntax and builds a parse tree representation of the program. A second pass traverses the tree to verify that the program respects all semantic rules as well. The single-pass strategy is typically more efficient, but multiple passes allow for better modularity and flexibility (i.e., can often order things arbitrarily in the source program).

Types and Declarations

We begin with some basic definitions to set the stage for performing semantic analysis. A *type* is a set of values and a set of operations operating on those values. There are three categories of types in most programming languages:

Base types	**`int`**, **`float`**, **`double`**, **`char`**, **`bool`**, etc. These are the primitive types provided directly by the underlying hardware. There may be a facility for user-defined variants on the base types (such as C **`enum`**s).
Compound types	arrays, pointers, records, **`struct`**s, **`union`**s, classes, and so on. These types are constructed as aggregations of the base types and simple compound types.
Complex types	lists, stacks, queues, trees, heaps, tables, etc. You may recognize these as abstract data types. A language may or may not have support for these sort of higher-level abstractions.

In many languages, a programmer must first establish the name and type of any data object (e.g., variable, function, type, etc). In addition, the programmer usually defines the lifetime. A *declaration* is a statement in a program that communicates this information to the compiler. The basic declaration is just a name and type, but in many languages it may include modifiers that control visibility and lifetime (i.e., **`static`** in C, **`private`** in Java). Some languages also allow declarations to initialize variables, such as in C, where you can declare and initialize in one statement. The following C statements show some example declarations:

```
double calculate(int a, double b);  // function prototype

int x = 0;              // global variables available throughout
double y;         // the program

int main() {
   int m[3];      // local variables available only in main
   char *n;
   ...
}
```

Function declarations or *prototypes* serve a similar purpose for functions that variable declarations do for variables. Function and method identifiers also have a type, and the compiler can use ensure that a program is calling a function/method correctly. The compiler uses the prototype to check the number and types of arguments in function calls. The location and qualifiers establish the visibility of the function (Is the function global? Local to the module? Nested in another procedure? Attached to a class?) Type declarations (e.g., C **`typedef`**, C++ classes) have similar behaviors with respect to declaration and use of the new typename.

Type Checking

Type checking is the process of verifying that each operation executed in a program respects the type system of the language. This generally means that all operands in any expression are of appropriate types and number. Much of what we do in the semantic analysis phase is type checking. Sometimes the rules regarding operations are defined by other parts of the code (as in function prototypes), and sometimes such rules are a part of the definition of the language itself (as in "both operands of a binary arithmetic operation must be of the same type"). If a problem is found, e.g., one tries to add a char pointer to a double in C, we encounter a *type error*. A language is considered *strongly-typed* if each and every type error is detected during compilation. Type checking can be done compilation, during execution, or divided across both.

Static type checking is done at compile-time. The information the type checker needs is obtained via declarations and stored in a master symbol table. After this information is collected, the types involved in each operation are checked. It is very difficult for a language that only does static type checking to meet the full definition of strongly typed. Even motherly old Pascal, which would appear to be so because of its use of declarations and strict type rules, cannot find every type error at compile time. This is because many type errors can sneak through the type checker. For example, if **`a`** and **`b`** are of type **`int`** and we assign very large values to them, **`a * b`** may not be in the acceptable range of **`int`**s, or an attempt to compute the ratio between two integers may raise a division by zero. These kinds of type errors usually cannot be detected at compile time. C makes a somewhat paltry attempt at strong type checking—things as the lack of array bounds checking, no enforcement of variable initialization or function return create loopholes. The typecast operation is particularly dangerous. By taking the address of a location, casting to something inappropriate, dereferencing and assigning, you can wreak havoc on the type rules. The typecast basically suspends type checking, which, in general, is a pretty risky thing to do.

Dynamic type checking is implemented by including type information for each data location at runtime. For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type". The execution of any operation begins by first checking these type tags. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution. For example, when an add operation is invoked, it first examines the type tags of the two operands to ensure they are compatible. LISP is an example of a language that relies on dynamic type checking. Because LISP does not require the programmer to state the types of variables at compile time, the compiler cannot perform any analysis to determine if the type system is being violated. But the runtime type system takes over during execution and ensures that type integrity is maintained. Dynamic type checking clearly comes with a runtime performance penalty, but it usually much more difficult to subvert and can report errors that are not possible to detect at compile-time.

Many compilers have built-in functionality for correcting the simplest of type errors. *Implicit type conversion*, or *coercion*, is when a compiler finds a type error and then changes the type of the variable to an appropriate type. This happens in C, for example, when an addition operation is performed on a mix of integer and floating point values. The integer values are implicitly promoted before the addition is performed. In fact, any time a type error occurs in C, the compiler searches for an appropriate conversion operation to insert into the compiled code to fix the error. Only if no conversion can be done, does a type error occur. In a language like C++, the space of possible automatic conversions can be enormous, which makes the compiler run more slowly and sometimes gives surprising results.

Other languages are much stricter about type coercion. Ada and Pascal, for example, provide almost no automatic coercions, requiring the programmer to take explicit actions to convert between various numeric types. The question of whether to provide a coercion capability or not is controversial. Coercions can free a programmer from worrying about details, but they can also hide serious errors that might otherwise have popped up during compilation. PL/I compilers are especially notorious for taking a minor error like a misspelled name and re-interpreting it in some unintended way. Here's a particular classic example:

```
DECLARE (A, B, C) CHAR(3);
B = "123"; C = "456"; A = B + C;
```

The above PL/1 code declares A, B, and C each as 3-character array/strings. It assigns B and C string values and then adds them together. Wow, does that work? Sure, PL/I automatically coerces strings to numbers in an arithmetic context, so it turns B and C into 123 and 456, then it adds them to get 579. How about trying to assign a number to a string? Sure, why not! It will convert a number to string if needed. However, herein lies the rub: it converts the string using a default width of 8, so it actually converted the result to "   579". And because A was declared to only hold 3 characters, it will truncate (silently), and A gets assigned "   ". Probably not what you wanted, eh? (The first design principle for PL/I was "Anything goes! If a particular combination of symbols has a reasonably sensible meaning, that meaning will be made official.")

Case Study: ML Data Type

ML, or Meta-Language, is an important functional language developed in Edinburgh in the 1970's. It was developed to implement a theorem prover, but recently, it has gained popularity as a general purpose language. ML deals with data types in a novel way. Rather than require declarations, ML works very hard to infer the data types of the arguments to functions. For example:

```
fun mult x = x * 10;
```

requires no type information because ML infers that x is an integer since it is being multiplied by an integer. The only time a programmer must supply a declaration is if ML cannot infer the types. For example,

```
fun sqr x = x * x;
```

would result in a type error because the multiplication operator is *overloaded*, i.e., there exist separate multiplication operations for reals and for integers. ML cannot determine which one to use in this function, so the programmer would have to clarify:

```
fun sqr x:int = x * x;
```

The fact that types do not have to be declared unless necessary, makes it possible for ML to provide one of its most important features: *polymorphism*. A polymorphic function is one that takes parameters of different types on different activations. For example, a function that returns the number of elements in a list:

```
fun length(L) = if L = nil then 0 else length (tl(L)) + 1;
```

(Note: **`tl`** is a built-in function that returns all the elements after the first element of a list.) This function will work on a list of integers, reals, characters, strings, lists, etc. Polymorphism is an important feature of most object-oriented languages also. It introduces some interesting problems in semantic analysis, as we will see a bit later.

Designing a Type Checker

When designing a type checker for a compiler, here's the process:

1. identify the types that are available in the language
2. identify the language constructs that have types associated with them
3. identify the semantic rules for the language

To make this process more concrete, we will present it in the context of Decaf. Decaf is a somewhat strongly typed language like C since declarations of all variables are required at compile time. In Decaf, we have base types (**`int`**, **`double`**, **`bool`**, **`string`**), and compound types (arrays, classes, interfaces). An array can be made of any type (including other arrays). ADTs can be constructed using classes, but they aren't handled in any way differently than classes, so we don't need to consider them specially.

Now that we know the types in our language, we need to identify the language constructs that have types associated with them. In Decaf, here are some of the relevant language constructs:

constants	obviously, every constant has an associated type. A scanner tells us these types as well as the associated lexeme.
variables	all variables (global, local, and instance) must have a declared type of one of the base types or the supported compound types.
functions	functions have a return type, and each parameter in the function definition has a type, as does each argument in a function call.
expressions	an expression can be a constant, variable, function call, or some operator (binary or unary) applied to expressions. Each of the various expressions have a type based on the type of the constant, variable, return type of the function, or type of operands.

The other language constructs in Decaf (**`if`**, **`while`**, **`Print`**, assignments, etc.) also have types associated with them, because somewhere in each of these we find an expression.

The final requirement for designing a type checking system is listing the semantic rules that govern what types are allowable in the various language constructs. In Decaf, the operand to a unary minus must either be **`double`** or **`int`**, the expression used in a loop test must be of **`bool`** type, and so on. There are also general rules, not just a specific construct, such as all variables must be declared, all classes are global, and so on.

These three things together (the types, the relevant constructs, and the rules) define a *type system* for a language. Once we have a type system, we can implement a type checker as part of the semantic analysis phase in a compiler.

Implementation

The first step in implementing a type checker for a compiler is to record type information for each identifier. All a scanner knows is the name of the identifier so that it what is passed to the parser. Typically, the parser will create some sort of "declaration" record for each identifier after parsing its declaration to be stored for later. On encountering uses of that identifier, the semantic analyzer can lookup that name and find the matching declaration or report when no declaration has been found. Let's consider an example. In Decaf, we have the following productions that are used to parse a variable declaration.

```
VariableDecl-> Variable ;

Variable       -> Type identifier

Type           -> int
               -> bool
               -> double
               -> string
               -> identifier
               -> Type []
```

Consider the following variable declarations:

```
int a;
double b;
```

The scanner stores the name for an identifier lexeme, which the parser records as an attribute attached to the token. When reducing the **`Variable`** production, we have the type associated with the **`Type`** symbol (passed up from the **`Type`** production) and the name associated with the **`identifier`** symbol (passed from the scanner). We create a new variable declaration, declaring that identifier to be of that type, which can be stored in a symbol table for lookup later on.

Representing base types and array types are pretty straightforward. Classes are a bit more involved because the class needs to record a list or table of all fields (variables and methods) available in the class to enable access and type checking on the fields. Classes also need to be able to support inheritance of all parent fields that might be implemented by linking the parent's table into the child's or copying the entries in the parent table to the child's. Interfaces are similar to classes, but have only method prototypes, and no implementation or instance variables.

Once we have the type information stored away and easily accessible, we use it to check that the program follows the general semantic rules and the specific ones concerning the language constructs. For example, what types are allowable to be added? Consider Decaf's expression productions:

```
Expr  -> Constant |
         Lvalue |
         Expr + Expr |
         Expr - Expr |
         ....

LValue   -> identifier

Constant -> intConstant |
            doubleConstant |
                         ...
```

In parsing an expression such as **`x + 7`**, we would apply the **`LValue`** and **`Constant`** productions to the two sides respectively. Passed up would be the identifier information for the variable on the left and the constant from the right. When we are handling the **`Expr`** + **`Expr`** production, we examine the type of each operand to determine if it is appropriate in this context, which in Decaf, means the two operands must be both **`int`** or both **`double`**.

The semantic analysis phase is all about verifying the language rules, especially those that are too complex or difficult to constrain in the grammar. To give you an idea, here are a few semantic rules from the Decaf spec:

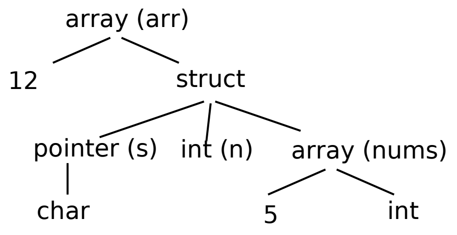
arrays	the index used in an array selection expression must be of integer type
expressions	the two operands to logical **&&** must both be **`bool`** type, the result is **`bool`** type
functions	the type of each actual argument in a function call must be compatible with the formal parameter
classes	if specified, the parent of a class must be a properly declared class type
interfaces	all methods of the interface must be implemented if a class states that it implements the interface

As we can see from the above examples, much of semantic checking has to do with types, but, for example, we also check that identifiers are not re-used within the same scope or that break only appears inside a loop. The implementation of a type checker consists of working through all of the defined semantic rules of a language and making sure they are consistently followed.

Type Equivalence of Compound Types

The equivalence of base types is usually very easy to establish: an **`int`** is equivalent only to another **`int`**, a **`bool`** only to another **`bool`**. Many languages also require the ability to determine the equivalence of compound types. A common technique in dealing with compound types is to store the basic information defining the type in tree structures.

```
struct {
   char *s;
   int n;
   int nums[5];
} arr[12];
```



Here is a set of rules for building type trees:

arrays	two subtrees, one for number of elements and one for the base type
structs	one subtree for each field
pointers	one subtree that is the type being referenced by the pointer

If we store the type information in this manner, checking the equivalence of two types turns out to be a simple recursive tree operation. Here's an outline of a recursive test for structural equivalence:

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
   if (tree1 == tree2)  // if same type pointer, must be equiv!
      return true;
   if (tree1->type != tree2->type)     // check types first
      return false;
   switch (tree1->type) {
      case T_INT: case T_DOUBLE: ...
         return true;      // same base type
      case T_PTR:
         return (AreEquivalent(tree1->child[0], tree2->child[0]);
      CASE T_ARRAY:
         return (AreEquivalent(tree1->child[0], tree2->child[0]) &&
                (AreEquivalent(tree1->child[1], tree2->child[1]);
      ...
```

The function looks simple enough, but is it guaranteed to terminate? What if the type being compared is a record that is recursive (i.e., contains a pointer to a record of the same type?) Hmmm… we need to be a bit more careful! To get around this, we could mark the tree nodes during traversal to allow us to detect cycles or limit equivalence on pointer types to in name only.

User Defined Types

The question of equivalence becomes more complicated when you add user-defined types. Many languages allow users to define their own types (e.g., using **`typedef`** in C, or **`type`** in Pascal). Here is a Pascal example:

```
type
   little = array[1..5] of integer;
   small = array[1..5] of integer;
   big = array[1..10] of integer;

var
   a, b: array[1..5] of integer;
   c: array[1..5] of integer;
   d, e: little;
   f, g: small;
   h, i: big;
```

When are two types the same? Which of the types are equivalent in the above example? It depends on how one defines "equivalence", the two main options are *named* versus *structural* equivalence. If the language supports named equivalence, two types are the same if and only if they have the same name. Thus **`d`** and **`e`** are type-equivalent, so are **`f`** and **`g,`** and **`h`** and **`i`**. The variables **`a`** and **`b`** are also type-equivalent because they have identical (but unnamed) types. (Any variables declared in the same statement have the same type.) But **`c`** is a different, anonymous type. And even though the **`small`** type is a synonym for **`little`** which is a synonym for an array of 5 integers, Pascal, which only supports named equivalence, does not consider **`d`** to be type-equivalent to **`a`** or **`f`**. The more general form of equivalence is *structural equivalence.* Two types are structurally equivalent if a recursive traversal of the two type definition trees matches in entirety. Thus, the variables **`a`** through **`g`** are all structurally equivalent but are distinct from **`h`** and **`i`**.

Which definition of equivalence a language supports is a part of the definition of the language. This, of course, has an impact on the implementation of the type checker of the compiler for the language. Clearly, a language supporting named equivalence is much easier and quicker to type check than one supporting structural equivalence. But there is a trade-off. Named equivalence does not always reflect what is really being represented in a user-defined type. Which version of equivalence does C support? Do you know? How could you find out? (The first programming language that allowed compound and complex data structures was Algol 68. This language allowed for recursively defined type expressions and used structural equivalence.)

Type Compatibility, Subtyping

In addition to establishing rules for type equivalency, the type system also defines type compatibility. Certain language constructs may require equivalent types, but most allow for substitution of coercible or compatible types.

We've already talked a bit about type coercion. An **`int`** and a **`double`** are not type equivalent, but a function that takes a double parameter may allow an integer argument to be passed because an integer can be coerced to a double without loss of precision. The reverse may or may not be true: in C, a **`double`** is substitutable for an **`int`** (it is truncated); in Java, a typecast is required to force the truncation. This sort of automatic coercion affects both the type checker and the code generator, since we need to recognize which coercions are valid in a particular context and if required, generate the appropriate instructions to actually do the conversion.

Subtypes are a way of designating freely compatible types. If a data type has all of the behavior and features of another type, to where it is freely substitutable for that type, we say it is a subtype of that type. C's **`enum`**s, for example, allow you to define new subtypes of **`int`**, similar to Pascal subrange type. A subtype is compatible with its parent type, which means an expression of the subtype can always be substituted where the general type was expected. If a function is expected an **`int`** as a parameter, substituting an **`enum`** that will only have value 1 or 2, for example, is perfectly fine. Thus the type checker needs to have an awareness of compatible types to allow such a substitution.

In object-oriented languages, inheritance and interfaces allow other ways for subtypes to be defined. The type checker allows an instance of a subclass can be freely substituted for an instance of the parent class.

Scope Checking

A simple shell scripting language might require all variables to be declared at the top-level so they are visible everywhere. But that throws all identifiers into one big pot and prevents names from ever being used again. More often a language offers some sort of control for *scopes*, constraining the visibility of an identifier to some subsection of the program. Global variables and functions are available anywhere. Local variables are only visible within certain sections. To understand how this is handled in a compiler, we need a few definitions. A *scope* is a section of program text enclosed by basic program delimiters, e.g., **`{}`** in C, or **`begin-end`** in Pascal. Many languages allow *nested scopes* that are scopes defined within other scopes. The scope defined by the innermost such unit is called the *current scope*. The scopes defined by the current scope and by any enclosing program units are known as *open scopes*. Any other scope is a *closed*.

As we encounter identifiers in a program, we need to determine if the identifier is accessible at that point in the program. This is called *scope checking*. If we try to access a local variable declared in one function in another function, we should get an error message. This is because only variables declared in the current scope and in the open scopes containing the current scope are accessible.

An interesting situation can arise if a variable is declared in more than one open scope. Consider the following C program:

```
int a;

void Binky(int a) {
   int a;

   a = 2;
   ...
}
```

When we assign to **`a`**, should we use the global variable, the local variable, or the parameter? Normally it is the innermost declaration, the one nearest the reference, which wins out. Thus, the local variable is assigned the value 2. When a variable name is re-used like this, we say the innermost declaration *shadows* the outer one. Inside the **`Binky`** function, there is no way to access the other two **`a`** variables because the local variable is shadowing them and C has no mechanism to explicitly specific which scope to search.

There are two common approaches to the implementation of scope checking in a compiler. The first is to implement an individual symbol table for each scope. We organize all these symbol tables into a *scope stack* with one entry for each open scope. The innermost scope is stored at the top of the stack, the next containing scope is underneath it, etc. When a new scope is opened, a new symbol table is created and the variables declared in that scope are placed in the symbol table. We then push the symbol table on the stack. When a scope is closed, the top symbol table is popped. To find a name, we start at the top of the stack and work our way down until we find it. If we do not find it, the variable is not accessible and an error should be generated.

There is a disadvantage to this approach, besides the obvious overhead of creating additional symbol tables and doing the stack processing. All global variables will be at the bottom of the stack, so scope checking of a program that accesses a lot of global variables through many levels of nesting can run slowly. The overhead of a table per scope can also contribute to memory bloat in the compiler.

Nevertheless, this approach has the advantage that each symbol table, once populated, remains immutable for the rest of the compilation process. The declaration records associated with each symbol may change as you add more information in the process, but the mappings themselves will not. Often times, immutable data structures lead to more robust code.

The other approach to the implementation of scope checking is to have a single global table for all the scopes. We assign to each scope a scope number. Each entry in the symbol table is assigned the scope number of the scope it is contained in. A name may appear in the symbol table more than once as long as each repetition has a different scope number.

When we encounter a new scope, we increment a scope counter. All variables declared in this scope are placed in the symbol table and assigned this scope's number. If we then encounter a nested scope, the scope counter is incremented once again and any newly declared variables are assigned this new number. Using a hash table, new names are always entered at the front of the chains to simplify the searches. Thus, if we have the same name declared in different nested scopes, the first occurrence of the name on the chain is the one we want.

When a scope is closed, all entries with the closing scope number are deleted from the table. Any previously shadowed variables will now be accessible again. If we try to access a name in a closed scope, we will not find it in the symbol table causing an error to be generated. The disadvantage of the single combined symbol table is that closing a scope can be an expensive operation if it requires traversing the entire symbol table.

There are two scoping rules that can be used in block-structured languages—*static scoping* and *dynamic scoping*. In static scoping, a function is called in the environment of its definition (i.e., its lexical placement in the source text), where in dynamic scoping, a function is called in the environment of its caller (i.e., using the runtime stack of function calls). For a language like C or Decaf, the point is moot, because functions cannot be nested and can only access their local variables or global ones, but not the local variables of any other functions. But other languages such as Pascal or LISP allow non-local access and thus need to establish scoping rules for this. If inside the **`Binky`** function, you access a non-local identifier **`x`**, does it consider the static structure of the program (i.e., the context in which **`Binky`** was defined, which may be nested inside other functions in those languages)? Or does it use the dynamic structure to examine the call stack to find the nearest caller who has such a named variable? What if there is no **`x`** in the enclosing context—can this be determined at compile time for static scoping? What about dynamic? What kinds of data structure are necessary at compile-time and run-time to support static or dynamic scoping? What can you do with static scoping that you can't with dynamic or vice versa? Over time, static scoping has won out over dynamic— what might be the reasoning it is preferred?

Object Oriented Issues

Consider the two distinct pieces of a class definition: its interface and its implementation. Some object-oriented languages tie the two together and a subclass inherits both. In such a case, the subclass has an "is-a" relationship with its parent, and an instance of the subclass can be substituted wherever an instance of the superclass is needed: it is a subtype.

A language may support inheritance of implementation without interface, via a feature like C++ **`private`** inheritance. The new class has the behavior internally, but it is not exposed to the client. A **`Stack`** class might use **`private`** inheritance from a generic **`Vector`** class to get all the code for managing a linear collection but without allowing clients to call methods like **`insertAt`** that defeat the LIFO discipline. In such a case, the **`Stack`** is **not** a subtype of a **`Vector`**. (A similar feat could also be provided with composition of a **`Vector`** as an instance variable inside the **`Stack`** class).

It is also possible to inherit interface without implementation through a feature such as Java or Decaf interfaces. An *interface* is a listing of method prototypes only. There is no code, no instance variables, nothing more than declarations establishing the function signatures. When a class declares that it implements the interface, it is required to provide the implementation for all of the required methods. In this case, the relationship between the interface and the class is in the agreement to adhere to a behavior but not shared implementation. The class is a subtype of the interface type.

In a common form of inheritance, a derived class inherits both interface and implementation from the parent. Often a copy-based approach is used to implement inheritance. The storage for each derived class or subtype contains all the information needed from the parent because it was copied directly into the object.

Another issue in object-oriented languages is polymorphism. We saw this feature earlier when discussing ML, but the term takes on a slightly different meaning in a language like C++ or Java. Polymorphism in C++ refers to the situation in which objects of different classes can respond to the same messages. For example, if we have classes for boat, airplane, and car, the objects of these classes might all understand the message **`travel`**, but the action will be different for each class. In C++, polymorphism is implemented using *virtual functions*. Methods of the same name can have different definitions in different classes. For example, consider this excerpt from a hypothetical drawing program:

```
class Shape {
   public:
      virtual void Draw();
      virtual void Rotate(int degrees);
   ...
};

class Rect: public Shape {
   public:
      void Draw();
      void Rotate(int degrees) {}
   ...
};

class Oval: public Shape {
   public:
      void Draw();
      void Rotate(int degrees) {}
   ...
};
```

If we have an array of different shape objects, we can rotate all of them by placing the following statement inside a loop that traverses the array:

```
shapes[i]->Rotate(45);
```

We are rotating all the objects in the list without providing any type information. The receiving object itself will respond with the correct implementation for its shape type.

The primary difference between **`virtual`** functions and non-**`virtual`** functions is their binding times. Binding means associating a name with a definition or storage location. In C++, the names of non-**`virtual`** functions are bound at compile time. The names of **`virtual`** functions are bound at run-time, at the time of the call to the method. To implement this, each **`virtual`** method in a derived class reserves a slot in the class definition record, which is created at run-time. A constructor fills in this slot with the location of the **`virtual`** function defined in the derived class, if one exists. If it does not exist, it fills in the location with the function from the base class.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.

J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985.