Cimple

Language Reference Manual

Graham Barab (gmb2154)
Shankara Pailoor (sp3485)
Panchampreet Kaur (pk2506)

Introduction----------

The Cimple programming language is a statically-typed, low-level compiled language that is syntactically similar to C. Its purpose is to offer the fast execution speeds of C while simplifying the process of writing and maintaining code by offering additional language constructs that free the programmer to organize logic in a variety of ways. These constructs include classes with inheritance, and anonymous functions with closures, further expanding and modularizing the ways in which the programmer may define the functionality and organization of code.

Comparison with C----------

Inheritance

Unlike C, Cimple allows user-defined custom data-types (more usually called classes) to benefit from code-reuse, by inheriting the members of other classes. This follows a typical parent-child hierarchy observed in most object-oriented languages. Inheriting from other structs makes use of the `extends` keyword, such that a child struct "extends" the functionality of another struct, which is its parent. Here is an illustration of how this works:

```
struct Person {
            string name;
            Person(string theName)
            {
                  name = theName;
            }
      }
      struct Student extends Person{
            string uni
            Student(string name, string theUni)
            {
                  super(name);
                  uni = theUni;
            }
      }
```

Another keyword worth noting here is `super`. It allows access to a parent struct's methods and can only be used within the struct definition. In the above example, the constructor of the parent class is invoked by calling `super(name);`.
To access a method, this could be written: `super.getName();`

Interfaces

An interface is an abstract type which defines a *Method Set,* which is a collection of methods encapsulated in scope. It is a structure which enforces certain properties on an object, which implements one. The receiver for this method set is *empty*. In particular all interfaces have the same receiver.

```
Interface :
     "Interface" Identifier MethodSet

Ex:
     interface pet {
               void feed();
               void walk();
     }
```

Method Sets

A method set is a list of methods belonging to the same receiver. A method set is defined internally

Ex:
```
{
   ReturnType1 (Receiver) func_name1();
   ReturnType2 (Receiver) func_name2 ();
}
```

The `Receiver` must begin with a capital letter or maybe empty. It is a semantic error for it to be empty for a non interface type.

Methods

A method is a function with a receiver associated with it.

Ex: `ReturnType (Receiver_opt) func_name ()`

Automatic Variables and Heap Variables

As in C, variables can be stored on the stack ("automatic variables") or on the heap. Automatic variables, as the name implies, are automatically deallocated upon exiting scope. Heap-allocated variables, however, do not. A variable can be stored on the heap using the `make` keyword, followed by the name of the type. If the type is a custom struct, a constructor method call can also be used to initialize the data members to their default values. The memory allocated for heap variables is not deallocated until a statement of the form `clean <id>` is used, where `<id>` is an identifier that refers to a heap-allocated variable.

Anonymous functions

These are a new feature in Cimple, not found in C. These allow for a more intuitive scheme of declaration of function arguments to other functions than what would be done in C, using a function pointer instead. They provide compactness to the program and enhance the readability. More is discussed about these in a later section.

Lexical Conventions----------

Identifiers

These are strings of alphanumeric characters, which refer to virtual memory locations that hold variable data of a specific type. They can also refer to function names. When an identifier is initially declared, it must be preceded by the name of the type of data that it represents. The identifier itself should necessarily start with a letter, and can otherwise be composed of letters and digits as well as the special character '_' (underscore).

Keywords

These are words that hold a special meaning for the Cimple compiler, and are thus reserved to only be used in a particular context. Here is a list of the Cimple keywords:

```
auto
register
static
extern
typedef
void
char
short
int
long
float
double
signed
unsigned
const
volatile
struct
union
case
default
if
else
switch
while
do
for
goto
continue
break
return
func
string
```

```
true
false
func
extends
implements
make
clean
super
```

Literals

Literals are supposed to be values that are interpreted exactly the way they are written, unlike identifiers which can be interpreted as some other value (which they have been assigned to hold). Literals come in different types in Cimple:

Integer literals : An integer literal is simply any sequence of digits. It is of type int, unless it is suffixed with 'l' or 'L' to indicate the long data type, 'u' or 'U' to indicate an unsigned int. It may also be written in hexadecimal form by prepending "0x" and restricting the following characters to hexadecimal digits, namely [0-9] and [a-f] or [A-F]. Additionally, octal notation may be used by prepending a leading '0' to the literal.
- Examples: 324, 8, 90000, 037, 0xF, 037

Floating-point literals :- A floating-point literal is a sequence of digits that include either a decimal point, an optionally signed exponent (indicated by 'e' or 'E'), or both. The type of a floating-point literal is double unless suffixed with 'f' or 'F', in which case it is a float.
Examples: 54.5, 986.3, 0.65f, 1.2e11

Character literals :- A character constant is a single alphanumeric, punctuation and special ASCII character enclosed in single quotation marks. Additionally, certain control characters such as tab and newline may be indicated by a sequence of two characters: a backslash followed by another letter, all within single quotation marks.
Examples: 'a', 'B', '9', '&', '%', '\t', '\n', '\r'

String literals :- A string literal is a sequence of alphanumeric characters and control characters as specified in the "Character literals" section. It is enclosed within double quotation marks.
Examples: "Cimple is simple to use."
Boolean literals :- A boolean literal is represented by one of two keywords: "true" or "false".
Examples: true, false

- Separators
  These are, as the name suggests, characters which are used to denote separation between tokens. It should be noted that these are single-character values. The separators used in Cimple are: {}, [], ().

Operators----------

Operators are symbols that command the language to carry out specific mathematical, relational or logical computations over a set of data items to produce a final result. Following is the list of all the operators that Cimple understands. Operator precedence is specified in the “Expressions” section.

Unary Arithmetic Operators

+

The value of an expression formed by the unary plus and its operand is the positive value of its operand.

−

The value of an expression formed by the unary minus and its operand is the negated operand.

++

This is can be used as either a postfix or prefix unary operator that acts on an identifier (variable name) representing an integral value. In the prefix case, the value of identifier is incremented, and the expression formed by the operator and the operand is equal to this new value. In the postfix case, the value of the identifier is also incremented, but the value of the expression is equal to the value of the identifier before having been incremented.

−−

As with the unary increment operator, the unary decrement operator takes an identifier of integral type as operand, and has prefix and postfix versions. As both a prefix or postfix operator, the value of the operand is decremented, but the value of the expression formed by the prefix operator and its operand is the new decremented value, whereas the value of the expression formed by the postfix operator and its operand is the value prior to decrementing.

Unary Bitwise Operators

~

The one’s complement operator inverts the bit pattern of its integral-typed operand.

Binary Arithmetic Operators

Note: All arithmetic operators take one operand of arithmetic type on each side of the operator. The type of the expressions that result from a binary arithmetic operator depends on any promotions applied to any of its operands.

*

The value of the expression formed by the multiplication operator is the product of the operands.

/

The value of the expression formed by the division operator is the quotient of the operands.

+

The value of the expression formed by the addition operator is the sum of its operands, unless the operands are of type `string`, in which case the result of the expression is a new string that is the concatenation of the operands.

–

The value of the expression formed by the subtraction operator is the difference of its operands.

Binary Bitwise Operators

<<

The left shift operator takes two integral operands. The value of the expression in binary is the bit pattern of the left operand shifted left by the number of bits specified by the right operand. The new least significant bits are 0.

>>

The right shift operator takes two integral operands. The value of the expression in binary is the bit pattern of the left operand shifted right by the number of bits specified by the right operand. The new most significant bits are 0.

&

The bitwise “and” operator takes two integral operands. The value of the expression is an integer whose binary representation is a comparison of the values of matching bit positions in the operands that can be expressed in the following truth table.

0 0 -> 0
0 1 -> 0
1 0 -> 0
1 1 -> 1

|

The bitwise “or” operator takes two integral operands. The value of the expression is an integer whose binary representation is a comparison of the values of matching bit positions in the operands that can be expressed in the following truth table.

0 0 -> 0
0 1 -> 1
1 0 -> 1
1 1 -> 1

^

The bitwise “exclusive or” operator takes two integral operands. The value of the expression is an integer whose binary representation is a comparison of the values of matching bit positions in the operands that can be expressed in the following truth table.

0 0 -> 0
0 1 -> 1
1 0 -> 1
1 1 -> 0

Unary Logical Operators

`!`

The logical “not” operator evaluates to true if its operand is false, and evaluates to false if its operand is true.

Binary Logical Operators

`&&`

The logical “and” operator evaluates to true if both of its boolean-typed operands evaluate to true. Otherwise, the expression has a value of false.

`||`

The logical “or” operator evaluates to true if either of its boolean-typed operands evaluate to true. Otherwise, the expression has a value of false.

Note: All expressions formed by logical operators and their operands are of boolean type - “true” or “false”.

Comments----------

Multi-line comments in Cimple are anything which is enclosed within the character-pairs `/*` and `*/`. A single-line comment is any text that begins with the characters `//`

```
/* This is a multi-
line comment */

// Whereas this is a single line comment
```

Comments contain information which is ignored by the compiler, but that is helpful to the programmer writing and maintaining the code to which it refers. Comments cannot be nested.

Data Types----------

These refer to the forms of data that Cimple can accept to perform operations on. Following is a list of data types that Cimple recognizes:

Integers

There are 16, 32, and 64-bit integer types available in Cimple, which can be specified with the keywords `short`, `int` and `long`, respectively.

Floating-Point

There are 32 and 64-bit floating-point types in Cimple, which can be specified with the keywords `float` and `double`, respectively.

Characters

The `char` type is an 8-bit integer with the intended use of storing a single character.

Typed Pointers

A typed pointer is an unsigned integer whose size is dependent on the target architecture, and that points to a (potentially variable) location in memory which holds a value of that type.

Arrays

Arrays are 0-indexed lists of same-typed values. They are indicated by writing the number of items the array should hold, enclosed by square brackets at the end of the name of an identifier, i.e. `int integerList[10];` Individual elements from the array can later be accessed by specifying its index into the array enclosed by square brackets at the end of the name of the identifier, i.e.

```
// Set the tenth element in the array to the value 10
integerList[9] = 10;
```

Strings

Strings are built-in data types in Cimple, and are implemented as arrays of characters. It is specified by the `string` keyword.

Void

The void type indicates an absence of a value, and is used primarily as a return type of a function to indicate that it is used purely for it's side effects, such as printing. It can also be used to indicate the nonexistence of arguments in a function definition.

Structs

Structures are the primary means of creating custom data types. Structs encapsulate data members of arbitrary type and functions that act on or with these data members. Their size is variable and dependent on the data members defined by the user. Constructors can be defined that initialize the state of a struct by implicitly being called whenever an instance of that struct is allocated.

Function Types----------

Named Functions

A named function is one that is declared in global scope in the following manner:

```
<type> <id> (<type> <id>, <type> <id>,...);
```

Where the first <type> indicates the type of value (of those listed in the "Data Types" section) returned by the function, the first <id> is a valid identifier which serves as a name for the function, and a set of parentheses that enclose a comma separated list of the identifiers and types of the parameters.

The definition of the function begins with the same format as the declaration without the trailing semicolon, and is followed by a pair of curly braces that enclose a sequence of statements that define what the function does.

Anonymous Functions

Anonymous functions can be defined inline, without the need for an identifier. Because this means that a function could begin its definition within some scope other than global, it is valid for the scope of the anonymous function to reference variable names (identifiers) declared outside of its scope. In this case, the compiler will copy the values of these variables to a data structure that is passed as an implicit argument to the function. An anonymous function is defined as follows:

func (<type>)(<type> <id>, <type> <id>, …) { <statements> }

The first set of parentheses can only have one type indicated, and represents the return type of the anonymous function. The second set of parentheses contains the parameter list, as with the named functions.

Expressions (lowest to highest precedence)————————

- Assignment-Expression
  - a = b; a += 1;
- Conditional Expression
  - (a > b) ? c : d
- Logical-AND-Expression
  - a && b
- logical-OR-expression
  - a || b
- And-expression
  - a & b
- Inclusive-Or-expression
  - a | b
- Equality-Expression
- Additive Expression
  - Left recursive, expands to multiplicative expressions
- Multiplicative Expression
  - Left recursive, expands to primary expressions
- Primary Expression
  - Consists of integer, float, string, char constants
  - Parenthetical expressions

Statements

- Expression Statements
  - `expression;`
- Jump Statements
  - `goto identifier;`
  - `continue;`
  - `break;`
  - `return expression_opt;`
- Selection Statement
  - `if (expression) statement`
  - `if (expression) statement else statement`

        - `switch (expression) statement`
    - Labeled Statement
        - `case expression : statement`
        - `default : statement`
        - `identifier: statement`
- Loops
    - `while( expression ) statement`
    - `do statement while( expression )`
    - `for (expression_opt; expression_opt; expression_opt) statement`

    Here we have `expression_opt` defined in our grammar as

```
expression_opt:
      expression
      | none
```

Scope

Scope is delimited by a set of curly braces `{ }`. Any variables declared after a left curly brace go “out of scope” after the corresponding right curly brace. That is to say, it is an error to refer to an identifier declared in some scope from the enclosing scope.