CGC Reference Manual

Lieyang Chen, Tianze Huang, Zhuoxuan Li, Fanhao Zeng
{lc3548, th2887, zl2890, fz2320}@columbia.edu

February 23, 2021

Contents

1 Overview

The CGC programming language is a language built upon C language, but comes with many handy syntax and features, and most importantly added garbage collection feature. In addition to the three primitive data types (int, char, float) and pointer similar to C, we provides a new built-in object type `Array` that could be used to wrap all primitive data types.

- Garbage collection

  The CGC language provides a garbage collector which will find unused objects on the heap and deletes them to free up memory without the user knowing it.

- Handy syntax/features

  The CGC language will provide built-in methods for `Array` type, such as `len`, `append`, `pop`. The CGC also supports control flows like range `For` Loops

2 Lexical Conventions

2.1 Identifiers

- Valid identifiers maybe include ASCII letters, decimal digits and underscore.
- An valid identifier must begin with a letter and can not be a CGC keyword.

Examples:

```
// valid identifier
int studentId
void get_bar2() {}

// invalid identifier
int static = 3
void 2getId() {}
```

2.2 Operators

- int: +, -, *, /, ==, <, >, <=, >=, =, &&, ||, !
- char: +, -, *, /, ==, <, >, <=, >=, =, &&, ||, !
- float: +, -, *, /, <, >, <=, >=, =, ==

- Array: =, ==, [ ]

2.3 Keywords

- `for, if, else, elseif, return, int, float, char, Array, void, const, class, static`
- Keywords are reserved identifiers that can not be used as variable names.

2.4 Literals

Literals represents strings or one of CGC's primitive types: int, float, char.

- int: any sequence of integers between 0 and 9
- float: a number in decimal form, a fraction, and an exponent
- char: a single character within single quotes
- string: a sequence of characters enclosed in single or double quotation marks

Examples:

```
// int literals
-5
0
130

// float literals
2.5
1e23

// char literals
'a'
'3'

// string literals
"Hi there!"
"When is iphone 13 coming out?"
```

2.5 Separators

- Parentheses are used to override the default precedence of expression evaluation.
- Semicolons are used to indicate the end of an expression.

Examples:

```
int x = (a + b) * c; // Using () to override default operator precedence

x = 3; y = x + 1; // Using ; to indicate the end of an expression

/*
Technically you can write all code in one line.
*/
```

2.6 Comments

- Single-line comments use double backslashes (//). Multi-line comments are denoted with a /* */ notation.

Examples:

```
int x = 3 // This is a single-line comment
/*
And this is a multi-line comment
*/
```

3 Data Types

CGC is a statically typed language, so all of the variables must be declared first before they can be used and all of them have a data type at compile time. CGC is also a strongly typed language, it does not support variable type coercion and implicit casts. These features enable CGC detects program errors easily in the compile time.

There are two kinds of data types in CGC language, primitive type and object type. The primitive data types of CGC are `int`, `char`, `float`. Besides them, CGC provides a new built-in object type `Array` that could be used to wrap all primitive data types as well as itself. `Array` in CGC is quite similar to the vector in C++.

3.1 Primitive Data Types

Primitive data types in CGC are predefined and reserved as keywords. Specifically, there are 3 primitive types:

`int`: the `int` data type is a 32-bit signed two's complement integer. `int` is able to represents integer ranges from -2147483648 to 2147483647. `int` variables are considered as signed by default, unsigned is not a keyword in CGC.`int` is the

only one type in CGC used to stores integer, CGC does not support keywords like `short` or `long` to specify different variable length.

`float`: The float data type is a 64-bit IEEE 754 floating point.

`char`: The char data type is a single unsigned 8-bit ASCII character.
The operators of `int`, `float` and `char` are listed as below, detailed specification of these operators are listed at part 5.
– The numerical comparison operators <, >, <=, and >=
– The numerical equality operators ==
– The multiplicative operators * and /
– The additive operators + and -
– The unary minus operator –
– The unary logical negation operator !
– The logical AND operator &&
– The logical OR operator ||
– The assignment operator =
– The comma operator ;

3.2 Object Data Types

CGC provides a new built-in object type `Array` that could be used to wrap all primitive data types as well as itself (e.g., `c` below). The `Array` object is allocated on heap, so it is always created during run time. The garbage collection mechanism described in Section 4 will be used for memory management of `Array` object. The `Array` object has three built-in member variables: `size`, `ref_count` and `addr`. The `ref_count` variable will be briefly discussed in this section and more details will be covered in Section 4. The `size` variable refer to the array size. The `addr` variable is the address where the actual array begins on heap. In CGC, the `Array` object manipulated by user is actually an address to a block of memory on heap which contains the above three members. Such a block contains another address(the `addr` member) which references the actual array. The actual array is allocated on heap as well. There can be two cases when initializing an `Array` variable:

- When assigning an array of literal(e.g., `"1234"` in example below) to an `Array` variable, a block of memory whose size equals to the total size of `size, ref_count` and `addr` will be allocated on the heap. The address of this memory block will be assigned to the `Array` variable. The `size` member will be set to the size of literal array and `ref_count` will be set to 1. Then another block of memory with size equal to the size of literal array will be allocated on heap and the address of this memory block will be assigned to `addr`. Finally, the elements of literal array will be copied to this memory block.

- When assigning another `Array` variable to a new `Array` variable, the new `Array` variable will simply be set to the address that references the same memory location as the assigning variable.

The `Array` is type mutable. For example, the statement `b[1] = '3'` is legal in CGC because the `Array` object is manipulating its own array instead of some array on the text section. When assigning a literal array to a already existing `Array` variable, the size of the existing `Array` object will be modified if the size of the literal array is different from its member variable `size`. Namely, the memory allocated on heap will be re-allocated if the sizes are different. The member variable `size` and `addr` will be modified accordingly, but the member variable `ref` will remain unchanged.

Array examples:

```
Array<char> a = "1234"; // Case1: Initialize array of characters,
    a.size = 4, a.ref = 1
Array<char> b = a; // Case2: Assign address of memory location
    referenced by a to b, b.size = 4, b.ref=2
b[1] = '3'; // Array indexing and mutability
Array<Array<int>> c = {{1,2},{1,2,3}}; //Array of integer arrays
```

CGC also provides a series of built-in methods for Array object data type. Details of these methods can be found in part 6 when describing CGC's standard library.

– The Array length method `len()`
– The Array element append method `append()`
– The Array element pop method `pop()`
– Insert an element into specific position `insert()`
– Remove an element from specific position `remove()`

3.3 User-Defined Data Types

In CGC, users can define their own data types by using the keyword `class`. The `class` keyword is used to define new data types and describe how they are implemented. In the body of a `class`, users are able to define its members (variables, methods). The CGC `class` does not support complex Object-Oriented programming features such as inheritance.

Similar to `Array`, `class` use the same built-in parameter called `ref_count` to record the number of references that an instance have. Memory on heap will be released when the `ref_count` of an instance is 0.

```
//class
class foo {
    void get_bar() {
        return bar;
    }
```

```
    int bar;
}
int main{
    foo foo_example;
    foo_example.bar = 10;
    int result = foo_example.get_bar();
    printf("%d", result);// result has a value of 10 now
    printf("%d", foo_example.ref_count);// prints 1
}
//foo_example.ref_count = 0 here, corresponding memory space
    released.
```

3.4 Memory Model

In CGC, all function calls are by defalut "pass by value".In CGC language, user can using `class` and `Array` to get a space in memory. CGC also provides a reference-counting garbage collector to take care of the memory allocated on the heap. As a result, explicit manually freeing memory are not supported in CGC.

3.5 Declarable Storage Class

The CGC language maintains a keyword `static`, which is a declarable storage class. A variable without a storage class declaration in a block, will be treated as a local variable of that block, and it will be discarded once the end of the block has been executed.In contrast, a static local variable is also local in a block, but it retains its value independently and will not be discarded on exit of the block.

```
int Example1()
{
    int a = 0;
    a = a + 1;
    return a;
    //default local variable's scope only within the block
}

int Example2()
{
    static int a = 0;
    a = a + 1;
    return a;
    //static local variable persists until end of the program
}

int main()
{
```

```
    for(int i = 0; i < 3; i = i + 1)
    {
        printf("Round %d\n", i + 1);
        printf("auto variable = %d\n", Example1());
        printf("static variable = %d\n", Example2());
    }
//By default, a local variable initialized every iteration
//static variable contains previous value and increment
    return 0;
}
```

---

4 Garbage Collection

There are two main reasons for which dynamically memory allocation is useful in C language:

- When you want to share a variable across different function scopes, so that all functions are referring to the same memory region of that variable.

- When you want to pass a huge object back to caller or forward it to callee by its reference(namely pointer) instead of copying value.

Based on these observations, the implementation of dynamically allocated memory can help speed up program by reducing the overhead which is caused during function call. However, the downside of it is that it is always annoying for the programmer to manually free these allocated memory. Therefore, we introduce the garbage collector in CGC.

`Array` and `class` objects in CGC almost support all dynamically memory allocation implementations. Since these objects are allocated on heap, the garbage collection in CGC is especially designed for these built-in data types. The main mechanism behind GC(garbage collection) is counting the number of references to the memory region on the heap. To achieve this implementation, we make `Array` and `class` objects have a built-in parameter called `ref_count` to trace the number of references they currently have.
The `ref_count` member will be incremented by 1 when:

- The object is passed to other function as an argument.
- The object is returned to the calling function as an argument and the returned argument is assigned to a variable in the calling function.
- The object is assigned to a variable.

The `ref` member will be decremented by 1 when:

- The variable which references the memory on heap leaves its current scope.

When the last variable which references the memory on heap leaves its scope, the The `ref_count` member becomes 0 and the heap memory will be released. For the sake of simplicity, an `Array` or `class` object is passed by reference instead of copying its memory.

Examples:

```
fun(Array<int>[] array) //pass by reference
{
    printf("%d", array.ref_count);
    return array;
}

int main()
{
    Array<int> array;
    printf("%d", array.ref_count);// prints 1
    Array<int> array2 = fun(array);// prints 2
    printf("%d", array.ref_count);// prints 2
    fun(array);// prints 3
    printf("%d", array.ref_count);// prints 2
    {
        Array<int> array2 = array;
        printf("%d", array.ref_count);// prints 3
    }
    printf("%d", array.ref_count);// prints 2
    Array<int> array3 = array;
    printf("%d", array3.ref_count);// prints 3
}// array.ref=0, memory referenced by array is released
```

5 Statements and Expressions

5.1 Statements

A CGC program is made up of a list of statements. A statement is one of the following:

```
Expression
Class declaration
Function definition
Return statement
If statement
If-else statement
For loop
```

Blocks of statements can be enclosed using curly braces.

5.1.1 If-Else Statements

If statements consist of a condition (an expression) and a series of statements. The series of state- ments is evaluated if the condition evaluates to True. If the condition evaluates to False, either the program continues or an optional else clause is executed.

Examples:

```
if condition{
    //series of statements
}
else{
    //other series of statements
}
```

5.1.2 For Statements

There are two ways to write a `For` Loop in CGC. The first kind of `For` Loop iterate over an `Array`. It consists of a looping variable, an instance of a `Array`. The series of statements is evaluated for each item in the `Array` in order, where the looping variable is assigned to the first element of the list for the first iteration, the second for the second iteration, etc. to be looped over, and a series of statements.

```
Array<int> x = {1,2,3}
//second format
for (int i = 0; i < x.len(); i = i + 1) {
    x[i] = x[i] + 1;
}
```

5.2 Expressions and Operators

Expressions are part of statements that are evaluated into expressions and variables using operators. These can be arithmetic expressions or a function call.

5.2.1 Unary Operators

CGC has two types of operators: unary operators and binary operators.
In CGC, the unary operators are NEG and NOT, i.e ”-” and ”!”.

```
int x = -1 // represents the negative number 1
```

NOT represents negation of a boolean expression. It can also be applied to integers and floats, where any non-zero number is considered to be True, and False otherwise.

```
int a = 1;
int b = 1;
int c = 0;
if (a == b){
    //series of statements that will be evaluated
}
if (!(a == c)){
    //series of statements that will be evaluated
}
if (!a) {
   //series of statements that will not be evaluated
}
```

5.2.2 Binary Operators

The following list describes the binary operators in CGC. All of the operators act on two expressions. Unless otherwise stated, they act only on primitives.

1. Assignment Operator
   The assignment operator stores values into variables. This is done with the "=" symbol. The value of the right side is stored in the variable on the left side.

   ```
   int x = 1; // x = 1
   int y = 0; // y = 0
   y = x; // y = 1
   ```

2. Arithmetic Operator

   a. Addition is performed on two values of the same type. Addition between different types is not permitted.

   ```
   int x = 1;
   int y = 2;
   int z = x + y; // z = 3;
   float f = 1.2 + 2.3; // f = 2.5
   ```

   b. Subtraction is performed on two values of the same type. Subtraction between different types is not permitted.

   ```
   int x = 1;
   int y = 2;
   int z = x - y; // z = -1;
   float f = 2.5 - 2.3; // f = 0.2
   ```

c. Multiplication is performed on two values of the same type. Multiplication between different types is not permitted.

```
int x = 2;
int y = 3;
int z = x * y; // z = 6;
float f = 2.0 * 2.5; // f = 5.0
```

d. Division is performed on two values of the same type. Division between different types is not permitted.

```
int x = 4;
int y = 3;
int z = y / x; // z = 1;
float f = 5.0 / 2.0; // f = 2.5
```

3. Relational Operators
Relational operators determine how the operands relate to another. There are two values as inputs and outputs either true or false. The operators includes: $==$ $>, <, >=, <=, \&\&, \|\|$

```
int x = 1;
int y = 2;
int z = 3;
if (x > y){
//Evaluate to be false
}
 if (x < y){
//Evaluate to be true
}
 if (x == y){
//Evaluate to be false
}
if (x == y || x < z){
//Evaluate to be true
}
```

5.3 Operator Precedence

The following is an operator precedence table for all the operators, from lowest to highest precedence.

5.4 Functions

A function is a type of statement. It takes in a list of arguments and return one value. The body of a function is delimited by curly braces. The return type is

Operator	Meaning	Associativity
;	Sequencing	Left
=	Assignment	Right
||	Or	Left
&&	And	Left
==	Equality	Left
>, <, >=, <=	Comparison	Left
+ -	Addition/ Subtraction	Left
* /	Multiplication/ Division	Left
!	Not	Right

specified before to the function name. An example of function declaration is as follows:

```
int add_one(int x)
{
    int y = x + 1;
    return y;
}
```

5.5 Function Calls

Functions are called using their identifier and arguments inside parentheses. For example:

```
int x = 1;
add_one(x);
```

5.5.1 Variable Assignment from Functions

A function may be called as the right value of a variable assignment. The variable would be assigned to the return value of the function.

```
int x = 1;
int z = add_one(x); // z would be 2
```

6 Standard Library

6.1 Array

CGC's built-in `Array` data type associates with many library methods:

- Get the size of `Array` object x: x.len().

- Append a new integer y to the end of `Array<int>` object x: x.append(y).
- Pop the the last element of `Array<int>` object x and return it: int y = x.pop().
- Get the element of `Array<int>` object x at a given position 0: int y = x[0].
- Remove the element of `Array<int>` object x at a given position 0: x.remove(0).
- Insert an integer y to `Array<int>` object x at a given position 0: x.insert(0, y).

6.2 printf()

Examples:

```
    int main()
    {
        printf("%d", 1);// prints 1
        printf("%c", 'c');// prints c
        printf("%s", "string");// prints string
    }
```

7 Examples

```
// An example of garbage collection of class object.
class example1
{
    int examp_method()
    {
        return value;
    }

    int value;
}

int main()
{
    example1 examp_a;
    example1 examp_b;
    example1 tmp;
    examp_a.value;
    printf("%d", examp_a.ref_count); // prints 1
    printf("%d", examp_b.ref_count); // prints 1
    a = tmp;
    // prints 2, now the space pointed by previous examp_a is released.
```

```
    printf("%d", tmp.ref_count);

} // space of tmp and examp_b are also released now.

// A simple binary search algorithm in CGC
class example2
{
  int binarySearch(int target)
  {
    if (array[0] == target) return 0;
    // binary search
    for (int left, right = len[array] - 1; left < right; )
    {
      int mid = left + (right - left) / 2;
      if (array[mid] == target) return mid;
      else if (array[mid] < target) left = mid + 1;
      else right = mid;
    }
    return -1;
  }

  void setArray(Array<int> a)
  {
    array = a;
  }

  Array<int> array;

};
```