VC Language Definition

Version 13.0 -- 21 February, 2017

COMP3131/9102: Programming Languages and Compilers

Semester 1, 2017

1. Introduction

VC is a Variant of C. It consists of mostly a subset of C plus some language features from Java.

This language, designed for use in an undergraduate compiler subject, consists of a few primitive types, one-dimensional arrays, control structures, expressions, compound statements (i.e., blocks) and functions, which are largely taken from C.

The major Java aspects of the language are as follows:

1. VC provides a boolean data type, **boolean**, borrowed from Java. In VC, all boolean expressions must evaluate to a value of type **boolean**, which is either `true` or `false`. C does not have a boolean type while the standard C++ supports a boolean type. In C and C++, any nonzero value represents `true` and any zero value represents `false`.
2. In VC, as in Java, the operands of an operator are guaranteed to be evaluated in a specific evaluation order, namely, from left to right. In C and C++, the evaluation order is left unspecified. In the case of `f() + g()`, for example, VC dictates that `f` is always evaluated before `g`.
3. In VC, as in Java, assignment coercion (i.e., implicit type conversion) from **float** to **int** is not allowed. The following VC code, which is legal in C, must trigger a compile-time error:

```
float f = 1.0; // 1.0 is a float literal in VC
int   i = f;   // error: incompatible =
```

Apart from C-style arrays, every VC function is a syntactically legal Java method. Therefore, only the language features unique to C are described in detail.

2. Grammar Notation

The syntax of VC is specified by a context-free grammar (CFG) written in the notation of EBNF (extended Backus-Naur Form). An EBNF is effectively BNF plus regular expressions.

A context-free grammar (CFG) consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. We follow the convention that the left-hand side symbol of the first production in a grammar is the *start symbol*.

Starting from the start symbol, a CFG specifies a language, namely, the set of all possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

In this specification, terminals are represented either by a sequence of bold face alphanumeric characters (e.g., **return** is a reserved keyword in a return statement) or by a sequence of characters enclosed in double quotes (e.g., "<=" represents the less-than-and-equal-to operator). Sometimes, when it is clear from the context that a symbol is a terminal, the double quotes are omitted.

Nonterminals are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by an arrow ->. The other variations are to use := or ::= instead of ->.

The right-hand side of a production may contain an iteration construct written in the notation of regular expressions to denote multiple occurrences of a symbol (or a group of symbols in parentheses). The following iteration constructs are used:

```
( X )?  an optional occurrence of X
( X )*  zero or more occurrences of X
( X )+  one or more occurrences of X
```

If X represents a single symbol, the grouping parentheses "(" and ")" above are often omitted.

The VC grammar is given as follows:

```
program               ->  ( func-decl | var-decl )*

// declarations
func-decl             -> type identifier para-list compound-stmt
var-decl              -> type init-declarator-list ";"
init-declarator-list-> init-declarator ( "," init-declarator )*
init-declarator       -> declarator ( "=" initialiser )?
declarator            -> identifier
                      |  identifier "[" INTLITERAL? "]"
initialiser           -> expr
                      |  "{" expr ( "," expr )*  "}"

// primitive types
type                  -> void | boolean | int | float

// identifiers
identifier            -> ID

// statements
compound-stmt         -> "{" var-decl* stmt*  "}"
stmt                  -> compound-stmt
                      |  if-stmt
                      |  for-stmt
                      |  while-stmt
                      |  break-stmt
                      |  continue-stmt
                      |  return-stmt
                      |  expr-stmt
if-stmt               -> if "(" expr ")" stmt ( else stmt )?
for-stmt              -> for "(" expr? ";" expr? ";" expr? ")" stmt
while-stmt            -> while "(" expr ")" stmt
break-stmt            -> break ";"
continue-stmt         -> continue ";"
return-stmt           -> return expr? ";"
expr-stmt             -> expr? ";"


// expressions
expr                  -> assignment-expr
assignment-expr       -> ( cond-or-expr "=" )* cond-or-expr
cond-or-expr          -> cond-and-expr
                      |  cond-or-expr "||" cond-and-expr
cond-and-expr         -> equality-expr
                      |  cond-and-expr "&&" equality-expr
equality-expr         -> rel-expr
                      |  equality-expr "==" rel-expr
                      |  equality-expr "!=" rel-expr
rel-expr              -> additive-expr
                      |  rel-expr "<" additive-expr
                      |  rel-expr "<=" additive-expr
                      |  rel-expr ">" additive-expr
```

```
                     |  rel-expr ">=" additive-expr
additive-expr       -> multiplicative-expr
                     |  additive-expr "+" multiplicative-expr
                     |  additive-expr "-" multiplicative-expr
multiplicative-expr -> unary-expr
                     |  multiplicative-expr "*" unary-expr
                     |  multiplicative-expr "/" unary-expr
unary-expr          -> "+" unary-expr
                     |  "-" unary-expr
                     |  "!" unary-expr
                     |  primary-expr

primary-expr        -> identifier arg-list?
                     | identifier "[" expr "]"
                     | "(" expr ")"
                     | INTLITERAL
                     | FLOATLITERAL
                     | BOOLLITERAL
                     | STRINGLITERAL

// parameters
para-list           -> "(" proper-para-list? ")"
proper-para-list    -> para-decl ( "," para-decl )*
para-decl           -> type declarator
arg-list            -> "(" proper-arg-list? ")"
proper-arg-list     -> arg ( "," arg )*
arg                 -> expr
```

3. Program Structure

A VC program consists of a sequence of zero or more variable and function declarations residing in one single file. Thus, VC does not support separate compilation.

Communication between the functions is by parameters and values returned by the functions.

4. Lexical Structure

This section describes the character set, comment conventions and token set in the language.

4.1 Character Set

A VC program consists of a sequence of characters from the ASCII character set. Blank, tab, formfeed (i.e., the ASCII FF), carriage return (i.e., the ASCII CR) and newline (i.e., the ASCII LF) are *whitespace characters*. Lines are terminated by the ASCII CR, LF or CR LF. The two characters CR immediately followed By LF are counted as one line terminator, not two. The line terminator characters are ignored except that they may delimit another token or an end-of-line comment. This definition of lines can be used to determine the line numbers produced by a VC compiler.

4.2 Comments

There are two kinds of comments:

- A traditional comment:

  ```
  /* text */
  ```

  All the text from /* to */ is ignored as in C, C++ and Java.
- An end-of-line comment:

  ```
  // text
  ```

All the text from // to the end of the line is ignored as in C++ and Java.

As in C, C++ and Java, the following rules are enforced:

- Comments do not nest.
- /* and */ have no special meaning in comments that begin with //.
- // has no special meaning in comments that begin with /*.

A formal grammar for these two kinds of comments can be found in §3.7, Java Language Specification

4.3 Token Set

A VC program is a sequence of five kinds of tokens: identifiers, keywords, operators, separators and literals. Two tokens can be separated by a whitespace, a comment, an operator or a separator.

All the tokens of VC are enumerated below:

- Identifiers
  An *identifier* is an unlimited-length sequence of letters, digits and underscores, the first of which must be a letter or underscore:

  ```
  identifier -> letter (letter | digit)*
      letter -> A | B | ... | Z | a | b | ... | z | _
       digit -> 0 | 1 | 2 | ... | 9
  ```

  VC is case-sensitive, meaning that two identifiers composed of the same characters that differ only in case (e.g., "position" and "Position" are distinct).

- Keywords
  The following character sequences are reserved as *keywords* and cannot be used as identifiers:

  boolean break continue else for float if int return void while

- Operators
  The following 14 tokens are the *operators*:

Name	Tokens
Arithmetic Operators	+ - * /
Relational Operators	< <= > >=
Equality Operators	== !=
Logical Operators	|| && !
Assignment Operator	=

  where `!` is the unary logical complement operator and `+` and `-` are overloaded to represent the unary plus and minus, respectively.

- Separators
  The following six ASCII characters are the *separators* (i.e., punctuators):

  ```
  {   }   (   )   [   ]   ;   ,
  ```

- Literals
  A *literal* is a source representation of a value of either an int type, float type, boolean type or string type.

An *integer literal* is always expressed in decimal (base 10), consisting of a sequence of at least one digit:

```
int_literal -> digit (digit)*
      digit -> 0 | 1 | 2 | ... | 9
```

An integer literal is of type **int**.

A *floating-point literal* has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part and an exponent. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer. At least one digit, in either the whole number or the fraction part, and either a decimal point or an exponent are required. All other parts are optional.

```
  float_literal -> digit* fraction exponent?
                | digit+ .
                | digit+ .? exponent
         digit -> 0 | 1 | 2 |... | 9
      fraction -> . digit+
      exponent -> (E | e) (+ | -)? digit+
```

Examples of floating-point literals are:

```
1.2   1.  .1   1e2   1.2E+2   1.2e-2  .1E2
```

A floating-point literal is of type **float**.

A *boolean literal* consists of either `true` or `false`, formed from ASCII letters. While `true` and `false` might appear to be keywords, they are technically boolean literals. The two boolean literals cannot be used as identifiers.

A *string literal* consists of zero or more characters enclosed in double quotes. The quotes are not part of the string, but serve to delimit it.

The complete set of escape sequences supported is:

`\b`	`backspace`
`\f`	`formfeed`
`\n`	`newline`
`\r`	`carriage return`
`\t`	`horizontal tab`
`\'`	`single quote`
`\"`	`double quote`
`\\`	`backslash`

It is a compile-time error for a newline to appear after the opening " and before the closing matching ".

A string literal is of type **string**. Strings can only be used in the two built-in functions `putString()` and `putStringLn()` as described in Section 9.

5. Types and Values

VC is a strongly-typed language, which means that the types of all variable and expressions are known at compile time. Types limit the values that a variable can hold and that an expression can produce and limit the operations supported on those values. Strong typing facilitates error detection at compile time.

The types of the VC language are divided into two categories: primitive types and array types. The primitive types are **void, boolean, int** and **float**. The array types are **boolean[], int[]** and **float[]**, which denote the arrays of **boolean, int** and **float**, respectively.

Function declarations can specify **void** in place of a return value to indicate that the function does not return a value.

5.1 The `void` Type and Values

The **void** type specifies an empty set of values. It is used only as the type returned by functions that generate no value.

5.2 The boolean Type and Values

The **boolean** type represents a logical quantity with two possible values: `true` and `false`. There are five boolean operators:

```
==   !=   &&   ||   !
```

A *boolean expression* is an expression that evaluates to `true` or `false`. Boolean expressions determine the control flow in **if** and **while**. Only boolean expressions can be used in these control flow statements. Boolean expressions can also determine the control flow in **for** statements.

5.3 The int Type and Values

The values of type **int** are 32-bit signed two's complement integers in the following ranges:

```
-2147483648 --- 2147483647
```

The following operators can act on integer values:

```
+   -   *   /   <   <=   >   >=   ==   !=
```

The first four operators always produce a value of type **int**. The remaining six operators always result a value of type **boolean**.

Here, + and - represent both the binary and unary plus and minus operators, respectively.

The logical operators `&&`, `||` and `!` cannot act on integer values.

5.4 The float Type and Values

A float value is a 32 bit single-precision number. The exact values of this type are implementation-dependent. The following operators can act on floating-point values:

- The binary arithmetic operators `+`. `-`, `*` and `/`, which result in a value of type **float**.
- The unary plus and minus operators `+` and `-`, which result in a value of type **float**.
- The relational operators `<`, `<=`, `>` and `>=`, which result in a value of type **boolean**.
- The equality operators `==` and `!=`, which result in a value of type **boolean**.

The logical operators `&&`, `||` and `!` cannot act on floating-point values.

5.4 Array Types and Their Values

VC permits only one-dimensional arrays. The following features are from C:

- Array subscripts start at 0. If an array has *n* elements, we say *n* is the *length* of the array; the elements of the array are referenced from 0 to *n-1*.

- A subscript can be any integer expression, i.e., any expression of type **int**.
- The values (or precisely, *r-values*) of an array type are *pointers* to array objects. In other words, the value of a variable of an array type is the *address* of element zero of the array.

Compared to C, however, one-dimensional arrays in VC are more restrictive :

- The element type of an array can only be **boolean, int** or **float**.
- An array variable itself (without the associated square brackets []) cannot appear in any VC expressions (partially because VC does not yet support C-style pointers). The only exception is that an array name itself can be used as an argument in a function call. One example is given below to illustrate this point:

  ```
  void f(int x[]) { }
  int main() {
  int x[] = {1, 2};
  f(x);  //  OK
  x + 1; //  ERROR
  }
  ```

6. Variables

In VC, every variable declaration is also a variable definition. That is, a variable declaration specifies not only the type for an identifier but also reserves storage for the identifier.

All variables must be declared before use.

In the language, every variable declared by the programmer must be **boolean**, **int**, **float**, **boolean[]**, **int[]** or **float[]**.

There are three kinds of variables: global variables, local variables and function parameters. Their declarations and initialisations are described below.

6.1 Global Variables: Declarations and Initialisations

Global variables are declared outside a function using *var-decl*. For examples, these are legal variable declarations:

```
boolean b;                     // a variable of type boolean
int i;                         // a variable of type int
float f = 2.5;                 // a variable of type float
boolean ba[] = {true, false}; // a variable of type boolean[]
int ia[] = {1, 2};             // a variable of type int[]
float fa[100] = {1.0, 2 };     // a variable of type float[]
```

A global variable is created at the program startup and destroyed when the program completes. A global variable without an initialising expression is initialised to *a default value of 0* as follows:

Type	Default Value
boolean	false
int	0
float	0.0
boolean[]	{ false, false, ... }
int[]	{ 0, 0, ... }

float[]	{ 0.0, 0.0, ...}

In a global array variable declaration, the length of the array must be specified unless its *init-declarator* specifies an initial value for the variable being declared.

An *array initialiser* is written as a comman-separated list of expressions, enclosed by "{" and "}". It is defined by the following production in the VC grammar:

```
initialiser  -> "{" expr ( "," expr )* "}"
```

The following specification from K & R's C book (2nd ed) applies to VC (p. 219):

```
    The initializer for an array is a brace-enclosed list of initialisers for
    its members. If the array has unknown size, the number of initializers
    determines the size of the array, and its type becomes complete. If the
    array has a fixed size, the number of initializers may not exceed the
    number of members of the array; if there are fewer, the trailing members
    are initialized with 0.
```

In addition, the expressions in an array initialiser are executed from left to right in the textual order. The *n*th initialiser specifies the value for the *n-1*st array element. Each expression must be assignment-compatible (Section 7.1) with the array's element type, or a compile-time error occurs.

6.2 Local Variables: Declarations and Initialisations

Local variables are declared inside the body of a function (i.e., inside a block) also using *var-decl*.

The declarations of local variables and the associated initialisations with initialisers are exactly the same as that of global variables.

A local variable declared in a block is created whenever the flow of control enters the block and destroyed as soon as the flow of control leaves the block. Unlike a global variable, a local variable may be associated with more than one object during the execution of the program. A local variable that is not initialised with an initialising expression is deemed to contain garbage.

The initialisation for local arrays using array initialisers follows exactly the same rules as global arrays.

6.3 Function Parameters: Declarations and Initialisations

Function parameters name argument values passed to a function. Function parameters are declared using *para-dec*. The production used dictates that a function parameter cannot be initialised at its declaration using an initialising expression.

Each time when a function is invoked, each of its parameters is initialised with the value of the corresponding argument from the function call.

A parameter of array type has exactly the same meaning as in C:

- When an array variable is passed (as an argument) to a function, what is passed is the location (i.e., address) of element zero of the array (which, by definition, is the value of the array variable). Thus, any modifications made by the callee on the array will be visible on the corresponding argument.
- The length of an array in a parameter declaration is irrelevant. Thus,

  ```
  void f(int a[10]) { }
  ```

  is equivalent to:

  ```
  void f(int a[]) { }
  ```

7. Expressions

The usual arithmetic, relational, equality, logical and assignment expressions are provided. The following table summarises all the operators and the rules for precedence and associativity.

Operator	Precedence	Associativity
+ - !	1	right to left
* /	2	left to right
+ -	3	left to right
< <= > >=	4	left to right
== !=	5	left to right
&&	6	left to right
||	7	left to right
=	8	right to left

The operators on the same row have the same precedence and the rows are in order of decreasing precedence.

The + and - in the first row are the unary plus and minus operators, respectively.

7.1 Type Coercions

An expression is a *mixed-mode expressions* if it has operands of different types. In VC, as in C, C++ and Java, mixed-mode expressions are permitted. When an operator has operands of different types, they will be converted *implicitly* by the compiler to a common type according to some prescribed rules. Such an implicit type conversion is known as a *type coercion*. This should be contrasted with an explicit type conversion, also known as casts, which do not exist in VC. In general, the term "type conversion" is meant to be an explicit type conversion.

In VC, the operands of the following operators:

```
+    -    *    /    <    <=    >    >=    ==    !=
```

can have either type **int** or **float**. If one operand is **float**, the compiler will implicitly convert the other to **float**. Therefore, if at least one of the operands of a binary operator is of type **float**, then the operation is a floating-point operation. If both operands are of type **int**, then the operation is an integral operation.

Assignment coercions occur when the value of an expression is assigned to a variable -- the value of the expression is converted to the type of the left side, which is the type of the result.

The following coercions are permitted:

- If the type of the variable is **int**, the expression must be of the type **int** or a compile-time error occurs.
- If the type of the variable is **float**, the expression must have either the type **int** or **float** or a compile-time error occurs.
- If the type of the variable is **boolean**, the expression must be of the type **boolean** or a compile-time error occurs.

If the type of an expression can be coerced to the type of a variable by these rules, the expression is said to be *assignment-compatible* to the variable.

Since an argument of a function call is an expression, type coercions also take place when arguments are passed to functions.

7.2 Evaluation Order

Like Java but unlike C and C++, VC requires the operands of an operator to be evaluated in a specific *evaluation order*, namely, from left to right. Therefore, the left-hand operand of a binary operator must be evaluated first before any part of the right-hand operand is evaluated.

Thus,

```
int main() {
  int i = 3;
  j = (i = 4) * i;
  putIntLn(j);
}
```

prints

```
16
```

It is not permitted for this program to print `12` instead of `16`.

Similar, in a function call (called a method call in Java), the actual parameters must be evaluated from left to right.

Thus,

```
int foo(int i, int j) {
  return i * j;
}

int main() {
  int i = 3;
  putIntLn( foo(i=4, i) );
}
```

prints

```
16
```

It is not permitted for this program to print `12` instead of `16`.

Every operand of an operator must be evaluated before any part of the operation itself is performed. The two exceptions are the logical operators && and ||, which are still evaluated from left to right, but it is guaranteed that evaluation will stop as soon as the truth and falsehood is known. In the case of &&, the right operand is evaluated only if the left operand evaluates to `true`. In the case of ||, the right operand is evaluated only if the left operand evaluates to `false`. This is known as the *short-circuit evaluation*.

8. Statements and Control Flow

VC supports a number of statements in C, including **if** statement, **for** statement, **while** statement, **break** statement, **continue** statement, **return** statement, expression statements, and compound statements (blocks).

The if Statement

In an **if** statement:

```
if (expression)
```

```
     statement_1
else
     statement_2
```

expression, which must be of the type **boolean**, is first evaluated. If it is `true`, *statement*$_1$ is executed. If it is `false`, *statement*$_2$ is executed.

As in C, C++ and Java, the VC language suffers from the so-called dangling-else problem, illustrated by the following example:

```
if (i >= 0) if (i <= 9) putBoolLn(true); else putBoolLn(false);
```

There are two possible interpretations (i.e, parse trees) illustrated below by indentations:

```
if (i >= 0)
  if (i <= 9)
    putBoolLn(true);
  else
    putBoolLn(false);
```

```
if (i >= 0)
  if (i <= 9)
    putBoolLn(true);
else
  putBoolLn(false);
```

Like C, C++ and Java, the VC decrees that an else must belong to the innermost **if**. Therefore, the first interpretation will be used.

The while Statement

In a **while** statement:

```
while (expression)
     statement
```

expression, which must be of the type **boolean**, is first evaluated. If it is `true`, then *statement* is executed and *expression* is re-evaluated. This evaluation/execution cycle repeats until *expression* becomes `false`.

The for Statement

The **for** statement

```
for (expression_1; expression_2; expression_3)
     statement
```

is equivalent to

```
expression_1;
while (expression_2) {
    statement
    expression_3
}
```

except for the behavior of **continue**. In the **while**, a **continue** statement causes the test part, i.e., *expression*$_2$, to be executed immediately. In the **for**, control passes to the increment step, i.e., *expression*$_3$.

Each of the three expressions in a **for** statement is optional so that the traditional infinite loop in C and Java is supported:

```
for (;;) {
  ...
}
```

The break Statement

This has the same semantics as in C.

The continue Statement

This has the same semantics as in C.

The return Statement

A **return** statement aims at transferring control to the caller of the function that contains it.

A **return** statement with no *Expr* must be contained within a function that is declared using the keyword **void** or a compile-time error occurs.

A **return** statement with an *Expr* must be contained within a function that is declared to return a value or a compile-time error occurs.

Syntactically, a value returned from a function cannot be of an array type.

Expression Statements

An expression becomes a statement if it is followed by a semicolon. For example, the four expressions

```
i = 1
foo(1,2)
i + 2
100
```

become statements as in:

```
i = 1;
foo(1,2);
i + 2;
100;
```

However, the last two are illegal Java expression statements.

Blocks

Braces { and } are used to group variable declarations and statements together into a *compound statement*, i.e., *block*. The VC grammar implies that the statements inside a function must be grouped in a block, and multiple statements after **if**, **else**, **while** and **for** must also be in a block. Unlike Java, the variable declarations, if any, in a block must precede the statements, if any, in that same block.

9. Functions

In VC, every function declaration is also a function definition. That is, A *function declaration* specifies the name of the function, the type of the return value and the number and types of the arguments that must be supplied in a call to the function as well as the body of the function.

VC does not support function overloading. Thus, a function must be defined exactly once.

The return type of a function must be a primitive type.

Finally, the language consists of 11 simple I/O functions that provide input and output for VC programs:

```
int getInt():
    reads and returns an integer value from the standard input
void putInt(int i):
    prints the value of the integer i to the standard output
void putIntLn(int i):
    same as putInt except that it also prints a newline
float getFloat():
    reads and returns a floating-point value from the standard input
void putFloat(float f):
    prints the value of the float f to the standard output
void putFloatLn(float f):
    same as putFloat except that it also prints a newline
void putBool(boolean b):
    prints the value of the boolean b to the standard output
void putBoolLn(boolean b):
    same as putBoolLn except that it also prints a new line
void putString("a string"):
    prints the value of the string to the standard output
void putStringLn("a string"):
    same as putStringLn except that it also prints a new line
void putLn():
    prints a newline to the standard output
```

10. Parameter Passing -- Call by Value

Like C, all arguments (including arrays) in VC are passed "by value." The called function is given the its value in a temporary variable. Thus, the called function cannot alter the variable in the calling function in any way. Recall that the value of an array variable is the address of element zero of the array.

11. Scope Rules

Scope rules govern declarations (defining occurrences of identifiers) and their uses (i.e., applied occurrences of identifiers).

The identifier tokens specified in the productions for *func-decl*, *var-decl* and *formal-para* in the VC grammar are defining occurrences. All other uses of identifier tokens in the grammar are applied occurrences.

The *scope* of a declaration is the region of the program over which the declaration can be referred to. A declaration is said to be *in scope* at a point in the program if its scope includes that point.

A *block* is is a language construct that can contain declarations. There are two types of blocks in the VC language:

- The outermost block is the entire program.
- Each compound statement forms a block by itself.

VC exhibits *nested block structure* since blocks may be nested one within another. Therefore, there may be many scope levels:

- All function declarations, built-in or user-defined, and all global variables make up the outermost block at scope level 1.
- Variable declarations inside an inner block are local to that block. Every inner block is completely enclosed by another block. If enclosed by the outermost block, that inner block is said to be at scope level 2. If enclosed by a level-2 block, that inner block is at scope level 3,

and so on.

There are two additional rules on the scope restrictions:

1. No identifier can be defined more than once in the same block. This implies that an identifier cannot represent both a global variable and a function name simultaneously.
2. *Most closed nested rule:* For every applied occurrence of an identifier in a block, there must be a corresponding declaration, which is in the smallest enclosing block that contains any declaration of that identifier.

The following VC program:

```
 1 int f() {
 2   return 200;
 3 }
 4 int i = 1;
 5 int main() {
 6   int main;
 7   main = f();
 8   putIntLn(i);
 9   {
10     int i = 2;
11     int main;
12     int f;
13     main = f = 100;
14     putIntLn(i);
15     putIntLn(main);
16     putIntLn(f);
17   }
18   putIntLn(main);
19 }
```

compiles and prints:

```
1
2
100
100
200
```

In this program, there are three scope levels:

Declaration	Level	Scope
putIntLn (built-in)	1	Entire Program
f (line 1)	1	lines 1 - 11 and 18 - 19
i (line 4)	1	lines 4 - 9 and 18 - 19
main (line 5)	1	line 5
main (line 6)	2	lines 6 - 10 and 18 - 19
i (line 10)	3	lines 10 - 17
main (line 11)	3	lines 11 - 17
f (line 12)	3	lines 12 - 17

The variable `main` declared in line 6 is said to *hide* the function declaration `main` in line 5. The variable `main` declared in line 11 hides the variable declaration `main` in line 6. The variable `i` declared in line 10 hides the global variable `i` in line 4. The variable `f` declared in line 12 hides the function declaration `f` in line 1.

The scopes of the declarations f in line 1, i in line 4 and main in line 6 are not contiguous. Such gaps are known as scope holes, where the corresponding declarations are hidden or invisible.

As a matter of style, it is advised not to introduce variables that conceal names in an outer scope. This is the major reason why Java disallows a variable declaration from hiding another variable declaration of the same name in an outer scope. Therefore, the VC program above is a bad programming style.

12. The main function

Each VC program must have a `main` function of the form:

```
int main() { // no parameters are allowed
  ...
}
```

Due to the scope rules, the main function is usually the last function in a program.

Every VC program begins executing at the `main` function.

The `main` is not allowed to call itself recursively.

Jingling Xue