Stint

Language Reference Manual

Jiang Wu, jw3026
Ningning Xia, nx2120
Sichang Li, sl3484
Tingting Ai, ta2355
Yiming Xu, yx2213

Oct. 25th, 2012

Table of Contents

1. Introduction

The programming language *Stint* is a text-processing language, which contains a useful collection of built-in string manipulation functions. *Stint* provides efficient solutions for manipulations and conversions between the most common-used data types: string and int. When processing textual data and test file, *Stint* shows its advantages in several aspects:

- ✓ Intelligently distinguish between string and int.
- ✓ Provide a simple and direct way to do mathematic operations for numbers in string without extracting them and transferring data type.
- ✓ Define more effective text-manipulation via operators. Meanwhile, maintain traditional string features and functions.
- ✓ Make input and output functions more convenient.

2. Lexical Conventions

2.1 Identifiers

An identifier can only include 26 alphabets in lower or upper case (a-z, A-Z), digits (1-9), and underline (‘ _ ’); the first character must be alphabets in lower case.

2.2 Comments

Double-slash (“ // ”) is the only comment sign, and it only works for single line.

```
//This is a comment
open "data.txt";  // can append to a statement
```

2.3 End-of-Statement

Semicolon (“ ; ”) is used to indicate the end of one statement.

2.4 Keywords

The following identifiers are reserved as keywords/special function and may not be used otherwise:

```
int     string   boolean
if      else     return
while   open     close
false   true     std
eof     break    void
```

2.5 Constants

In *Stint*, there are 3 types of constants: integer constant, string constant and boolean constant.

2.5.1 Integer Constants

An integer constant consists of a sequence of numbers without a decimal point.

```
int n = 100; // 100 is an integer constant
```

2.5.2 String Constants

A string constant is enclosed in double quote marks (“ " " ”).

```
// below are examples of string constants
string str = "This is a string";
string title = "Product Name\tPrince\t\n";
```

Stint also contains the following escape sequence as constants in order to represent some special characters such as new line, tab and backslash:

Table 1. Escape Sequence in String Constants

Character Name	Escape Sequence
Newline	\n
Horizontal tab	\t
Double quotation marks	\”
Backslash	\\

2.5.3 Boolean Constants

The reserved boolean constants are **`true`** and **`false`**.

```
boolean a = true;
boolean b = false;
```

3. Data Type & Conversion

3.1 Data Types

There are three types of data in *Stint* – integer, string and boolean.

3.1.1 Integer

In *Stint*, the only supported integer type is **`int`**, which represents a sequence of digits. Integers are signed and with fixed size of 32 bits.

3.1.2 String

The string in *Stint* is defined as a sequence of ASCII characters enclosed in double quotes, e.g. “abc”. A string can have potentially unlimited length as long as the computing resource, e.g. memory, allows.

A string in *Stint* is a dynamic structure, which keeps a list of sub-strings or integers. *Stint* predefined some special operators in order to modify or search the sub-strings dynamically, e.g. “ `.< >` ” (extracting), “ | ” (splitting), “ # ” (highlighting), etc. (See details in Section 4.3)

3.1.3 Boolean

A boolean type can take either **`true`** or **`false`** as its value, which is used for condition determination.

Some main features of the three data types are illustrated in the table below.

Table 2. Summary of Data Type in Stint

Primitive Types	Length	Range	Default Value
int	32 bits	-2147483648 to 2147483647	0
string	>= 8 bits	Any permutation of ASCII characters	“” (empty)
boolean	8 bits	**`true`** or **`false`**	**`false`**

3.2 Type Conversion

In *Stint*, there are two types of conversions, i.e. from integer to string and from boolean to string, both of which are done implicitly. No explicit conversion is allowed. All other invalid type conversions, e.g. from string to integer or from integer to boolean, will lead to a compilation error.

3.2.1 Integer-to-String

An integer can be converted to a string implicitly, but not vice versa. After casting, the new variable name cannot be the same as the original one; otherwise there will be a compilation error.

```
int num = 12;
string snum = num; // snum = "12" after conversion.
```

```
string snum = "12";
int num = snum;     // this assignment is invalid
```

Besides assigning, the implicit conversion can be done automatically as well when an integer is used in a string operation.

```
int num = 12;
string snum = "abc" + num; // snum = "abc12"
```

Even though a string cannot be converted into an integer directly, this conversion can be done by using a special operator defined in *Stint*, i.e. " `.< >` ". (See details in Section 4.3)

3.2.2 Boolean-to-String

A boolean value can be converted to a string, but not the other way around.

```
boolean b = true;
string s= b;        // s = "true" when b is true
```

Trying to convert an integer or a string to a boolean will lead to a compilation error though it seems make sense.

```
int n = 1;
boolean b = n;      // Compilation error.
```

4. Expressions & Operators

4.1 General Expression

4.1.1 Assignment: *destin* = *source*

Copy and assign the value of *source* to *destin*.

4.1.2 Parenthesis: (*expression*)

Expression in parenthesis will be assigned a higher precedence than those isn't in.

4.1.3 Equality: *expr1* == *expr2* *expr1* != *expr2*

Check whether *expr1* and *expr2* have the same value.

4.1.4 Input Stream: >> *source expr*

4.1.5 Output Stream: << *destin expr*

The subject of stream is indicated by *source*/*destin* that can be file name or **`std`** for standard I/O. Read a whole line from *source* to *expr* or write the content of *expr* to *destin*.

4.1.6 Function Call: *func-name* (*arg1, arg2, …*)

4.2 Numeric Operator

4.2.1 Arithmetic Operator: *expr1* + *expr2* *expr1* – *expr2*
expr1 * *expr2* *expr1* / *expr2*

Basic arithmetic operation for integer: add, subtraction, multiplication, and division.

4.2.2 Relational Operator: *expr1 > expr2* *expr1 >= expr2*
expr1 < expr2 *expr1 <= expr2*

Compare expr1 and expr2 with above relations. Return a Boolean value.

4.3 String Operator

4.3.1 Position Indicator: *@ position*

Indicate specific position where the operation will be done. Use with string operator “+” and “-” (See examples in 4.3.2/4.3.3).

4.3.2 Append (Insert): *expr1 + expr2*
expr1 + expr2 @ pos

Append *expr2* to the end of *expr1*. When using “@”, append *expr2* after the character indicated by *pos* of *expr1* (use as insert).
Use a number smaller than the lowest index (negative number) or larger than the highest index will put *expr2* at the head/tail of *expr1*.

```
str = “ab” + “c”;        // str = “abc”
str = “ab” + “c” @ 0;    // str = “acb”
```

4.3.3 Delete: *expr1 – expr2*
expr1 – expr2 @ pos

Delete the first *expr2* in *expr1*. When using “@”, delete the first *expr2* occurring after the character indicated by *pos* in *expr1* (not include this character).
The operator will give *expr1* back if no matching of *expr2* found in *expr1*.

```
str = “cabc” – “c”;          // str = “abc”
str = “cabc” – “c” @ 1;      // str = “cab”
```

4.3.4 Sub-string Extractor: [*index*]
[*index* , *length*]

Get the sub-string of a single character at *index*, or starting from *index* of *length*.

```
str = “abcd”; str = str[0];       // str = “a”
str = “abcd”; str = str[1, 2];    // str = “bc”
```

4.3.5 Integer Set Extractor: .< *index* >

Get the set of integer with *index*.

```
string str = “a12b56c”;
int integer = str.<0>;        // integer = 12
```

4.3.6 String Set Extractor: < *index* >

Get the set of string with *index*. Set of string may vary with user’s setting (see detail in 4.37/4.38).

4.3.7 String Splitter: *string* | *sep*

Change the set of *string* as sub-strings separated by *sep*. Return the amount of sets having been split.

```
string str = “this is a sentence in string”;
int num = str | “ ”;     // num = 6
string temp = str<0>;    // temp = “this”
temp = str<4>;           // temp = “sentence”
```

4.3.8 String Finder: *string # substr*

Change the set of *string* as all sub-strings equaling *substr*. Return the number of *substr* in *string*.

```
string str  = "this is his thesis";
int num = str # "is";  // num = 4
str<2> = "er";     // str = "this is her thesis"
```

4.3.9 String Remover: *~ expression*

Remove *expression* from the string it belongs to.

```
string str = "a12b56c";
~str.<0>;          // str = "ab56c"
~str[0, 3];        // str = "6c"
```

4.4 Boolean Operator

4.4.1 And: *expr1 && expr2*
4.4.2 Or: *expr1 || expr2*
4.4.3 Not: *! expression*

4.5 Precedence

The precedence of above operators is typically in the following order from high to low. All three boolean operators are in the same order, so they are not included in the table below.

Table 3. Precedence of Stint Operators

Precedence	**Operator**
1 (highest)	( )
2	| , #
3	[ ], < >, .< >
4	*, /
5	+, -
6	>, >=, <, <=, ==, !=
7 (lowest)	=, <<, >>

5. Declarations

5.1 Variable Declaration

Variables must be declared before they are used in the program. A variable declaration has the following form:

var_type var_name;

The *var_type* can be **int**, **boolean** or **string**. The *var_name* can be any valid identifier. If a variable is declared, in the following assignment, value assigned to the variable must have exactly the same type as declared. Otherwise, it's a syntax error. A single semicolon must be followed by the declaration.
Variables can also be initialized during the declaration. A declaration with initialization has the following form:

var_type var_name = expression;

The expression must have exactly the same type as *var_type*. Otherwise, it's a syntax error.

5.2 Function Declaration

A function declaration has the following form:

return-type function-name (type parameter1, type_parameter2….)

The detail of this part is talk in section 7.1.

6. Statements

6.1 Expression Statement

An expression statement is composed of primary expressions separated by a single semicolon at the end of each expression.

6.2 If Statement

If statement consists of keywords **`if`** and **`else`**. It has the following two varieties:

```
if ( expression ) {
    statement
}
if  ( expression ) {
    statement1
} else {
    statement2
}
```

The expression must be of **`boolean`**. Statements must be surrounded by open and closed curly bracket. In the first case, if the *expression* is evaluated to **`true`**, then *statement* is executed. Otherwise statements after the *if* statement is executed. In the second case, if the *expression* is evaluated to **`true`**, then *statement1* is executed, otherwise *statement2* is executed.

6.3 While Statement

While statements consists of keyword **`while`** and it allows a statement to be executed for any number of times. It has the following format:

```
while ( expression ) {
    statement
}
```

The expression must be **`boolean`**. Statements must be surrounded by open and closed curly bracket. The *expression* is evaluated before the execution of the statement and *statement* will be executed until the expression is evaluated to **`false`**.

6.4 Break Statement

Break statement consists of keyword **`break`**. It's used to jump out of the while loop. It's followed by a single semicolon. The following is an example of using break statement:

```
while ( true ) {
    break;
}
```

6.5 Open Statement

Open statement consists of keyword **`open`**. It's used to open a file. It takes a string as the file name. It has the following format:

open filename

6.5 Close Statement

Close statement consists of keyword **`close`**. It's used to close a file. It takes a string as the file name. It has the following format:

close filename

6.7 Return Statement

Return statement consists of keyword **`return`**. A function must have a return statement to return its value to its caller. It can return an expression that is evaluated to type **`int`**, **`bool`** or **`string`**, or it can return nothing when the function uses **`void`** as its return type.

return expression;
return;

The return statement must be followed by a single semicolon.

7. Functions

7.1 Function Definition

A function in *Stint* is typically the same as a function in C or a method in Java. It takes a series of parameters and returns a value after execution of the code within the function.
The signature of a function is as follows:

return-type function-name (*type parameter1, type_parameter2….*)
{
statements
……
return value;
}

All functions have global scope within the program it was create. And the names of functions are unique among all the functions and variables.
The return type must be one among **`void`**, **`int`**, **`string`**, and **`boolean`**. We suggest to use **`boolean`** as the return type to replace **`void`**, **`true`** indicating execution success, otherwise **`false`**.
The body of a function must be included in a " `{ }` " just like C and Java do. The variables defined within the body of a function have local scope. That's to say, all variables have the same local scope. We don't recommend creating variables in the body of a function except for temporary variables. If a function with variable definitions within its body gets called multiple times, these variables will be reinitiated.
All program will start from **`boolean`** `main()` function. Return **`true`** means program terminates successfully.

7.2 Build-in Functions

- **string** `ReadFile(`**string** `filenme)`

  This function is used to get a line of content from a file certain opened. A pointer variable will add itself every time after this function is executed. If it has reached the end of a file, the return value became **eof**.

- **boolean** `PrintFile(`**string** `s,` **string** `filename)`

  This function is used to print String s to the end of a text file. It can be a file that hasn't been opened yet, the function will automatically do the rest of work.

- **string** `toUpperCase(`**string** `s)`

  The return string will be the same as input screen but in all upper case.

- **string** `toLowerCase(`**string** `s)`

  Similar as toUpperCase(), this one return string s with all characters in lower case.

- **boolean** `replceAll(`**string** `dest,` **string** `s)`

  This function is used to replace all the matched sections in dest to s.

8. Scope

In *Stint*, the scope is defined as the region within a program in which a certain identifier/function can be accessed.
All identifiers are of local scope, i.e. they can only be accessed within the function body in which it is declared. The local scope is confined by the nearest pair of curly brackets. Identifiers/variables cannot be accessed until declared.
Functions are of global scope from the position they are defined till the end of code. Function calls are possible as long as the target function has been defined before the current position.