C+ Language Reference Manual

By Alexander Stein & Eric Johnson

Columbia University
Dr. Stephen A. Edwards
COMS W4115 S2017 – Programming Langagues & Translators

Identifiers

In CPlus, identifiers are combinations of characters, numbers, and the special character '_'; they must begin with a letter, and they are case-sensitive. There are two types of identifiers:

(1) Value and Function Identifiers: these must begin with a lower-case letter; they represent variable values of all datatypes as well as function names. Defined formally as

```
        ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

(2) Struct Identifiers: these must begin with a capital letter; they represent globally-defined structure types, used to group several datatypes into a single record. Defined formally

```
        ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

Keywords

The reserved keywords in CPlus include types, control flow indicators, and built-in functions:

if	else	for	while	return
int	char	size_t	string	char
bool	void	true	false	sizeof
struct	NULL	printf	atoi	strdup
printb	print	printbig	malloc	free

Literals

Literals are the primitive values which represent the core data of any language, in this language there are only five

1 Integer Literals: Any combination of one or more numerals - `['0'-'9']+`
2 Character Literals: Any ASCII Character, including all specials - `['\x00'-'\x7F']`
3 Boolean Literals : Either of `true` or `false`.
4 String Literals: Any combination of characters found between "" marks; with supported escape sequences:

\ - escape character
\b – backspace character

\n – newline character
\f – form feed
\r – carriage return
\t – tab character

5 Separators: Used to help build the abstract syntax tree and determine semantics, discarded after parsing; in this language we use:

‘(‘ – LPAREN
‘)’ – RPAREN
‘{‘ – LBRACE
‘}’ – RBRACE
‘[‘ – LSQUARE
‘]’ – RSQUARE
‘;’ – SEMI
‘,’ – COMMA
‘/*’ – open comment
‘*/’ – close comment

Operators

There are three categories of operators in CPlus, which dictate all the mathematical, memory-access, and other logical functionality of the language:

(1) Binary Operators : these operate on two expressions, and the order of which expression is on the left or right is significant

+	addition of two signed 32-bit integers – returns 32-bit int
-	subtraction of two signed 32-bit integers – returns 32-bit int
*	multiplication of two signed 32-bit integers – returns 32-bit int
/	division of two signed 32-bit integers – returns 32-bit int
%	modulus of two signed 32-bit integers – retruns 32-bit int
==	physical equality of two primitive literals of same type – returns bool
!=	physical inequality of two primitive literals of same type – returns bool
<	less than of two 32-bit integers – returns bool
<=	less-than-equal-to of two 32-bit integers – returns bool
>	greater than of two 32-bit integers – returns bool
>=	greater-than-equal-to of two 32-bit integers – returns bool
&&	logical AND of two bools – returns bool
||	logical OR of two bools – returns bool

(2) Unary Operators : these operate on a single expression, and significance is placed on whether the operator comes before (prefix) or after (postfix) the expression

-	Applied before integer, makes it negative (cannot be chained together)
!	Applied after bool, inverts its value
++	Applied before or after integer, increases its value by one
--	Applied before or after integer, decrements its value by one

*   Applied before an identifier, retrieves its value from memory
&   Applied before an identifier, returns its memory address
->   Applied after struct identifier, retrieve struct from mem and get member
.   Applied after struct identifier, get member

(3) Assignment Operators : these operate an two expressions, the lefthand of which must evaluate to an *lvalue* and the righthand of which must evaluate to an *rvalue.* We have only implemented two for our simple purposes

=   store *rvalue* at the address *lvalue*
%=   *lvalue* must point to an integer, performs ‘m[lvalue] = m[lvalue] % rvalue’

Operator Precedence & Associativity

In decreasing order from top to bottom:

OPERATORS	**ASSOCIATIVITY**
*** & SIZEOF ()**	Right-to-left
() [] . -> ++ -- (POSTFIX)	Left-to-right
! ++ -- - (PREFIX)	Right-to-left
() (CAST)	Right-to-left
*** / %**	Left-to-right
+ -	Left-to-right
< > <= >=	Left-to-right
== !=	Left-to-right
&&	Left-to-right
||	Left-to-right
= %=	Right-to-left

Data Types

The data, this is the good stuff. What can we store, manipulate, understand? There are 8 datatypes in CPlus, six of which are primitive, and two of which are complex

Int	32-bit integer
Size_t	64-bit integer
Bool	Boolean values true or false
Char	any of the asci values described as charlit above
String	any string between two “” quotes described above
Void	The type we return when we don’t want to return anything
Struct(ID)	A struct with the name ID, cannot exist without ID
Pointer(typ)	A pointer to any of the above datatypes

It is worth noting that we have implemented arrays in CPlus, but have done so using pointers under the hood, instead of defining an explicit array type.

Declarations

CPlus requires that – inside of a function – all declarations (even with initializers) for any variable used later on in that function must take place first, before any operations. Declarations are explicitly defined in the Context Free Grammar section, and are summarized below:

(1) Literal Variables: Datatype followed by Value Identifier. Cannot directly declare a void variable. Legal variable declarations look like this:

```
int a; bool b; char c; string d; Node n; Edge e;
```

(2) Functions: Datatype followed by Function Identifier followed by LBRACE function-body RBRACE. Datatype can be either primitive or complex. Legal function declarations look like this:

```
int x { … }
bool* { … }
Node** { … } – where Node was previously defined as a struct
```

(3) Structures: `struct` keyword followed by a Struct Identifier then LBRACE struct-body RBRACE SEMI. Note that the struct body can only contain un-initialized Literal Variable declarations. Legal struct declarations look like this:

```
struct Node {
        int x;
        char* y;
        Edge z; /* previously declared struct Edge */
};
```

(4) Pointers: Datatype followed by * followed by Value Identifier. Can chain multiple * together in a single declaration. Legal pointer declarations look like this:

```
int* x;
bool** y;
Node* np;
```

(5) Arrays: Arrays are not explicit datatypes, and they resolve to pointers when referenced. They do, however, have their own declarations as syntactic sugar. Datatype – possibly complex – followed by Value Identifier the RSQUARE integer-literal LSQUARE. The integer-literal must be greater than 0 and is not optional. Variable length arrays are not supported. Legal array declarations look like this:

```
int a[10];
bool* b[10];
Node* nodeIndices[100];
```

(6) Initialization: With the exception of structs and arrays, all of the above declarations can optionally come paired with initializers. Furthermore, declarations of the same type can be chained together in a list through commas. Memory allocation is also possible in the initialization of a declaration. Legal initializations for declarations look like this:

```
int x = 5;
int y = (300 * 21 + 4 ) / 2;
int*x = (int*) malloc(20*sizeof(int)); /* defacto 20-int array */
Node* nodes = (Node*) malloc(sizeof(Node));
int a = 5 , b = 6, q = x + 1;
```

Scope Rules

Standard static scoping rules apply, i.e. it is determined spatially based on position within the code at compile time.

Context-Free Grammar

TERMINALS: in uppercase bold

EOF	Epsilon	ID	LPAREN	RPAREN
LBRACE	RBRACE	COMMA	STRUCT	STRUCT_ID
SEMI	INT	SIZE_T	BOOL	STRING
CHAR	VOID	TIMES	LSQUARE	RSQUARE
ASSIGN	RETURN	IF	ELSE	FOR
WHILE	MOD_ASSIGN	OR	AND	EQ
NEQ	LT	GT	GEQ	LEQ
NOT	MINUS	INC	DEC	DOT
ARROW	PRINTF	PRINT	PRINTB	PRINTBIG
MALLOC	FREE	STRDUP	ATOI	PLUS
TIMES	DIVIDE	MOD	LITERAL	STRINGLIT
CHARLIT	TRUE	FALSE	AMP	SIZEOF
NULL				

nonterminals: in lowercase italics

program	decls	declaration	func_decl	struct_decl
typ	formals_opt	formal_list	declaration_list	stmt_list
init_declarator_list	expr	add_expr	init_declarator	stmt
selection_statement	iteration_statement	expr_opt	assignment_expression	logical_or_expr
postfix_expr	logical_and_expr	equality_expr	relational_expr	add_expr
unary_expr	cast_expr	unary_operator	postfix_expr	built_in_expr
actuals_list_opt	primary_expr	actuals_list	mult_expr	

Productions:

program →
decls **EOF**

decls →
decls declaration
| *decls func_decl*
| *decls struct_decl*
| **Epsilon**

func_decl →
typ **ID LPAREN** *formals_opt* **RPAREN LBRACE** *declaration_list stmt_list* **RBRACE**

formals_opt →
Epsilon
| *formal_list*

formal_list →
typ **ID**
| *formal_list* **COMMA** *typ* **ID**

struct_decl →
STRUCT STRUCT_ID LBRACE *declaration_list* **RBRACE SEMI**

typ →
INT
| **SIZE_T**
| **STRING**
| **BOOL**
| **CHAR**

| **VOID**
| **STRUCT_ID**
| *typ* **TIMES**

declaration_list →
Epsilon
| *declaration_list declaration*

declaration →
typ init_declarator_list **SEMI**

init_declarator →
ID
| **ID LSQUARE** *add_expr* **RSQUARE**
| **ID ASSIGN** *expr*

init_declarator_list →
init_declarator
| *init_declarator_list* **COMMA** *init_declarator*

stmt_list →
Epsilon
| *stmt_list stmt*

stmt →
expr **SEMI**
| *selection_stmt*
| *iteration_stmt*
| **RETURN SEMI**
| **RETURN** expr **SEMI**
| **LBRACE** *stmt_list* **RBRACE**

selection_stmt →
IF LPAREN *expr* **RPAREN** *stmt* (%prec **NOELSE**)
| **IF LPAREN** *expr* **RPAREN** *stmt* **ELSE** *stmt*

iteration_stmt →
FOR LPAREN *expr_opt* **SEMI** *expr* **SEMI** *expr_opt* **RPAREN** *stmt*
| **WHILE LPAREN** *expr* **RPAREN** *stmt*

expr_opt →
Epsilon
| *expr*

expr →
assignment_expr

assignment_operator →
ASSIGN
| **MOD_ASSIGN**

assignment_expr →

logical_or_expr
| *postfix_expr assignment_operator expr*

logical_or_expr →
logical_and_expr
| *logical_or_expr* **OR** *logical_and_expr*

logical_and_expr →
equality_expr
| *logical_and_expr* **AND** *equality_expr*

equality_expr →
relational_expr
| *equality_expr* **EQ** *relational_expr*
| *equality_expr* **NEQ** *relational_expr*

relational_expr →
add_expr
| *relational_expr* LT *add_expr*
| *relational_expr* GT *add_expr*
| *relational_expr* LEQ *add_expr*
| *relational_expr* GEQ *add_expr*

cast_expr →
unary_expr
| **LPAREN** *typ* **RPAREN** *cast_expr*

unary_operator →
NOT
| **MINUS**

unary_expr →
postfix_expr
| *unary_operator postfix_expr*
| **INC** *postfix_expr*
| **DEC** *postfix_expr*

postfix_expr →
built_in_expr
| *postfix_expr* **INC**
| *postfix_expr* **DEC**
| *postfix_expr* **LPAREN** *actuals_list_opt* **RPAREN**
| *postfix_expr* **LSQUARE** *postfix_expr* **RSQUARE**
| *postfix_expr* **DOT ID**
| *postfix_expr* **ARROW ID**

built_in_expr →
primary_expr
| **PRINTF LPAREN** *actuals_list_opt* **RPAREN**
| **PRINT LPAREN** *actuals_list_opt* **RPAREN**
| **PRINTB LPAREN** *actuals_list_opt* **RPAREN**
| **PRINTBIG LPAREN** *actuals_list_opt* **RPAREN**
| **MALLOC LPAREN** *actuals_list_opt* **RPAREN**

| **FREE LPAREN** *actuals_list_opt* **RPAREN**
| **ATOI LPAREN** *actuals_list_opt* **RPAREN**
| **STRDUP LPAREN** *actuals_list_opt* **RPAREN**

actuals_list_opt →
Epsilon
| *actuals_list*

actuals_list →
expr
| *actuals_list* **COMMA** *expr*

add_expr →
mult_expr
| *add_expr* **PLUS** *mult_expr*
| *add_expr* **MINUS** *mult_expr*

mult_expr →
cast_expr
| *mult_expr* **TIMES** *cast_expr*
| *mult_expr* **DIVIDE** *cast_expr*
| *mult_expr* **MOD** *cast_expr*

primary_expr →
LPAREN *expr* **RPAREN**
| **LITERAL**
| **STRINGLIT**
| **CHARLIT**
| **TRUE**
| **FALSE**
| **ID**
| **AMP** primary_expr
| **TIMES** primary_expr
| **SIZEOF LPAREN** *typ* **RPAREN**
| **NULL**