C π

Edward Garcia (ewg2115), Naveen Revanna (nr2443)
Niket Kandya (nk2531), Sean Yeh (smy2112)

Introduction

C is the lingua franca of the computer world. It is a general purpose programming language and often used in systems level programming. In this project we will be implementing C π (Cpi) which is a subset of the C language. It will be designed to compile to ARM V6 assembly with the target platform being the Raspberry Pi (RPi) . C π will use the GNU assembler(as/gas), linker(ld) and linaro cross toolchain (gcc-linaro-arm-linux-gnueabihf-raspbian) for assembly to binary code generation on the RPi.

Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Comments

The characters /* introduce a comment, which terminates with the characters */. The character // introduce a comment which terminates upon reaching the end of a line

Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

Keywords

The following identifiers are reserved for use as keywords, and may not be used

otherwise:

int	char	struct	return	while
if	else	print	scan	void

Primitive Data Types

```
int:       32 bit integers
char:      8 bit character
int*:      32 bit pointer to an integer value
char*:     32 bit pointer to a character value
void*:     32 bit pointer to a castable value
```

Constants

There are several kinds of constants, as follows:

Integer constants

An integer constant is a sequence of digits.

Strings

A string is a sequence of characters surrounded by double quotes ‘‘ " ’’. A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte(\0) at the end of each string so that programs which scan the string can find its end. In a string, the character ‘‘ " ’’ must be preceded by a ‘‘\’’ ; in addition, the same escapes as described for character constants may be used.

Data Type Combinations

There is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;
functions which return objects of a given type;
pointers to objects of a given type;
structures containing objects of various types.

In general these methods of constructing objects can be applied recursively.

Objects and lvalues

An object is a manipulatable region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name ‘‘lvalue’’ comes from the assignment expression ‘‘E1 = E2’’ in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

Characters and integers

A char object may be used anywhere an int may be. In all cases the char is converted to an int by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two’s complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

Pointers and integers

Integers and pointers may be added and compared; in such a case the int is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each

subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in an appendix.

Primary expressions

Primary expressions involving . , −>, subscripting, and function calls group left to right.

Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is ‘‘array of . . .’’, then the value of the identifier expression is a pointer to the first object in the array, and the type of the expression is ‘‘pointer to . . .’’. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared ‘‘function returning . . .’’, when used except in the function-name position of a call, is converted to ‘‘pointer to function returning . . .’’.

Constant

A decimal or character constant is a primary expression.

String

A string is a primary expression. Its type is originally ‘‘array of char’’; but following the same rule as identifiers, this is modified to ‘‘pointer to char’’ and the result is a pointer to the first character in the string.

(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type ‘‘pointer to . . .’’, the subscript expression is int, and the type of the result is ‘‘ . . . ’’. The expression ‘‘E1[E2]’’ is identical (by definition) to ‘‘* ((E1) + (E2)) ’’.

primary-expression (expression-listopt)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type ‘‘function returning . . .’’, and the result of the function call is of type ‘‘ . . . ’’. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C$\pi$is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. Recursive calls to any function are permissible.

primary-lvalue . member-of-structure

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

primary-expression −> member-of-structure

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that E1 be of pointer type, the expression ‘‘E1−>MOS’’ is exactly equivalent to ‘‘(*E1).MOS’’.

Unary operators

Expressions with unary operators group right-to-left.

** expression*

The unary * operator means indirection: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is ‘‘pointer to . . .’’, the type of the result is ‘‘ . . . ’’.

& lvalue-expression

The result of the unary & operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is ‘‘ . . . ’’, the type of the result is ‘‘pointer to . . .’’.

- expression

The result is the negative of the expression, and has the same type. The type of the expression must be char, int.

! expression

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable only to ints or chars.

Multiplicative operators

The multiplicative operators *, /, and % group left-to-right.

*expression * expression*

The binary * operator indicates multiplication. If both operands are int or char, the result is int; No other combinations are allowed.

expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int or char, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

Additive operators

The additive operators + and − group left-to-right.-

expression + expression

The result is the sum of the expressions. If both operands are int or char, the result is int. If an int or char is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if P is a pointer to an object, the expression ‘‘P+1’’ is a pointer to another object of the same type as the first and immediately following it in storage. No other type combinations are allowed.

expression − expression

The result is the difference of the operands. If both operands are int, char the same type considerations as for + apply. If an int or char is subtracted from a pointer, the former is converted in the same way as explained under + above. If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an int representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

Shift operators

The shift operators << and >> group left-to-right.

expression << expression

expression >> expression

Both operands must be int or char, and the result is int. The second operand should be non-negative. The value of ‘‘E1<<E2’’ is E1 (interpreted as a bit pattern 16 bits long)

left-shifted E2 bits; vacated bits are 0-filled. The value of ‘‘E1>>E2’’ is E1 (interpreted as a two’s complement, 16-bit quantity) arithmetically right-shifted E2 bit positions. Vacated bits are filled by a copy of the sign bit of E1.

Relational operators

The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to.

expression < *expression*

expression > *expression*

expression <= *expression*

expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

Equality operators

expression == *expression*

expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

expression & *expression*

The & operator groups left-to-right. Both operands must be int or char; the result is an int which is the bitwise logical and function of the operands.

expression | *expression*

The | operator groups left-to-right. The operands must be int or char; the result is an

int which is the bit-wise inclusive or of its operands.

expression && expression

The && operator returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0. The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, char, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right. When both operands are int or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

expression , expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls and lists of initializers.

Declarations

Declarations are used within function definitions to specify the interpretation which C πgives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form declaration: decl-specifiers declarator-listopt ; The declarators in the declarator-list contain the identifiers being declared. The

decl-specifiers consist of at most one type-specifier and at most one storage class specifier.

```
decl-specifiers:
        type-specifier
        sc-specifier
        type-specifier sc-specifier
        sc-specifier type-specifier
```

Type specifiers

The type-specifiers are

type-specifier:

```
        int
        char
        struct { type-decl-list }
        struct identifier { type-decl-list }
        struct identifier
```

if the type-specifier is missing from a declaration, it is generally taken to be int.

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

```
declarator-list:
        declarator
        declarator , declarator-list
```

The specifiers in the declaration indicate the type of the objects to which the declarators refer. Declarators have the syntax:

declarator:

```
identifier
        * declarator
        declarator ( )
        declarator [ constant-expressionopt ]
        ( declarator )
```

The grouping in this definition is the same as in expressions.

Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier; it is this identifier that is declared. If

an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

If a declarator has the form *D for D a declarator, then the contained identifier has the type ‘‘pointer to . . .’’, where ‘‘ . . . ’’ is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form D( ) then the contained identifier has the type ‘‘function returning ...’’, where ‘‘ . . . ’’ is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

```
D[constant-expression] or D[ ]
```

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. In the second the constant 1 is used. Such a declarator makes the contained identifier have type ‘‘array.’’ If the unadorned declarator D would specify a non array of type ‘‘. . .’’, then the declarator ‘‘D[i]’’ yields a 1-dimensional array with rank i of objects of type ‘‘. . .’’. I

Structure declarations

Recall that one of the forms for a structure specifier is

```
struct { type-decl-list }
```

The type-decl-list is a sequence of type declarations for the members of the structure:

```
type-decl-list:
```

type-declaration

```
type-declaration type-decl-list
```

A type declaration is just a declaration which does not mention a storage class (the storage class ‘‘member of structure’’ here being understood by context).

```
type-declaration:
     type-specifier declarator-list ;
```

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. Another form of structure specifier is struct identifier `{ type-decl-list }` This form is the same as the one just

discussed, except that the identifier is remembered as the structure tag of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

```
struct identifier
```

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself. A simple example of a structure declaration where its use is illustrated more fully, is

```
struct tnode {
      char tword[20];
      int count;
      struct tnode *left;
      struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort. The names of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

Statements

Except as indicated, statements are executed in sequence.

Expression statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compound-statement:
      { statement-list }
statement-list:
           statement
           statement statement-list
```

Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered elseless if.

While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is

returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

Null statement

The null statement has the form

;

A null statement is useful to carry a label just before the ‘‘}’’ of a compound statement or to supply a null body to a looping statement such as while.

Definitions

A Cπprogram consists of a sequence of Definitions. Definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. A definition declares an identifier to have a specified type. The type-specifier may be empty, in which case the type is taken to be int.

Function definitions

Function definitions have the form

```
function-definition:
     type-specifier-opt function-declarator function-body
```

A function declarator is similar to a declarator for a ‘‘function returning ...’’ except that it lists the formal parameters of the function being defined.

```
function-declarator:
     declarator ( parameter-listopt )
```

parameter-list:

```
identifier
identifier , parameter-list
```

The function-body has the form

```
function-body:
     type-decl-list function-statement
```

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be

declared only here. The function-statement is just a compound statement which may have declarations at the start.

```
function-statement:
     { declaration-listopt statement-list }
```

Since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared ‘‘array of ...’’ are adjusted to read ‘‘pointer to ...’’. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted). A free return statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

Initializations

Arrays cannot be initialized during their declaration. As a special case, a single string may be given as the initializer for an array of chars; in this case, the characters in the string are taken as the initializing values. Structures also cannot be initialized during their declaration. An example initialization for a structure and array are shown below.

```
struct tnode {
     char tword[20];
     int count;
};

tnode s;
s.count = 5;
s.tword = “Hello”;

int arr[3];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
```

Scope rules

A complete Cπprogram source text must be kept in a single file. This makes it possible for variables only to have a lexical scope.It is essentially the region of a program

during which the identifier may be used without drawing ‘‘undefined identifier’’ diagnostics. It is an error to redeclare identifiers already declared in the current context.

Types revisited

This section summarizes the operations which can be performed on objects of certain types.

Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the . or −> operators); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message.

Functions

The only thing that can be done with a function is - call it.

Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [ ] is interpreted in such a way that ‘‘E1[E2]’’ is identical to ‘‘*((E1) + (E2))’’. Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

Examples.

These examples are intended to illustrate some typical Cπ constructions as well as a serviceable style of writing Cπ programs.

Inner product

This function returns the inner product of its array arguments.

```
int inner ( v1, v2, n )
int v1 [ ] , v2 [ ] ;
{
      int sum ;
      int i ;
      sum = 0 ;
      i=0;
      while (i<n){
            sum = sum + (v1 [ i ] * v2 [ i ]);
            i= i + 1;
      }
      return ( sum );
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
int inner ( v1, v2, n )
int *v1, *v2 ;
{
      int sum ;
      sum = 0.0 ;
      while ( n ){
            *v1 = *v1 + 1;
            *v2 = *v2 + 1
            sum = sum + (*v1 * *v2);
            n = n - 1;
      }
      return ( sum ) ;
}
```

BinarySearch.cpi

```
int binary_search(int array[], int start, int end, int element) {

       if (start > end)
              return -1;
       else {
```

```
                int mid = ((start + end)/2);
                int temp = array[mid];
                if (temp == element) {
                        return mid;
                } else if (temp > element) {
                        return binary_search(array, start, mid - 1,
element);
                } else {
                        return binary_search(array, mid + 1, end,
element);
                }
        }
}

int bin_search(int array[], int size, int element) {
        return binary_search(array, 0, size - 1, element);
}

int main() {
       int arr[] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512};
       int size = 10;
       int target = scan();
       int result = bin_search()
       print("Binary search of ", target, "in arr returns",
bin_search(arr,size,target));
       return 0;
}
```

Unsupported features from ANSI C:

- Floating point variables and operations - no double and float.
- short and long integers.
- Unsigned integers
- break and continue
- Enums.
- sizeof and Ternary operators
- Increment and decrement operators.
- Const
- `for`, `do-while` and `switch` statements.
- Storage class specifiers - auto, register, volatile static and extern.
- Multi-file compilation and linkage.
- Preprocessing - no # directives.
- Function pointers and void pointers.
- Function inlining.
- Static and volatile function.
- Variable function arguments - Ellipsis (...)

- No type casting

References

Ritchie, Dennis M. "C reference manual." *Programming Languages*. Springer Berlin Heidelberg, 1983. 386-416.