Final Progress Report, Course Project, Group 7

Compiler Design (CS335), 2021-22-II

Vishesh Kaushik (170805)
Pulkit Gopalani (180564)
Abhishek Mugal (170028)
Sathvik Bhagavan (170638)

All contributed equally i.e. 25%

April 25, 2022

1 Introduction

We implement a **C++ compiler for the MIPS architecture, in Python 3.**

2 Source language (C++) specification

2.1 Native Data types

Native Data types that we propose to use in the language are as follows:

2.1.1 Void type

The **void** type is use to declare an empty set of values. It is generally used to declare pointers to arbitrarily typed variables or to declare the return type of functions that returns no values.

2.1.2 Character type

The **char** type stores the members in the character set used in American Standard Code for Information Interchange(ASCII).It has **%c** placeholder.

2.1.3 Integer types

1. The **int** type can represent integers with a pre-defined range $[-2^{31}, 2^{31} - 1]$ and it has a size of 4bytes. It has **%d** placeholder.
2. **long long int** is another integer type having range $[-2^{63}, 2^{63} - 1]$ and uses 8 bytes for storage. It has **%lld** placeholder.
3. **unsigned int** is also an integer type which can store only whole numbers having range $[0, 2^{32} - 1]$ and uses 4 bytes. It has **%u** placeholder.

2.1.4 Floating-point types

Floating-point types provides an approximation of decimal values over a wide range of magnitude. It uses IEEE-754 representation for this purpose. It has two types which are as follows:
1. float : It is the smallest floating point type and it has size of 4 bytes. It uses single precision format where value is stored as sign bit, 8-bit exponent, 23-bit significand. It has **%f** placeholder.
2. double: It is the bigger than float floating point type and it has size of 8 bytes. It uses double precision format where value is stored as sign bit, 11-bit exponent, 52-bit significand. It has **%lf** placeholder.

2.1.5 Boolean types

A variable with **bool** type can store anyone of the two values true and false. The storage required for this type is 1 byte.

2.1.6 nullptr_t

NULL is a keyword which is a null pointer constant of **nullptr_t** type and it is convertible to any raw pointer type.

2.2 Variables and Expressions

Variables give named storage that the program can manipulate. Each variable in this language has a specific type which helps in determining the size and the type of values it can store.

Rules for Variable Naming:

- The name of the variable can consist of letters,digits and the underscore character.
- It must begin with either a letter or an underscore.
- This language is case sensitive which means upper and lowercase letters are distinct.
- The name must not be a keyword or reserved word.

Expressions are sequences of operators and operands that are used for computing a value from the operands or designating functions.

2.2.1 Primary Expressions

Primary expressions are the fundamental building blocks of more complex expressions. They consist of names and literals. Literal is en element that directly represents a value.

```
int temp = 1;     // integer literal
float theta = 20.45;    // floating point literal
bool val = true;     // boolean literal
```

If not initialised, variables will be initialised to 0.
Postfix Expressions These expressions consists either expressions in which postfix operators follow a primary expression or the primary expression itself.

Operator Name	Notation	Operation	Syntax
Function call operator	( )	To pass function arguments during a function call	func_name (arg-expression-list)
Postfix increment operator	++	Increment in the operand value by one after solving postfix expression	postfix-expression ++
Postfix decrement operator	- -	Decrement in the operand value by one after solving postfix expression	postfix-expression - -
Explicit type conversion operator	type( )	allows explicit type conversion	type-name (expression-list)

Table 1: Postfix Operators

2.2.2 Expressions with Unary Operators

Unary operators act on only one operand in an expression.

Operator Name	Operation	Syntax
Address-of operator (&)	returns the address of its operand	address-of-expression : & cast-expression
Unary plus operator (+)	returns the value of its operand	+ cast-expression
Unary negation operator (-)	returns the negative value of its operand	- cast-expression
Logical negation operator (!) **(Unable to implement)**	returns the logical value of its operand	! cast-expression
Indirection operator (*)	dereferences a pointer and returns actual value	* cast-expression

Table 2: Expressions with Unary Operators

2.2.3 Expressions with Binary Operators

Binary operators act on two operands in an expression. The following are operators which we propose to use in our language. They have exactly the same meaning as defined in ANSI/ISO C++ International Standard (ISO/IEC FDIS 14882)

1. **Multiplicative operators** : $\{*, /, \%\}$
2. **Additive operators** : $\{+, -\}$
3. **Shift operators** : $\{>>, <<\}$
4. **Relational Operators** : $\{<, >, <=, >=, !=, ==\}$
5. **Bitwise operators** : $\{\&, \hat{}, |\}$
6. **Logical operators** : $\{\&\&, ||\}$
7. **Assignment operators** : $\{=, *=, /=, \%=, +=, -=, <<=, >>=, =, \hat{} =, |=\}$
8. **Comma operator** : $\{,\}$
9. **Class and Struct specific operator** : $\{., ->\}$

2.2.4 Conditional Operator

The conditional operator takes three operand and has the following syntax:

```
expression1 ? expression2 : expression3
```

2.3 Control structures

2.3.1 Selection (conditional) statements (`if-else`)

These statements are used for introducing conditional flow control (branching) in a C++ program. The basic structure of such a selection (`if-else`) statement is as follows (the `else` blocks are optional):

```
if(condition1){
    stmt1
}
else if(condition2){
    stmt2
}
else{
    stmt3
}
```

Here `condition{1,2,3}` are expressions which evaluate to either a `True` value (including a non-zero arithmetic value or a non-null pointer) or a `False` value (including arithmetic 0 and a null pointer). `stmt{1,2,3}` are C++ statement(s) which get executed based on the following scheme:

If `condition1` evaluates to `True`, then *only* `stmt1` is executed and `stmt{2,3}` are ignored. If `condition1` evaluates to `False` whereas `condition2` evaluates to `True`, then only `stmt2` is executed. If both `condition{1,2}` evaluate to `False`, then only `stmt3` is executed.

Note that any number of `(else if)` statements can be used after an `if` block, and using an `else` statement is optional. Nesting of `if-else` statements is also possible.

2.3.2 Iteration (loop) statements (`for, while`)

Iteration statements are used to execute a C++ statement (or a block of statements) in a repetitive manner, subject to an expression `condition` evaluating to `True`.

We consider the following iteration statements in our implementation:

(a) `for` **loop**

```
for(init_expr; condition; loop_expr){
    stmt
}
```

The init_expr is executed only once, just when the for loop begins. This is generally used to initialize loop variables.

After init_expr and before loop_expr or stmt, condition is evaluated. This is done for every iteration of the loop as long as it has not terminated. If condition evaluates to True (including non-zero arithmetic values or non-null pointers), control passes to stmt (Otherwise, the loop terminates). After stmt is executed, control returns to the loop, specifically to condition.

This sequence continues until the loop terminates, which can occur due to 2 (non-erroneous) reasons: condition evaluating to False (OR) encountering a break, goto or return statement in stmt.

Limitations

```
    for(int x=1;....      \\this doesn't work

    \\--------------------------

    int x;                \\but can be done this way
    for(x=1;..            \\so that the for loop works
```

(b) `while` **loop**

```
while(condition){
    stmt
}
```

The condition is evaluated for every iteration, before the loop (stmt) executes. If the evaluation turns out to be False (including arithmetic 0 and null pointer), the loop terminates. Otherwise, stmt is executed and control returns to condition.

This sequence continues until the loop terminates, which can occur due to 2 (non-erroneous) reasons: condition evaluating to False (OR) encountering a break, goto or return statement in stmt.

Note that any loop can have multiple loops nested inside it (i.e. in stmt).

2.3.3 Jump statements (break, continue, return, goto)

Jump statements are used to transfer program execution (control) to a specific location in the program, based on the type of statement used. These can be used to terminate a loop (break,) skip a loop iteration (continue), exit a function (return), or simply jump to a given location in the program (goto).

We consider the following jump statements in our implementation:

(a) `break`

```
loop1{
    stmt1
    break;
    stmt2
}
stmt
```

As soon as the break statement is encountered, the *nearest* enclosing loop (here loop1) is terminated, and the control returns to the statement just after the ended loop (here stmt).

Note that all statements in the loop before the break statement will be executed as usual (here stmt1), and the statements following it (here stmt2) will be ignored.

(b) `continue`

```
loop1{
    stmt1
    continue;
    stmt2
}
```

As soon as the `continue` statement is encountered, the control transfers to next iteration of the *nearest* enclosing loop (here `loop1`).

More specifically:

- `for` loop
  The control returns to execute `loop_expr` after which `condition` is evaluated. Based on this evaluation, loop may continue or terminate.
  Restriction
  The for loop wont increment the iterator by itself when continue is used. Therefore, explicit increment is required when continue is used in a for loop. Otherwise the program will fall into an infinite loop.
- `while` loop
  The control returns to evaluate `condition`. Based on this evaluation, loop may continue or terminate.

Note that all statements in the loop before the `continue` statement will be executed as usual (here `stmt1`), and the statements following it (here `stmt2`) will be ignored *only in the iteration in which* `continue` *is encountered.*

(c) `return`

`return` statement is used to terminate execution of a function and return some appropriate output value, if applicable. More details are provided in the section on functions.

(d) `goto`

```
stmt1
goto loc;
stmt2
```

`goto` is used to transfer program control to a specific location `loc` in the program. This is specified by placing an identifier `loc` at the required location. In the above snippet, `stmt1` will be executed, and depending on `loc`, `stmt2` may or may not be executed.

2.4 Input – Output (I/O) statements

We will implement `input, output`, as input and output functions.

2.4.1 Output

```
output("<placeholder1>,<placeholder2>,..",<var_name1>,<var_name2>,...);
```

Limitations

- Output can handle only one entity at a time. That is output function will only have one parameter. This parameter can be a string, identifier or a float/int number.

```
output("<placeholder1>);
```

2.4.2 Escape sequences

An escape sequence has a different meaning from itself when used inside a character array. The following escape sequences will be used in the language.

Escape Sequences	Meaning
\n	new line
\t	Tab (Horizontal)
\\	Backslash

Table 3: Escape Sequences

Character arrays can also be printed using **output** command. For Eg: output(”hello world\n”); will print **hello world** and cursor will move to the next line.

2.4.3 Input

```
input("<placeholder1>,<placeholder2>,..",&<var_name1>,&<var_name2>,...);
```

Limitations

- Input function is limited to single parameter which will be an identifier which can only store a string, identifier or a float/int number.

  ```
  input("<placeholder1>);
  ```

2.5 Arrays

2.5.1 Stack Declaration

Arrays can be declared as:

```
<datatype> <name> [<size>]
```

where size is a constant expression, its value should be known at compile time. Arrays can be initialized as

```
// 1. Elements are initialized separately
<datatype> <name> [<size>]{value1, value2, value3, ...}

// 2. Elements are initialized with the same value
<datatype> <name> [<size>]{value}
```

If it is not initialized, then it will contain random/garbage values.

2.5.2 Passing arrays to functions (Unable to implement)

When arrays are passed to a function, only the pointer to the first element is passed for both stack and heap based declarations. So in order to iterate in the array, a second parameter containing the size of the array needs to be passed.

```
<dtype> <func name> (<dtype> *<array name>, unsigned int size, ... ) {
    ...
}

//Another way
<dtype> <func name> (<dtype> <array name>[], unsigned int size, ... ) {
    ...
}
```

2.5.3 Accessing array elements

Array elements can be accessed using '[]' operator. For multi dimensional arrays, appropariate number of '[]' can be provided for accessing a particular element.

```
<datatype> <name> [i] = <expression>
<datatype> <name> [i][j][k]... = <expression>
```

Array elements can also be accessed using pointer arithmetic.

2.5.4 Limitations

- Arithmetic operations on array elements is not allowed directly after accessing the array element using indices. But one can store he value of the array element in a new variable, perform operations and then reassign the value to array element using it's index. The following are not allowed:

  ```
  ar[i]=ar[i]+1;  //1
  t=ar[i]+1;      //2
  t=ar[i]+ar[j]   //3
  ```

```
*(<datatype> <name> + i) = <expression>
*(*(*(<datatype> <name> + i) + j) + k)... = <expression>
```

2.6 Functions

2.6.1 Declaration

Functions can be declared as:

```
<datatype> <function name> (<parameters>);
```

2.6.2 Definition

Functions can be defined as:

```
<datatype> <function name> (<parameters>) {
    ...
    return ...
}
```

Function will end whenever it encounters **return** keyword. Control will return to the caller function afterwards. The data type of the returned variable/constant should be the return type of the function.

2.6.3 Default Arguments (Unable to implement)

Last parameter or parameters can be assigned default values while defining the function. While calling, the user can leave out these arguments or can pass them with some value.

An example is:

```
void  <function name> (<parameter_1>, <parameter_2> = 10) {
    ...
}

<function name> (<argument_1>);
<function name> (<argument_1>, <argument_2>);
```

2.6.4 Recursion

Functions can be used to call themselves recursively.

```
<datatype>  <function name> (<parameters>) {
    ...
    <datatype> <variable> =   <function name> (<parameters>);
    ...
}
```

2.6.5 Limitations

- Within a function definition all declarations should be at the beginning of the function, that is, no variable declaration is allowed after a statement other than a declaration.

  ```
  <datatype> <function name> (<parameters>) {
      <declarations>  \\ all declarations first strictly
      <all other statements> \\other statements strictly
                             \\after declarations
      return ...
  }
  ```

- Function returns can't be directly used in arithmetic operations

  ```
  t=f(x)+g(y);  \\this doesn't work

  \\-----------------------
  ```

```
    t1=f(x)         \\but can be done this way
    t2=g(y)
    t= t1+t2
```

2.7 User Defined Data Types

2.7.1 Struct

Struct declaration:

```
struct  <struct_name> {
    <data_member_declarations>
    <function_member_definitions>
};
```

Struct declaration with variable declaration:

```
struct  <struct_name> {
    <data_member_declarations>
    <function_member_definitions>
} <struct_var_name>;
```

A **struct** consists of members which can be data or functions. The order of declaration of members doesn't matter.

```
//two ways of declaring a new struct variable
struct  <struct_name> <struct_var_name>;  //1
<struct_name> <struct_var_name>;  //2

//assigning values to members
//using the dot operator to assign values to data member
<struct_var_name>.<data_member_name> = <value>;

//calling a void return type function
<struct_var_name>.<function_member_name>(<parameters>);

//calling a non-void return type function and assigning the return value
//to a variable
<var_datatype> <var_name> = <struct_var_name>.<function_member_name>(<
parameters>);


//USING STRUCT POINTERS AND ACCESSING MEMBERS

//declaring an object pointer
struct  <struct_name>* <struct_pointer_name> = &<struct_var_name>;

//assigning values to members
//using the arrow operator to assign values to data member
<struct_pointer_name> -> <data_member_name> = <value>;
```

Default access permissions for struct members is public in C++.

2.7.2 Class (Unable to implement)

Class declaration:

```
class  <class_name> {
    <data_member_declarations>
    <function_member_definitions>
};
```

Class declaration with object declaration:

```
class  <class_name> {
    <data_member_declarations>
    <function_member_definitions>
} <obj_name>;
```

A class too consists of members which can be data or functions. The order of declaration of members doesn't matter. Data member declarations and function member definitions are same as that of variables and functions in c++ respectively, as described in the previous sections.
Furthermore for each class member we can have access specifiers:

```
class  <class_name> {
    access_specifier:
        <member_name_1>;
    access_specifier:
        <member_name_2>;
    .
    .
    .
};
```

In our implementation we will only be dealing with two access specifiers that i.e. public and private.
Public: These members' values can change. Objects can both access and change the value of these members.
Private: These members' values can't be changed or accessed. Objects can only access the value of these members. However, the value of these members can't be changed.
Now we see how to use the class we defined through the instantiations of objects:

```
//declaring an object
<class_name> <obj_name>;

//assigning values to members
//using the dot operator to assign values to data member
<obj_name>.<data_member_name> = <value>;

//calling a void return type function
<obj_name>.<function_member_name>(<parameters>);

//calling a non-void return type function and assigning the return value
//to a variable
<var_datatype> <var_name> = <obj_name>.<function_member_name>(<
parameters>);


//USING POINTERS FOR OBJECTS AND ACCESSING MEMBERS

//declaring an object pointer
<class_name>* <obj_pointer_name> = &<class_obj_name>;

//assigning values to members
//using the dot operator to assign values to data member
<obj_pointer_name> -> <data_member_name> = <value>;
```

Default access specifier for class members is private in C++.

2.8 Pointers/References

To get the address of a variable in the memory space we use the ampersand operator.
A pointer is declared using * unary operator. Pointer for different types are declared by adding a star in front of the data type. * shows that the data type of the variable is pointer type. We have a NULL pointer pointing to address 0x00.

```
//pointer declaration
<data_type>* <pointer_var_name>;

//getting the address of a variable and assigning to a pointer
```

```
<data_type>* <pointer_var1_name> = &<var2_name>;

//null pointer
<data_type>* <pointer_var1_name> = NULL;
```

We also use the ampersand operator for references:

```
//variable declaration
<data_type> <var1_name>;

//reference variable declaration
<data_type>& <var2_name> = <var1_name>
```

This variable var2 can be used to access the element stored at the address of var1.

3 Advanced feature Specification (Tentative)

- **(Unable to implement)** Multidimensional arrays

  These are array of arrays and can be declared as:

  ```
  <datatype> <name> [<size_1>][<size_2>][<size_3>]...
  ```

  where size_1, size_2, size_3 is the number of elements in that dimension. All the sizes should be constant expressions and known at compile time. They can be initialized as

  ```
  // All elements are initialized
  <datatype> <name> [<size_1>][<size_2>] = {{...}, {...}, {...}}
  ```

  If it is not initialized, then it will contain random/garbage values.

- **(Unable to implement)** File I/O : open, close, read, write functions

- Simple library functions, such as
  - **(Unable to implement)** string functions like string reverse, string copy, string compare, string length etc.
  - math functions like exponential,modulus,square root, trigonometric etc.

4 Extra features added:

- **Negative indexing of arrays**
  We have implemented that use of negative indices in arrays, the same way it is done in python3.

- **#include**
  We have included the utility of including header files as in standard c++.

  ```
  #include <<header_file_name>.h>;
  ```

- **Math Library**
  We have built our own math library as math.h which includes utilities like floor, ceil, abs, factorial, power, sin, tan, cos, e^x, e^{-x}.

- **typedef**
  We have implemented standard c++ typdef in our compiler.

  ```
  typedef <data_type> <alias_name>;
  ```

- **assert**
  "assert" has been implemented the same way as in standard python3

  ```
  assert(<boolean_expression>);
  ```

References

[1] C++ Language Reference.
`https://docs.microsoft.com/en-us/cpp/cpp/cpp-language-reference?view=msvc-170`

[2] C++ Standard Library Reference.
`https://docs.microsoft.com/en-us/cpp/standard-library/cpp-standard-library-reference?view=msvc-170`