Creating Simple Programs Using CH

This chapter introduces the primary concepts and commands needed to write simple programs in the CH language. CH is a language that incorporates C with subsets of C++, FORTRAN, Matlab, Java, and C shell. The small portion of CH that we will use here comes from C and C++. CH is an interpreted language.

CH allows us to program in **command mode** or **program mode**. In **command mode** each line of code is interpreted and executed as it is entered. Using CH in command mode, the programmer can learn the language syntax quickly and see the results of each line of code. It can also be used as a debugging aid. In **program mode** the whole program is entered using the text editor and then interpreted and executed.

General Syntax Rules

A CH statement can be placed anywhere on the line. Align the statements so that they are readable to make the program easier to debug. One space, many spaces or no spaces are all allowed between elements in a statement. For example, to assign the value 8 to the variable myCount, all of the following statements are correct.

```
myCount=8;
myCount = 8;
myCount     =       8;
```

In **program mode** a semicolon is required at the end of every statement with the following exceptions: comment statements, **main**, and **include**. A semicolon never follows these three types of statements. More than one statement can be written on a line.

Comment statements

Comments are not part of the code to be executed. They are inserted to add information to make the code easier to understand. Comments are placed between /* and */. Anything in the program between /* and */ is ignored by the interpreter. Comments can be spread over more than one line. When debugging a program, existing code can be omitted by making it a comment so that it will not be executed.

```
/*      This is a comment       */
```

Variables

A variable is a name that represents a value. Variable names in CH ***must start with a letter*** and can contain only letters (uppercase and lowercase), numbers, and the underscore character. Choose names that are meaningful. Words that are used in the language, like **if**, **else**, **int**, **float**, etc. cannot be used as variable names. They are usually called reserved words but are called keywords in CH.

Declaring Variables

Variables must be declared in the program before they are used. This means that the programmer assigns the data type of each variable. We will use variables that are declared as integers, floating point numbers, or strings. Integers are the positive and negative whole numbers and zero. Floating point numbers include the decimal portion of the number. In the example below, the term **int** is used to declare the variables i, j, and total_count as integer variables. The variables total_tax and total_savings are declared as floating point with the keyword **float**. When we declare variables as integers (**int**) and floating point numbers (**float**), we are using data types that are built into the language.

```
int  i, j, total_count;

float total_tax, total_savings;
```

Although string (**string_t**) is not a built-in data type of CH, we will be using it in a similar manner. A string is a group of characters which can include letters, numbers, spaces, and punctuation. The declaration is the same as for integers and floating point as shown in the example below.

```
string_t  letterGrade, last_name, Nick_Name;
```

Assignment Statements

Assignment statements are used to assign a value to a variable. The following format is used for assignment statements:

```
variable-name = expression;
```

The variable to be assigned a value is always on the left side of the equal sign. The expression on the right side can be another variable, a value, or an expression that has to be evaluated. The equal sign does not mean equal as it does in algebra; instead it means to make equal. The variable on the left is assigned the value of the expression on the right. For example, consider five variables declared to be integer variables:

```
int   avgAge, john, mary, joe, tom;
```

The following assignment statements give values to the variables john, mary, joe, tom and avgAge.

```
john = 2004 - 1985;
mary = 2 * john + 3;
joe = 3;
tom = john;
avgAge = (john + mary + joe + tom) / 4;
```

The first line of code below declares the variables Letter_Grade, highGrd, message1, LastMessage, LastName, and nick_name as strings (**string_t**). It is followed by an assignment statement.

```
string_t  Letter_Grade, highGrd, message1, LastMessage, LastName, nick_name;
LastName = “Jones”;
```

In the assignment statement above, the string variable LastName was assigned the value Jones. ***Notice that Jones is enclosed in quotation marks.*** When you are assigning a literal value to a string variable, it must be enclosed in quotation marks; otherwise it would be interpreted as a variable name.

In the example below, the variable nick_name is assigned the value of the variable lastName, which was Jones. Notice that there are no quotation marks. If the word, lastName, had been inside of quotation marks, the new value of nick_name would have been lastName instead of Jones.

```
nick_name = LastName;   /* Assigns the value of the variable LastName to nick_name*/

message1 = “Please enter your last name”;  /* Assigns the value of the literal string
                                              given between the quotation marks  */
LastMessage= message1;
Letter_Grade = “a”;
highGrd=Letter_grade;
```

Arithmetic Expressions

Just as in algebra, arithmetic expressions are evaluated from left to right with multiplication (*) and division (/) being evaluated before addition (+) and subtraction (-). Use parentheses to overrule this order or to clarify the expression. For instance,

```
bill = (2 + joe) * 2;
```

will set the value of the variable bill equal to 10 when the variable joe is equal to 3.

Input Statements

Input statements are used to input values and assign the values to variables.

cin >> variable-name;

If you declared a variable called yourAge,

int yourAge;

the following statement:

cin >> yourAge;

will cause the computer to wait for you to enter a value. After you enter a value and press return, the variable yourAge is equal to the value that you entered. The variable can be used as shown in the example below.

avgAge = (john + mary + joe + tom + yourAge) / 5;

If you want to enter two or more values using one input statement, use >> before each variable name. For example the statement

cin >> yourAge >> joe;

would allow you to enter your age, press return and then enter a value for joe, and press return again.

If you want to input a value into a string variable, ***the actual text entered would not contain quotation marks as we have seen in the assignment statement***. For the statement below that reads in the nickname, the user would enter Bobby, not “Bobby”.

cin >> nick_name;

Note: In CH, when **cin** is used to read in a string and an **int** or **float** variable was the last value read in the execution, the statement

getchar();

is necessary to remove the rest of the input line that was entered when the number value was read.
If your program is to be used by others, you should output a line to tell the users what they are to enter.

Output Statements

To write to the screen, use

cout << variable-name;

to output the value of any type of variable.

Use the form

cout << “literal string”;

or

cout << expression;

as shown in the following examples.

cout<< “text to be written to the screen”;

cout <<“Happy Birthday! You are “ <<yourAge + 1<<“ years old!”;

cout << “the average age is “ << avgAge;

In the statement above, the information within quotation marks will be output as typed followed by the value of the variable avgAge. Using the variables john and mary from the earlier example:

cout << “John’s age is “ <<john << “.”<< endl<<“Mary’s age is “<<mary << “.”<<endl;

The output would be:

John’s age is 19.
Mary’s age is 41.

In this example, **endl** will output an end of line character, which will force the rest of the output to the next line. Since there is an **endl** at the end of the line, output from the next **cout** will start on a new line.

The following code will write:
Please enter your age
to the screen and then wait for a value to be entered.

```
cout << “Please enter your age”;
cin>> yourAge;
```

When **cin** or **cout** statements are used in **program mode**, the following **include** statement must be inserted after the comment statements at the beginning of the program to allow the program to accept input and output using the **cin** and **cout** instructions.

```
#include <iostream.h>
```

Conditional Statements

The if-then statement allows the program to choose which statements to execute depending on a condition. The **if** statement has the form

```
if  (condition) {
    statement list
}
```

When the **if** statement is executed, the condition is evaluated, and if it is true, the CH code in the statement list (the **then** portion) is executed. If the condition is false, the statements in the list are skipped. ***The brackets can be omitted if there is only one statement in the list.***

The condition for floating point numbers and integers is an expression that uses at least one of the relational operators below. The condition must be in parentheses.

Relational operators used for integers and floating point:

Equal	= =	(**Note**: Two equal signs are used for the relational operator, equal.)
Not equal	! =	

Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
And	&&
Or	||

Note: logical ANDs and ORs are used to create composite conditions, i.e. conditions with multiple parts. In the case of AND, all sub-conditions of the composite condition must be true in order for the composite condition to evaluate to true. In the case of OR, it suffices that one sub-condition is true for the composite condition to be true.

```
if (john = = yourAge){
        cout << "John is your age!"<<endl;
}

if (john <  yourAge && mary <yourAge)
        cout << "You are older than John and Mary!";

if (john > avgAge) {
     aboveAvgCnt = aboveAvgCnt +1;
    cout << "John is older than the average age";
}
```

You may have a need to choose two paths depending on the value of the condition.
For this, use the **if-then-else** statement, with the form:

```
if  (condition) {
   statement list 1
}
else  {
   statement list 2
}
```

Statement list 1 (the **then** portion) will be executed if the condition is true. The **else** portion, statement list 2, will be executed if the condition is false.

```
if (john > yourAge) {
        oldestAge =john;
}
else {
      oldestAge=yourAge;
}
cout << "The oldest one of you is only " << oldestAge << " years old!";
```

If the value of john is greater than the value of yourAge, the value of john is assigned to the variable oldestAge. If the value of john is not greater than the value of yourAge, the value of yourAge is assigned to oldestAge. The statement following the **if** statement (following the final closing bracket) is the next statement executed. The **cout** statement in the example will be executed after the **if** statement no matter whether the **then** or the **else** portion was executed.

If statements can be nested inside the **then** portion or inside the **else** portion of other **if** statements. It is helpful to use the brackets to help to match the **else** portion to the proper **if**, since all **if** statements do not have an **else** portion. Experienced programmers use tabs to indent parallel statement lists for readability (See the following examples). No matter how the statements are lined up on the screen, the **else** portion always goes with the previous **if** unless shown by brackets to belong to another **if**.

```
if (yourAge > john){
```

```
            olderJohn  = olderJohn + 1;
          if (yourAge > joe){
             olderJoe  =  olderJoe + 1;
}      /* This bracket closes the then portion of the second if   */
}      /* This bracket closes the then portion of the first if   */
else {
          notOLDJohn = notOLDJohn + 1;
}      /* This bracket closes the else portion of the first if   */
```

Example of nested if statements

```
if (yourAge>john && yourAge>mary){
          cout << "You are older than John and Mary!";
}
else {
      if (john > yourAge && john >mary) {
      cout << "John is older than you and Mary!";
}
      else {
            if (Mary > yourAge && mary > john) {
              cout << "Mary is older than you and John!";
}
             else    {
             cout << "Some of you are the same age!";
}
}
}
```

Conditional Statements for String Variables

If statements work exactly the same for string variables as for integers and floating point, but since **string_t** is not a built-in data type, we will have to use the function **string compare** (**!strcmp)** in the condition to compare string values. The condition is true if the two strings being compared by the function, **!strcmp**, are equal. If the condition is true, the **then** portion is executed. The relational operators that work for **int** and **float** variables will not work for **string_t** variables. The **if statement** using the **compare string** function uses the following format:

```
if (!strcmp(string-variable-name, " literal string in quotes"))  {
     statements
}

if (!strcmp(string-variable-name1,  string-variable-name2 ))  {
     statements
}
```

The function name, !**strcmp**, is followed by a string variable and then either a literal string inside quotation marks or another string variable. See the examples below.

Example comparing a string variable and a literal string:

```
if (!strcmp (Letter_grade, "B+")) {
   cout << "You earned a B+."
}
```

Example comparing two string variables:

```
if (!strcmp (nick_name, firstName)) {
   cout << "You don't have a nick name."
```

```
}

if (!strcmp (Letter_grade , highGrd)) {
    cout << "You have the highest grade in the class! "<< Letter_grade <<" !"
}
else {
    cout << "Your grade was a   "<<Letter_grade <<" ."

}
```

The relational operators **and** (&&) and **or** (||) can be used between two functions in the **if** statement as shown in the example below.

```
if (!strcmp (Letter_grade, "b+") || !strcmp (Letter_grade, "B+")) {
    cout << "You earned a B+."
}
```

Loops

Looping statements allow programs to continue executing a block of statements for a fixed number of times or a number of times determined by the data. For example, the following code will output the numbers from 0 to 20 with the cube of the number.

```
int i;
i = 0;
while (i < 21) {
cout <<  i <<  "  " <<  i*i*i << endl;
i = i+1;                        /*  i is incremented within the loop  */
}
```

Initally ***i*** is 0 (and therefore less than 21) when the **while** loop is entered. But the last statement within the **while** loop adds 1 to ***i*** (now, i=1). The loop then recycles to the top, checks the value of ***i*** and executes again. As you can see, as the loop executes, it will continue to add 1 to ***i*** and then go back to the top of the **while** and compare ***i*** to 21. When i =21, the loop ends. As the loop executes, the cube of each of the integers from 0 to 20 is calculated and output.

The form of a **while** statement is:

```
while (condition) {
        statement list
}
```

When the **while** statement is executed, the condition is evaluated first. If it is true, the statements in the list are executed. Then the condition is evaluated again, and the process is repeated. When the condition is evaluated and it is false, the process is terminated, and the statement following the closing bracket of the **while** statement is executed.

Infinite Loops

It is possible to write an infinite loop: one that never terminates because the condition never becomes false. This condition usually occurs by mistake, so be careful to change the value of the variable inside the loop so that the condition will eventually become false. In the example above, if the statement that incremented ***i*** (i = i + 1;) had been omitted, the condition (i<21) would have always been true, and the program would have been stuck in the loop. Also remember to initialize the variable in the condition before you enter the loop. In the example above, ***i*** was set to zero (i = 0;) before the **while** statement which started the loop was executed.

If your program gets stuck in an infinite loop and you are using program mode, hold down the Control key and press c. This should stop the program. If it does not or if you are using **command mode**, press CTRL + Alt + Delete (or Backspace), open the task manager, go to applications, and end the CH program.

Using Command Mode

To use **command mode**, double-click on the CH program icon on your desktop. You can then enter statements in CH directly at the prompt in the window that opens.

Command mode not only allows you to check the syntax but to see how the code is working by easily obtaining intermediate values when necessary. ***When you exit CH, everything that you did will be deleted.*** If you open CH again, none of the variables that you declared in your previous session will be there. Occasionally an error will cause CH to close.

In **command mode** a variable name can be entered and the value of the variable will be written to the screen. This can be used in debugging to determine exactly what the program is doing. The semicolons at the end of every statement are not always necessary when you are in command mode.

In **command mode** only one statement can be typed on a line. If a statement is long and requires more than one line, use a backslash (\) at the end of a line to continue the statement on the next line.

```
avgAge = (john + mary + joe + tom + yourAge) / 5;
```

Using the Arrow Keys

If you want to enter the same statement or one similar to one you have already typed, you can use the *up-arrow* key. If the line that you want was six statements before to the one you just typed, press the up arrow repeatedly, until the statement that you want appears on the current line. If you go too far up, press the *down arrow*. When you get the statement that you want on the current line, you can use the *left* and *right arrows* to move back and forth to make changes. This allows you to change the code without retyping the entire line. For command mode, these keys represent a great convenience to avoid lots of retyping.

Using Program Mode

To use **program mode**, the entire program is written in a text editor and saved with a ch extension (e.g. your-filename.ch). We will be using Notepad, just as we did to create the HTML files.

Entering the Program in the Text Editor

The first line of code must be a comment statement. The **include** statement needed for input/output is next. It is followed by **main( )** and an opening curly bracket. Your program follows and ends with the closing curly bracket.

```
/*    First program in CH    */

#include <iostream.h>

main(){

        cout <<“Hello world!” <<endl ;

    }
```

After the program is saved

- To execute the program, double-click the CH icon on the desktop. Type (dot forward-slash your-filename) at the prompt in the window that opens:

./your-filename

It is not necessary to include the file extension with the filename here.
For example if you saved the file as age.ch you would type **./age**

- After you have executed the program, you can move back to Notepad, make changes, save the file, move back to the CH window and use the *up-arrow* key. Press *Enter* to execute the file again.

Developing the Algorithm

Before you can write a program, you need to understand the problem and develop a plan for solving it. What is the program supposed to do? How is the problem to be solved? What is the input? What is the output? What processing is necessary to create the correct output? Are any calculations necessary? What is the best way to perform the calculations? Do the calculations solve the problem? Are there any special cases where the calculation will not solve problem? Using pseudocode, develop a step-by-step plan that can be converted into CH to solve the problem. Does this algorithm solve the problem?

Writing the Program

Code the program from the algorithm that you developed in pseudocode. Insert comments to explain the code and to document how you are solving the problem so that others who see the program will understand how the program works.

Testing and Debugging

If you get syntax errors, try to correct the errors at the top of the error list and rerun the program. You can always enter some of the statements in **command mode** to test them. If you cannot figure out what is wrong, turn some of the statements into comments so that they are not executed. This will help you figure out the location of the error. The problem is simplified by debugging part of the program at a time. This is helpful for both syntax and logic errors. You can also insert **cout** statements to determine the value of particular variables at different locations in the program. Another useful tool in debugging is to find out which lines of code are executed. You can do this by inserting different output statements between existing statements in the code. By looking at the output, you can determine the flow of the program. Test your program with different data to see that it works correctly. Select test data that covers different types of situations to ensure that the program works as expected for all cases. Check the output to see that the program actually gives the correct answers.