CC-112 Programming Fundamentals

Structured Program Development in C - I

Nazar Khan
Department of Computer Science
University of the Punjab

Algorithm

- A *procedure* for solving a *computational* problem in terms of
  1. the *actions* to be executed, and
  2. the *order* in which these actions are to be executed.
- Specifying the order in which statements are to be executed in a computer program is called *program control*.

Pseudocode

- Informal language that helps you develop algorithms.
- Helps the programmer to “think out” a program before attempting to write it in a programming language.

Pseudocode of an algorithm to find the minimum of a list of numbers

```
Initialise min as 1st element of the list
Go through every element of the list starting from the 2nd
  If the current element is smaller than min
    Overwrite min by the current element
Display value of min
```

Control Structures

All programs can be written in terms of 3 control structures.

Sequence

Selection

1. if
2. if/else
3. switch

Repetition

1. while
2. do/while
3. for

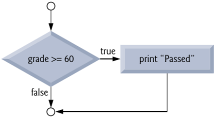
Control sturctures can be *stacked* or *nested*.

C program to find minimum of a list of numbers

```
// File name: find_min.c
// Program to find minimum number in a list of numbers.
// To compile and link: gcc find_min.c -o find_min
// To run: ./find_min
#include <stdio.h>

// function main begins program execution
int main( void )
{
  int number_list[] = {5, -6, 7, -17, 0, 23, 1000, -10, 12, 48}; // list of 10 integers
  int min; // variable to store the minimum number
  int i;   // variable to go through the list of numbers

  min = number_list[0];     // store 1st number in min
  i = 1;                    // start from the 2nd number
  while (i<10)              // go through every number
  {
    if (number_list[i] < min) // if current number is smaller than min
    {
      min = number_list[i];   // overwrite min by the current number
    }
    i = i + 1;                // set i to the position of the next number
  }
  printf( "The smallest number is %d\n", min ); // display the minimum
} // end function main
```

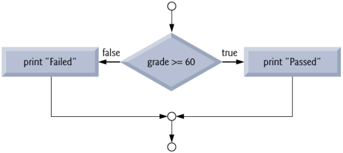
Selection Structures

- The *if* single-selection statement selects or ignores a single action.



Flowchart for *if* single-selection for displaying "Passed" if marks exceed or equal 60.

- The *if. . . else* double-selection statement selects between two different actions.



Flowchart for *if. . . else* binary-selection for displaying "Passed" or "Failed".

- The *switch* multiple-selection statement selects among many different actions based on the value of an expression.

An alternative to *if. . . else*

- C provides the *conditional operator* ?: which is closely related to the if. . . else statement.
- ?: is a *ternary operator*.
- Can either be used to return some expression

  ```
  printf( marks >= 60 ? "Passed" : "Failed" );
  ```

  or to execute some statement.

  ```
  marks >= 60 ? printf("Passed") : printf("Failed");
  ```

  Both are equivalent.

Nested if. . . else

Code with nesting

```
if ( marks >= 90 ) {
  printf( "A" );
} // end if
else {
  if ( marks >= 80 ) {
    printf( "B" );
  } // end if
  else {
    if ( marks >= 70 ) {
      printf( "C" );
    } // end if
    else {
      if ( marks >= 60 ) {
        printf( "D" );
      } // end if
      else {
        printf( "F" );
      } // end else
    } // end else
  } // end else
} // end else
```

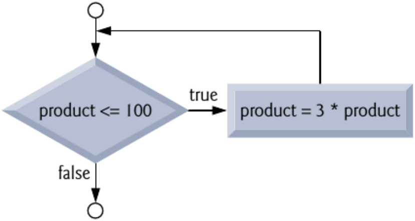
Code without nesting

```
if ( marks >= 90 )
{
  printf( "A" );
} // end if
else if ( marks >= 80 ) {
  printf( "B" );
} // end else if
else if ( marks >= 70 ) {
  printf( "C" );
} // end else if
else if ( marks >= 60 ) {
  printf( "D" );
} // end else if
else {
  printf( "F" );
} // end else
```

Avoid too much nesting *if possible*. More than 3 levels of nesting makes code less readable.

Repetition Structures

- The *while* iteration statement specifies that an action is to be repeated while a condition is true.
- Eventually, when the condition becomes false, the iteration terminates, and the first statement after the iteration statement executes.



While loop

Counter-controlled iterations

Ask for 10 students' marks and compute their average.

```
Set total to zero
Set marks counter to one

While marks counter is less than or equal to 10
  Input the next marks
  Add the marks into the total
  Add one to the marks counter

Set the class average to the total divided by ten
Print the class average
```

Ask for 10 students' marks and compute their average

Program

```c
// Filename: class_average_fixed.c
// Class average program with counter-controlled iteration.
// To compile and link: gcc class_average_fixed.c -o class_average_fixed
// To run: ./class_average_fixed
#include <stdio.h>

// function main begins program execution
int main( void )
{
  unsigned int counter; // number of grade to be entered next
  int grade; // grade value
  int total; // sum of grades entered by user
  int average; // average of grades

  // initialization phase
  total = 0; // initialize total
  counter = 1; // initialize loop counter

  // processing phase
  while ( counter <= 10 ) { // loop 10 times
    printf( "%s", "Enter grade: " ); // prompt for input
    scanf( "%d", &grade ); // read grade from user
    total = total + grade; // add grade to total
    counter = counter + 1; // increment counter
  } // end while

  // termination phase
  average = total / 10; // integer division
  printf( "Class average is %d\n", average ); // display result
} // end function main
```

While loop

Sentinel-controlled iterations

Ask for students' marks and compute their average.

Number of students is not known.

Initial attempt

```
1 Initialize variables
2 Input, add, and count quiz marks
3 Calculate and print class average
```

While loop

Sentinel-controlled iterations

Ask for students' marks and compute their average.

Number of students is not known.

Refinement

```
Initialize total to zero
Initialize counter to zero

Input 1st grade (possibly sentinel)
While user has not yet entered sentinel
  Add this grade into running total
  Add one to grade counter
  Input next grade (possibly sentinel)

If the counter is not equal to zero
  Set average to total divided by counter
  Print the average
else
  Print “No grades were entered”
```

While loop

Counter-controlled iterations

Ask for students' marks and compute their average.

Number of students is not known.

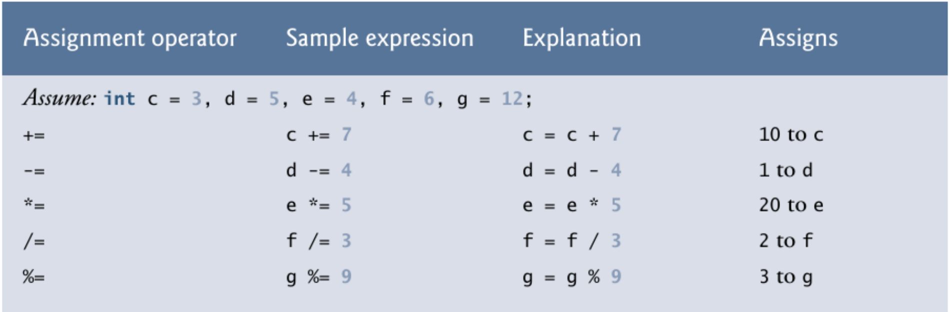
```
// Filename: class_average_dynamic.c
// Class average program with sentinal-controlled iteration.
// To compile and link: gcc class_average_dynamic.c -o class_average_dynamic
// To run: ./class_average_dynamic
#include <stdio.h>

// function main begins program execution
int main( void )
{
  unsigned int counter; // number of grades entered
  int grade; // grade value
  int total; // sum of grades
  float average; // number with decimal point for average

  // initialization phase
  total = 0; // initialize total
  counter = 0; // initialize loop counter

  // processing phase
  // get first grade from user
  printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
  scanf( "%d", &grade ); // read grade from user
```

While loop

Counter-controlled iterations

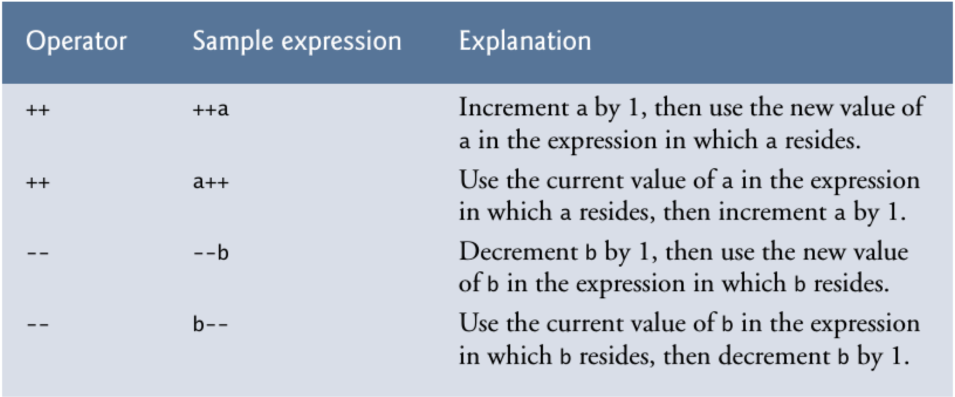
```
  // loop while sentinel value not yet read from user
  while ( grade != -1 ) {
    total = total + grade; // add grade to total
    counter = counter + 1; // increment counter

    // get next grade from user
    printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
    scanf("%d", &grade); // read next grade
  } // end while

  // termination phase
  // if user entered at least one grade
  if ( counter != 0 ) {
    // calculate average of all grades entered
    average = ( float ) total / counter; // avoid truncation

    // display average with two digits of precision
    printf( "Class average is %.2f\n", average );
  } // end if
  else { // if no grades were entered, output message
    printf( "No grades were entered\n" );
  } // end else
} // end function main
```

Arithmetic Assignment Operators



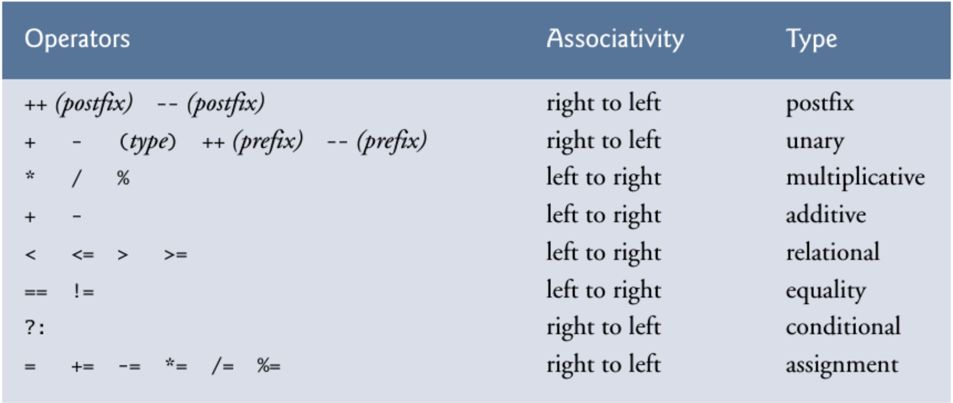
Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Increment/Decrement Operators



Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Precedences



Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - (*type*) ++ *(prefix)* -- *(prefix)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Nested Control Structures

You've been asked to provide a summary of results for 10 students. Next to each name a 1 is written if the student passed the exam or a 2 if the student failed. Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number who failed.
4. If more than eight students passed the exam, print the message "Bonus to instructor!"

Nested Control Structures

Initial pseudocode

1. Initialize variables
2. Input the ten quiz grades and count passes and failures
3. Print a summary of the exam results and decide whether instructor should receive a bonus

The most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C program is normally straightforward.

Many programmers write programs without ever using program-development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.

Nested Control Structures

Refinement

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
  Input the next exam result
  If the student passed
    Add one to passes
  else
    Add one to failures
  Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
  Print “Bonus to instructor!”
```

Nested Control Structures

```c
// Filename: results_summary.c
// Analysis of examination results.
// To compile and link: gcc results_summary.c -o results_summary
// To run: ./results_summary
#include <stdio.h>

// function main begins program execution
int main( void )
{
 // initialize variables in definitions
 unsigned int passes = 0; // number of passes
 unsigned int failures = 0; // number of failures
 unsigned int student = 1; // student counter
 int result; // one exam result

 // process 10 students using counter-controlled loop
 while ( student <= 10 ) {

  // prompt user for input and obtain value from user
  printf( "%s", "Enter result ( 1=pass,2=fail ): " );
  scanf( "%d", &result );
  // if result 1, increment passes
  if ( result == 1 ) {
    passes = passes + 1;
  } // end if
  else { // otherwise, increment failures
    failures = failures + 1;
  } // end else
```

Nested Control Structures

```
  student = student + 1; // increment student counter
} // end while

 // termination phase; display number of passes and failures
 printf( "Passed %u\n", passes );
 printf( "Failed %u\n", failures );
 // if more than eight students passed, print "Bonus to instructor!"
 if ( passes > 8 ) {
  printf( "Bonus to instructor!" );
 } // end if
} // end function main
```