Chp 8.

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program: • auto • register • static • extern **The auto Storage Class** The **auto** storage class is the default storage class for all local variables. `{ int mount; auto int month; }` The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables. **register** The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). `{ register int miles; }` The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be

stored in a register depending on hardware and implementation restrictions.

The static Storage Class The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared. In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class. `#include <stdio.h> /* function declaration */ void func(void); static int count = 5; /* global variable */ main() { while(count--) { func(); } return 0; } /* function definition */ void func( void ) { static int i = 5; /* local static variable */ i++; printf("i is %d and count is %d\n", i, count); }` When the above code is compiled and executed, it produces the following result: `i is 6 and count is 4 i is 7 and count is 3`

`i is 8 and count is 2 i is 9 and count is 1 i is 10 and count is 0` **The extern Storage Class** The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized, however, it points the variable name at a storage location that has been previously defined. When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below. **First File: main.c** `#include <stdio.h> int count; extern void write_extern(); main() { count = 5; write_extern(); }` **Second File: support.c** `#include <stdio.h> extern int count; void write_extern(void) {`

`printf("count is %d\n", count); }` Here, *extern* is being used to declare *count* in the second file, whereas it has its definition in the first file, main.c. Now, compile these two files as follows: `$gcc main.c support.c` It will produce the executable program **a.out**. When this program is executed, it produces the following result: `5`

chp 9.

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program: • auto • register • static • extern **The auto Storage Class** The **auto** storage class is the default storage class for all local variables. `{ int mount; auto int month; }` The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables. **register** The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). `{ register int miles; }` The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be

stored in a register depending on hardware and implementation restrictions.

The static Storage Class The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared. In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class. `#include <stdio.h> /* function declaration */ void func(void); static int count = 5; /* global variable */ main() { while(count--) { func(); } return 0; } /* function definition */ void func( void ) { static int i = 5; /* local static variable */ i++; printf("i is %d and count is %d\n", i, count); }` When the above code is compiled and executed, it produces the following result: `i is 6 and count is 4 i is 7 and count is 3`

`i is 8 and count is 2 i is 9 and count is 1 i is 10 and count is 0` **The extern Storage Class** The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized, however, it points the variable name at a storage location that has been previously defined. When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below. **First File: main.c** `#include <stdio.h> int count; extern void write_extern(); main() { count = 5; write_extern(); }` **Second File: support.c** `#include <stdio.h> extern int count; void write_extern(void) {`

`printf("count is %d\n", count); }` Here, *extern* is being used to declare *count* in the second file, whereas it has its definition in the first file, main.c. Now, compile these two files as follows: `$gcc main.c support.c` It will produce the executable program **a.out**. When this program is executed, it produces the following result: `5`

chp 10.

Decision-making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. Shown below is the general form of a typical decision-making structure found in most of the programming languages: C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value. C programming language provides the following types of decision-making statements. **Statement Description** if statement An **if statement** consists of a boolean expression followed by one or more statements. if...else statement An **if statement**

can be followed by an optional **else statement**, which executes when

C Programming
46
the Boolean expression is false. nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). switch statement A **switch** statement allows a variable to be tested for equality against a list of values. nested switch statements You can use one **switch** statement inside another **switch** statement(s). **if Statement** An **if** statement consists of a Boolean expression followed by one or more statements. **Syntax** The syntax of an 'if' statement in C programming language is: `if(boolean_expression) { /* statement(s) will execute if the boolean expression is true */ }` If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed. C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value. **Flow Diagram**
C Programming
47
Example `#include <stdio.h> int main () { /* local variable definition */ int a = 10; /* check the boolean condition using if statement */ if( a < 20 ) { /* if condition is true then print the following */ printf("a is less than 20\n" ); } printf("value of a is : %d\n", a); return 0; }` When the above code is compiled and executed, it produces the following result: `a is less than 20;`
C Programming
48

value of a is : 10 **if…else Statement** An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. **Syntax** The syntax of an **if...else** statement in C programming language is: if(boolean_expression) { /* statement(s) will execute if the boolean expression is true */ } else { /* statement(s) will execute if the boolean expression is false */ } If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed. C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value. **Flow Diagram**

Example #include <stdio.h> int main () { /* local variable definition */ int a = 100; /* check the boolean condition */ if(a < 20) { /* if condition is true then print the following */ printf("a is less than 20\n"); } else { /* if condition is false then print the following */ printf("a is not less than 20\n"); } printf("value of a is : %d\n", a); return 0; } When the above code is compiled and executed, it produces the following result: a is not less than 20; value of a is : 100 **if...else Statement** An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement. When using if…else if…else statements, there are few points to keep in mind: • An if can have zero or one else's and it must come after any else if's. • An if can have zero to many else if's and they must come before the else.

• Once an else if succeeds, none of the remaining else if's or else's will be tested. **Syntax** The syntax of an **if...else if...else** statement in C programming language is: if(boolean_expression 1) { /* Executes when the boolean expression 1 is true */ } else if(boolean_expression 2) { /* Executes when the boolean expression 2 is true */ } else if(boolean_expression 3) { /* Executes when the boolean expression 3 is true */ } else { /* executes when the none of the above condition is true */ } **Example** #include <stdio.h> int main () { /* local variable definition */ int a = 100; /* check the boolean condition */ if(a == 10) { /* if condition is true then print the following */

printf("Value of a is 10\n"); } else if(a == 20) { /* if else if condition is true */ printf("Value of a is 20\n"); } else if(a == 30) { /* if else if condition is true */ printf("Value of a is 30\n"); } else { /* if none of the conditions is true */ printf("None of the values is matching\n"); } printf("Exact value of a is: %d\n", a); return 0; } When the above code is compiled and executed, it produces the following result: None of the values is matching Exact value of a is: 100 **Nested if Statements** It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s). **Syntax** The syntax for a **nested if** statement is as follows: if(boolean_expression 1) {

`/* Executes when the boolean expression 1 is true */ if(boolean_expression 2) { /* Executes when the boolean expression 2 is true */ } }` You can nest **else if...else** in the similar way as you have nested *if* statements. **Example** `#include <stdio.h> int main () { /* local variable definition */ int a = 100; int b = 200; /* check the boolean condition */ if(a == 100) { /* if condition is true then check the following */ if(b == 200) { /* if condition is true then print the following */ printf("Value of a is 100 and b is 200\n"); } } printf("Exact value of a is : %d\n", a); printf("Exact value of b is : %d\n", b); return 0; }` When the above code is compiled and executed, it produces the following result:

`Value of a is 100 and b is 200 Exact value of a is : 100 Exact value of b is : 200` **switch Statement** A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**. **Syntax** The syntax for a **switch** statement in C programming language is as follows: `switch(expression){ case constant-expression : statement(s); break; /* optional */ case constant-expression : statement(s); break; /* optional */ /* you can have any number of case statements */ default : /* Optional */ statement(s); }` The following rules apply to a **switch** statement: • The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type. • You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon. • The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal. • When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached. • When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

• Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached. • A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case. **Flow Diagram Example** `#include <stdio.h> int main () { /* local variable definition */ char grade = 'B'; switch(grade) { case 'A' :`

`printf("Excellent!\n" ); break; case 'B' : case 'C' : printf("Well done\n" ); break; case 'D' : printf("You passed\n" ); break; case 'F' : printf("Better try again\n" ); break; default : printf("Invalid grade\n" ); } printf("Your grade is %c\n", grade ); return 0; }` When the above code is compiled and executed, it produces the following result: `Well done Your grade is B` **Nested switch Statements** It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise. **Syntax** The syntax for a **nested switch** statement is as follows: `switch(ch1) { case 'A': printf("This A is part of outer switch" );`

`switch(ch2) { case 'A': printf("This A is part of inner switch" ); break; case 'B': /* case code */ } break; case 'B': /* case code */ }` **Example** `#include <stdio.h> int main () { /* local variable definition */ int a = 100; int b = 200; switch(a) { case 100: printf("This is part of outer switch\n", a); switch(b) { case 200: printf("This is part of inner switch\n", a); } } printf("Exact value of a is : %d\n", a); printf("Exact value of b is : %d\n", b); return 0; }` When the above code is compiled and executed, it produces the following result:

`This is part of outer switch This is part of inner switch Exact value of a is : 100 Exact value of b is : 200` **The ? : Operator:** We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form: `Exp1 ? Exp2 : Exp3;` Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. The value of a ? expression is determined like this: 1. Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. 2. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Chp 11.

You may encounter situations when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages: C programming language provides the following types of loops to handle looping requirements. **Loop Type**

Description while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. for loop Executes a sequence of statements multiple times and abbreviates the code that

manages the loop variable.

do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body. nested loops You can use one or more loops inside any other while, for, or do..while loop. **while Loop** A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true. **Syntax** The syntax of a **while** loop in C programming language is: `while(condition) { statement(s); }` Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, the program control passes to the line immediately following the loop. **Flow Diagram**

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed. **Example** `#include <stdio.h> int main () { /* local variable definition */ int a = 10; /* while loop execution */ while( a < 20 ) { printf("value of a: %d\n", a); a++; }`

`return 0; }` When the above code is compiled and executed, it produces the following result: `value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15 value of a: 16 value of a: 17 value of a: 18 value of a: 19` **for Loop** A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. **Syntax** The syntax of a **for** loop in C programming language is: `for ( init; condition; increment ) { statement(s); }` Here is the flow of control in a ‘for’ loop: 1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears. 2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the ‘for’ loop. 3. After the body of the ‘for’ loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the ‘for’ loop terminates. **Flow**

Diagram Example #include <stdio.h> int main () { /* for loop execution */ for(int a = 10; a < 20; a = a + 1) { printf("value of a: %d\n", a);

} return 0; } When the above code is compiled and executed, it produces the following result: value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15 value of a: 16 value of a: 17 value of a: 18 value of a: 19 **do…while Loop** Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop. A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time. **Syntax** The syntax of a **do...while** loop in C programming language is: do { statement(s); }while(condition); Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false. **Flow Diagram Example** #include <stdio.h> int main () { /* local variable definition */ int a = 10; /* do loop execution */ do { printf("value of a: %d\n", a); a = a + 1; }while(a < 20); return 0;

} When the above code is compiled and executed, it produces the following result: value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15 value of a: 16 value of a: 17 value of a: 18 value of a: 19 **Nested Loops** C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept. **Syntax** The syntax for a **nested for loop** statement in C is as follows: for (init; condition; increment) { for (init; condition; increment) { statement(s); } statement(s); } The syntax for a **nested while loop** statement in C programming language is as follows: while(condition) { while(condition) {

statement(s); } statement(s); } The syntax for a **nested do...while loop** statement in C programming language is as follows: do { statement(s); do { statement(s); }while(condition); }while(condition); A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa. **Example** The following program uses a nested for loop to find the prime numbers from 2 to 100: #include <stdio.h> int main () { /* local variable definition */ int i, j; for(i=2; i<100; i++) { for(j=2; j <= (i/j); j++) if(!(i%j)) break; // if factor found, not prime if(j > (i/j)) printf("%d is prime\n", i); }

return 0; } When the above code is compiled and executed, it produces the following result: 2 is prime 3 is prime 5 is prime 7 is prime 11 is prime 13 is

```
prime 17 is prime 19 is prime 23 is prime 29 is prime 31 is prime 37 is prime
41 is prime 43 is prime 47 is prime 53 is prime 59 is prime 61 is prime 67 is
prime 71 is prime 73 is prime 79 is prime 83 is prime 89 is prime 97 is prime
```

Loop Control Statements Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. **Control Statement Description** break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. goto statement Transfers control to the labeled statement. **break Statement** The **break** statement in C programming has the following two usages: • When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. • It can be used to terminate a case in the **switch** statement (covered in the next chapter). If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block. **Syntax** The syntax for a **break** statement in C is as follows: `break;` **Flow Diagram**

Example `#include <stdio.h> int main () { /* local variable definition */ int a = 10; /* while loop execution */ while( a < 20 ) { printf("value of a: %d\n", a); a++; if( a > 15) { /* terminate the loop using break statement */ break; } }`

`return 0; }` When the above code is compiled and executed, it produces the following result: `value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15` **continue Statement** The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests. **Syntax** The syntax for a **continue** statement in C is as follows: `continue;` **Flow Diagram**

Example `#include <stdio.h> int main () { /* local variable definition */ int a = 10; /* do loop execution */ do { if( a == 15) { /* skip the iteration */ a = a + 1; continue; } printf("value of a: %d\n", a); a++;`

`}while( a < 20 ); return 0; }` When the above code is compiled and executed, it produces the following result: `value of a: 10 value of a: 11 value of a: 12` `value of a: 13 value of a: 14 value of a: 16 value of a: 17 value of a: 18 value of a: 19` **goto Statement** A **goto** statement in C programming provides an unconditional jump from the ‘goto’ to a labeled statement in the same function. **NOTE:** Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them. **Syntax** The syntax for a **goto** statement in C is as follows: `goto label; .. . label: statement;` Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram Example `#include <stdio.h> int main () { /* local variable definition */ int a = 10; /* do loop execution */ LOOP:do { if( a == 15) { /* skip the iteration */ a = a + 1; goto LOOP; } printf("value of a: %d\n", a); a++;`

`}while( a < 20 ); return 0; }` When the above code is compiled and executed, it produces the following result: `value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 16 value of a: 17 value of a: 18 value of a: 19` **The Infinite Loop** A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty. `#include <stdio.h> int main () { for( ; ; ) { printf("This loop will run forever.\n"); } return 0; }`

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop. **NOTE:** You can terminate an infinite loop by pressing Ctrl + C keys.