University of Nottingham
UK | CHINA | MALAYSIA

EEEE1042 - Lecture 4
Flow of control, if, while, for loops
Autumn Semester 2022.

Dr. Chan, Kheong Sann

kheongsann.chan@nottingham.edu.my

University of Nottingham Malaysia
Department of Electrical and Electronic Engineering

EEEE1042 C++ Programming: Scheduled classes

EEEE1042: for EE students, EEEE1032: for Mecha students.

Week	Dates	Lecture	EEEE1042 Practical	EEEE1032 Practical	Assessment
4	Sep 26 – 30	Thu2-4pm			
5	Oct 3 – 09	Thu2-4pm	Mon3-6pm	Wed3-6pm	
6	Oct 10 – 14	Thu2-4pm	P.H.	Wed3-6pm	
7	Oct 17 – 21	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT1 5%
8	Oct 24 – 28	Thu2-4pm	P.H.	Wed3-6pm	
9	Oct 31 –Nov 04	Project Week 1			
10	Nov 07 – 11	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT2 5%
11	Nov 14 – 18	Thu2-4pm	Mon3-6pm	Wed3-6pm	CW1 10%
12	Nov 21 – 25	Project Week 2			
13	Nov 28 –Dec 04	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT3 5%
14	Dec 05 – 09	Project Week 3			
15	Dec 12 – 16	Thu2-4pm	P.H.	Wed3-6pm	PT4 5%
16	Dec 19 – 23	Study Week			CW2 30%
17-18	Dec 26 – Jan 06	Study Weeks			
19-20	Jan 09 – 21	Final Exam (40%)			

Control Flow structure of a C program

- The control flow of a C-program starts at the beginning and flows down towards the end. Along the way it will meet certain control structures that will cause the flow to
  - Execute some code **depending on a condition** or **variable**.
    - `if else`
    - `switch case`
  - **Repetitively loop** over a section of code
    - `for`
    - `while`/`do while`
  - Execution can also **jump** to a predefined place in the code:
    - `goto`
  - Jump into a **function via function calls** and return.
- These structures use curly braces {} to delimit the code falling within the `if`,`else`, `for`, `while` etc... region. And by convention the code is indented one level.
  - Exception: If the contents of the braces {} comprise only a single C statement, the braces can be omitted.

Outline EEEE1042 C Lecture 4:

1. Making decisions with `if`
2. Selecting with `switch case`
3. Looping with `while`/`do-while`
4. Looping with `for`
5. Nested Loops
6. Function calls
7. Jumping with goto

Decisions with the if statement

- Conditional execution of code is handled by the `if else` structure in C which takes the form:

  ```
  if(condition){
      command1;
  } else {
      command2;
  }
  execution rejoins here after if-else-block
  ```

- *condition* must be a C expression that evaluates to a boolean output, either TRUE or FALSE.
- In C, 0 is the integer value assigned to FALSE. All other values are assigned TRUE. Examples:

```
if (1) {
// Always executes
}
```

```
if (0) {
// Never executes
}
```

Decisions with the if statement

- Conditional execution of code is handled by the `if else` structure in C which takes the form:

```
if(condition){
    command1;
} else {
    command2;
}
execution rejoins here after if-else-block
```

If *condition* is TRUE then control is transferred into the if block and the statements in *command1* are executed. Multiple statements can exist within the block.

- *condition* must be a C expression that evaluates to a boolean output, either TRUE or FALSE.
- In C, 0 is the integer value assigned to FALSE. All other values are assigned TRUE. Examples:

```
if (1) {
// Always executes
}
```

```
if (0) {
// Never executes
}
```

Decisions with the if statement

- Conditional execution of code is handled by the `if else` structure in C which takes the form:

```
if(condition){
   command1;
} else {
   command2;
}
execution rejoins here after if-else-block
```

If *condition* is FALSE then control is transferred to the `else` block and the statements in *command2* are executed. Multiple statements can exist within the block. The `else` block is optional.

- *condition* must be a C expression that evaluates to a boolean output, either TRUE or FALSE.
- In C, 0 is the integer value assigned to FALSE. All other values are assigned TRUE. Examples:

```
if (1) {
// Always executes
}
```

```
if (0) {
// Never executes
}
```

Decisions with the if statement

- Conditional execution of code is handled by the `if else` structure in C which takes the form:

```
if(condition){
   command1;
} else {
   command2;
}
execution rejoins here after if-else-block
```

In either case, execution resumes at the next instruction after the if block.

- *condition* must be a C expression that evaluates to a boolean output, either TRUE or FALSE.
- In C, 0 is the integer value assigned to FALSE. All other values are assigned TRUE. Examples:

```
if (1) {
// Always executes
}
```

```
if (0) {
// Never executes
}
```

`if` examples

- Example

```
if ((x>0) && (y<0) && (z==0)) {
   x++;
}
```

This can be written as:

```
if ((x>0) && (y<0) && (z==0)) x++;
```

if `x++` is the only action needing to be taken. Following the rule that braces are not needed if only a single command needs to be executed.

- `&&` and `||` have higher precendence than `>`, `<` and `==` so it is not necessary to bracket the conditions. Nevertheless it can be a good practice and can possibly save you from getting an unexpected result.
- Example

```
if (x>0) {
   return(x);
} else {
   return(-x);
}
```

```
if (x>0) return(x);
else return(-x);
// Control after if-else continues
here.
```

if else examples

If else statements can be consecutively strung one after the other:

```
#include <stdio.h>
int main () {
   int mark=88;
   if (mark==100) printf("%d: Perfect mark!\n",mark);
   else if (mark>90) printf("%d: Excellent mark!\n",mark);
   else if (mark>80) printf("%d: Very good mark!\n",mark);
   else if (mark>70) printf("%d: Good mark\n",mark);
   else if (mark>60) printf("%d: Average mark\n",mark);
   else if (mark>50) printf("%d: Below Average mark\n",mark);
}
```

Output:

```
88: Very good mark!
```

Exercise: Modify the above code such that it handles the extenuating cases `mark>100` or `mark<0` by printing `"Impossible!"` and any other mark by printing `"Sorry"` and test your code.

If else examples

```
#include <stdio.h>
int main () {
    int choice;
    printf("Please choose your desert:\n");
    printf("1. Ice cream\n");
    printf("2. Apple pie\n");
    printf("3. Chocolate Cake\n");
    printf("4. Tira misu\n");
    printf("5. Candy\n");
    scanf("%d",&choice);
    if (choice==1) printf("You chose Ice cream\n");
    else if (choice==2) printf("You chose Apple pie\n");
    else if (choice==3) printf("You chose Chocolate cake\n");
    else if (choice==4) printf("You chose Tira Misu\n");
    else if (choice==5) printf("You chose Candy\n");
    else printf("Invalid choice.\n");
}
```

Output:

```
$ ./helloWorld
Please choose your desert:
1. Ice cream
2. Apple pie
3. Chocolate Cake
4. Tira misu
5. Candy
3
You chose Chocolate Cake
```

If example for fopen

We have encountered the if structure when opening files that crams a lot of code into the if statement.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char filename[]="a.txt"; // name of the file to open/close
   /* Open the file for reading */
   if ((f=fopen(filename,"r"))!=NULL) {
      // Process the file in the if statement
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",filename);}
   /* Open the file for writing */
   if ((f=fopen(filename,"w"))!=NULL) {
      // Process the file in the if statement
      fclose(f); // close the file pointer
   } else {printf("Unable to open %s for reading\n",filename);}
}
```

If example for fopen

We have encountered the if structure when opening files that crams a lot of code into the if statement.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char filename[]="a.txt"; // name of th
   /* Open the file for reading */
   if ((f=fopen(filename,"r"))!=NULL) {
      // Process the file in the if state
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for 
   /* Open the file for writing */
   if ((f=fopen(filename,"w"))!=NULL) {
      // Process the file in the if statement
      fclose(f); // close the file pointer
   } else {printf("Unable to open %s for reading\n",filename);}
}
```

The statement `f=fopen(filename,"r")` tries to open the file for reading and returns a file pointer, which is then assigned to f. If fopen fails to open the file, it returns NULL which is also assigned to f

If example for fopen

We have encountered the if structure when opening files that crams a lot of code into the if statement.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char filename[]="a.txt"; // name of t
   /* Open the file for reading */
   if ((f=fopen(filename,"r"))!=NULL) {
      // Process the file in the if state
      fclose(f); // close the file pinte
   } else {printf("Unable to open %s for
   /* Open the file for writing */
   if ((f=fopen(filename,"w"))!=NULL) {
      // Process the file in the if statement
      fclose(f); // close the file pointer
   } else {printf("Unable to open %s for reading\n",filename);}
}
```

The assignment operator = returns the assigned value (which is either the pointer to the opened file on success, or NULL on failure). This is compared != with NULL. ie: the contents of the if block are executed if the file is successfully opened (f!=NULL).

If example for fopen

We have encountered the if structure when opening files that crams a lot of code into the if statement.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char filename[]="a.txt"; // name of the file to open/close
   /* Open the file for reading */
   if ((f=fopen(filename,"r"))!=NULL) {
      // Process the file in the if statement
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",filename);}
   /* Open the file for writing */
   if ((f=fopen(filename,"w"))!=NUL
      // Process the file in the if s
      fclose(f); // close the file p
   } else {printf("Unable to open %s
}
```

On the other hand, if f==NULL then the else statement is entered and the error message is printed out. In either case, the execution resumes after the if-else block

Outline EEEE1042 C Lecture 4:

1. Making decisions with `if`
2. Selecting with `switch case`
3. Looping with `while`/`do-while`
4. Looping with `for`
5. Nested Loops
6. Function calls
7. Jumping with goto

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   break;
}
```

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   break;
}
```

The switch statement takes a variable and examines its value. The variable can be an int, char, float or any C type.

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
  case val1 :
      command1 ;
  break;
  case val2 :
      command2 ;
  break;
      ⋮
  default:
      command3 ;
  break;
}
```

If the value of *variable* is *val1*, then the code located at *command1* is executed. Multiple commands can reside there.

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      :
   default:
      command3;
   break;
}
```

When the break; command is reached the execution exits the switch-case block. ie: execution jumps to the statement after the outer brace.

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   break;
}
```

If no break; command is encountered, execution will pass through the next case command and execute all remaining code until it falls out the end of the block.

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
  case val1:
    command1;
  break;
  case val2:
    command2;
  break;
    ⋮
  default:
    command3;
  break;
}
```

If the value of *variable* is *val2*, then control of the program jumps to the code located at *command2*. Multiple commands can reside there.

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   break;
}
```

Once again, when the break command is reached the control of the program exits the block

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   break;
}
```

If *variable* doesn't take on any of the case values, it reaches the default: case and executes the code located at *command3*

Switch-case

Switch-case statements allows the selection of a particular branch of code depending on the value of a variable. The syntax is:

```
switch(variable){
   case val1:
      command1;
   break;
   case val2:
      command2;
   break;
      ⋮
   default:
      command3;
   ~~break;~~
}
```

The final `break;` is not actually needed since the execution is going to drop out of the `switch-case` block regardless.

Example of switch-case

```c
#include <stdio.h>
int main() {
    char c = ’G’;
    switch (c){
    case ’R’:
        printf("Red\n");
        break;
    case ’G’:
        printf("Green\n");
        break;
    case ’B’:
        printf("Blue\n");
        break;
    default:
        printf("None\n");
        break;
    }
}
```

Output:

```
Green
```

Example of switch-case

```
#include <stdio.h>
int main() {
    char c = ’G’;
    switch (c){
    case ’R’:
        printf("Red\n");
        break;
    case ’G’:
        printf("Green\n");
        break;
    case ’B’:
        printf("Blue\n");
        break;
    default:
        printf("None\n");
        break;
    }
}
```

Question: What would happen if we didn’t have the `break;` statements?

Output:

```
Green
```

Example of switch-case 2

```
#include <stdio.h>
int main() {
    int i = 71;
    switch (i) {
    case 50 ... 60:
        printf("Pass\n");
        break;
    case 61 ... 70:
        printf("Good\n");
        break;
    case 71 ... 80:
        printf("Very good\n");
        break;
    default:
        printf("Fail\n");
        break;
    }
}
```

Output:

```
Very Good
```

Outline EEEE1042 C Lecture 4:

1. Making decisions with if
2. Selecting with switch case
3. Looping with while/do-while
4. Looping with for
5. Nested Loops
6. Function calls
7. Jumping with goto

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

```
while (condition) {
   command;
}
```

```
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

```
while (condition) {
   command;
}
```

The while construct starts out by testing *condition*. If *condition* is TRUE the body of `while` loop is entered. Otherwise control is transferred to the command after the `while` loop.

```c
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

```
while (condition) {
   command;
}
```

Then the commands inside the `while` block are executed.

```c
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

```
while (condition) {
   command;
}
```

When the end of the `while` block is reached, the *condition* is tested again.

```c
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

while (*condition*) {
  command ;
}

The `while` block is re-entered at the top if the condition is still TRUE. This continues until condition becomes FALSE at which point control goes to the next command after the while block.

```
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

While loops

The `while` command executes the contents of the block **while** the condition remains true, or **until** the condition becomes false.

```
while (condition) {
   command;
}
```

```c
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
   return 0;
}
```

```
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
```

While loops for reading a file string by string

We can use the while block structure to continually read a file until the EOF is reached.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char s[100]; // memory to hold the read line
   /* Open the file for reading */
   if ((f=fopen(argv[1],"r"))!=NULL) {
      while(fscanf(f,"%s",s)!=EOF) {
         printf("%s\n",s);
      }
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",argv[1]);}
}
```

While loops for reading a file string by string

We can use the while block structure to continually read a file until the EOF is reached.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char s[100]; // memory to hold the re
   /* Open the file for reading */
   if ((f=fopen(argv[1],"r"))!=NULL) {
      while(fscanf(f,"%s",s)!=EOF) {
         printf("%s\n",s);
      }
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",argv[1]);}
}
```

This line opens the file given in the first argument of the command line: argv[1] and enters the if if it's successfully opened.

While loops for reading a file string by string

We can use the while block structure to continually read a file until the EOF is reached.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char s[100]; // memory to hold the
   /* Open the file for reading */
   if ((f=fopen(argv[1],"r"))!=NULL)
      while(fscanf(f,"%s",s)!=EOF) {
         printf("%s\n",s);
      }
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",argv[1]);}
}
```

Now f is open for reading, this line uses fscanf to read a string from the stream into s. A string in the stream is delineated by white space. The control will go into the while loop as long as the fscanf succeeds, which means the next string was read into s. The contents of the while loop just prints that string to the screen.

While loops for reading a file string by string

We can use the while block structure to continually read a file until the EOF is reached.

```
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char s[100]; // memory to hold the
   /* Open the file for reading */
   if ((f=fopen(argv[1],"r"))!=NULL)
      while(fscanf(f,"%s",s)!=EOF) {
         printf("%s\n",s);
      }
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",argv[1]);}
}
```

Eventually, the end of file is reached, and when that happens, `fscanf` returns `EOF`. The `while` condition becomes false and the `while` block is NOT entered. Control jumps to statement after the while block which closes the file and then exits the main.

do-while loops

While and do-while loops are very similar:

while:

```
while(condition){
   command
}
```

1. Test condition
2. Execute command if true.

do-while:

```
do{
   command
}while(condition);
```

1. Enter loop, run command
2. Test condition, repeat if true.

The difference is that the `while` command **evaluates** *condition* **first** then enters the loop if condition is true.

The `do-while` loop **enters the body** and then **evaluates** *condition* at the end and repeats the loop if the condition is true.

This means a `do-while` loop is guaranteed to be executed at least once.
A `while` loop could be never entered if condition is initially false.

While vs do-while loops

While and do-while loops are very similar:

while:

```
while(condition){
   command
}
```

1. Test condition
2. Execute command if true.

do-while:

```
do{
   command
}while(condition);
```

1. Enter loop, run command
2. Test condition, repeat if true.

The difference is that the `while` command **evaluates** *condition* **first** then enters the loop if condition is true.

The `do-while` loop **enters the body** and then **evaluates** *condition* at the end and repeats the loop if the condition is true.

This means a `do-while` loop is guaranteed to be executed at least once.
A `while` loop could be never entered if condition is initially false.

While vs do-while loops

While and do-while loops are very similar:

while:

```
while(condition){
   command
}
```

1. Test condition
2. Execute command if true.

```
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   while( a < 15 ) {
     printf("while-loop: %d\n", a);
      a++;
   }
   return 0;
}
```

do-while:

```
do{
   command
}while(condition);
```

1. Enter loop, run command
2. Test condition, repeat if true.

```
#include <stdio.h>
int main () {
   int a = 10;
   /* while loop execution */
   do {
    printf("do-while-loop: %d\n", a);
    a++;
   } while (a<15);
   return 0;
}
```

Outline EEEE1042 C Lecture 4:

1. Making decisions with if
2. Selecting with switch case
3. Looping with while/do-while
4. Looping with for
5. Nested Loops
6. Function calls
7. Jumping with goto

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize;condition;increment) {
   commands;
}
```

```c
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize ; condition ; increment ) {
   commands ;
}
```

The for construct starts out by executing the initialization condition which typically initializes some counting variable. Multiple initialization commands can be used separated by commas ,

```c
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize; condition; increment) {
   commands;
}
```

Next, the condition is evaluated. If condition is TRUE, control enters the main body of the for loop. Otherwise it transfers to the statement after the end of the for loop. This behaviour is the same as the while loop.

```
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize; condition; increment) {
    commands;
}
```

If condition is TRUE, the loop is entered and commands inside are sequentially executed until they reach the ending brace }

```c
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize ; condition ; increment ) {
   commands ;
}
```

After the body of the loop is done, the increment command is executed. Multiple statements can be placed as increment separated by commas ,

```
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize ; condition ; increment ) {
   commands ;
}
```

After the increment statement, the condition is checked again, and if TRUE, the loop is executed again or the loop is exited if FALSE.

```
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize ; condition ; increment ) {
   commands ;
}
```

The process is repeated until the condition becomes FALSE at which point control goes to the next statement after the end of the for loop.

```c
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

For loops

While loops have one condition that is tested and continues iterating while that condition is true. For loops have 3 parts: an initialization part, a condition, and an increment.

```
for (initialize;condition;increment) {
    commands;
}
```

```c
#include <stdio.h>
int main () {
   int i,j;
   for(i=-3,j=0;i<3;i++,j++ ) {
      printf("i=%d and j=%d\n", i,j);
   }
   return 0;
}
```

Output:

```
i=-3 and j=0
i=-2 and j=1
i=-1 and j=2
i=0 and j=3
i=1 and j=4
i=2 and j=5
```

For-loop example

Compute numerically $S = \sum_{i=1}^{N} i$ and confirm it is equal to $\dfrac{N(N+1)}{2}$

```c
#include <stdio.h>
#define N 6
int main () {
   int i,s=0;
   for(i=1;i<=N;i++) {
      s=s+i; // Increment the sum by i
      printf("i=%2d and s=%2d\n", i,s);
   }
   printf("N*(N+1)/2= %d\n",N*(N+1)/2);
   return 0;
}
```

For-loop example

Compute numerically $S = \sum_{i=1}^{N} i$ and confirm it is equal to $\frac{N(N+1)}{2}$

```c
#include <stdio.h>
#define N 6
int main () {
   int i,s=0;
   for(i=1;i<=N;i++) {
      s=s+i; // Increment the sum by i
      printf("i=%2d and s=%2d\n", i,s);
   }
   printf("N*(N+1)/2= %d\n",N*(N+1)/2);
   return 0;
}
```

Output:

```
i= 1 and s= 1
i= 2 and s= 3
i= 3 and s= 6
i= 4 and s=10
i= 5 and s=15
i= 6 and s=21
i= 7 and s=28
i= 8 and s=36
i= 9 and s=45
N*(N+1)/2= 45
```

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) break;
}
```

```c
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    printf("Enter a string:\n");
    while (scanf("%s",s)>0) {
        if (strcmp(s,"break")==0) break;
        printf("You entered: %s\n", s);
        printf("Enter another string:\n");
    }
    printf("Broke out of the loop.\n");
    return 0;
}
```

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) break;
}
```

The break command breaks out of the inner-most loop, thereby terminating the loop. Control resumes at the first command after the loop.

```
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    printf("Enter a string:\n");
    while (scanf("%s",s)>0) {
        if (strcmp(s,"break")==0) break;
        printf("You entered: %s\n", s);
        printf("Enter another string:\n");
    }
    printf("Broke out of the loop.\n");
    return 0;
}
```

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) break;
}
```

```
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    printf("Enter a string:\n");
    while (scanf("%s",s)>0) {
        if (strcmp(s,"break")==0) break;
        printf("You entered: %s\n", s);
        printf("Enter another string:\n");
    }
    printf("Broke out of the loop.\n");
    return 0;
}
```

The break function jumps to here when executed.

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) continue;
}
```

```
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    for (i=0;i<10;i++) {
        printf("Enter a string:\n");
        scanf("%s",s);
        if (strcmp(s,"continue")==0) continue;
        printf("i=%d, You entered: %s\n",i,s);
    }
    printf("End of the for loop.\n");
    return 0;
}
```

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) continue;
}
```

The continue command skips everything in the remainder of the for block and begins the next iteration. The next statement to be executed after the continue is the *increment*, followed by testing the *condition* and re-entering the top of the block if it's TRUE.

```
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    for (i=0;i<10;i++) {
        printf("Enter a string:\n");
        scanf("%s",s);
        if (strcmp(s,"continue")==0) continue;
        printf("i=%d, You entered: %s\n",i,s);
    }
    printf("End of the for loop.\n");
    return 0;
}
```

break and continue

break and continue statements are used inside loops to affect the control of flow. They will not happen every iteration through the loop, they only happen when some other condition occurs.

```
for (initialize;condition;increment) {
   if(some condition happens) continue;
}
```

```
#include <stdio.h>
#include <string.h>
int main () {
    int i;
    char s[100];
    for (i=0;i<10;i++) {
        printf("Enter a string:\n");
        scanf("%s",s);
        if (strcmp(s,"continue")==0) continue;
        printf("i=%d, You entered: %s\n",i,s);
    }
    printf("End of the for loop.\n");
    return 0;
}
```

The continue statement skips the remainder of the block and continues to the next iteration.

Outline EEEE1042 C Lecture 4:

1. Making decisions with `if`
2. Selecting with `switch case`
3. Looping with `while`/`do-while`
4. Looping with `for`
5. **Nested Loops**
6. Function calls
7. Jumping with `goto`

Nested loops

In more complex programs, loops are often nested one inside the other:

```c
#include <stdio.h>
int main () {
   int i,j;
   for (i=0;i<3;i++) {
      printf("Outer loop i=%d.\n",i);
      for (j=0;j<3;j++) {
         printf("\t Inner loop j=%d.\n",j);
      }
   }
   printf("End of the for loop.\n");
   return 0;
}
```

Nested loops

In more complex programs, loops are often nested one inside the other:

```c
#include <stdio.h>
int main () {
  int i,j;
  for (i=0;i<3;i++) {
    printf("Outer loop i=%d.\n",i);
    for (j=0;j<3;j++) {
      printf("\t Inner loop j=%d.\n",j
    }
  }
  printf("End of the for loop.\n");
  return 0;
}
```

Output:

```
Outer loop i=0.
         Inner loop j=0.
         Inner loop j=1.
         Inner loop j=2.
Outer loop i=1.
         Inner loop j=0.
         Inner loop j=1.
         Inner loop j=2.
Outer loop i=2.
         Inner loop j=0.
         Inner loop j=1.
         Inner loop j=2.
```

Nested loops

Nested loops are often used in processing or printing 2D arrays.

```
#include <stdio.h>
int main () {
   int i,j;
   float a[3][4]={
         {0, 1, 2, 3} , /* initializers for a[0] */
         {4, 5, 6, 7} , /* initializers for a[1] */
         {8, 9, 10, 11} /* initializers for a[2] */
   };
   for (i=0;i<3;i++) {
      for (j=0;j<4;j++) {
         printf(" %f",a[i][j]);
      }
      printf("\n");
   }
   printf("End of the double for loop.\n");
   return 0;
}
```

Nested loops

Nested loops are often used in processing or printing 2D arrays.

```
#include <stdio.h>
int main () {
   int i,j;
   float a[3][4]={
         {0, 1, 2, 3} , /*
         {4, 5, 6, 7} , /* initializers for a[1] */
         {8, 9, 10, 11} /* initializers for a[2] */
   };
   for (i=0;i<3;i++) {
      for (j=0;j<4;j++) {
         printf(" %f",a[i][j]);
      }
      printf("\n");
   }
   printf("End of the double for loop.\n");
   return 0;
}
```

Output:

```
0.000000 1.000000 2.000000 3.000000
4.000000 5.000000 6.000000 7.000000
8.000000 9.000000 10.000000 11.000000
End of the double for loop.
```

Outline EEEE1042 C Lecture 4:

1. Making decisions with `if`
2. Selecting with `switch case`
3. Looping with `while`/`do-while`
4. Looping with `for`
5. Nested Loops
6. Function calls
7. Jumping with goto

Function calls

The flow of control is also impacted by putting your codes into subfunctions which are then called from outside:

```
returnType functionName(type1 parm1, type2 parm2, ...) {
   functionDefCommands;
}
  ⋮
z=functionName(x,y,...); // Call function defined above.
```

```c
#include <stdio.h>
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    printf("In main function, x=%d\n",x);
    return 0;
}
```

Function calls

The flow of control is also impacted by putting your codes into subfunctions which are then called from outside:

```
returnType functionName(type1 parm1, type2 parm2, ...) {
   functionDefCommands;
}
  ⋮
z=functionName(x,y,...); // Call function defined above.
```

```c
#include <stdio.h>
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    printf("In main function, x=%d\n",x);
    return 0;
}
```

Output:

```
In function printAndAdd: x+y=9
In main function, x=9
```

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in `.h` files for re-use in other programs.

```c
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration

int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}

int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in .h files for re-use in other programs.

```
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration


int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}


int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

The function declaration, declares what the function does to the compiler, just like `int x;` declares `x` to be an `int` to the compiler. After seeing this line, the compiler knows how to handle the function call: taking 2 `int`s and returning another `int`.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in `.h` files for re-use in other programs.

```
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration

int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}

int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

Function declarations should occur **before first use** of the function, and are typically stored in a header `.h` file that can be packed and reused by other programs.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in .h files for re-use in other programs.

```
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration


int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}


int printAndAdd (int x, int y) { // functi
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

Sometimes if you are being lazy, and hacking together a quick test function, you can use the function definition to declare the function by putting the definition above you main, and doing away with the function declaration altogether. Then the compiler will know how to handle the function when it's called.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in .h files for re-use in other programs.

```
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration

int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}

int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

This is the function call. When the compiler meets this line, it knows `printAndAdd` is a function that takes two `int`s as inputs and returns an `int` as output. So it can handle this line which is doing what is expected.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in `.h` files for re-use in other programs.

```c
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration


int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}


int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd:  x+y=%d\n",z);
    return(z);
}
```

This is the function definition. The compiler can meet this line in the form of source code or as object code or from another library. If it's source code, it is compiled to object code.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in .h files for re-use in other programs.

```c
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration


int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}


int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

So long as the object code is available to the linker, it will link up all the object code modules to form the executable.

Function declarations

Function declarations make the function known to the compiler before first use. They can be stored in `.h` files for re-use in other programs.

```
#include <stdio.h>
int printAndAdd (int x, int y); // function declaration


int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}

int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

Modular programming practice puts all code that performs a self-contained task into its own subfunction, and will store those declarations/subfunctions in a header-file and library so they can be re-used.

Outline EEEE1042 C Lecture 4:

1. Making decisions with `if`
2. Selecting with `switch case`
3. Looping with `while`/`do-while`
4. Looping with `for`
5. Nested Loops
6. Function calls
7. Jumping with goto

Jumping with goto

The goto command instructs the program execution to jump to a different labeled location in the code:

```
goto abcd;
  ⋮
other code;
  ⋮
abcd:
```

Jumping with goto

The goto command instructs the program execution to jump to a different labeled location in the code:

```
goto abcd;
   ⋮
other code;
   ⋮
abcd:
```

The goto construct transfers execution control to the named label in the code. It is generally an antiquated command that does not see much use today. The reason is because it's usage tends to encourage unmodular programming which becomes difficult to maintain. See here for an explanation.

Jumping with goto

The goto command instructs the program execution to jump to a different labeled location in the code:

```
goto abcd;
   ⋮
other code;
   ⋮
abcd:
```

The goto construct transfers execution control to the named label in the code. It is generally an antiquated command that does not see much use today. The reason is because it's usage tends to encourage unmodular programming which becomes difficult to maintain. See here for an explanation.

```c
#include <stdio.h>
int main () {
   int i,j;
   for (i=0;i<3;i++) {
      for (j=0;j<4;j++) {
         printf(" (i,j)=(%d,%d)\n",i,j);
         if (i==1 && j==2) goto exitLoop;
      }
      printf("\n");
   }
   exitLoop:
   return 0;
}
```

Jumping with goto

The goto command instructs the program execution to jump to a different labeled location in the code:

```
goto abcd;
   ⋮
other code;
   ⋮
abcd:
```

The goto construct transfers execution control to the named label in the code. It is generally an antiquated command that does not see much use today. The reason is because it's usage tends to encourage unmodular programming which becomes difficult to maintain. See here for an explanation.

```c
#include <stdio.h>
int main () {
   int i,j;
   for (i=0;i<3;i++) {
      for (j=0;j<4;j++) {
         printf(" (i,j)=(%d,%d)\n",i,j);
         if (i==1 && j==2) goto exitLoop;
      }
      printf("\n");
   }
   exitLoop:
   return 0;
}
```

Output:

```
 (i,j)=(0,0)
 (i,j)=(0,1)
 (i,j)=(0,2)
 (i,j)=(0,3)

 (i,j)=(1,0)
 (i,j)=(1,1)
 (i,j)=(1,2)
```

External resources

Some external sites you can go to learn about C for, while, if :

- www.mycplus.com
- www.guru99.com
- CProgramming blog
- www.tutorialspoint.com