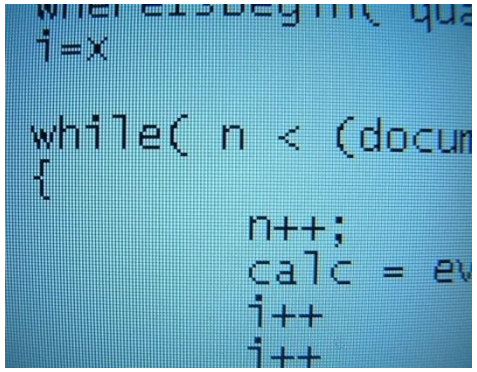
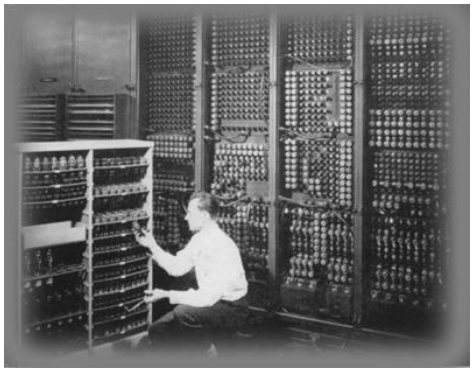
University of Reading





CS1PR16

Introduction to C: Syntax and Semantics

Learning Objectives

University of Reading

- Distinguish the terms semantics and syntax
- Formulate syntactical definitions using EBNF
- Describe the workflow of creating and running programs
- Describe the syntax and semantics of basic C-Programs
- Apply the principles of sequence, selection, iteration
  - to write simple programs in the programming language C

Outline

University of Reading

- Background to the C programming language
- Syntax vs Semantics
- The extended Backus-Naur form (EBNF)
- Basics of the C programming language
  - Sequence
  - Selection
  - Iteration

Definitions

University of Reading

*A **programming language** is a **vocabulary** and set of **grammatical rules** for instructing a computer or computing device to **perform specific tasks***

[Wikipedia]

Syntax: *the **set of rules, principles**, and processes that govern the **structure of sentences***

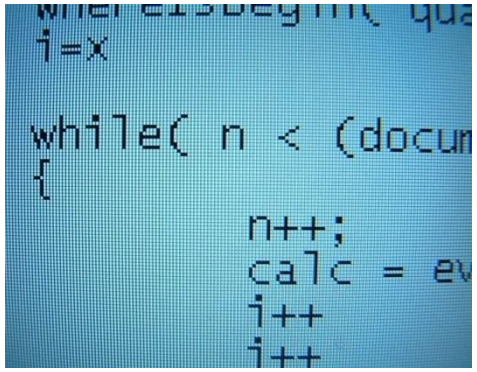
[Wikipedia]

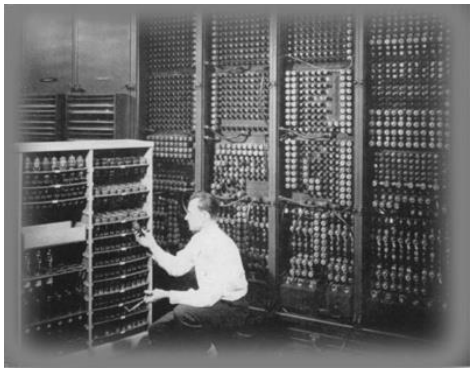
- How do programs "look like", what is a syntactically correct program?
- Standardised representation (e.g., specific Pseudocode)

Semantics: the linguistic and philosophical study of **meaning** in language, **programming languages** …

[Wikipedia]

- What does a written program mean, what does it do?
- What is the semantics causing the program to do?





University of Reading

Background to the C programming language

What is C?

University of Reading

- C is a low-level programming language
  - Allows to be translated into assembler that can be executed by a computer
  - Direct control over hardware
    - highest performance, enables to use hardware features
- History:
  - C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories for use with the Unix operating system
  - In 1989, the *American National Standards Institute* (ANSI) published ANSI X3.159-1989 "*Programming Language C*", often referred to as ANSI C
- Standardisation:
  - Nowadays managed by the **International Standards Organisation (ISO)**
    - part of the United Nations
  - Current standard is C2017
  - The next standard is C202X

The First Program: Hello World

University of Reading

Stored in a file called hello.c

```
#include <stdio.h>
int main(){
        /* first examples traditionally say this */
        printf("Hello World\n");
        return 0;
}
```

- **Syntax**: **how the program looks like?**
  - Uses characters like #, { } , ( ) , ; , " and **keywords** (printf, return)
  - This is a **valid** C program
- **Semantics: what does the program do?**
  - It outputs on the console "Hello World" followed by a newline
  - This appears to be a program that makes "sense" semantically
- The standard defines both: the Syntax and the Semantics
- How do we run this program?

Step by Step: for the code

University of Reading

- The details are here for your reference; will be discussed!
- int main(void)
  - This line specifies the main function
    - It defines where the program will start executing code
  - You need this in every program
- printf("Text");
  - A function used for printing on the screen (console window)
    - defined in the file stdio.h
  - Any character with a \ before is an 'escape character'
    - \n – New Line or \t – Tab
- return 0;
  - The return statement
  - Value after return is returned to who called the current function
  - Immediately ends the current function

Executing C and Assembler

University of Reading

Stored in a file called hello.c

IP →

```
#include <stdio.h>
int main(){
        /* first examples traditionally say this */
        printf("Hello World\n");
        return 0;
}
```

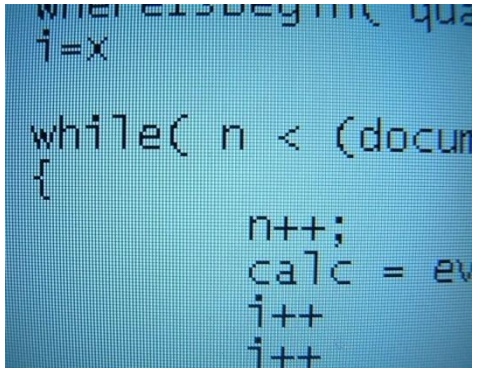
- When executing the program, imagine the instruction pointer
  - Sequentially processes each statement (ended by ;)

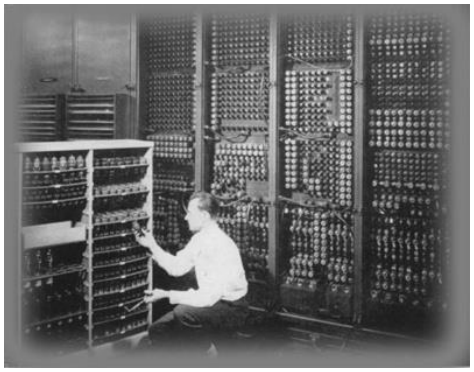
- In reality, a statement may represents many assembler instructions
  - The compiler makes the correct translation
  - Also some statements don't result in actual instructions…

Formatting

University of Reading

- The braces { and } are blocks containing statements
  - We indent/tabulate the code inside – makes it easier to read
- In C semicolons ' ; ' are used to terminate statements
- C is free format
  - Line breaks and white space can go between items
  - We provide one statement per line (for readability)
- C is case sensitive
  - The identifier `Printf` is different from `printf`





The extended Backus-Naur form (EBNF)

The Extended Backus-Naur Form

University of Reading

- EBNF = Metasyntax notation: language to specifies syntax
  - Extended BNF: There are many variants, we use a simplified variant
  - The EBNF is a grammar that is used to define a "language"
  - All texts are part of the language that can be "produced" by the grammar
- Elements of the EBNF:
  - Terminal symbol (string), the actual "text"
    - Example: "1" or "number" => **we often omit** " " if it is unambiguous
  - Non-terminal: variable that define production rules
    - Example: one = "1"
  - Production rules: define what a non-terminal can produce
    - | = OR, either the left or the right production must be applied
    - [x] option (1 or 0 times)
    - {x} repetition (0 to n times)
    - (x) grouping: belongs together; useful for OR
  - Comments: /* **MY COMMENT** */, document rules, support understandability

EBNF: Example

University of Reading

- Example: numbers

  digit1 = "1" | "2" | "3" | "4" | "5" | "6"| "7" | "8" | "9"

  /* This means that the non-terminal symbol digit1 is the text 1-9 */

  digit = "0"|digit1 /* a digit is either 0 OR 1-9 */

  number = "0"| digit1 {digit}

  /* A number is either 0 OR 1-9 followed by any number of digits*/

- For any number, we can write how it is produced:

  e.g., 120 is an element of the language of the numbers, produced by:

  number -> digit1 {digit} -> 1 {digit} -> 1 digit1 {digit} -> 12 {digit} -> 120

  – "->" means apply a production rule
  – In the example, we applied them on the leftmost possibility

- Any string that cannot be produced by this "grammar" is not a part of the language, e.g., "1a2"

EBNF Further

University of Reading

- EBNF of mathematical expressions (excerpt)

  expression = number " + " number | (number " + " number)

  | number " * " number | (number " * " number) | number

- It is sometimes not easy to define a language with EBNF!
- To reduce confusion add <> around non-terminals
  - Sometimes people use UPPERCASE as well
- An alternative formulation for repetition and option is:

  <RULE> = <SOMETHING>| <SOMETHING> <RULE>

  Example: <DIGITS> = <DIGIT> | <DIGIT> <DIGITS>

Group Work 1

University of Reading

Task

Write down the EBNF formulation of the language of all palindromes

- Palindrome: Word that can be read forwards and backwards
  - e.g. HANNAH, GIG
- Time: 3 min
- Share: 2 min

Group Work 1: Solution

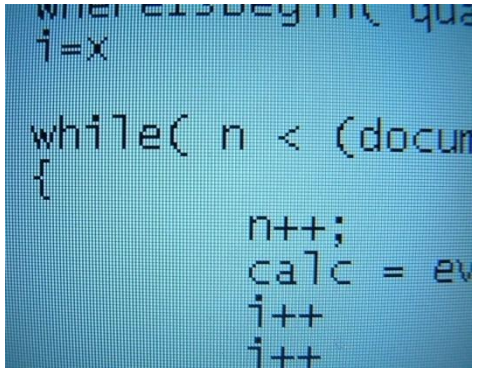
University of Reading

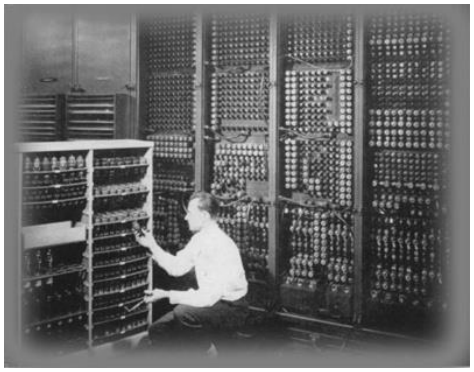
<pal> = "a" <pal> "a" |"b" <pal> "b" | … | "z" <pal> "z"

| "a" | "b" | … |"z"

To generate gig:

- <pal> -> g <pal> g -> gig





University of Reading

Basics of the C programming language

The Syntax of the C Language

University of Reading

- We'll follow the ISO 9899:202x draft
- Remember, we must cover: Sequence, Selection, Iteration
  - We will look at the C syntax in EBNF
- A source file is written using the ASCII (or UTF) character set
  - Characters: a-z, A-Z
  - Digits: 0-9
  - Graphic characters:
    - ! " # % & ' ( ) + * , - . / : ; < = > ? [\ ] ^ _ { | } ~
  - Control characters, i.e., non-printable character
    - Example: whitespace: space, newline, tabulator

Comments

University of Reading

- **Comments are not considered to be part of the language**
  - They are removed before the code is compiled
- Everything after // is a comment until the end of that line
- Comments added starting with /* and ending with */
  - Everything in-between is ignored
- **Always Comment Code**
  - Use comments to remove ambiguity and document the program
  - Companies often have standards for coding, and these include how to comment

Simplified C Syntax: Sequence

University of Reading

- A compound statement are *declarations* followed by statements

  <compound-statement> = {<declaration>} {<statement>}

- A statement executes an operation that can be of many types:

  <statement> = <labeled-statement>

  | <expression-statement>

  | **<compound-statement> /* this nesting allows to produce a sequence */**

  | <selection-statement>

  | <iteration-statement>

  | <jump-statement>

- A typical mathematical assignment is an **expression-statement**
  - **<expression-statement>** = [<expression>] ";" */* The ; is important */*
  - *Example: x = 2*5 + y;*
    - *Assign x the value that is defined by the expression on the right side*
- **Expressions evaluate to a value**
- Statements can be grouped together into blocks

  "{" <compound-statement> "}"

Operators used in Expressions

University of Reading

- Operators define the semantics of the expression
- Arithmetic operators

  `+, -, *, / (division), %(modulo)`
- Unary operators `(in contrast to binary, apply to one value)`

  `++(increment), --(decrement)`
- Logical operators

  `&& (and), || (or), ! (negation)`
- Bitwise operators

  `& (and), | (or), ^ (xor), ~ (complement)`
- Bitwise shift operators

  `<< (shift left), >> (shift right)`
- Relational operators

  `==, <, <=, >, >=, != (not equal)`
- Assignment operators

  `=, +=, *=, /=, %=`

Sequence: Examples

University of Reading

- Sequence of two expression-statements:

```
x = 5 + 5; // compute 5 + 5 then assign the result to the variable x
y = (x + 1)*x;  // use the x from the previous line!
```

- Generally, code is evaluated statement by statement, left to right
  - E.g. first compute x, then compute (x+1), then multiply result by x and assign it to variable y
  - **There are operator precedence rules that define the order of the evaluation**
  - **You can use "(<expression>)" to first evaluate expression**
- Further interesting expressions:

```
y = x == z; // compare x and z for identity, if they are, y is TRUE otherwise FALSE
y += 5; // is identical to y = y + 5;
z = x++; // increment operator, add one to x and set z to the result
z = 5 & a; // bitwise operator
z = 5 && a; // logical operator
```

Increment and Decrement

University of Reading

- To add 1 to a variable, you would write x = x + 1;
- Many CPU's have fast increment and decrement ops
  - To utilise these use ++ or --
- If the operator is in front of the variable, e.g. ++x, then the variable is incremented or decremented before being used
  - pre-increment or pre-decrement
- If the operator is after the variable, e.g. x--, then the variable is incremented or decremented after being used
  - post-increment or post-decrement

Increment and Decrement

University of Reading

```
#include <stdio.h>

int main()
{ // This starts a block of compound-expressions
  int thisYear = 2017;
  printf("1 Current year is %d\n", thisYear);
  printf("2 Next year will be %d\n", ++thisYear);
  printf("3 Previous year was %d\n", --thisYear);
  printf("4 thisYear++ gives %d\n", thisYear++);
  { // start another block
    printf("5 thisYear is now %d\n", thisYear);
  } // end another block
  return 0;
} // This ends the block
```

What will be the value of thisYear on the different printf's?

Time: 1 Minute

Logical Operators

University of Reading

- Logic knows two states for a statement
  - True (e.g., the sun looks yellow)
  - False (e.g., I'm 150 years old)
- In C
  - 0 represents False
  - 1, or any non zero value, represents True
- Logical operations: truth tables define the outcome

Truth Tables

University of Reading

&& is logical AND

Expression1	Expression2	Expression1 && Expression2
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

|| is logical OR

Expression1	Expression2	Expression1 || Expression2
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

! is not (negation)

Expression	!Expression
FALSE	TRUE
TRUE	FALSE

Bitwise Operators (again)

University of Reading

- Bitwise operators apply logical operators on each bit
  - Allows the manipulation of individual bits!
- Take two Bytes a = $[a7 \ldots a0]_2$ and b = $[b7 \ldots b0]_2$
  - Result c = $[c7 \ldots c0]_2$ where
  - Combine c7 = a7 <op> b7, a6 <op> b6, ..., a0 <op> b0
- And: <op> = & => a & b
  - Result: ci = 1 if ai = 1 AND bi = 1 (0 otherwise)
- Or: a | b
  - Result: ci = 1 if ai = 1 OR bi = 1 (or both)
- Xor: a ^ b
  - Result: ci = 1 if one of ai or bi is 1
- Complement: ~a
  - Result: ci = 0 if ai = 1 and ci = 1 if ai = 0

Bitwise Shift Operators

University of Reading

- Move the bits around
- b << i shifts the bits in b by i positions to the left
  - Filling new positions with zeros
  - Assume b = $[b7 \ldots b0]_2$
  - Example: c = b << 2
    - c = $[b5\ b4\ b3\ b2\ b1\ b0\ 0\ 0]_2$
- a >> i shifts the bits in A by i positions to the right
  - Filling new positions with zeros (other modes are possible)
- Question: what does the bitshifting do to binary numbers?
  - What is the (decimal) result of $[00000001]_2$ << 2

Relational Operators

University of Reading

Standard Algebraic Equality Operator or Relational Operator	C Equality or Relational Operator	Example of C Condition	Meaning of C Condition
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
>	>	x > y	x is greater than y
<	<	x < y	x is lesser than y
>=	>=	x >= y	x is greater than or equal to y
<=	<=	x <= y	x is lesser than or equal to y

Assignment Operators

University of Reading

- = means 'becomes' or gets; not equals in the math. sense
- Evaluates the expression to the **right-hand side (rvalue)**
- Assigns the value to the expression on the **left-hand side (lvalue)**

- For example:

```
x = x + y;
x = y * 1.05;
```

Confusing Equality (==) and Assignment (=) Operators

University of Reading

- Sometimes, we wrongly use = instead of == to test equality
  - Does not ordinarily cause syntax errors, still compiles
- Example, consider

```
if( payCode = 4 )     /* likely an error */
   printf("You get a bonus!\n");
```

- The expression `payCode = 4` is first evaluated
  - assigning payCode to the value 4, this also evaluates to 4!
- Then the condition is evaluated which is 4
  - This is non-zero, hence the if will always be true!

Declarations

University of Reading

- So far, we used variables but didn't think about what **objects** they are
  - Are they numbers? Natural numbers, real numbers, complex numbers?
- Remember: compound statement are *declarations* followed by statements

  <compound-statement> = {<declaration>} {<statement>}
- C uses a static type system
  - A "variable" is "declared" to be of a specific type, it becomes an object at runtime
  - The compiler remembers the type and ensures that the type is used correctly
  - **Variables must be declared before they can be used**
- **Basic types**
  - int: integer, **whole number**, e.g., INT_MIN, … -10, …, 0, 1, …, INT_MAX
  - unsigned integer: **natural number**, e.g., 0, ..., UINT_MAX
  - float: floating-point number (nearly like a real number), e.g., 1.52451
  - double: more precise floating-point number (more digits!)
  - char: a single character
- String:
  - char *: an array of characters (more information later)

Standard Types

The standard integer types are portable (platform independent)

- Use: `#include <stdint.h>`
  - `#include <stdbool.h>`
- The size (e.g., 64) indicates the number of bits!

- Bool: a truth value (TRUE = 1 or FALSE = 0)
- int8_t: an 8-bit integer
- int16_t: a 16-bit integer
- int32_t: a 32-bit integer
- int64_t: a 64-bit integer
- uint8_t: an 8-bit UNSIGNED integer
- …
- uint64_t: a 64-bit UNSIGNED integer
- Note that integer division truncates remainder: 7 / 5 = 1
- Modulus operator (%) returns the remainder: 7 % 5 = 2

Range of Values

University of Reading

Type	Bytes	Bits	Range
char	1	8	-128 to +127
int8_t	1	8	-128 to 127
uint8_t	1	8	0 to 255
int32_t	4	32	-2,147,483,648 to +2,147,483,647
uint32_t	4	32	0 to +4,294,967,295

Generally for an unsigned type:

- $\mathbf{max} = 2^b - 1$ **, i.e., use every bit for the representation**
- $min = 0$

For a signed type, data is stored in 2-complement (you'll learn later)

- $\mathbf{max} = 2^{b-1} - 1$
- $\mathbf{min} = -2^{b-1}$

Declaration of Types

University of Reading

- Syntax: `<Declaration> = <Type> <Identifier>;`
- A variable is an object referred to by a symbolic name, the identifier
  - This is a unique name ("x", or "this_is_my_variable")
- Example:

```
int x;
x = 5 / 2;  // WARNING, division with integers lead to whole numbers!
// x is now 2 !
// Use the modulo operator to get the remainder!
x = 5 % 2;  // The result is 1
```

- Example 2:

```
char v;
v = 'a';  // variable v is set to the ASCII value of the character a
char * string = "This is a string";
```

Valid Identifiers

University of Reading

- The symbolic names of an identifier:
  - Can consist of letters, numbers and underscores (_)
  - Cannot start with a number
  - Cannot be a C keyword

- For example, the following are valid identifiers:

  Num1, num1, _answer

- The following are invalid identifiers:

  1test, #hash, C S 1 PC (whitespace), int

C Built-in keywords

University of Reading

- There are a number of built-in keywords in C
  - Types and qualifiers:
    - `void, int, char, float, double, short, long, signed, unsigned, …`
    - `register, static, volatile, const, extern, restrict`
  - Conditional/Looping:
    - `for, if, else, break, switch, case, continue, goto, do, while`
  - Types and structures:
    - `enum, typedef, struct, union, sizeof`
  - Functions
    - `return`
- As you can see, there are not many!
- Keywords cannot be used as an identifier

Group Work 2: Variables

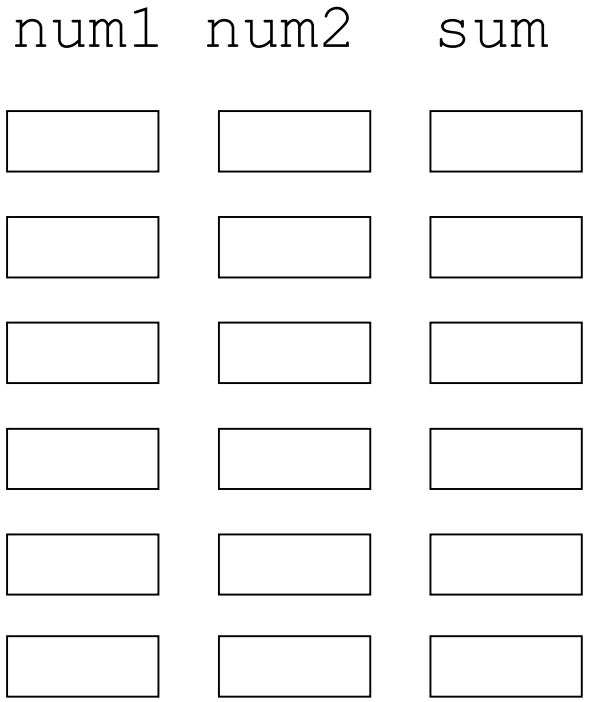
University of Reading

Task

Think about the value of the variables as the program executes:

Use "?" if you cannot know (**undefined**)

```
int num1;
int num2;
int sum;
num1 = 2;
num2 = 5;
sum = num1 + num2;
```



- Time: 2 min; Share: 2 min

Example Program

University of Reading

```c
#include <stdio.h>

int main(void){
  // Declaration (Definition to be taught soon)
  int num = 12; // define and assign in one step
  char letter;
  float fNum;

  // statements
  letter = 'A';
  fNum = 12.34;
  num = num + 5;

  // let's output the value of the objects:
  // printf() uses format specifiers to output values
  printf("num is %d \n", num); // %d: print integer
  printf("letter is %c \n", letter); // %c: character
  printf("fNum is %f \n", fNum); // %f: floating-point

  return 0;
}
```

What does this code output?

Selection

University of Reading

- Remember: a statement can be of many types:

  <statement> = <labeled-statement>

  | <expression-statement>

  | <compound-statement> (* note that this produces sequence *)

  | <selection-statement>

  | <iteration-statement>

  | <jump-statement>

- There are two types of selection statements:

  <selection-statement> = if (<expression>) <statement>

  | if (<expression>) <statement> else <statement>

  | switch (<expression>) <statement>

- And a special ternary expression that allows for selection:

  <expression> = <expression> ? <expression> : <expression>

  Ternary: three parts

Ternary Operator

University of Reading

- The ternary operator is a special expression

  <expression> = <expression> ? <expression> : <expression>

- It allows the evaluation of an expression depending on a condition
- Works like this:

```
condition ? if true expression : if false expression
```

- Examples

```
min = (x < min) ? x : min;
```

  What does this code do?

- Can be used as a statement, too (disposing the value):

```
(ourCash > 0) ? printf("Solvent\n") : printf("Broke\n");
```

The if Conditional Statement

University of Reading

- if checks the value of an expression and runs the statement if TRUE

  if (<expression>) <statement>

- Use the following notation:

```
if( condition ){
   // Code to be executed if the condition is true
}
```

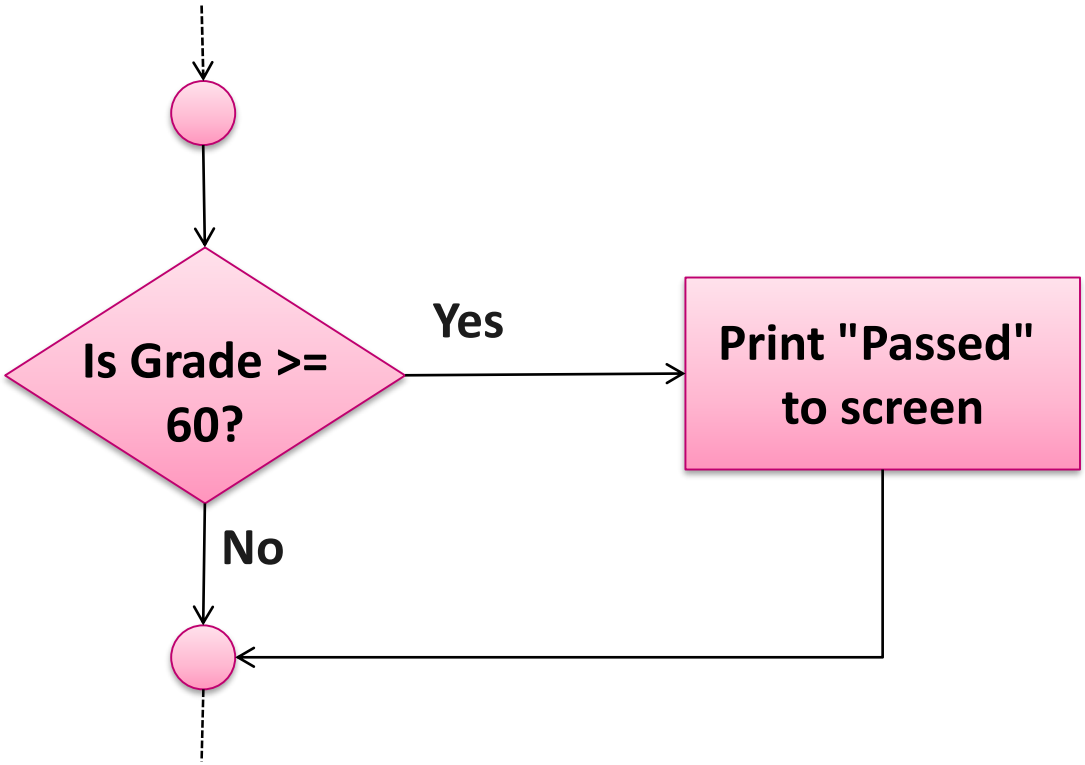
- If the condition is TRUE, then the program executes the statement
- If the condition is FALSE, then the statement is ignored

The if Conditional Statement

University of Reading

- Example pseudocode:

```
If a student's grade is greater or equal than 60
   Print "Passed"
End if
```



The if Conditional Statement

University of Reading

- We want the >= comparison operator
- If we assume grade is of type `int`

```
if( grade >= 60 )
{
   printf("Passed");
}
```

Remembering false is 0 and non-zero is true, then a decision can be made on any expression.

Example:

3 - 4 is true

Why? – The decision is done by subtraction and results in a non-zero answer

`if…else` conditional statement

University of Reading

- if may be supplemented by the else command.

- Use the `else` statement to indicate what to do when the condition is FALSE (0)

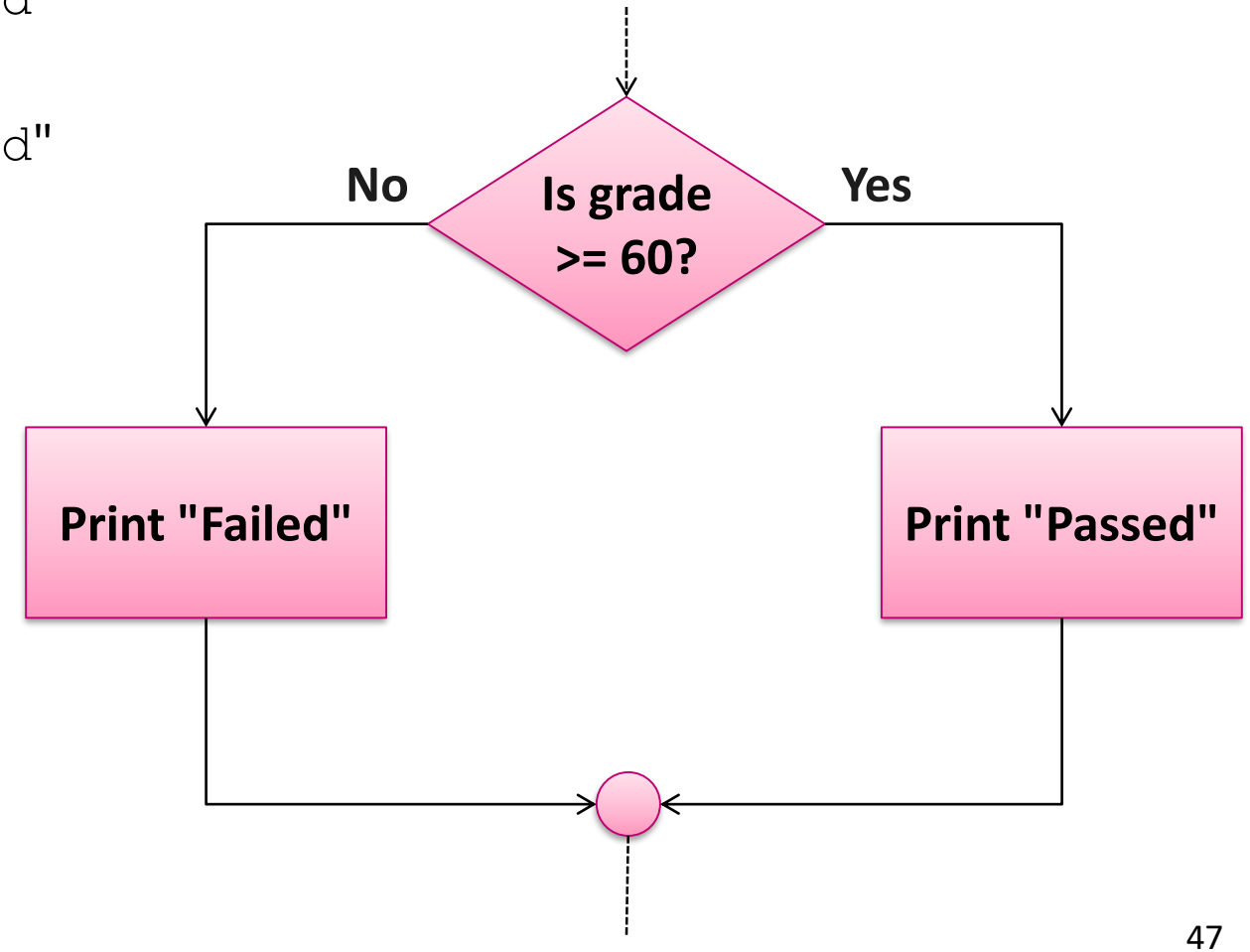
```
if( condition ){
    // Code is executed when condition is true
}else{
    // Code is executed when condition is false
}
```

if…else conditional statement

University of Reading

Pseudocode:

```
If a student's grade is >= 60
  Print "Passed"
else
  Print "Failed"
```



if…else conditional statement

University of Reading

Don't need { and } for just one line of code, however recommended for clarity

- In C code:

```
if( grade >= 60 ){
    printf("Passed\n");
}else{
    printf("Failed\n");
}
```

- Notice the tabulation to help read the code
- Placed { and } to define blocks of statements

Nested `if…else`

University of Reading

- Nest if...else statements together to test for many conditions
- Nested if…else statements
- Once a condition is met, then the rest of the statements are skipped

- Deep indentation usually not used in practice
  - Consider using Functions (taught later)

Nested `if…else`

University of Reading

- To have more than one statement to execute after the condition, we need to compound the statements in a block:

```
if ( grade >= 60 ){
   printf("Passed.\n");
}else{
   printf("Failed.\n");
   printf("You must take this again.\n");
}
```

- What would be printed without the braces?
- It is good practice always to add a block!

Nested if…else

University of Reading

- Remember: always add a block and indent correctly
- This code has a nested if-else, but it is difficult to judge what it does!
  - Ambiguity of else => where does it belong to?

```
if( grade >= 90 )
  printf("Grade A\n");
else
if( grade >= 80 )
  printf("Grade B\n");
else
if( grade >= 70 )
  printf("Grade C\n");
else
  printf("Failed\n");
```

Last else is the "catch the rest" condition.

Switch Statement

University of Reading

- The switch statement can be used for integer datatypes
- Instead of many if...else statements, add a switch:

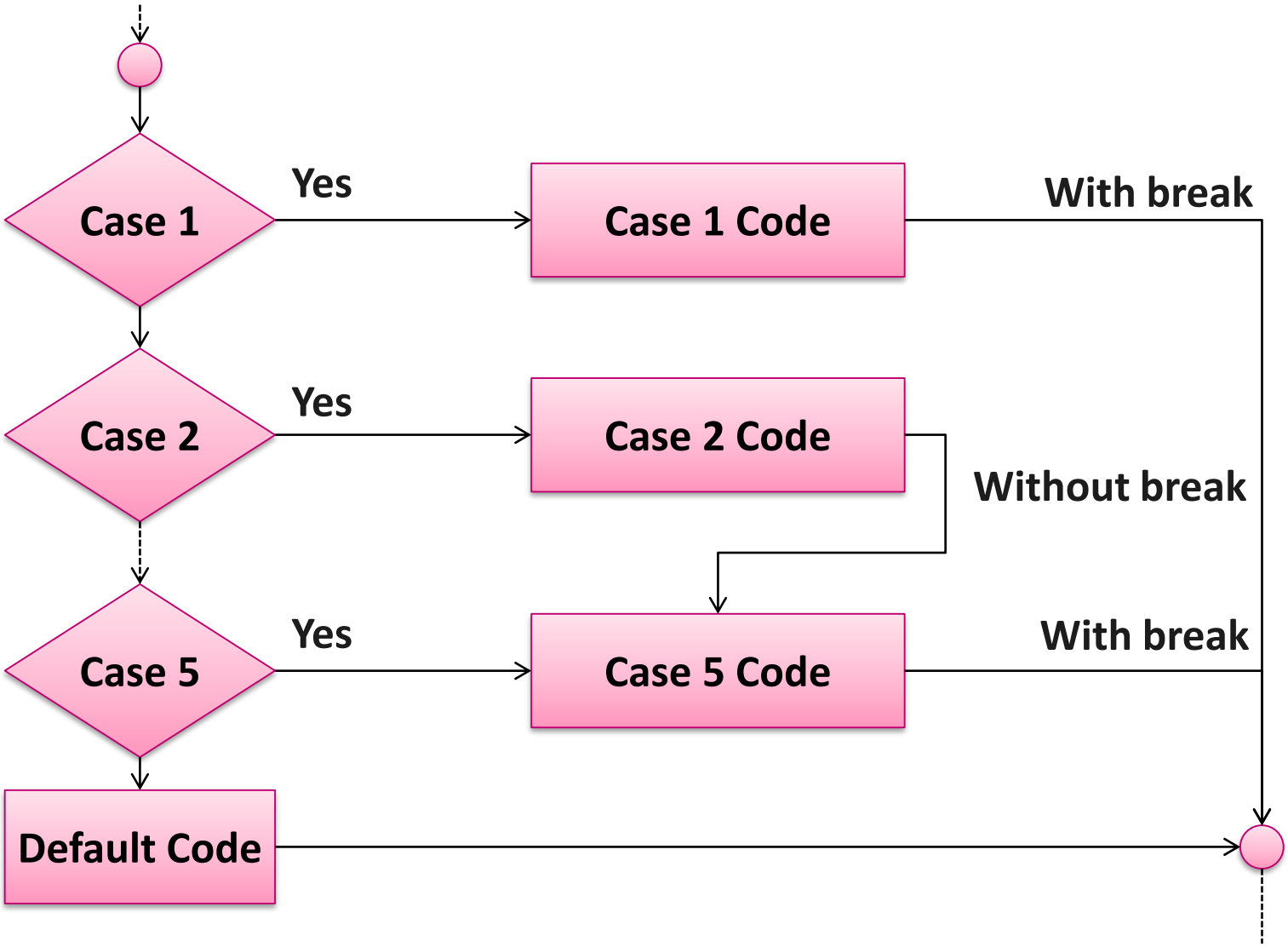
```
switch(number)
{
  case 1: printf("Special case, 1\n"); break;
  case 2:
  case 3: printf("prime\n"); break;
  case 4: printf("even\n"); break;
  case 5: printf("prime\n"); break;

  default : printf("out of range\n"); break;
}
```

Note "break"

Note no “break” here

Switch Statement

University of Reading



Iteration: Jumps

University of Reading

- Remember: a statement can be of many types:

  <statement> = <labeled-statement>
  | <expression-statement>
  | <compound-statement> (* note that this produces sequence *)
  | <selection-statement>
  | <iteration-statement>
  | <jump-statement>

- A jump statement and labeled statement belong together

  <label>: // note the ":"
  <statement>
  <statements>
  goto <label>;

- This means that after completing the <statements>
  - The next instruction to execute will be the statement after the label
  - That can lead to infinite loops!
  - Typically, the goto statement is combined with selection!

Example Iteration

University of Reading

```
#include <stdio.h>
int main(){
  int n=0;

  loop: ;
    printf("\n%d", n);
    n++;
    if ( n < 10 ){
      goto loop;
    }
  return 0;
}
```

What does this code do?

The Practice of Jumps

University of Reading

- Strictly, goto is never needed in a structured program

  "[…] code that relies on goto statements is generally harder to understand and maintain […] goto should be used rarely, if at all"

  (Kernighan & Richie)

- Instead, we use for the control flow the loop constructs
  - while, for

- All of them are implemented under the hood "using" goto

Iteration

University of Reading

- Remember: a statement can be of many types:

  <statement> = <labeled-statement>
  | <expression-statement>
  | <compound-statement> (* note that this produces sequence *)
  | <selection-statement>
  | <iteration-statement>
  | <jump-statement>

- There are three equivalent types of loops in C
  - Each being convenient for another case
  - They can be transformed to each other

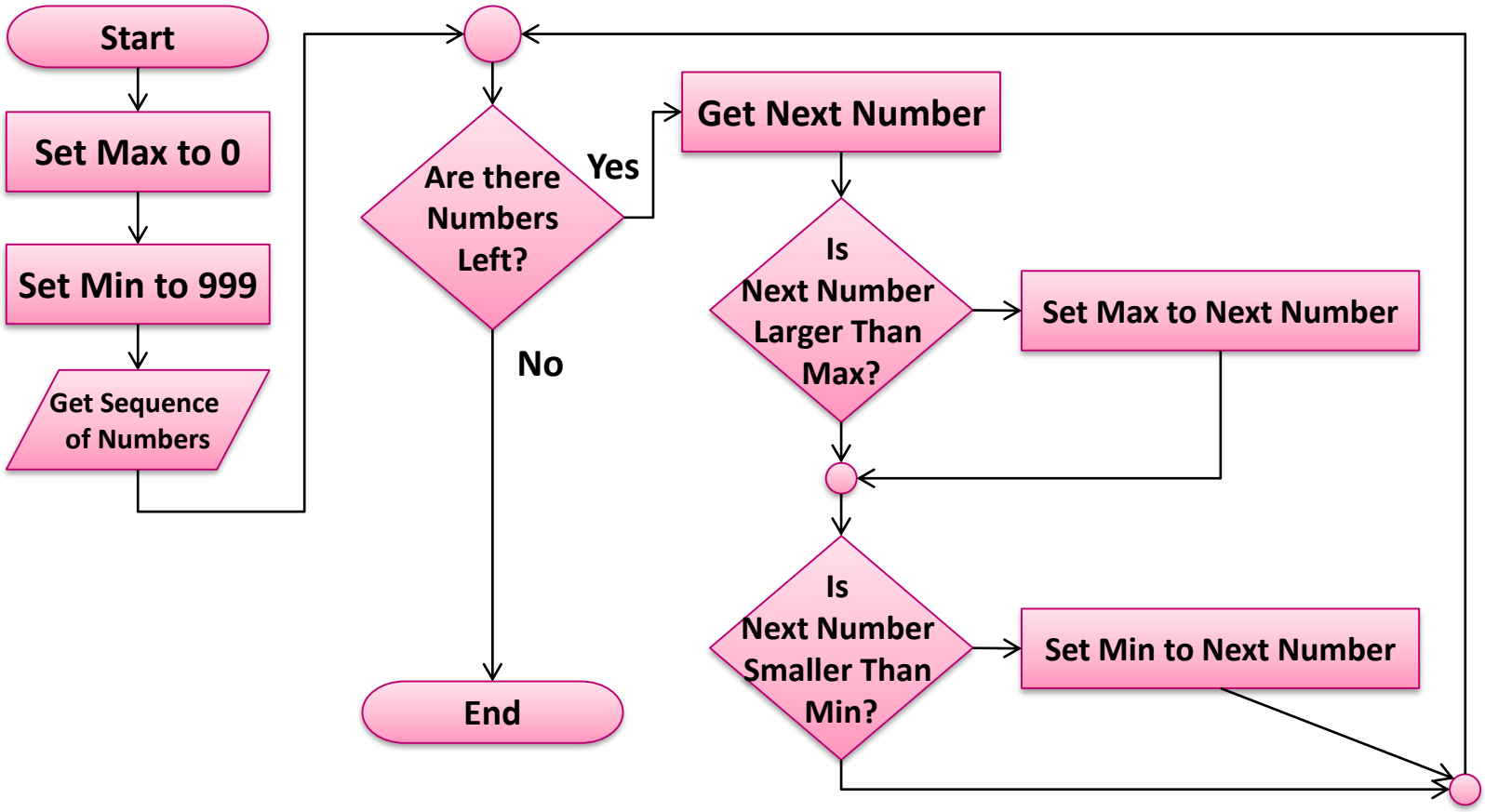
  <iteration-statement> = while (<expression>) <statement>
  | do <statement> while (<expression>);
  | for ([<expression>] ; [<expression>]; [<expression>]) <statement>

- When choosing a loop, consider what exactly you need

Range Algorithm Example

University of Reading

If you recall our range algorithm, you can see that there is a loop



Controlling the Iteration

University of Reading

- A loop is a sequence of statements executed repeatedly
  - Keep trying for a solution
  - Run the same operation N times
- A condition determines when to stop "looping"
  - For loop: initialises and iterates over a condition
    - A control variable may be used to count repetitions
  - While loop: sentinel controlled repetition
    - Indefinite repetition
    - Used when the number of repetitions is not known
    - The sentinel value indicates when to stop looping
    - Manual abort using the break statement

While Loop

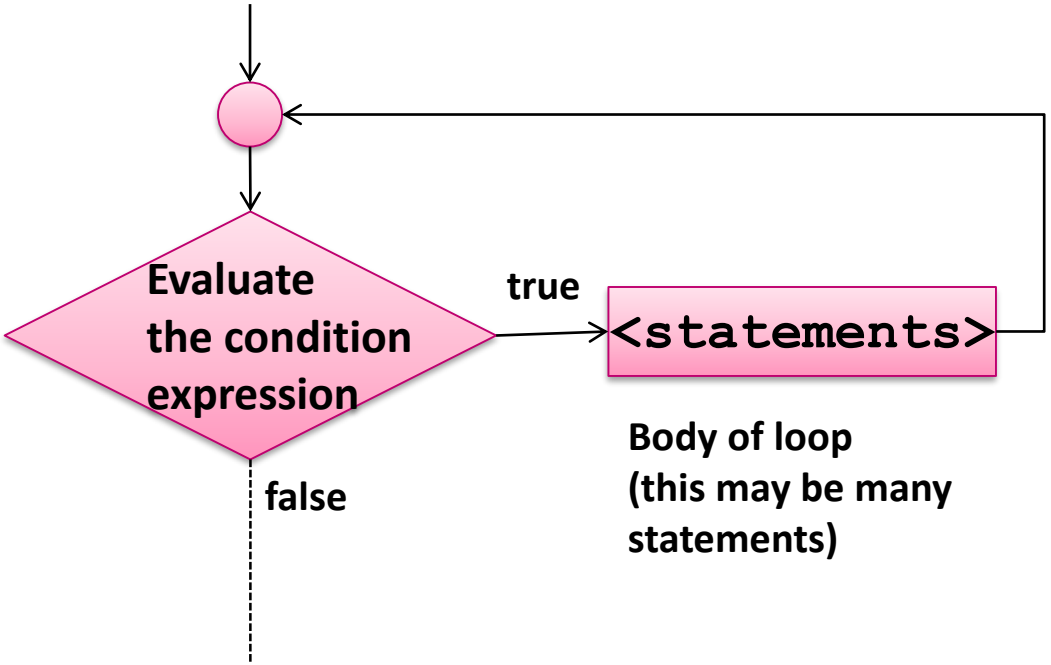
University of Reading

- The while loop is a sentinel-controlled repetition loop
- The program keeps looping while a condition is true
- Key Points:
  - Condition is checked before the loop begins
  - Statements inside NOT executed if condition is false to start with

```
while( condition ){
        <statements> // Execute statements
}
```

While Loop Flowchart

University of Reading



While Loop

University of Reading

```
#include <stdio.h>
int main(){
  int counter = 1;
  while(counter <= 10){
        printf("%d\n", counter);
        counter++;
  }
  return 0;
}
```

- The condition needs to be a function of the code in the loop
  - Otherwise, you'll end up with an infinite loop

While Loop

University of Reading

- A sentinel may be part of the loop code
  - or a failure condition:

```
#include <stdio.h>
int main()
{
  int ALRIGHT = 1;
  int work_result = ALRIGHT;
  while( work_result == ALRIGHT ){
    // do some work as long as there is no error
    work_result = // something …
  }
  return 0;
}
```

- Remember: a useful algorithm terminates for all inputs

Do-While Loop

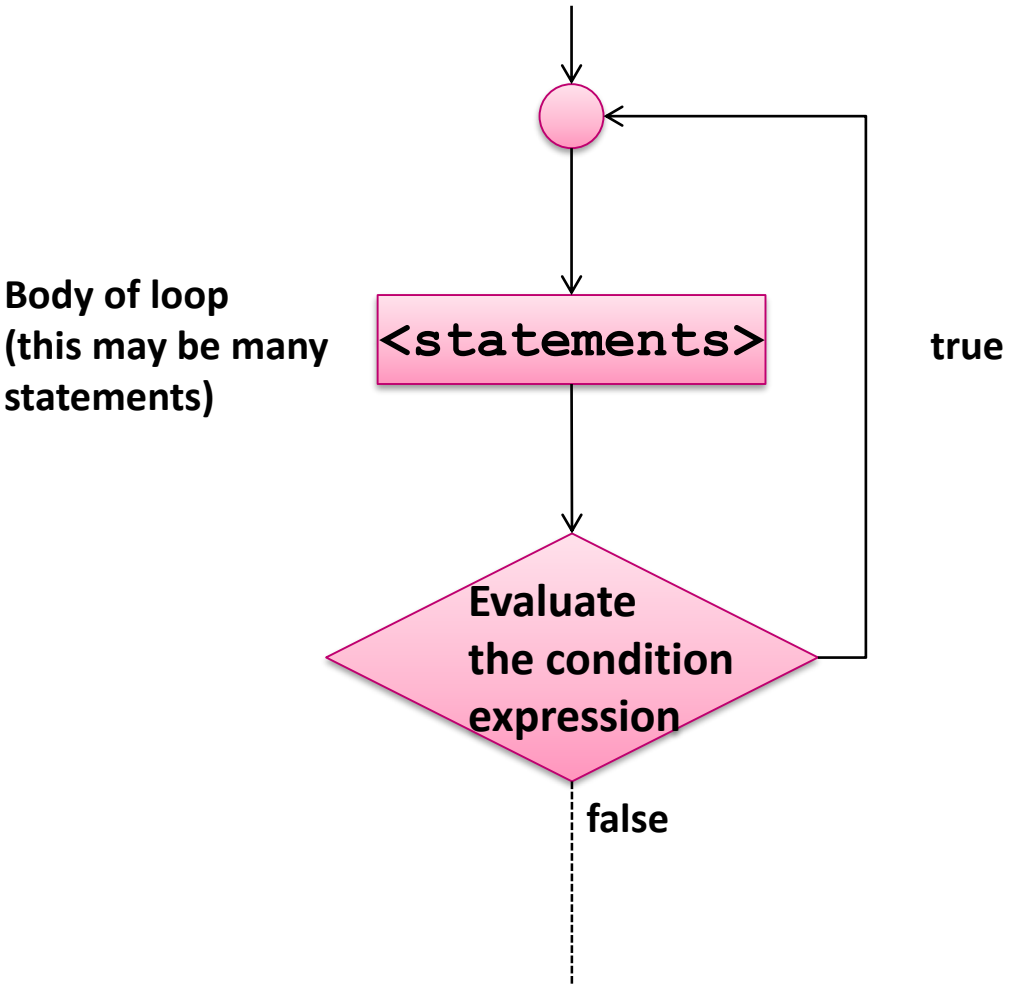
University of Reading

- The do-while is like the while loop with one difference:
  - Condition is checked at the end of the loop
  - Statements are always executed at least once

```
do{
   // Statements here
} while ( condition ); // Note the semicolon
```

Do-While Loop Flowchart

University of Reading



For Loop

University of Reading

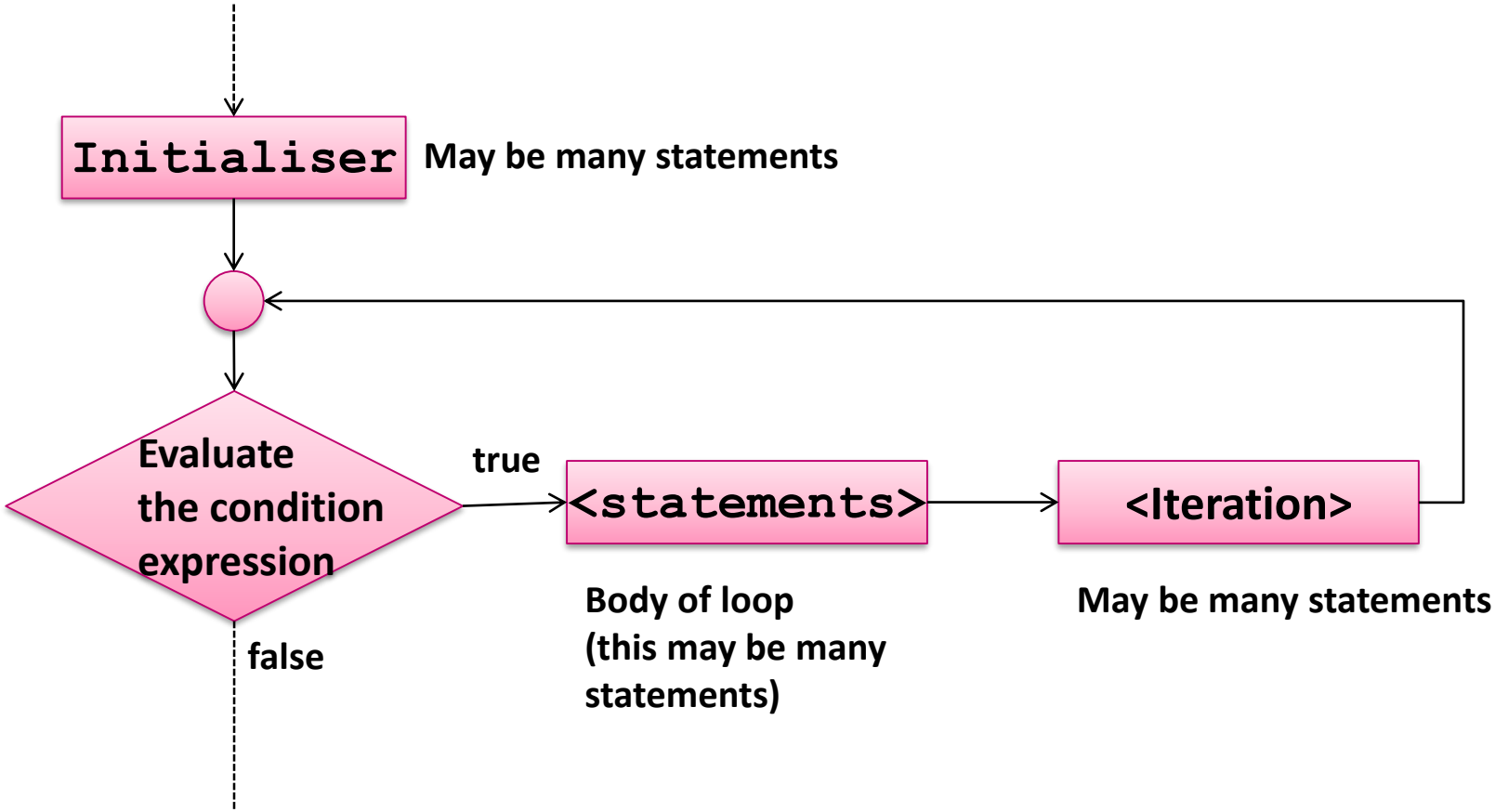
- The for loop is optimal for counting or "iterating"
  - It allows a group of instructions to be executed a given number of times, e.g. summing the numbers in an array
- Key Points:
  - Has an innate ability to handle counters

```
for( <initialiser> ; <condition> ; <iteration>){
    <statements>
}
```

  - If the loop condition is initially FALSE, then the body is not executed
  - Initialiser, condition and iteration can be multiple statements
    - Separated by " ; "

For Loop Flowchart

University of Reading



For Loop: Example Iteration

University of Reading

- Iterating over a number of items

```
int counter;
```

Counter is the control variable

Initialise counter

Counter limit

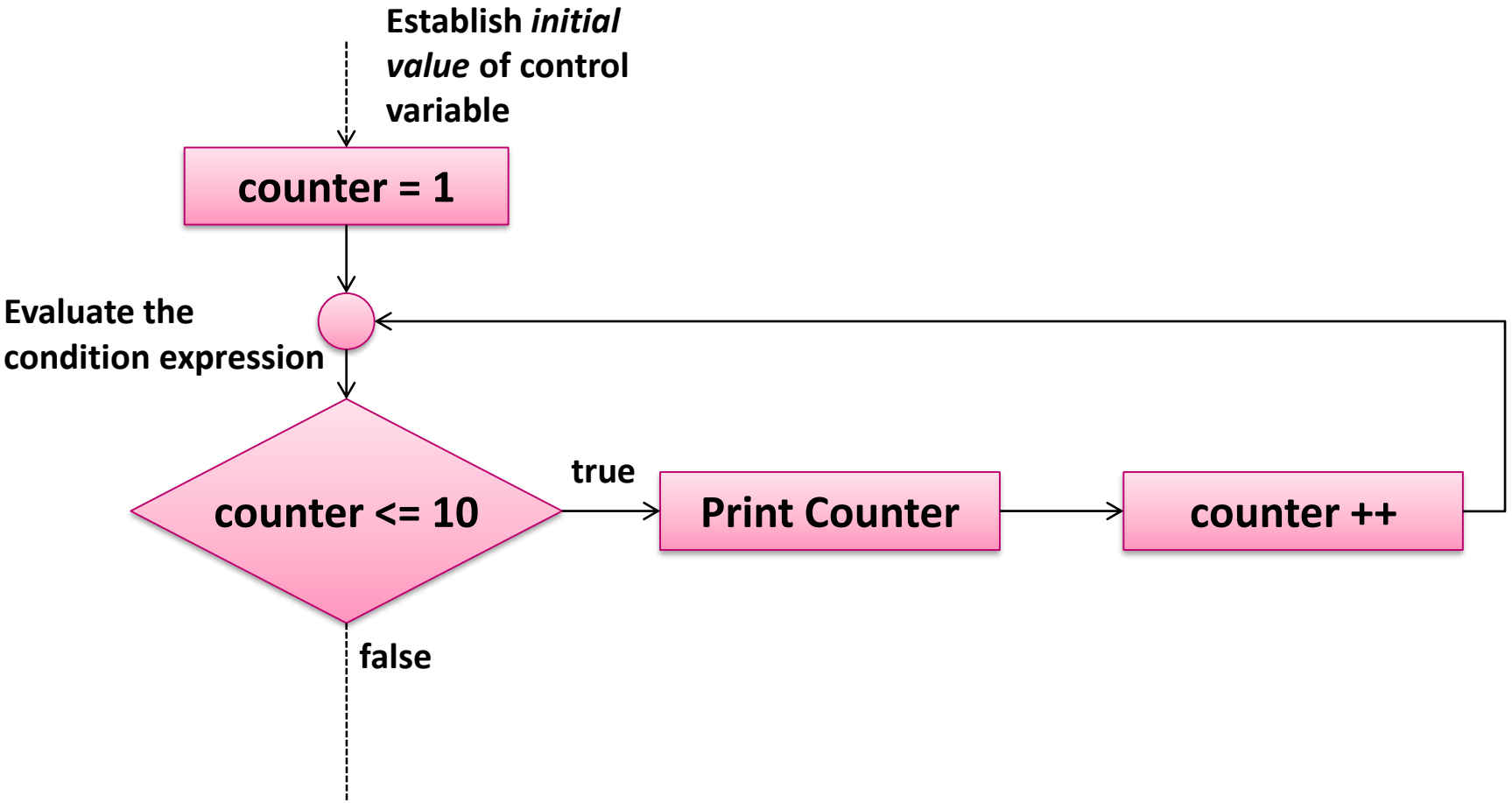
Increment command

```
for(counter=1; counter<=10; counter++){
    printf("%d\n", counter);
}
```

- What does this code do / output?

For Loop Flowchart

University of Reading



Break: to abort Loop/Switch

University of Reading

- The break command causes an immediate exit from
  - Loop: while, for, do-while
  - Switch statement
- Execution continues with the first statement after (loop/switch)
- Common uses of the break statement
  - Escape early from a loop
  - Skip the remainder of a switch statement

Break

University of Reading

```
for( count = 0; count < 10; ++count )
{
        if( something happened ){
                break;
        }
}
```

Continue: go on with the next!

University of Reading

- The continue statement skips the remaining statements in the body of a while, for or do-while statement
- Proceeds with the next iteration of the loop
- while and do-while loop
  - Loop-continuation test is evaluated immediately after the continue statement is executed
- for loop
  - Increment expression is executed, then the loop-continuation test is evaluated

Continue

University of Reading

```
do{
    read a number from the user into var number
    if ( number < 0 ){
        printf("invalid number\n");
        continue;
    }
   // this code here is skipped when the if is true

    total += number;
} while( number != sentinel );
```

Summary: A (Useful Program)

University of Reading

- What does this program do?

```
#include <stdio.h>

int main()
{
    int N=10;
    int number = 1;
    while(N > 0){

            if ( is_prime(number) ){
                printf("Prime: %d\n", number);
                N = N - 1;
            }
            number++;
    }
    return 0;
}
```

Summary: A (Useful Program)

University of Reading

- Output the first N prime numbers

```
#include <stdio.h>

int main()
{
    int N = 10;
    int number = 1;
    while(N > 0){

            if ( is_prime(number) ){
                printf("Prime: %d\n", number);
                N = N - 1;
            }
            number++;
    }
    return 0;
}
```

Assume we know how to check for prime

- A more efficient algorithm would be the
  - Sieve of Eratosthenes

Summary

University of Reading

- Syntax: "What is written"
- Semantics: "What does it mean"
- The EBNF allows to formalising the specification of a "language"
- Workflow: Write the program (ASCII file), compile, run
- A basic C-Program
  - Uses the principles of sequence, selection, iteration
  - There are various types of expressions
  - You must understand the syntax and semantics of the language!
- Basically, you know all the ingredients to implement algorithms!
  - All you need is practice!