

# Programming in C

Dr. Neill Campbell  
Neill.Campbell@bristol.ac.uk

University of Bristol

September 24, 2021



# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

# About the Course

These course notes were originally based on :

## **C By Dissection (3rd edition)**

*Al Kelley and Ira Pohl*

because I liked arrays being taught late(r). I've since changed my mind a little & have re-jigged the notes quite heavily for this year.

# Resources

- Free : [https://en.wikibooks.org/wiki/C\\_Programming](https://en.wikibooks.org/wiki/C_Programming)

# Resources

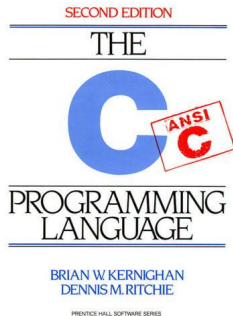
- Free : [https://en.wikibooks.org/wiki/C\\_Programming](https://en.wikibooks.org/wiki/C_Programming)
- A list of more : <https://www.linuxlinks.com/excellent-free-books-learn-c/>

# Resources

- Free : [https://en.wikibooks.org/wiki/C\\_Programming](https://en.wikibooks.org/wiki/C_Programming)
- A list of more : <https://www.linuxlinks.com/excellent-free-books-learn-c/>
- Whatever you use, make sure it's **C99** that's being taught, not something else e.g. C11 or C++.

# Resources

- Free : [https://en.wikibooks.org/wiki/C\\_Programming](https://en.wikibooks.org/wiki/C_Programming)
- A list of more : <https://www.linuxlinks.com/excellent-free-books-learn-c/>
- Whatever you use, make sure it's **C99** that's being taught, not something else e.g. C11 or C++.
- If you fall in love with C and know you're going to use it for the rest of your life, the reference 'bible' is K&R 2nd edition. It's not a textbook for those new to programming, though.



- Talk to your friends, ask for help, work together.



# Computer Science Ethos

- Talk to your friends, ask for help, work together.
- Never pass off another persons work as your own.

# Computer Science Ethos

- Talk to your friends, ask for help, work together.
- Never pass off another persons work as your own.
- Do not pass work to others - either on paper or electronically - even after the submission deadline.

- Talk to your friends, ask for help, work together.
- Never pass off another persons work as your own.
- Do not pass work to others - either on paper or electronically - even after the submission deadline.
- If someone takes your code and submits it, we need to investigate where it originated - all students involved will be part of this.

- Talk to your friends, ask for help, work together.
- Never pass off another persons work as your own.
- Do not pass work to others - either on paper or electronically - even after the submission deadline.
- If someone takes your code and submits it, we need to investigate where it originated - all students involved will be part of this.
- Don't place your code on publicly accessible sites e.g. github - other students may have extensions etc.

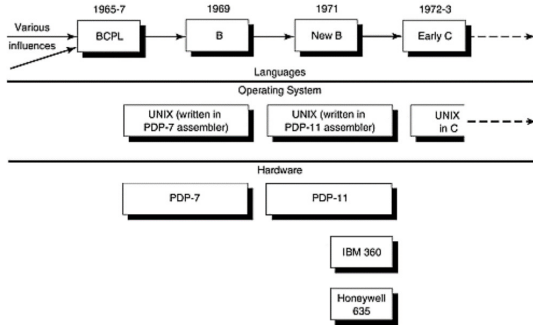
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards

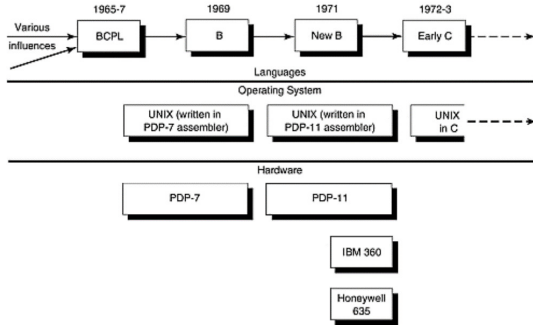
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970

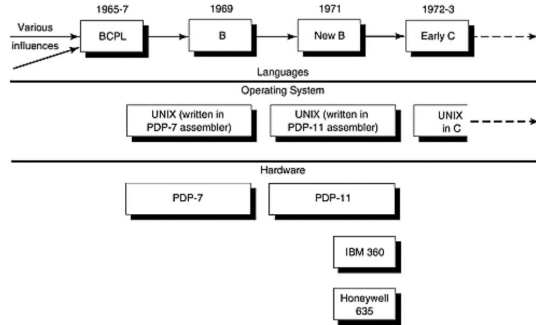
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.

# History of C

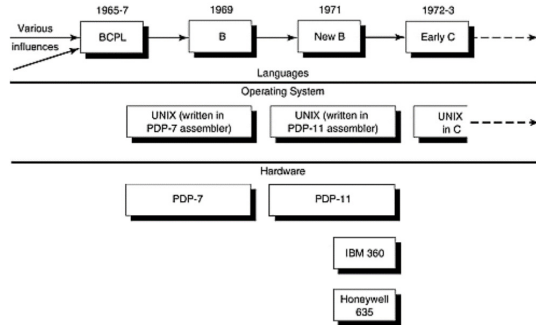


From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.



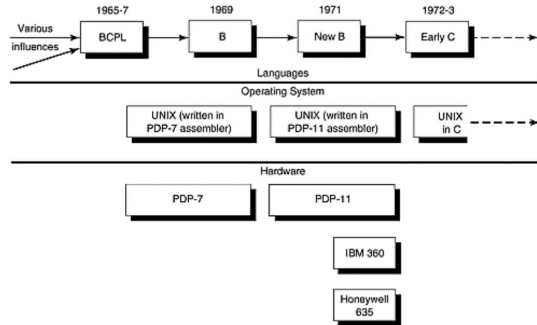
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978

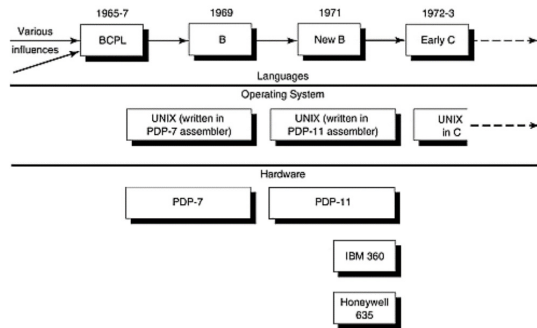
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978
- ANSI C

# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978
- ANSI C
- C99 (COMSM1201)

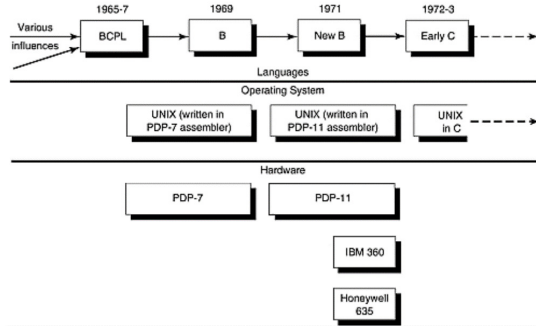
# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978
- ANSI C
- C99 (COMSM1201)
- C++ - Object Oriented Programming (OOP)







# History of C



From *Deep C Secrets* by Peter Van Der Linden

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978
- ANSI C
- C99 (COMSM1201)
- C++ - Object Oriented Programming (OOP)
- Java (Subset of C++, WWW enabled).








# Why C ?

Jun 2021	Jun 2020	Change	Programming Language
1	1		 C
2	3	▲	 Python
3	2	▼	 Java
4	4		 C++
5	5		 C#
6	6		 Visual Basic
7	7		 JavaScript

<https://www.tiobe.com/tiobe-index/>

- One of the most commonly used programming languages according to tiobe.com








# Why C ?

Jun 2021	Jun 2020	Change	Programming Language
1	1		 C
2	3	▲	 Python
3	2	▼	 Java
4	4		 C++
5	5		 C#
6	6		 Visual Basic
7	7		 JavaScript

<https://www.tiobe.com/tiobe-index/>

- One of the most commonly used programming languages according to tiobe.com
- Low-level (c.f. Java)

# Why C ?








Jun 2021	Jun 2020	Change	Programming Language	
1	1			C
2	3	▲		Python
3	2	▼		Java
4	4			C++
5	5			C#
6	6			Visual Basic
7	7			JavaScript

<https://www.tiobe.com/tiobe-index/>

- One of the most commonly used programming languages according to tiobe.com
- Low-level (c.f. Java)
- Doesn't hide nitty-gritty










# Why C ?

Jun 2021	Jun 2020	Change	Programming Language
1	1		 C
2	3	▲	 Python
3	2	▼	 Java
4	4		 C++
5	5		 C#
6	6		 Visual Basic
7	7		 JavaScript

<https://www.tiobe.com/tiobe-index/>

- One of the most commonly used programming languages according to tiobe.com
- Low-level (c.f. Java)
- Doesn't hide nitty-gritty
- Fast ?

# Why C ?

Jun 2021	Jun 2020	Change	Programming Language
1	1		 C
2	3	▲	 Python
3	2	▼	 Java
4	4		 C++
5	5		 C#
6	6		 Visual Basic
7	7		 JavaScript

<https://www.tiobe.com/tiobe-index/>

- One of the most commonly used programming languages according to tiobe.com
- Low-level (c.f. Java)
- Doesn't hide nitty-gritty
- Fast ?
- Large parts common to Java

- Was traditionally Lectured 2(or 3) hours a week for weeks 1-12

- Was traditionally Lectured 2(or 3) hours a week for weeks 1-12
- In the blended world, I'll post the equivalent online, broken into manageable chunks

# Programming and Software Engineering

- Was traditionally Lectured 2(or 3) hours a week for weeks 1-12
- In the blended world, I'll post the equivalent online, broken into manageable chunks
- Programming (C), data structures, algorithms - searching, sorting, string processing, trees etc.

- Weekly (unmarked) exercises that, if completed, should ensure you are able to pass the unit.

- Weekly (unmarked) exercises that, if completed, should ensure you are able to pass the unit.
- Approximately three/four assignments and one lab test.

- Weekly (unmarked) exercises that, if completed, should ensure you are able to pass the unit.
- Approximately three/four assignments and one lab test.
- One major project due in early TB2 (35%).



- Weekly (unmarked) exercises that, if completed, should ensure you are able to pass the unit.
- Approximately three/four assignments and one lab test.
- One major project due in early TB2 (35%).
- Hard to gauge timings, so don't make any plans in advance - I'll change it if we're going too fast.

# Help with Computers

- Any problems with the computers e.g. installing the correct S/W, accessing lab machines :  
<http://www.bris.ac.uk/it-services/>

# Help with Computers

- Any problems with the computers e.g. installing the correct S/W, accessing lab machines :  
<http://www.bris.ac.uk/it-services/>
- They are also the people to see about passwords etc.

# Help with Computers

- Any problems with the computers e.g. installing the correct S/W, accessing lab machines : <http://www.bris.ac.uk/it-services/>
- They are also the people to see about passwords etc.
- This page also links to the rather useful Laptop & Mobile Clinic.

# Help with the Unit

- All information is available via the Blackboard site (which will point you to other sites including [github.com](https://github.com), MS Streams, MS Teams etc.)

# Help with the Unit

- All information is available via the Blackboard site (which will point you to other sites including [github.com](https://github.com), MS Streams, MS Teams etc.)
- Online will mainly be via myself giving 'live' Q&A session, the associated MS Teams group with Forum, and Teaching Assistants in our on-campus / face-to-face labs.

# Help with the Unit

- All information is available via the Blackboard site (which will point you to other sites including [github.com](https://github.com), MS Streams, MS Teams etc.)
- Online will mainly be via myself giving 'live' Q&A session, the associated MS Teams group with Forum, and Teaching Assistants in our on-campus / face-to-face labs.
- TAs are not allowed to write pieces of code for you, nor undertake detailed bug-fixing of your program.

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

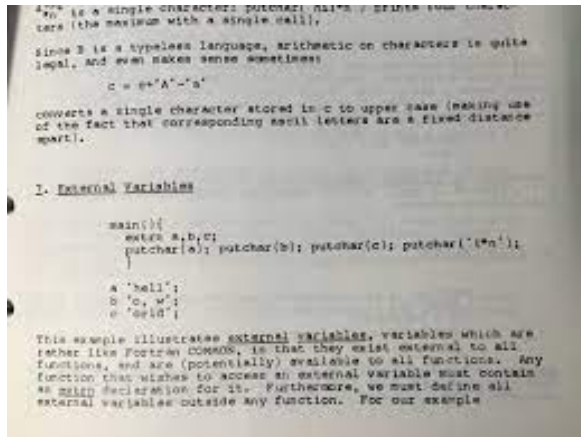
G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

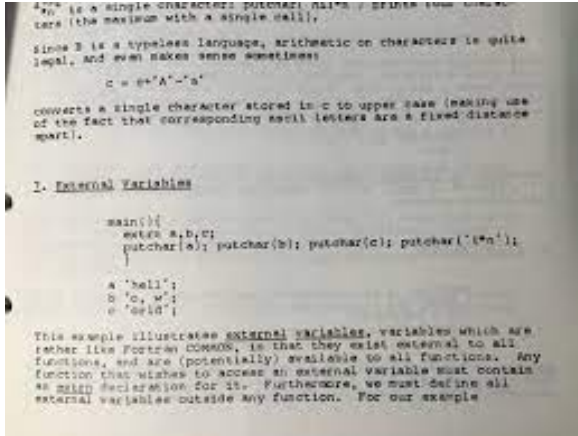


# Hello World!



Hello World first seen in: Brian Kernighan, *A Tutorial Introduction to the Language B*, 1972

# Hello World!



Hello World first seen in: Brian Kernighan, *A Tutorial Introduction to the Language B*, 1972

```
1  /* The traditional first program
2  in honour of Dennis Ritchie
3  who invented C at Bell Labs
4  in 1972 */
5
6  #include <stdio.h>
7
8  int main(void)
9  {
10
11     printf("Hello , world!\n");
12     return 0;
13
14 }
```

Execution :

Hello , world!

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`
- Statements are grouped using braces,  
`{ ... }`

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`
- Statements are grouped using braces,  
`{ ... }`
- `printf()` One of the pre-defined library functions being called (invoked) using a single argument the string :  
`"Hello, world!\n"`

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`
- Statements are grouped using braces,  
`{ ... }`
- `printf()` One of the pre-defined library functions being called (invoked) using a single argument the string :  
`"Hello, world!\n"`
- The `\n` means print the single character *newline*.



# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`
- Statements are grouped using braces,  
`{ ... }`
- `printf()` One of the pre-defined library functions being called (invoked) using a single argument the string :  
`"Hello, world!\n"`
- The `\n` means print the single character *newline*.
- Notice all declarations and statements are terminated with a semi-colon.

# Dissecting the 1st Program

- Comments are bracketed by the `/*` and `*/` pair.
- `#include <stdio.h>`  
Lines that begin with a `#` are called preprocessing directives.
- `int main(void)`  
Every program has a function called `main()`
- Statements are grouped using braces,  
`{ ... }`
- `printf()` One of the pre-defined library functions being called (invoked) using a single argument the string :  
`"Hello, world!\n"`
- The `\n` means print the single character *newline*.
- Notice all declarations and statements are terminated with a semi-colon.
- `return(0)` Instruct the Operating System that the function `main()` has completed successfully.

# Area of a Rectangle

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      // Compute the area of a rectangle
6      int side1, side2, area;
7      side1 = 7;
8      side2 = 8;
9      area = side1 * side2;
10
11     printf("Length of side 1 = %d metres\n", side1);
12     printf("Length of side 2 = %d metres\n", side2);
13     printf("Area of rectangle = %d metres squared\n", area);
14     return 0;
15 }
```

Execution :

```
Length of side 1 = 7 metres
Length of side 2 = 8 metres
Area of rectangle = 56 metres squared
```

# Dissecting the Area Program

- `//` One line comment.

# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.

# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.
- `int side1, side2, area;` *Declaration*

# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.
- `int side1, side2, area;` *Declaration*
- `side2 = 8;` *Assignment*

# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.
- `int side1, side2, area;` *Declaration*
- `side2 = 8;` *Assignment*
- `printf()` has 2 Arguments. The *control string* contains a `%d` to indicate an integer is to be printed.



# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.
- `int side1, side2, area;` *Declaration*
- `side2 = 8;` *Assignment*
- `printf()` has 2 Arguments. The *control string* contains a `%d` to indicate an integer is to be printed.

# Dissecting the Area Program

- `//` One line comment.
- `#include <stdio.h>` Always required when using I/O.
- `int side1, side2, area;` *Declaration*
- `side2 = 8;` *Assignment*
- `printf()` has 2 Arguments. The *control string* contains a `%d` to indicate an integer is to be printed.

```
1  preprocessing directives
2
3  int main(void)
4  {
5      declarations
6
7      statements
8  }
```

# Arithmetic Operators

- `+`, `-`, `/`, `*`, `%`

# Arithmetic Operators

- $+$  ,  $-$  ,  $/$  ,  $*$  ,  $\%$
- Addition, Subtraction, Division, Multiplication, Modulus.

# Arithmetic Operators

- $+$  ,  $-$  ,  $/$  ,  $*$  ,  $\%$
- Addition, Subtraction, Division, Multiplication, Modulus.
- Integer arithmetic discards remainder i.e.  
 $1/2$  is 0 ,  $7/2$  is 3.

# Arithmetic Operators

- $+$  ,  $-$  ,  $/$  ,  $*$  ,  $\%$
- Addition, Subtraction, Division, Multiplication, Modulus.
- Integer arithmetic discards remainder i.e.  
 $1/2$  is 0 ,  $7/2$  is 3.
- Modulus (Remainder) Arithmetic.  
 $7\%4$  is 3,  $12\%6$  is 0.

# Arithmetic Operators

- $+$  ,  $-$  ,  $/$  ,  $*$  ,  $\%$
- Addition, Subtraction, Division, Multiplication, Modulus.
- Integer arithmetic discards remainder i.e.  
 $1/2$  is 0 ,  $7/2$  is 3.
- Modulus (Remainder) Arithmetic.  
 $7\%4$  is 3,  $12\%6$  is 0.
- Only available for integer arithmetic.

# The Character Type

```
1 // Demonstration of character arithmetic
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char c;
7
8     c = 'A';
9     printf("%c ", c);
10    printf("%c\n", c+1);
11    return 0;
12 }
```

Execution :

A B



# The Character Type

```
1 // Demonstration of character arithmetic
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char    c;
7
8     c = 'A';
9     printf("%c ", c);
10    printf("%c\n", c+1);
11    return 0;
12 }
```

- The keyword char stands for character.

Execution :

A B

# The Character Type

```
1 // Demonstration of character arithmetic
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char    c;
7
8     c = 'A';
9     printf("%c ", c);
10    printf("%c\n", c+1);
11    return 0;
12 }
```

Execution :

A B

- The keyword char stands for character.
- Used with single quotes i.e. 'A', or '+'.

# The Character Type

```
1 // Demonstration of character arithmetic
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char    c;
7
8     c = 'A';
9     printf("%c ", c);
10    printf("%c\n", c+1);
11    return 0;
12 }
```

Execution :

A B

- The keyword `char` stands for character.
- Used with single quotes i.e. `'A'`, or `'+'`.
- Some keyboards have a second single quote the **back quote** ```

# The Character Type

```
1 // Demonstration of character arithmetic
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char    c;
7
8     c = 'A';
9     printf("%c ", c);
10    printf("%c\n", c+1);
11    return 0;
12 }
```

Execution :

A B

- The keyword `char` stands for character.
- Used with single quotes i.e. `'A'`, or `'+'`.
- Some keyboards have a second single quote the **back quote** ```
- Note the `%c` conversion format.

# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

- In C there are three common floating types :

# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

- In C there are three common floating types :

- float

# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

● In C there are three common floating types :

- 1 float
- 2 double



# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

• In C there are three common floating types :

- 1 float
- 2 double
- 3 long double

# Floating Types

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      double x, y;
7
8      x = 1.0;
9      y = 2.0;
10
11     printf("Sum of x & y is %f.\n", x + y);
12
13     return 0;
14
15 }
```

Execution :

Sum of x & y is 3.000000.

- In C there are three common floating types :
  - 1 float
  - 2 double
  - 3 long double
- The *Working Type* is doubles.

# The Preprocessor

- A `#` in the first column signifies a preprocessor statement.

# The Preprocessor

- A `#` in the first column signifies a preprocessor statement.
- `#include <file.h>` Exchange this line for the entire contents of `file.h`, which is to be found in a standard place.

# The Preprocessor

- A `#` in the first column signifies a preprocessor statement.
- `#include <file.h>` Exchange this line for the entire contents of `file.h`, which is to be found in a standard place.
- `#define PI 3.14159265358979` Replaces all occurrences of `PI` with `3.14159265358979`.

# The Preprocessor

- A `#` in the first column signifies a preprocessor statement.
- `#include <file.h>` Exchange this line for the entire contents of `file.h`, which is to be found in a standard place.
- `#define PI 3.14159265358979` Replaces all occurrences of `PI` with `3.14159265358979`.
- Include files generally contain other `#define`'s and `#include`'s (amongst other things).

# Using printf()

- `printf( fmt-str, arg1, arg2, ...);`

<code>%c</code>	Characters
<code>%d</code>	Integers
<code>%e</code>	Floats/Doubles (Engineering Notation)
<code>%f</code>	Floats/Doubles
<code>%s</code>	Strings

# Using printf()

- `printf( fmt-str, arg1, arg2, ...);`

<code>%c</code>	Characters
<code>%d</code>	Integers
<code>%e</code>	Floats/Doubles (Engineering Notation)
<code>%f</code>	Floats/Doubles
<code>%s</code>	Strings

- Fixed-width fields: `printf("F:%7f\n", f);`

F: 3.0001



# Using printf()

- `printf( fmt-str, arg1, arg2, ...);`

<code>%c</code>	Characters
<code>%d</code>	Integers
<code>%e</code>	Floats/Doubles (Engineering Notation)
<code>%f</code>	Floats/Doubles
<code>%s</code>	Strings

- Fixed-width fields: `printf("F:%7f\n", f);`  
F: 3.0001
- Fixed Precision: `printf("F:%.2f\n", f);`  
F:3.00

# Using scanf()

- Similar to printf() but deals with *input* rather than *output*.

%c	Characters
%d	Integers
%f	Floats
%lf	Doubles
%s	Strings

# Using scanf()

- Similar to printf() but deals with *input* rather than *output*.
- `scanf(fmt-str, &arg1, &arg2, ...);`

%c	Characters
%d	Integers
%f	Floats
%lf	Doubles
%s	Strings

# Using scanf()

- Similar to printf() but deals with *input* rather than *output*.
- `scanf(fmt-str, &arg1, &arg2, ...);`
- Note that the *address* of the argument is required.

%c	Characters
%d	Integers
%f	Floats
%lf	Doubles
%s	Strings

# Using scanf()

- Similar to printf() but deals with *input* rather than *output*.
- `scanf(fmt-str, &arg1, &arg2, ...);`
- Note that the *address* of the argument is required.

%c	Characters
%d	Integers
%f	Floats
%lf	Doubles
%s	Strings

- Note doubles handled differently than floats.

# While Loops

```
while (test is true) {  
    statement 1;  
    ...  
    statement n;  
}
```

# While Loops

```
while (test is true) {  
    statement 1;  
    ...  
    statement n;  
}
```

```
1  // Sums are computed.  
2  #include <stdio.h>  
3  
4  int main(void)  
5  {  
6  
7      int cnt = 0;  
8      float sum = 0.0, x;  
9      printf("Input some numbers: ");  
10     while (scanf("%f", &x) == 1) {  
11         cnt = cnt + 1;  
12         sum = sum + x;  
13     }  
14  
15     printf("\n%s%5d\n%s%5f\n\n",  
16           "Count:", cnt, "Sum: ", sum);  
17     return 0;  
18 }
```

Execution :

Input some numbers: 1 5 9 10

Count: 4

Sum: 25.000000

# Common Mistakes

- Missing "

```
printf("%c\n", ch);
```



# Common Mistakes

- Missing "

```
printf("%c\n", ch);
```

- Missing ;

```
a = a + 1
```

# Common Mistakes

- Missing "

```
printf("%c\n, ch);
```

- Missing ;

```
a = a + 1
```

- Missing Address in scanf()

```
scanf("%d", a);
```

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

- C has a grammar/syntax like every other language.

- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.

- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.
- Valid Identifiers :  
k, `_id`, `iamanidentifier2`, `so_am_i`.

- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.
- Valid Identifiers :  
k, `_id`, `iamanidentifier2`, `so_am_i`.
- **Invalid** Identifiers :  
`not#me`, `101_south`, `-plus`.

- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.
- Valid Identifiers :  
k, \_\_id, iamanidentifier2, so\_\_am\_\_i.
- **Invalid** Identifiers :  
not#me, 101\_\_south, -plus.
- Constants :  
17 (decimal), 017 (octal), 0x17 (hexadecimal).



- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.
- Valid Identifiers :  
k, `_id`, `iamanidentifier2`, `so_am_i`.
- **Invalid** Identifiers :  
`not#me`, `101_south`, `-plus`.
- Constants :  
17 (decimal), 017 (octal), 0x17 (hexadecimal).
- String Constant enclosed in double-quotes :  
"I am a string"

# Operators

- All operators have rules of both *precedence* and *associativity*.

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .
- The associativity of  $+$  is left-to-right, thus  $1 + 2 + 3$  is equivalent to  $(1 + 2) + 3$ .

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .
- The associativity of  $+$  is left-to-right, thus  $1 + 2 + 3$  is equivalent to  $(1 + 2) + 3$ .
- Increment and decrement operators :  
 $i++$ ; is equivalent to  $i = i + 1$ ;

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .
- The associativity of  $+$  is left-to-right, thus  $1 + 2 + 3$  is equivalent to  $(1 + 2) + 3$ .
- Increment and decrement operators :  
 $i++$ ; is equivalent to  $i = i + 1$ ;
- May also be prefixed  $--i$ ;

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .
- The associativity of  $+$  is left-to-right, thus  $1 + 2 + 3$  is equivalent to  $(1 + 2) + 3$ .
- Increment and decrement operators :  
 $i++$ ; is equivalent to  $i = i + 1$ ;
- May also be prefixed  $--i$ ;

# Operators

- All operators have rules of both *precedence* and *associativity*.
- $1 + 2 * 3$  is the same as  $1 + (2 * 3)$  because  $*$  has a higher precedence than  $+$ .
- The associativity of  $+$  is left-to-right, thus  $1 + 2 + 3$  is equivalent to  $(1 + 2) + 3$ .
- Increment and decrement operators :  
 $i++$ ; is equivalent to  $i = i + 1$ ;
- May also be prefixed  $--i$ ;

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, c = 0;
6      a = ++c;
7      int b = c++;
8      printf("%d %d %d\n", a, b, ++c);
9      return 0;
10 }
```

Question : What is the output ?



# Assignment

- The `=` operator has a low precedence and a right-to-left associativity.

# Assignment

- The `=` operator has a low precedence and a right-to-left associativity.
- `a = b = c = 0;` is valid and equivalent to :  
    `= (b = (c = 0));`

# Assignment

- The `=` operator has a low precedence and a right-to-left associativity.
- `a = b = c = 0;` is valid and equivalent to :  
    `= (b = (c = 0));`
- `i = i + 3;` is the same as `i += 3;`

# Assignment

- The `=` operator has a low precedence and a right-to-left associativity.
- `a = b = c = 0;` is valid and equivalent to :  
`= (b = (c = 0));`
- `i = i + 3;` is the same as `i += 3;`
- Many other operators are possible e.g.  
`-=`, `*=`, `/=`.

# Assignment

- The `=` operator has a low precedence and a right-to-left associativity.
- `a = b = c = 0;` is valid and equivalent to :  
    `= (b = (c = 0));`
- `i = i + 3;` is the same as `i += 3;`
- Many other operators are possible e.g.  
    `-=`, `*=`, `/=`.

# Assignment

- The = operator has a low precedence and a right-to-left associativity.
- `a = b = c = 0;` is valid and equivalent to :  
    `= (b = (c = 0));`
- `i = i + 3;` is the same as `i += 3;`
- Many other operators are possible e.g.  
    `-=`, `*=`, `/=`.

```
1 // 1st few powers of 2 are printed.
2
3 #include <stdio.h>
4
5 int main(void)
6 {
7     int i = 0, power = 1;
8
9     while (++i <= 10){
10         printf("%5d", power *= 2);
11     }
12     printf("\n");
13     return 0;
14 }
```

Execution :

```
      2      4      8      16      32      64      128      256
512 1024
```

# The Standard Library

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int i, n;
7      printf("Randomly distributed integers are printed.\n"
8             "How many do you want to see? ");
9      do{
10         i = scanf("%d", &n);
11     }while(i != 1);
12     for (i = 0; i < n; ++i) {
13         if (i % 4 == 0)
14             printf("\n");
15         printf("%12d", rand());
16     }
17     printf("\n");
18     return 0;
19 }
```

Execution :

Randomly distributed integers are printed.  
How many do you want to see? 11

1804289383	846930886	1681692777	1714636915
1957747793	424238335	719885386	1649760492
596516649	1189641421	1025202362	

# The Standard Library

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int i, n;
7      printf("Randomly distributed integers are printed.\n"
8             "How many do you want to see? ");
9      do{
10         i = scanf("%d", &n);
11     }while(i != 1);
12     for (i = 0; i < n; ++i) {
13         if (i % 4 == 0)
14             printf("\n");
15         printf("%12d", rand());
16     }
17     printf("\n");
18     return 0;
19 }
```

- Definitions required for the proper use of many functions such as `rand()` are found in `stdlib.h`.

Execution :

```
Randomly distributed integers are printed.
How many do you want to see? 11
```

```
1804289383    846930886    1681692777    1714636915
1957747793    424238335     719885386    1649760492
 596516649   1189641421   1025202362
```



# The Standard Library

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int i, n;
7      printf("Randomly distributed integers are printed.\n"
8             "How many do you want to see? ");
9      do{
10         i = scanf("%d", &n);
11     }while(i != 1);
12     for (i = 0; i < n; ++i) {
13         if (i % 4 == 0)
14             printf("\n");
15         printf("%12d", rand());
16     }
17     printf("\n");
18     return 0;
19 }
```

Execution :

Randomly distributed integers are printed.  
How many do you want to see? 11

1804289383	846930886	1681692777	1714636915
1957747793	424238335	719885386	1649760492
596516649	1189641421	1025202362	

- Definitions required for the proper use of many functions such as `rand()` are found in `stdlib.h`.
- Do not mistake these header files for the libraries themselves !

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

**D: Flow Control**

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- $(a < b)$  returns the value *0* or *1*.

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- $(a < b)$  returns the value *0* or *1*.
- $(i == 5)$  is a **test** not an **assignment**.

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- $(a < b)$  returns the value *0* or *1*.
- $(i == 5)$  is a **test** not an **assignment**.
- $(!a)$  is either *true* (*1*) or *false* (*0*).



# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- $(a < b)$  returns the value *0* or *1*.
- $(i == 5)$  is a **test** not an **assignment**.
- $(!a)$  is either *true* (*1*) or *false* (*0*).
- $(a \&\& b)$  is *true* if both *a* and *b* are *true*.

# Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	<b>equal to</b>
!=	not equal to
!	not
&&	logical AND
	logical OR

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- $(a < b)$  returns the value *0* or *1*.
- $(i == 5)$  is a **test** not an **assignment**.
- $(!a)$  is either *true* (*1*) or *false* (*0*).
- $(a \&\& b)$  is *true* if both *a* and *b* are *true*.
- Single  $\&$  and  $|$  are *bitwise* operators not comparisons - more on this later.

# Short-Circuit Evaluation

```
if(x >= 0.0 && sqrt(x) < 10.0){  
    ..... /* Do Something */  
}
```

It's not possible to take the `sqrt()` of a negative number. Here, the `sqrt()` statement is never reached if the first test is *false*. In a logical AND, once any expression is *false*, the whole must be *false*.

# The if() Statement

Strictly, you don't need braces if there is only one statement as part of the if :

```
if (expr)
    statement
```

If more than one statement is required :

```
if (expr) {
    statement - 1
    .
    .
    .
    statement - n
}
```

However, we will **always** brace them, even if it's not necessary.

# The if() Statement

Strictly, you don't need braces if there is only one statement as part of the if :

```
if (expr)
    statement
```

If more than one statement is required :

```
if (expr) {
    statement - 1
    .
    .
    statement - n
}
```

However, we will **always** brace them, even if it's not necessary.

Adding an else statement :

```
if (expr) {
    statement - 1
    .
    .
    statement - n
}
else {
    statement - a
    .
    .
    statement - e
}
```

# A Practical Example of if:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int    x, y, z;
6
7      printf("Input three integers: ");
8      if (scanf("%d%d%d", &x, &y, &z) != 3){
9          printf("Didn't get 3 numbers?\n");
10         return 1;
11     }
12     int min;
13     if (x < y){
14         min = x;
15     }
16     // Nasty, dropped braces:
17     else
18         min = y;
19     if (z < min){
20         min = z;
21     }
22     printf("The minimum value is %d\n", min);
23     return 0;
24 }
```

Execution :

Input three integers: 5 7 -4  
The minimum value is -4

# The while() Statement

```
while(expr)
    statement
```

This, as with the for loop, may execute compound statements :

```
while(expr){
    statement - 1
    .
    .
    .
    statement - n
}
```

However, we will **always** brace them, even if it's not necessary.

# The while() Statement

```
while(expr)
    statement
```

This, as with the for loop, may execute compound statements :

```
while(expr){
    statement -1
    .
    .
    .
    statement -n
}
```

However, we will **always** brace them, even if it's not necessary.

```
1  // Simple while countdown
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7
8      int n = 9;
9
10     while(n > 0){
11         printf("%d ", n);
12         n--;
13     }
14     printf("\n");
15     return 0;
16 }
```

Execution :

9 8 7 6 5 4 3 2 1



# The for() Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){  
    statement -1  
    .  
    .  
    .  
    statement -n  
}
```

and may be thought of as :

```
init;  
while(test){  
    statement -1  
    .  
    .  
    .  
    statement -n  
    loop;  
}
```

# The for() Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){  
    statement -1  
    .  
    .  
    .  
    statement -n  
}
```

In the for() loop, note :

- Semi-colons separate the three parts.

and may be thought of as :

```
init;  
while(test){  
    statement -1  
    .  
    .  
    .  
    statement -n  
    loop;  
}
```

# The for() Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){  
    statement -1  
    .  
    .  
    .  
    statement -n  
}
```

and may be thought of as :

```
init;  
while(test){  
    statement -1  
    .  
    .  
    .  
    statement -n  
    loop;  
}
```

In the for() loop, note :

- Semi-colons separate the three parts.
- Any (or all) of the three parts could be empty.

# The for() Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){  
    statement -1  
    .  
    .  
    .  
    statement -n  
}
```

and may be thought of as :

```
init;  
while(test){  
    statement -1  
    .  
    .  
    .  
    statement -n  
    loop;  
}
```

In the for() loop, note :

- Semi-colons separate the three parts.
- Any (or all) of the three parts could be empty.
- If the test part is empty, it evaluates to *true*.

# The for() Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){  
    statement -1  
    .  
    .  
    .  
    statement -n  
}
```

and may be thought of as :

```
init;  
while(test){  
    statement -1  
    .  
    .  
    .  
    statement -n  
    loop;  
}
```

In the for() loop, note :

- Semi-colons separate the three parts.
- Any (or all) of the three parts could be empty.
- If the test part is empty, it evaluates to *true*.
- `for(;;){ a+=1; }` is an infinite loop.

# A Triply-Nested Loop

```
1 // Triples of integers that sum to N
2 #include <stdio.h>
3
4 #define N 7
5
6 int main(void)
7 {
8     int cnt = 0, i, j, k;
9
10    for(i = 0; i <= N; i++){
11        for(j = 0; j <= N; j++){
12            for(k = 0; k <= N; k++){
13                if(i + j + k == N){
14                    ++cnt;
15                    printf("%3d%3d%3d\n", i, j, k);
16                }
17            }
18        }
19    }
20    printf("\nCount: %d\n", cnt);
21    return 0;
22 }
```

Output :

0	0	7
0	1	6
0	2	5
0	3	4
0	4	3
0	5	2
0	6	1
0	7	0

etc.

4	3	0
5	0	2
5	1	1
5	2	0
6	0	1
6	1	0
7	0	0

Count: 36

# The Comma Operator

This has the lowest precedence of all the operators in C and associates left-to-right.

```
a = 0 , b = 1;
```

Hence, the for loop may become quite complex :

```
for(sum = 0, i = 1; i <= n; ++i){  
    sum += i;  
}
```

An equivalent, but more difficult to read expression :

```
for(sum = 0 , i = 1; i <= n; ++i, sum += i);
```

Notice the loop has an empty body, hence the semicolon.

# The do-while() Loop

```
do {  
    statement - 1  
    .  
    .  
    .  
    statement - n  
} while ( test );
```

Unlike the while() loop, the do-while() will always be executed at least once.



# The do-while() Loop

```
do {  
    statement -1  
    .  
    .  
    .  
    statement -n  
} while ( test );
```

Unlike the while() loop, the do-while() will always be executed at least once.

```
1  // Simple do-while countdown  
2  
3  #include <stdio.h>  
4  
5  int main(void)  
6  {  
7  
8      int n = 9;  
9  
10     /* This program always prints at least one  
11        number, even if n initialised to 0 */  
12     do{  
13         printf("%d ", n);  
14         n--;  
15     }while(n > 0);  
16     printf("\n");  
17     return 0;  
18 }
```

Execution :

9 8 7 6 5 4 3 2 1

# The switch() Statement

```
switch (val) {  
    case 1 :  
        a++;  
        break;  
    case 2 :  
    case 3 :  
        b++;  
        break;  
    default :  
        c++;  
}
```

- The val must be an integer.

# The switch() Statement

```
switch (val) {  
    case 1 :  
        a++;  
        break;  
    case 2 :  
    case 3 :  
        b++;  
        break;  
    default :  
        c++;  
}
```

- The val must be an integer.
- The break statement causes execution to jump out of the loop. No break statement causes execution to 'fall through' to the next line.

# The switch() Statement

```
switch (val) {  
    case 1 :  
        a++;  
        break;  
    case 2 :  
    case 3 :  
        b++;  
        break;  
    default :  
        c++;  
}
```

- The val must be an integer.
- The break statement causes execution to jump out of the loop. No break statement causes execution to 'fall through' to the next line.
- The default label is a catch-all.

# The switch() Statement

```
1  /* A Prime number can only be divided
2     exactly by 1 and itself */
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8
9      int i, n;
10     do{
11         printf("Enter a number from 2 - 9 : ");
12         n = scanf("%d", &i);
13     }while( (n!=1) || (i<2) || (i>9) );
14     switch(i){
15         case 2:
16         case 3:
17         case 5:
18         case 7:
19             printf("That's a prime!\n");
20             break;
21         default:
22             printf("That is not a prime!\n");
23     }
24     return 0;
25 }
```

Execution :

```
Enter a number from 2 - 9 : 1
Enter a number from 2 - 9 : 0
Enter a number from 2 - 9 : 10
Enter a number from 2 - 9 : 3
That's a prime!
```

# The Conditional (?) Operator

As we have seen, C programmers have a range of techniques available to reduce the amount of typing :

```
expr1 ? expr2 : expr3
```

If `expr1` is *true* then `expr2` is executed, else `expr3` is evaluated.

# The Conditional (?) Operator

As we have seen, C programmers have a range of techniques available to reduce the amount of typing :

```
expr1 ? expr2 : expr3
```

If `expr1` is *true* then `expr2` is executed, else `expr3` is evaluated.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int    x, y, z;
6
7      printf("Input three integers: ");
8      if (scanf("%d%d%d", &zx, &zy, &z) != 3){
9          printf("Didn't get 3 numbers?\n");
10         return 1;
11     }
12     int min;
13     min = (x < y) ? x : y;
14     min = (z < min) ? z : min;
15     printf("The minimum value is %d\n", min);
16     return 0;
17 }
```

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

**E: Functions**

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings



# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16 }
17
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16 }
17
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

- Execution begins, as normal, in the `main()` function.

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16
17 }
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

- Execution begins, as normal, in the `main()` function.
- The function *prototype* is shown at the top of the file. This allows the compiler to check the code more thoroughly.

# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16 }
17
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

- Execution begins, as normal, in the `main()` function.
- The function *prototype is shown* at the top of the file. This allows the compiler to check the code more thoroughly.
- The function is defined between two braces.

# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16 }
17
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

- Execution begins, as normal, in the `main()` function.
- The function *prototype is shown* at the top of the file. This allows the compiler to check the code more thoroughly.
- The function is defined between two braces.
- The function `min()` returns an `int` and takes two `int`'s as arguments. These are copies of `j` and `k`.

# Simple Functions

```
1  #include <stdio.h>
2
3  int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("\nOf the two values %d and %d, " \
14           "the minimum is %d.\n\n", j, k, m);
15     return 0;
16 }
17
18
19 int min(int a, int b)
20 {
21     if (a < b)
22         return a;
23     else
24         return b;
25 }
```

Execution :

Input two integers: 5 2

Of the two values 5 and 2, the minimum is 2.

- Execution begins, as normal, in the `main()` function.
- The function *prototype is shown* at the top of the file. This allows the compiler to check the code more thoroughly.
- The function is defined between two braces.
- The function `min()` returns an `int` and takes two `int`'s as arguments. These are copies of `j` and `k`.
- The `return` statement is used to return a value to the calling statement.

# Call-by-Value

In the following example, a function is passed an integer using call by value:

```
1  #include <stdio.h>
2
3  void fnc1(int x);
4
5  int main(void)
6  {
7
8      int x = 1;
9
10     fnc1(x);
11     printf("%d\n", x);
12 }
13
14 void fnc1(int x)
15 {
16     x = x + 1;
17 }
```

Execution :

1

# Call-by-Value

In the following example, a function is passed an integer using call by value:

```
1  #include <stdio.h>
2
3  void fnc1(int x);
4
5  int main(void)
6  {
7
8      int x = 1;
9
10     fnc1(x);
11     printf("%d\n", x);
12 }
13
14 void fnc1(int x)
15 {
16     x = x + 1;
17 }
```

Execution :

1

- The function does not change the value of `x` in `main()`, since `x` in the function is effectively only a **copy** of the variable.



# Call-by-Value

In the following example, a function is passed an integer using call by value:

```
1  #include <stdio.h>
2
3  void fnc1(int x);
4
5  int main(void)
6  {
7
8      int x = 1;
9
10     fnc1(x);
11     printf("%d\n", x);
12 }
13
14 void fnc1(int x)
15 {
16     x = x + 1;
17 }
```

Execution :

1

- The function does not change the value of `x` in `main()`, since `x` in the function is effectively only a **copy** of the variable.
- A function which has no return value, is declared `void` and, in other languages, might be termed a *procedure*.

# Call-by-Value

In the following example, a function is passed an integer using call by value:

```
1  #include <stdio.h>
2
3  void fnc1(int x);
4
5  int main(void)
6  {
7
8      int x = 1;
9
10     fnc1(x);
11     printf("%d\n", x);
12 }
13
14 void fnc1(int x)
15 {
16     x = x + 1;
17 }
```

Execution :

1

- The function does not change the value of `x` in `main()`, since `x` in the function is effectively only a **copy** of the variable.
- A function which has no return value, is declared `void` and, in other languages, might be termed a *procedure*.
- Most parameters used as arguments to functions in C are copied - this is known as *call-by-value*. We'll see the alternative, *call-by-reference*, later.

# Testing

# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10           n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.

# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10           n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.
- A factor is a number by which a larger (whole/integer) number can be divided.

# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10           n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.
- A factor is a number by which a larger (whole/integer) number can be divided.
- 36 has 6 factors: 1, 2, 3, 4, 6, 12 and 36 itself.

# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10             n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.
- A factor is a number by which a larger (whole/integer) number can be divided.
- 36 has 6 factors: 1, 2, 3, 4, 6, 12 and 36 itself.
- How do we know the program works though ?

# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10             n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.
- A factor is a number by which a larger (whole/integer) number can be divided.
- 36 has 6 factors: 1, 2, 3, 4, 6, 12 and 36 itself.
- How do we know the program works though ?
- Running it ?

Number of factors in 12 is 6



# Testing

```
1  #include <stdio.h>
2
3  int numfactors(int f);
4
5  int main(void)
6  {
7
8      int n = 12;
9      printf("Number of factors in %d is %d\n", \
10             n, numfactors(n));
11     return 0;
12 }
13
14 int numfactors(int k)
15 {
16
17     int count = 0;
18
19     for(int i=1; i<=k; i++){
20         if( (k%i)==0) {
21             count++;
22         }
23     }
24     return count;
25 }
```

- This is a (not very good) function to compute the number of factors a number has.
- A factor is a number by which a larger (whole/integer) number can be divided.
- 36 has 6 factors: 1, 2, 3, 4, 6, 12 and 36 itself.
- How do we know the program works though ?
- Running it ?  
Number of factors in 12 is 6
- We need something more automated.

# Pre- and Post-Conditions

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8
9      int n = 12;
10     printf("Number of factors in %d is %d\n", \
11           n, numfactors(n));
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     assert(k >= 1); // Avoid trying zero
21     for(int i=1; i<=k; i++){
22         if( (k%i)==0) {
23             count++;
24         }
25     }
26     assert(count <= k);
27     return count;
28 }
```

# Pre- and Post-Conditions

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8
9      int n = 12;
10     printf("Number of factors in %d is %d\n", \
11           n, numfactors(n));
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     assert(k >= 1); // Avoid trying zero
21     for(int i=1; i<=k; i++){
22         if( (k%i)==0) {
23             count++;
24         }
25     }
26     assert(count <= k);
27     return count;
28 }
```

- Pre-conditions check the inputs to functions, typically their arguments.

# Pre- and Post-Conditions

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8
9      int n = 12;
10     printf("Number of factors in %d is %d\n", \
11           n, numfactors(n));
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     assert(k >= 1); // Avoid trying zero
21     for(int i=1; i<=k; i++){
22         if( (k%i)==0) {
23             count++;
24         }
25     }
26     assert(count <= k);
27     return count;
28 }
```

- Pre-conditions check the inputs to functions, typically their arguments.
- Post-conditions check the returns from functions.

# Pre- and Post-Conditions

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8
9      int n = 12;
10     printf("Number of factors in %d is %d\n", \
11           n, numfactors(n));
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     assert(k >= 1); // Avoid trying zero
21     for(int i=1; i<=k; i++){
22         if( (k%i)==0) {
23             count++;
24         }
25     }
26     assert(count <= k);
27     return count;
28 }
```

- Pre-conditions check the inputs to functions, typically their arguments.
- Post-conditions check the returns from functions.
- An assert simply states some test that **ought** to be true. If not, the program aborts with an error.

# Pre- and Post-Conditions

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8
9      int n = 12;
10     printf("Number of factors in %d is %d\n", \
11           n, numfactors(n));
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     assert(k >= 1); // Avoid trying zero
21     for(int i=1; i<=k; i++){
22         if( (k%i)==0) {
23             count++;
24         }
25     }
26     assert(count <= k);
27     return count;
28 }
```

- Pre-conditions check the inputs to functions, typically their arguments.
- Post-conditions check the returns from functions.
- An assert simply states some test that **ought** to be true. If not, the program aborts with an error.
- There's a sense that this is somehow *safer*, but we haven't exactly done much testing on it to ensure the correct answers are returned.

# Assert Testing

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8      assert(numfactors(17) == 2);
9      assert(numfactors(12) == 6);
10     assert(numfactors(6) == 4);
11     assert(numfactors(0) == 0); // ?
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     for(int i=1; i<=k; i++){
21         if( (k%i)==0) {
22             count++;
23         }
24     }
25     return count;
26 }
```

# Assert Testing

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8      assert(numfactors(17) == 2);
9      assert(numfactors(12) == 6);
10     assert(numfactors(6) == 4);
11     assert(numfactors(0) == 0); // ?
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     for(int i=1; i<=k; i++){
21         if( (k%i)==0) {
22             count++;
23         }
24     }
25     return count;
26 }
```

- We will use assert testing in this style **every** time we write a function.



# Assert Testing

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8      assert(numfactors(17) == 2);
9      assert(numfactors(12) == 6);
10     assert(numfactors(6) == 4);
11     assert(numfactors(0) == 0); // ?
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     for(int i=1; i<=k; i++){
21         if( (k%i)==0) {
22             count++;
23         }
24     }
25     return count;
26 }
```

- We will use assert testing in this style **every** time we write a function.
- These tests tend to get quite long, so we generally collect them in a function called test() which itself is called from main().

# Assert Testing

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8      assert(numfactors(17) == 2);
9      assert(numfactors(12) == 6);
10     assert(numfactors(6) == 4);
11     assert(numfactors(0) == 0); // ?
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     for(int i=1; i<=k; i++){
21         if( (k%i)==0) {
22             count++;
23         }
24     }
25     return count;
26 }
```

- We will use assert testing in this style **every** time we write a function.
- These tests tend to get quite long, so we generally collect them in a function called test() which itself is called from main().
- If there is no error, there is no output from this program.

# Assert Testing

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int numfactors(int f);
5
6  int main(void)
7  {
8      assert(numfactors(17) == 2);
9      assert(numfactors(12) == 6);
10     assert(numfactors(6) == 4);
11     assert(numfactors(0) == 0); // ?
12     return 0;
13 }
14
15 int numfactors(int k)
16 {
17
18     int count = 0;
19
20     for(int i=1; i<=k; i++){
21         if( (k%i)==0) {
22             count++;
23         }
24     }
25     return count;
26 }
```

- We will use assert testing in this style **every** time we write a function.
- These tests tend to get quite long, so we generally collect them in a function called test() which itself is called from main().
- If there is no error, there is no output from this program.
- By #define'ing NDEBUG before the #include <assert.h>, all assertions are ignored, allowing them to be used during development and switched off later.

# Self-test : Multiply

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)

```
1  /* Try to write mult(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mult( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mult( int a,  int b)
18 {
19     // To be completed
20
21 }
22
23
24 void test(void)
25 {
26     assert(mult(5,3) == 15);
27     assert(mult(3,5) == 15);
28     assert(mult(0,3) == 0);
29     assert(mult(3,0) == 0);
30     assert(mult(1,8) == 8);
31     assert(mult(8,1) == 8);
32 }
```

# Self-test : Multiply

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)
- Use iteration (a loop) to achieve this.

```
1  /* Try to write mul(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mul( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mul( int a,  int b)
18 {
19
20     // To be completed
21
22 }
23
24 void test(void)
25 {
26     assert(mul(5,3) == 15);
27     assert(mul(3,5) == 15);
28     assert(mul(0,3) == 0);
29     assert(mul(3,0) == 0);
30     assert(mul(1,8) == 8);
31     assert(mul(8,1) == 8);
32 }
```

# Self-test : Multiply

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)
- Use iteration (a loop) to achieve this.
- $7 \times 8$  is computed by adding up 7 eight times.

```
1  /* Try to write mul(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mul( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mul( int a,  int b)
18 {
19     // To be completed
20
21 }
22
23
24 void test(void)
25 {
26     assert(mul(5,3) == 15);
27     assert(mul(3,5) == 15);
28     assert(mul(0,3) == 0);
29     assert(mul(3,0) == 0);
30     assert(mul(1,8) == 8);
31     assert(mul(8,1) == 8);
32 }
```

# Self-test : Multiply

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)
- Use iteration (a loop) to achieve this.
- $7 \times 8$  is computed by adding up 7 eight times.
- Use `assert()` calls to test it thoroughly - I've given you some to get you started.

```
1  /* Try to write mult(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mult( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mult( int a,  int b)
18 {
19     // To be completed
20 }
21
22
23
24 void test(void)
25 {
26     assert(mult(5,3) == 15);
27     assert(mult(3,5) == 15);
28     assert(mult(0,3) == 0);
29     assert(mult(3,0) == 0);
30     assert(mult(1,8) == 8);
31     assert(mult(8,1) == 8);
32 }
```

# Program Layout

It is normal for the `main()` function to come first in a program :

```
#include <stdio.h>
#include <stdlib.h>

list of function prototypes

int main(void)
{
    . . . . .
}

int f1(int a, int b)
{
    . . . . .
}

int f2(int a, int b)
{
    . . . . .
}
```



# Program Layout

It is normal for the `main()` function to come first in a program :

```
#include <stdio.h>
#include <stdlib.h>

list of function prototypes

int main(void)
{
    . . . . .
}

int f1(int a, int b)
{
    . . . . .
}

int f2(int a, int b)
{
    . . . . .
}
```

However, it is theoretically possible to avoid the need for function prototypes by defining a function before it is used :

```
#include <stdio.h>
#include <stdlib.h>

int f1(int a, int b)
{
    . . . . .
}

int f2(int a, int b)
{
    . . . . .
}

int main(void)
{
    . . . . .
}
```

We will **never** use this second approach - put `main()` first with the prototypes above it.

# Replacing Functions with Macros

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

Input two integers: 5 2

Minimum is 2

# Replacing Functions with Macros

- There's sometimes a (tiny) time penalty for using functions.

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

Input two integers: 5 2

Minimum is 2

# Replacing Functions with Macros

- There's sometimes a (tiny) time penalty for using functions.
- The contents of the functions are saved onto a special stack, so that when you return to the function, its variables and state can be restored.

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

```
Input two integers: 5 2
Minimum is 2
```

# Replacing Functions with Macros

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

Input two integers: 5 2

Minimum is 2

- There's sometimes a (tiny) time penalty for using functions.
- The contents of the functions are saved onto a special stack, so that when you return to the function, its variables and state can be restored.
- [https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

# Replacing Functions with Macros

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

```
Input two integers: 5 2
Minimum is 2
```

- There's sometimes a (tiny) time penalty for using functions.
- The contents of the functions are saved onto a special stack, so that when you return to the function, its variables and state can be restored.
- [https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)
- Historically, for small functions that needed to be fast, programmers might have `#define` a macro.

# Replacing Functions with Macros

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

```
Input two integers: 5 2
Minimum is 2
```

- There's sometimes a (tiny) time penalty for using functions.
- The contents of the functions are saved onto a special stack, so that when you return to the function, its variables and state can be restored.
- [https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)
- Historically, for small functions that needed to be fast, programmers might have `#define` a macro.
- There's a problem though - what happens if we used `m = MIN(i++, j++)` ; ?

# Replacing Functions with Macros

```
1  #include <stdio.h>
2
3  #define MIN(A, B) ((A)<(B)?(A):(B))
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = MIN(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16 }
```

Execution :

Input two integers: 5 2

Minimum is 2

- There's sometimes a (tiny) time penalty for using functions.
- The contents of the functions are saved onto a special stack, so that when you return to the function, its variables and state can be restored.
- [https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)
- Historically, for small functions that needed to be fast, programmers might have `#define` a macro.
- There's a problem though - what happens if we used `m = MIN(i++, j++)` ; ?
- This is expanded to `((i++)<(j++)?(i++):(j++))` which is **not** what was intended.



# The inline modifier

- In C99 the inline modifier was introduced  
[https://en.wikipedia.org/wiki/Inline\\_function](https://en.wikipedia.org/wiki/Inline_function)

*... serves as a compiler directive that suggests (but does not require) that the compiler substitute the body of the function inline by performing inline expansion, i.e. by inserting the function code at the address of each function call, thereby saving the overhead of a function call.*

```
1  #include <stdio.h>
2
3  static inline int min(int a, int b);
4
5  int main(void)
6  {
7
8      int j, k, m;
9
10     printf("Input two integers: ");
11     scanf("%d%d", &j, &k);
12     m = min(j, k);
13     printf("Minimum is %d\n", m);
14     return 0;
15 }
16
17 inline int min(int a, int b)
18 {
19     if (a < b)
20         return a;
21     else
22         return b;
23 }
24 }
```

Execution :

```
Input two integers: 5 2
Minimum is 2
```

# Factorials via Iteration

- A repeated computation is normally achieved via *iteration*, e.g. using `for()`:

# Factorials via Iteration

- A repeated computation is normally achieved via *iteration*, e.g. using `for()`:
- Here we compute the factorial of a number - the factorial of 4, written as  $4!$ , is simply  $4 \times 3 \times 2 \times 1$ .

# Factorials via Iteration

- A repeated computation is normally achieved via *iteration*, e.g. using `for()`:
- Here we compute the factorial of a number - the factorial of 4, written as  $4!$ , is simply  $4 \times 3 \times 2 \times 1$ .
- Obviously, we'd do more assert tests in the full version.

# Factorials via Iteration

- A repeated computation is normally achieved via *iteration*, e.g. using `for()`:
- Here we compute the factorial of a number - the factorial of 4, written as  $4!$ , is simply  $4 \times 3 \times 2 \times 1$ .
- Obviously, we'd do more assert tests in the full version.

# Factorials via Iteration

- A repeated computation is normally achieved via *iteration*, e.g. using `for()`:
- Here we compute the factorial of a number - the factorial of 4, written as  $4!$ , is simply  $4 \times 3 \times 2 \times 1$ .
- Obviously, we'd do more assert tests in the full version.

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int fact(int a);
5
6  int main(void)
7  {
8      assert(fact(4) == 24);
9      assert(fact(1) == 1);
10     assert(fact(0) == 1);
11     assert(fact(10) == 3628800);
12     return(0);
13 }
14
15 int fact(int a)
16 {
17     int i;
18     int tot = 1;
19
20     for(i=1; i<=a; i++){
21         tot *= i;
22     }
23     return tot;
24 }
25
26
27 }
```

# Factorials via Recursion (Advanced)

- We could achieve the same result using recursion.

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int fact(int a);
5
6  int main(void)
7  {
8      assert(fact(4) == 24);
9      assert(fact(1) == 1);
10     assert(fact(0) == 1);
11     assert(fact(10) == 3628800);
12     return(0);
13 }
14
15
16 int fact(int a)
17 {
18
19     if(a > 0)
20         return ( a * fact(a - 1) );
21     else
22         return 1;
23 }
24 }
```

# Factorials via Recursion (Advanced)

- We could achieve the same result using recursion.
- The factorial of 4 can be thought of as  $4 \times 3!$

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int fact(int a);
5
6  int main(void)
7  {
8      assert(fact(4) == 24);
9      assert(fact(1) == 1);
10     assert(fact(0) == 1);
11     assert(fact(10) == 3628800);
12     return(0);
13 }
14
15
16 int fact(int a)
17 {
18
19     if(a > 0)
20         return ( a * fact(a - 1) );
21     else
22         return 1;
23 }
24
```



# Factorials via Recursion (Advanced)

- We could achieve the same result using recursion.
- The factorial of 4 can be thought of as  $4 \times 3!$
- A recursive function calls *itself* - there may be many versions of the same function 'alive' at the same time during execution.

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int fact(int a);
5
6  int main(void)
7  {
8      assert(fact(4) == 24);
9      assert(fact(1) == 1);
10     assert(fact(0) == 1);
11     assert(fact(10) == 3628800);
12     return(0);
13 }
14
15
16 int fact(int a)
17 {
18
19     if(a > 0)
20         return ( a * fact(a - 1) );
21     else
22         return 1;
23 }
24 }
```

# Self-test : Multiply (Advanced)

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)

```
1  /* Try to write mult(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mul( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mul( int a,  int b)
18 {
19
20     // To be completed
21
22 }
23
24 void test(void)
25 {
26     assert(mul(5,3) == 15);
27     assert(mul(3,5) == 15);
28     assert(mul(0,3) == 0);
29     assert(mul(3,0) == 0);
30     assert(mul(1,8) == 8);
31     assert(mul(8,1) == 8);
32 }
```

# Self-test : Multiply (Advanced)

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)
- Use recursion to achieve this.

```
1  /* Try to write mult(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mul( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mul( int a,  int b)
18 {
19     // To be completed
20 }
21
22
23
24 void test(void)
25 {
26     assert(mul(5,3) == 15);
27     assert(mul(3,5) == 15);
28     assert(mul(0,3) == 0);
29     assert(mul(3,0) == 0);
30     assert(mul(1,8) == 8);
31     assert(mul(8,1) == 8);
32 }
```

# Self-test : Multiply (Advanced)

- Write a simple function `int mul(int a, int b)` which multiplies two integers together **without** the use of the multiply symbol in C (i.e. the `*`)
- Use recursion to achieve this.
- Use `assert()` calls to test it thoroughly.

```
1  /* Try to write mul(a,b) without using
2     any maths cleverer than addition.    */
3
4  #include <stdio.h>
5  #include <assert.h>
6
7  int mul( int a,  int b);
8  void test(void);
9
10 int main(void)
11 {
12     test();
13
14     return 0;
15 }
16
17 int mul( int a,  int b)
18 {
19     // To be completed
20 }
21
22 void test(void)
23 {
24     assert(mul(5,3) == 15);
25     assert(mul(3,5) == 15);
26     assert(mul(0,3) == 0);
27     assert(mul(3,0) == 0);
28     assert(mul(1,8) == 8);
29     assert(mul(8,1) == 8);
30 }
31
32 }
```

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

**F: Data Types**

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

# Fundamental Data types

- [ unsigned | signed ]
- [ long | short ]
- [ char | int | float | double ]
- The use of int implies signed int without the need to state it.
- Likewise unsigned short means unsigned short int.

Type	Minimum size (bits)	Format specifier
char	8	%c
signed char	8	%c (or %hhi for numerical output)
unsigned char	8	%c (or %hhu for numerical output)
short short int signed short signed short int	16	%hi or %hd
unsigned short unsigned short int	16	%hu
int signed signed int	16	%i or %d
unsigned unsigned int	16	%u
long long int signed long signed long int	32	%li or %ld
unsigned long unsigned long int	32	%lu
long long long long int signed long long signed long long int	64	%lli or %lld
unsigned long long unsigned long long int	64	%llu
float		scanf(): %f, %g, %e, %a
double		%lf, %lg, %le, %la
long double		%Lf, %Lg, %Le, %La

# Binary Storage of Numbers

In an unsigned char :

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	1	0	0	1	1	0	0

The above represents :

$$1 * 64 + 1 * 8 + 1 * 4 = 76.$$

- Floating operations need not be exact.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      float d = 0.1;
7      printf("%.12f\n", 3.0*d);
8      return 0;
9  }
```

Execution :

0.3000000004470

# Binary Storage of Numbers

In an unsigned char :

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	1	0	0	1	1	0	0

The above represents :

$$1 * 64 + 1 * 8 + 1 * 4 = 76.$$

- Floating operations need not be exact.

- Not all floats are representable so are only approximated.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      float d = 0.1;
7      printf("%.12f\n", 3.0*d);
8      return 0;
9  }
```

Execution :

0.3000000004470



# Binary Storage of Numbers

In an unsigned char :

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	1	0	0	1	1	0	0

The above represents :

$$1 * 64 + 1 * 8 + 1 * 4 = 76.$$

- Floating operations need not be exact.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      float d = 0.1;
7      printf("%.12f\n", 3.0*d);
8      return 0;
9  }
```

Execution :

0.3000000004470

- Not all floats are representable so are only approximated.
- Since floats may not be stored exactly, it doesn't make sense to try and compare them:

```
if ( d == 0.3 )
```

# Binary Storage of Numbers

In an unsigned char :

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	1	0	0	1	1	0	0

The above represents :

$$1 * 64 + 1 * 8 + 1 * 4 = 76.$$

- Floating operations need not be exact.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      float d = 0.1;
7      printf("%.12f\n", 3.0*d);
8      return 0;
9  }
```

Execution :

0.3000000004470

- Not all floats are representable so are only approximated.
- Since floats may not be stored exactly, it doesn't make sense to try and compare them:

```
if ( d == 0.3 )
```

- Therefore, we don't allow this by explicitly using the compiler warning flag: `-Wfloat-equal`

# sizeof()

To find the exact size in bytes of a type on a particular machine, use sizeof(). On a Dell Windows 10 laptop running WSL:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5
6      printf("char      :%3ld\n", sizeof(char));
7      printf("short     :%3ld\n", sizeof(short));
8      printf("long      :%3ld\n", sizeof(long));
9      printf("unsigned   :%3ld\n", sizeof(unsigned));
10     printf("long long  :%3ld\n", sizeof(long long));
11     printf("float      :%3ld\n", sizeof(float));
12     printf("dbl        :%3ld\n", sizeof(double));
13     printf("long dbl   :%3ld\n", sizeof(long double));
14     printf("\n");
15
16     return 0;
17 }
```

Execution :

char	:	1
short	:	2
long	:	8
unsigned	:	4
long long	:	8
float	:	4
dbl	:	8
long dbl	:	16

# Mathematical Functions

- There are no mathematical functions built into the C language.
- However, there are many functions in the maths library which may be linked in using the **-lm** option with the compiler.
- Functions include :  
    `sqrt()` `pow()` `round()`  
    `fabs()` `exp()` `log()`  
    `sin()` `cos()` `tan()`
- Most take doubles as arguments and return doubles.

# Casting

```
1  /* Compute the Area of a Sphere
2     to the nearest integer      */
3  #include <stdio.h>
4  #include <math.h>
5
6  #define PI 3.14159265358979323846
7
8  int main(void)
9  {
10     double r;
11     printf("Enter a radius : ");
12     scanf("%lf", &r);
13     // Make sure radius is positive
14     r = fabs(r);
15     double a = 4.0 / 3.0 * PI * pow(r, (double) 3);
16     printf("Area of your ball = %f\n", a);
17     printf("Area of your ball = %.2f\n", a);
18     printf("Area of your ball = %d\n", (int)a);
19     printf("Area of your ball = %.0f\n", round(a));
20     return 0;
21 }
```

Execution :

```
Enter a radius : 7.75
Area of your ball = 1949.816390
Area of your ball = 1949.82
Area of your ball = 1949
Area of your ball = 1950
```

- An explicit type conversion is called a *cast*.
- *If it moves - cast it*. Don't trust the compiler to do it for you !

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** enum day.

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.



# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

- An example of their use:

```
enum day d1;  
...  
d1 = fri;
```

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

- An example of their use:

```
enum day d1;
```

```
...  
d1 = fri;
```

- The default numbering may be changed as well:  

```
enum fruit{apple=7, pear, orange=3, lemon};
```

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

- An example of their use:

```
enum day d1;
```

```
...  
d1 = fri;
```

- The default numbering may be changed as well:  

```
enum fruit{apple=7, pear, orange=3, lemon};
```
- Use enumerated types as constants to aid readability - they are self-documenting.

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

- An example of their use:

```
enum day d1;
```

```
...  
d1 = fri;
```

- The default numbering may be changed as well:  

```
enum fruit{apple=7, pear, orange=3, lemon};
```
- Use enumerated types as constants to aid readability - they are self-documenting.
- Declare them in a header (`.h`) file.

# Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value 0, the second has the value 1 and so on.

- An example of their use:

```
enum day d1;
```

```
...  
d1 = fri;
```

- The default numbering may be changed as well:  

```
enum fruit{apple=7, pear, orange=3, lemon};
```
- Use enumerated types as constants to aid readability - they are self-documenting.
- Declare them in a header (`.h`) file.
- Note that the type is `enum day`; the keyword `enum` is not enough.

- Sometimes it is useful to associate a particular name with a certain type, e.g.:  
`typedef int colour;`

# Typedefs

- Sometimes it is useful to associate a particular name with a certain type, e.g.:  
`typedef int colour;`
- Now the type `colour` is synonymous with the type `int`.



- Sometimes it is useful to associate a particular name with a certain type, e.g.:  
`typedef int colour;`
- Now the type `colour` is synonymous with the type `int`.
- Makes code self-documenting.

- Sometimes it is useful to associate a particular name with a certain type, e.g.:  
`typedef int colour;`
- Now the type `colour` is synonymous with the type `int`.
- Makes code self-documenting.
- Helps to control complexity when programmers are building complicated or lengthy user-defined types (See Structures 8).

# Combining typedefs and enums

- Often typedef's are used in conjunction with enumerated types:

```
#include <stdio.h>
#include <assert.h>

enum day {sun,mon,tue,wed,thu,fri,sat};
typedef enum day day;

day find_next_day(day d);

int main(void)
{
    assert(find_next_day(mon)==tue);
    assert(find_next_day(sat)==sun);
    assert(find_next_day(sun)==mon);
    return 0;
}
```

```
day find_next_day(day d)
{
    day next_day;
    switch(d){
        case sun:
            next_day = mon;
            break;
        case mon:
            next_day = tue;
            break;
        case tue:
            next_day = wed;
            break;
        case wed:
            next_day = thu;
            break;
        case thu:
            next_day = fri;
            break;
        case fri:
            next_day = tue;
            break;
        case sat:
            next_day = sun;
            break;
        default:
            printf("I wasn't expecting that !\n");
    }
    return next_day;
}
```

```
enum veg {beet, carrot, pea};  
typedef enum veg veg;  
veg v1, v2;  
v1 = carrot;
```

- We can combine the two operations into one:

```
typedef enum veg {beet,carrot,pea} veg;  
veg v1, v2;  
v1 = carrot;
```

```
enum veg {beet, carrot, pea};  
typedef enum veg veg;  
veg v1, v2;  
v1 = carrot;
```

- We can combine the two operations into one:

```
typedef enum veg {beet,carrot,pea} veg;  
veg v1, v2;  
v1 = carrot;
```

- Assigning:

```
v1 = 10;
```

is very poor programming style !

# Booleans

- Before C99 you might have been tempted to define your own Boolean type:

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  typedef enum bool {false, true} bool;
5
6  int main(void)
7  {
8
9      bool b = true;
10     if (b){
11         printf("It's true!\n");
12     }
13     else{
14         printf("It's false!\n");
15     }
16     return 0;
17 }
```

Execution :

It 's true!

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <assert.h>
4
5  int main(void)
6  {
7
8      bool b = true;
9      if (b){
10         printf("It's true!\n");
11     }
12     else{
13         printf("It's false!\n");
14     }
15     return 0;
16 }
```

Execution :

It 's true!

# Booleans

- Before C99 you might have been tempted to define your own Boolean type:

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  typedef enum bool {false, true} bool;
5
6  int main(void)
7  {
8
9      bool b = true;
10     if (b){
11         printf("It's true!\n");
12     }
13     else{
14         printf("It's false!\n");
15     }
16     return 0;
17 }
```

Execution :

It 's true!

- However, we can just use `#include <stdbool.h>`

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <assert.h>
4
5  int main(void)
6  {
7
8      bool b = true;
9      if (b){
10         printf("It's true!\n");
11     }
12     else{
13         printf("It's false!\n");
14     }
15     return 0;
16 }
```

Execution :

It 's true!

Rewrite/complete this code using typedefs and enums to create self-documenting code in any manner you wish.

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  // Argument 1 is temperature
5  // Argument 2 is scale (0=>Celsius, 1=>Fahrenheit)
6  int fvr(double t, int s);
7
8  int main(void)
9  {
10     assert(fvr(37.5, 0)==1);
11     assert(fvr(36.5, 0)==0);
12     assert(fvr(96.5, 1)==0);
13     assert(fvr(99.5, 1)==1);
14     return 0;
15 }
16
17 int fvr(double t, int s)
18 {
19 }
```



# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

**H : Constructed Types - 1D Arrays & Structures**

I : Characters & Strings

# 1D Arrays

- One-Dimensional arrays are declared by a type followed by an identifier with a bracketed constant expression:

```
float x[10];
```

```
int k[ARRAY_SIZE];
```

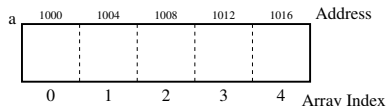
```
float y[i*2];
```

The following, however, is not valid:

```
float y[i*2];
```

- Arrays are stored in contiguous memory, e.g.:

```
int a[5];
```



```
1  #include <stdio.h>
2
3  #define N 500
4
5  int main(void)
6  {
7
8      /* allocate space a[0]...a[N-1] */
9      int a[N];
10     int i, sum = 0;
11     /* fill array */
12     for (i = 0; i < N; ++i){
13         a[i] = 7 + i * i;
14     }
15     /* print array */
16     for (i = 0; i < N; ++i){
17         printf("a[%d]=%d ", i, a[i]);
18     }
19     /* sum elements */
20     for (i = 0; i < N; ++i){
21         sum += a[i];
22     }
23     /* print sum */
24     printf("\nsum=%d\n", sum);
25     return 0;
26 }
```

- Arrays are indexed **0** to **n-1**.

# 1D Arrays : Initialisation

By default, arrays are uninitialised. When they are declared, they may be assigned a value:

```
float x[7] = {-1.1,0.2,2.0,4.4,6.5,0.0,7.7};
```

or,

```
float x[7] = {-1.1, 0.2};
```

the elements 2 ... 6 are set to zero.

Also:

```
int a[] = {3, 8, 9, 1};
```

is valid, the compiler assumes the array size to be 4.

- Accessing an array out of bounds will not be identified by the compiler. It may cause an error at run-time. One frequent result is that an entirely unrelated variable is altered.
- `a[5] = a[4] + 1;`
- `k[9]++;`
- `n[12+i] = 0;`

# 1D Arrays : Call by Reference

- Here, the array is passed by *Reference* - no copy of the array is made - the function processes the array that was created inside `main()`, despite it apparently having a 'different' name.
- All arrays are passed like this in C - we'll see later when we look at *pointers* why this is the case.

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <assert.h>
4  #define MAX 5
5  // Pass array, AND number of elements
6  int set_mean(int a[], unsigned int num);
7
8  int main(void)
9  {
10     int x[MAX], m;
11     x[0] = 2; x[1] = 3; x[2] = 3; x[3] = 3; x[4] = 3;
12     m = set_mean(x, 5); assert(m==3); assert(x[0] == m);
13     x[0] = 5; x[1] = 5; x[2] = 5; x[3] = 5; x[4] = 5;
14     m = set_mean(x, 5); assert(m==5); assert(x[2] == m);
15     assert(set_mean(x, 1)==5);
16     x[0] = 1; x[1] = 2; x[2] = 3;
17     assert(set_mean(x, 3)==2);
18     m = set_mean(x, 3); assert(m==2); assert(x[1] == m);
19     // Should also check for num != 0 ??
20 }
21
22 // Mean rounded later from doubles - each element of array set to mean
23 int set_mean(int a[], unsigned int num)
24 {
25     double tot = 0;
26     for(unsigned int i=0; i<num; i++){
27         tot += (double)a[i];
28     }
29     int mn = round(tot / (double) num);
30     for(unsigned int i=0; i<num; i++){
31         a[i] = mn;
32     }
33     return mn;
34 }
```

- A structure type allows the programmer to aggregate components into a single, named variable. Other languages call these *Records* or *Tuples*.
- Each component has individually named members.
- ```
struct employee {  
    long id;  
    double salary;  
    short age;  
};
```
- `struct` is a keyword, `employee` is the structure tag name, and `id`, `salary` and `age` are members of the structure.
- A statement of the form :  

```
struct employee e1, e2;
```

actually creates storage for the variables.
- A member is accessed using the member operator `“.”`
- ```
e1.salary = 35000.2;  
e2.age = 29;
```
- The member name must be unique within the same structure.
- Arrays of structures are possible, i.e.:  

```
struct employee team[400];
```

# Arrays of Structures

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

#define SUITS 4
#define PERSUIT 13
#define DECK (SUITS*PERSUIT)
#define SHUFFLE 3

typedef enum {hearts, diamonds, spades, clubs} suit;

struct card {
    suit st;
    int pips;
};

typedef struct card card;

void shuffle_deck(card d[DECK]);
void init_deck(card d[DECK]);
void print_deck(card d[DECK], int n);

int main(void)
{
    card deck[DECK];

    init_deck(deck);
    print_deck(deck, 7);
    shuffle_deck(deck);
    print_deck(deck, 7);
    return 0;
}
```

```
void init_deck(card d[DECK])
{
    for(int i=0; i<DECK; i++){
        // Number 1 .. 13
        d[i].piPs = (i/PERSUIT) + 1;
        switch (i/PERSUIT) {
            case hearts: d[i].st = hearts; break;
            case diamonds: d[i].st = diamonds; break;
            case spades: d[i].st = spades; break;
            case clubs: d[i].st = clubs; break;
            // Force an abort ?
            default : assert(false);
        }
    }
}

void shuffle_deck(card d[DECK])
{
    for(int i=0; i<SHUFFLE*DECK; i++){
        int n1 = rand()%DECK;
        int n2 = rand()%DECK;
        card c = d[n1]; d[n1] = d[n2]; d[n2] = c;
    }
}
```

# Arrays of Structures

```
void print_deck(card d[DECK], int n)
{
    for(int i=0; i<n; i++){
        switch(d[i].pips){
            case 11:
                printf("Jack");
                break;
            case 12:
                printf("Queen");
                break;
            case 13:
                printf("King");
                break;
            default:
                printf("%2d", d[i].pips);
        }
        switch(d[i].st){
            case hearts :
                printf(" of Hearts\n");
                break;
            case diamonds :
                printf(" of Diamonds\n");
                break;
            case spades:
                printf(" of Spades\n");
                break;
            default :
                printf(" of Clubs\n");
        }
        printf("\n");
    }
}
```

Execution :

1 of Hearts  
2 of Hearts  
3 of Hearts  
4 of Hearts  
5 of Hearts  
6 of Hearts  
7 of Hearts

Jack of Diamonds  
Queen of Diamonds  
2 of Clubs  
9 of Diamonds  
5 of Hearts  
6 of Hearts  
Queen of Hearts

- The print\_deck() function is clearly messy ! We can simplify this a little when we understand strings.

# Table of Contents

A: Preamble

B: Hello, World

C: Grammar

D: Flow Control

E: Functions

F: Data Types

G: Prettifying (New Types and Aliasing)

H : Constructed Types - 1D Arrays & Structures

I : Characters & Strings



# Storage of Characters

- Characters are stored in the machine as one byte (generally 8-bits storing one of **256** possible values).
- These may be thought of as characters, or very small integers.
- Only a subset of these 256 values are required for the printable characters, space, newline etc.
- Declaration:

```
char c;
```

```
c = 'A';
```

```
or :
```

```
char c1 = 'A', c2 = '*', c3 = ';' ;
```

- The particular integer used to represent a character is dependent on the encoding used. The most common of these, used on most UNIX and PC platforms, is ASCII.

lowercase	'a'	'b'	'c'	...	'z'
ASCII value	97	98	99	...	112
uppercase	'A'	'B'	'C'	...	'Z'
ASCII value	65	66	67	...	90
digit	'0'	'1'	'2'	...	'9'
ASCII value	48	49	50	...	57
other	'&'	'*'	'+'	...	
ASCII value	38	42	43	...	

# Using Characters

- When using `printf()` and `scanf()` the formats `%c` and `%d` do very different things :  

```
char c = 'a';  
printf("%c\n", c); /* prints : a */  
printf("%d\n", c); /* prints : 97 */
```
- Hard-to-print characters have an escape sequence i.e. to print a newline, the 2 character escape `'\n'` is used.

Escape sequence	Hex value	Character
<code>\a</code>	07	Alert (Beep, Bell)
<code>\b</code>	08	Backspace
<code>\e</code>	1B	Escape character
<code>\f</code>	0C	Formfeed Page Break
<code>\n</code>	0A	Newline (Line Feed)
<code>\r</code>	0D	Carriage Return
<code>\t</code>	09	Horizontal Tab
<code>\v</code>	0B	Vertical Tab
<code>\\</code>	5C	Backslash
<code>\'</code>	27	Apostrophe
<code>\"</code>	22	Double quote
<code>\?</code>	3F	Question mark

# Using getchar() and putchar()

```
1  /* Outputs characters twice */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7
8      char c;
9
10     do{
11         c = getchar();
12         putchar(c);
13         putchar(c);
14     }while(c != '!');
15     putchar('\n');
16
17     return 0;
18 }
```

Execution :

```
abc!
aabbcc!!
```

This has the unfortunate problem of requiring a 'special' character to terminate. More aggressively, the user could terminate by pressing CTRL-C.

```
1  /* Outputs characters twice */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7
8      char c; // char or int ?
9
10     while ((c = getchar()) != EOF) {
11         putchar(c);
12         putchar(c);
13     }
14     putchar('\n');
15
16     return 0;
17 }
```

Execution :

```
abc
aabbcc
```

The end-of-file constant is defined in `stdio.h`. Although system dependent, `-1` is often used. On the UNIX system this is generated when the end of a file being piped is reached, or when CTRL-D is pressed.

# Capitalization

```
1  /* Outputs characters twice */
2
3  #include <stdio.h>
4
5  #define CAPS ('A' - 'a')
6
7  int main(void)
8  {
9      int c;
10     while ((c = getchar()) != '!'){
11         if (c >= 'a' && c <= 'z'){
12             putchar(c + CAPS);
13         }
14         else{
15             putchar(c);
16         }
17     }
18     putchar('\n');
19     return 0;
20 }
21 }
```

Execution :

Hello World!  
HELLO WORLD

This is more easily achieved by using some of the definitions found in ctype.h.

Macro	true returned if:
isalnum(int c)	Letter or digit
isalpha(int c)	Letter
iscntrl(int c)	Control character
isdigit(int c)	Digit
isgraph(int c)	Printable (not space)
islower(int c)	Lowercase
isprint(int c)	Printable
ispunct(int c)	Punctuation
isspace(int c)	White Space
isupper(int c)	Uppercase
isxdigit(int c)	Hexadecimal
isascii(int c)	ASCII code

Some useful functions are :

Function/Macro	Returns:
int tolower(int c)	Lowercase c
int toupper(int c)	Uppercase c
int toascii(int c)	ASCII code for c

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main(void)
5  {
6
7      int c;
8      while ((c = getchar()) != EOF){
9          if (islower(c)){
10             putchar(toupper(c));
11         }
12         else{
13             putchar(c);
14         }
15     }
16     putchar('\n');
17     return 0;
18 }
```

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main(void)
5  {
6
7      int c;
8      while ((c = getchar()) != EOF){
9          /* toupper() returns non-lowercae
10             chars unaltered */
11             putchar(toupper(c));
12         }
13         putchar('\n');
14         return 0;
15     }
```

Execution :

Hello World!  
HELLO WORLD!

- Strings are 1D arrays of characters.
- Any character in a string may be accessed as an array element.
- The important difference between strings and ordinary arrays is the **end-of-string sentinel** `'\0'` or null character.
- The string "abc" has a *length* of 3, but its *size* is 4.
- Note `'a'` and `"a"` are different. The first is a character constant, the second is a string with 2 elements `'a'` and `'\0'`.

## Initialising Strings :

- `char w[6] = "Hello";`
- `char w[250];`  
`w[0] = 'a';`  
`w[1] = 'b';`  
`w[2] = 'c';`  
`w[3] = '\0';`
- `scanf("%s", w);`  
Removes leading spaces, reads a string (terminated by a space or EOF). Adds a null character to the end of the string.
- `char w[250] = {'a', 'b', 'c', '\0'};`

# Unused Letters and string.h

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  #define ALPHASIZE 26
5
6  int main(void)
7  {
8      char s[100] = "The Quick Brown Fox Leaps" \
9                  "Over the Lazy Dog";
10     short used[ALPHASIZE] = {0};
11     char c;
12     int i = 0;
13     while(s[i]){
14         c = tolower(s[i]);
15         if(islower(c)){
16             used[c - 'a'] = 1;
17         }
18         i++;
19     }
20     for(i=0; i<ALPHASIZE; i++){
21         if(!used[i]){
22             printf("%c has not been used.\n", i+'a');
23         }
24     }
25     return 0;
26 }
```

Execution :

```
j has not been used.
m has not been used.
```

In #include <string.h> :

```
char *strcat(char *dst, const char *src);
int strcmp(const char *s1, const char *s2);
```

- strcat() appends a copy of string src, including the terminating null character, to the end of string dst.
- strcmp() compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively.

# More string.h

In `#include <string.h>` :

```
char *strcpy(char *dst, const char *src);  
unsigned strlen(const char *s);
```

- `strcpy()` copies string `src` to `dst` including the terminating null character, stopping after the null character has been copied.
- `strlen()` returns the number of bytes in `s`, not including the terminating null character.

One way to write the function `strlen()`

```
1  #include <stdio.h>  
2  #include <string.h>  
3  #include <assert.h>  
4  
5  unsigned nstrlen(const char *s);  
6  
7  int main(void)  
8  {  
9      assert(nstrlen("Neill")==5);  
10     assert(nstrlen("")==0);  
11     assert(nstrlen("\n")==1);  
12     assert(nstrlen("abcdef")==strlen("abcdef"));  
13     return 0;  
14 }  
15  
16 unsigned nstrlen(const char s[])  
17 {  
18     register int n = 0;  
19  
20     while(s[n] != '\0'){  
21         ++n;  
22     }  
23     return n;  
24 }
```