Programming in C

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

University of Bristol

October 4, 2021

Table of Contents

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

Resources

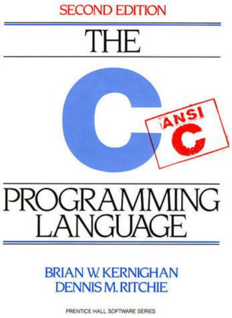
- Free : https://en.wikibooks.org/wiki/C_Programming
- A list of more : https://www.linuxlinks.com/excellent-free-books-learn-c/

Resources

- Free : https://en.wikibooks.org/wiki/C_Programming
- 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
- 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.



Computer Science Ethos

- 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.

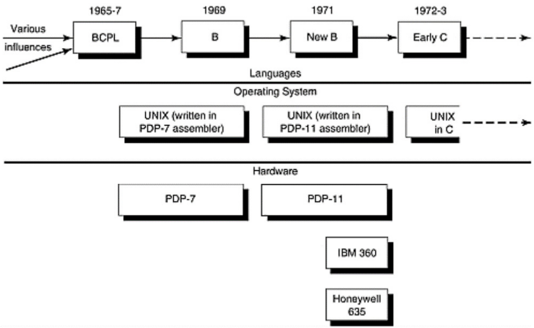
Computer Science Ethos

- 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.
- If someone takes your code and submits it, we need to investigate where it originated - all students involved will be part of this.

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.
- 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.

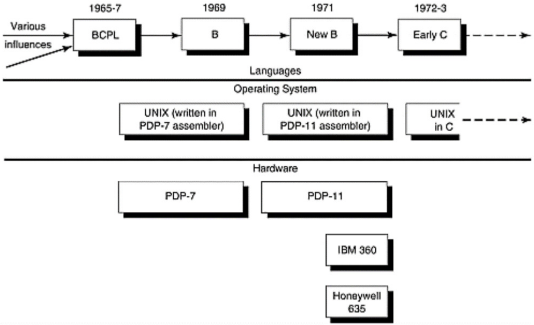
History of C



From **Deep C Secrets** by *Peter Van Der Linden*

- BCPL - Martin Richards

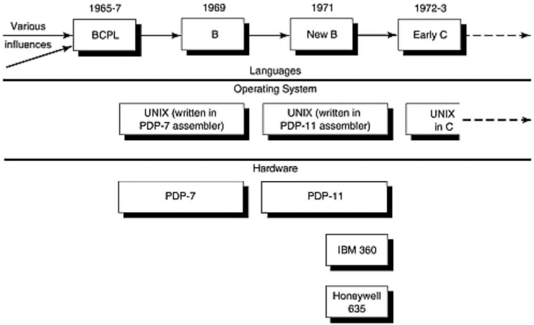
History of C



From **Deep C Secrets** by *Peter Van Der Linden*

- BCPL - Martin Richards
- B - Ken Thomson 1970

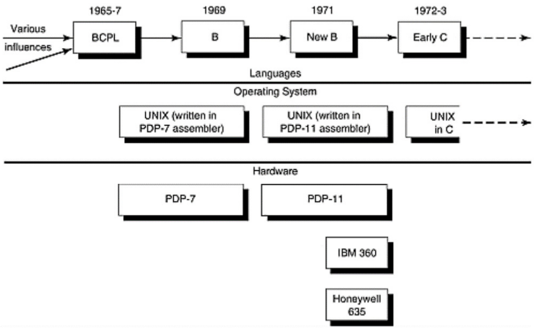
History of C



From **Deep C Secrets** by *Peter Van Der Linden*

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

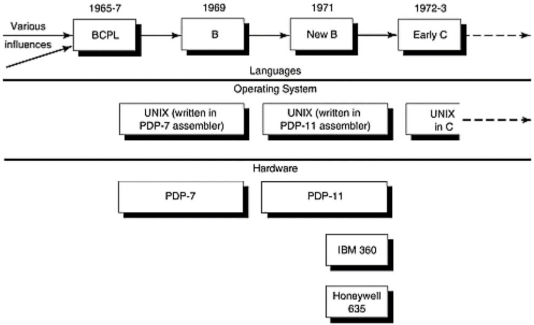
History of C



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.

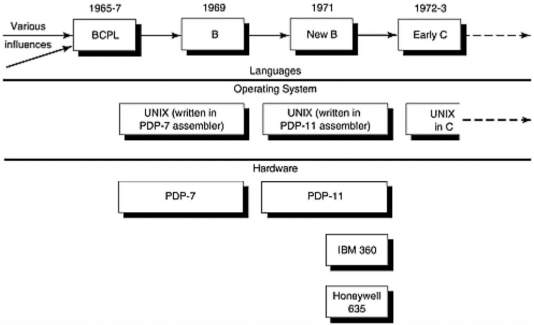
History of C



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

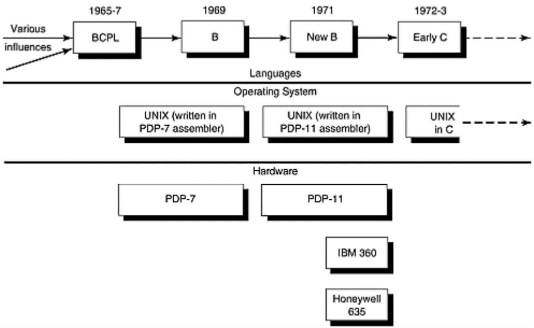
History of C



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

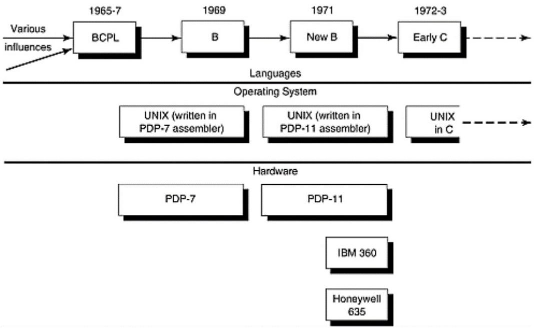
History of C



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)

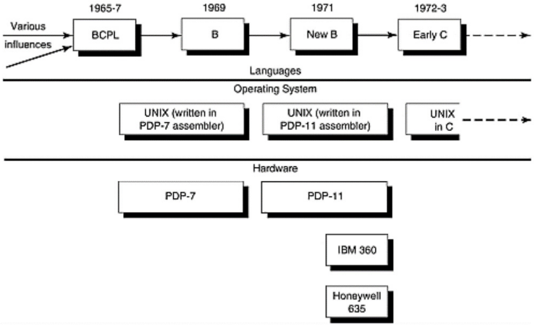
History of C



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	v	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	v	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	v	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	v	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

Programming and Software Engineering

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

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 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.

Assessment

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

Assessment

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

Assessment

- 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%).

Assessment

- 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`, MS Streams, MS Teams etc.)

Help with the Unit

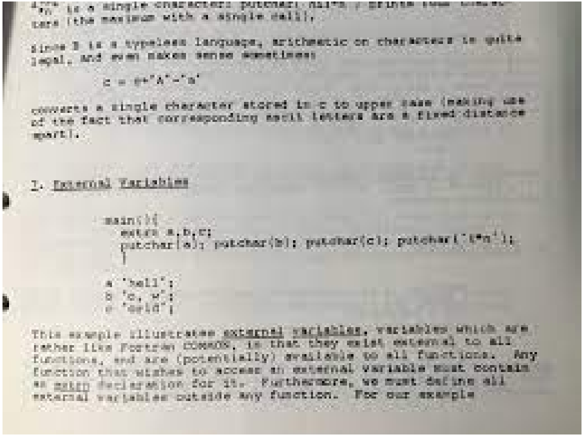
- All information is available via the Blackboard site (which will point you to other sites including `github.com`, MS Streams, MS Teams etc.)
- Online will mainly be via myself giving 'live' Q&A session, the associated MS Teams group with Forum, and Teaching Assistants in our on-campus / face-to-face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

Help with the Unit

- All information is available via the Blackboard site (which will point you to other sites including `github.com`, 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

Hello World!



'*n' is a single character: putchar('hi*n') prints four characters (the maximum with a single call).

Since B is a typeless language, arithmetic on characters is quite legal, and even makes sense sometimes:

```
c = c+'A'-'a'
```

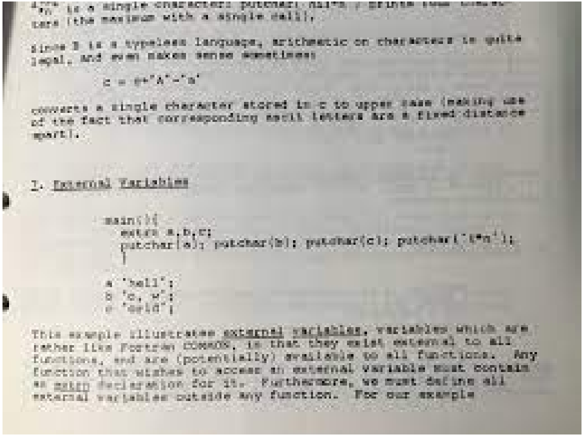
converts a single character stored in c to upper case (making use of the fact that corresponding ascii letters are a fixed distance apart).

7. External Variables

```
main( ) {
  extrn a,b,c;
  putchar(a); putchar(b); putchar(c); putchar('!*n');
  }

a 'hell';
b 'o, w';
c 'orld';
```

This example illustrates external variables, variables which are rather like Fortran COMMON, in that they exist external to all functions, and are (potentially) available to all functions. Any function that wishes to access an external variable must contain an extrn declaration for it. Furthermore, we must define all external variables outside any function. For our example

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



'*n' is a single character: putchar('hi*n') prints four characters (the maximum with a single call).

Since B is a typeless language, arithmetic on characters is quite legal, and even makes sense sometimes:

```
c = c+'A'-'a';
```

converts a single character stored in c to upper case (making use of the fact that corresponding ascii letters are a fixed distance apart).

7. External Variables

```
main( ) {
  extrn a,b,c;
  putchar(a); putchar(b); putchar(c); putchar('!*n');
  }

a 'hell';
b 'o, w';
c 'orld';
```

This example illustrates external variables, variables which are rather like Fortran COMMON, in that they exist external to all functions, and are (potentially) available to all functions. Any function that wishes to access an external variable must contain an extrn declaration for it. Furthermore, we must define all external variables outside any function. For our example

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

```c
/* The traditional first program
in honour of Dennis Ritchie
who invented C at Bell Labs
in 1972 */

#include <stdio.h>

int main(void)
{

    printf("Hello, world!\n");
    return 0;

}
```

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

```c
#include <stdio.h>

int main(void)
{
    // Compute the area of a rectangle
    int side1, side2, area;
    side1 = 7;
    side2 = 8;
    area = side1 * side2;

    printf("Length of side  1 = %d metres\n", side1);
    printf("Length of side  2 = %d metres\n", side2);
    printf("Area of rectangle = %d metres squared\n", area);
    return 0;
}
```

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.

```
preprocessing directives

int main(void)
{
    declarations

    statements
}
```

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

```
// Demonstration of character arithmetic
#include <stdio.h>

int main(void)
{
   char   c;

   c = 'A';
   printf("%c ", c);
   printf("%c\n", c+1);
   return 0;
}
```

Execution :

A B

The Character Type

```
// Demonstration of character arithmetic
#include <stdio.h>

int main(void)
{
   char   c;

   c = 'A';
   printf("%c ", c);
   printf("%c\n", c+1);
   return 0;
}
```

Execution :

A B

- The keyword `char` stands for character.

The Character Type

```c
// Demonstration of character arithmetic
#include <stdio.h>

int main(void)
{
    char   c;

    c = 'A';
    printf("%c ", c);
    printf("%c\n", c+1);
    return 0;
}
```

Execution :

A B

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

The Character Type

```c
// Demonstration of character arithmetic
#include <stdio.h>

int main(void)
{
    char   c;

    c = 'A';
    printf("%c ", c);
    printf("%c\n", c+1);
    return 0;
}
```

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

```c
// Demonstration of character arithmetic
#include <stdio.h>

int main(void)
{
   char   c;

   c = 'A';
   printf("%c ", c);
   printf("%c\n", c+1);
   return 0;
}
```

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

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

Execution :

```
Sum of x & y is 3.000000.
```

Floating Types

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

Execution :

```
Sum of x & y is 3.000000.
```

- In C there are three common floating types :

Floating Types

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

Execution :

```
Sum of x & y is 3.000000.
```

- In C there are three common floating types :
  1. float

Floating Types

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

Execution :

```
Sum of x & y is 3.000000.
```

- In C there are three common floating types :
  1. float
  2. double

Floating Types

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

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

```c
#include <stdio.h>

int main(void)
{

    double x, y;

    x = 1.0;
    y = 2.0;

    printf("Sum of x & y is %f.\n", x + y);

    return 0;

}
```

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 tings).

Using printf()

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

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

Using printf()

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

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

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

Using printf()

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

%c	Characters
%d	Integers
%e	Floats/Doubles (Engineering Notation)
%f	Floats/Doubles
%s	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;
}
```

```c
// Sums are computed.
#include <stdio.h>

int main(void)
{

    int cnt = 0;
    float sum = 0.0, x;
    printf("Input some numbers: ");
    while (scanf("%f", &x) == 1) {
       cnt = cnt + 1;
       sum = sum + x;
    }

    printf("\n%s%5d\n%s%5f\n\n",
           "Count:", cnt, "Sum: ", sum);
    return 0;
}
```

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

Grammar

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

Grammar

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

Grammar

- 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.

Grammar

- 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`.

Grammar

- 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).

Grammar

- 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;

```
#include <stdio.h>

int main(void)
{
    int a, c = 0;
    a = ++c;
    int b = c++;
    printf("%d %d %d\n", a, b, ++c);
    return 0;
}
```

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. -=, *=, /=.

```c
// 1st few powers of 2 are printed.

#include <stdio.h>

int main(void)
{
    int   i = 0, power = 1;

    while (++i <= 10){
        printf("%5d", power *= 2);
    }
    printf("\n");
    return 0;
}
```

Execution :

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

The Standard Library

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

int main(void)
{
    int   i, n;
    printf("Randomly distributed integers are printed.\n"
           "How many do you want to see?  ");
    do{
        i = scanf("%d", &n);
    }while(i != 1);
    for (i = 0; i < n; ++i) {
        if (i % 4 == 0)
            printf("\n");
        printf("%12d", rand());
    }
    printf("\n");
    return 0;
}
```

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

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

int main(void)
{
    int   i, n;
    printf("Randomly distributed integers are printed.\n"
           "How many do you want to see?  ");
    do{
        i = scanf("%d", &n);
    }while(i != 1);
    for (i = 0; i < n; ++i) {
        if (i % 4 == 0)
            printf("\n");
        printf("%12d", rand());
    }
    printf("\n");
    return 0;
}
```

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`.

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

int main(void)
{
    int   i, n;
    printf("Randomly distributed integers are printed.\n"
           "How many do you want to see?  ");
    do{
        i = scanf("%d", &n);
    }while(i != 1);
    for (i = 0; i < n; ++i) {
        if (i % 4 == 0)
            printf("\n");
        printf("%12d", rand());
    }
    printf("\n");
    return 0;
}
```

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

Comparisons

<table>
<tr><td><</td><td>less than</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td><=</td><td>less than or equal to</td></tr>
<tr><td>>=</td><td>greater than or equal to</td></tr>
<tr><td>==</td><td>equal to</td></tr>
<tr><td>!=</td><td>not equal to</td></tr>
<tr><td>!</td><td>not</td></tr>
<tr><td>&&</td><td>logical AND</td></tr>
<tr><td>||</td><td>logical OR</td></tr>
</table>

Comparisons

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	**equal to**
!=	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
==	**equal to**
!=	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
==	**equal to**
!=	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
==	**equal to**
!=	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
==	**equal to**
!=	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
==	**equal to**
!=	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
==	**equal to**
!=	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:

```c
#include <stdio.h>

int main(void)
{
    int   x, y, z;

    printf("Input three integers:  ");
    if(scanf("%d%d%d", &x, &y, &z) != 3){
        printf("Didn't get 3 numbers?\n");
        return 1;
    }
    int min;
    if (x < y){
        min = x;
    }
    // Nasty, dropped braces:
    else
        min = y;
    if (z < min){
        min = z;
    }
    printf("The minimum value is %d\n", min);
    return 0;
}
```

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.

```
// Simple while countdown

#include <stdio.h>

int main(void)
{

   int n = 9;

   while(n > 0){
      printf("%d ", n);
      n--;
   }
   printf("\n");
   return 0;
}
```

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
}
```

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.

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

```c
// Triples of integers that sum to N
#include <stdio.h>

#define N 7

int main(void)
{
   int   cnt = 0, i, j, k;

   for(i = 0; i <= N; i++){
      for(j = 0; j <= N; j++){
         for(k = 0; k <= N; k++){
            if(i + j + k == N){
               ++cnt;
               printf("%3d%3d%3d\n", i, j, k);
            }
         }
      }
   }
   printf("\nCount: %d\n", cnt);
   return 0;
}
```

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.

```
// Simple do-while countdown

#include <stdio.h>

int main(void)
{

    int n = 9;

    /* This program always prints at least one
       number, even if n initialised to 0 */
    do{
       printf("%d ", n);
       n--;
    }while(n > 0);
    printf("\n");
    return 0;
}
```

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

```
/* A Prime number can only be divided
   exactly by 1 and itself */

#include <stdio.h>

int main(void)
{

   int i, n;
   do{
      printf("Enter a number from 2 - 9 : ");
      n = scanf("%d", &i);
   }while( (n!=1) || (i<2) || (i>9) );
   switch(i){
      case 2:
      case 3:
      case 5:
      case 7:
         printf("That's a prime!\n");
         break;
      default:
         printf("That is not a prime!\n");
   }
   return 0;
}
```

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 programers 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 programers 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.

```
#include <stdio.h>

int main(void)
{
    int   x, y, z;

    printf("Input three integers:  ");
    if(scanf("%d%d%d", &x, &y, &z) != 3){
       printf("Didn't get 3 numbers?\n");
       return 1;
    }
    int min;
    min = (x < y) ? x : y;
    min = (z < min) ? z : min;
    printf("The minimum value is %d\n", min);
    return 0;
}
```

Table of Contents

Simple Functions

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = min(j, k);
    printf("\nOf the two values %d and %d, " \
    "the minimum is %d.\n\n", j, k, m);
    return 0;

}

int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

Execution :

```
Input two integers: 5 2

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

Simple Functions

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    m = min(j, k);
    printf("\nOf the two values %d and %d, " \
    "the minimum is %d.\n\n", j, k, m);
    return 0;

}

int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

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.

Simple Functions

```
#include <stdio.h>

int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = min(j, k);
    printf("\nOf the two values %d and %d, " \
    "the minimum is %d.\n\n", j, k, m);
    return 0;

}

int min(int a, int b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

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

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{

   int j, k, m;

   printf("Input two integers: ");
   scanf("%d%d", &j, &k);
  m = min(j, k);
   printf("\nOf the two values %d and %d, " \
   "the minimum is %d.\n\n", j, k, m);
   return 0;

}

int min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}
```

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

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = min(j, k);
    printf("\nOf the two values %d and %d, " \
    "the minimum is %d.\n\n", j, k, m);
    return 0;

}

int min(int a, int b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

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

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = min(j, k);
    printf("\nOf the two values %d and %d, " \
    "the minimum is %d.\n\n", j, k, m);
    return 0;

}

int min(int a, int b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

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:

```c
#include <stdio.h>

void fnc1(int x);

int main(void)
{
    int x = 1;

    fnc1(x);
    printf("%d\n", x);
}

void fnc1(int x)
{
    x = x + 1;
}
```

Execution :

1

Call-by-Value

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

```c
#include <stdio.h>

void fnc1(int x);

int main(void)
{
    int x = 1;

    fnc1(x);
    printf("%d\n", x);
}

void fnc1(int x)
{
    x = x + 1;
}
```

Execution :

1

- The function does not change the value of `x` in `main()`, since a 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:

```c
#include <stdio.h>

void fnc1(int x);

int main(void)
{
    int x = 1;

    fnc1(x);
    printf("%d\n", x);
}

void fnc1(int x)
{
    x = x + 1;
}
```

Execution :

1

- The function does not change the value of `x` in `main()`, since a 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:

```c
#include <stdio.h>

void fnc1(int x);

int main(void)
{
   int x = 1;

   fnc1(x);
   printf("%d\n", x);
}

void fnc1(int x)
{
   x = x + 1;
}
```

Execution :

1

- The function does not change the value of `x` in `main()`, since a 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

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

Testing

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
           n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

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

Testing

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
           n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

- 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

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
              n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

- 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.
- 12 has 6 factors: $1, 2, 3, 4, 6$ and 12 itself.

Testing

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
           n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

- 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.
- 12 has 6 factors: $1, 2, 3, 4, 6$ and 12 itself.
- How do we know the program works though ?

Testing

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
           n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

- 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.
- 12 has 6 factors: $1, 2, 3, 4, 6$ and 12 itself.
- How do we know the program works though ?
- Running it ?

  Number of factors in 12 is 6

Testing

```c
#include <stdio.h>

int numfactors(int f);

int main(void)
{

    int n = 12;
    printf("Number of factors in %d is %d\n", \
           n, numfactors(n));
    return 0;
}

int numfactors(int k)
{

    int count = 0;

    for(int i=1; i<=k; i++){
        if( (k%i)==0) {
            count++;
        }
    }
    return count;
}
```

- 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.
- 12 has 6 factors: $1, 2, 3, 4, 6$ and 12 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

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

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   assert(k >= 1); // Avoid trying zero
   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   assert(count <= k);
   return count;
}
```

Pre- and Post-Conditions

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

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   assert(k >= 1); // Avoid trying zero
   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   assert(count <= k);
   return count;
}
```

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

Pre- and Post-Conditions

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

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   assert(k >= 1); // Avoid trying zero
   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   assert(count <= k);
   return count;
}
```

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

Pre- and Post-Conditions

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

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   assert(k >= 1); // Avoid trying zero
   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   assert(count <= k);
   return count;
}
```

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

Pre- and Post-Conditions

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

int numfactors(int f);

int main(void)
{

   int n = 12;
   printf("Number of factors in %d is %d\n", \
          n, numfactors(n));
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   assert(k >= 1); // Avoid trying zero
   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   assert(count <= k);
   return count;
}
```

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

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

int numfactors(int f);

int main(void)
{
   assert(numfactors(17) == 2);
   assert(numfactors(12) == 6);
   assert(numfactors(6) == 4);
   assert(numfactors(0) == 0); // ?
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

Assert Testing

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

int numfactors(int f);

int main(void)
{
   assert(numfactors(17) == 2);
   assert(numfactors(12) == 6);
   assert(numfactors(6) == 4);
   assert(numfactors(0) == 0); // ?
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

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

Assert Testing

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

int numfactors(int f);

int main(void)
{
   assert(numfactors(17) == 2);
   assert(numfactors(12) == 6);
   assert(numfactors(6) == 4);
   assert(numfactors(0) == 0); // ?
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

- 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

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

int numfactors(int f);

int main(void)
{
   assert(numfactors(17) == 2);
   assert(numfactors(12) == 6);
   assert(numfactors(6) == 4);
   assert(numfactors(0) == 0); // ?
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

- 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

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

int numfactors(int f);

int main(void)
{
   assert(numfactors(17) == 2);
   assert(numfactors(12) == 6);
   assert(numfactors(6) == 4);
   assert(numfactors(0) == 0); // ?
   return 0;
}

int numfactors(int k)
{

   int count = 0;

   for(int i=1; i<=k; i++){
      if( (k%i)==0) {
         count++;
      }
   }
   return count;
}
```

- 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 multiples two integers together **without** the use of the multiply symbol in C (i.e. the $*$)

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult( int a,  int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult( int a,  int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Self-test : Multiply

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

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult(int a, int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult(int a, int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Self-test : Multiply

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

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult(int a, int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult(int a, int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Self-test : Multiply

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

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult(int a, int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult(int a, int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

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

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

Execution :

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

Replacing Functions with Macros

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

Execution :

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

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

Replacing Functions with Macros

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

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.

Replacing Functions with Macros

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
    m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

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

Replacing Functions with Macros

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

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
- Historically, for small functions that needed to be fast, programmers might have #define a macro.

Replacing Functions with Macros

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

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
- 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

```c
#include <stdio.h>

#define MIN(A, B) ((A)<(B)?(A):(B))

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = MIN(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}
```

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
- 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

  ... 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.

```c
#include <stdio.h>

static inline int min(int a, int b);

int main(void)
{

    int j, k, m;

    printf("Input two integers: ");
    scanf("%d%d", &j, &k);
   m = min(j, k);
    printf("Minimum is %d\n", m);
    return 0;

}

inline int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

Execution :

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

Factorials via Iteration

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

Factorials via Iteration

- A repeated computation 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 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 verson.

Factorials via Iteration

- A repeated computation 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 verson.

Factorials via Iteration

- A repeated computation 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 verson.

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

int fact(int a);

int main(void)
{
    assert(fact(4) == 24);
    assert(fact(1) == 1);
    assert(fact(0) == 1);
    assert(fact(10) == 3628800);
    return(0);

}

int fact(int a)
{

    int i;
    int tot = 1;

    for(i=1; i<=a; i++){
        tot *= i;
    }
    return tot;

}
```

Factorials via Recursion (Advanced)

- We could achieve the same result using recursion.

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

int fact(int a);

int main(void)
{
    assert(fact(4) == 24);
    assert(fact(1) == 1);
    assert(fact(0) == 1);
    assert(fact(10) == 3628800);
    return(0);

}

int fact(int a)
{

    if(a > 0)
        return ( a * fact(a - 1) );
    else
        return 1;

}
```

Factorials via Recursion (Advanced)

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

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

int fact(int a);

int main(void)
{
    assert(fact(4) == 24);
    assert(fact(1) == 1);
    assert(fact(0) == 1);
    assert(fact(10) == 3628800);
    return(0);

}

int fact(int a)
{

    if(a > 0)
       return ( a * fact(a - 1) );
    else
       return 1;

}
```

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.

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

int fact(int a);

int main(void)
{
    assert(fact(4) == 24);
    assert(fact(1) == 1);
    assert(fact(0) == 1);
    assert(fact(10) == 3628800);
    return(0);

}

int fact(int a)
{

    if(a > 0)
       return ( a * fact(a - 1) );
    else
       return 1;

}
```

Self-test : Multiply (Advanced)

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

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.      */

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

int mult( int a,  int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult( int a,  int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Self-test : Multiply (Advanced)

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

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult(int a, int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult(int a, int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Self-test : Multiply (Advanced)

- Write a simple function int mul(int a, int b) which multiples 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.

```c
/* Try to write mult(a,b) without using
   any maths cleverer than addition.     */

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

int mult(int a, int b);
void test(void);

int main(void)
{
   test();

   return 0;
}

int mult(int a, int b)
{

// To be completed

}

void test(void)
{
   assert(mult(5,3) == 15);
   assert(mult(3,5) == 15);
   assert(mult(0,3) == 0);
   assert(mult(3,0) == 0);
   assert(mult(1,8) == 8);
   assert(mult(8,1) == 8);
}
```

Table of Contents

Fundamental Data types

- **[** unsigned | signed **]**

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

Fundamental Data types

- [unsigned | signed]
- [long | short]

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

Fundamental Data types

- [unsigned | signed]
- [long | short]
- [char | int | float | double]

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

Fundamental Data types

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

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

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.

```c
#include <stdio.h>

int main(void)
{

    float d = 0.1;
    printf("%.12f\n", 3.0*d);
    return 0;
}
```

Execution :

```
0.300000004470
```

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.

```c
#include <stdio.h>

int main(void)
{

    float d = 0.1;
    printf("%.12f\n", 3.0*d);
    return 0;
}
```

Execution :

```
0.300000004470
```

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.

```c
#include <stdio.h>

int main(void)
{

    float d = 0.1;
    printf("%.12f\n", 3.0*d);
    return 0;
}
```

Execution :

```
0.300000004470
```

- Not all floats are representable so are only approximated.

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.

```c
#include <stdio.h>

int main(void)
{

    float d = 0.1;
    printf("%.12f\n", 3.0*d);
    return 0;
}
```

Execution :

```
0.300000004470
```

- 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:

```c
    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.

```
#include <stdio.h>

int main(void)
{

    float d = 0.1;
    printf("%.12f\n", 3.0*d);
    return 0;
}
```

Execution :

```
0.300000004470
```

- 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:

```
#include <stdio.h>

int main(void)
{

    printf("char        :%3ld\n", sizeof(char));
    printf("short       :%3ld\n", sizeof(short));
    printf("long        :%3ld\n", sizeof(long));
    printf("unsigned    :%3ld\n", sizeof(unsigned));
    printf("long long   :%3ld\n", sizeof(long long));
    printf("float       :%3ld\n", sizeof(float));
    printf("dbl         :%3ld\n", sizeof(double));
    printf("long dbl    :%3ld\n", sizeof(long double));
    printf("\n");

    return 0;
}
```

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:

```c
#include <stdio.h>

int main(void)
{

    printf("char        :%3ld\n", sizeof(char));
    printf("short       :%3ld\n", sizeof(short));
    printf("long        :%3ld\n", sizeof(long));
    printf("unsigned    :%3ld\n", sizeof(unsigned));
    printf("long long   :%3ld\n", sizeof(long long));
    printf("float       :%3ld\n", sizeof(float));
    printf("dbl         :%3ld\n", sizeof(double));
    printf("long dbl    :%3ld\n", sizeof(long double));
    printf("\n");

    return 0;
}
```

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.

Mathematical Functions

- There are no mathematical functions built into the C language.
- However, there are many functions in the maths library which may linked in using the **-lm** option with the compiler.

Mathematical Functions

- There are no mathematical functions built into the C language.
- However, there are many functions in the maths library which may linked in using the **-lm** option with the compiler.
- Functions include :

```
sqrt()  pow() round()
fabs() exp()  log()
sin()  cos()  tan()
```

Mathematical Functions

- There are no mathematical functions built into the C language.
- However, there are many functions in the maths library which may 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

```c
/* Compute the Area of a Sphere
   to the nearest integer       */
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979323846

int main(void)
{
   double r;
   printf("Enter a radius : ");
   scanf("%lf", &r);
   // Make sure radius is positive
   r = fabs(r);
   double a = 4.0 / 3.0 * PI * pow(r, (double) 3);
   printf("Area of your ball = %f\n", a);
   printf("Area of your ball = %.2f\n", a);
   printf("Area of your ball = %d\n", (int)a);
   printf("Area of your ball = %.0f\n", round(a));
   return 0;
}
```

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*.

Casting

```c
/* Compute the Area of a Sphere
   to the nearest integer       */
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979323846

int main(void)
{
    double r;
    printf("Enter a radius : ");
    scanf("%lf", &r);
    // Make sure radius is positive
    r = fabs(r);
    double a = 4.0 / 3.0 * PI * pow(r, (double) 3);
    printf("Area of your ball = %f\n", a);
    printf("Area of your ball = %.2f\n", a);
    printf("Area of your ball = %d\n", (int)a);
    printf("Area of your ball = %.0f\n", round(a));
    return 0;
}
```

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

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.

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.

Typedefs

- 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`.

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`.
- Makes code self-documenting.

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`.
- Makes code self-documenting.
- Helps to control complexity when programmers are building complicated or lengthy user-defined types (See Structures later).

Combining typedefs and enums

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

```c
#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;
}
```

```c
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;
}
```

Style

```
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;
  ```

Style

```
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:

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

typedef enum bool {false , true} bool;

int main(void)
{

   bool b = true;
   if (b){
      printf("It's true!\n");
   }
   else{
      printf("It's false!\n");
   }
   return 0;
}
```

Execution :

```
It's true!
```

Booleans

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

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

typedef enum bool {false, true} bool;

int main(void)
{

    bool b = true;
    if (b){
        printf("It's true!\n");
    }
    else{
        printf("It's false!\n");
    }
    return 0;
}
```

Execution :

```
It's true!
```

Booleans

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

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

typedef enum bool {false , true} bool;

int main(void)
{

    bool b = true;
    if (b){
        printf("It's true!\n");
    }
    else{
        printf("It's false!\n");
    }
    return 0;
}
```

Execution :

```
It's true!
```

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

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

int main(void)
{

    bool b = true;
    if (b){
        printf("It's true!\n");
    }
    else{
        printf("It's false!\n");
    }
    return 0;
}
```

Execution :

```
It's true!
```

Fever

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

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

// Argument 1 is temperature
// Argument 2 is scale (0=>Celsius, 1=>Farenheit)
int fvr(double t, int s);

int main(void)
{
    assert(fvr(37.5, 0)==1);
    assert(fvr(36.5, 0)==0);
    assert(fvr(96.5, 1)==0);
    assert(fvr(99.5, 1)==1);
    return 0;
}

int fvr(double t, int s)
{
}
```

Table of Contents

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];
  ```

  The following, however, is not valid:

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

1D Arrays

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

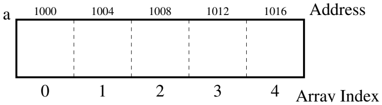
  ```
  float x[10];
  int k[ARRAY_SIZE];
  ```

  The following, however, is not valid:

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

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

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



1D Arrays

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

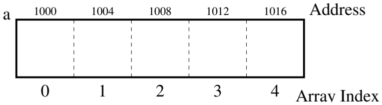
  ```
  float x[10];
  int k[ARRAY_SIZE];
  ```

  The following, however, is not valid:

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

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

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



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

1D Arrays

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

  float x[10];

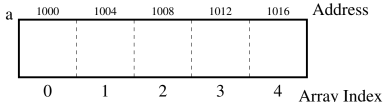
  int k[ARRAY_SIZE];

  The following, however, is not valid:

  float y[i*2];

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

  int a[5];



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

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

  float x[10];

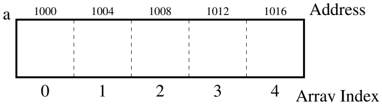
  int k[ARRAY_SIZE];

  The following, however, is not valid:

  float y[i*2];

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

  int a[5];



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

```c
#include <stdio.h>

#define N 500

int main(void)
{

    /* allocate space a[0]...a[N-1] */
    int a[N];
    int i, sum = 0;
    /* fill array */
    for (i = 0; i < N; ++i){
       a[i] = 7 + i * i;
    }
    /* print array */
    for (i = 0; i < N; ++i){
       printf("a[%d]=%d ", i, a[i]);
    }
    /* sum elements */
    for (i = 0; i < N; ++i){
       sum += a[i];
    }
    /* print sum */
    printf("\nsum=%d\n", sum);
    return 0;
}
```

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.

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.

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;

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]++;`

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

```c
#include <stdio.h>
#include <math.h>
#include <assert.h>
#define MAX 5

// Pass array, AND number of elements
void set_array(int a[MAX], unsigned int len, int n);

int main(void)
{
   int x[MAX] = {2, 3, 3, 3, 3};
   set_array(x, 5, 3); assert(x[0] == 3);
   x[0] = 5; x[1] = 5; x[2] = 5; x[3] = 5; x[4] = 5;
   set_array(x, 5, 4); assert(x[2] == 4);
   set_array(x, 1, 0); assert(x[0] == 0);
   x[0] = 1; x[1] = 2; x[2] = 3;
   set_array(x, 3, 2);
   assert(x[2] == 2); assert(x[3] == 4);
}

// Set all values of array (size len) to n
void set_array(int a[MAX], unsigned int len, int n)
{
   if(len == 0){
      return;
   }
   for(unsigned int i=0; i<len; i++){
     a[i] = n;
   }
}
```

1D Arrays : Call by Reference

```c
#include <stdio.h>
#include <math.h>
#include <assert.h>
#define MAX 5

// Pass array, AND number of elements
void set_array(int a[MAX], unsigned int len, int n);

int main(void)
{
   int x[MAX] = {2, 3, 3, 3, 3};
   set_array(x, 5, 3); assert(x[0] == 3);
   x[0] = 5; x[1] = 5; x[2] = 5; x[3] = 5; x[4] = 5;
   set_array(x, 5, 4); assert(x[2] == 4);
   set_array(x, 1, 0); assert(x[0] == 0);
   x[0] = 1; x[1] = 2; x[2] = 3;
   set_array(x, 3, 2);
   assert(x[2] == 2); assert(x[3] == 4);
}

// Set all values of array (size len) to n
void set_array(int a[MAX], unsigned int len, int n)
{
   if(len == 0){
      return;
   }
   for(unsigned int i=0; i<len; i++){
     a[i] = n;
   }
}
```

- 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.

1D Arrays : Call by Reference

```c
#include <stdio.h>
#include <math.h>
#include <assert.h>
#define MAX 5

// Pass array, AND number of elements
void set_array(int a[MAX], unsigned int len, int n);

int main(void)
{
   int x[MAX] = {2, 3, 3, 3, 3};
   set_array(x, 5, 3); assert(x[0] == 3);
   x[0] = 5; x[1] = 5; x[2] = 5; x[3] = 5; x[4] = 5;
   set_array(x, 5, 4); assert(x[2] == 4);
   set_array(x, 1, 0); assert(x[0] == 0);
   x[0] = 1; x[1] = 2; x[2] = 3;
   set_array(x, 3, 2);
   assert(x[2] == 2); assert(x[3] == 4);
}

// Set all values of array (size len) to n
void set_array(int a[MAX], unsigned int len, int n)
{
   if(len == 0){
      return;
   }
   for(unsigned int i=0; i<len; i++){
     a[i] = n;
   }
}
```

- 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.

Structures

- A structure type allows the programmer to aggregate components into a single, named variable. Other languages call these *Records* or *Tuples*.

Structures

- 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.

Structures

- 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;
  };
  ```
  ```

Structures

- 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.
  ```

Structures

- 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.
  ```

Structures

- 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.

Structures

- 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 {`
  ```
  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 "."

Structures

- 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;
  ```

Structures

- 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.
  ```

Structures

- 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

```c
#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);
void test(void);

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

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

Arrays of Structures

```c
#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);
void test(void);

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

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

```c
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

```c
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

 4 of Spades
Jack of Spades
 7 of Clubs
 9 of Spades
10 of Spades
 7 of Hearts
 2 of Spades
```

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

Testing

```
void test(void)
{
   int n = 0;
   card d[DECK];
   init_deck(d);
   // Direct assignment
   card c = {hearts, 10};
   // 1st element initialised correctly
   assert(d[9].pips == c.pips);
   assert(d[9].st == c.st);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      // Happens 1 time in 52 ?
      if((d[0].st == c.st) && (d[0].pips == c.pips)){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

- Note the direct ability to copy a structure.

Testing

```c
void test(void)
{
   int n = 0;
   card d[DECK];
   init_deck(d);
   // Direct assignment
   card c = {hearts, 10};
   // 1st element initialised correctly
   assert(d[9].pips == c.pips);
   assert(d[9].st == c.st);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      // Happens 1 time in 52 ?
      if((d[0].st == c.st) && (d[0].pips == c.pips)){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

- Note the direct ability to copy a structure.
- You can't compare them using == though.

Testing

```
void test(void)
{
   int n = 0;
   card d[DECK];
   init_deck(d);
   // Direct assignment
   card c = {hearts, 10};
   // 1st element initialised correctly
   assert(d[9].pips == c.pips);
   assert(d[9].st == c.st);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      // Happens 1 time in 52 ?
      if((d[0].st == c.st) && (d[0].pips == c.pips)){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

- Note the direct ability to copy a structure.
- You can't compare them using == though.
- Tricky to think of a good test for shuffle_deck.

Testing

```
void test(void)
{
   int n = 0;
   card d[DECK];
   init_deck(d);
   // Direct assignment
   card c = {hearts, 10};
   // 1st element initialised correctly
   assert(d[9].pips == c.pips);
   assert(d[9].st == c.st);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      // Happens 1 time in 52 ?
      if((d[0].st == c.st) && (d[0].pips == c.pips)){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

- Note the direct ability to copy a structure.
- You can't compare them using == though.
- Tricky to think of a good test for shuffle_deck.
- You could also typedef away the array, e.g.:

  ```
  typedef card deck[DECK];

  void shuffle_deck(deck d);
  ```

  But this hides the fact it's an array (which seems odd?)

Table of Contents

Storage of Characters

- Characters are stored in the machine as one byte (generally 8-bits storing one of **256** possible values).

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 a characters, or very small integers.

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 a characters, or very small integers.
- Only a subset of these 256 values are required for the printable characters, space, newline etc.

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 a 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 = ';' ;
```

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 a 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 = ';' ;
```

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 a 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 */

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

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

Using getchar() and putchar()

```c
// Outputs characters twice

#include <stdio.h>

int main(void)
{

    char c;

    do{
        c = getchar();
        putchar(c);
        putchar(c);
    }while(c != '!');
    putchar('\n');

    return 0;
}
```

Execution :

```
abc123!
aabbcc112233!!
```

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

Using getchar() and putchar()

```c
// Outputs characters twice

#include <stdio.h>

int main(void)
{

    char c;

    do{
        c = getchar();
        putchar(c);
        putchar(c);
    }while(c != '!');
    putchar('\n');

    return 0;
}
```

Execution :

```
abc123!
aabbcc112233!!
```

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

```c
// Outputs characters twice

#include <stdio.h>

int main(void)
{

    char c; // char or int ?

    while ((c = getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }
    putchar('\n');

    return 0;
}
```

Execution :

```
abc123
aabbcc112233
```

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

```c
// Outputs characters twice

#include <stdio.h>

#define CAPS ('A' - 'a')

int main(void)
{
    int   c;
    while ((c = getchar()) != '!'){
        if (c >= 'a' && c <= 'z'){
            putchar(c + CAPS);
        }
        else{
            putchar(c);
        }
    }
    putchar('\n');

    return 0;
}
```

Execution :

```
Hello World!
HELLO WORLD
```

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

Capitalization

```c
// Outputs characters twice

#include <stdio.h>

#define CAPS ('A' - 'a')

int main(void)
{
    int  c;
    while ((c = getchar()) != '!'){
        if (c >= 'a' && c <= 'z'){
          putchar(c + CAPS);
        }
        else{
            putchar(c);
        }
    }
    putchar('\n');

    return 0;
}
```

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

ctype.h

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

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{

    int c;
    while ((c = getchar()) != EOF){
        if (islower(c)){
            putchar(toupper(c));
        }
        else{
            putchar(c);
        }
    }
    putchar('\n');
    return 0;
}
```

ctype.h

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

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{

    int c;
    while ((c = getchar()) != EOF){
        if (islower(c)){
            putchar(toupper(c));
        }
        else{
            putchar(c);
        }
    }
    putchar('\n');
    return 0;
}
```

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{

    int c;
    while ((c = getchar()) != EOF){
        /* toupper() returns non-lowercae
           chars unaltered */
        putchar(toupper(c));
    }
    putchar('\n');
    return 0;
}
```

Execution :

```
Hello World!
HELLO WORLD!
```

Strings

- Strings are 1D arrays of characters.

Strings

- Strings are 1D arrays of characters.
- Any character in a string may be accessed as an array element.

Strings

- 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.

Strings

- 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.

Strings

- 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'.

Strings

- 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'.

Strings

- 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";`

Strings

- 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';
  ```
  ```

Strings

- 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.
  ```

Strings

- 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

```c
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>

#define ALPHASIZE 26

int main(void)
{
   char s[100] = "The Quick Brown Fox Leaps" \
                 "Over the Lazy Dog";
   bool used[ALPHASIZE] = {false};
   int i = 0;
   while(s[i]){
      char c = tolower(s[i]);
      if(islower(c)){
         used[c - 'a'] = true;
      }
      i++;
   }
   for(i=0; i<ALPHASIZE; i++){
      if(!used[i]){
         printf("%c has not been used.\n", i+'a');
      }
   }
   return 0;
}
```

Execution :

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

Unused Letters and string.h

```c
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>

#define ALPHASIZE 26

int main(void)
{
    char s[100] = "The Quick Brown Fox Leaps" \
                  "Over the Lazy Dog";
    bool used[ALPHASIZE] = {false};
    int i = 0;
    while(s[i]){
        char c = tolower(s[i]);
        if(islower(c)){
            used[c - 'a'] = true;
        }
        i++;
    }
    for(i=0; i<ALPHASIZE; i++){
        if(!used[i]){
            printf("%c has not been used.\n", i+'a');
        }
    }
    return 0;
}
```

Execution :

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

In #include <string.h> :

char *strcat(char dest[], 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.

Unused Letters and string.h

```c
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>

#define ALPHASIZE 26

int main(void)
{
   char s[100] = "The Quick Brown Fox Leaps" \
                 "Over the Lazy Dog";
   bool used[ALPHASIZE] = {false};
   int i = 0;
   while(s[i]){
      char c = tolower(s[i]);
      if(islower(c)){
         used[c - 'a'] = true;
      }
      i++;
   }
   for(i=0; i<ALPHASIZE; i++){
      if(!used[i]){
         printf("%c has not been used.\n", i+'a');
      }
   }
   return 0;
}
```

Execution :

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

In #include <string.h> :

char *strcat(char dest[], 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.

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.

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.

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()

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

unsigned nstrlen(const char s[]);

int main(void)
{
    assert(nstrlen("Neill")==5);
    assert(nstrlen("")==0);
    assert(nstrlen("\n")==1);
    assert(nstrlen("abcdef")==nstrlen("fedcba"));
    return 0;
}

unsigned nstrlen(const char s[])
{
    register unsigned n = 0;

    while(s[n] != '\0'){
       ++n;
    }
    return n;
}
```

The sprintf() Function

In #include <string.h> : This is very similar to the function printf(), except that the output is stored in a string rather than written to the output. It is defined as:

```
int sprintf(string, control-arg, other args);
```

For example:

```
int i = 7;
float f = 17.041;
char str[100];
sprintf(str, "%d %f", i, f);
printf("%s\n", str);
```

Outputs : 7 17.041000

This is useful if you need to create a string for passing to another function for further processing.

The sprintf() Function

In #include <string.h> : This is very similar to the function printf(), except that the output is stored in a string rather than written to the output. It is defined as:

int sprintf(string, control-arg, other args);

For example:

```
int i = 7;
float f = 17.041;
char str[100];
sprintf(str, "%d %f", i, f);
printf("%s\n", str);
```

Outputs : 7 17.041000

This is useful if you need to create a string for passing to another function for further processing.

```c
#define SMALLSTR 20
void print_card(char s[], card c)
{

   char pipstr[SMALLSTR];
   char suitstr[SMALLSTR];
   switch(c.pips){
      case 11:
         strcpy(pipstr, "Jack");
         break;
      case 12:
         strcpy(pipstr, "Queen");
         break;
      case 13:
         strcpy(pipstr, "King");
         break;
      default:
         sprintf(pipstr, "%2d", c.pips);
   }
   switch(c.st){
      case hearts :
         strcpy(suitstr, "Hearts");
         break;
      case diamonds :
         strcpy(suitstr, "Diamonds");
         break;
      case spades:
         strcpy(suitstr, "Spades");
         break;
      default :
         strcpy(suitstr, "Clubs");
   }
   sprintf(s, "%s of %s", pipstr, suitstr);
}
```

sprintf() and sscanf()

```c
#define FIRSTCARD " 1 of Hearts"
void test(void)
{
   int n = 0;
   char str[BIGSTR];
   card d[DECK];
   init_deck(d);
   // Direct assignment
   print_card(str, d[0]);
   // 1st element initialised correctly
   assert(strcmp(str, FIRSTCARD)==0);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      print_card(str, d[0]);
      // Happens 1 time in 52 ?
      if(strcmp(str, FIRSTCARD)==0){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

sprintf() and sscanf()

```c
#define FIRSTCARD " 1 of Hearts"
void test(void)
{
   int n = 0;
   char str[BIGSTR];
   card d[DECK];
   init_deck(d);
   // Direct assignment
   print_card(str, d[0]);
   // 1st element initialised correctly
   assert(strcmp(str, FIRSTCARD)==0);
   for(int i=0; i<1000; i++){
      shuffle_deck(d);
      print_card(str, d[0]);
      // Happens 1 time in 52 ?
      if(strcmp(str, FIRSTCARD)==0){
         n++;
      }
   }
   // Is this a reasonable test ?
   assert((n > 10) && (n < 30));
}
```

```c
// Simple demo of sscanf (and fgets in passing)
#include <stdio.h>
#include <assert.h>

#define BIGSTR 1000
#define SMLSTR 100
#define DAYSINYEAR 365.2425

#include <stdio.h>

int main(void)
{


   printf("Please type your first name and your age\n");
   char bigstr[BIGSTR];
   fgets(bigstr, BIGSTR, stdin);
   char name[SMLSTR];
   int age;
   // Note no "&" before name : passed by reference already
   assert(sscanf(bigstr, "%s %d\n", name, &age)==2);
   printf("%s, you've lived approximately %.0f days\n",
          name, ((double)(age)+0.5)*DAYSINYEAR);
   return 0;

}
```

Execution :

```
Please type your first name and your age
Joe 25
Joe, you've lived approximately 9314 days
```

Table of Contents

Initializing 2D Arrays

A 2D array is declared as follows:

```
#define ROWS 3
#define COLS 5
int a[ROWS][COLS];
```

2D array initialisation :

```
int b[2][3] = {1, 2, 3, 4, 5, 6};
int b[2][3] = {{1, 2, 3}, {4, 5, 6}};
int b[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

Initializing 2D Arrays

A 2D array is declared as follows:

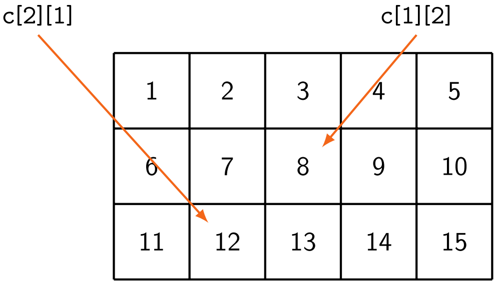
```
#define ROWS 3
#define COLS 5
int a[ROWS][COLS];
```

2D array initialisation :

```
int b[2][3] = {1, 2, 3, 4, 5, 6};
int b[2][3] = {{1, 2, 3}, {4, 5, 6}};
int b[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

Although 2D arrays are stored in a contiguous block of memory, we may think of them as a 2D rectangle of data.

```
int c[3][5] = {{1,2,3,4,5}, {6,7,8,9,10},
               {11,12,13,14,15}};
```



2D Distance

```c
#include <stdio.h>
#include <math.h>

#define  M  7
#define  N  9

int main(void)
{
   int   a[M][N];

   // Fill Array
   for (int j = 0; j < M; ++j){
      double y = ((double)j - ((double)(M-1)/2.0));
      for(int i = 0; i < N; ++i){
         double x = ((double)i - ((double)(N-1)/2.0));
         a[j][i] = round(sqrt(x*x + y*y));
      }
   }

   // Print Array
   for (int j = 0; j < M; j++){
      for(int i = 0; i < N; i++){
         printf("%d", a[j][i]);
      }
      printf("\n");
   }
   printf("\n");

   return 0;
}
```

Execution :

```
544333445
443222344
432111234
432101234
432111234
443222344
544333445
```

Cards (again!)

```c
#define SMALLSTR 20
void print_card(char s[], card c)
{
   // Note the +1 below : zero pips not used, but makes easier coding ?
   char pipnames[PERSUIT+1][SMALLSTR] = {"Zero", "One", "Two", "Three",
                                          "Four", "Five", "Six", "Seven",
                                          "Eight", "Nine", "Ten", "Jack",
                                          "Queen", "King"};
   char suitnames[SUITS][SMALLSTR] = {"Hearts", "Diamonds", "Spades", "Clubs"};
   sprintf(s, "%s of %s", pipnames[c.pips], suitnames[c.st]);
}
```

- The $2D$ arrays of characters here have one string per row.

Cards (again!)

```c
#define SMALLSTR 20
void print_card(char s[], card c)
{
   // Note the +1 below : zero pips not used, but makes easier coding ?
   char pipnames[PERSUIT+1][SMALLSTR] = {"Zero", "One", "Two", "Three",
                                         "Four", "Five", "Six", "Seven",
                                         "Eight", "Nine", "Ten", "Jack",
                                         "Queen", "King"};
   char suitnames[SUITS][SMALLSTR] = {"Hearts", "Diamonds", "Spades", "Clubs"};
   sprintf(s, "%s of %s", pipnames[c.pips], suitnames[c.st]);
}
```

- The $2D$ arrays of characters here have one string per row.
- They are of a fixed-width, sometime called *ragged-right* or *jagged-right* arrays.

Storage Classes

- **auto**

  ```
  auto int a, b, c;
  auto float f;
  ```

  Because this is the default, it is seldom used.

Storage Classes

- **auto**

  ```
  auto int a, b, c;
  auto float f;
  ```

  Because this is the default, it is seldom used.

- **extern**
  Tells the compiler to look for the variable elsewhere, possibly another file.

Storage Classes

- **auto**

  ```
  auto int a, b, c;
  auto float f;
  ```

  Because this is the default, it is seldom used.

- **extern**
  Tells the compiler to look for the variable elsewhere, possibly another file.

- **register**
  Informs the compiler to place the variable in a high-speed memory register if possible, i.e. if there are enough such registers available & the hardware supports this.

Storage Classes

- **auto**

  ```
  auto int a, b, c;
  auto float f;
  ```

  Because this is the default, it is seldom used.

- **extern**
  Tells the compiler to look for the variable elsewhere, possibly another file.

- **register**
  Informs the compiler to place the variable in a high-speed memory register if possible, i.e. if there are enough such registers available & the hardware supports this.

Storage Classes

- **auto**

  auto int a, b, c;

  auto float f;

  Because this is the default, it is seldom used.
- **extern**
  Tells the compiler to look for the variable elsewhere, possibly another file.
- **register**
  Informs the compiler to place the variable in a high-speed memory register if possible, i.e. if there are enough such registers available & the hardware supports this.

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

void printstuff(void);

#define MAXLOOP 20

int main(void)
{
    int r = rand() % MAXLOOP;
    for(int i=0; i<r; i++){
        printstuff();
    }
    return 0;
}

void printstuff(void)
{
    static int cnt = 0;
    printf("You've been here %d times\n", ++cnt);
}
```

Execution :

```
You've been here 1 times
You've been here 2 times
You've been here 3 times
```

Table of Contents

Call-by-Value

```c
#include <stdio.h>

void changex(int x);

int main(void)
{
    int x = 1;

    changex(x);
    printf("%d\n", x);
}

void changex(int x)
{
    x = x + 1;
}
```

Execution :

1

Call-by-Value

```c
#include <stdio.h>

void changex(int x);

int main(void)
{
    int x = 1;

    changex(x);
    printf("%d\n", x);
}

void changex(int x)
{
    x = x + 1;
}
```

Execution :

1

- In the program, the function cannot change the value of `v` as defined in `main()` since a **copy** is made of it.

Call-by-Value

```c
#include <stdio.h>

void changex(int x);

int main(void)
{
    int x = 1;

    changex(x);
    printf("%d\n", x);
}

void changex(int x)
{
    x = x + 1;
}
```

Execution :

1

- In the program, the function cannot change the value of `v` as defined in `main()` since a **copy** is made of it.
- To allow a function to modify the value of a variable passed to it we need a mechanism known as **call-by-reference**, which uses the **address** of variables (pointers).

Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.

Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.

Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.

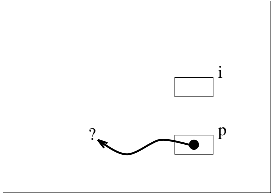
Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.

- `int i, *p;`



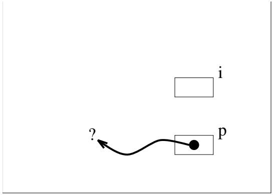
Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.

- `int i, *p;`
- Here `i` is an `int` and `p` is of type *pointer to int*.

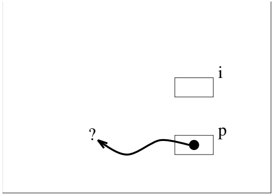


Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

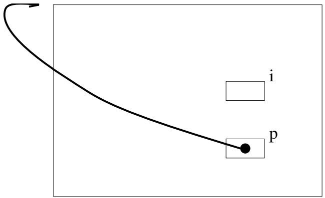
  `scanf("%d", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.
- `int i, *p;`
- Here `i` is an `int` and `p` is of type *pointer to int*.
- Pointers have a legal range which includes the special address 0 and a set of positive integers which are the machine addresses of a particular system.



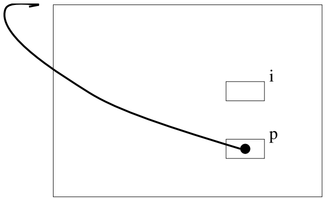
The *NULL* Pointer

- `p = NULL;`



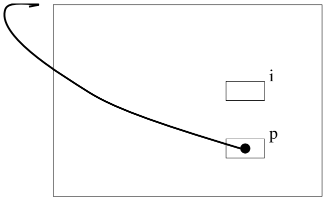
The *NULL* Pointer

- `p = NULL;`

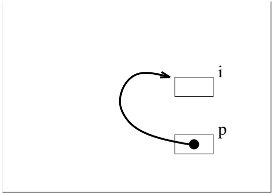


The *NULL* Pointer

- p = NULL;

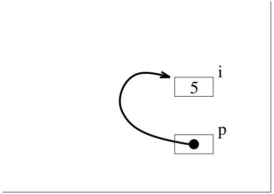


- p = &i;



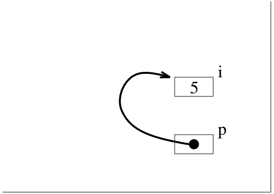
Equivalence of i and $*p$

- i = 5;



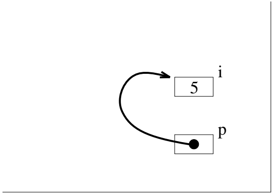
Equivalence of *i* and **p*

- i = 5;



Equivalence of *i* and **p*

- i = 5;




```c
#include <stdio.h>

int main(void)
{

    int i, *p;

    p = &i;
    i = 5;
    printf("%i\n", *p);
    i = 17;
    printf("%i\n", *p);
    *p = 99;
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
5
17
99
```

scanf Again

```c
#include <stdio.h>

int main(void)
{

    int i, *p;

    p = &i;
    printf("Please Type a number : ");
    scanf("%i", &i);
    printf("%i\n", i);
    printf("Please Type a number : ");
    scanf("%i", p);
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
Please Type a number : 70
70
Please Type a number : 3
3
```

scanf Again

```c
#include <stdio.h>

int main(void)
{

    int i, *p;

    p = &i;
    printf("Please Type a number : ");
    scanf("%i", &i);
    printf("%i\n", i);
    printf("Please Type a number : ");
    scanf("%i", p);
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
Please Type a number : 70
70
Please Type a number : 3
3
```

- In many ways the dereference operator * is the inverse of the address operator &.

```c
float x = 5, y = 8, *p;
p = &x;
y = *p;
```

scanf Again

```
#include <stdio.h>

int main(void)
{

    int i, *p;

    p = &i;
    printf("Please Type a number : ");
    scanf("%i", &i);
    printf("%i\n", i);
    printf("Please Type a number : ");
    scanf("%i", p);
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
Please Type a number : 70
70
Please Type a number : 3
3
```

- In many ways the dereference operator * is the inverse of the address operator &.

```
    float x = 5, y = 8, *p;
p = &x;
y = *p;
```

- What is this equivalent to ?

The *swap* Function

```
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
   int   a = 3, b = 7;

   // 3 7 printed
   printf("%i %i\n", a, b);
   swap(&a, &b);
   // 7 3 printed
   printf("%i %i\n", a, b);
   return 0;
}

void swap(int *p, int *q)
{
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Execution :

3 7
7 3

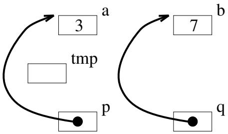
The *swap* Function

```c
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
    int   a = 3, b = 7;

    // 3 7 printed
    printf("%i %i\n", a, b);
    swap(&a, &b);
    // 7 3 printed
    printf("%i %i\n", a, b);
    return 0;
}

void swap(int *p, int *q)
{
    int   tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

Execution :

3 7
7 3

- At beginning of function:



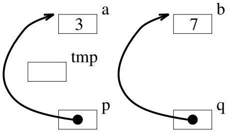
The *swap* Function

```c
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
    int   a = 3, b = 7;

    // 3 7 printed
    printf("%i %i\n", a, b);
    swap(&a, &b);
    // 7 3 printed
    printf("%i %i\n", a, b);
    return 0;
}

void swap(int *p, int *q)
{
    int   tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

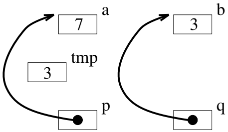
Execution :

3 7
7 3

- At beginning of function:



- At end of function:



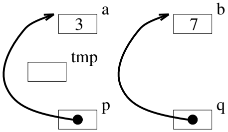
The *swap* Function

```c
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
    int    a = 3, b = 7;

    // 3 7 printed
    printf("%i %i\n", a, b);
    swap(&a, &b);
    // 7 3 printed
    printf("%i %i\n", a, b);
    return 0;
}

void swap(int *p, int *q)
{
    int    tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

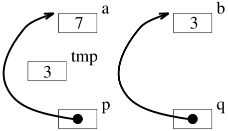
Execution :

3 7
7 3

- At beginning of function:



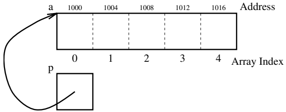
- At end of function:



- Remeber that the variables *a* and *b* are not in the scope of *swap()*.

Arrays are Pointer?

- An array name by itself is simply an address.

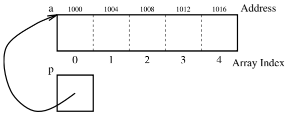


Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

  ```
  int a[5];
  int *p;
  ```

  declares an array of 5 elements, and a is the address of the start of the array.



Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

  int a[5];

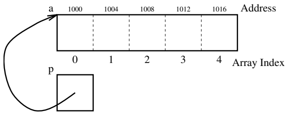
  int *p;

  declares an array of 5 elements, and a is the address of the start of the array.
- Assigning:

  p = a;

  is completely valid and the same as:

  p = &a[0];



Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

  int a[5];

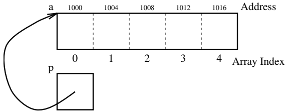
  int *p;

  declares an array of 5 elements, and a is the address of the start of the array.
- Assigning:

  p = a;

  is completely valid and the same as:

  p = &a[0];



Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

  int a[5];

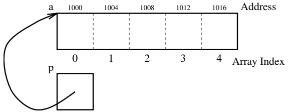
  int *p;

  declares an array of 5 elements, and a is the address of the start of the array.
- Assigning:

  p = a;

  is completely valid and the same as:

  p = &a[0];



- To assign p to point to the next element, we could either :

  p = a + 1;

  p = &a[1];

Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

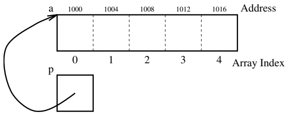
  int a[5];
  int *p;

  declares an array of 5 elements, and a is the address of the start of the array.
- Assigning:

  p = a;

  is completely valid and the same as:

  p = &a[0];



- To assign p to point to the next element, we could either :

  p = a + 1;
  p = &a[1];
- Notice that p = a + 1 advances the pointer **4** bytes and not 1 byte. This is because an integer is 4 bytes long and p is a pointer to an int.

Arrays are Pointer?

- An array name by itself is simply an address.
- For instance:

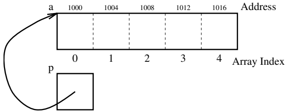
  `int a[5];`
  `int *p;`

  declares an array of 5 elements, and `a` is the address of the start of the array.
- Assigning:

  `p = a;`

  is completely valid and the same as:

  `p = &a[0];`



- To assign `p` to point to the next element, we could either :

  `p = a + 1;`
  `p = &a[1];`
- Notice that `p = a + 1` advances the pointer **4** bytes and not 1 byte. This is because an integer is 4 bytes long and `p` is a pointer to an int.
- we can use the pointer `p` is exactly the same way as normal, i.e.:

  `*p = 5;`

Summing an Array

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

   int n[NUM] = {10, 12, 6, 7, 2};

   printf("%i\n", sum(n));
   return 0;
}

int sum(int a[])
{
   int sum = 0;

   for(int i=0; i<NUM; i++){
      sum += a[i];
   }
   return sum;
}
```

Execution :

37

Summing an Array

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += a[i];
    }
    return sum;
}
```

Execution :

37

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += *(a + i);
    }
    return sum;
}
```

Execution :

37

Summing an Array

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += a[i];
    }
    return sum;
}
```

Execution :

37

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += *(a + i);
    }
    return sum;
}
```

Execution :

37

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;
    int *p = a;

    for(int i=0; i<NUM; i++){
        sum += *p;
        p++;
    }
    return sum;
}
```

Execution :

37

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.
- But, like any other type, we could pass a pointer instead.

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.
- But, like any other type, we could pass a pointer instead.
- The complication is that to access the elements of a structure via a pointer, we use the “->” operator, and not the “.”.

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.
- But, like any other type, we could pass a pointer instead.
- The complication is that to access the elements of a structure via a pointer, we use the “->” operator, and not the “.”.

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.
- But, like any other type, we could pass a pointer instead.
- The complication is that to access the elements of a structure via a pointer, we use the "->" operator, and not the ".".

```c
void print_deck(card d[DECK], int n)
{
   char str[BIGSTR];
   for(int i=0; i<n; i++){
      print_card(str, &d[i]);
      printf("%s\n", str);
   }
   printf("\n");
}

#define SMALLSTR 20
void print_card(char s[], const card* p)
{
   // Note the +1 below : zero pips not used, but makes easier coding ?
   char pipnames[PERSUIT+1][SMALLSTR] = {"Zero", "One", "Two", "Three",
                                        "Four", "Five", "Six", "Seven",
                                        "Eight", "Nine", "Ten", "Jack",
                                        "Queen", "King"};
   char suitnames[SUITS][SMALLSTR] = {"Hearts", "Diamonds", "Spades", "Clubs"};
   sprintf(s, "%s of %s", pipnames[p->pips], suitnames[p->st]);
}
```