Programming in C

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

University of Bristol

June 22, 2021

Table of Contents

1. A: Preamble
2. B: Hello, World
3. C: Grammar
4. D: Flow Control
5. E: Functions

About the Course

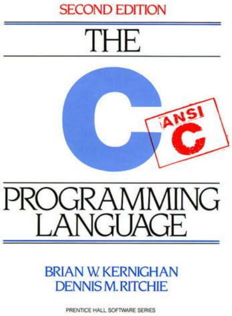
These course notes were originally based on :

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
- A list of more : https://www.linuxlinks.com/excellent-free-books-learn-c/
- Whatever you use, make sure it's **ANSI C** or **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.


SECOND EDITION

THE C PROGRAMMING LANGUAGE

ANSI C

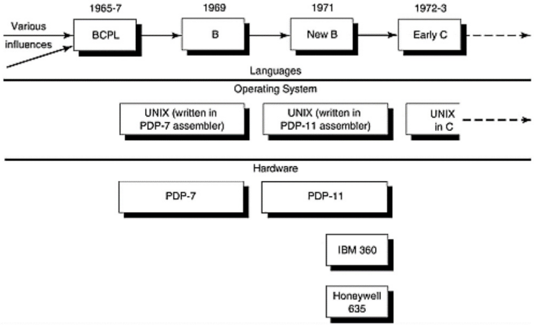
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

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.

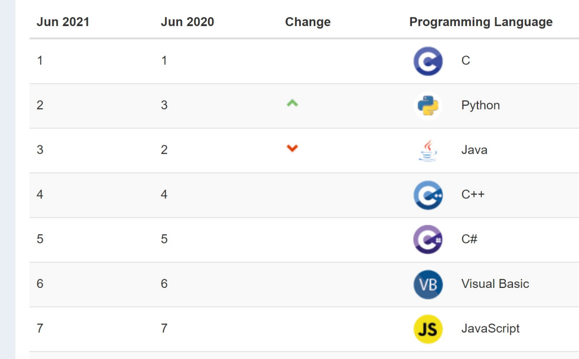
History of C



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

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

Why C ?



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

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

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

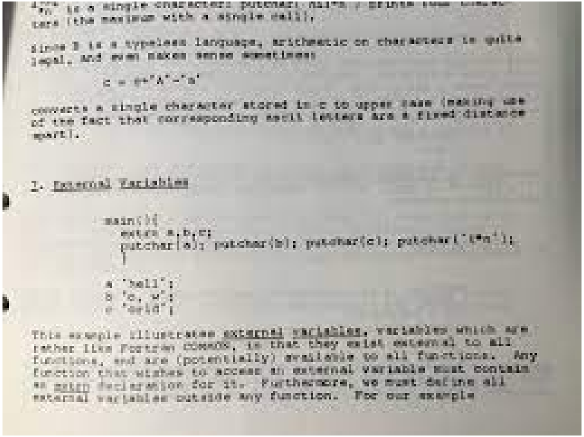
- Any problems with the computers e.g. installing the correct S/W, accessing lab machines : `http://www.bris.ac.uk/it-services/`.
- They are also the people to see about passwords etc.
- This page also links to the rather useful Laptop & Mobile Clinic.

Help with the Unit

- Further information is available via the Blackboard site.
- Help will mainly be via myself giving ‘live’ Q&A session, the associated MS Teams group and the corresponding Forum.
- You will often work in a peer group (approx 15 people).
- There will be a group of Teaching Assistants to help each of these groups.
- 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

```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;

}
```

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;
}
```

Output :

```
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.
- #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

- + , - , / , *, %
- 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

```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;
}
```

- 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.
- Output :

```
A B
```

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;

}
```

Output :

```
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.
- `#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

- 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*.
- `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;
}
```

```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%12f\n\n",
                "Count:", cnt, " Sum:", sum);
    return 0;
}
```

Common Mistakes

- Missing "

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

- Missing ;

```
a = a + 1
```

- Missing Address in scanf()

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

Table of Contents

1. A: Preamble
2. B: Hello, World
3. C: Grammar
4. D: Flow Control
5. E: Functions

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*.
- $1 + 2 * 3$ 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;

```c
#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.
- 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;
}
```

Output :

```
    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;
}
```

```
Randomly distributed integers will be 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

1. A: Preamble
2. B: Hello, World
3. C: Grammar
4. D: Flow Control
5. E: Functions

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.

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;
}
```

Output:

```
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.

```
// Simple while countdown

#include <stdio.h>

int main(void)
{

   int n = 9;

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

Output :

```
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;
}
```

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

 ... etc ...

  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.

```
// 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;
}
```

Output :

```
9 8 7 6 5 4 3 2 1
```

The switch() Statement

```c
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

```c
/* 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;
}
```

Output :

```
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.

```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;
   min = (x < y) ? x : y;
   min = (z < min) ? z : min;
   printf("The minimum value is %d\n", min);
   return 0;
}
```

Table of Contents

1. A: Preamble

2. B: Hello, World

3. C: Grammar

4. D: Flow Control

5. E: Functions

Simple Functions

```c
#include <stdio.h>

int min(int a, int b);

int main(void)
{
   int   j, k, m;

   j = 6;
   k = 9;
   m = min(j, k);
   printf("Minimum of %d & %d = " \
          "%d\n", j, k, m);
   return 0;
}

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

Output :

```
Minimum of 6 & 9 = 6
```

- 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 `min()` returns an `int` and takes two `int`'s as arguments.
- The function is defined between two braces.
- The `return` statement is used return a value to the calling statement.
- A function which has no return value, is declared `void` and is equivalent to a procedure.

Assert

The assert macro is defined in the header file `assert.h`. This is used to ensure the value of an expression is as we expect it to be.

Assert

```
#include <assert.h>

double f(int a, int b)
{

   double x;

   assert(a > 0);

   /* precondition */
   assert(b >= 7 && b <= 11);

      .
      .
      .

   /* postcondition */
   assert(x >= 1.0);
```

Program Layout

It is common 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)
{
```

Call-by-Value

However, it is 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)
{
```

Call-by-Value

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

```
#include <stdio.h>

void fnc1(int a);

int main(void)
{

   int x = 1;

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

}
```

Call-by-Value

```
void fnc1(int a)
{

  a = a + 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.

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 `assert()` calls in `main()` test it thoroughly.

Recursion

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

```
#include <stdio.h>

int fact(int a);

int main(void)
{

   int a, f;

   printf("Input a number :\n");
   scanf("%d", &a);
   f = fact(a);
   printf("%d! is %d\n", a, f);

   return(0);

}
```

Call-by-Value

```
int fact(int a)
{

  int i;
  int tot = 1;

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

We could also achieve this via *recursion* :

Call-by-Value

```
#include <stdio.h>

int fact(int a);

int main(void)
{

   int a, f;

   printf("Input a number :\n");
   scanf("%d", &a);
   f = fact(a);
   printf("%d! is %d\n", a, f);

   return(0);
}

int fact(int a)
```