



C Programs

- **Objective**
 - To look at C as a language
- **Contents**
 - Characteristics of C
 - C software development lifecycle
 - The elements of a C program
 - Identifiers and keywords
 - Declarations and expressions
 - Layout and guidelines
- **Summary**
- **Practical**
 - To study, build, link and run a small working program



The objective of this chapter is to cover enough of the language to enable you to have a 'hands-on' practical session. The topics include program structure, simple input and output, the definition and manipulation of data and the C programming lifecycle. During the discussion of the lifecycle, we emphasise the implementation stage and mention some of the current tools available.

C_Programming

What is C ?

C is a general purpose programming language

C has...

1. economy of expression
2. a modern control flow and data structures
3. a rich set of operators

C is not...

1. a very high-level language
2. a big language
3. specialised to any area of application

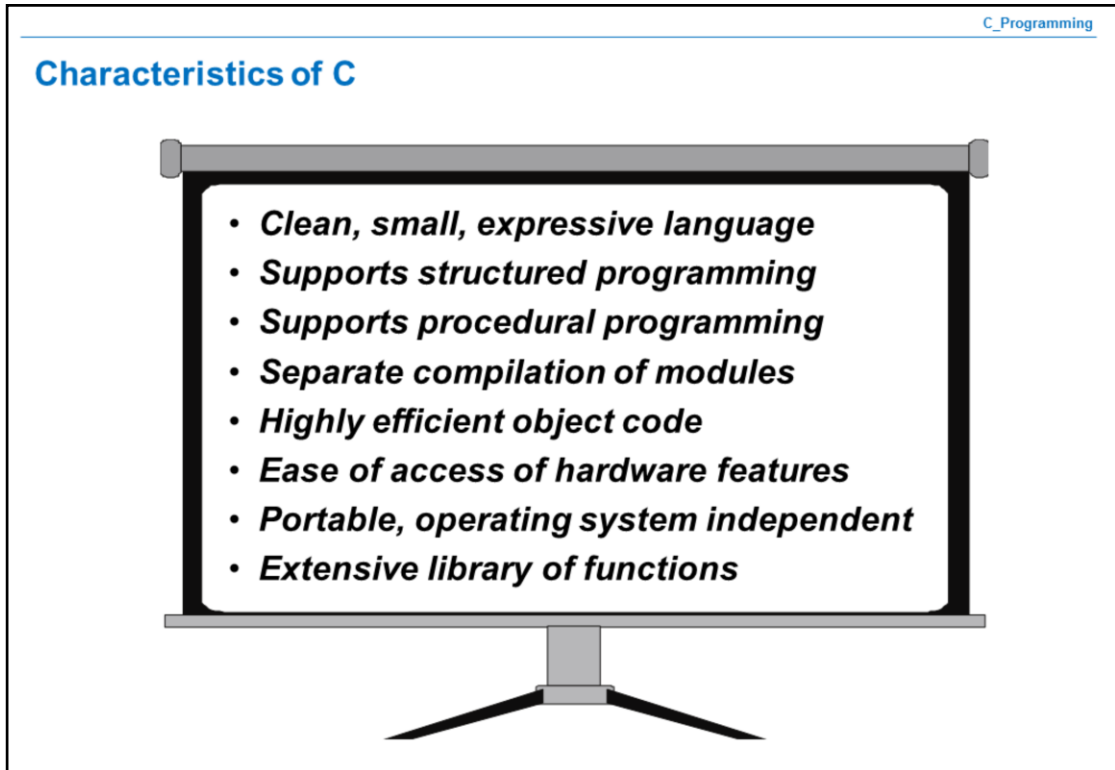
***Dennis M. Ritchie
Brian W. Kernighan***

C is a high-level language (HLL) designed to help the development of Unix. It was created in order to enable Unix to be the first portable operating system.

The language itself was very much like the other HLLs of the time, but with rather a terse syntax. It gained initial popularity as a systems language because of its low-level facilities. It was, however, high level and had the data types, control flow and modularity of its rivals.

It grew to be a popular language throughout the academic world because of its connection with Unix. Its powerful features soon became known to a wider circle, and non-Unix systems started to attract C compiler writers. By the late '70s, C had become known as a portable language, since a compiler was found on a variety of systems.

As C gradually found its way into the business environments, it extended its status as a systems programming language, to become an applications language. Since the beginning of the '80s, it has come from being one of the more-popular applications languages, to become the most popular.



The characteristics of the language listed above are fairly self-explicit. Further details are included here.

C has 32 keywords, three decision-making and three looping constructs. It is block structured and is based around '{' and '}' BEGIN/END tokens. It supports a spectrum of data manipulation, from bit through to record/field handling, using 45 operators.

A module, containing a logical part of the complete program, is held in a single file and can be compiled as a stand-alone unit.

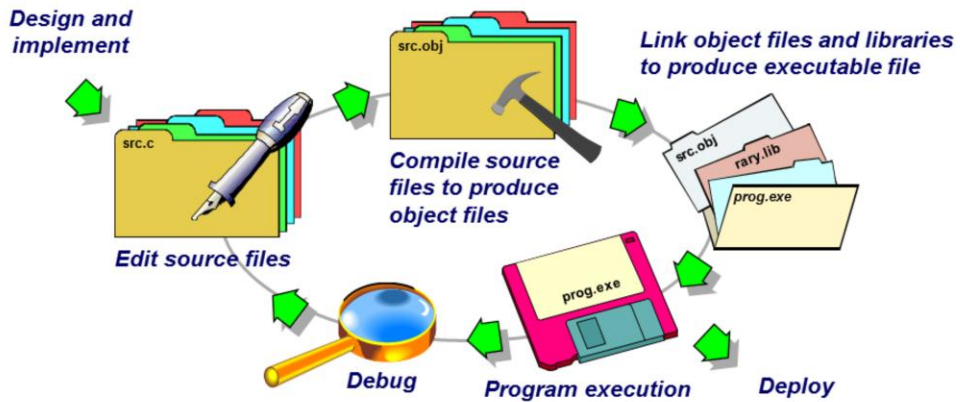
Constructs within the language lend themselves quite naturally to small and fast executable code. There is support (usually non-portable) for access to the underlying hardware and software. There is, however, much support for the writing of portable code. This includes the use of the standard library, modular programming and support provided by the preprocessor.

The library contains over one hundred standard routines. The library usually also includes other useful routines; possibly several hundred. A full list is found at the end of the *Working With Large Programs* chapter.

The 'Write-Only' reputation was gained during C's infancy, when systems programmers and blatant hackers(!) used C's free-style formatting and terse syntax to the full - for whatever reason! This reputation is fast losing ground as C coding standards have emerged, and better design practices have been adopted.

The C Software Development Lifecycle

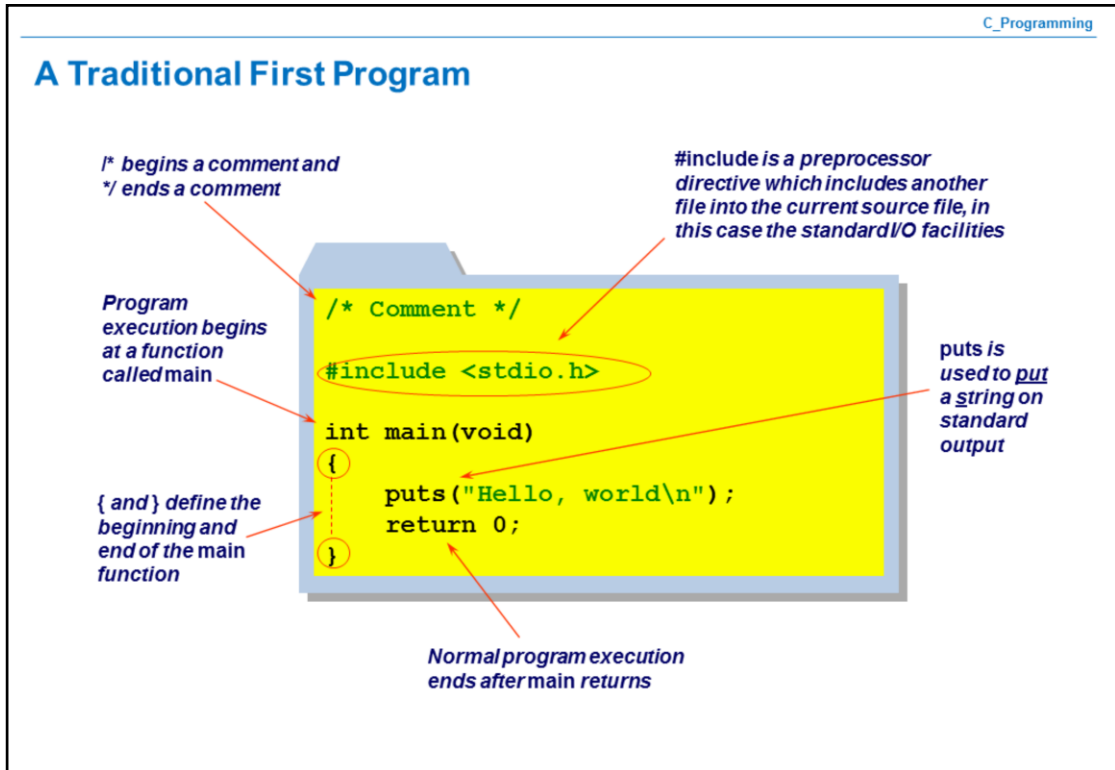
- C is a compiled language that follows a traditional *edit-compile-link-debug* cycle



The C development lifecycle is fairly typical. A C source file is compiled, and if there are no errors, an object file is created, typically with either an `.obj` or `.o` extension. Errors at this stage are usually pure language syntax errors.

The object file is the input to the linker. It is rare that there is only one input file. The linker requires enough input to create a complete executable. This usually consists of startup code, one or more libraries and enough object files to create a complete program. Errors at this stage fall into two categories: missing or misspelt functions and incorrect search-path information. If all is well, out pops an executable, usually with an extension (if any) of `.exe` or `.com`, depending on your platform.

It is at this stage that problems really occur! Runtime errors are notoriously the worst, but at least there are debugging tools available.



As a function, `main` has a return value that is typically used to communicate the status of the program's run back to the calling environment, i.e. the operating system. A return value 0 is used to signify success.

The main function may also be defined like this:

```
int main(int argc, char * argv[])  
{  
    ...  
}
```

The parameters (see *Functions* chapter) `argc` and `argv` may be used to refer to command line arguments.

Adding Two Numbers

{ defines the start of a block

Declarations introduce variables at the start of a block

scanf is an input function

printf is an output/print function

} defines the end of a block

```
/* A program to add two integers */  
#include <stdio.h>  
  
int main(void)  
{  
    int first, second, result;  
    puts("Please enter two integers");  
    scanf("%d %d", &first, &second);  
  
    result = first + second;  
    printf("Their sum is %d\n", result);  
    return 0;  
}
```

Simple statements are terminated with a semi-colon

Note that comments cannot be nested:

```
/* the programmer has attempted to embed  
   /* this comment */ within this comment  
   , but the italicized portion is outside  
   and will cause a compile error */
```

Statements may appear anywhere within a block and are executed in order. Variables must be declared before use. A declaration is considered to be a statement and must appear at the start of a block. Variables are visible from their point of declaration onwards to the end of their enclosing block.

Identifiers

- Identifiers name program elements

- Identifier rules

- Made of letters and digits
- Must start with a letter
- Underscore is considered a letter
- Case sensitive

answer_42 ✓
42_answer ✗

Compiler error

different ✓
Different; ✓

Names of functions

```
/* A program to add two integers */  
#include <stdio.h>  
int main(void)  
{  
    int first, second, result;  
    /*...*/  
    printf("Their sum is %d\n", result);  
    return 0;  
}
```

Names of variables

Like a number of other languages, C is case sensitive. This means that the case of letters in identifiers is significant, and not simply the spelling. For instance, `different` and `Different` are different. This is different to case-free languages such as Pascal and Basic.

The identifier `42_answer` is not legal because it starts with a digit.

The example above contains many identifiers, most of which are in bold. However, if you look closely you will see that there are some identifiers that are not in bold. For example, `int` and `return`. As we will see next, these are special identifiers. They are reserved identifiers called keywords.

Keywords

- Keywords are reserved identifiers
 - Keywords have *fixed* meanings
 - The name of a variable/function cannot be a keyword

```
/* A program to add two integers */
#include <stdio.h>

int main(void)
{
    int first, second, result;
    puts("Please enter two integers");
    scanf("%d %d", &first, &second);

    result = first + second;
    printf("Their sum is %d\n", result);
    return 0;
}
```

types

```
char
int
double
...
```

flow control

```
return
if
else
for
...
```

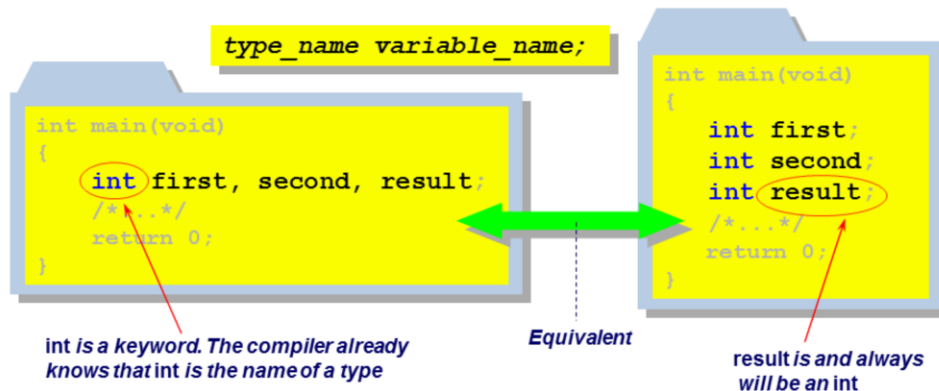
more...

Keywords are reserved identifiers. Keywords have a fixed meaning as defined by the C language standard. Keywords are the fundamental elements on which C is constructed. Keywords may not be used as variable or function names: the compiler will flag an error should you attempt to do so. In many editing environments keywords are highlighted in a different colour, font or style. This is known as *syntax highlighting*. Comments are often marked in a different style to executable code. Standard C has 32 keywords. Not all of these will be covered on the course.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Declarations

- A declaration introduces a named variable into a scope
 - All variables have a *name* and a *type*
 - The *type* must be specified and cannot change
 - C is a statically typed language



A declaration introduces a named variable into a scope. In the example above the objects named `first`, `second` and `result` are declared in the first statement (in bold). As we saw on the previous slide, `int` is a keyword; it is the name of one of built-in types; it is a type name. Hence all three objects are ints, and will remain ints for their entire lifetime. Variables are visible from their point of declaration onwards to the end of their enclosing block.

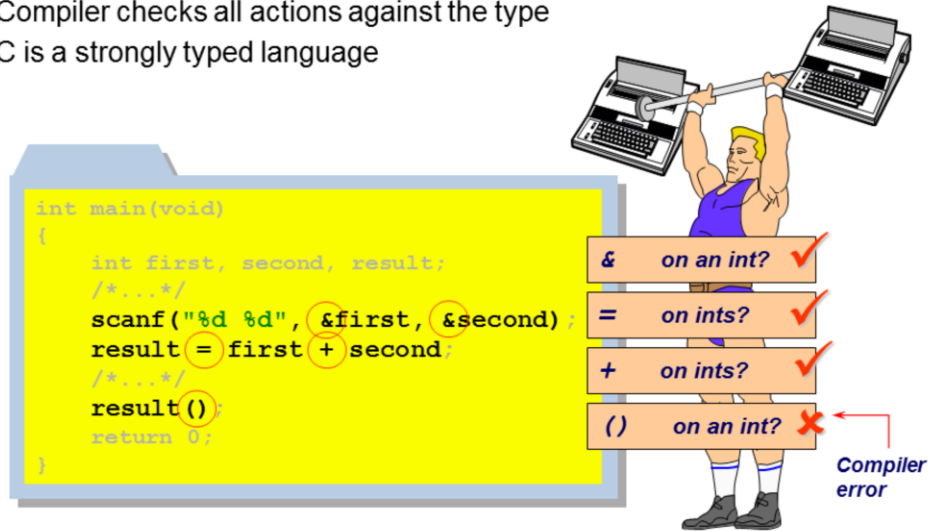
The essence of a declaration is that the leftmost identifier is the name of a type. We will see in later chapters that C has mechanisms to define and name new user-defined types. For example if the identifier `date` (which is not a keyword) had been defined to be the name of a user-defined type, then

```
date today;
```

would be a declaration of a variable called `today` of type `date`.

Expressions

- **An expression is an action on a declared variable**
 - Actions are given symbols called operators
 - Compiler checks all actions against the type
 - C is a strongly typed language



In the code box above, the compiler checks the statement

```
result = first + second;
```

First the compiler checks the sub-expression

```
first + second
```

It knows that `first` and `second` are ints. It checks that operator `+` can be used in this context. It can, so this sub-expression is legal.

The compiler then checks the full-expression

```
result = sub-expression
```

It knows that `result` is an int, and it also knows the type of the sub-expression is also an int. It checks that operator `=` can be used in this context. It can, so the full expression is legal, hence the entire statement is legal.

Later the compiler checks the statement

```
result();
```



It knows that `result` is still an int. It checks that operator `()` can be used in this context. It cannot, so the statement is illegal and the compiler issues an error message.

Strong typing is natural. It is the way human beings think and work. You can press play on a video recorder because it is a video recorder and it supports that behaviour. You can't press play on a pen because, well, it's a pen, and it doesn't support that behaviour. The type of a thing determines the behaviour of that thing.

Exercise: Spot the Bugs

Which of the following are not legal identifiers?

```
first
i
return
stdio
MAX_SIZE
Employee-Number
room_101
main
Int
_exit
value_
999Emergency
$sys_dev
EndOfFile
```



```
include <stdio.h>

int main(void)
{
    puts("Please enter a number");
    scanf("%d", &input);

    puts('Square is ');
    printf("%d\n", input * input);

    return 0
}
```

What syntax problems are there with the above program?

The only identifiers in the list above that are not legal are: `return` because it is a keyword; `Employee-Number`, because it contains `-` which the compiler will interpret as the minus sign; `999Emergency`, because it starts with a number; and `$sys_dev`, because it contains `$` which is not a legal identifier character (note that on some systems it is allowed as an extension).

Although `stdio` is used in a `#include` directive, this is the name of a header file rather than a part of the language. `main` is a normal identifier rather than a keyword, but when used as a function name it has special significance to the linker as the program's entry point. Names with leading underscores are strictly speaking legal, but are reserved and so should not be used.

Here is a corrected version of the source code, with the changes marked in *italics*:

```
#include <stdio.h>

int main(void)
{
    int input;
    puts("Please enter a number: ");
    scanf("%d", &input);

    puts("Square is ");
    printf("%d\n", input * input);

    return 0;
}
```

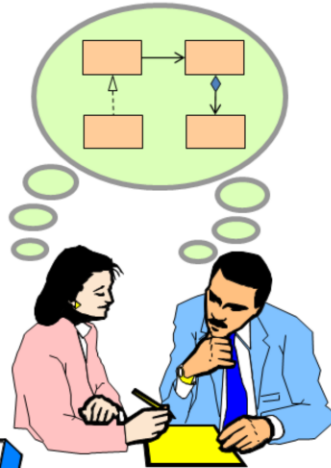
Layout

- **C is a free format language**
 - Spacing is not important to the compiler
 - Spacing is very important to people
 - Guidelines help make programs readable

```
int main(void)
{
    ...
    if (input < 0)
    {
        ...
    }
    ...
}
```



0110101011010



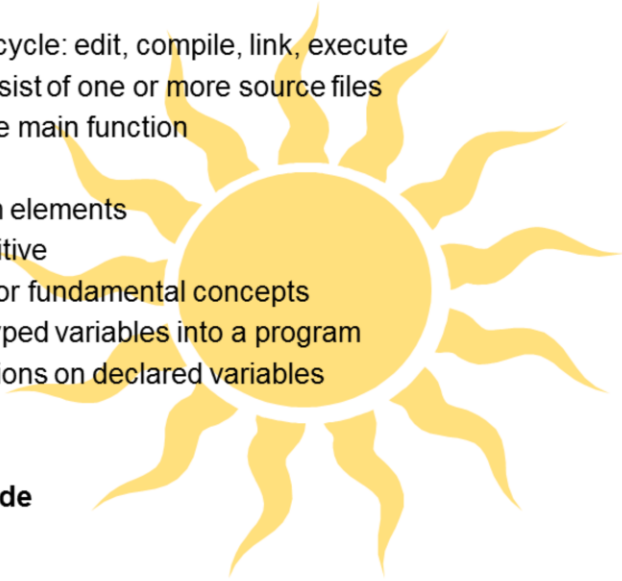
It is important to remember that C is a *free format* language, which means that the layout is not significant to the compiler when it comes to distinguishing one statement from the next. Statements must be explicitly terminated using a semi-colon. Any statements that run over a single line do not require a continuation character to indicate the overrun.

There is a limitation that identifiers, keywords and literals cannot be broken over new lines.

Although the language and compiler are not sensitive to spacing, human readers of the code certainly are! Common spacing and indentation conventions are worth following to ensure that the code is consistent, readable and maintainable. Some initial guidelines are given at the end of this chapter.

Summary

- **Programs**
 - Classic development lifecycle: edit, compile, link, execute
 - Programs physically consist of one or more source files
 - Execution begins with the main function
- **Syntax**
 - Identifiers name program elements
 - Identifiers are case sensitive
 - Keywords are reserved for fundamental concepts
 - Declarations introduce typed variables into a program
 - Expressions perform actions on declared variables
- **Layout**
- **Free format**
- **Remember people read code**



Guidelines

```
#include <stdio.h>

int main(void)
{
    int input;

    puts("Please enter an integer");
    scanf("%d", &input);

    printf("The number was %d\n", input);

    return 0;
}
```

- *Have no more than one simple statement per line*
- *Use blank lines between functions and logically grouped sections of code*
- *Use indentation consistently*
- *Use space around binary operators*

Can you see how these guidelines are applied in the above code?



Consistent layout and spacing conventions help to make code more readable and maintainable.

Multiple statements per line can be confusing. Multiple declarations of the same type can be placed on the same line, but again too many of these can reduce readability. It is often suggested that a declaration should declare only one variable.

Consistent indentation of code means that the source layout reflects the actual runtime flow of the code, making it easier to read and trace through. Indentation is commonly used with statements that continue over one line, as in the above example. Comments are normally indented to match the indentation level of the surrounding statements.

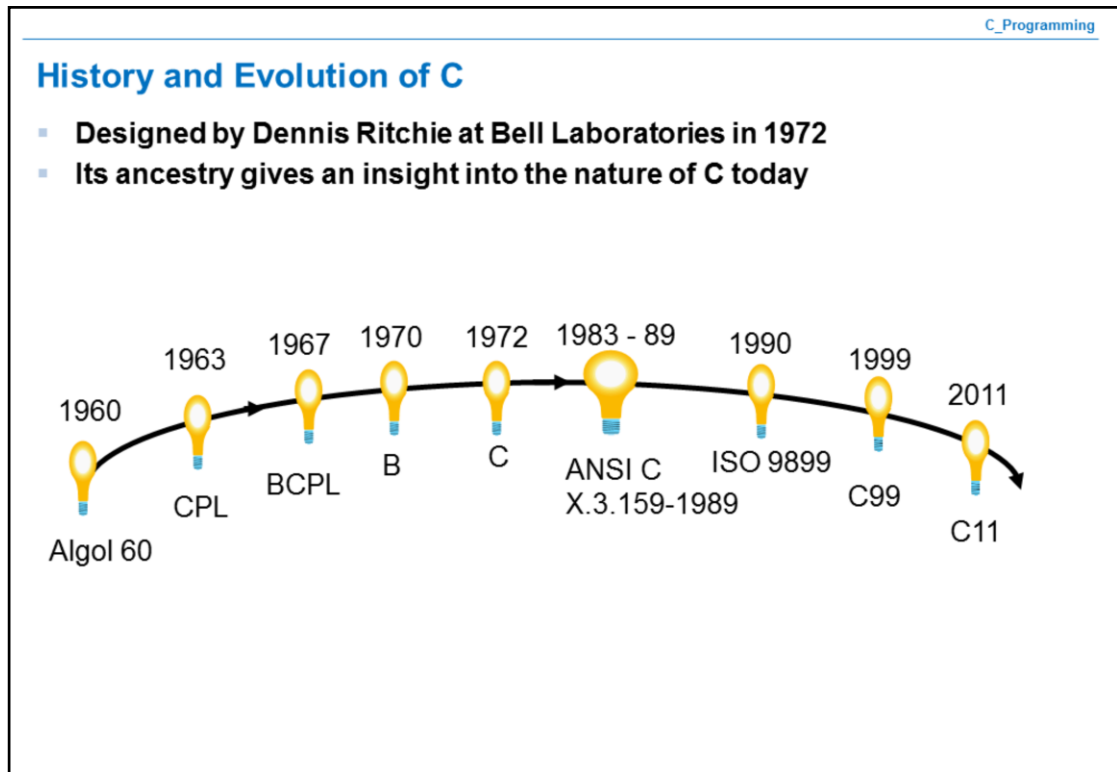
Spacing around binary operators, such as + and -, tends to make expressions clearer:

```
area=length*width;      /* unclear */
area = length * width; /* recommended */
```

Conversely, space is not normally used around parentheses or unary operators:

```
input = - input; /* too much space */
input = -input;  /* recommended */

result = ( 2 + 1 ) * 3; /* too much space */
input = (2 + 1) * 3;    /* recommended */
```



C's major ancestor was the BCPL language; a British language that was basically an assembler language with high-level language syntax. BCPL took many of its constructs and syntax from Algol. C, as it is today, first came to light in 1972; the same year as Pascal!

For over a decade, it ran wild! Every machine had its own version. Only the Unix versions seemed to be controlled in any way. By the time the American National Standards Institute (ANSI) Committee got its hands on it, the problem seemed insurmountable. The creators Kernighan and Ritchie kept tight control of the language, which meant that there was a 'standard' known affectionately as K&R C. Starting from this standard, a new, more realistic and robust language emerged. It took six years to reach a conclusion, but even then, many people thought that the language was still too lax, and others thought that the changes were too drastic.

The ANSI C (also known as C89) was soon to be superseded by ISO in 1990, i.e. the C standard is now accepted in all countries that subscribe to ISO, the International Standards Organisation. The C standard is also known as BS EN ISO 9899, which indicates its acceptance as a British, European and international standard. There was an addendum to this known as the Normative Addendum 1 (NA1 ratified in 1995).

C99 just made it weeks before the millenium. This course does not cover this standard, but is covered in detail in QA's Advanced C course.

The ISO C++ standard was ratified in 1998, and that C++ is *not* a strict super set of C.

The current (mid 2013) C programming language standard (C11) [ISO/IEC 9899](#) was adopted by ISO and IEC in 2011.