To get a deep understanding of C and C++, it is useful to know the history of these wonderful programming languages. It is perhaps even more important to appreciate the driving forces, motivation and the spirit that has shaped these languages into what we have today.

In the first half of this talk we go back to the early days of programmable digital computers. We will take a brief look at really old machine code, assembler, Fortran, IAL, Algol 60 and CPL, before we discuss the motivations behind BCPL, B and then early C. We will also discuss influential hardware architectures represented by EDSAC, Atlas, PDP-7, PDP-11 and Interdata 8/32. From there we quickly move through the newer language versions such as K&R C, C89, C99 and C11.

In the second half we backtrack into the history again, now including Simula, Algol 68, Ada, ML, Clu into the equation. We will discuss the motivation for creating C++, and with live coding we will demonstrate by example how it has evolved from the rather primitive “C with Classes” into a supermodern and capable programming language as we now have with C++11/14 and soon with C++17.

A 90 minute session at ACCU 2015, April 23, Bristol, UK
Part I

History and spirit of C
- The short version
- Before C
- Early C and K&R
- ANSI C
- Modern C
- Q&A

Part II

History and spirit of C++
- Before C++
- Developing the initial versions of C++ (pre-1985)
- Development of C++ (after-1985)
- Evolution of C++ by examples
Part I

History and spirit of C
- The short version
- Before C
- Early C and K&R
- ANSI C
- Modern C
- Q&A

(~90 minutes)

Part II

History and spirit of C++
- Before C++
- Developing the initial versions of C++ (pre-1985)
- Development of C++ (after-1985)
- Evolution of C++ by examples

(a few minutes)
Part I

History and spirit of C
- The short version
- Before C
- Early C and K&R
- ANSI C
- Modern C
- Q&A

(≈90 minutes)

Part II

History and spirit of C++
- Before C++
- Developing the initial versions of C++ (pre-1985)
- Development of C++ (after-1985)
- Evolution of C++ by examples

(a few minutes)
To get a deep understanding of C, it is useful to know the history of this wonderful programming language. It is perhaps even more important to appreciate the driving forces, motivation and the spirit that has shaped the language into what we have today.

In this talk we go back to the early days of programmable digital computers. We will take a brief look at really old machine code, assembler, Fortran, IAL, Algol 60 and CPL, before we discuss the motivations behind BCPL, B and then early C. We will also discuss influential hardware architectures represented by EDSAC, Atlas, PDP-7, PDP-11 and Interdata 8/32. From there we quickly move through the newer language versions such as K&R C, C89, C99 and C11.

A ~90 minute session at ACCU 2015, April 23, Bristol, UK
This is based on research partly done together with Jon Jagger
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C".

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer
- ENIAC
  - The entry keyword
  - Influence from Smalltalk
  - Summer of ‘69
  - Ada Lovelace
  - DEC PDP-8
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

ENIAC
The entry keyword
Influence from Smalltalk
Summer of ’69
Ada Lovelace
DEC PDP-8
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer

- ENIAC
- The entry keyword
- Influence from Smalltalk
- Summer of ’69
- Ada Lovelace
- DEC PDP-8
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer

- ENIAC
- The entry keyword
- Influence from Smalltalk
- Summer of ’69
- Ada Lovelace
- DEC PDP-8
Let’s play bingo! Write down 7 words/names/concepts/whatever that you expect/hope to be discussed/mentioned in a talk titled “History and Spirit of C”.

- Unix
- Dennis Ritchie
- BCPL
- K&R
- ANSI C
- Portability
- Trust the programmer

- ENIAC
- The entry keyword
- Influence from Smalltalk
- Summer of ’69
- Ada Lovelace
- DEC PDP-8

3
The history of C
The history of C in 90
The history of C in 90 seconds
At Bell Labs.
Back in 1969.
Ken Thompson wanted to play.
Ken Thompson wanted to play.
He found a little used PDP-7.
Ended up writing a nearly complete operating system from scratch.
In about 4 weeks.

“Essentially one person for a month, it was just my self.”

(Ken Thompson, 1989 Interview)
In pure assembler of course.

```
GO,
  LAS
  SPA 'CMA
  JMP GO
  DAC CNTSET
  LAC ONE
  DAC BIT
  CLL  /CLEAR THE LINK

LOOP,
  LAC CNTSET
  DAC CNT
  LAC BIT

LOOPI,
  ISZ CNT
  JMP LOOP1
  RAL
  DAC BIT
  LAS
  SMA  /IF ACS0=1, RESET TIME CONSTANT
  JMP LOOP
  JMP GO

/STORAGE FOR PROGRAM DATA
CNT,  0
BIT,  0
CNTSET,  0
ONE,  1
START GO
```
Dennis Ritchie soon joined the effort.
While porting Unix to a PDP-11
While porting Unix to a PDP-11

Ken
While porting Unix to a PDP-11

Dennis

Ken
they invented C,

```c
main( ) {
    printf("hello, world");
}
```
heavily inspired by Martin Richards’ portable systems programming language BCPL.

GET "LIBHDR"
LET START() BE WRITES("Hello, World")
In 1972 Unix was rewritten in C,
and later ported to many other machines
aided by Steve Johnson's Portable C Compiler.
C also gained popularity outside the realm of PDP-11 and Unix.

K&R (1978)
Initially K&R was the definitive reference until the language was standardized by ANSI and ISO in 1989/1990, and thereafter updated in 1999 and 2011.
At Bell Labs. Back in 1969. Ken Thompson wanted to play. He found a little used PDP-7. Ended up writing a nearly complete operating system from scratch. In about 4 weeks. In pure assembler of course. Dennis Ritchie soon joined the effort. While porting Unix to a PDP-11 they invented C, heavily inspired by Martin Richards’ portable systems programming language BCPL. In 1972 Unix was rewritten in C, and later ported to many other machines aided by Steve Johnsons Portable C Compiler. C gained popularity outside the realm of PDP-11 and Unix. Initially the K&R was the definitive reference until the language was standardized by ANSI and ISO in 1989/1990 and thereafter updated in 1999 and 2011.
Ken Thompson, Dennis Ritchie and 20+ more technical staff from Bell Labs had been working on the very innovative Multics project for several years.
The MULTICS ("Multiplexed Information and Computing Service) was started in 1964, as a cooperative project led by MIT's Project MAC (Multiple Access Computing), General Electric and Bell Labs.

Bell Labs pulled out of the project in 1969.
Multics was a huge project, with great ambitions. It was a secure time-sharing system with lots of advanced features, and it was one of the few operating systems at the time written in a high level language, PL/1.

```plaintext
FACT: PROC;
DCL I FIXED, PRINT ENTRY, F ENTRY RETURNS(FIXED), N INT;
DO I = 1 TO 10;
CALL PRINT("Factorial is", F(I));
END;
F: PROC (N) FIXED;
DCL N FIXED;
IF N = 0 THEN RETURN(1);
RETURN(N*F(N-1));
END F;
END FACT;
```
While working on the Multics projects, Dennis and Ken had also been exposed to the very portable language systems programming language BCPL.

"Both of us were really taken by the language and did a lot of work with it." (Ken Thompson, 1989 interview)
BCPL, Basic CPL, had been described and implemented for the Project MAC in 1967 by a visiting researcher, Martin Richards from Cambridge University.

**BCPL** is a simple recursive programming language designed for compiler writing and system programming; it was derived from true CPL (Combined Programming Language) by removing those features of the full language which make compilation difficult namely, the type and mode matching rules and the variety of definition structures with their associated scope rules.
Before visiting MIT, Martin Richards had been actively involved in developing a compiler for a very ambitious programming language - CPL.

```plaintext
function Euler [function Fct, real Eps; integer Tim] = result of
  §1 dec §1.1 real Mn, Ds, Sum
  integer i, t
  index n=0
  m = Array [real, (0, 15)] §1.1
  i, t, m[0] := 0, 0, Fct[0]
  Sum := m[0]/2
  §1.2 i := i + 1
  Mn := Fct[i]
  for k = step 0, 1, n do
    m[k], Mn := Mn, (Mn + m[k])/2
  test Mod[Mn] < Mod[m[n]] ∧ n < 15
    then do Ds, n, m[n+1] := Mn/2, n+1, Mn
    or do Ds := Mn
    Sum := Sum + Ds
    t := (Mod[Ds] < Eps) → t + 1, 0 §1.2
  repeat while t < Tim
  result := Sum §1.
```
Designed jointly by the Mathematical Laboratory at the University of Cambridge and the University of London Computer Unit
for the Atlas computer (ordered in 1961, operational in 1964)
CPL was designed and partly implemented before the Atlas computer was operational. Martin Richard and the others had to work on the EDSAC 2 computer.

EDSAC 2 users in 1960
Which was an upgrade of the EDSAC computer. Arguably, the first electronic digital stored-program computer. It ran its first program May 6, 1949

Maurice Wilkes and Bill Renwick in front of the complete EDSAC

http://en.wikipedia.org/wiki/Electronic_Delay_Storage_Automatic_Calculator
Maurice Wilkes' himself commenting on the 1951 film about how EDSAC was used in practice:

https://youtu.be/x-vS0WcJyNM
The EDSAC 1951 film abridged version

Commentary by M. V. Wilkes
The EDSAC 1951 film abridged version

Commentary by M. V. Wilkes
EDSAC

EDSAC was constructed from electronic components, using a total of about 20,000 vacuum tubes. It was designed and built by Maurice Wilkes and his team at Cambridge University. The EDSAC was commissioned in 1949 and is now available as a working example at the University of Cambridge Computer Laboratory.

Initial Orders

The EDSAC was initially used for scientific calculations. Its instructions were designed to be simple and straightforward, allowing for easy programming.

The Squares Program

The program, written by Maurice Wilkes in 1949, computes the following table of squares and differences of the numbers from 1 to 10.

<table>
<thead>
<tr>
<th>Order</th>
<th>Number</th>
<th>Square</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>36</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>49</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>81</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td>19</td>
</tr>
</tbody>
</table>

The Green House

The program was named the "Green House" after the building where it was located. The name referred to the building's color at the time of the EDSAC's commissioning.

Links

- [http://www.cl.cam.ac.uk/~mr10/edsacposter.pdf](http://www.cl.cam.ac.uk/~mr10/edsacposter.pdf)
- [http://www.cl.cam.ac.uk/~mr/Edsac.html](http://www.cl.cam.ac.uk/~mr/Edsac.html)

The contrasted tape segments etched on the Tea Room glass panels.
“Hi” on the EDSAC / Initial Orders I

T44S  31          T _end+1       mark end of program
E38S  32          E _start       jump to beginning of program
*S  33  lshift  *              letter shift
HS  34  _H      H               letter H
IS  35  _I      I               letter I
&S  36  lf      &               LF - line feed character
@S  37  cr      @               CR - carriage return character
O33S  38  _start  O lshift     prepare for printing letters
O34S  39          O _H          print H
O35S  40          O _I          print I
O36S  41          O lf          print lf
O37S  42          O cr          print cr
ZS  43  _end    Z               end of program

T44SE38S*SHSIS&@S033S034S035S036S037SZS
“Count to 10” on the EDSAC / Initial Orders 1

T62S 31 T_end+1 mark end of program
E43S 32 E_start jump to beginning of program
#S 33 fshift # figure shift
&S 34 lf & LF - line feed character
@S 35 cr @ CR - carriage return character
PS 36 dummy P dummy (used to reset Acc)
P0S 37 first P 0 first value
P9S 38 last P 9 last value
P1S 39 incr P 1 increment
PJ 40 cur P current value
PS 41 d P d - digit to be printed
XS 42 _start X nop
O33S 43 O fshift prepare for printing digits
T36S 44 T dummy reset Acc
A37S 45 A first load first
T40S 46 T cur store to cur
XS 47 _loop X nop
T36S 48 T dummy reset Acc
A40S 49 A cur load current value
L512S 50 L 2^(11-2) Acc << 11, create a digit
T41S 51 T d store digit to be printed
O41S 52 O d print digit
A40S 53 A cur load current value
A39S 54 A incr acc += 1
T40S 55 T cur store current value
A38S 56 A last load last value
S40S 57 S cur last - cur < 0, should we break?
E48S 58 E_loop if no, jump to loop
O34S 59 O lf print line feed
O35S 60 O cr print carriage return
ZS 61 _end Z stop program
I pretended I was a student, who had won a single chance to run my program on this precious computer.

The program did actually ran on the very first attempt!
"FizzBuzz" on the EDSAC / Initial Orders 1

```
T123S 31  T L_end  mark end of program
E60S 32  E L_start  jump to the beginning of program
#S 33 FS  # figure shift
#S 34 LS  # letter shift
&S 35 LF  & linefeed character
@S 36 CR  @ carriage return character
P100S 37 100  P 100  constant 100
P18S 38 10  P 10  constant 10
P5S 39 5  P 5  constant 5
P3S 40 3  P 3  constant 3
P1S 41 1  P 1  constant 1
Q5 42 1'Q  constant figure 1
PS 43 0'P  constant figure 0
BS 44'B  B constant letter B
CS 45'F  F constant letter F
IS 46'I  I constant letter I
US 47'U  U constant letter U
ZS 48'Z  Z constant letter Z

used to flush and reset the accumulator

number to be printed, negative if counter is mod 3 or mod 5
digit to be printed

T123S 31  T L_end  mark end of program
E60S 32  E L_start  jump to the beginning of program
#S 33 FS  # figure shift
#S 34 LS  # letter shift
&S 35 LF  & linefeed character
@S 36 CR  @ carriage return character
P100S 37 100  P 100  constant 100
P18S 38 10  P 10  constant 10
P5S 39 5  P 5  constant 5
P3S 40 3  P 3  constant 3
P1S 41 1  P 1  constant 1
Q5 42 1'Q  constant figure 1
PS 43 0'P  constant figure 0
BS 44'B  B constant letter B
CS 45'F  F constant letter F
IS 46'I  I constant letter I
US 47'U  U constant letter U
ZS 48'Z  Z constant letter Z

used to flush and reset the accumulator

number to be printed, negative if counter is mod 3 or mod 5
digit to be printed
```
### FizzBuzz on the EDSAC / Initial Orders 1

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T510S 31</td>
<td>_L_end</td>
</tr>
<tr>
<td>E50S 32</td>
<td>_L_start</td>
</tr>
<tr>
<td>034S</td>
<td>_L_next</td>
</tr>
<tr>
<td>035S</td>
<td>_L_count</td>
</tr>
<tr>
<td>036S</td>
<td>_L_print</td>
</tr>
<tr>
<td>T49S</td>
<td>_dummy</td>
</tr>
<tr>
<td>G109S</td>
<td>L_print10s</td>
</tr>
<tr>
<td>S38S</td>
<td>L_count10s</td>
</tr>
<tr>
<td>A50S</td>
<td>_cnt</td>
</tr>
<tr>
<td>T49S</td>
<td>_dummy</td>
</tr>
<tr>
<td>O48S</td>
<td>_Z</td>
</tr>
<tr>
<td>O48S</td>
<td>_Z</td>
</tr>
<tr>
<td>O45S</td>
<td>_F</td>
</tr>
<tr>
<td>O36S</td>
<td>_LS</td>
</tr>
<tr>
<td>034S</td>
<td>_L_next</td>
</tr>
<tr>
<td>035S</td>
<td>_L_count</td>
</tr>
<tr>
<td>036S</td>
<td>_L_print</td>
</tr>
<tr>
<td>T49S</td>
<td>_dummy</td>
</tr>
<tr>
<td>G109S</td>
<td>L_print10s</td>
</tr>
<tr>
<td>S38S</td>
<td>L_count10s</td>
</tr>
<tr>
<td>A50S</td>
<td>_cnt</td>
</tr>
<tr>
<td>T49S</td>
<td>_dummy</td>
</tr>
<tr>
<td>O48S</td>
<td>_Z</td>
</tr>
<tr>
<td>O48S</td>
<td>_Z</td>
</tr>
<tr>
<td>O45S</td>
<td>_F</td>
</tr>
<tr>
<td>O36S</td>
<td>_LS</td>
</tr>
</tbody>
</table>
“FizzBuzz” on the EDSAC / Initial Orders 1

T49S 73 L_notBuzz T_dummy reset Acc
T49S 74 A_cnt load Acc with _cnt
T49S 75 L_Buzz T_dummy subtract 5
E101S 108 L_count10s loop unconditionally
A50S 77 A_cnt add 5, restore previous value
A50S 78 S_cnt subtract 1, to check if Acc was 0
S39S 79 E_notBuzz jump if Acc was not 0, ie number was not divisible by 5
E86S 80 T_num set _num to negative value, flag that no value should be printed
T51S 81 O_FS prepare for printing letters
T49S 82 O_B output B
T49S 83 O_U output U
T49S 84 O_LS output Z
T49S 85 O_0 output Z
T49S 86 O_Z output Z
T49S 87 O_FS end program to be printed
T49S 88 A_cnt load counter
T51S 89 S_100 subtract 100, check if we should stop
G98S 93 G_L_not100 jump if not 100 yet
O42S 94 O_LS prepare printing letters
O43S 95 O_FS output Z
O43S 96 O_FS output Z
ZS 97 Z end the program
"FizzBuzz" on the EDSAC / Initial Orders 1

T49S  98  L_not100  T dummy reset Acc
T52S  99  T d reset digit

A59S 100  A cnt load counter
S38S 101  L_count10s S _10 subtract 10
G109S 102  G L_print10s goto print 10s if Acc < 0
T51S 103  T num store number
E101S 108  E L_count10s loop unconditionally
T49S 109  L_print10s T dummy reset Acc
A52S 110  A d load digit

L512S 114  L 2^11-2 Acc << 11, create a printable figure
T52S 115  T d store digit

A41S 113  A _1 increase digit
T50S 118  T cnt store Acc into _cnt
A40S 117  A _1 add 3, restore previous value
“FizzBuzz” on the EDSAC / Initial Orders 1

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html
“FizzBuzz” on the EDSAC / Initial Orders 1

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html

There is a small bug in the program. Did you notice?
“FizzBuzz” on the EDSAC / Initial Orders 1

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html
“FizzBuzz” on the EDSAC / Initial Orders 1

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html

Here is a quick and dirty fix!
“FizzBuzz” on the EDSAC / Initial Orders 1

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html
“FizzBuzz” on the EDSAC / Initial Orders I

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html
“FizzBuzz” on the EDSAC / Initial Orders I

Try this program on NISHIO Hirokazu’s EDSAC Simulator
http://nhiro.org/learn_language/repos/EDSAC-on-browser/index.html

Enjoy!
Speedcoding, John Backus, 1953 on the IBM 701

IBM 701 operator's console

IBM 701 processor frame

http://en.wikipedia.org/wiki/IBM_701
Backus later did work on the IBM 704
Fortran (appeared 1957, designed by John Backus)

The initial release of FORTRAN for the IBM 704 contained 32 statements, including:

- **DIMENSION** and **EQUIVALENCE** statements
- Assignment statements
- Three-way arithmetic **IF** statement, which passed control to one of three locations in the program depending on whether the result of the arithmetic statement was negative, zero, or positive
- **IF** statements for checking exceptions (**ACCUMULATOR OVERFLOW**, **QUOTIENT OVERFLOW**, and **DIVIDE CHECK**); and
- **IF** statements for manipulating sense switches and sense lights
- **GOTO**, computed **GOTO**, **ASSIGN**, and assigned **GOTO**
- **DO** loops
- Formatted I/O: **FORMAT**, **READ**, **READ INPUT TAPE**, **WRITE**, **WRITE OUTPUT TAPE**, **PRINT**, and **PUNCH**
- Unformatted I/O: **READ TAPE**, **READ DRUM**, **WRITE TAPE**, and **WRITE DRUM**
- Other I/O: **END FILE**, **REWIND**, and **BACKSPACE**
- **PAUSE**, **STOP**, and **CONTINUE**
- **FREQUENCY** statement (for providing optimization hints to the compiler).

**FORTRAN II** [edit]

IBM’s **FORTRAN II** appeared in 1958. The main enhancement was to support **procedural programming** by allowing user-written subroutines and functions which returned values, with parameters passed by **reference**. The **COMMON** statement provided a way for subroutines to access common (or **global**) variables. Six new statements were introduced:

- **SUBROUTINE**, **FUNCTION**, and **END**
- **CALL** and **RETURN**
- **COMMON**
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
    READ INPUT TAPE 5, 501, IA, IB, IC
  501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C IS GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
    IF (IA) 777, 777, 701
  701 IF (IB) 777, 777, 702
  702 IF (IC) 777, 777, 703
  703 IF (IA+IB-IC) 777,777,704
  704 IF (IA+IC-IB) 777,777,705
  705 IF (IB+IC-IA) 777,777,799
  777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
  799 S = FLOATF (IA + IB + IC) / 2.0
   AREA = SQRT( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) * 
             (S - FLOATF(IC)))
    WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
  601 FORMAT (4H A= ,I5,5H B= ,I5,5H C= ,I5,8H AREA= ,F10.2, 
             + 13H SQUARE UNITS)
    STOP
END

Simple FORTRAN II program
IAL (aka Algol 58) (designed by Friedrich L. Bauer, Hermann Bottenbruch, Heinz Rutishauser, Klaus Samelson, John Backus, Charles Katz, Alan Perlis, Joseph Henry Wegstein)

```
procedure Simps (F( ), a, b, delta, V);
comment a, b are the min and max, resp. of the points def. interval of integ. F( ) is the function to integrated.
delta is the permissible difference between two successive Simpson sums V is greater than the maximum absolute value of F on a, b;
begin
  Simps: Ibar: = V \times (b - a)
  n : = 1
  h : = (b - a) / 2
  J : = h \times (F(a) + F(b))
  S : = 0;
  for k : = 1 \to n
    S : = S + F(a + (2 \times k - 1) \times h)
    I : = J + 4 \times h \times S
  if (delta < abs (I - Ibar)) then
    begin
      Ibar: = I
      J : = (I + J) / 4
      n : = 2 \times n; h : = h / 2
      go to J1 end
  Simps : = 1 / 3
return integer (k, n)
end Simps
```

http://en.wikipedia.org/wiki/ALGOL_58
EDSAC 2 users in 1960
A scaled down version of Atlas (called Titan / Atlas2) was ordered in 1961, delivered to Cambridge in 1963, but not usable until early 1964.
a programming language was needed!

Many existing programming languages was concidered, but....
Algol 60 was criticized as not enabling efficient compilation, call by name being cited as a main cause. A second area of concern was the side effects of procedures necessitating a strict left-to-right rule for the evaluation of expressions.
ALGOL 60 was just “a language, not a programming system”

procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
   value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m,
is transferred to y, and the subscripts of this element to i and k;
begin
   integer p, q;
y := 0; i := k := 1;
for p := 1 step 1 until n do
   for q := 1 step 1 until m do
      if abs(a[p, q]) > y then
         begin y := abs(a[p, q]);
            i := p; k := q
         end
end Absmax

Algo 60 was criticized as not enabling efficient compilation, call by name being cited as a main cause. A second area of concern was the side effects of procedures necessitating a strict left-to-right rule for the evaluation of expressions.
Fortran IV was too tied up to IBM 709/7090

```fortran
C THE TPK ALGORITHM
C FORTRAN IV STYLE
DIMENSION A(11)
FUN(T) = SQRT(ABS(T)) + 5.*)T**3
READ (5,1) A
1 FORMAT (5F10.2)
DO 10 J = 1, 11
   I = 11 - J
   Y = FUN(A(I+1))
   IF (400.0-Y) 4, 8, 8
   WRITE (6,5) I
4   WRITE (6,9) I, Y
5   FORMAT(I10, F12.6)
   GO TO 10
8   WRITE (6,9) I, Y
   FORMAT(I10, F12.6)
10 CONTINUE
STOP
END
```

From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
http://www.fortran.bcs.org/2005/fortran/img10.jpg
begin
real a, b, c, Sx, Sy, Sxx, Sxy, Syy, nextx, nexty
integer n
read (nextx)

2: Sx = 0; Sy = 0; Sxx = 0; Sxy = 0; Syy = 0
n = 0

1: read (nexty); n = n + 1
Sx = Sx + nextx; Sy = Sy + nexty
Sxx = Sxx + nextx^2; Syy = Syy + nexty^2
Sxy = Sxy + nextx*nexty

3: read (nextx); ->1 unless nextx = 999 999
a = (n*Sxy - Sx*Sy)/(n*Sxx - Sx^2)
b = (Sy - a*Sx)/n
c = Syy - 2(a*Sxy + b*Sy) + a^2*Sxx - 2a*b*Sx + n*b^2
newline
print fl(a,3); space; print fl(b,3); space; print fl(c,3)
read (nextx); ->2 unless nextx = 999 999
stop
end of program

“the use of compiler-compiler technology frightened us”
But, hey....
In the early 1960's, it was common to think "we are building a new computer, so we need a new programming language."

(David Hartley, in 2013 article)
CPL

Cambridge Programming Language
CPL

Cambridge Programming Language
Cambridge Plus London
CPL

Cambridge Programming Language

Cambridge Plus London
CPL

Cambridge Programming Language
Cambridge Plus London
Combined Programming Language
CPL

Cambridge Programming Language
Cambridge Plus London
Combined Programming Language
(Cristophers’ Programming Language)
"anything not explicitely allowed should be forbidden ... nothing should be left undefined, as occurs in ALGOL 60"

"It was envisaged that [the language] would be sufficiently general and versatile to dispense with machine-code programming as far as possible"
"anything not explicity allowed should be forbidden ... nothing should be left undefined, as occurs in ALGOL 60"

"It was envisaged that [the language] would be sufficiently general and versatile to dispense with machine-code programming as far as possible"
Advanced were made in understanding the evaluation of expressions so as to recognize not just the value of data but also its location. Taking terminology related to the assignment statement, we developed the concept of left-hand and right-hand values ... this enabled an assignment statement to have the generalized form

\[ \text{<expression>} := \text{<expression>} \]

the first being evaluated in left-hand mode to reveal a location and the second in right-hand mode to obtain a value to be assigned to that location.

From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Advanced were made in understanding the evaluation of expressions so as to recognize not just the value of data but also its location. Taking terminology related to the assignment statement, we developed the concept of left-hand and right-hand values ... this enabled an assignment statement to have the generalized form

\[ \text{<expression>} := \text{<expression>} \]

the first being evaluated in left-hand mode to reveal a location and the second in right-hand mode to obtain a value to be assigned to that location.

From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
The main features of CPL


The paper provides an informal account of CPL, a new programming language currently being implemented for the Titan at Cambridge and the Atlas at London University. CPL is based on, and contains the concepts of, ALGOL 60. In addition there are extended data descriptions, command and expression structures, provision for manipulating non-numerical objects, and comprehensive input-output facilities. However, CPL is not just another proposal for the extension of ALGOL 60, but has been designed from first principles and has a logically coherent structure.
function Euler [function Fct, real Eps; integer Tim] = result of
§1 dec §1.1 real Mn, Ds, Sum
integer i, t
index n=0
m = Array [real, (0, 15)] §1.1
i, t, m[0] := 0, 0, Fct[0]
Sum := m[0]/2
§1.2 i := i + 1
Mn := Fct[i]
for k = step 0, 1, n do
  m[k], Mn := Mn, (Mn + m[k])/2
  test Mod[Mn] < Mod[m[n]] ∧ n < 15
    then do Ds, n, m[n+1] := Mn/2, n+1, Mn
    or do Ds := Mn
  Sum := Sum + Ds
  t := (Mod[Ds] < Eps) → t + 1, 0 §1.2
repeat while t < Tim
result := Sum §1.
Martin Richards started as a research student in 1963.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

as ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

As ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963

As ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

As ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963 as ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963

as ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

as ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963

as ML that were influenced by Christopher Strachey's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963

as ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963.

as ML that were influenced by Christopher’s ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley’s article “CPL: Failed Venture or Noble Ancestor?” (2013)
Martin Richards started as a research student in 1963.

As ML that were influenced by Christopher's ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

as ML that were influenced by Christopher’s ideas.

My role in the CPL project was to help with the implementation of the Cambridge CPL compiler. The task was daunting because we were working with a new language that included many of the innovations found in Algol 60 that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.
Martin Richards started as a research student in 1963.

My role in the CPL project was to help develop the Cambridge CPL compiler. The task was daunting because it included many of the innovations found in ML that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
Martin Richards started as a research student in 1963.

My role in the CPL project was to help design the Cambridge CPL compiler. The task was daunting because the language that we were working on included many of the innovations found in ML that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

“Christopher Strachey and the Cambridge CPL Compiler”, Martin Richards
From David Hartley’s article “CPL: Failed Venture or Noble Ancestor?” (2013)
Martin Richards started as a research student in 1963.

My role in the CPL project was to help build the Cambridge CPL compiler. The task was daunting because it included many of the innovations found in ML that were known to be difficult to implement efficiently. But CPL was larger. It had more datatypes, and it was one of the first languages to adopt a scheme whereby the types of variables could be deduced without the user having to explicitly declare them. In addition to call-by-value and call-by-name, it had call-by-reference. It had two kinds of procedures: fixed and free, distinguished by whether their free variables were effectively called by value or by reference. It also allowed label variables and the passing of labels as arguments combined with a goto statement that not only allowed jumps out of procedures (analogous to the use of longjmp in C), but also jumps to labels in inner blocks causing the intervening declarations to be obeyed. Later in the project the language provided structures, unions and pointers, together with runtime garbage collection.

"Christopher Strachey and the Cambridge CPL Compiler", Martin Richards
From David Hartley's article "CPL: Failed Venture or Noble Ancestor?" (2013)
CPL was once compared to the invention of a pill that could cure every type of ill.
Writing a compiler for CPL was too difficult.
Writing a compiler for CPL was too difficult.

Cambridge never succeeded writing a working CPL compiler.
Writing a compiler for CPL was too difficult.

Cambridge never succeeded writing a working CPL compiler.

Development on CPL ended December 1966.
Inspired by his work on CPL, Martin Richards wanted to create a language:
Inspired by his work on CPL, Martin Richards wanted to create a language:

• that was simple to compile
• with direct mapping to machine code
• that assumes the programmer know what he is doing
Inspired by his work on CPL, Martin Richards wanted to create a language:

- that was simple to compile
- with direct mapping to machine code
- that assumes the programmer know what he is doing

"The philosophy of BCPL is not one of the tyrant who thinks he knows best and lay down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions.” (The BCPL book, 1979)
Inspired by his work on CPL, Martin Richards wanted to create a language:

- that was simple to compile
- with direct mapping to machine code
- that assumes the programmer know what he is doing

"The philosophy of BCPL is not one of the tyrant who thinks he knows best and lay down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions." (The BCPL book, 1979)
Inspired by his work on CPL, Martin Richards wanted to create a language:

• that was simple to compile
• with direct mapping to machine code
• that assumes the programmer know what he is doing

"The philosophy of BCPL is not one of the tyrant who thinks he knows best and lay down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions.” (The BCPL book, 1979)
BCPL is a simple recursive programming language designed for compiler writing and system programming. It was derived from true CPL (Combined Programming Language) by removing those features of the full language which make compilation difficult namely, the type and mode matching rules and the variety of definition structures with their associated scope rules.
BCPL is a simple recursive programming language designed for compiler writing and system programming. It was derived from true CPL (Combined Programming Language) by removing those features of the full language which make compilation difficult, namely, the type and mode matching rules and the variety of definition structures with their associated scope rules.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. A simplified syntax.
2. All data items have R-values which are bit patterns of the same length and the type of an R-value depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.
3. BCPL has a manifest named constant facility.
4. Functions and routines may only have free variables which are manifest named constants or whose L-values are manifest constants (i.e., explicit functions or routines, labels or global variables).
5. The user may manipulate both L and R-values explicitly.
6. There is a scheme for separate compilation of segments of a program.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. A simplified syntax.
2. All data items have Rvalues which are bit patterns of the same length and the type of an Rvalue depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.
3. BCPL has a manifest named constant facility.
4. Functions and routines may only have free variables which are manifest named constants or whose Lvalues are manifest constants (i.e., explicit functions or routines, labels or global variables).
5. The user may manipulate both L and Rvalues explicitly.
6. There is a scheme for separate compilation of segments of a program.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. A simplified syntax.

2. All data items have R-values which are bit patterns of the same length and the type of an R-value depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.

3. BCPL has a manifest named constant facility.

4. Functions and routines may only have free variables which are manifest named constants or whose L-values are manifest constants (i.e., explicit functions or routines, labels or global variables).

5. The user may manipulate both L- and R-values explicitly.

6. There is a scheme for separate compilation of segments of a program.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. A simplified syntax.
2. All data items have Rvalues which are bit patterns of the same length and the type of an Rvalue depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.
3. BCPL has a manifest named constant facility.
4. Functions and routines may only have free variables which are manifest named constants or whose Lvalues are manifest constants (i.e., explicit functions or routines, labels or global variables).
5. The user may manipulate both L and Rvalues explicitly.
6. There is a scheme for separate compilation of segments of a program.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. A simplified syntax.
2. All data items have Rvalues which are bit patterns of the same length and the type of an Rvalue depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.
3. BCPL has a manifest named constant facility.
4. Functions and routines may only have free variables which are manifest named constants or whose Lvalues are manifest constants (i.e., explicit functions or routines, labels or global variables).
5. The user may manipulate both L and Rvalues explicitly.
6. There is a scheme for separate compilation of segments of a program.
BCPL is the heart of the BCPL Compiling System; it is a language which looks much like true CPL [1] but is, in fact, a simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

1. **A simplified syntax.**

2. **All data items have Rvalues which are bit patterns of the same length and the type of an Rvalue depends only on the context of its use and not on the declaration of the data item.** This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.

3. **BCPL has a manifest named constant facility.**

4. **Functions and routines may only have free variables which are manifest named constants or whose lvalues are manifest constants (i.e., explicit functions or routines, labels or global variables).**

5. **The user may manipulate both L and Rvalues explicitly.**

6. **There is a scheme for separate compilation of segments of a program.**
Lucky and humble fans meet Martin Richards, the inventor of BCPL
So what is the link between BCPL and B and C?
Interviewer: Did you develop B?

Thompson: I did B.

Interviewer: As a subset of BCPL?

Thompson: It wasn't a subset. It was almost exactly the same.

... Thompson: It was the same language as BCPL, it looked completely different, syntactically it was, you know, a redo. The semantics was exactly the same as BCPL. And in fact the syntax of it was, if you looked at, you didn't look too close, you would say it was C. Because in fact it was C, without types.

...
The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution.

From the HOPL article by Dennis Ritchie in 1993

BCPL, B and C differ syntactically in many details, but broadly they are similar.
B is a computer language intended for recursive, primarily non-numeric applications typified by system programming. B has a small, unrestrictive syntax that is easy to compile. Because of the unusual freedom of expression and a rich set of operators, B programs are often quite compact.
B is a computer language intended for recursive, primarily non-numeric applications typified by system programming. B has a small, unrestricted syntax that is easy to compile. Because of the unusual freedom of expression and a rich set of operators, B programs are often quite compact.
Users' Reference to B, Ken Thompson, January 1972

The BCPL Reference Manual, Martin Richards, July 1967
An RVALUE is a binary bit pattern of a fixed length (which is implementation dependent), it is usually the size of a computer word. Rvalues may be used to represent a variety of different kinds of objects such as integers, truth values, vectors or functions. The actual kind of object represented is called the TYPE of the Rvalue.

An rvalue is a binary bit pattern of a fixed length. On the PDP-11 it is 16 bits. Objects are rvalues of different kinds such as integers, labels, vectors and functions. The actual kind of object represented is called the type of the rvalue.
excerpt from the BCPL reference manual (Richards, 1967), page 6

A BCPL expression can be evaluated to yield an Rvalue but its type remains undefined until the Rvalue is used in some definitive context and it is then assumed to represent an object of the required type. For example, in the following function application

$$(B^*\{i\} \rightarrow f, g) [1, Z[i]]$$

the expression $$(B^*\{i\} \rightarrow f, g)$$ is evaluated to yield an Rvalue which

excerpt from the B reference manual (Thompson, 1972), page 6

A B expression can be evaluated to yield an rvalue, but its type is undefined until the rvalue is used in some context. It is then assumed to represent an object of the required type. For example, in the following function call

$$(b?f:g[i])(1,x>1)$$

The expression $$(b?f:g[i])$$ is evaluated to yield an rvalue which
An LVALUE is a bit pattern representing a storage location containing an Rvalue. An Lvalue is the same size as an Rvalue and is a type in BCPL. There is one context where an Rvalue is interpreted as an Lvalue and that is as the operand of the monadic operator $rv$. For example, in the expression

\[ rv\ f[i] \]

the expression $f[i]$ is evaluated to yield an Rvalue which is then

An lvalue is a bit pattern representing a storage location containing an rvalue. An lvalue is a type in B. The unary operator $*$ can be used to interpret an rvalue as an lvalue. Thus

\[ *x \]

evaluates the expression $x$ to yield an rvalue, which is then
C is a new computer language designed for both non-numerical and numerical applications. The fundamental types of objects with which it deals are characters, integers, and single- and double-precision numbers, but the language also provides multidimensional arrays, structures containing data of mixed type, and pointers to data of all types.

C is based on an earlier language B, from which it differs mainly in the introduction of the notions of types and of structures. This paper is a reference manual for the original implementation of C on the Digital Equipment Corporation PDP-11/45 under the UNIX time-sharing system. The language is also available on the IBM 6000 and IBM S/370.
Interesting fact:
Interesting fact:

The C74 reference manual does not mention BCPL at all.
Interesting fact:

The C74 reference manual does not mention BCPL at all. It does not even mention the B reference manual by Ken Thompson.
Interesting fact:

The C74 reference manual does not mention BCPL at all. It does not even mention the B reference manual by Ken Thompson.

REFERENCES

“Good artists copy. Great artists steal.”

Picasso?
good_research_labs(knowledge k);
great_research_labs(knowledge && k);

/* Bell Labs? */
BCPL

• Designed by Martin Richards, appeared in 1966, typeless (everything is a word)
• Influenced by Fortran and Algol
• Intended for writing compilers for other languages
• Simplified version of CPL by "removing those features of the full language which make compilation difficult"

GET "LIBHDR"

GLOBAL $(
    COUNT: 200
    ALL: 201
)$

LET TRY(LD, ROW, RD) BE
    TEST ROW = ALL THEN
        COUNT := COUNT + 1
    ELSE $(
        LET POSS = ALL & ~(LD | ROW | RD)
        UNTIL POSS = 0 DO $(
            LET P = POSS & ~POSS
            POSS := POSS - P
            TRY(LD + P << 1, ROW + P, RD + P >> 1)
        )
    )
$

LET START() = VALOF $(
    ALL := 1
    FOR I = 1 TO 12 DO $(
        COUNT := 0
        TRY(0, 0, 0)
        WRITF("%I2-QUEENS PROBLEM HAS %I5 SOLUTIONS*\n", I, COUNT)
        ALL := 2 * ALL + 1
    )
)$
RESULTIS 0
PDP-7
(18-bit computer, introduced 1965)

THIS IS A SAMPLE PROGRAM

GO,
LAS
SPA: CMA
JMP GO
DAC #CNTSET
LAC (1
DAC #BIT
CLL.

LOOP,
LAC CNTSET
DAC CNT
LAC BIT
ISZ #CNT
JMP -.1
RAL
DAC BIT
LAS
SMA
JMP LOOP
JMP GO

START GO
/* The following program will calculate the constant e-2 to about 4000 decimal digits, and print it 50 characters to the line in groups of 5 characters. */

main() {
    extrn putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n)
        v[i++] = 1;
    while(col<2*n) {
        a = n+1;
        c = i = 0;
        while (i<n) {
            c =+ v[i] *10;
            v[i++] = c%a;
            c =/ a--;
        }

        putchar(c+'0');
        if(!(++col%5))
            putchar(col%50?' ':'*n');
    }
    putchar('*n*n');
}

v[2000];
n 2000;

Designed by Ken Thompson, appeared in ~1969, typeless (everything is a word)
"BCPL squeezed into 8K words of memory and filtered through Thompson's brain"
/* The following program will calculate the constant e-2 to about
4000 decimal digits, and print it 50 characters to the line in
groups of 5 characters. */

main() {
    extrn putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n) { v[i++] = 1; }
    while(col<2*n) {
        a = n+1 ;
        c = i = 0;
        while (i<n)
            c =+ v[i] *10;
            v[i++] = c%a;
            c =/ a--;
        }
        putchar(c+'0');
        if(!(++col%5))
            putchar(col%50?' ': '*n');
    } 
    putchar('*n*n');
}

v[2000];
n 2000;

Designed by Ken Thompson, appeared in ~1969, typeless (everything is a word)
"BCPL squeezed into 8K words of memory and filtered through Thompson's brain"
/* The following program will calculate the constant e-2 to about 4000 decimal digits, and print it 50 characters to the line in groups of 5 characters. */

main() {
    extrn putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n) {v[i++] = 1;
    while(col<2*n) {
        a = n+1;
        c = i = 0;
        while (i<n) {
            c =+ v[i] *10;
            v[i++] = c%a;
            c =/ a--;
        }
        putchar(c+'0');
        if(!(++col%5))
            putchar(col%50?' ': '*n');
    }
    putchar('*n*n');
}

v[2000];
n 2000;

Designed by Ken Thompson, appeared in ~1969, typeless (everything is a word)
"BCPL squeezed into 8K words of memory and filtered through Thompson's brain"
/* The following program will calculate the constant e-2 to about
4000 decimal digits, and print it 50 characters to the line in
groups of 5 characters. */

main() {
    extern putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n) {v[i++] = 1;}
    while(col<2*n) {
        a = n+1 ;
        c = i = 0;
        while (i<n) {
            c += v[i] *10;
            v[i++] = c%a;
            c /= a--;
        }
        putchar(c+'0');
        if(!(++col%5)) {putchar(col%50?' ': '*n');
        } else {putchar('*n*n');
    }
}

v[2000];
n 2000;

B

Designed by Ken Thompson, appeared in ~1969, typeless (everything is a word)
"BCPL squeezed into 8K words of memory and filtered through Thompson's brain"
PDP-11

- 16-bit computer
- introduced 1970
- orthogonal instruction set
- byte-oriented
Early C

• Designed by Dennis Ritchie and Ken Thompson
• Developed during 1969-1972 in parallel with Unix
• Developed because of the PDP-11, a 16-bit, byte-oriented machine
• C introduced more types: integer types, characters and floating point types
• A key design principle was to make C amenable to translation by simple compilers
• Storage limitations often demanded a one-pass technique in which output was generated as soon as possible.
• While C had been ported to other architectures, until about 1977 Unix itself had only been running on DEC architectures.
• The PCC (Portable C Compiler, Stephen C. Johnson) was an important reference implementation
• It was not until 1977-1979 that the portability of Unix was demonstrated
• very productive time 1977-1979 for C as Unix was ported to new platforms
K&R C

The seminal book "The C Programming Language" (1978) acted for a long time as the only formal definition of the language.

```c
/* C78 example, K&R C */

mystrcpy(s,t)
char *s;
char *t;
{
  int i;
  for (i = 0; (*s++ = *t++) != '\0'; i++)
    ;
  return(i);
}

main()
{
  char str1[10];
  char str2[] = "Hello, C78!";
  int len = mystrcpy(str1, str2);
  int i;
  for (i = 0; i < len; i++)
    putchar(str1[i]);
  exit(0);
}
```
Standardization of C started in 1983

Many people don't realize how unusual the C standardization effort, especially the original ANSI C work, was in its insistence on standardizing only tested features. Most language standard committees spend much of their time inventing new features, often with little consideration of how they might be implemented. Indeed, the few ANSI C features that were invented from scratch — e.g., the notorious “trigraphs” — were the most disliked and least successful features of C89.

-- Henry Spencer
Standardization of C started in 1983

Many people don't realize how unusual the C standardization effort, especially the original ANSI C work, was in its insistence on standardizing only tested features. Most language standard committees spend much of their time inventing new features, often with little consideration of how they might be implemented. Indeed, the few ANSI C features that were invented from scratch — e.g., the notorious “trigraphs” — were the most disliked and least successful features of C89.

-- Henry Spencer
• Dennis Ritchie not involved (except for the “noalias must go” article)
• Committee met four times a year, from 83 til publication
• All meetings in the US (due to political issues between ANSI and ISO)
• The committee avoided inventing features
• All features had to be demonstrated by one or more existing compilers
• Hot topic: value preserving vs unsigned preserving (value preserving won)
• The idea of text files vs binary files (due to Microsofts CR/NL vs Unix NL)
• The standard was delayed about 2 years due to a US protest
/* C89 example, ANSI C */

#include <stdio.h>

int mystrcpy(char *s, const char *t)
{
    int i;
    for (i = 0; (*s++ = *t++) != '\0'; i++)
        ;
    return i;
}

int main(void)
{
    char str1[10];
    char str2[] = "Hello, C89!";
    size_t len = mystrcpy(str1, str2);
    size_t i;
    for (i = 0; i < len; i++)
        putchar(str1[i]);
    return 0;
}

ANSI C / C89 / C90

Soon after it was all ISO/IEC
ISO/IEC 9899/AMD1:1995, aka “C95”

- Add more extensive support for international character sets (mostly done by Japan)
- Corrected some details
C99

C99 added a lot of stuff to C89, perhaps too much. Especially a lot of features for scientific computing was added, but also a few things that made life easier for programmers.

// C99 example, ISO/IEC 9899:1999
#include <stdio.h>

size_t mystrcpy(char *restrict s, const char *restrict t)
{
    size_t i;
    for (i = 0; (*s++ = *t++) != \0; i++)
        ;
    return i;
}

int main(void)
{
    char str1[10];
    char str2[] = "Hello, C99!";
    size_t len = mystrcpy(str1, str2);
    size_t i = 0; i < len; i++)
        putchar(str1[i]);
}
The main focus:
- security, eg Anneks K (the bounds checking library, contributed by Microsoft)
- support for multicore systems (threads from WG14, memory model from WG21)

The most interesting features:

- Type-generic expressions using the _Generic keyword.
- Multi-threading support
- Improved Unicode support
- Removal of the gets() function
- Bounds-checking interfaces
- Anonymous structures and unions
- Static assertions
- Misc library improvements

Made a few C99 features optional.
WG14 meeting at Lysaker, April 2015
• Currently working on defect reports
• There are some nasty/interesting differences between C11 and C++11
• IEEE 754 floating point standard updated in 2008
• CPLEX - C parallel language extentions (started after C11)
mystrcpy(s, t)
char *s;
char *t;
{
    int i;
    for (i = 0; (*s++ = *t++) != '\0'; i++)
        return(i);
}

main()
{
    char str1[10];
    char str2[] = "Hello, C78!";
    int len = mystrcpy(str1, str2);
    int i;
    for (i = 0; i < len; i++)
        putchar(str1[i]);
    exit(0);
}

// C99 example, ISO/IEC 9899:1999
#include <stdio.h>

size_t mystrcpy(char *restrict s,
const char *restrict t)
{
    size_t i;
    for (i = 0; (*s++ = *t++) != '\0'; i++)
        return i;
}

int main(void)
{
    char str1[10];
    char str2[] = "Hello, C99!"
    size_t len = mystrcpy(str1, str2);
    size_t i;
    for (i = 0; i < len; i++)
        putchar(str1[i]);
    return 0;
}
Evolution of Keywords in C (1972-2011)
<table>
<thead>
<tr>
<th>auto</th>
<th>goto</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>extrn</td>
<td>return</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td></td>
<td>switch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case</td>
</tr>
</tbody>
</table>
from B to C (1972-1974)

auto
goto
extrn
return
if
else
while
switch
case
<table>
<thead>
<tr>
<th>int</th>
<th>auto</th>
<th>goto</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>extrn</td>
<td>return</td>
<td>else</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td>while</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td>switch</td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
<td>case</td>
</tr>
</tbody>
</table>
from B to C (1972-1974)

<table>
<thead>
<tr>
<th>int</th>
<th>auto</th>
<th>goto</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>extrn</td>
<td>return</td>
<td>else</td>
</tr>
<tr>
<td>float</td>
<td>static</td>
<td></td>
<td>while</td>
</tr>
<tr>
<td>double</td>
<td>register</td>
<td></td>
<td>switch</td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
<td>case</td>
</tr>
</tbody>
</table>
from B to C (1972-1974)

<table>
<thead>
<tr>
<th>int</th>
<th>auto</th>
<th>goto</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>extrn</td>
<td>return</td>
<td>else</td>
</tr>
<tr>
<td>float</td>
<td>static</td>
<td>break</td>
<td>while</td>
</tr>
<tr>
<td>double</td>
<td>register</td>
<td>continue</td>
<td>switch</td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
<td>case</td>
</tr>
</tbody>
</table>
from B to C (1972-1974)

- int
- char
- float
- double
- struct
- auto
- extrn
- static
- register
- goto
- return
- break
- continue
- if
- else
- while
- switch
- case
- default
- do
- for
from B to C (1972-1974)

int char float double struct
auto extrn static register
goto return break continue
if else while switch case default do for
sizeof entry
from B to C (1972-1974)

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
<tr>
<td>struct</td>
<td></td>
</tr>
<tr>
<td>auto</td>
<td></td>
</tr>
<tr>
<td>extern</td>
<td></td>
</tr>
<tr>
<td>static</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>while</td>
<td></td>
</tr>
<tr>
<td>switch</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td></td>
</tr>
<tr>
<td>entry</td>
<td></td>
</tr>
</tbody>
</table>
Early C (1974)

<table>
<thead>
<tr>
<th>int</th>
<th>auto</th>
<th>goto</th>
<th>if</th>
<th>sizeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>else</td>
<td>entry</td>
</tr>
<tr>
<td>float</td>
<td>static</td>
<td>break</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>register</td>
<td>continue</td>
<td>switch</td>
<td></td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
<td>case</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>do</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for</td>
<td></td>
</tr>
</tbody>
</table>
from Early C to K&R C (1974-1978)

int
char
float
double
struct

auto
extern
static
register

goto
return
break
continue

if
else
while
switch
case
default
do
for

sizeof
entry
from Early C to K&R C (1974-1978)

int  char  float  double  struct  short  long  union  unsigned
auto  extern  static  register
goto  return  break  continue
if   else   while  switch  case  default  do  for
sizeof  entry
from Early C to K&R C (1974-1978)

int
char
float
double
struct
short
long
union
unsigned
auto
extern
static
register
goto
return
break
continue
if
else
while
switch
case
default
do
for
sizeof
entry
typedef
# K&R C (1978)

<table>
<thead>
<tr>
<th>int</th>
<th>auto</th>
<th>goto</th>
<th>if</th>
<th>sizeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>else</td>
<td>entry</td>
</tr>
<tr>
<td>float</td>
<td>static</td>
<td>break</td>
<td>while</td>
<td>typedef</td>
</tr>
<tr>
<td>double</td>
<td>register</td>
<td>continue</td>
<td>case</td>
<td></td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
<td>default</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td>do</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td>for</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
from K&R C to ANSI C (1978-1989)

int     auto     goto     if     sizeof
char    extern  return   else   entry
float   static  break    while  typedef
double  register continue switch
double  struct  union    case
struct  short   long     default
short   long    union     do
long    union   unsigned  for
from K&R C to ANSI C (1978-1989)

<table>
<thead>
<tr>
<th>C Keywords (K&amp;R C)</th>
<th>ANSI C Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, char, float, double, struct, short, long, union, unsigned, signed, enum, void</td>
<td>auto, extern, static, register, goto, return, break, continue, if, else, while, switch, case, default, do, for, sizeof, entry, typedef</td>
</tr>
</tbody>
</table>
from K&R C to ANSI C (1978-1989)

int
char
float
double
struct
short
long
union
unsigned
signed
enum
void
auto
extern
static
register
volatile
const
goto
return
break
continue
if
else
while
switch
case
default
do
for
sizeof
entry
typedef
from K&R C to ANSI C (1978-1989)

int char float double struct short long union unsigned signed signed enum void
auto extern static register volatile const
goto return break continue if else while switch case default do for
sizeof entry typedef
The entry keyword came from PL/I and allowed multiple entry points into a function. The keyword was implemented by some compilers but was never standardized.

(stackoverflow.com/questions/254395)
<table>
<thead>
<tr>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>signed</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>auto</td>
</tr>
<tr>
<td>extern</td>
</tr>
<tr>
<td>static</td>
</tr>
<tr>
<td>register</td>
</tr>
<tr>
<td>volatile</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>goto</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>while</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>for</td>
</tr>
<tr>
<td>sizeof</td>
</tr>
<tr>
<td>typedef</td>
</tr>
</tbody>
</table>

**ANSI C (1989)**
from ANSI C to C99 (1989-1999)

int char float double struct short long union unsigned signed signed enum void

auto extern static register volatile const

goto return break continue

if else while switch case default do for
siz eof typedef
from ANSI C to C99 (1989-1999)

- _Bool
- _Complex
- _Imaginary

- int
- char
- float
- double
- struct
- short
- long
- union
- unsigned
- signed
- enum
- void

- auto
- extern
- static
- register
- volatile
- const
- goto
- return
- break
- continue
- if
- else
- while
- switch
- case
- default
- do
- for

- sizeof
- typedef
from ANSI C to C99 (1989-1999)
from C99 to C11 (1999-2011)

_Bool
_Complex
_Imaginary

int    auto    goto    if    sizeof
char   extern  return  else
float  static  break   while
double register continue switch
double volatile continue
struct const
double restrict do
long   inline
long
union
union
short
union
unsigned
union
signed
double
void
void
from C99 to C11 (1999-2011)
from C99 to C11 (1999-2011)

_int
_char
_float
_double
_struct
_short
_long
_union
_unsigned
_signed
_enum
_void
_auto
Extern
_static
_register
_volatile
_const
_restrict
_inline
_goto
_return
_break
_continue
_if
_else
_while
_switch
_case
_default
_do
_for
 sizeof
typedef
_alignas
_atomic
_thread_local
from C99 to C11 (1999-2011)

<table>
<thead>
<tr>
<th>C99</th>
<th>C11</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td></td>
</tr>
<tr>
<td>_Complex</td>
<td></td>
</tr>
<tr>
<td>_Imaginary</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>auto</td>
</tr>
<tr>
<td>char</td>
<td>extern</td>
</tr>
<tr>
<td>float</td>
<td>static</td>
</tr>
<tr>
<td>double</td>
<td>register</td>
</tr>
<tr>
<td>struct</td>
<td>volatile</td>
</tr>
<tr>
<td>short</td>
<td>const</td>
</tr>
<tr>
<td>long</td>
<td>restrict</td>
</tr>
<tr>
<td>union</td>
<td>inline</td>
</tr>
<tr>
<td>unsigned</td>
<td>_Alignas</td>
</tr>
<tr>
<td>signed</td>
<td>_Atomic</td>
</tr>
<tr>
<td>enum</td>
<td>_Thread_local</td>
</tr>
<tr>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C99</th>
<th>C11</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td></td>
</tr>
<tr>
<td>do</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td></td>
</tr>
<tr>
<td>switch</td>
<td></td>
</tr>
<tr>
<td>while</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td></td>
</tr>
<tr>
<td>typedef</td>
<td></td>
</tr>
<tr>
<td>_Noreturn</td>
<td></td>
</tr>
<tr>
<td>_Static_assert</td>
<td></td>
</tr>
<tr>
<td>_Alignof</td>
<td></td>
</tr>
<tr>
<td>_Generic</td>
<td></td>
</tr>
</tbody>
</table>
The spirit of C

trust the programmer
• let them do what needs to be done
• the programmer is in charge not the compiler

keep the language small and simple
• small amount of code → small amount of assembler
• provide only one way to do an operation
• new inventions are not entertained

make it fast, even if it's not portable
• target efficient code generation
• int preference, int promotion rules
• sequence points, maximum leeway to compiler

rich expression support
• lots of operators
• expressions combine into larger expressions

The history of C
At Bell Labs. Back in 1969. Ken Thompson wanted to play. He found a little used PDP-7. Ended up writing a nearly complete operating system from scratch. In pure assembler of course. In about 4 weeks! Dennis Ritchie soon joined the effort. While porting Unix to a PDP-11 they invented C, heavily inspired by Martin Richards’ portable systems programming language BCPL. In 1972 Unix was rewritten in C, and later ported to many other machines aided by Steve Johnson's Portable C Compiler. C gained popularity outside the realm of PDP-11 and Unix. Initially the K&R was the definitive reference until the language was standardized by ANSI and ISO in 1989/1990 and thereafter updated in 1999 and 2011.
To get a deep understanding of C++, it is useful to know the history of this wonderful programming language. It is perhaps even more important to appreciate the driving forces, motivation and the spirit that has shaped this language into what we have today.

We assume you know the history and spirit of C. We will now include Simula, Algol 68, Ada, ML, Clu into the equation. We will discuss the motivation for creating C++, and with live coding we will demonstrate by example how it has evolved from the rather primitive “C with Classes” into a supermodern and capable programming language as we now have with C++11/14 and soon with C++17.

A lightning talk at ACCU 2015, April 23, Bristol, UK
The history of C++
The history of C++ in 5
The history of C++ in 5 minutes
Before C++

with approximately the words of Bjarne Stroustrup himself as copied from "The Design and Evolution of C++", Bjarne Stroustrup, 1994
I was working on my PhD thesis
in the Computing Laboratory at
in the Computing Laboratory at University of Cambridge.
I was working on a simulator to study alternatives for the organization of system software for distributed systems.

The initial version of this simulator was written in Simula

```plaintext
Begin
  class Glyph;
    virtual procedure print is procedure print;
  begin;
  end;

  glyph class char (c);
    character c;
  begin;
    procedure print;
      outchar(c);
    end;

  glyph class line (elements);
    ref (glyph) array elements;
  begin;
    procedure print;
    begin;
      integer i;
      for i := 1 step 1 until upperbound (elements, 1) do
        elements (i).print;
      outimage;
    end;
  end;

  ref (glyph) rg;
  ref (glyph) array rgs (1..4);

  main_program:
    rgs (1):= new char ('A');
    rgs (2):= new char ('b');
    rgs (3):= new char ('b');
    rgs (4):= new char ('a');
    rg:= new line (rgs);
    rg.print;
end;
```

http://en.wikipedia.org/wiki/Simula
and ran on the IBM 360/165 mainframe.

System/370 model 165
The concepts of Simula and object orientation became increasingly helpful as the size of the program increased. Unfortunately, the implementation of Simula did not scale the same way.
Eventually, I had to rewrite the simulator in ? and run it on the experimental CAP computer.
Eventually, I had to rewrite the simulator in BCPL and run it on the experimental CAP computer.
The experience of coding and debugging the simulator in BCPL was horrible. BCPL makes C look like a very high-level language and provides absolutely no type checking or run-time support.
The experience of coding and debugging the simulator in BCPL was horrible. BCPL makes C look like a very high-level language and provides absolutely no type checking or run-time support.
Upon leaving Cambridge, I swore never again to attack a problem with tools as unsuitable as those I had suffered while designing and implementing the simulator.

A good tool should:
- have support for program organization, eg classes, concurrency, strong type checking
- produce programs that run as fast as the BCPL programs
- support separately compiled units into a program
- allow for highly portable implementations
After finishing my PhD Thesis in Cambridge I got a job at
After finishing my PhD Thesis in Cambridge I got a job at Bell Labs.
Where I learned C properly from people like Stu Feldman, Steve Johnson, Brian Kernighan, and Dennis Ritchie.
Developing the initial version of C++ (pre-1985)
• Simula gave classes
- Simula gave classes
- Algol68 gave operator overloading and references
• Simula gave classes
• Algol68 gave operator overloading and references
• Algol68 also gave the ability to declare variables anywhere in a block
• Simula gave classes
• Algol68 gave operator overloading and references
• Algol68 also gave the ability to declare variables anywhere in a block
• The only direct influence from BCPL was
• Simula gave classes
• Algol68 gave operator overloading and references
• Algol68 also gave the ability to declare variables anywhere in a block
• The only direct influence from BCPL was // comments
Development of C++ (post-1985)
ML (Robin Milner, 1973) influenced exceptions

```ocaml
fun factorial n = let
  fun fac (0, acc) = acc
  | fac (n, acc) = fac (n - 1, n * acc)
  in
  if (n < 0) then raise Fail "negative argument"
  else fac (n, 1)
end
```
CLU (Barbara Liskov, 1974) also influenced exception handling

```plaintext
sum_stream = proc (s: stream) returns (int) signals (overflow,
    unrepresentable_integer(string),
    bad_format(string))

sum: int := 0
num: string

while true do
    % skip over spaces between values; sum is valid, num is meaningless
    c: char := stream$sgetc(s)
    while c = '' do
        c := stream$sgetc(s)
    end
    % read a value; num accumulates new number, sum becomes previous sum
    num := ""
    while c ~= '' do
        num := string$append(num, c)
        c := stream$sgetc(s)
    end
    except when end_of_file: end
    % restore sum to validity
    sum := sum + $2i(num)
end
except when end_of_file: return(sum)
    when unrepresentable_integer: signal unrepresentable_integer(num)
    when bad_format, invalid_character (s): signal bad_format(num)
    when overflow: signal overflow
    end
end sum_stream
```
Ada (Jean Ichbiah++, 1980) influenced templates, namespaces and exceptions

```ada
with Ada.Text_IO;
package body Example is

  i : Number := Number'First;

  procedure Print_and_Increment (j: in out Number) is

    function Next (k: in Number) return Number is
      begin
        return k + 1;
      end Next;

    begin
      Ada.Text_IO.Put_Line ( "The total is: " & Number'Image(j) );
      j := Next (j);
      end Print_and_Increment;

-- package initialization executed when the package is elaborated

begin
  while i < Number'Last loop
    Print_and_Increment (i);
  end loop;
end Example;
```
80's
C with classes, C++/CFront, ARM
C++ was improved and became standardized

90's

X3J16, C++arm, WG21, C++98, STL
Ouch...Template Metaprogramming
C++03, TR1, Boost and other external libraries

While the language itself saw some minor improvements after C++98, Boost and other external libraries acted like laboratories for experimenting with potential new C++ features. Resulting in...
With the latest version C++ feels like a new language
The future of C++?