C

Stein Dale
stein.dale@ntnu.no

Eirik A. Nygaard
eirikald@pvv.ntnu.no

The origin of C

C is quirky, flawed and an enormous success.

- Dennis M. Ritchie

- C invented and first implemented on DEC PDP-11 on UNIX in the 1970s
- Originates from BCPL and B
- Originally described in Kernighan & Ritchie "The C Programming language" (1978)
- Socalled structured, middle level programming language
- Highly portable
- Denoted "Programmer' s language"

C

TPG4162 Essentials

The mandatory ”Hello world!”

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

int main(void)
{
        printf(“Hello, world!\n”);

        return EXIT_SUCCESS;
}
```

- C is block structured
- variables and functions have a defined scope

Comments

- Two forms:

  /* A possibly multi line comment */

  // Strictly a one line comment

```
const float PI = 3.1415;    // Set the constant PI


/************************************************
 * This function takes the two edge lengths of
 * a rectangle as input for calculating its area.
 ***********************************************/

int rect_area(int a, int b)
{
        return a*b;
}
```

Built-in data types

- Data types
  - char
  - int
  - float
  - double
  - void

- Modifiers
  - signed
  - unsigned
  - long
  - short

  - modifiers do not apply to void

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

int main(void)
{
  int    a = 5;
  float  b = 0.51;
  double c = 50.5e-5;
  char   d = 'a';

  printf("a: %d, b: %f, c: %f, d: %c/%d \n“,
              a, b, c, d, d);

  return EXIT_SUCCESS;
}
```

```
 a: 5, b: 0.510000, c: 0.000505, d: a/97
```

Variables

- Declarations and definitions

  type variable name;

  type variable name = value;

  type var1 = val1, var2 = val2;
- NB: upper-/lowercase matters
- Should initialize variables

```
int      iNumParts  = 10;             // signed 32-bit int
float    fTolerance = 0.15;           // ~6 digit precision
double dEpsilon    = 0.000000001;  // ~12 digit precision
```

Arrays

- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by an index
- C inflicts nil-based indexing

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

int main ()
{
    // print all elements of the array

    for (int i=0; i<5; i++)
    {
        printf("%d ", myArray[i]);
    }
}
```

Array init ialization in global scope

Nil-based indexing

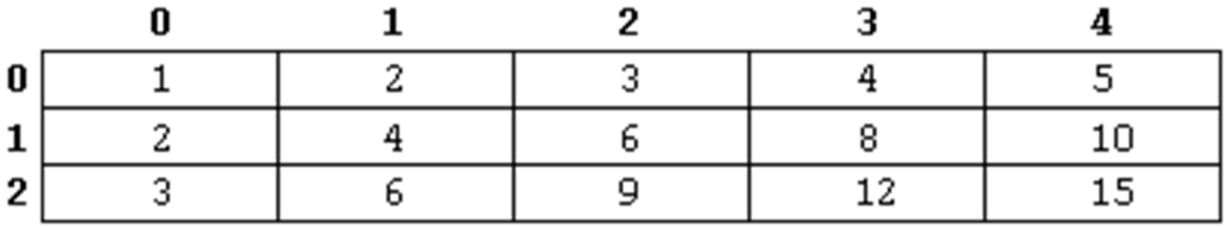
Arrays: multi-dimensional

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
    for (n=0; n<HEIGHT; n++)
    {
        for (m=0; m<WIDTH; m++)
        {
                jimmy[n][m] = (n+1)*(m+1);
        }
    }
    return 0;
}
```

Global scope



	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Strings

- In C, a string is defined as a character array terminated by an ASCII zero, i.e. ‘\0’
- Thus necessary to declare a character array to be one character longer than the longest string to hold
- Manipulation and comparison of strings performed by wide range of functions, e.g.:
  - strcpy(s1, s2); // copies s2 into s1
  - strcat(s1, s2); // concatinates s2 onto end of s1
  - strlen(s); // returns length of s
  - strcmp(s1, s2); // returns

0	if s1 and s2 is the same
< 0	if s1 < s2
> 0	if s1 > s2

Strings

```
char sName = “Jon”;    // nil-terminated string constant, init legal in global scope

main()
{
        char sName2[80];

        strcpy(sName2, sName);

        // When strings are equal, difference is zero.
        if (strcmp(sName, sName2) == 0) printf(“strings equal);

        strcat(sName2, “ Kleppe”);

        printf(“Institute Mgr IPT: %s \n”, sName2);
}
```

Assignment

- var = expression;

```
a = 10;
```

```
b = 15.0 + 25.5;
```

```
str = “Some string”;
```

Only possible in global scope! Elsewhere use built-in functions

Binary operations

```
if (a < 10 && b > 15)
{
        ...
}
```

```
a += 10; // a = a + 10
```

Unary operations

- --
- ++
- sizeof

- !
- ~
- -

```
int a = -10;
int b = ++a;
```

Functions

```
int square(int x);
```

```
int square(int x)
{
        return x * x;
}
```

```
void do_stuff(int x)
{
        if (x > 10) return;
        ...
}
```

- In C, these are all called functions
- does not distinguish between subroutines and functions (like FORTRAN)

Functions

```
int area(int a, int b)
{
        return a * b;
}
```

```
int area();

int b = area(10);
```

avoids erroneous usage

```
void worker(int x, int y)
{
        return x * y;
}
```

avoids wrong return

C is strong-typed:

- function arguments must match in type and number
- return types are type checked

Benefits enormous in large code projects and sw maintenance

If

```
if (expression)
{
    do_work();
}
```

```
if (something)
{
    do_special_work();
}
else
{
    do_other_work();
}
```

```
if (something)
{
    do_work();
}
else if (something_else)
{
    do_secret_work();
}
else
{
    do_other_work();
}
```

Note: recommended use of brackets to hinder mistakes during code extension

Switch

```
int a = 2;

switch (a)
{
        case 1:
                puts(“one”);
                break;

        case 2:
                puts(“two”);

        case 3:
                puts(“three”);
                break;

        default:
                puts(“unknown”);
                break;
}
```

Lacks a break!
What happens?

Loops

```
for (int i = 0; i < 10; i++)
{
        printf(“Hello”);
}
```

```
for (i = 0; i < 10; i++)
{
        if (something)              break;
        else if (something_else)          continue;
     ...
     ...
}
```

***Recommendation**: omit brackets only for single line statements*

```
while (a < 10)
{
        a = 2*a;
}
```

```
do
{
        printf(“Haha, funny!”);

} while ( StillFunny(i++) );
```

Binary operations (1)

	Logical Operators
AND	&&
OR	||
NOT	!

Relational Operators

<

>

<=

>=

==

!=

Binary operations (2)

Aritmetic operators combinable e.g. with assignment

PLUS	+
MINUS	-
MULTIPLY	*
DIVISION	/
MODULO	%

Bit Operators
(advanced)

&	AND
|	OR
^	XOR
<<	SHIFT LEFT
>>	SHIFT RIGHT

File opening

- To open a file you need to use the **fopen** function, which returns a **FILE*** pointer [more on pointers later]

- The FILE pointer is used to:
  - distinguish an actual file from others wrt operations
  - to change current position in the file
  - perform i/o operations

- **fopen** returns NULL on failure

File opening

- Signatures:

  FILE *fopen(const char *_filename_, const char *_mode_);

  NB: use additional backslash if filename contains ‘\’

- _mode_ may assume the following values [text files]:
  - _r_ : read only
  - _w_ : write (file need not exist)
  - _a_ : append (file need not exist)
  - _r+_ : open for reading and writing, start at beginning
  - _w+_ : open for reading and writing (overwrite file)
  - _a+_ : open for reading and writing (append if file exists)

- Add "_b_" if file is binary, i.e. "rb", "wb", "ab", "r+b", "w+b", "a+b"

File positioning

- Positioning in an open file is performed by:

  int fseek(*FILE* **pFile*, ***long*** *offset*, ***int*** *origin***);**

- Arguments:
  - pointer to FILE structure to relocate position indicator
  - offset: number of bytes to offset position indicator with regard to origin
  - origin is an integer specifying origin position:
    - SEEK_SET: origin is the start of file
    - SEEK_CUR: origin is the current position
    - SEEK_END: origin is the end of file

- Return value
  - zero: function performed successfully
  - nonzero: an error occurred

- fseek uses 32 bit FILE* on some platforms => limit 2 GB seeks

File Reading

size_t fread (void* *ptr*, size_t *size*, size_t *nmemb*, FILE * *pFile*)

fread reads *nmemb* data items of size *size* from the named input stream into the array pointed to by *ptr*.

- **fread** stops when
  - *nmemb* items have been read
  - unexpected occurrence, e.g. file read error, encountering end of file,…
- Each data item is a sequence of bytes of length *size*.
- Upon return, **fread** sets the file pointer to the byte past the last byte that has been read
- **fread** returns the number of items actually read

File reading

```
#include <stdio.h>

const int NUMSAMPLES =  1750;                    // number of samples

char matrix[NUMSAMPLES];                         // global array


void ReadSEGYTrace1Samples (char* sFileName)
{
    FILE *pFile = fopen(sFileName, "rb");

    fseek(pFile, 3600, SEEK_SET);                        // skip file header
     fseek(pFile,   240, SEEK_CUR);                      // skip trace header

    fread(matrix, 1, NUMSAMPLES, pFile);         // more on this later….


    fclose(pFile);
}
```

Pointers

- Correct understanding is critical to successful C programming as pointers are:
  - the means by which functions can modify calling arguments
  - used to support C’s dynamic allocation routines
  - can improve the efficiency of certain routines
- Pointers are one of C’s strongest features, but also the most dangerous feature, e.g.:
  - uninitialized pointers (wild pointers) => program crash
  - easily misused which may cause bugs that are hard to find

Pointers

- a = &b; // adress of
- *a = 21; // dereference assignment
- b = *a; // dereference assignment

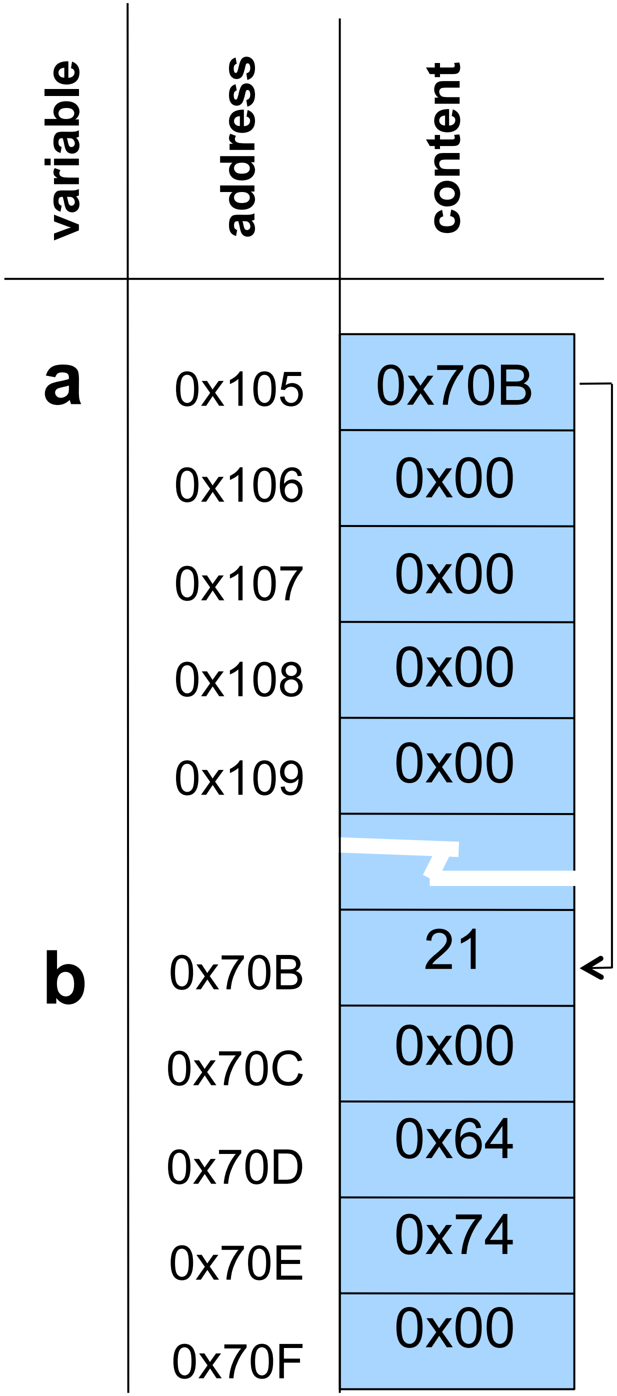
```
void main(void)
{
   int *a;      // define a pointer to a mem location interpreted as int
   int  b;      // define a memory location holding an int

   b = 21;      // b is set to 21
   a = &b;      // let a point to the location of b

   *a = 42;     // mem location pointed to by a set to int value 42

   printf("*a:%d, b:%d \n",*a, b);
}
```

What happens?



variable	address	content
a	0x105	0x70B
	0x106	0x00
	0x107	0x00
	0x108	0x00
	0x109	0x00
b	0x70B	21
	0x70C	0x00
	0x70D	0x64
	0x70E	0x74
	0x70F	0x00

Arrays and pointers

a[5] = 10; equivalent to *(a + 5) = 10;

```
void strcpy(char *s, char *t)
{
   int i=0;
   while ((s[i]=t[i]) != ‘\0’) i++;
}
```

Array example

```
void strcpy(char *s, char *t)
{
   while ((*s++ = *t++) != ‘\0’);
}
```

Pointer example

```
void strcpy(char *s, char *t)
{
  while(*s++ = *t++);
}
```

Pointer example condensed

Pointers: Memory allocation

- C malloc(), calloc(), realloc(), free()
- C++ new, delete

STL

Standard template library

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

```

int* a = malloc(100*sizeof(int));

if (!a) … handle allocation error …

a[0] = 10;
…

free(a);
```

C++ new

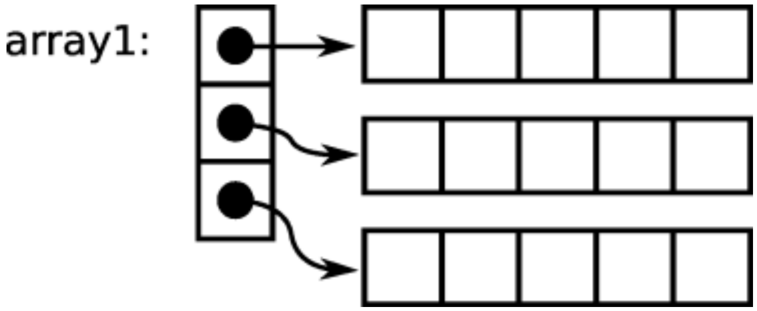
Pointers: heap allocation

```
int **array1 = malloc(nrows * sizeof(int *));

for(i = 0; i < nrows; i++)
{
        array1[i] = malloc(ncolumns * sizeof(int));
}
```

C++ new

C++ new



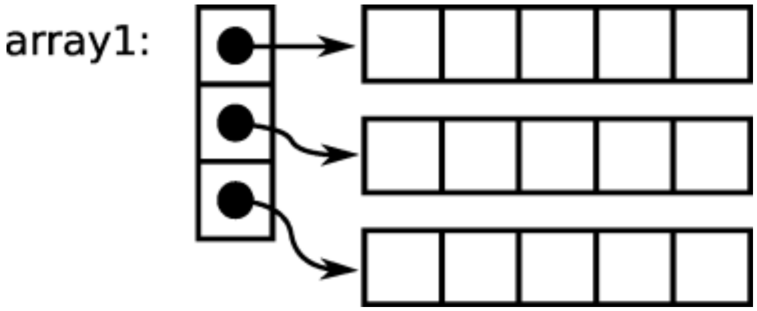
Pointers: free memory

```
for(i = 0; i < nrows; i++)
{
     free((void*) array1[i]);
}

free((void *)array1);
```

C++ delete

C++ delete



Parameter passing

- **Pass by value:** function operates on a
  copy of input parameter
  thus preventing any side effect to caller of the function.

variable

- **Pass by reference**: parameter to a function is a
  reference
  providing direct access to a variable of caller.

pointer

Parameter passing: examples

Pass by value

```
#include <stdio.h>

void foo(int x)
{
    printf("In foo(): %d\n", x);
    x = 10;
    printf("In foo(): %d\n", x);
}

int main(void)
{
    int i = 5;

    printf("In main(): %d\n", i);
    foo(i);
    printf("In main(): %d\n", i);

    return 0;
}
```

```
In main(): 5
In foo(): 5
In foo(): 10
In main(): 5
```

Pass by reference

```
#include <stdio.h>

void foo(int *x)
{
    printf("In foo(): %d\n", *x);
    *x = 10;
    printf("In foo(): %d\n", *x);
}

int main(void)
{
    int i = 5;

    printf("In main(): %d\n", i);
    foo(&i);
    printf("In main(): %d\n", i);

    return 0;
}
```

```
In main(): 5
In foo(): 5
In foo(): 10
In main(): 10
```

File reading revisited

```
#include <stdio.h>

char matrix[13483][1750];                      // global byte array


void ReadSEGYfile(char* sFileName)
{
    FILE* pFile = fopen(sFileName, "rb");      // fopen() fills in the FILE structure and returns a pointer to the data
    fseek(pFile, 3600, SEEK_SET);              // skip file header

    for (int i = 0; i < 13483; i++)            // loop over number of traces
    {
        fseek(pFile,   240, SEEK_CUR);         // skip 240B trace header

        for (int j = 0; j < 1750; j++)         // loop through the 1750 samples
        {
            fread(&matrix[i][j], 1, 1, pFile); // read each sample (1 byte each)
        }
    }
    fclose(pFile);
}
```

C

Additional constructs

Ternary conditional

```
(expression) ? do_work() : do_other_work();
```

```
bool bGreaterThan10 = (iCount > 10) ? TRUE : FALSE;
```

Data structures

struct

```
struct sphere
{
        int x, y;
        float r;
};
```

```
struct sphere ball;

ball.x = 10;
ball.y = 5;
ball.r = 0.5;
```

union

```
union object
{
        double size;
        int sheep;
};
```

- Seldom necessary
- Platform concerns

Enum

- Name your constants

```
enum VEHICLE
{
        CAR,
        BOAT,
        PLANE,
};
```

```
enum VEHICLE used_vehicle = CAR;

if (used_vehicle == CAR)
{
        ...
}
```

Preprocessor

- #include <stdio.h>
- #include “myheader.h”
- #define MY_PI 3
- #ifdef MY_PI ... #endif
- #if expr ... #elif expr ... #else ... #endif
- #pragma

Preprocessor example: include guard

- Ensure a header file is included just once.

```
// a.h

#ifndef __A_H
#define __A_H

#include “b.h”
pos_t *my_position;

#endif
```

```
// b.h

typedef struct
{
        int x, y;
} pos_t;
```

```
// a.c

#include "a.h"
#include "b.h"

int main(void)
{
   pos_t *pos;
   ........

   return 0;
}
```

- Use #ifndef, #define, #endif

Typedef

- Make your own types

```
typedef void *pointer;
```

```
typedef struct position
{
        int x;
        int y;

} position_t;
```

Casting

- Bypass type checking
- Useful when dealing with void pointers

```
float fValue = 3.14;
int   iValue = (int) fValue;

printf(“%d \n”, iValue);
```

Guess the value...?

Type qualifiers

- const
- volatile
- static

```
const int a = 10;

volatile char reg = '\xED';

static int b = 255;

void counter(void)
{
        static int c;

        return c++;
}
```

Function pointers

```
int bar(int count, char *name)
{
        printf(“I am: %s \n”, name);

        return count << 3;
}


void foo(void)
{
        int (*function)(int count, char *name);
        function = bar;

    function(0xA, “Ritchie”);
}
```

Goto

```
int iArray = malloc(100*sizeof(int));
if (!iArray) goto CleanUp;

float fArray = malloc(100*sizeof(float));
if(!fArray) goto CleanUp;...
...

CleanUp:
if (iArray) free (iArray);
if (fArray) free (fArray);
```

Seldomly justifiable

```
puts(“Starting”);

ourloop:
puts(“In the loop”);

goto ourloop;
```

Typically completely unneccesary