The Programming Language C

- Designed in the 1970s by Dennis Ritchie for/with UNIX
- Implementation language of UNIX
- Improved version standardised as "**ANSI C**" (1989)
- ***'C is a relatively "low level" language.'***
- Popular for system programming
- "Portable Assembler"
- Latest standard: **C99** (ISO, 1999)
  — only minor changes over ANSI C

C as Programming Language

- Imperative
- Block-structured, but no nested functions
- Weak static typing
- Only machine-oriented primitive types
- Arithmetic overloaded on number types
- Only array and pointer type constructions are polymorphic
- Only low-level data type constructions
- No automatic memory management
- ***Pointer arithmetic***

Operator Precedence

() [] → .	left to right
! ~ ++ −− + − * & (*type*) sizeof	right to left
* / %	left to right
+ −	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
_ ? _ : _	right to left
= += −= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Order of evaluation is **unspecified!**

Type Conversions: Promotions and Casts

- "Narrower" operands are automatically converted into "wider" type.
- Assignment of "wider" values to "narrower" variables may produce a warning
- A char is an 8-bit (signed or unsigned) integer
- An explicit **type cast** is a prefix operator "(typename)"

`if`

Conditional statement ("**double-selection statement**"):

$$\textbf{\textit{if}}\ condition\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2$$

In C:

$$\texttt{if (}\ condition\ \texttt{)}\ S_1\ \texttt{else}\ S_2$$

The "**single-selection statement**"

$$\texttt{if (}\ condition\ \texttt{)}\ S_1$$

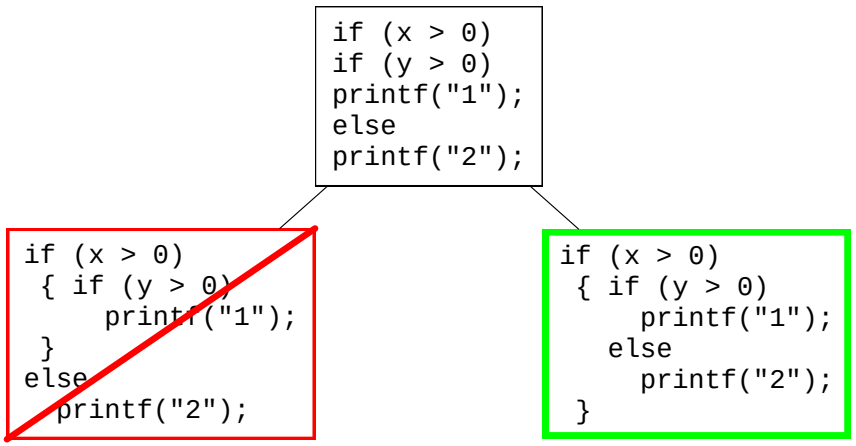
is an abbreviation for:

$$\texttt{if (}\ condition\ \texttt{)}\ S_1\ \texttt{else \{\}}$$

It is good style to make this explicit!

Dangling `else`

What is the structure of the following "nested `if`"?



Every `else` belongs to the "***closest possible***" `if`.

Conditional Expressions

- Many (functional) programming languages have **conditional expressions**:

  $max\ (x,y)\ =\ \textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y$
  $fact\ n\ =\ \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * fact\ (n-1)$
- C has conditional expressions, too, but with a strange syntax:

  $$condition\ ?\ expr_1\ :\ expr_2$$

  (One-way conditional **expressions** would be nonsense!)

  Example:

  float *floatMax*(float *x*, float *y*)
  { return *x* > *y* ? *x* : *y* ; }

A Better Syntax for Conditional Expressions

```
#define IF ((                          /* condExp.c */
#define THEN )?(
#define ELSE ):(
#define ENDIF ))

float floatMax(float x, float y)
{ return IF x > y THEN x ELSE y ENDIF; }

#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("%s\n", IF argc > 1 THEN argv[1]
                 ELSE "<no arguments>"
          ENDIF);
  return 0;
}
```

Conditional Evaluation

"An expression containing && or || operators is evaluated only until truth or falsehood is known."

x && y ≡ *IF x THEN y ELSE FALSE ENDIF*

x || y ≡ *IF x THEN TRUE ELSE y ENDIF*

Therefore the following is safe:

```
if (x ≠ 0 && 12 / x < q)
   k = 7 / x;
else
   fprintf(stderr, "There is a problem.\n")
```


C-Truth

- Modern languages have a predefined datatype for truth values, also called **Boolean** values
- C allows values of any integral type (including characters and pointers) to be used in conditions.

Truth-value use of integral values:

Integral	—*interpretation* →	**Truth**	—*translation* →	Integral
0		False		0
non-zero		True		1

Better than nothing:

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```


Truth Tables

x	*y*	$x \wedge y$	$x \vee y$
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

C-Truth Tables

Integral	—*interpretation* →	**Truth**	—*translation* →	Integral
0		False		0
non-zero		True		1

x	y	*x && y*	*x || y*
0	0	0	0
0	non-zero	0	1
non-zero	0	0	1
non-zero	non-zero	1	1

The Comma Operator

> "A pair of expression separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand."
>
> Kernighan & Ritchie

Contrived Example: `k = (printf("initialising k\n"), 1);`

Comma expressions are frequently used in loop headers:

```
k = 0;
while ( k++, k*k < n ) {}
printf("k = %d\n", k);
```

Usually **not good style!**

Expressions versus Statements in C

- Adding a semicolon turns **any expression** into a statement.

  `6 * 7;` — *comma.c:10: warning: statement with no effect*

- **Assignments** and **function calls** are ***expressions:***
  `k = printf("%d\n", m) + 7;`
  `m = 5 * (k = k + 1);`
- Type and value of an **assignment** are the type of the assigned l-value and the value assigned to it.
- Return type and value of **standard library functions** are documented e.g. in their `man` pages.
- Sequencing of expressions: the **comma** operator
- Sequencing of statements: juxtaposition

Updating Assignment Operators

`var += expr;`	abbreviates	`var = var + expr;`
`var -= expr;`	abbreviates	`var = var - expr;`
`var *= expr;`	abbreviates	`var = var * expr;`
`var /= expr;`	abbreviates	`var = var / expr;`
`var %= expr;`	abbreviates	`var = var % expr;`

Compare readability:

```
myArray[3*k+i][j-5] += d + 2 * k;
```

```
myArray[3*k+i][j-5] = myArray[3*k+i][j-5] + d + 2 * k;
```


Increment and Decrement Operators

Used as **statements**,

`var++;`	and	`++var;`	abbreviate	`var += 1;`
`var--;`	and	`--var;`	abbreviate	`var -= 1;`

Assignment used as expression returns the assigned value:

```
float x, y = 3.5; int k;
x = 4.2 + (k = y);          /* k == 3  &&  x = 7.2 */
```

Used as **expressions**,

`++var`	abbreviates	`(var = var + 1)`
`var++`	abbreviates	`(var0 = var, var += 1, var0)`

The Dangers of Increment and Decrement Operators

```
#include <stdio.h>   /* incr.c */
int main() {
  int k=13, m, p;

  m = 2;
  p = ++m * (k - ++m);
  printf("p = %d\n", p);

  m = 2;
  p = (k - ++m) * ++m;
  printf("p = %d\n", p);

  return 0;
}
```

```
p = 36

p = 40
```

Expressions with side-effects are dangerous!

Lessons

- **Never** have more than one side-effecting subexpression in a single expression!
- **Don't rely** on the **expression evaluation order** of one particular implementation!
- **Don't overuse** "++" and "- -"
- **Always** use `-Wall`
- **Always** strive to understand warning and error messages!
- **Always** heed warnings!

The while Repetition Statement

while (*condition*) *body*

Intuitive meaning:

"While *condition* evaluates to true, execute *body*."

A ***single* statement** has to be given as *body*:

- often, *body* is a block,i.e., a sequence of statements enclosed in braces { ... }
- quite frequently, the body is empty, visible as one of
  - an empty *expression statement* ";"
  - an empty *block* "{}"

do ... while

For loops where the body needs to be executed once **before** the condition is first tested:

```
do
   body
while ( condition )
```

≡

```
body
while ( condition )
   body
```

body has to be a single *statement.*

It is recommended

- to enclose *body* in braces, even when it is a primitive statement
- to **indent** the *body*

`for` Loops in Oberon

```
FOR v := beg TO end BY step DO statements END
```

"A `for` statement specifies the repeated execution of a statement sequence while a progression of values is assigned to an integer variable called the *control variable* of the `for` statement."

In Oberon, as in most languages with `for` loops, the iteration bounds are calculated **once**, before the body is first executed.

for Statements in C

First explanation:

```
for ( init ; condition ; step )
   body
```

≡

```
init;
while ( condition )
 {
   body;
   step
 }
```

Typical use — "proper for loop":

```
for ( k = start ; k ≤ end ; k++ )
   body
```

One kind of **atypical** use — while loop:

```
for ( ; condition ; ) body
```


break and continue

- break terminates execution of the enclosing for, while, do ... while, switch statement
- continue transfers control to the end of the *body* of for, while, do ... while statements
  - in while and do ... while statements, the *condition* is tested, and, if applicable, looping continued
  - in for statements, *step* is executed before testing the *condition*

continue in for statements

Explaining also continue in for statements:

```
for ( init ; condition ; step )
   body
```

≡

```
init;
if ( condition )
 {
   do
      body;
   while ( step , condition )
 }
```

break **and** continue **in Nested Loops — Problem**

```
int i, j;
for ( i = 1; i ≤ 10; i++ )
 {
  for ( j = 1; j ≤ 10; j++)
   {
    if (11111% (i * j) == 0)
      break;    /* breaks only out of inner loop! */
    ...
   }
  ...
}
```

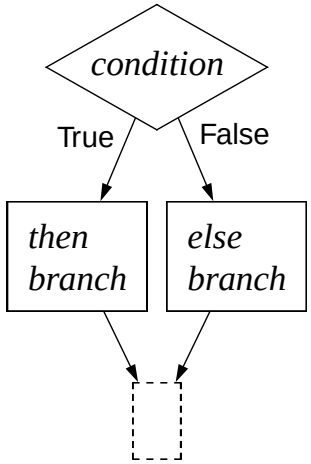
Some languages allow to break out of specified loops.

break **and** continue **in Nested Loops — One Solution**

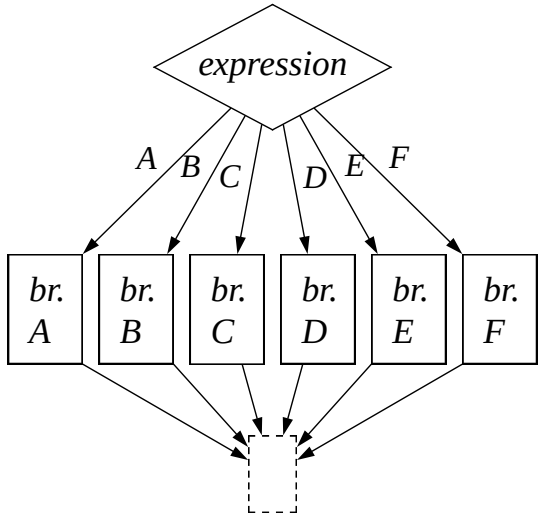
```
int i, j;
BOOL break_i;
for ( i = 1, break_i = FALSE ; i ≤ 10; i++ )
 {
  for ( j = 1; j ≤ 10; j++)
   {
    if (11111% (i * j) == 0)
      { break_i = TRUE; break; } /* breaks inner loop */
    ...
   }
  if (break_i)
   { break_i = FALSE; break; } /* breaks outer loop */
  ...
}
```


Multiple Selection

Double Selection

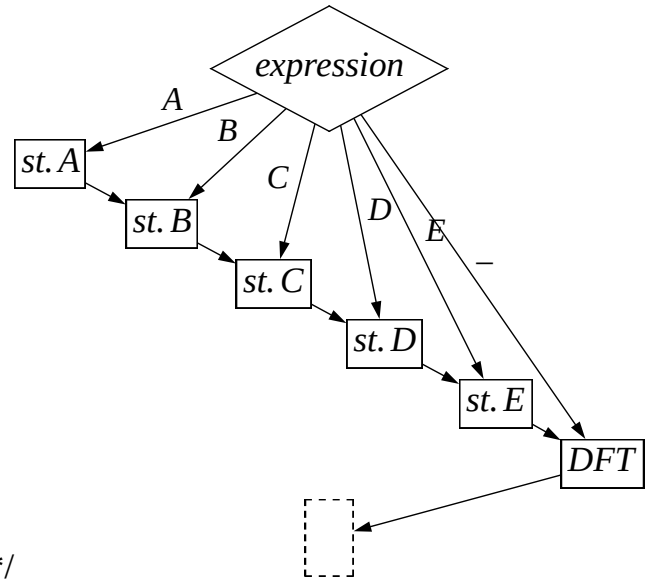


Multiple Selection



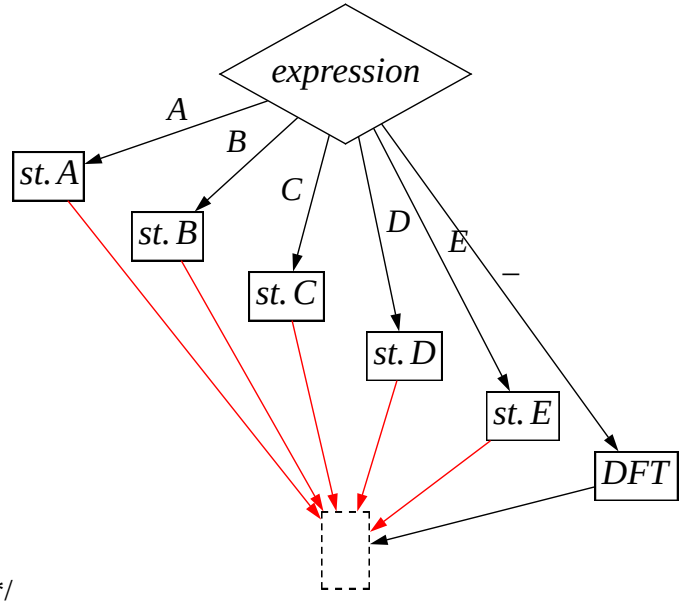
switch **Statement**

```
switch (expr) {
  case A:
    stmtsA;
  case B:
    stmtsB;
  case C:
    stmtsC;
  case D:
    stmtsD;
  case E:
    stmtsE;
  default:
    stmtsDFT;
} /* end switch */
```



switch used as Multiple-Selection Statement

```
switch (expr) {
  case A:
    stmtsA;
    break;
  case B:
    stmtsB;
    break;
  case C:
    stmtsC;
    break;
  case D:
    stmtsD;
    break;
  case E:
    stmtsE;
    break;
  default:
    stmtsDFT;
} /* end switch */
```



switch — Mixed Style

```
switch (c) {
  case 'A':
  case 'a':
    stmtsA;
    break;
  ...
  case ' ':
    stmtsS;
  case '\n':
  case '\t':
    stmtsC;
    break;
  ...
  default:
    stmtsDFT;
} /* end switch */
```

Attention:

It is easy to get mixed up here!

- **indent carefully!**
- **document case groups!**

Subprograms

- A **subprogram** is a (parameterised) fragment of a program.
- A **subprogram call** is an instantiation of a subprogram with *actual parameters.*
  - **Function** calls are expressions
  - **Procedure** calls are statements
- The purpose of introducing subprograms is **modularisation**.
- Modular components are accessed via **interfaces** — the interface of a subprogram consists of:
  - **type:** argument types, result type
  - **specification:** properties, description of effects

Subprograms in C

- Every expression can be used as a statement:
  - No procedures necessary — only **functions**
  - Functions with return type void are "intended as procedures"
  - Many functions that are often used as procedures have non-void return types

    — know and check!
- Types of functions are formally captured in "**prototypes**"
- No further part of function specifications is ***formally*** supported by C

Function Types and Prototypes

Mathematics	**C**
$\sin : \mathbb{R} \to \mathbb{R}$	double *sin*(double *x*);
$\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	int *gcd*(int *k*, int *m*);
$pow : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$	double *pow*(double *x*, double *y*);

Prototypes are function **declarations**.

- Prototypes are *implied* by (ANSI-style) function **definitions.**
- The common part is also called **function header**.
- After the prototype has been seen by the compiler, the function name and its type are known.
- Prototypes can be used as "forward-declarations".
- `*.h` files frequently contain `extern` prototypes.

Local Variable — Global Variables

```
int k;

int f(double h)
{
  int n;
  ...
}
```

- *k* is a **global variable**
- *n* is a **local variable**
- *h* is a **formal parameter** — inside the body this is equivalent to a **local variable**

Scope and Instances of Variables in C

- **All variables** are *visible* only **after declaration**
- **Global variables** are *visible* **in the file of their declaration**
- **Local variables** are *visible* **in the block of their declaration**

- For **all variables**, an instance is created when control flow passes their **definition**.
- **Global variables** have only **one** instance
- static **(local) variables** have only **one** instance
- **Local variables** have **one instance for each call of the function/block**

Scope and Side-Effects

```
#include <stdio.h>
int x = 0;

int incrX( ) { x++; return x; }
```

What is the type of `incrX`?

- **Prototype:**

  int *incrX*(void);
- **Mathematical:**

  $\texttt{incrX} : \mathbb{1} \to \texttt{int}$

This is not the whole interface to `incrX`!

Scope and Side-Effects — Simulation

```
#include <stdio.h>                    /* incrX.c */
int x = 0;
int incrX( ) { x++; return x; }
int main() {
  int x = 10, y;
  y = incrX();
  printf("%d  %d  %d\n", x, y, incrX());
  return 0;}
```

```
global  main()                          Output
  x      x    y
  0                 init.
        10          init.
  1                 x++;
                    y = incrX();
  2                 x++;
                                      10  1  2
```


Scope and Side-Effects

```
#include <stdio.h>                    /* incrX.c */
int x = 0;
int incrX( ) { x++; return x; }
int main() {
  int x = 10, y;
  y = incrX();
  printf("%d  %d  %d\n", x, y, incrX());
  return 0;}
```

Local variables **shadow** variables from outer scopes.

Side-effects:

– *incrX* changes the value of a variable not mentioned in its formal interface.

– The return value of *incrX* depends on a variable not mentioned in its formal interface.

Pure Functions

Pure functions have no side-effects:

– Return values depend only on the actual parameters

– No global variables are updated

– No I/O is performed

Math library functions are “almost pure”:

- In case of error, the global variable `errno` is set.
- Floating-point precision may depend on, e.g., compiler switches.

Repeated Function Calls

```
#include <stdio.h>                    /* squares.c */

int f(int k)
{ return 2 * k + 1; }

int main() {
  int s = 0, i;
  for(i = 0; i < 4; i++)
   { s += f(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```

Repeated Function Calls 2

```
#include <stdio.h>                /* squares2.c */

int f(int k) {
  k *= 2;
  return ++k;
}

int main() {
  int s = 0, i;
  for(i = 0; i < 4; i++)
   { s += f(i); printf("%d  %d\n", i, s); }
  return 0;
}
```


Alternating Function Calls

```
#include <stdio.h>                /* series1.c */

int f(int k) { k += 2; return k + 1; }

int g(int m) { return 2 * m * m - 1; }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += f(i);
     s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Nested Function Calls 1

```
#include <stdio.h>                /* series2.c */

int f(int k) { k += 2; return k + 1; }

int g(int m) { return (m + 1) * f(m); }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```


Nested Function Calls 2

```
#include <stdio.h>                /* series3.c */

int f(int k) { k += 2; return k + 1; }

int g(int m) { return (m - 1) * f(m); }

int main() {
  int s = 0, i;
  for(i = 0; i < 3; i++)
   { s += f(i);
     s += g(i);
     printf("%d  %d\n", i, s);
   }
  return 0;
}
```

static **Local Variables**

```
#include <stdio.h>        /* squares4.c */

int step(int n) {
  static int d = 1;
  static int q = 1;
  int r = n * q;
  d += 2;
  q += d;
  return r;
}

int main() {
  int i;
  for(i = 1; i <4 ; i++)  printf("%d  %d\n", i, step(i));
  return 0;
}
```

```
int d = 1;
int q = 1;

int step(int n) {
  int r = n * q;
  d += 2;
  q += d;
  return r;
}
```


static

static local variables are global variables **visible only within the enclosing block**.

— **"private state"**

static global variables are **visible only within the enclosing file**.

Functions, too, can be made **visible only within the enclosing file** by declaring them static.

Non-static local variables are also called **automatic**.

Arrays

int *myArray*[10];

declares an **array**:

- the **name** of the array is "*myArray*"
- the **type** of the array is "int[10]"
- the array has 10 elements
- each element is a **variable of type** int
- the elements are arranged **consecutively** in memory
- the elements have **indices** 0, …, 9
- the element with index *i* is "*myArray*[*i*]"

Array Initialisation

```
int myArray[7] = { 23, 34, 45, 56, 67, 78, 89 };
/* result:          [ 23, 34, 45, 56, 67, 78, 89 ] */

int myArray[7] = { 23, 34, 45, 56, 67 };
/* result:          [ 23, 34, 45, 56, 67,  0,  0 ] */

int myArray[]  = { 23, 34, 45, 56, 67 };
/* result:          [ 23, 34, 45, 56, 67 ] */

int myArray[7] = {  1};
/* result:          [  1,  0,  0,  0,  0,  0,  0 ] */

int myArray[7] = {  0 };
/* result:          [  0,  0,  0,  0,  0,  0,  0 ] */

int myArray[7] = { };          /* Not ANSI C! */
```

- Automatic arrays are **not automatically initialised**!
- static arrays are, by default, initialised with zero values.

Character Arrays

Strings are treated as **character arrays**:

```
char myString[] = "hello";
/* result:          [ 'h', 'e', 'l', 'l', 'o', '\0' ] */
```

- Strings are terminated by the null character '\0'
- '\0' is the last array element of a string constant
- String generation/copying functions need **space for the terminator**

  strdup(), *fscanf*("%s", ...)

  Hard to ensure for user input! — *fscanf*("%79s", ...)
- String processing functions never read past the terminator

  strcmp(), *strncmp*(), *strstr*()

Variable Array Dinemnsions (C99)

```
#include <stdio.h>                          /* VarArray.c */

void printRow(int r[], int size)
{ int j; for (j=0; j<size; j++) printf("%7d ", r[j]); printf("\n"); }

void make_and_print_array(int k) {
  int j, ar[ k ];              /* array dimension k is a variable! */
  for (j=0; j<k; j++) ar[j] = 101 * (j + 1);
  printRow(ar, k);
}

int main(void) { make_and_print_array(4); return 0; }
```

- In ANSI C, dimensions must be **constant** expressions
- In C99, **variables** seem to be allowed.

Passing Arguments by Value / by Reference

In, e.g., Oberon (but not in C), if a variable name *v* is passed as argument to a function *f*, this can be done in two ways:

- **pass by value**: the value of *v* is passed to *f*, where it is bound to the formal parameter name.

  Usually (including in C), the formal parameter is treated as a local variable.
- **pass by reference**: *f* obtains a **reference** to *v* and turns its formal parameter name into an **alias** for *v*

C:

- in general: **pass by value**;
- **array** arguments should be understood as **passed by reference** (they are const pointers passed by value).

Pass by Reference Example 1

```
#include <stdio.h>                          /* ArrayArgs1.c */
#define SIZE 5

void show(int ar[SIZE])
{ int i; for (i = 0; i < SIZE; i++) printf("%7d %7d\n", i, ar[i]); }

void update(int ar[SIZE], int i, int v) { ar[i] = v; }

int main(void) {
  int i;
  int counts[SIZE];
  for (i = 0; i < SIZE; i++) counts[i] = 10 * i;
  update (counts, 3, 333);
  show (counts);
  return 0;
}
```

Pass by Reference Example 2

```
#include <stdio.h>                          /* ArrayArgs2.c */
#define SIZE 5

void show(int ar[SIZE])
{ int i; for (i = 0; i < SIZE; i++) printf("%7d %7d\n", i, ar[i]); }

void swap(int ar[], int i, int j)
{ int tmp = ar[i]; ar[i] = ar[j]; ar[j] = tmp; }

int main(void) {
  int i, counts[SIZE];
  for (i = 0; i < SIZE; i++) counts[i] = 11 * i;
  swap (counts, 0, 3);
  show (counts);
  return 0;
}
```


Arrays in structs Are Passed By Value!

```
#include <stdio.h>                          /* StructArrayTest.c */
#define SIZE 10
struct test { int ar[SIZE]; } t;

void show(struct test t)
{ int i; for (i = 0; i < SIZE; i++) printf("%7d %7d\n", i, t.ar[i]); }

void update(struct test t, int i, int v)
{ t.ar[i] = v; show(t); }

int main(void) {
  int i; for (i = 0; i < SIZE; i++) t.ar[i] = 10 * i;
  update(t, 2, 222);
  show(t);
  return 0;
}
```


Two-Dimensional Arrays

int *myTable*[4][10];

declares a **two-dimensional array**:

- two-dimensional array can be understood as arrays of one-dimensional arrays
- the **name** of the array is "*myTable*"
- the **type** of the array is "int[4][10]"
- the array has 4 elements
- each element is an **array of type** int[10]
- the elements are arranged **consecutively** in memory
- the elements have **indices** 0, 1, 2, 3

Initialisation of Two-Dimensional Arrays

```
int myTable[2][3] = { { 1, 2, 3}, { 4, 5, 6} };
/* result:          [ [ 1, 2, 3], [ 4, 5, 6] ] */

int myTable[2][3] = { { 1     }, { 4, 5, 6} };
/* result:          [ [ 1, 0, 0], [ 4, 5, 6] ] */

int myTable[2][3] = { { 1, 2, 3} };
/* result:          [ [ 1, 2, 3], [ 0, 0, 0] ] */
```

Accessing Rows of Two-Dimensional Arrays

```
#include <stdio.h>                          /* TwoDimTest.c */
void printRow(int r[10])
{ int j; for (j=0; j<10; j++) printf("%7d %7d\n", j, r[j]); }

int main(void) {
  int myTable[4][10];
  int i,j;
  for(i=0; i<4; i++)
    for(j=0; j<10; j++)
      myTable[i][j] = 100 * i + j;
  printRow(myTable[2]);
  return 0;
}
```


Testing Initialisation of Two-Dimensional Arrays

```
#include <stdio.h>                          /* TwoDimInitTest.c */
void printRow(int r[3])
{ int j; for (j=0; j<3; j++) printf("%7d ", r[j]); printf("\n"); }

void printTable(int t[2][3]) {
  int j;
  for (j=0; j<2; j++) printRow(t[j]);
  printf("\n");
}

int main(void) {
  int myTable[2][3] = { { 1, 2, 3} };
  printTable(myTable);
  return 0;
}
```


Linear Search — Version 1

```
/* Compare key to every element of array until the location
   is found or until the end of array is reached; return
   subscript of element if found, or -1 if key is not found.
 */
int linearSearch( const int array[], int key, int size )
{
   int n;                    /* counter */

   for ( n = 0; n < size; ++n ) { /* loop through array */
      if ( array[ n ] == key )
         return n;           /* return location of key */
   }
   return –1;                 /* key not found */
}
```


Linear Search — Version 2

```
#define NOT_FOUND –1   /* safe, since not a legal index. */

int linear_search(int data[], int target, int n)
{
  /* Use parameter n as local counter variable.
   * Initial value is one larger than largest index.
   */
  while ( ––n ≥ 0 )
    if ( data[n] == target )
      break;             /* found target */

  return n;  /* If while-condition failed, n is NOT_FOUND;
                if loop terminated by break, n is the target index.
              */
}
```

Linear Search — Version 3

```
#define NOT_FOUND –1    /* safe, since not a legal index. */

int linear_search(int data[], int target, int n)
{
  /* Use parameter n as local counter variable.
   * Initial value is one larger than largest index.
   */
  while ( ––n ≥ 0 && data[n] ≠ target ) {}

  return n;  /* If first condition failed, n is NOT_FOUND;
                if second condition failed, n is the target index.
             */
}
```


Pointers

Pointers are a low-level mechanism that allows to simulate important high-level constructs:

- call by reference
- (certain) subarrays
- recursive datastructures
- graph-like datastructures
- higher-order functions

Pointers

- A **pointer** is a memory address
- **Pointer** variables are variables whose values are memory addresses
- Referencing a value though a pointer is called *indirection*
- Pointer variable declaration in C:

  ```
  int *countPtr, count;
  ```

  – *countPtr* is a variable of type "int *", i.e., a "**pointer to an** int"
  – *count* is a variable of type int.

The Address Operator &

```
int y = 5;
int * yPtr;

yPtr = &y;
```

- The operand of & must be a **variable**
- & can not be applied to
  – constants
  – expressions
  – register variables

The Pointer Dereferencing Operator *

```
int y = 5;
int * yPtr;

yPtr = &y;

printf( "%d\n", *yPtr );
*yPtr = 7;
printf( "%d\n", y);
```

The dereferencing operator, applied to a pointer value *p*, returns **the variable *p* points to**, i.e.,

- in an expression: the **contents** of the memory address *p*
- to the left of an assignment: the **variable** at address *p*

Make sure p points to a sensible address!

Simulate Execution! — 1

```
#include <stdio.h>                    /* pointers1.c */

void swap(int * p, int * q)
{
   int h = *p;
   *p = *q;
   *q = h;
}

int main () {
  int i=4, j=7;     swap( &i, &j );
  printf("i = %d; j = %d\n", i, j); return 0;
}
```


Simulate Execution! — 2

```
#include <stdio.h>                    /* pointers2.c */

void swap(int * p, int * q)
{
   int h = *p;
   *p = *q;
   *q = h;
}

int main () {
  int i=4, j=7;     swap( &i, &i );
  printf("i = %d; j = %d\n", i, j); return 0;
}
```


Simulate Execution! — 3

```
#include <stdio.h>                    /* pointers3xor.c */

void swap(int * p, int * q)
{                   /* *p = a                  ∧ *q =  b */
   *p ^= *q;        /* *p = a^b                ∧ *q =  b */
   *q ^= *p;        /* *p = a^b                ∧ *q = b^a^b = a */
   *p ^= *q;        /* *p = a^b^a = b          ∧ *q = a */
}

int main () {
  int i=4, j=7;              swap( &i, &j );
  printf("i = %d; j = %d\n", i, j); return 0;
}
```

Simulate Execution! — 4

```
#include <stdio.h>                          /* pointers4xor.c */

void swap(int * p, int * q)
{                   /* *p = a              ∧ *q =  b */
  *p ^= *q;         /* *p = a^b            ∧ *q =  b */
  *q ^= *p;         /* *p = a^b            ∧ *q = b^a^b = a */
  *p ^= *q;         /* *p = a^b^a = b      ∧ *q = a */
}

int main () {
  int i=4, j=7;              swap( &i, &i );
  printf("i = %d; j = %d\n", i, j); return 0;
}
```


const and Pointers

`int *`	`p;`	*p* contains a *non-constant* pointer to *non-constant* data
`const int *`	`p;`	*p* contains a *non-constant* pointer to *constant* data
`int * const`	`p;`	*p* contains a *constant* pointer to *non-constant* data
`const int * const`	`p;`	*p* contains a *constant* pointer to *constant* data

- **constant**: read-only
- **non-constant**: **variable**, read-write

The sizeof Operator

sizeof is a keyword used like a function that accepts as single argument

- **any variable**, or
- **any type**,

and returns an integral value of type *size_t* indicating

- how many bytes are reserved for the given **variable,** or
- how many bytes are reserved for variables of the given **type.**

Note: For array variables, this yields sizeof(*element_type*) * *array_size*

General sizeof Rules

- A byte has 8 bits
- In C, characters are 8-bit integral values
- On an *n*-bit architecture, **int**, **float**, and pointers occupy *n* bits, i.e., *n*/8 bytes
- **double** variables occupy twice as much space as **float**s
- **long** occupies not less space than **int**
- **short** occupies not more space than **int**

Pointer Expressions and Pointer Arithmetic

A T-pointer *ptr* can be understood as an

"abstract index into memory considered as an array of T-variables"

- $ptr + 1$ is "the next index" — it points to the next T-variable
- when considering pointers as integers (for example when printing with "%p") the difference between $ptr + 1$ and *ptr* is sizeof(T)
- for pointers of the same type (e.g. after $ptrB = ptr + n$) one may calculate the ***pointer difference*** $ptrB - ptr$, which will be *n*.

Other **pointer arithmetic** operators:

- "+=", "–=", "++", "––"

Relationship between Pointers and Arrays

Array variables can be understood as **const** pointers, *for which the size of space pointed to is known.*

Assume:

```
double b[5];
double * bPtr;
bPtr = b;
```

Then:

- $bPtr == \&b[0]$
- $bPtr + 1 == \&b[1]$
- $*(bPtr + 3) == b[3]$
- $bPtr + 3 == b + 3$
- $bPtr[1] == b[1]$

Arrays of Strings

```
#include <stdio.h>                    /* suits.c */
int main() {
  const char * suit[4] = {"Hearts", "Diamonds",
                          "Clubs", "Spades"};
  int i;
  for (i = 0; i < 4; i++)
    printf("suit[%d] = %p\t: "%s"\n", i, suit[i], suit[i]);
  printf("suit[2][3] = '%c'\n", suit[2][3]);
  suit[3] = "Shovels";
  for (i = 0; i < 4; i++)
    printf("suit[%d] = %p\t: "%s"\n", i, suit[i], suit[i]);
  return 0;
}
```


Arrays of Strings — Comments

- char * *suit*[4] — an array of char * values
- char * values interpreted as beginning of zero-terminated character strings
- **Can** be considered as an array of arrays — *suit*[2][3] works.
- Subarrays can be **anywhere** — nothing is known about relative arrangement in memory of *suit*[2] and *suit*[3]