Programming Fundamentals

- Lecturer: Dr. Arnold Meijster
- e-mail: a.meijster@rug.nl
- room: 5161.343 (Bernoulliborg)

GCD: Euclid’s algorithm

```
/* a==A, b==B */
while (b != 0) {
   int r = a%b;
   a = b;
   b = r;
}
/* a==gcd(A,B) */
```

In the previous lecture, you’ve seen that the above code can be used to compute gcd(a,b).

But what to do if we want to compute the greatest common divisor of three numbers?
For example gcd3(150, 12, 9)=gcd(gcd(150,12),9)=gcd(6,9)=3.

GCD3: Euclid’s algorithm

```
int a, b, c;
scanf ("%d %d %d", &a, &b, &c);
/* a==A, b==B, c==C */
while (b != 0) {
   int r = a%b;
   a = b;
   b = r;
}
/* a==gcd(A,B), c==C */
while (c != 0) {
   int r = a%c;
   a = c;
   c = r;
}
/* a==gcd(gcd(A,B),C) */
printf("gcd(a,b,c)=%d\n", a);
```

Procedural Programming

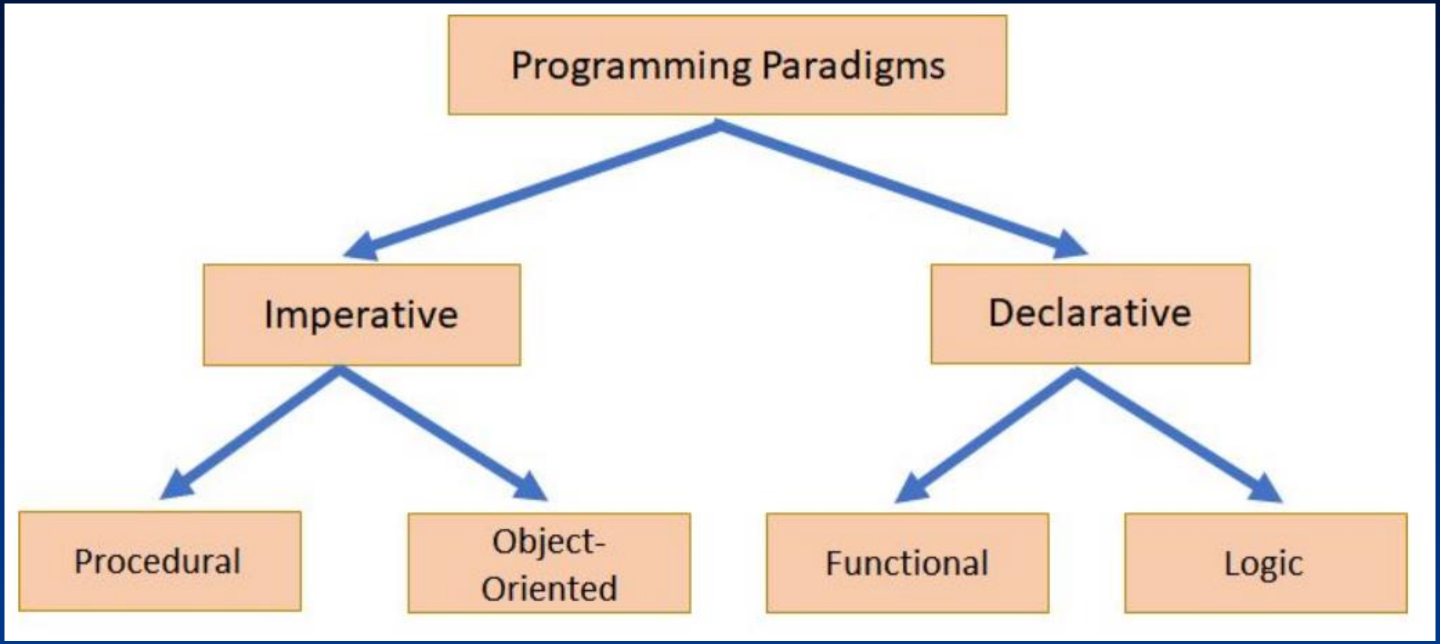
Clearly, this kind of code duplication should be avoided. We want code like:

```
int a, b, c;
scanf ("%d %d %d", &a, &b, &c);
a = gcd(a,b);
a = gcd(a,c);
printf("gcd(a,b,c)=%d\n", a);
```

Or maybe even nicer:

```
int a, b, c;
scanf ("%d %d %d", &a, &b, &c);
printf("gcd(a,b,c)=%d\n", gcd(gcd(a,b),c));
```

Procedural Programming



From Wikipedia: "**Procedural programming** is a programming paradigm, derived from imperative programming, based on the concept of the *procedure call*. Procedures (a type of routine) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself."

Functions

A *function* is a program fragment that

- can be used from any other place in the program (‘function call’).
- performs a specific task.
- is logically coherent.
- can return a result (value).

Function prototype

Functions, like variables, must be declared.

A function declaration is called a *function prototype*:

```
    void printValue(int n);
```

The function declaration defines only the function type. It does not say anything about the implementation of the function.

Math.h: prototypes of (some) functions

Mathematical functions are available in the include file `math.h`

```
#include <math.h>

#define M_PI 3.14159265358979323846

double pow (double g, double m);
double sqrt (double x);

double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double exp(double x);
double log(double x);
double log2(double x);
double log10(double x);

double round(double x);
double floor(double x);

int abs(int x);
double fabs(double x);
```

Examples:

```
double s2 = sqrt(2);

int r = round(5.6);

int eight = log2(256);

int two = log10(100);
```

Function definition

The definition of a function consists of its prototype + the *body* that implements the function.

General form:

```
<returntype> functionName(<type1> parameter1, …,
                          <typeN> parameterN) {
   /* body of the function */
}
```

`<returntype>` is the type of the function result.

The return type is `void` if the function does not return a result.

We use the following conventions for choosing function names:

- camelCase
- No result (void): imperative form, e.g. `void printValue(int n)`
- With result: description of result, e.g. `int isPrimeNumber(int n)`.

The parameters of a function can be used in the body of the function as if they were declared and initialised as *local variables*.

Example: void function (without result)

The following function prints the sum of its parameters:

```
void printSum(int a, int b, int c) {
  printf("%d", a + b + c);
}
```

Example: function with result

The following function *returns* the sum of its parameters:

```
int sum3(int a, int b, int c) {
  return a + b + c;
}
```

Example: function with result

The following function *returns* the maximum of its two parameters:

```
int maximum(int a, int b) {
  return (a > b ? a : b);
}
```

gcd: function with result

```
int gcd(int a, int b) {
  /* a==A, b==B */
  while (b != 0) {
     int r = a%b;
     a = b;
     b = r;
  }
  /* a==gcd(A,B) */
   return a;
}
```

Function call

Once a function is defined, it can be used anywhere in the program (‘function call’).

A function call consists of the name of the function and a list of *arguments* (between parentheses).

Each argument must be an *expression* of the right type. The *values* of the expressions are used to ‘initialise’ the parameters of the function.

The call of a void-function is a ‘normal’ statement, e.g. **`printValue(n)`**.

The call of a function that returns a result is an expression having the same type as the return-type of the function, e.g. **`x = y + 3*someFunction(n)`**.

Example

```
int maximum(int a, int b) {
  return (a > b ? a : b);
}

void printMaximum(int p, int q, int r, int s) {
  printf("%d\n", maximum(maximum(p, q), maximum(r, s)));
}

int main(int argc, char *argv[]) {
  printMaximum(1, 2, 3, 4);
  return 0;
}
```

Example (another version)

```
int maximum(int a, int b) {
  return (a > b ? a : b);
}

void printMaximum(int p, int q, int r, int s) {
  int x = maximum(p, q);
  int y = maximum(r, s);
  printf("%d\n", maximum(x,y));
}

int main(int argc, char *argv[]) {
  printMaximum(1, 2, 3, 4);
  return 0;
}
```

Scope

- The *scope* of a variable is the region in the code in which the variable ‘exists’.

- The scope of a variable is the region that starts at the declaration of the variable and ends at the closing bracket } of the corresponding code block.
    - Exception 1: parameters of a function are declared outside a code block: their scope is the entire function.
    - Exception 2: ‘*global variables*’ (see next item)

- *Global variables* are variables that are declared outside a function. They are (usually) declared at the top of a program. Their scope is the entire program.
    - Advice: use global variables only if really needed.

- Variables that are declared in a function or within a code block (i.e. between a pair of brackets) are called ‘*local variables*’. Their scope is the code block that they are declared in.

- This may lead to a phenomenon called ‘*shadowing*’. A local variable makes a variable (from a surrounding scope) with the same name temporarily invisible.

```c
#include <stdio.h>

int x=1; /* global x */

void print() {
   printf("%d\n", x);
}

void printX(int x) {/* local x (shadows global x) */
   printf("%d\n", x);
}

void someFunction() {
   int x = 3; /* local x (shadows global x) */
   print();
   printX(x);
   {
     int x = 4; /* local x (shadows surrounding local x) */
     print();
     printX(x);
   }
   printX(x);
}

int main(int argc, char *argv[]) {
  int x = 2; /* local x (shadows global x) */
  print();
  printX(x);
  printf("%d\n", x);
  someFunction();
  printf("%d\n", x);
  return 0;
}
```

```
Output:

1

2

2

1

3

1

4

3

2
```

```
#include <stdio.h>

int x=1; /* global x */

void print() {
   printf("%d\n", x);
}

void printX(int x) {/* local x (shadows global x) */
   printf("%d\n", x);
}

void someFunction() {
   x = 3;   // was: int x = 3;
   print();
   printX(x);
   {
     int x = 4; /* local x (shadows surrounding local x) */
     print();
     printX(x);
   }
   printX(x);
}

int main(int argc, char *argv[]) {
  int x = 2; /* local x (shadows global x) */
  print();
  printX(x);
  printf("%d\n", x);
  someFunction();
  printf("%d\n", x);
  return 0;
}
```

Try to figure out the output yourself!

Prime factorization

Exercise: write a program that repeatedly asks the user to type in an integer >1 (or 1 to stop) and then prints the prime factorization of the input number.

```
Please, type a natural number >1 (1=stop): 123456
2^6 * 3^1 * 643^1
Please, type a natural number >1 (1=stop): 997
997^1
Please, type a natural number >1 (1=stop): 64
2^6
Please, type a natural number >1 (1=stop): 192
2^6 * 3^1
Please, type a natural number >1 (1=stop): 1
Goodbye.
```

Prime factorization

```
int main(int argc, char *argv[]) {
  int number = readNumber();
  while (number > 1) {
    printFactorization(number);
    number = readNumber();
  }
  printf("Goodbye.\n");
  return 0;
}
```

readNumber

```
int readNumber() {
  int number;
  do {
    printf("Type a natural number >=1 (1=stop): ");
    scanf("%d", &number);
  } while (number < 1);
  return number;
}
```

printFactorization

```
void printFactorization(int number) {
  int divisor = 2;
  int first = 1;  /* 1 == TRUE */
  while (number != 1) {
    int exponent = countFactors(number, divisor);
    if (exponent > 0) {
      if (first) {
        first = 0;
      } else {
        printf(" * ");
      }
      printf ("%d^%d", divisor, exponent);
      number /= exponentiation(divisor, exponent);
    }
    divisor++;
  }
  printf("\n");
}
```

printFactorization (efficient)

```
void printFactorization(int number) {
  int divisor = 2;
  int first = 1;  /* 1 == TRUE */
  while (divisor*divisor <= number) {
    int exponent = countFactors(number, divisor);
    if (exponent > 0) {
      if (first) {
        first = 0;
      } else {
        printf(" * ");
    }
      printf ("%d^%d", divisor, exponent);
      number /= exponentiation(divisor, exponent);
    }
    divisor++;
  }
} if (number > 1) {
    printf((first ? "%d^1" : " * %d^1"), number);
  }
  printf("\n");
}
```

printFactorization: Efficieny analysis

Let us compute `printFactorization(int 999983);`

The number 999983 is the largest prime below a million.

The initial version of the program would require 999983 iterations.

The second (more efficient) version would require $\sqrt{999983} \approx 1000$ iterations. A major improvement! This algorithm is (for this input) about 1000 times faster.

countFactors

The function call `countFactors(x,y)` returns the number of times that we can divide `x` by `y`.

```
int countFactors(int n, int base) {
  int cnt=0;
  while (isDivisor(n, base)) {
    n /= base;
    cnt++;
  }
  return cnt;
}
```

isDivisor

The function call **`isDivisor(x,y)`** returns true (i.e. 1) if **`x`** is an integer multiple of **`y`**.

```
int isDivisor (int x, int y) {
  return (x%y == 0);
}
```

Exponentiation (see week 2)

```
int exponentiation(int number, int exponent) {
  /* returns number^exponent */
  int m=1;
  while (exponent!=0) {
    if (exponent%2 == 0) {
      number = number*number;
      exponent = exponent/2;
    } else {
      m = m*number;
      exponent--;
    }
  }
  return m;
}
```

Call-by-value

What is the output of this program?

```
void nextInt(int n) {
  n++;
  printf("%d\n", n);
}

int main(int argc, char *argv[]) {
  int n = 1;
  printf("%d\n", n);
  nextInt(n);
  printf("%d\n", n);
  return 0;
}
```

```
Output:
1
2
1
```

Call-by-value

We make a distinction between *formal* and *actual* parameters:

- Formal parameters: *parameter names* from the prototype of a function
- Actual parameters: *values* that are passed to a function

Call-by-value: When we call a function, the *value of expressions* that are passed to the function are *copied* into the formal parameters before execution of the function.

A value is not a variable. Hence, it cannot change!

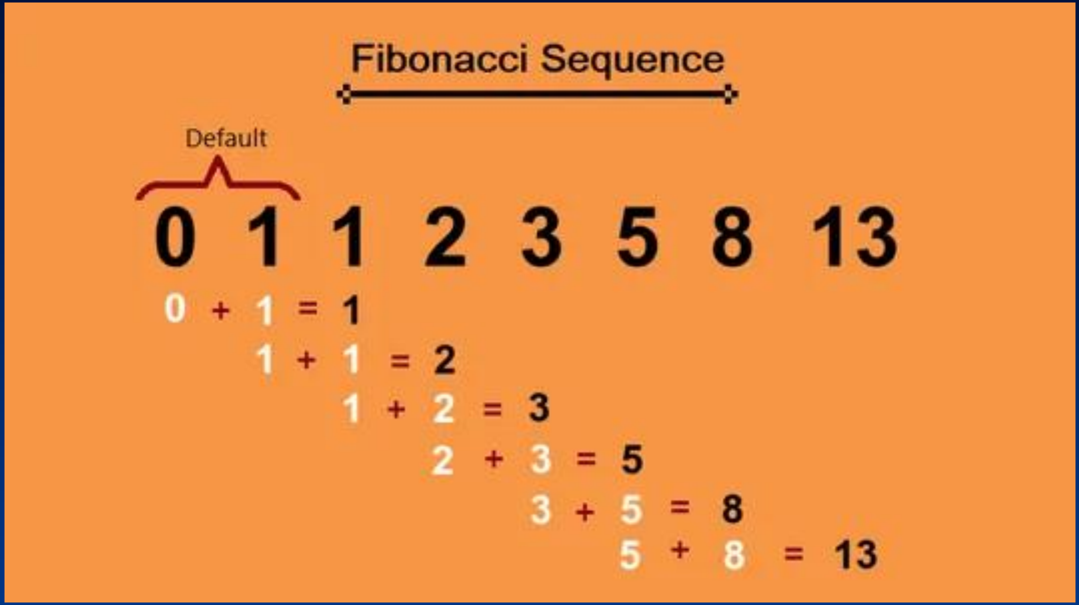
Conclusion: Due to call-by-value it is not possible to change the value of actual parameters!

Call-by-value

```
void nextInt(int n) {
  n++;
  printf("%d\n", n);
}

int main(int argc, char *argv[]) {
  nextInt(1); // We pass a value! 1++ is meaningless
  return 0;
}
```

```
int main(int argc, char *argv[]) {
  int n = 1;
  nextInt(n); // same here
  return 0;
}
```

Fibonacci again: 0, 1, 1, 2, 3, 5, …



```
int fib(int n) {
  int a = 0;
  int b = 1;
  for (int i=0; i < n; i++) {
    b = a + b;
    a = b - a;
  }
  return a;
}
```

Fibonacci: one, one, two, three, five, …

In which year do we reach a number of pairs that is at least (a given) **n**?

```
int young = 1;
int mature = 0;
int year = 0;
int n;
Scanf("%d", &n);
while (young + mature < n) {
   mature = mature + young;
   young = mature - young;
   year++;
}
/* here: young + mature >= n  (logical negation of guard) */
printf("In year %d we have at least %d pairs.\n", year, n);
```

The code below is easier to understand, but less efficient due to repeated recomputation!

```
int year = 0;
while (fib(year) < n) {
   year++;
}
printf("In year %d we have at least %d pairs.\n", year, n);
```

Function calling another function: Palindromic Number

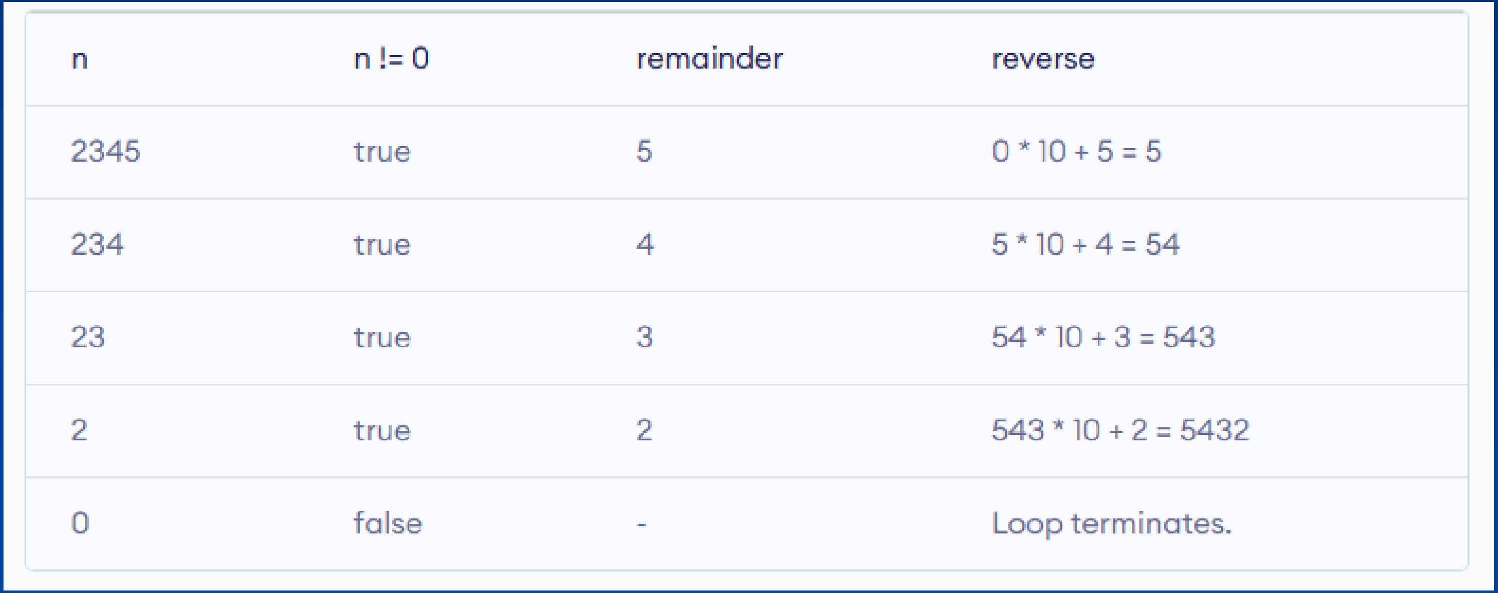
Write a function `isPalindromicNumber(int n)` that returns 1 if n is a palindromic number, otherwise it returns 0.

- `isPalindromicNumber(121)` returns 1
- `isPalindromicNumber(123321)` returns 1
- `isPalindromicNumber(1231)` returns 0

```
int isPalindromicNumber(int n) {
  return (reverseInt(n) == n); // returns 0,1(false,true)
}
```

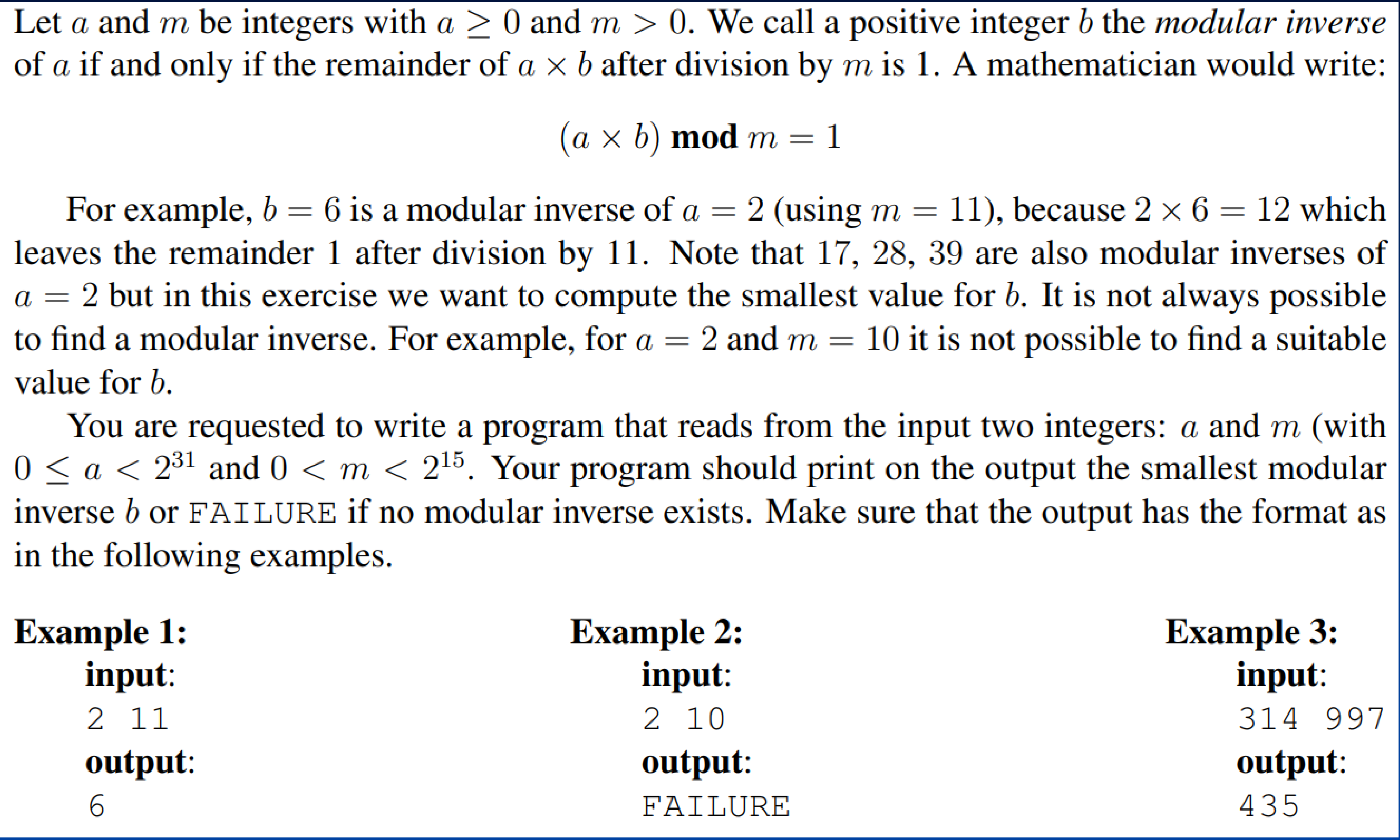
reverseInt function

```
int reverseInt(int n) {
  int reverse = 0;
  while (n != 0) {
    int remainder = n%10;
    reverse = reverse*10 + remainder;
    n /= 10;
  }
  return reverse;
}
```



n	n != 0	remainder	reverse
2345	true	5	0 * 10 + 5 = 5
234	true	4	5 * 10 + 4 = 54
23	true	3	54 * 10 + 3 = 543
2	true	2	543 * 10 + 2 = 5432
0	false	-	Loop terminates.

Old lab exercise: modular inverse



Let a and m be integers with $a \geq 0$ and $m > 0$. We call a positive integer b the *modular inverse* of a if and only if the remainder of $a \times b$ after division by m is 1. A mathematician would write:

$$(a \times b) \textbf{ mod } m = 1$$

For example, $b = 6$ is a modular inverse of $a = 2$ (using $m = 11$), because $2 \times 6 = 12$ which leaves the remainder 1 after division by 11. Note that 17, 28, 39 are also modular inverses of $a = 2$ but in this exercise we want to compute the smallest value for b. It is not always possible to find a modular inverse. For example, for $a = 2$ and $m = 10$ it is not possible to find a suitable value for b.

You are requested to write a program that reads from the input two integers: a and m (with $0 \leq a < 2^{31}$ and $0 < m < 2^{15}$. Your program should print on the output the smallest modular inverse b or `FAILURE` if no modular inverse exists. Make sure that the output has the format as in the following examples.

Example 1:

input:
```
2 11
```
output:
```
6
```

Example 2:

input:
```
2 10
```
output:
```
FAILURE
```

Example 3:

input:
```
314 997
```
output:
```
435
```

Old lab exercise: modular inverse

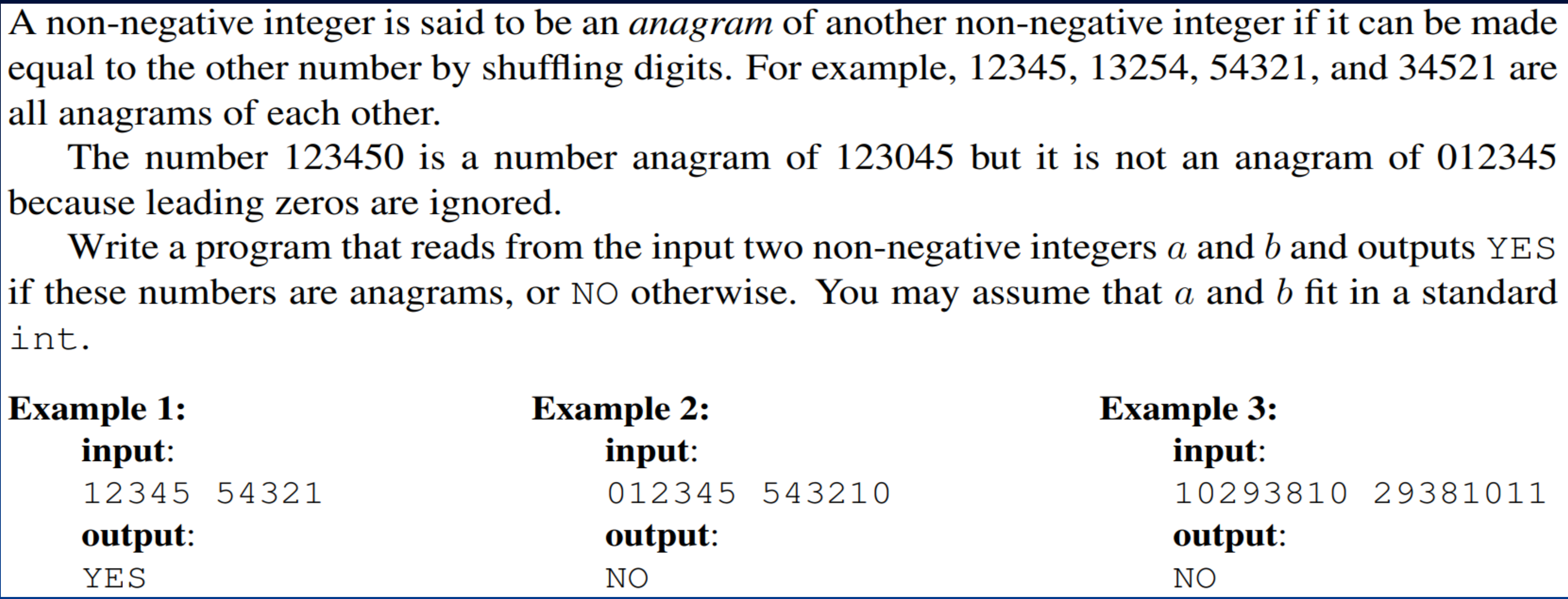
```c
int modInverse(int a, int modulus) {
  a %= modulus;
  int b = 1;
  while ((b < modulus) && (a*b % modulus != 1)) {
    b++;
  }
  return (b >= modulus ? 0 : b);   // 0 means no inverse found
}

int main(int argc, char *argv[]) {
  int a, m;
  scanf("%d %d", &a, &m);
  int b = modInverse(a, m);
  if (b == 0) {
    printf("FAILURE\n");
  } else {
    printf("%d\n", b);
  }
  return 0;
}
```

Old lab exercise: modular inverse

The following version of modInverse is slightly more efficient: it never multiplies or divides (modulus).

```
int modInverse(int a, int modulus) {
  a %= modulus;
  int ab = a, b = 1;     // ab == (a*b)%modulus
  while ((b < modulus) && (ab != 1)) {
    // invariant: ab == (a*b)%modulus
    b++;
    ab += a;
    if (ab >= modulus) {
      ab -= modulus;
    }
  }
  return (ab == 1 ? b : 0);    // 0 means no inverse found
}
```

Old lab exercise: number anagram



A non-negative integer is said to be an *anagram* of another non-negative integer if it can be made equal to the other number by shuffling digits. For example, 12345, 13254, 54321, and 34521 are all anagrams of each other.

The number 123450 is a number anagram of 123045 but it is not an anagram of 012345 because leading zeros are ignored.

Write a program that reads from the input two non-negative integers a and b and outputs `YES` if these numbers are anagrams, or `NO` otherwise. You may assume that a and b fit in a standard `int`.

Example 1:
input:
```
12345 54321
```
output:
```
YES
```

Example 2:
input:
```
012345 543210
```
output:
```
NO
```

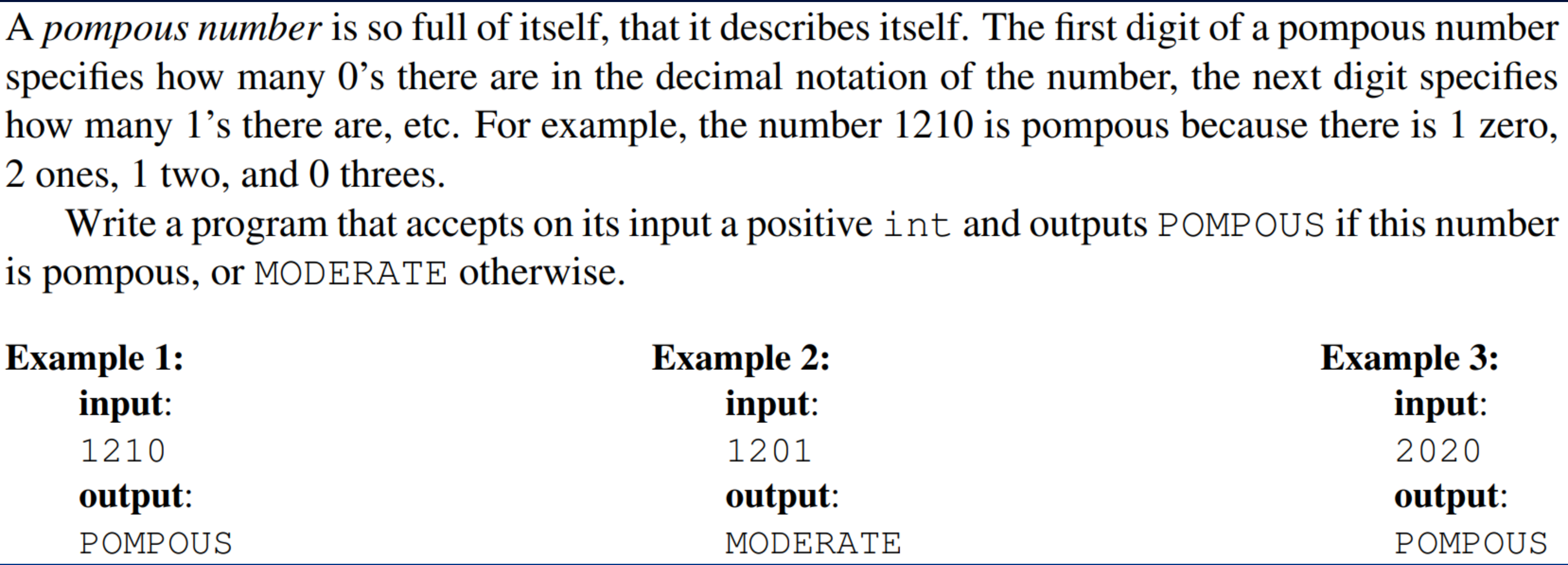
Example 3:
input:
```
10293810 29381011
```
output:
```
NO
```

```
int main(int argc, char *argv[]) {
  int a, b, digit = 0;
  scanf("%d %d", &a, &b);
  while ((digit < 10) && (digitCount(digit, a) == digitCount(digit, b))) {
    digit++;
  }
  printf(digit == 10 ? "YES\n" : "NO\n");
  return 0;
}
```

digitCount function

```
int digitCount(int digit, int n) {
   int cnt=0;
   while (n != 0) {
      cnt += (n%10 == digit);
      n /= 10;
   }
   return cnt;
}
```

Old lab exercise: pompous number



A *pompous number* is so full of itself, that it describes itself. The first digit of a pompous number specifies how many 0's there are in the decimal notation of the number, the next digit specifies how many 1's there are, etc. For example, the number 1210 is pompous because there is 1 zero, 2 ones, 1 two, and 0 threes.

Write a program that accepts on its input a positive `int` and outputs `POMPOUS` if this number is pompous, or `MODERATE` otherwise.

Example 1:

input:
```
1210
```
output:
```
POMPOUS
```

Example 2:

input:
```
1201
```
output:
```
MODERATE
```

Example 3:

input:
```
2020
```
output:
```
POMPOUS
```

```
int main(int argc, char *argv[]) {
  int n;
  scanf("%d", &n);
  printf(isPompousNumber(n) ? "POMPOUS\n" : "MODERATE\n");
  return 0;
}
```

Old lab exercise: pompous number

```
int isPompousNumber(int n) {
  int mask = maxPower10(n);
  int m=n, digit = 0;
  while (mask != 0) {
    if (digitCount(digit, n) != m/mask) {
      break;
    }
    m %= mask;
    mask /= 10;
    digit++;
  }
  return (mask == 0);
}
```

Old lab exercise: pompous number

```
int maxPower10(int n) {
   int mp = 1000000000;
   // maximum int has 10 digits
   while (n/mp == 0) {
      mp /= 10;
   }
   return mp;
}
```

Arrays

An *array* is used to store a *sequence* of values of the same type.

An array is a numbered list.

- A member of the array is called an *element*.
- Each element is a ‘variable’ of the corresponding type.

The number/location of an element is called its *index*.

Arrays: Declaration/initialisation

There are three ways to declare an array:

- `type arrayName[N];`
- `type arrayName[N] = {value1, value2, …, valueN};`
- `type arrayName[] = {value1, value2, …, valueN};`

In the first case, the array is declared, but not initialised.

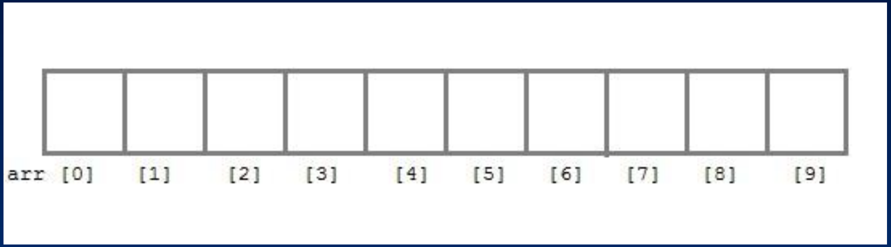
In the second case, the array is declared and initialised.

In the third case, the arrays is declared and initialised as well. Since we give an explicit list of values, the compiler is able to infer the correct size of the array.

Examples

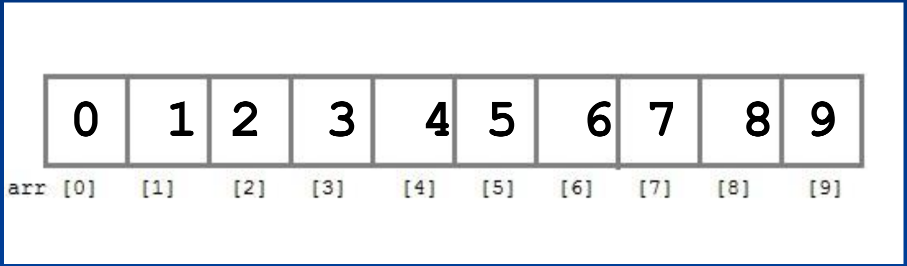
First form:

```
int arr[10];
```



Second form:

```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```



Third form:

```
int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Vectors in 100 D space

What about vectors in 100D space?

```
typedef double vector100[100];
vector100 y;
```

Note the (strange) syntax of the typedef.

The following seems more natural, but is WRONG!

```
typedef double[100] vector100;
```

Array-indexing

Accessing an array-element is called *indexing*.

The element with index `i` from an array `a` is `a[i]`.

The first element has index 0.

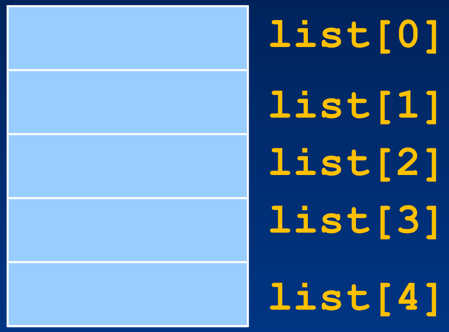
So, the last element of an array of length `n` is `a[n-1]`.

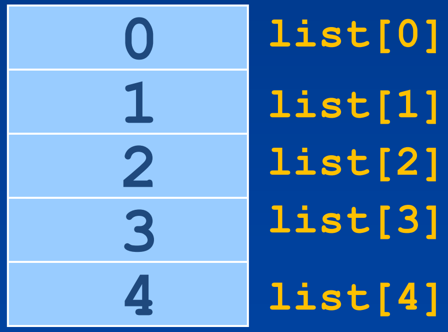
Assigning values to an array

For loops are often used to assign values to an array

Example:

```
int list[5];
for (int i=0; i<5; i++) {
   list[i] = i;
}
```





Out of bounds: segmentation fault

An array with size `n` has no element with index `n`!

Accessing `a[i]`, where `i<0` or `i>=n`, is an error: `segmentation fault`.

```
int a[10];
for (int i=0; i <= 10; i++) {
   a[i] = i;
}
```

C does not check bounds on arrays

Accessing `a[10]` may give a segmentation fault or overwrite other memory locations

Example: smallest element

```
#define N 100
int list[N];
```

Let us assume that the array is already filled with data.

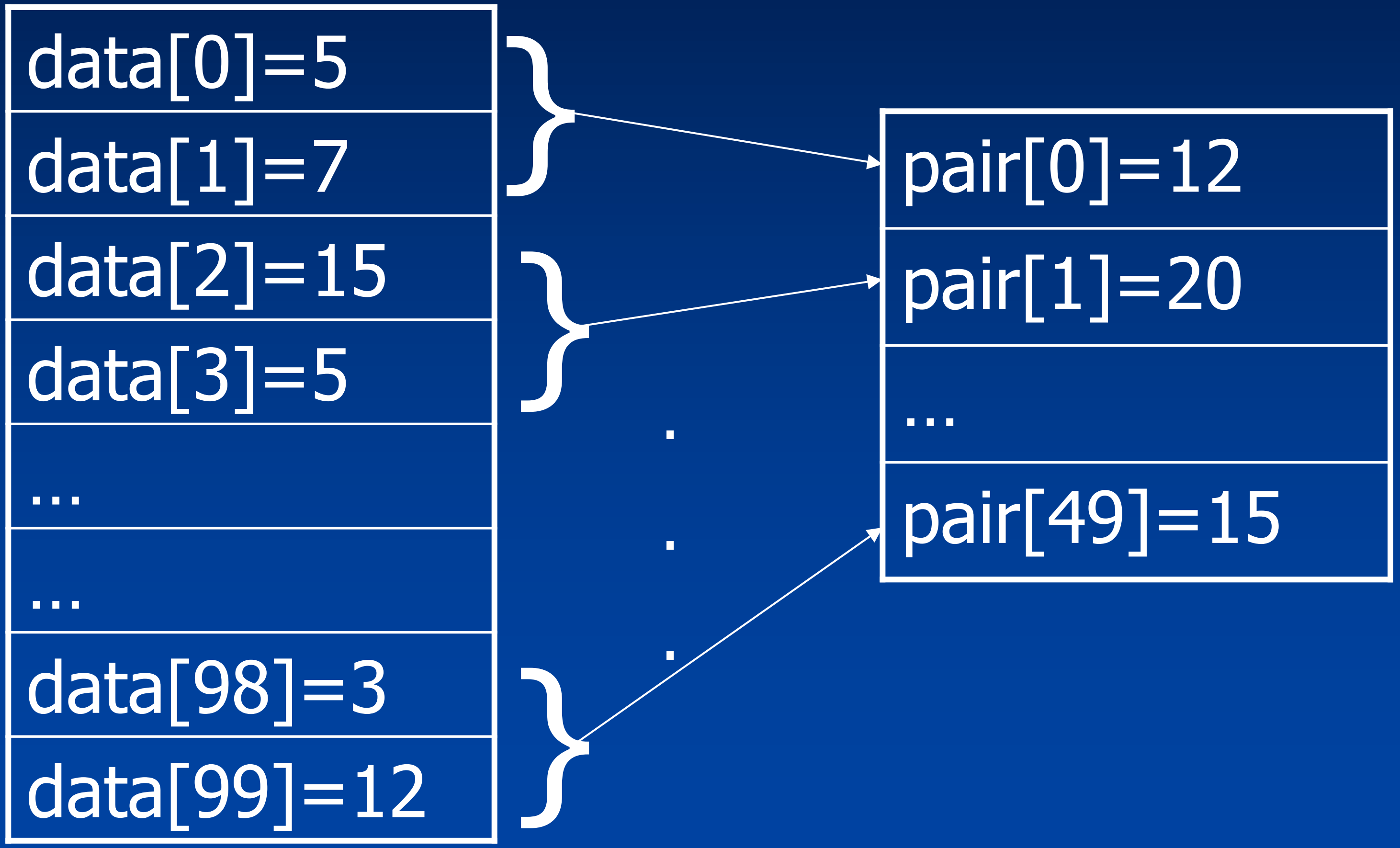
Assignment: determine the smallest element of the array and the index(es) at which it is found.

Example: smallest element

```
int minimum = list[0];
for (int index=1; index < N; index++) {
  if (list[index] < minimum) {
    minimum = list[index];
  }
}

printf("Minimum: %d\n", minimum);

printf("Location(s):");
for (int index=0; index < N; index++) {
  if (list[index] == minimum){
    printf(" %d", index);
  }
}
printf("\n");
```

Example: pair sum

Find sum of every pair in **`data`** and write in into the **`pair`** array.



Example: pair sum

```
int data[100], pair[50];
/* we assume that data contains values */

for (int i=0; i<50; i++){
   pair[i]= data[2*i] + data[2*i+1];
}
```

An alternative is:

```
for (int i=j=0; i<50; i++,j+=2){
   pair[i]= data[j] + data[j+1];
}
```

Example: wheel of fortune

Given a circle consisting of 12 squares, each containing an integer. Furthermore, there is an arrow that points to exactly one of the squares.

Assignment: write a program fragment that moves the arrow clockwise such that it points to the next square that contains the same value as the square that it initially pointed to.

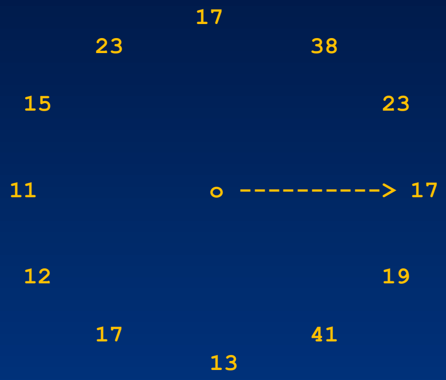
```
             17
      23             38

 15                       23

11            o -----------> 17

 12                       19

      17             41
              13
```

Example: wheel of fortune

We represent the wheel using an array:

```
int wheel[12]={17, 38, 23, 17, 19, 41, 13, 17, 12, 11, 15, 23};
```

The arrow is represented by an integer: `int arrow`.



```
  initialValue = wheel[arrow];
  arrow = (arrow + 1)%12;
  while (wheel[arrow] != initialValue) {
    arrow = (arrow + 1)%12;
  }
```

Note the repetition of the statement `arrow = (arrow + 1)%12;`
Hence, a do-while-statement may be a better choice:

```
        initialValue = wheel[arrow];
        do {
          arrow = (arrow + 1)%12;
        } while (wheel[arrow] != initialValue);
```

Call by value: swapping ints

The function `swapInts` DOES NOT swap two int variables. The function is valid C, but it is a useless function: it swaps the formal parameters a and b, not the actual parameters!

```
void swapInts(int a, int b) {
  printf("a=%d, b=%d\n", a, b);
  int h = a;
  a = b;
  b = h;
  printf("a=%d, b=%d\n", a, b);
}

int main(int argc, char *argv[]) {
  int a = 1, b = 2;
  swapInts(a,b);
  printf("a=%d, b=%d\n", a, b);
  return 0;
}
```

```
Output:

a=1, b=2

a=2, b=1

a=1, b=2
```

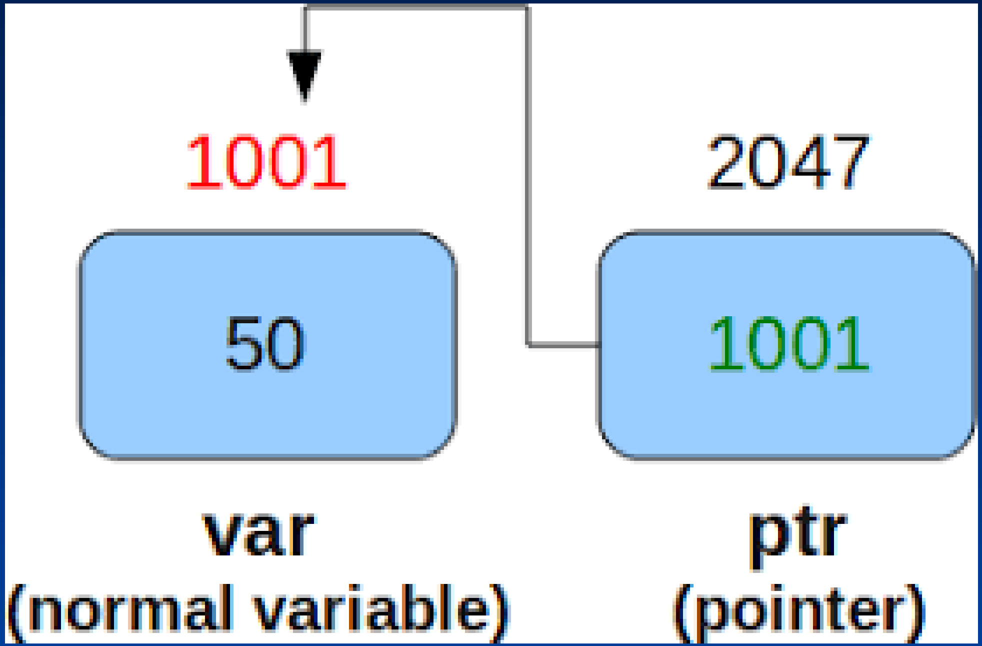
Passing an array to a function

But, the function **`swapArray`** DOES swap the two int values that are stored in the array **`a[]`**.

```
void swapArray(int a[2]) {
  printf("a[0]=%d, a[1]=%d\n", a[0], a[1]);
  int h = a[0];
  a[0] = a[1];
  a[1] = h;
  printf("a[0]=%d, a[1]=%d\n", a[0], a[1]);
}

int main(int argc, char *argv[]) {
  int a[2] = {1,2};
  swapArray(a);
  printf("a[0]=%d, a[1]=%d\n", a[0], a[1]);
  return 0;
}
```

```
Output:

a[0]=1, a[1]=2

a[0]=2, a[1]=1

a[0]=2, a[1]=1
```

Intermezzo: pointers

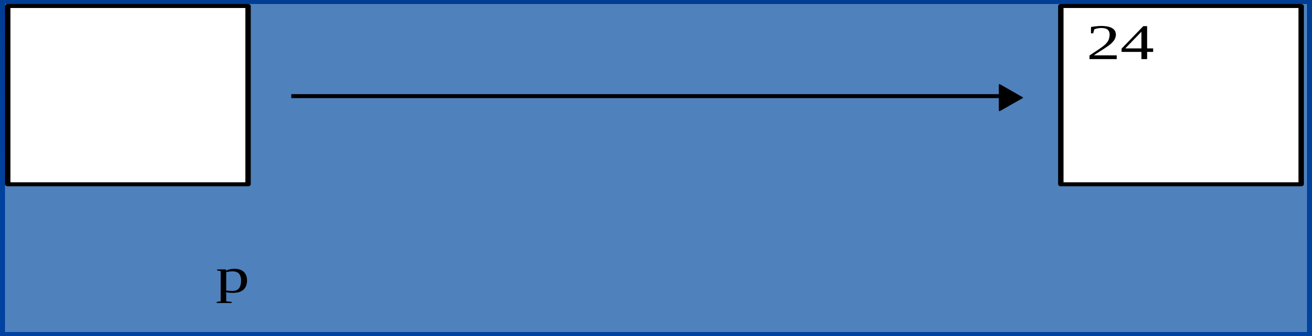


Intermezzo: Pointers

A **pointer** is a variable that contains a memory address (location in the memory of the computer).

A pointer has a name, a value and a type.

It is important to realise that there is a difference between the value of a pointer (an address) and the value to which it points (in this example 24, an `int`).



Declaring and Casting Pointers

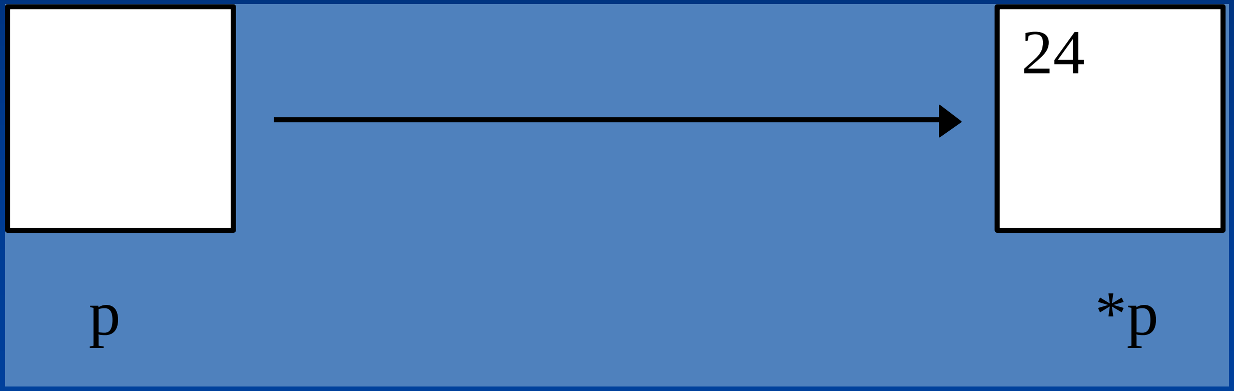
For each data type T you can define a pointer of the type "pointer to T":

`int *p;`	pointer to an `int` (i.e. an `int`-pointer)
`char *q;`	pointer to a `char` (i.e. a **char**-pointer)
`float **w;`	pointer to a `float`-pointer

Dereferencing Pointers (indirection operator)

`int *p`: `p` is an `int`-pointer

`*p` is the content of the memory location that `p` points to. So, `*p` is a ‘normal’ `int` variable.

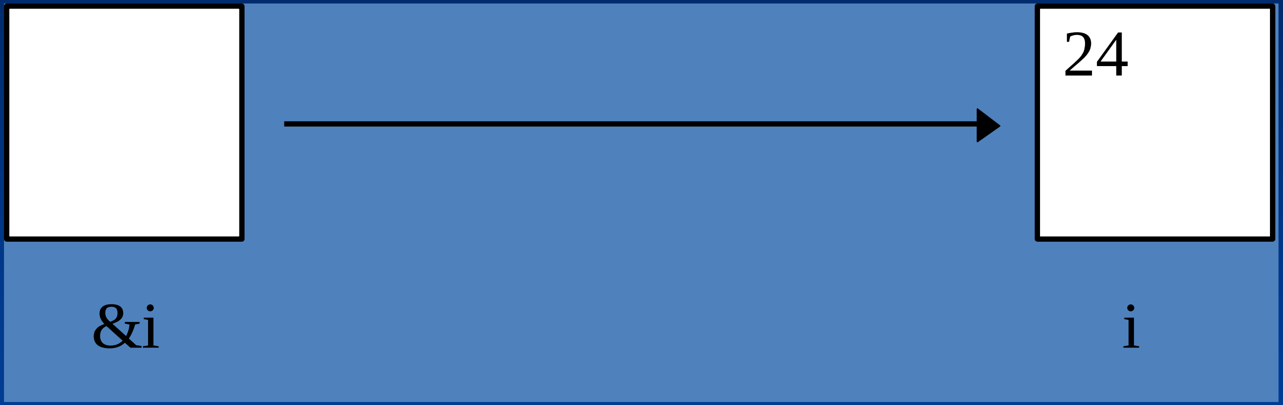


```
*p = 24;
```

Address Operator: &

`int i = 24;` `i` is a normal `int` variable

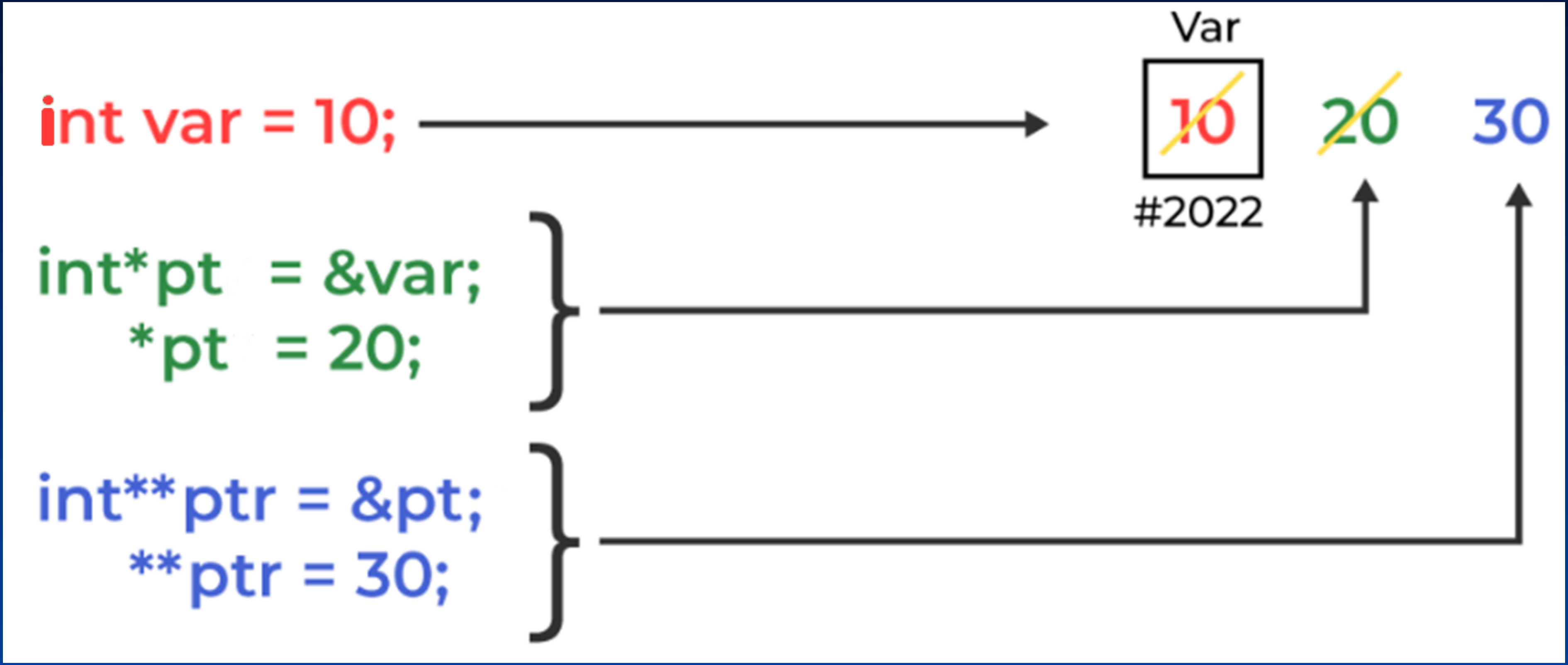
`&i` is an `int`-pointer: the address of the variable `i`



`*(&i)=*(&i)+1;`

is a clumsy notation for `i++;`

Pointers and aliasing



`*pt`, `**ptr` are the same as `var`.
We call them *aliases* of `var`.

Pointers and aliasing

```
int main () {
  int *p, q; // p is a pointer, q is a normal int

  q = 1;
  printf ("q=%d\n", q);

  *(&q) = *(&q) + 1;
  printf ("q=%d\n", q);

  p = &q;
  *p = *p + 1;
  printf ("q=%d\n", q);

  return 0;
}
```

```
Output:

1

2

3
```

Pointers as function parameters

A function can have pointer parameters.

You can use this to ‘simulate’ a parameter passing mechanism that is known as *call-by-reference*.

```
void nextInt(int *n) {
   (*n)++;
   printf("%d\n", *n);
}

int main(int argc, char *argv[]) {
   int n = 1;
   printf("%d\n", n);
   nextInt(&n);
   printf("%d\n", n);
   return 0;
}
```

```
Output:

1

2

2
```

An Exercise

- What is the output produced by this code?

```
void func (int a, int *b) {
  a = 7 ;
  *b = a ;
  b = &a ;
  *b = 4 ;
  printf("%d, %d\n", a, *b) ;
}

int main(int argc, char *argv[]) {
  int m = 3, n = 5;
  func(m, &n) ;
  printf("%d, %d\n", m, n) ;
  return 0;
 }
```

Output:

4, 4

3, 7

Swapping ints

We now know how to make a function that swaps two `int`s. We need to use call-by-reference!

```
void swapInts(int *pa, int *pb) {
   int h = *pa;
   *pa = *pb;
   *pb = h;
}
```

Of course, you need to call the function as follows:

```
swapInts(&a, &b);
```

Swapping two int pointers

We can repeat this technique to implement the swapping of two `int`-pointers. We use again call-by-reference!

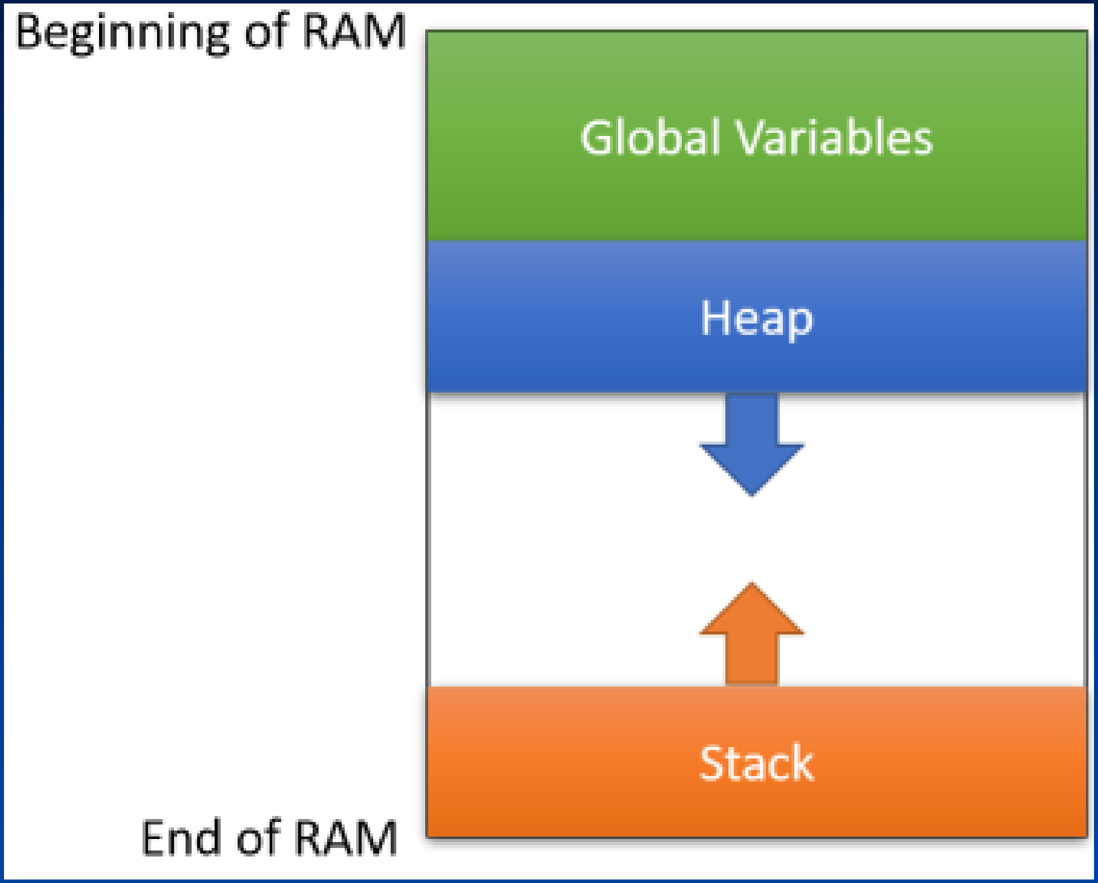
```
void swapIntPointers(int **ppa, int **ppb) {
   int *h = *ppa;
   *ppa = *ppb;
   *ppb = h;
}
```

Of course, we need to call the function as follows:

```
   swapIntPointers(&pa, &pb);
```

where `pa` and `pb` are of the type `int*`

Memory layout



Local variables and arguments to functions are *pushed* on the stack. So, by calling a function, the stack ‘grows’.

When a function returns these variables and arguments are removed (*popped*). So, on return from a function the stack ‘shrinks’.

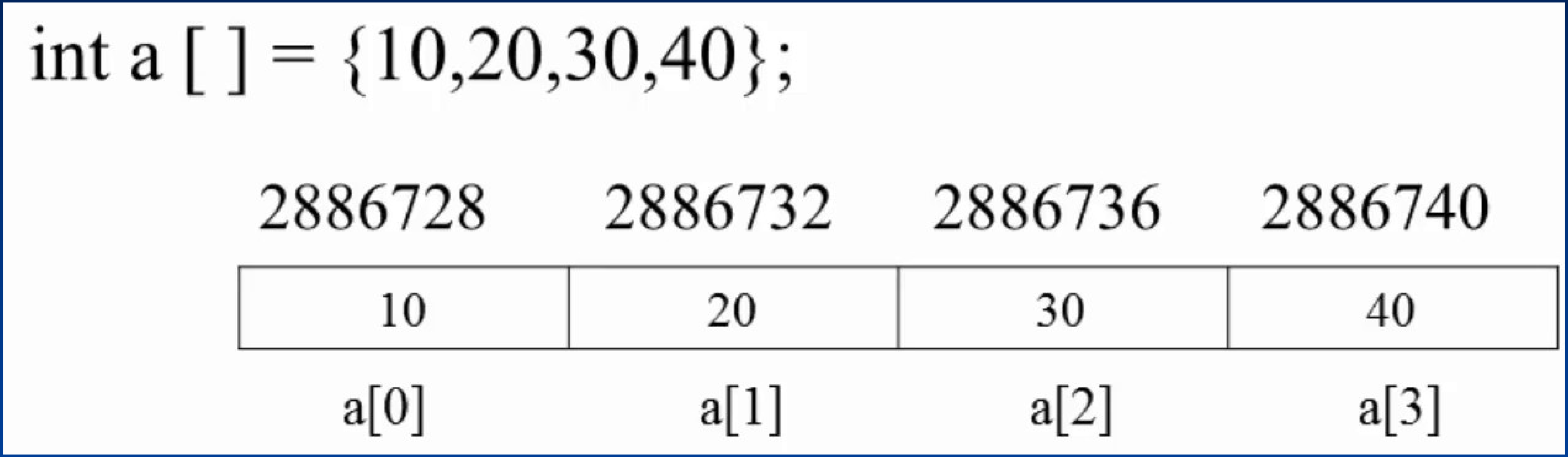
Passing Arrays to Functions

When we call a function that has an array argument, only a pointer (reference) to the first element of the array is pushed on the stack.

This avoids excessive copying of data on the stack.

Passing Arrays to Functions

```
void someFunction(int arr[]);
```



When we call `someFunction(a)` then only `&a[0]` (which is address 2886728) is pushed on the stack.

Passing Arrays to Functions

Arrays are always passed by **reference**

- Modifications to the array are reflected to the caller!

The array name is the *address* of the first element

- If `a` is an array, than the name `a` refers actually to `&a[0]`

The *actual number* of array elements that are passed will vary, so the *actual size* of the array is usually passed as another argument to the function.

Passing Arrays to Functions

So, typically we will write functions like:

```
void someFunction(int length, int arr[]);
```

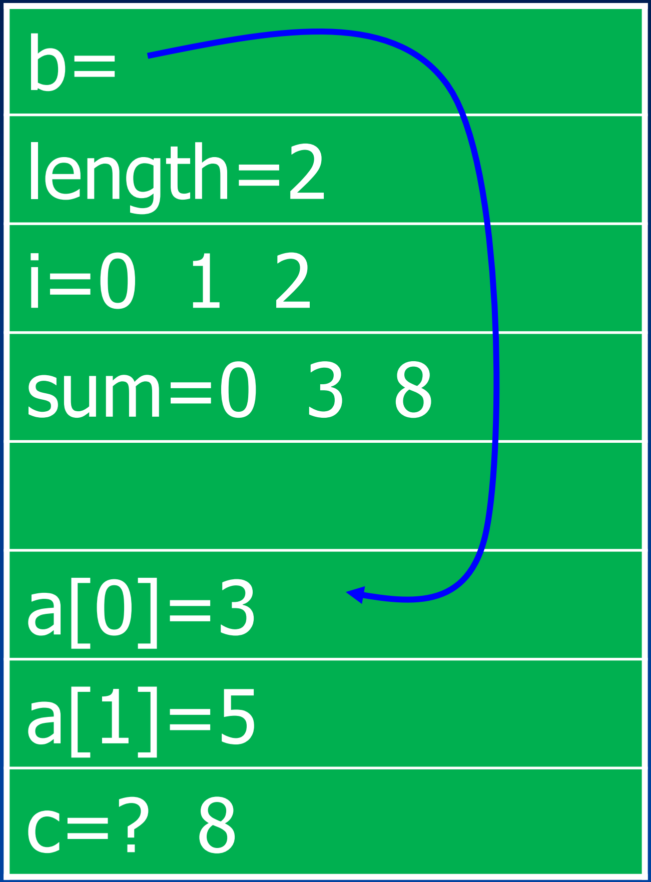
Which is equivalent with

```
void someFunction(int length, int *arr);
```

Modifications to the array arr are reflected to the caller!

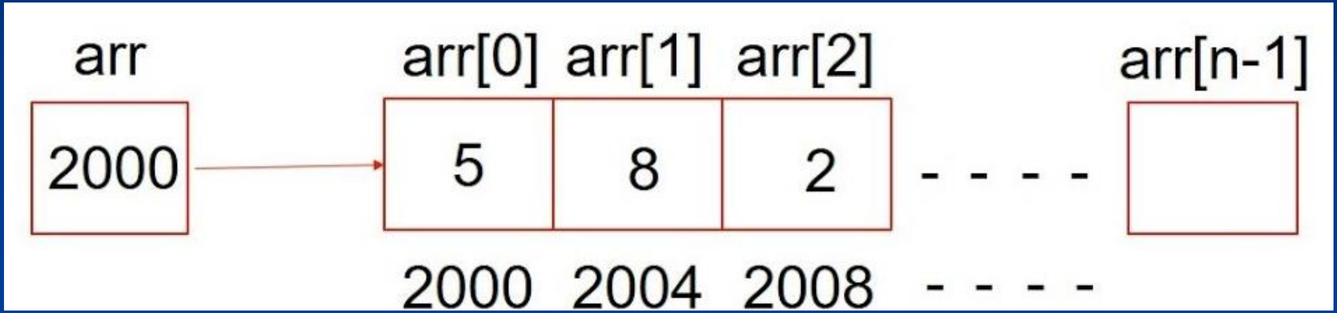
Example: summing an array

```
int arraySum(int length, int b[]) {
   int sum=0;
   for(int i=0; i < length; i++) {
     sum += b[i];
   }
   return sum;
}

int main(int argc, char *argv[]) {
  int a[2]={3, 5};
  int c = arraySum(2,a);
  printf("sum is %d\n", c);
  return 0;
}
```



Passing Arrays to Functions

```
int findValue(int n, int a[], int value) {
  // n is the length of the array
  for(int i=0; i < n; i++) {
    if (a[i] == value) {
      return i;
    }
   return -1;
}
```



```
int arr[1000];
// assume the array is filled with data
int idx = findValue(1000, arr, 2); // passes address 2000
// idx == 2
```

Example: smallest element

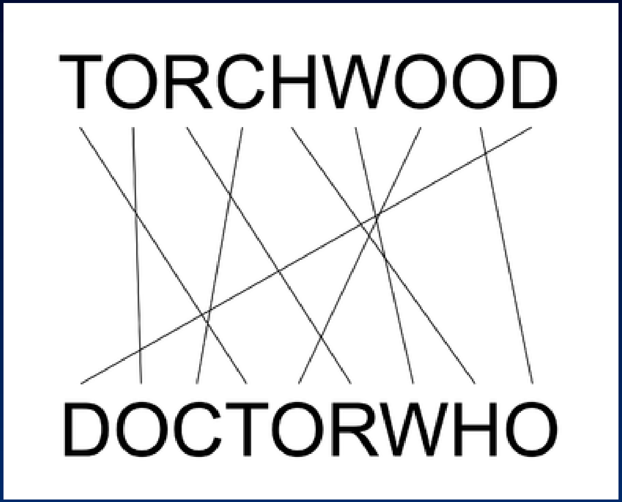
```
void printSmallestElements(int length, int *list) {
  int minimum = list[0];
  for (int index=1; index < length; index++) {
    if (list[index] < minimum) {
      minimum = list[index];
    }
  }

  printf("Minimum: %d\n", minimum);

  printf("Location(s):");
  for (int index=0; index < length; index++) {
    if (list[index] == minimum){
      printf(" %d", index);
    }
  }
  printf("\n");
}
```

min/max of an array

```
void minMax(int length, int arr[], int minmax[2]) {
  minmax[0] = arr[0];
  minmax[1] = arr[0];
  for (int i=1; i < length; i++) {
    minmax[0] = (arr[i] < minmax[0] ? arr[i] : minmax[0]);
    minmax[1] = (arr[i] > minmax[1] ? arr[i] : minmax[1]);
  }
}
```

```
int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int extrema[2];
minMax(8, a, extrema);
printf ("Minimum: %d\n", extrema[0]);
printf ("Maximum: %d\n", extrema[1]);
```

Anagram/permutation?



Given two character arrays of length 100:

```
char a[100], b[100];
```

We assume that both arrays already contain uppercase letters (‘A’..’Z’).

Task: Write a function

```
isAnagram(int length, char a[], char b[])
```

that returns 1 if `a[]` is a *permutation* of `b[]`.

Permutations

ABCD	ABDC	ACBD
ACDB	ADBC	ADCB
BACD	BADC	BCAD
BCDA	BDAC	BDCA
CABD	CADB	CBAD
CBDA	CDAB	CDBA
DABC	DACB	DBAC
DBCA	DCAB	DCBA

4!=4 x 3 x 2 x 1 = 24 permutations

Anagram/Permutation?

```
int isAnagram(int length, char a[], char b[]) {
  int i, histogramA[26], histogramB[26];
  /* initialize histograms */
  for (i=0; i<26; i++) {
    histogramA[i] = histogramB[i] = 0;
  }
  for (i=0; i<length; i++) { /* count occurrences in a[] and b[] */
    histogramA[a[i]-'A']++; /* Note the indexing! e.g 'C'-'A'== 2 */
    histogramB[b[i]-'A']++;
  }
  /* check if histogramA[i] == histogramB[i] for all i */
  for (i=0; i<26; i++) {
    if (histogramA[i]!= histogramB[i]) {
      break;
    }
  }
  return (i == 26);
}
```

Anagram/Permutation?

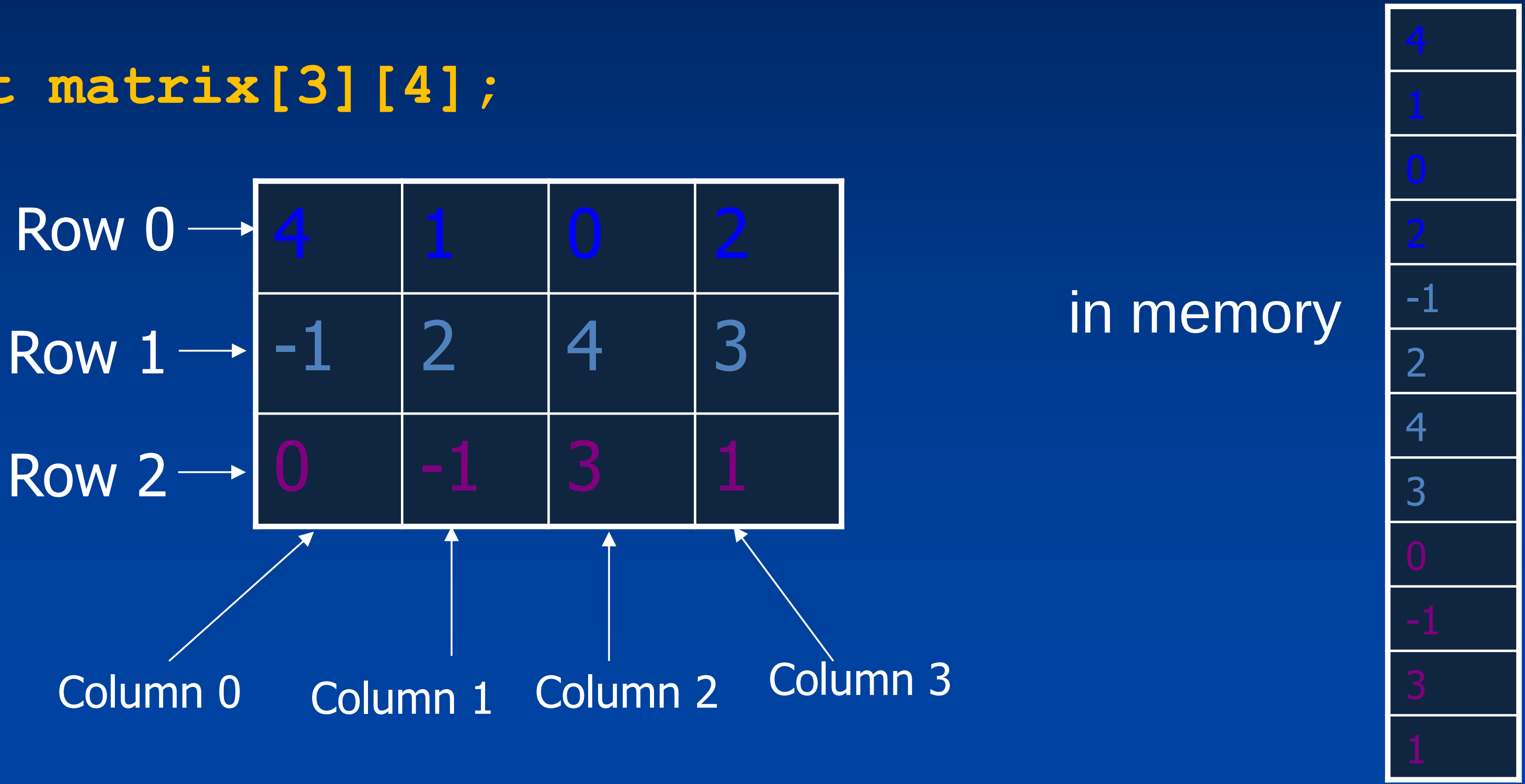
```
int isAnagram(int length, char a[], char b[]) {
  int i, diff[26];
  /* initialize histogram h[] */
  for (i=0; i<26; i++) {
    diff[i] = 0;
  }
  for (i=0; i<length; i++) {
    diff[a[i]-'A']++;
    diff[b[i]-'A']--;
  }
  for (i=0; i<26; i++) {  /* check if diff[i] == 0 for all i */
    if (diff[i]!= 0) {
      break;
    }
  }
  return (i==26);
}
```

Multi-dimensional arrays

The element type of an array may be any type.

So, the element type can itself be an array type.
This results in multi-dimensional arrays (for example, a matrix).

```
int matrix[3][4];
```



Initialization of a matrix

```
int x[3][4] = {
   {4, 1, 0, 2},
   {-1, 2, 4, 3},
   {0, -1, 3, 1}
};

int x[][4] = {
   {4, 1, 0, 2},
   {-1, 2, 4, 3},
   {0, -1, 3, 1}
};
```

$$\begin{array}{l|c|c|c|c|}
\hline
\text{Row 0} \rightarrow & 4 & 1 & 0 & 2 \\
\hline
\text{Row 1} \rightarrow & -1 & 2 & 4 & 3 \\
\hline
\text{Row 2} \rightarrow & 0 & -1 & 3 & 1 \\
\hline
 & \uparrow & \uparrow & \uparrow & \uparrow \\
 & \text{Column 0} & \text{Column 1} & \text{Column 2} & \text{Column 3}
\end{array}$$

Exercise

Write a nested loop to initialize a 2D array as follows

0	1	2
1	2	3
2	3	4
3	4	5

```
int x[4][3];
for(int i=0; i<4; i++){
   for(int j=0; j<3; j++) {
      x[i][j] = i + j;
   }
}
```

Multi-dimensional arrays

`int x[3][4];`
A matrix is an array of arrays.

So, **`x[1]`** is an array of 4 elements (type **`int[]`**).

This is the reason why we write **`x[i][j]`**, and NOT **`x[i,j]`**.

Summing rows of a matrix

```
int arraySum(int length, int a[]) { /* returns sum of 1D array */
   int sum=0;
   for(int i=0; i < length; i++) {
     sum += a[i];
   }
   return sum;
}

int main(int argc, char *argv[]) {
  int x[3][4] = {
    {4, 1, 0, 2},
    {-1, 2, 4, 3},
    {0, -1, 3, 1}
  };
  for (int i=0; i<3; i++) {
    printf("sum in row %d is %d\n", i, arraySum(4, x[i]));
  }
  return 0;
}
```

End week 3