

03 C Programming

CSC 230

Department of Computer Science
University of Victoria

C Programming

These notes cover aspects of C that are relevant to interfacing with ARM code and the understanding of computer architecture and organization

These are NOT complete notes on C programming

These notes are only a starting point → consult any C manual (not just online)

Important Concepts Here

- ❑ Variables and memory allocation for them
- ❑ Pointers
- ❑ Regular control structures (IF/ELSE, WHILE, FOR, SWITCH)
- ❑ Data structures, static allocation (arrays up to 2 dimensions)
- ❑ Declaration of a function or procedure
- ❑ Call to a function or procedure
- ❑ Input parameters passing by value
- ❑ Input parameters passing by reference
- ❑ Returning output parameters
- ❑ File I/O in Lab and by self study (documentation posted)

Small Example 1

```
#include <stdio.h>          /* include standard library */

int main() {                /* define a function main */
    int a, b, c;            /* declare variables */
    a = 3;                  /* assignment statements */
    b = 5;
    c = a + b;
    printf("a + b = %d\n", c); /* call libr. function */
    return 0;
}
```

main

as a function called by OS
and that returns to OS

What exactly does a
declaration of "int" mean?

C Integer Datatypes (assuming 32 bit words)

short 16 bit integer → short h1, h2, h3;
 or short int h1, h2, h3;

int 32 bit integer → int i, j, k;

Long 32 or 64 bit integer? → long a, b, c;
 or long int a, b, c;

char 1 byte character → char temp;

Note: we can also attach the keyword **unsigned** to these declarations → unsigned int clock;

What does a declaration of an integer type mean exactly?

```
int  Csize, Num;
```

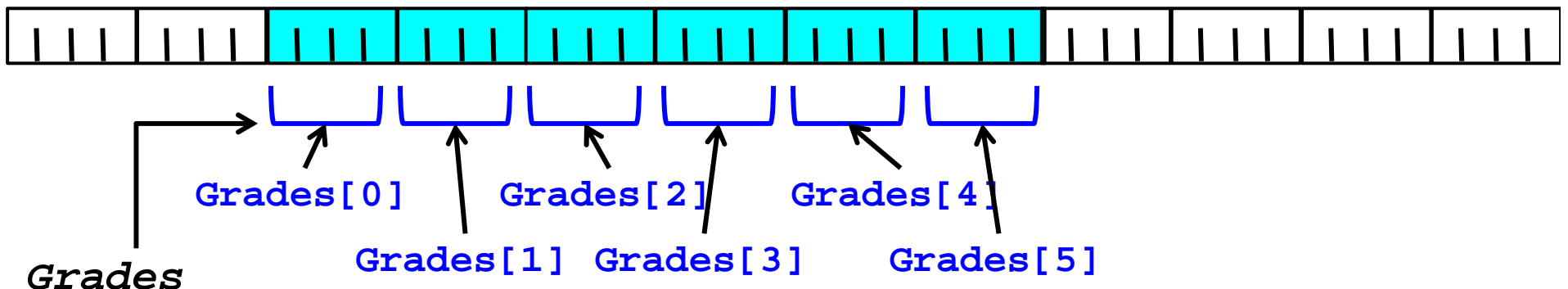
1. Ask to allocate space in RAM for 2 words of 32-bits (4 bytes each).
2. *Where are they?* No need to know precisely where in RAM, they are within the same “frame” of space in RAM where the program is allocated space overall. By using the labels the corresponding address is referenced and the location in memory accessed.
3. In the code, whenever “Csize” is used directly, it refers to the *content* of memory at the address for the allocation of “Csize” → the program need never know the address itself.
4. Similarly for “Num” .

What about a data structure like an array?

Consecutive elements
Homogeneous type

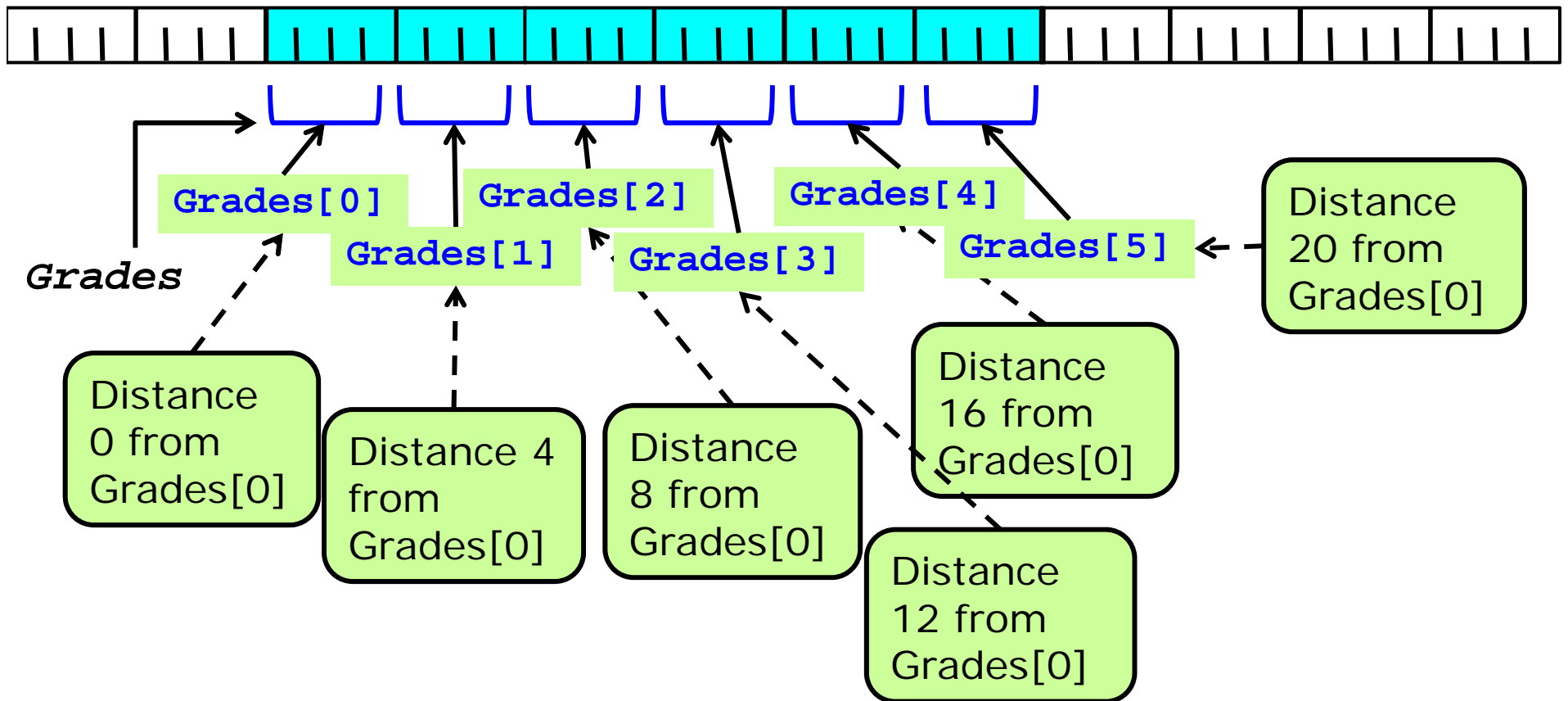
```
int Grades[6];
```

1. It asks to allocate in memory 6 consecutive words of 32 bits each = 4 bytes each
2. The array name refers to the address of the leftmost word at position [0]
3. Each word can be accessed by its index relative to position [0] and its given address in memory



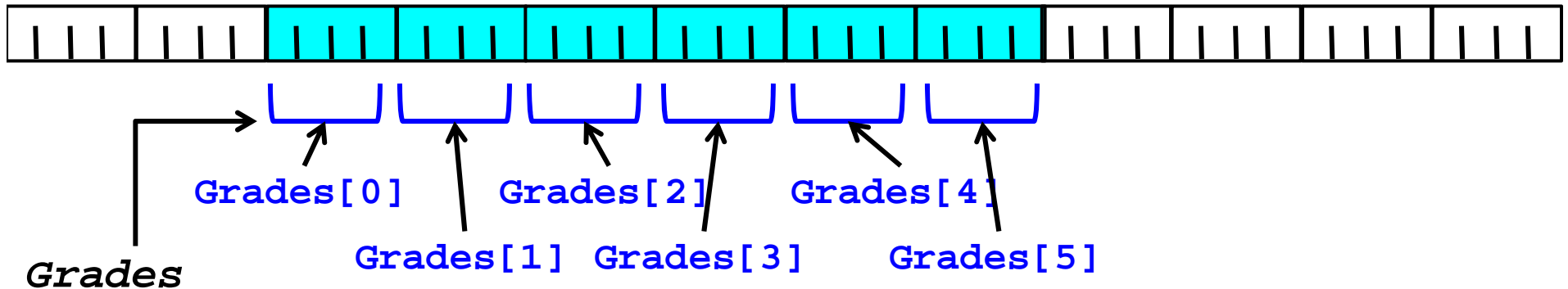
Consider each element of the array

```
int Grades[6];
```



Consider the addresses of each element of the array

```
int Grades[6];
```



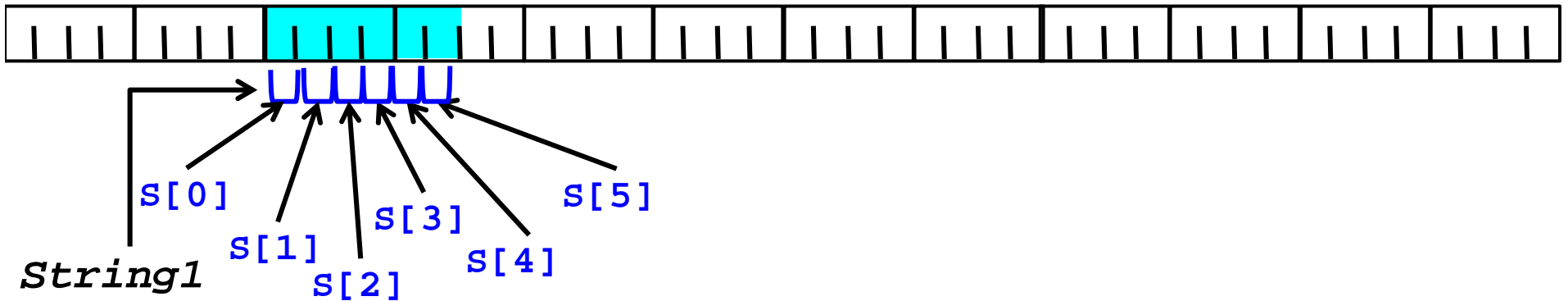
If the array **Grades** has been allocated space in RAM by the OS at address 0x0000 0124 (as an example), then:

Grades[0]	at address →	0x0000 0124
Grades[1]	at address →	0x0000 0128
Grades[2]	at address →	0x0000 012C
Grades[3]	at address →	0x0000 0130
Grades[4]	at address →	0x0000 0134
Grades[5]	at address →	0x0000 0138

We can access each element through pointers or through regular indexes

Similarly for an array of characters

`char S[6];`



If the array `S` has been allocated space in RAM by the OS at address `0x0000 01A8` (as an example), then:

<code>S[0]</code>	at address	→	<code>0x0000 01A8</code>
<code>S[1]</code>	at address	→	<code>0x0000 01A9</code>
<code>S[2]</code>	at address	→	<code>0x0000 01AA</code>
<code>S[3]</code>	at address	→	<code>0x0000 01AB</code>
<code>S[4]</code>	at address	→	<code>0x0000 01AC</code>
<code>S[5]</code>	at address	→	<code>0x0000 01AD</code>

We can access each element through pointers or through regular indexes

The C Language

Main characteristics :

- ❑ created as a language for systems programming,
- ❑ easy access to specific machine locations,
- ❑ easy manipulation of individual bits,
- ❑ every construct executes efficiently.

❑ There are two main versions of C:

- ✓ **ANSI C** (American National Standards Institute)
- ✓ **ISO C** (International Standards Organization)
- ✓ (*Original C* also known as *K&R C* (for UNIX by Kernighan and Ritchie))

Portability of C Code

- ❑ C is used as a portable programming language
 - It runs on (almost) all platforms, ranging from the smallest microprocessor to the largest supercomputer.
- ❑ A portability issue is that the sizes of some basic data types are not rigidly defined.
 - The `int` type is simply an integer with a natural size for the computer.
 - It is 16 bits on very old machines and some embedded systems,
 - it is 32 bits on older machines and most embedded systems,
 - it is 64 bits on almost all newer machines and some embedded systems.
 - The sizes of the `float` and `double` types (and other types) similarly depend on the computer.
- ❑ Truly portable C code must use the preprocessor. (More later.)

Portability of C

Did you know:

32 bit UNIX systems have a variation of the Y2K problem. UNIX systems store times as the number of seconds since January 1, 1970.

On Tuesday, 19 January 2038 the seconds will overflow!

- ❑ C is used as a portable language.
 - It runs on (almost) all microprocessors.
- ❑ A portability issue arises if the data types are not right.
 - The `int` type is similar on all computers.
 - It is 16 bits on some (old) computers.
 - it is 32 bits on most current machines, and
 - it is 64 bits on some newer machines.
 - The sizes of the `float` and `double` types (and other types) similarly depend on the computer.
- ❑ Truly portable C code must use the preprocessor. (More later.)

2-dimensional Arrays

```
char Matrix[4][4];
```

*how is memory
organized
for these?*

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	A	B
3	C	D	E	F

2-dimensional Arrays

```
char CH[4][4];
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
CH[0,0]	CH[0,1]	CH[0,2]	CH[0,3]	CH[1,0]	CH[1,1]	CH[1,2]	CH[1,3]	CH[2,0]	CH[2,1]	CH[2,2]	CH[2,3]	CH[3,0]	CH[3,1]	CH[3,2]	CH[3,3]

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	A	B
3	C	D	E	F



Row major ordering:

it assigns successive elements to successive memory locations, moving across each row from top to bottom.

2-dimensional Arrays

```
char CH[4][4];
```

0	4	8	C	1	5	9	D	2	6	A	E	3	7	B	F
CH[0,0]	CH[1,0]	CH[2,0]	CH[3,0]	CH[0,1]	CH[1,1]	CH[2,1]	CH[3,1]	CH[0,2]	CH[1,2]	CH[2,2]	CH[3,2]	CH[0,3]	CH[1,3]	CH[2,3]	CH[3,3]

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	A	B
3	C	D	E	F



Column major ordering:

it is the other allocation frequently used.

FORTRAN and various dialects of BASIC
(e.g., older versions of Microsoft
BASIC) use this method to index arrays.

Overall Structure of a C Program

preprocessor statements

#include

#define

type definitions

not covered in this course

global variable declarations

and we will avoid these

{ functions, declarations }

one of the functions in the
program must be named
main



Small Example 1

```
#include <stdio.h>          /* include standard library */

int main() {                /* define a function main */
    int a, b, c;            /* declare variables */
    a = 3;                  /* assignment statements */
    b = 5;
    c = a + b;
    printf("a + b = %d\n", c); /* call libr. function */
    return 0;
}
```

*main function should return 0
to indicate success (non-zero
to indicate an error)*

Preprocessor statements

#include to include declarations from libraries

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

#define text replacement – useful for constants
(it will be an EQU in assembly language)

```
#define     MAXSIZE   10
.....
int myarray[MAXSIZE];
```

Easy Output

```
int  x;  
printf("whatever message you want here  \n");  
printf("whatever number x = %d as you like \n", x);
```

newline character

format for integer

(%d, %2d, %3d...)

Output:

- strings of characters for messages, headings, etc.
- integers

Easy Output: better and more precise

```
int x;  
fprintf(stdout, "whatever message you want here \n");  
fprintf(stdout, "number x = %d as you like \n", x);
```

FPRINTF

- because all output is really directed to a file
- STDOUT is the default "file" for standard output=screen
- custom file name can be placed (see Notes on I/O in C)

*use it this way
in your code*

Easy Input – Read an integer

```
int y;  
scanf("%d", &y);
```

← address of y

← format for integer

```
long x;  
scanf("%ld", &x);
```

← address of x

← format for long integer

Also look at **fscanf** to read from an opened input file.
(More on **scanf** later)

Declaration of variables and initialization

```
int i, j, k;  
char cx, tt;
```

```
int i = 0;  
char c = 's';
```

-- declaration with initialization

- ❑ Variables must be declared BEFORE they are used.
- ❑ Variables may be global or local.
- ❑ Constants: integers, characters, floating point numbers and strings.
- ❑ Some special characters: `\n` `\t` `\'` `\"` `\\`
- ❑ Hexadecimal numbers: `0x45DF`

Small Example (2D array)

```
int Rsize1 = 3;  
int Csize1 = 3;  
int mymatrix[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

1	2	3
4	5	6
7	8	9

**Declare and initialize (compile-time initialization) a 3 x 3
2-dimensional array of integers**

Control Structures .1

initialize
test for continuing
advance to next iteration

```
for( expr1; expr2; expr3 )  
    statement
```

```
while(expression)  
    statement
```

```
do  
    statement  
while (expression)
```

Statements are terminated
by a semicolon ;

A *statement* in any structure
can be a sequence of
statements enclosed in { }

```
{  
    statement;  
    statement;  
    :  
    statement;  
}
```

Control Structures .2

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

```
if (expression)  
{  
    statements;  
}  
else {  
    statements;  
}
```

```
if (expression1)  
    statement1  
else if (expression2)  
    statement2  
else if (expression3)  
    statement3  
...  
else  
    statement_n
```

```
switch(expression) {  
    case const1:  
        statement1;  
        break;  
    case const2:  
        statement2;  
        break;  
    ...  
    default:  
        statement_n;  
        break;  
}
```

Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division (int. quotient)
%	modulus (remainder)
++	increment (pre or post)
--	decrement (pre or post)

Bitwise Operators

&	AND
	OR
^	XOR
~	1's complement (invert)
>>	right shift
<<	left shift

Assignment Operators

= += -= *= /= &= |= ^=

Relational and Logical Operators

(the result is always 0 or 1)

==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
&&	AND
	OR
!	NOT

Pointer Operators

&	address of
*	de-reference (follow the ptr)

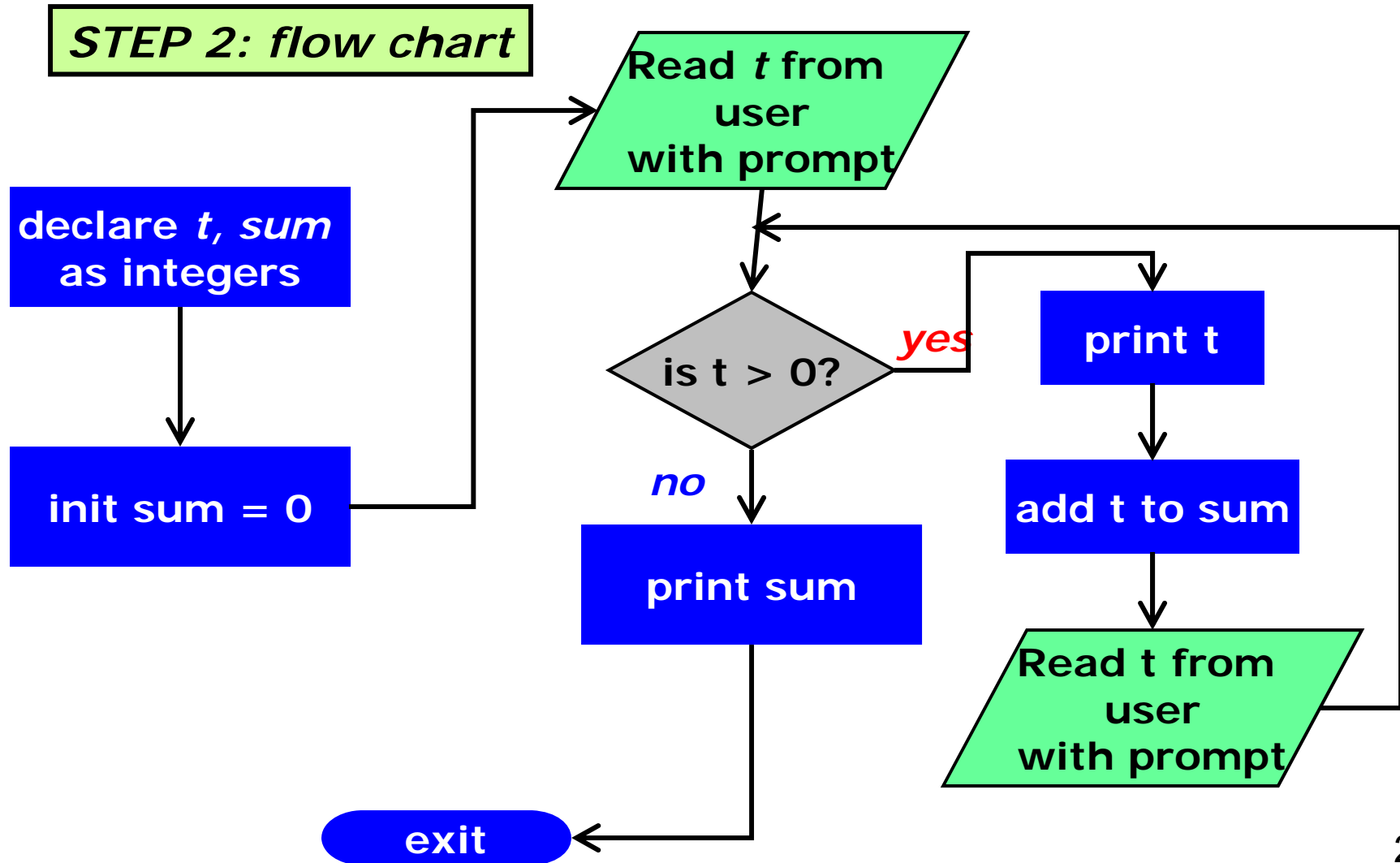
A pointer is a variable whose value is the address of another variable, i.e. a variable that points to another variable.

Small programming problem 1: read a series of positive integers from users, print and sum them. Stop after receiving negative integer and print sum.

STEP 1: pseudo-code in precise English

- 1. Declare variables for integers and for sum*
- 2. Initialize sum to 0*
- 3. Read integer from user*
- 4. While integer is positive then*
 - print integer,*
 - add it to sum*
 - and read next one (go to top of loop)*
- 5. Print sum and exit*

Small programming problem 1: read a series of positive integers from users, print and sum them. Stop after receiving negative integer and print sum.



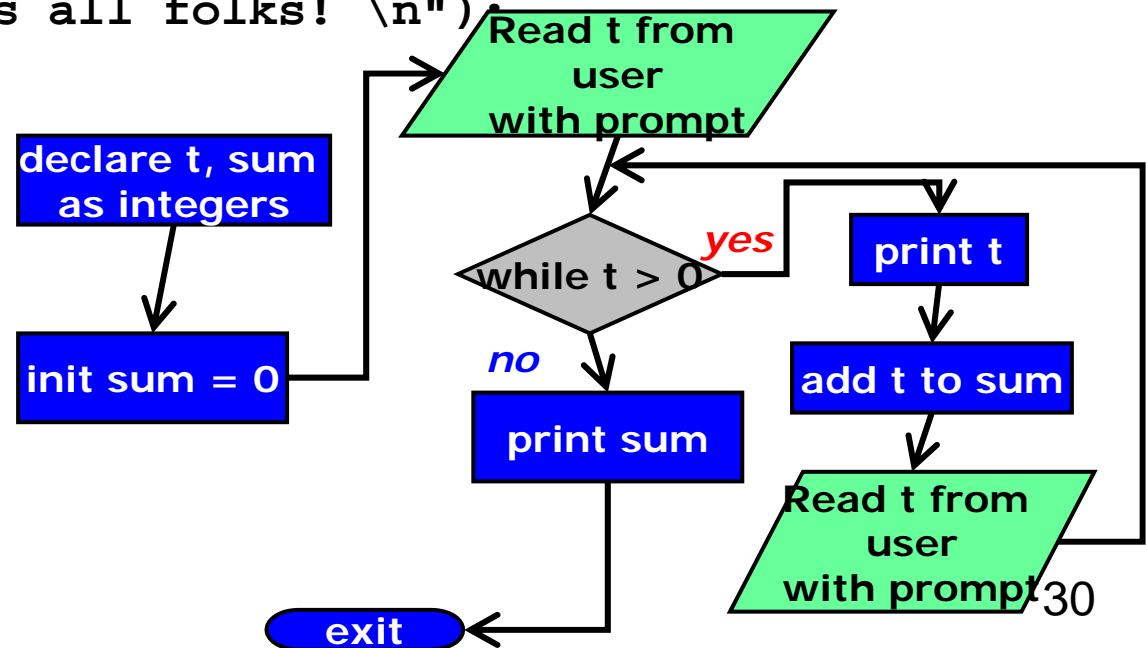
```

/* read positive integers from users and sum them.
Stop when negative integer and print sum */
#include <stdio.h>
int main() {
    int t;
    int sum;
    sum = 0;
    fprintf(stdout, "Enter a positive integer to sum, or"
                " a negative one to quit \n");
    . . . . .

    fprintf(stdout, "Sum is %d \n", sum);
    fprintf(stdout, "That's all folks! \n");
    return 0;
} /* end main */

```

STEP 3a: code the beginning and end of program output



```
/* read positive integers from users and sum them.  
Stop when negative integer and print sum */
```

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

```
int main() {
```

```
    int t;
```

```
    int sum;
```

```
    sum = 0;
```

```
    printf("Enter a positive integer to sum, or"  
           " a negative one to quit \n");
```

```
    scanf("%d", &t);
```

```
    printf("t is %d \n", t );
```

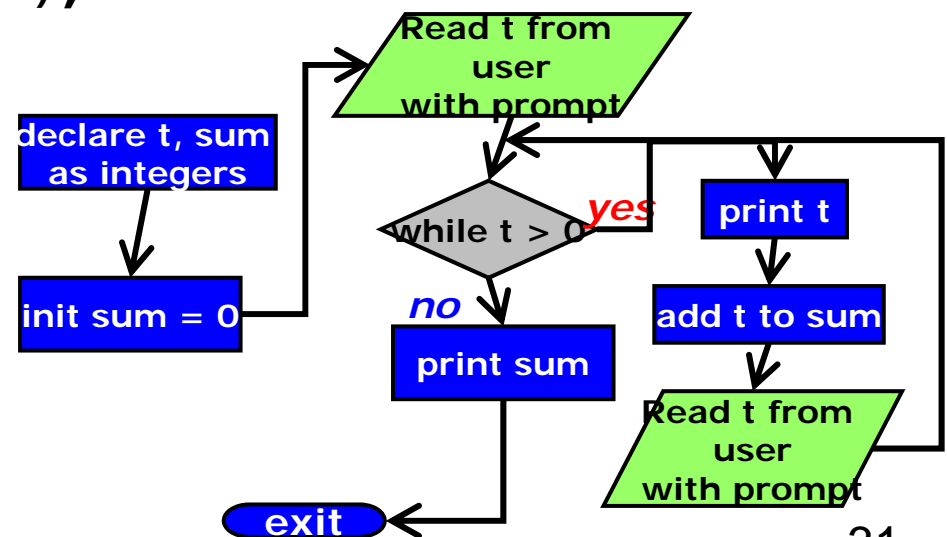
```
    printf("Sum is %d \n", sum);
```

```
    printf("That's all folks! \n");
```

```
    return 0;
```

```
} /* end main */
```

STEP 3b: Read and echo input once – test it out!



```
/* read positive integers from users and sum them.
Stop when negative integer and print sum */
```

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

```
int main() {
```

```
    int t;
```

```
    int sum;
```

```
    sum = 0;
```

```
    printf("Enter a positive integer to sum, or"
           " a negative one to quit \n);
```

```
    scanf("%d", &t);
```

```
    while(t >= 0) {
```

```
        printf("t is %d \n", t );
```

```
        .../* processing will be here */.....
```

```
        printf("Enter a positive integer to sum, or"
               " a negative one to quit \n);
```

```
        scanf("%d", &t);
```

```
    } /*end while*/
```

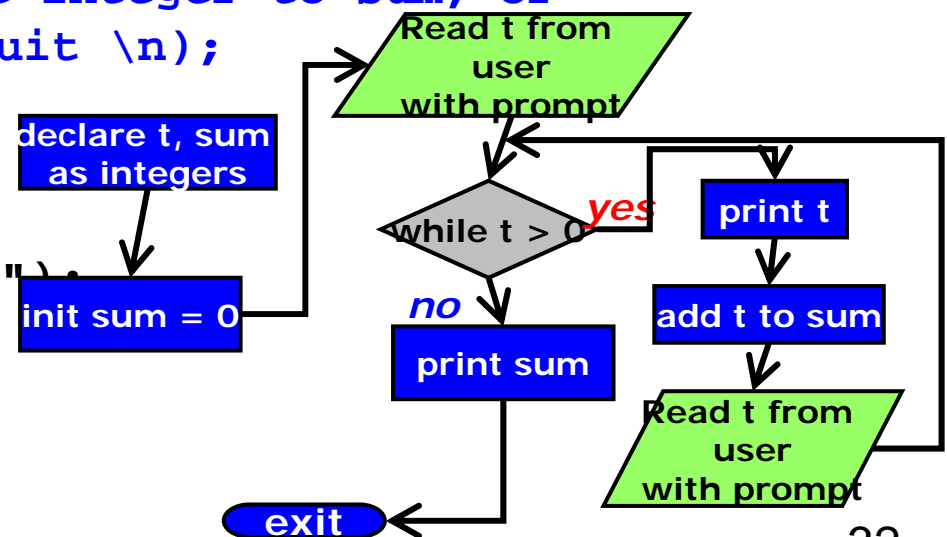
```
    printf("Sum is %d \n", sum);
```

```
    printf("That's all folks! \n");
```

```
    return 0;
```

```
} /* end main */
```

**STEP 3c: Loop with
echoing of input only –
leave computing till later**




```
/* read positive integers from users and sum them.
Stop when negative integer and print sum */
```

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

```
int main() {
```

```
    int t;
```

```
    int sum;
```

```
    sum = 0;
```

```
    printf("Enter a positive integer to sum, or"
           " a negative one to quit \n");
```

```
    scanf("%d", &t);
```

```
    while(t >= 0) {
```

```
        printf("t is %d \n", t );
```

```
        sum = sum + t;
```

```
        printf("Enter a positive integer to sum, or"
               " a negative one to quit \n");
```

```
        scanf("%d", &t);
```

```
    } /*end while*/
```

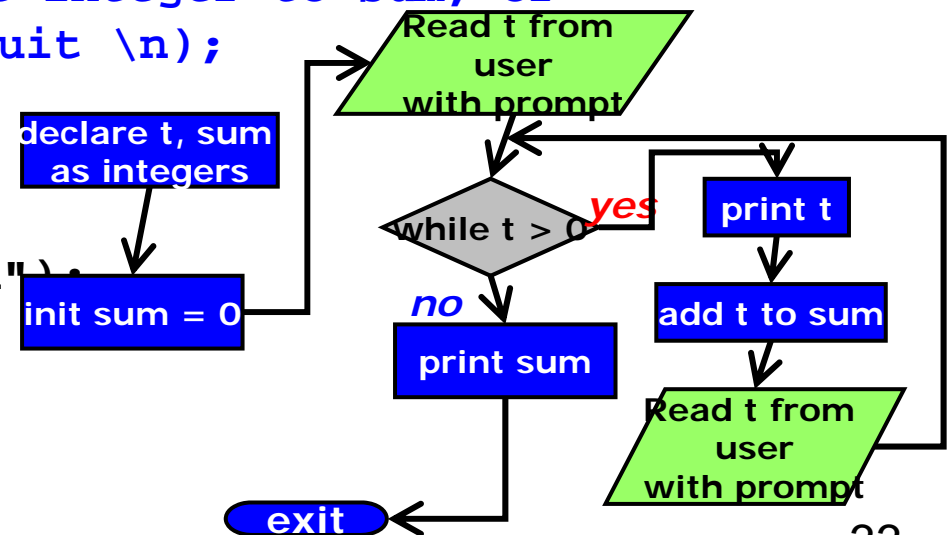
```
    printf("Sum is %d \n", sum);
```

```
    printf("That's all folks! \n");
```

```
    return 0;
```

```
} /* end main */
```

STEP 4: code the processing inside



```

/* read positive integers from users and sum them.
Stop after receiving negative integer and print sum */
#include <stdio.h>
int main() {
    char *prompt = "Enter a positive integer to sum, or"
                  " a negative one to quit \n";

    int n;
    int sum;
    sum = 0;
    printf(prompt);
    scanf("%d", &n);
    while(n >= 0) {
        printf("n is %d \n", n );
        sum = sum + n;
        printf(prompt);
        scanf("%d", &n);
    } /*end while*/
    printf("Sum is %d \n", sum);
    printf("That's all folks! \n");
    return 0;
} /* end main */

```



variations....

Operators: by examples

Post - increment
Post - decrement

```
int x;
```

```
x++;  $\leftrightarrow$  x=x+1;
```

```
x--;  $\leftrightarrow$  x=x-1;
```

Pre - increment
Pre - decrement

```
int y;
```

```
++y;  $\leftrightarrow$  y=y+1;
```

```
--y;  $\leftrightarrow$  y=y-1;
```

```
int b,x;  
b=5;  
x=b++;
```

\Rightarrow $\begin{matrix} x = ? & x = 5 \\ b = ? & b = 6 \end{matrix}$

```
int b,x;  
b=5;  
x=++b;
```

\Rightarrow $\begin{matrix} x = ? & x = 6 \\ b = ? & b = 6 \end{matrix}$

use only if you truly understand the semantics

Operators: by examples

Bitwise operators

0 0 1 0 1 0 1 1

&

1 0 1 1 1 0 0 1

=

0 0 1 0 1 0 0 1

5 | ~3;

0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1
or ~
1 1 1 1 1 1 0 0

0 0 0 0 0 1 0 1 (5)
1 1 1 1 1 1 0 0 (~3)
1 1 1 1 1 1 0 1 (5 | ~3)

```
int b;  
scanf("%d",&b);  
if ((b & 32) != 0) {  
    printf("Valid number\n");  
}
```



What condition is checked here?

The given number has the 6th least significant bit equal to 1.

(32 = 00...0100000)

Array Example A1.1

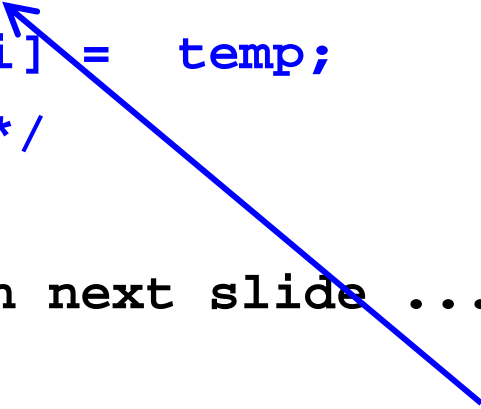
```
/* first attempt to read and echo an array of integers*/
/* Enter number of elements in an array followed by one
   element per line*/
#include <stdio.h>
#define MAXARRAY 100 /*limit on array size*/
int main() {
    int size;
    int i, temp;
    int myArray[MAXARRAY];

    printf("Enter a positive integer = size of array or
           Enter a negative integer to quit \n");
    scanf("%d", &size); /*read actual size of array*/
```

Array Example A1.2

```
/*read in elements of array and store*/
while( size > 0 ) {
    printf("Size is %d  \n", size);
    printf("Enter elements one per line\n");
    for( i = 0; i < size; i++ ) {
        scanf("%d", &temp);
        myArray[i] = temp;
    } /* end for */

    /* continued on next slide ... */
}
```



*First element of array is
myArray[0], last element is
myArray[n-1]*

Array Example A1.3

```
/*print out the array*/
printf("Array elements are:\n");
for( i = 0; i < size; i++ ) {
    temp = myArray[i];
    printf("%d \n", temp);
} /* end for */
printf("\n");

/* ask for next array if any */
printf("Enter a positive integer = size of
       array or Enter a negative integer
       to quit \n"););

scanf("%d", &n);
} /* end while */
printf("That's all folks!\n");
return 0;
}
```

Compound Data Types - Arrays

```
int    a[20], b[5];      /*one-dimensional array (vector)*/  
int    table [10][5];    /*two-dimensional array */
```

```
a[0] = a[1] = 1;          /* store the first 20 */  
for(i = 2; i < 20; i++) { /* Fibonacci numbers */  
    a[i] = a[i-2] + a[i-1];  
}
```

There are useful library functions (NOT used in this course):

```
#include <string.h>  
int    vec1[10], vec2[10]  
  
/* block copy vec1 to vec2 */  
bcopy (vec2, vec1, sizeof(vec1) );  
memcpy(vec2, vec1, sizeof(vec1) );  
  
/* block compare */  
test = bcmp (vec1, vec2, sizeof(vec1) );  
test = memcmp (vec1, vec2, sizeof(vec1) );
```


Compound Data Types - Pointers

Pointer variables hold values which are the *addresses* of data objects in memory

```
int    *ip;
```

```
char    *cp1, *cp2;
```

3 pointer variables shown here:

➤ *ip* can contain addresses of data objects of type `int`

➤ *cp1* and *cp2* can contain addresses for type `char`

Note the difference:

```
int v;    /* creates a variable called v of type int
           (integer), the content of v is an integer value */
```

```
int *vp;  /* creates a variable vp of type pointer to integer.
           The value (content) of vp is an address
           of an integer */
```

Pointers, pointers, pointers.....

```
int    x, y;    /* integer x */
int    *px;     /*pointer to an integer */

px = &x;        /* "&" is a unary operator which obtains
                  the address of a data location*/

y = *px;        /* "*" is the operator to dereference from
                  a pointer (gets to the data value).
                  That is, *px is the content of the
                  location to which the pointer px points*/

y = x;          /*same as both lines above executed together*/
```

Operations on pointers:

assignment, comparison, add/subtract a constant

```
char  *cp1, linebuffer [100];
.....
cp1 = linebuffer;      /*same as cp1 = &linebuffer[0] */
                       /* cp1 points to first element */

*cp1 = 'A';            /* means linebuffer[0] = 'A' */

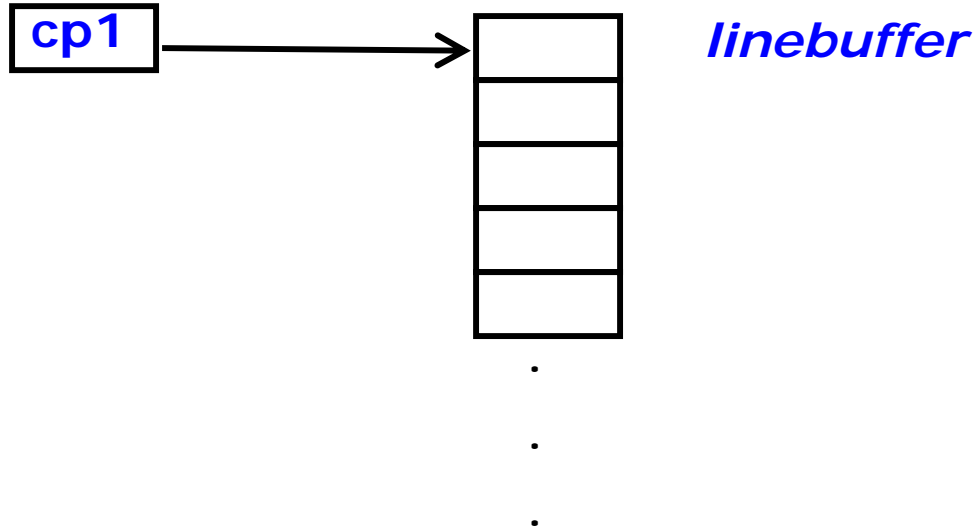
cp1++;                 /*advance to 2nd element*/

*cp1 = 'B';            /* means linebuffer[1] = 'B' */

cp1++;                 /*advance to 3rd element*/

*cp1 = 'C';            /* means linebuffer[2] = 'C' */
.....
```

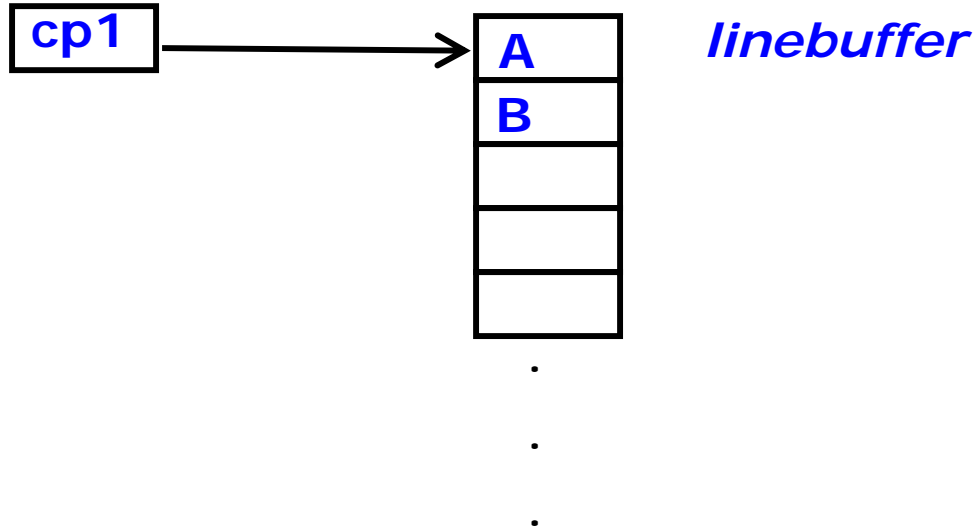
```
char  *cp1, linebuffer [100];  
.....  
cp1 = linebuffer;      /* same as cp1 = &linebuffer[0] */  
                        /* cp1 points to first element */
```



```

char  *cp1, linebuffer [100];
.....
cp1 = linebuffer;      /* same as cp1 = &linebuffer[0] */
                        /* cp1 points to first element */
*cp1++ = 'A';          /* means linebuffer[0] = 'A'
                        and advance to 2nd element */
*cp1++ = 'B';          /* means linebuffer[1] = 'B' */
                        and advance to 3rd element */
.....

```



Pointers and arrays (advanced)

Equivalence between *pointer dereferencing* and *subscripting an array*

`*linebuffer` same as `linebuffer[0]`

`*(linebuffer+5)` same as `linebuffer[5]`

in general

`*(arrayname + expr)` same as `arrayname[expr]`

in fact, array indexing is considered to be a redundant concept in C and is provided only as a convenient notation

Array index or pointer?

Consider the following C code segments that initialize an array to integers using indices, or pointers:

```
void init_indices(int a[], int s) {  
    int i;  
    for(i = 0; i < s; i++)  
        a[i] = 1;  
}
```

```
void init_pointers(int *a, int s) {  
    int *p;  
    for(p = a; p < (a+s); p++)  
        *p = 1;  
}
```

Look in more depth!

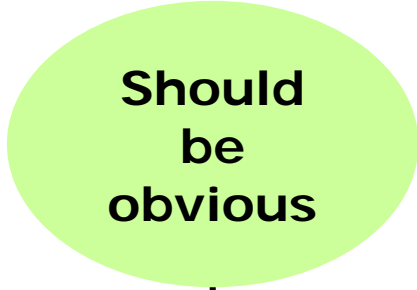
```
#define MAXLEN 10
int main()    {
    int myarray1[MAXLEN];
    int    ii, entry;
    int    *ip;
    entry=1;

    for(ii = 0; ii < MAXLEN; ii++){
        myarray1[ii]= entry++;
    }
    printf("\n hello friends, the array contains:\n");

    for(ii = 0; ii < MAXLEN; ii++){
        printf("%d ",myarray1[ii]);
    }

    exit(0);
}
```

Should
be
obvious



Look in more depth – repeat with pointers

```
/*Repeat using pointers */
```

```
entry=2;  
for(ip = myarray1; ip < (myarray1+MAXLEN); ip++){  
    *ip = entry++;  
}  
printf("\n\n hello friends, the array contains:\n");  
for(ip = myarray1; ip < (myarray1+MAXLEN); ip++){  
    printf("%d ", *ip);  
}
```

`ip` is assigned
the address of
`myarray1`,
i.e. address of
`myarray1[0]`

`ip` checked
against the
address of the
last element of
`myarray1`

`ip` here is
incremented to
the address of
the next element
(i.e. +4 bytes)

*the "+" here has the
semantics of adding the
correct number for bytes*

What is a subroutine?

- ❑ A subroutine is a portion of code to performs a specific task
- ❑ It can be relatively independent of the remaining code
- ❑ Also named : *function*, *procedure*, *method*
- ❑ All programming languages have support for subroutines, for *calls* to them and *returns* from them
- ❑ **Subroutine** can be seen as the more general label for functions, but can have *side-effects* outside of the simple "*return value*" of functions
- ❑ Some programming languages make very little syntactic distinction between functions and subroutines.

What is the difference between functions and procedures?

The ideal goal

FUNCTION

- has a single return value
- has no side effects on input parameters

PROCEDURE

- has no single return value
- side effects on parameters are allowed

C implements both and does not really make a distinction – simply gives the syntactic and semantics support for both choices and one can mix and match

Aim for the ideal above, justify logically to yourself why you absolutely need to break the paradigm

The practical side: implementation by examples in C

- ❑ all *subroutines* are regarded as "functions"
- ❑ **procedures** are functions which return "void"
- ❑ *all parameters are passed by value by default*
that is → no side effects
that is → a copy is sent to the function
- ❑ *parameters are passed by reference by using pointers*
that is → their contents can be changed
that is → their addresses is sent to the procedure

Subroutines: Functions and Procedures

```
/* function power */  
/* Given a and b it returns the result of  
   computing a to the power of b*/  
long power (int a, int b) {  
    long temp;  
    ... code for the computation of power ...  
    return temp;  
} /* end power */
```

```
/* procedure swap */  
/* Given x and y, it exchanges their  
   content*/  
void swap (int *x, int *y) {  
    int temp;  
    ... code for swap ...  
    return;  
} /*end procedure swap*/
```

Subroutines: Functions and Procedures

passed by value
- no changes

```
/* function power */  
/* Given a and b it returns the result of  
   computing a to the power of b*/  
long power (int a, int b) {  
    long temp;  
    ... code for the computation of power ...  
    return temp;  
} /* end power */
```

return value

passed by reference
- can be changed

```
/* procedure swap */  
/* Given x and y, it exchanges their  
   content*/  
void swap (int *x, int *y) {  
    int temp;  
    ... code for swap ...  
    return;  
} /*end procedure swap*/
```

no return value

A Function

```
long  x;  
int   r, t;  
.....  
x = power(r,t);  
.....
```

where :

- a copy of r and t is sent to the function
- only x changes
- r and t are the ACTUAL parameters
- a and b are the FORMAL parameters
- r will correspond to a and t to b
- a and b can be used and changed inside the function, yet the changes are not reflected outside to r and t

```
/* function power */  
/* Given a and b it returns the result of  
   computing a to the power of b*/  
long power(int a, int b) {  
    long temp;  
    ... code for the computation of power ...  
    return temp;  
} /* end power */
```

A Procedure

```
int  f, g;  
.....  
swap(&f,&g);  
.....
```

where :

- the addresses of f and g are sent to the procedure swap
- the content of f and g can change
- the addresses of f and g cannot change
- f and g are the ACTUAL parameters
- x and y are the FORMAL parameters
- f will correspond to x and g to y
- *x and *y are pointers
- &f and &g denote the addresses of f and g

```
/* procedure swap */  
/* Given x and y, it exchanges their  
content*/  
void swap(int *x, int *y) {  
    int temp;  
    ... code for swap ...  
    return;  
} /*end procedure swap*/
```


small examples

```
void function1(void) {  
    /* some code */  
}
```

```
int function2(void) {  
    /* some code */  
    return (some int)  
}
```

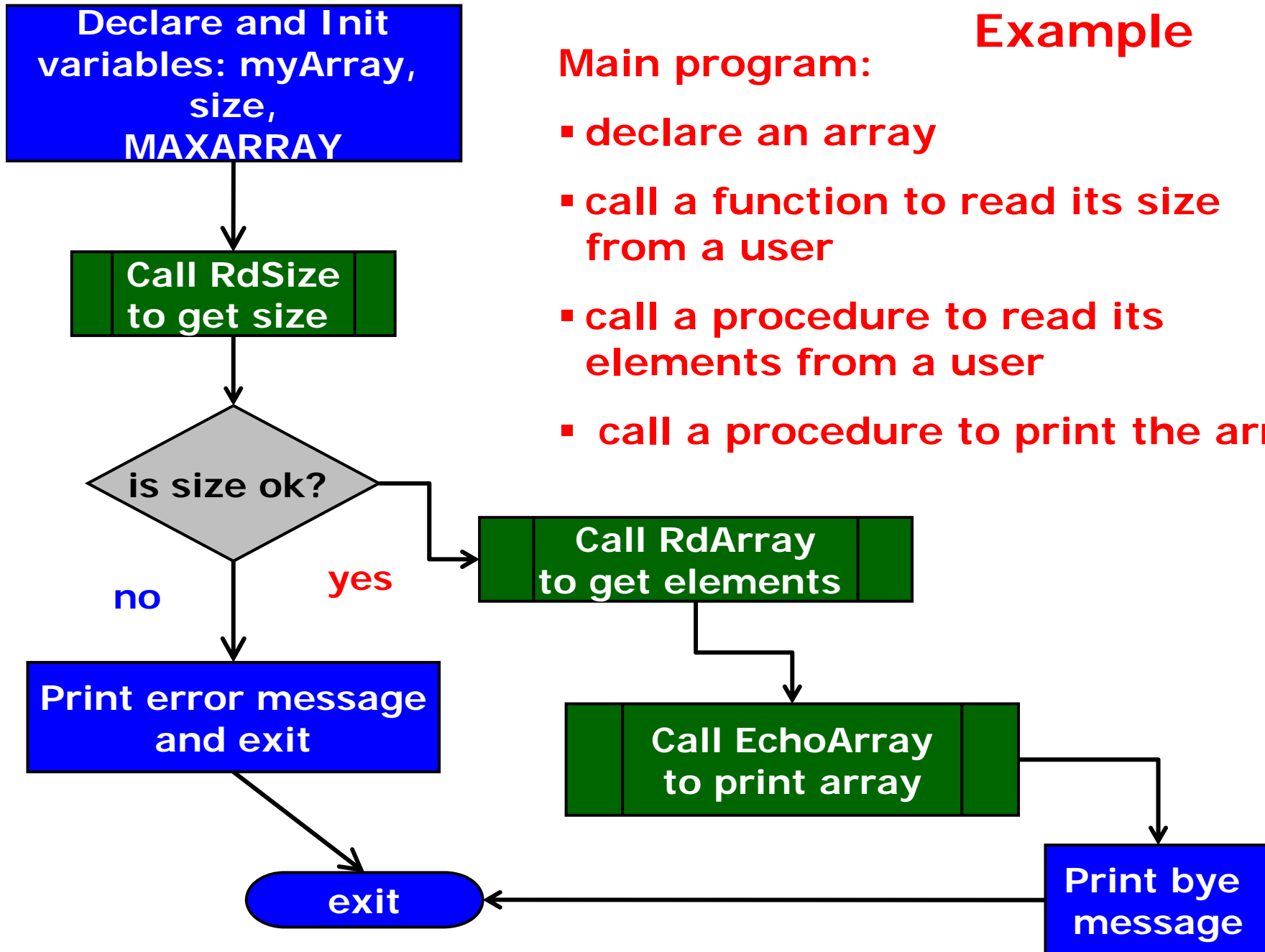
```
char function3(int somevar) {  
    /* some code */  
    return (somechar);  
}
```

```
void function4(int *somevar) {  
    /* some code */  
}
```

Example

Main program:

- declare an array
- call a function to read its size from a user
- call a procedure to read its elements from a user
- call a procedure to print the array



Part 1

```
#include <stdio.h>
#define MAXARRAY 100 /*limit on array size*/
int main() {
    int size;
    long myArray[MAXARRAY];
    size = RdSize1(); /*read size - use example 1*/
    if (size < 0) {
        printf("Size entered is incorrect\n");
        return 1; /* exit the program */
    }
    RdArray(myArray,size); /*read the elements*/
    echo_array(myArray, size); /* print the array */
    printf("All done!\n");
    return 0;
}
```

```

/** Function RdSize1 - example 1 *****/
/* To read from user the length of 1D array
   Returns size of array if okay, -1 if problems. */
int RdSize1() {
    char *prompt =
        "Enter a positive integer = size of array \n");
    int Length;

    printf(prompt); /* prompt user for size */
    scanf("%d", &Length); /*read size from user*/
    if (( Length > MAXARRAY) || (Length <= 0)) {
        return -1;          /*invalid size*/
    }
    else return(Length);
}

```

Part 1

Part 2

```
#include <stdio.h>
#define MAXARRAY 100 /*limit on array size*/
int main() {
    int size,okay;
    long myArray[MAXARRAY];
    okay = RdSize2(&size);/*read size - use example 2*/
    if (okay < 0) {
        printf("Size entered is incorrect\n");
        return 1; /* exit the program */
    }
    RdArray(myArray,size); /*read the elements*/
    echo_array(myArray, size); /* print the array */
    printf("All done!\n");
    return 0;
}
```

```

/** Function RdSize2 - example 2***/
/* To read from user the length of 1D array
   Returns size of array if okay, -1 if problems. */
int RdSize( int *Length) {
    char *prompt =
        "Enter a positive integer = size of array \n");
    printf(prompt); /* prompt user for size */
    scanf("%d", &Length);
    if (( Length > MAXARRAY) || (Length <= 0)) {
        return -1;          /*invalid size*/
    }
    else return(1);
}

```

Part 2

```

/** Function RdArray *****/
/* Function to read n elements into an array. */
void RdArray( long SomeArray[], int n ) {
    char *prompt =
        "Enter a positive integer = size of array \n");
    long temp;
    int ii;
    for( ii = 0; ii < n; ii++ ) {
        printf("enter the elements, one per line\n");
        scanf("%ld", &temp);
        SomeArray[ii] = temp;
    }
    printf("Thanks for the array elements\n");

} /* end RdArray */

```

Could you re-write this using pointers?

```
/* Function echo_array */  
/* Function to print the elements of an array,  
   given its address and its size */  
void echo_array( long SomeArray[], int size ) {  
    int i;  
    printf("The array elements are:\n");  
    for( i = 0; i < size; i++ ) {  
        printf("%ld\n", SomeArray[i]);  
    }  
} /* end echo_array */
```

Could you re-write this using
pointers?

Template: declarations and calling of functions

```
#include ...
```

```
#define ...
```

```
/*function foo*/  
int  foo(long a) {  
....code for foo....  
    return (...);  
}  /*end foo*/
```

```
/*procedure proc1*/  
void  proc1(int *b, int c) {  
...code for proc1....  
}  /*end proc1*/
```

```
int main( ) {  
.....  
    x = foo(y);      /*call to foo*/  
    proc1(&k,y);     /*call to proc1*/  
.....  
}  /*end main*/
```

Using pointers to pass by reference

```
int main( ) {  
    ....  
    int x, y, z;  
  
    x = 3; y = 3; z = 3;  
  
    foo(&x, y, z);  
    .....  
} /*end main*/
```

What is the value of x and y after procedure *foo* is executed?

x= 8
y= 3

```
void foo( int *a, int b, int c ) {  
    *a = *a + 5;  
    b = b + 5;  
    ...  
} /*end foo*/
```

Extra: How does "fscanf" work?

C library function which reads a number from a given file source into an integer

→ in `stdio.h`

`fscanf` terminates when:

- new line character is found
- end of file is found
- blanks or tab characters are read

Prototype:

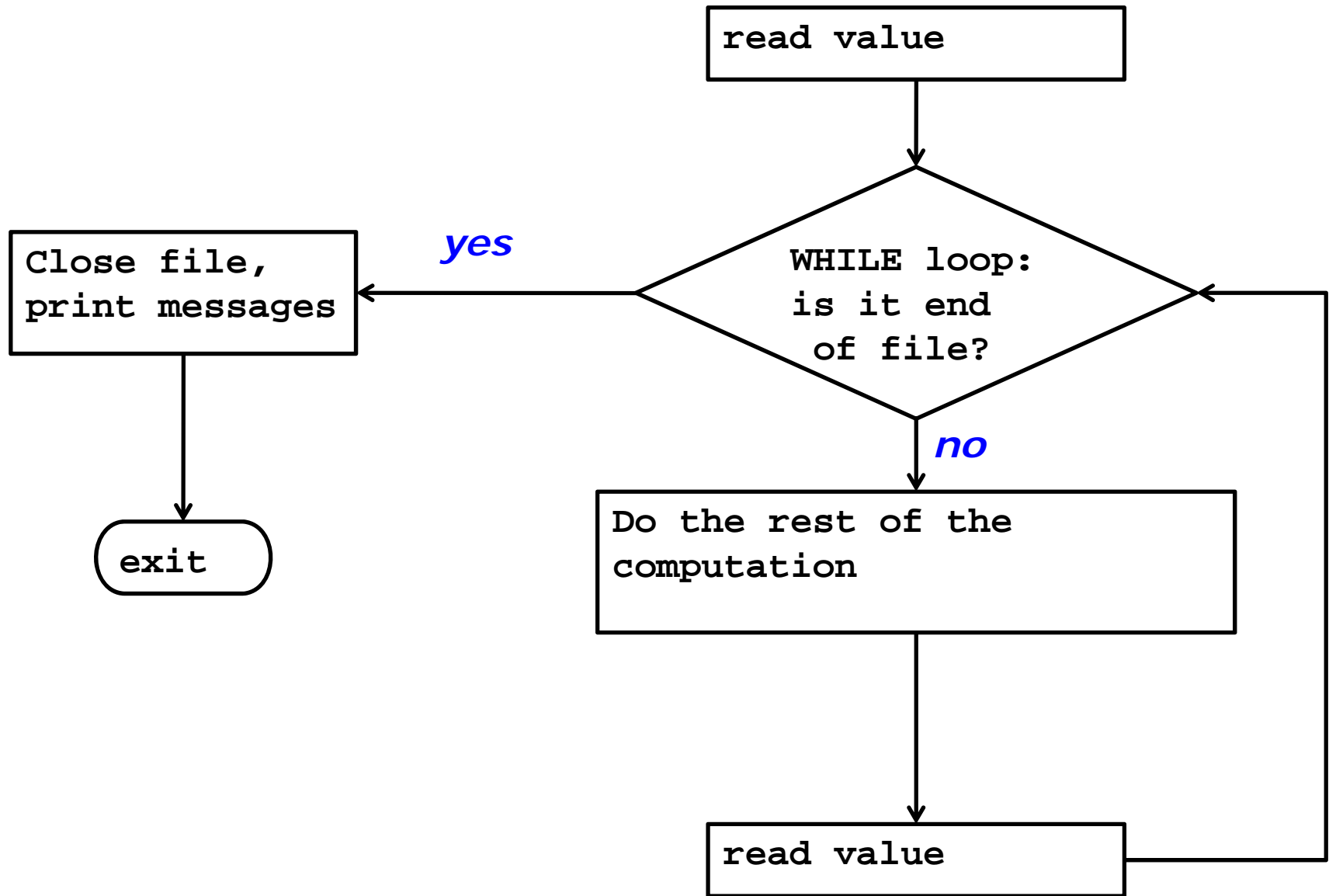
```
int fscanf (FILE *file, const char *format, ...);
```

Number of integers read → =1 if all okay

`stdin` or pointer to file as given by `fopen`

Characters read in and translated to integers

Read till end of file: STYLE 1



Read till end of file: STYLE 1

```
nir=fscanf(fpin,"%d",&RsizeM);/*read row size of a matrix */
```

```
while (nir == 1) {          /* while not end of file*/

    fscanf(fpin,"%d",&CsizeM);    /*read column size*/
    for (i=0;i<RsizeM;i++) {      /*read matrix*/
        for . . . . . (no room here)
    }
    /*print the matrix and the sizes*/
    fprintf(stdout, "\n\n*** MATRIX ***");
    fprintf(stdout, "  Size = %2d x %2d\n",RsizeM,CsizeM);
    for (. . . . . (no room here)
    }
    fprintf(stdout,"\n");

```

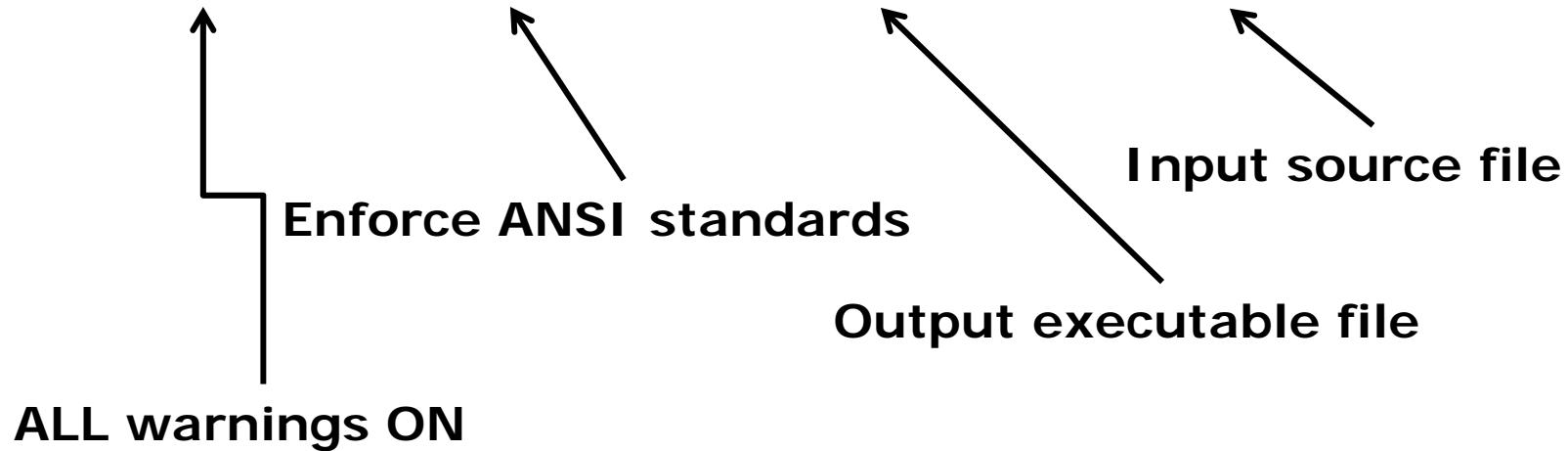
YOUR NEW CODE GOES HERE

```
nir=fscanf(fpin,"%d",&RsizeM); /*read next row size*/
```

```
}
```

The compiler and options

`gcc -Wall -ansi -o Example Example.c`



The compiler and options: consequences

There are a set of syntactic constructs not acceptable in ANSI, for example:

❑ Comments

```
/* comments here */  
/* are the only acceptable types */
```

```
Code here // comments here, not acceptable
```

❑ Declarations

```
/* acceptable */  
int i;  
for (i=0;i<100;i++) { do something}
```

```
/* not acceptable */  
for (int i=0;i<100;i++) { do something}
```