CS-231 – LESSON 03 – POINTERS AND STRUCTURES IN C

GIUSEPPE TURINI

Kettering UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides and your notes.
- *"The C Programming Language (2nd Edition)"*, ch. 5-6.

Homework:

- Code from scratch all C programs on these slides.
- Solve all exercises in *"The C Programming Language (2nd Edition)"*, ch. 5-6.

Additional Resources:

- *"The C Programming Language (2nd Edition)."*
- *"GNU - The GNU C Reference Manual"*
- *"Eclipse – C/C++ Development Toolkit (CDT) User Guide"*

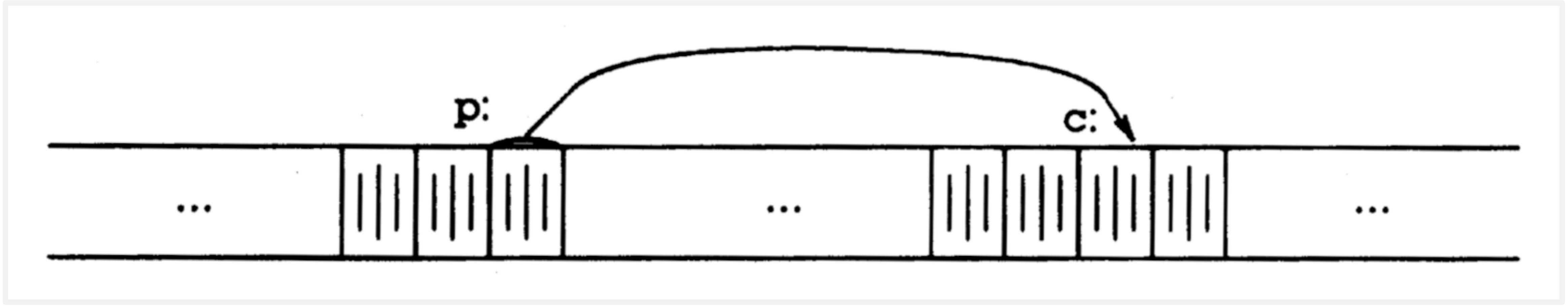
POINTERS AND ARRAYS IN C

A pointer is a variable that stores the memory address of another variable.

This is a simplified picture of how memory is organized.

A typical computing machine includes an *"array"* of consecutively numbered memory cells. Memory cells can be manipulated individually of in contiguous groups.
In this organization, a pointer is a group of memory cells that can store a memory address.

So, if *"c"* is a *"char"* and *"p"* is a pointer to "c" (*"char*"*), in memory the situation is this:



POINTER OPERATORS

The unary operator *“&”* returns the memory address of an *“object”*.
The *“&”* operator only applies to *“objects”* in memory: variables, and array elements.
The *“&”* operator cannot be applied to: expressions, and constants (and *“register”* variables).

The unary operator *“*”* performs the *“indirection”* or *“dereferencing”* operation.
The *“*”* operator applied to a pointer, accesses the pointed *“object”* (memory address pointed).

```c
// File "094A-PointerOperators.c"
int main() { // Testing basic pointer operators.
   int x =1; int y = 2; // Local variables (definitions).
   int z[10]; // z is an array of 10 int (declaration).
   int *intp; // intp is a pointer to int (declaration).
   intp = &x; // intp now points to x (definition).
   y = *intp; // y now stores the int value pointed by intp (x).
   *intp = 0; // The int variable pointed by intp is now assigned 0.
   intp = &z[1]; // intp now points to second int element in array of int z;
   *intp = y; // The int array item pointed by intp is now assigned the value of y.
   return 0; } // Normal exit.
```

POINTERS AND DATA TYPES

A pointer is constrained to point to a specific data type. There is no exception!

A pointer to *"void"* (*"void*"*) is used to store any type of pointer by cannot be dereferenced itself.

If *"xp"* points to *"x"*, then *"*xp"* can occur in any context where *"x"* could.

Pointers are variables, so they can also be used without dereferencing. For example: *"xp = yp;"*

POINTERS AND FUNCTION ARGUMENTS

In C, all function arguments are passed *"by value"* (the only exceptions are arrays).
So, that there is no direct way for the called function to alter a variable in the calling code.

To achieve this: We can pass pointers as arguments to functions, to allow the called function to alter the values pointed by those pointers.

Example: A swap function in C uses pointer arguments to swap values in the calling code.

```c
// File "096A-SwapFunction.c"
...
void swap(int *ap, int *bp) { // Swap function (definition).
   // Swapping values (a,b) pointed by input pointers (ap,bp).
   int temp = *ap; // temp stores value pointed by ap.
   *ap = *bp; // Value pointed by ap, now stores value pointed by bp.
   *bp = temp; // Value pointed by bp, now stores value stored in temp.
}
```

POINTERS AND ARRAYS

In C, there is a strong relationship between pointer and arrays.

Any operation done by array subscripting can also be done with pointers.
If Ap points to a particular array item, then Ap+1 points to the next array item, etc.
This is true regardless of the type (or size) of the array items.
In general, the pointer version will be faster than the array subscripting version.

The correspondence between array indexing and pointer arithmetic is very close.
By definition, an array variable stores the address of array item at index 0.

```
int A[10];
Ap = &A[0]; // Variables Ap and A store identical values.
Ap = A; // This assignment has the same meaning of the previous assignment.
int x = *(Ap+1); // Variable x now stores the value of array item at index 1.
int y = Ap[2]; // Variable y now stores the value of array item at index 2.
A = Ap; // ILLEGAL: Ap is a pointer variable, but A is an array name (NOT a variable).
A++; // ILLEGAL: Ap (as variable) can be incremented, but A (as an array name) cannot.
```

POINTERS AND ARRAYS (2)

Remember that there is a difference between an array name and a pointer.
A pointer is a variable, but an array name is **not** a variable.

When an array is passed to a function, the function can use it either as an array or as a pointer.
It is possible to pass a subarray to a function, by passing a pointer to the subarray first item.

If the item exists, you can use backward array indexing (A[-1]).
Negative array indices refer to items preceding item at index 0.

Note: Remember that is illegal to refer to *"objects"* that are not within the array bounds.

Note: Pointers are variables, so they can be stored in arrays (arrays of pointers).

POINTER ARITHMETIC

If p is a pointer to an array item, then p++ increments p to point to the next array item.
This is the simplest form of pointer arithmetic (memory address arithmetic).

In general, a pointer can be initialized just as any other variable.
However, usually the only meaningful initialization values are 0 or an address expression.
C guarantees that 0 is never a valid memory address for data.

Pointers and integers are not interchangeable.
Zero (0) is the sole exception (p=0; p==0; p!=0).
The symbolic constant *"NULL"* is often used instead of 0 (*"NULL"* is defined in *"<stdio.h>"*).

Pointers may be compared under certain circumstances.
If p and q point to items of the same array, then we can test if p<q, etc.
Comparisons have undefined behavior for pointers not pointing to items of the same array.

POINTER ARITHMETIC (2)

The **valid** pointer operations are:

- Assignment of pointers of the same data type.
- Adding/subtracting an integer to a pointer.
- Subtracting or comparing 2 pointers to items of the same array.
- Assigning 0/NULL to a pointer, or comparing a pointer to 0/NULL.
- All other pointer arithmetic is illegal!

The **illegal** pointer operations are:

- Adding 2 pointers.
- Multiply/divide/shift/mask pointers.
- Adding floating-point values to pointers.
- Assign a pointer of a type to a pointer of another type without a cast (except for "*void**").

POINTERS TO CHARACTERS

There is an important difference between these 2 definitions:

```
char messageArray[] = "now is the time"; // Array of characters (ending with '\0').
char *messagePointer = "now is the time"; // Pointer to character (string constant).
```

"messageArray" is an array of characters, just big enough to store those characters and the final '\0' that are used to initialize it. Individual characters in this array can be changed, but *"messageArray"* will always refer to the same storage.

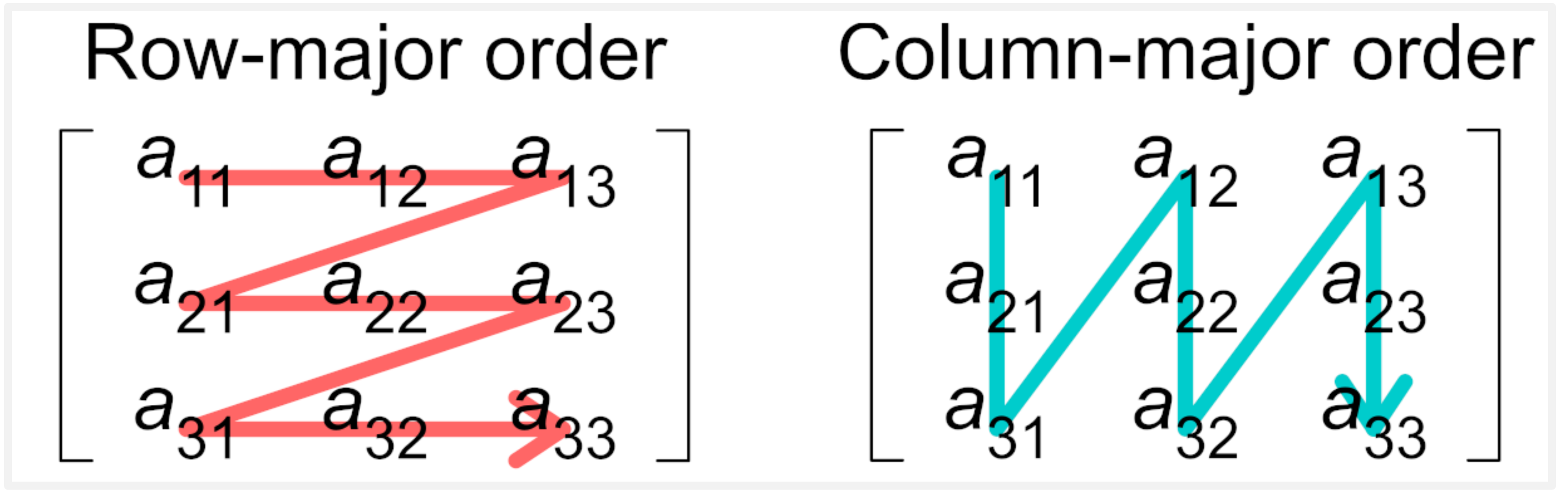
"messagePointer" is a pointer to a character, initialized to point to a string constant. This pointer can be modified to point elsewhere, but the result is undefined if you try to modify the string constant content.

MULTI-DIMENSIONAL ARRAYS

In C, a 2-dimensional array is really a 1-dimensional array, in which each item is a 1-dimensional array. The subscripts for a 2-dimensional array are written as:

```
int x = array2D[row][col]; // Subscripts for a 2D array in C.
```

In C, the items of a 2-dimensional array are stored by rows. So, the rightmost subscript (column) varies fastest as items are accessed in storage order (row-major order).


Row-major order
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

MULTI-DIMENSIONAL ARRAYS (2)

If a 2-dimensional array is passed to a function, the parameter declaration in the function must include only the number of columns (the number of rows is irrelevant). This is because what is passed is actually a pointer to an array of rows (a 2D array). So:

```
void f(int A[2][13]) { ... } // Both array sizes are specified.

void f(int A[][13]) { ... } // Only number of columns is specified.

void f(int (*A)[13]) { ... } // 2D array specified as a pointer to an array of 13 integers.
// Note: * operator must be in parentheses, because [] have higher precedence.
```

An array is initialized by a list of initializers in braces.
Each row of a multi-dimensional array is initialized by a corresponding sub-list.

```
int A[2][3] = { { 10, 20, 30 }, { 40, 50, 60 } }; // 2 sublists, each including 3 values.
```

MULTI-DIMENSIONAL ARRAYS (3)

Consider the difference between a 2-dimensional array and an array of pointers:

```
int A[10][20]; // A 2-dimensional array of ints.
int *B[10]; // A 1-dimensional array of pointers to ints.
```

Then, both A[3][4] and B[3][4] are syntactically lega references to a single int value.

But, A is a true 2-dimensional array (200 int-sized contiguous memory locations).

For B, however, we only have 10 memory locations to store pointers (not initialized yet).

Now, assume that each B item points to a 20-item array (then we have 200 ints in memory).

However, each B item could point to a differently sized array, in a different memory location.

POINTERS TO FUNCTIONS

In C, a function itself is not a variable. However, it is possible to define **pointers to functions**.

How to use pointers to functions is illustrated by coding a C program to *"sort run arguments with custom comparison function"* (for example: numerically or lexicographically).

```
// File "119A-SortRunArgumentsUsingPointersToFunctions.c"
#include <stdio.h> // For printf function etc.
#include <string.h> // For strcmp function etc.
#include <stdbool.h> // For bool macro.
#include <stdlib.h> // For atof function.

#define MAXLINES 5000 // Macro: max num of run arguments to be sorted.

void writeLines(char *lines[], int numLines);
void quickSort(char *lines[], int left, int right, int (*comp)(void *v1, void *v2));
void swap(char *v[], int i, int j);
int numcmp(char *s1, char *s2); // Numeric comparison function (to be passed as pointer).
...
```

POINTERS TO FUNCTIONS (2)

Program to *"sort run arguments with custom comparison function"* (continued).

```
...
char *linesPointers[MAXLINES]; // External var: array of pointers to strings to be sorted.

int main(int argc, char *argv[]) { // Sorting input run arguments.
   if(argc > 1) { // Check if run arguments are 2 or more (skip 1st run argument).
      // Transfer run arguments from argv into linesPointers (optiona).
      int numLines = 0;
      for(int i = 1; i < argc; i++) { linesPointers[i - 1] = argv[i]; numLines++; }
      // Sort linesPointers array using quicksort with custom comparison function.
      bool numericComparison = true; // Flag to select numeric/lexicographic comparison.
      if(numericComparison) {
         printf("Sorting run arguments using numeric comparison.\n"); fflush(stdout);
         quickSort(linesPointers, 0, numLines - 1, (int (*)(void*, void*)) numcmp); }
      else {
         printf("Sorting run arguments using lexicographic comparison.\n"); fflush(stdout);
         quickSort(linesPointers, 0, numLines - 1, (int (*)(void*, void*)) strcmp); }
      ...
```

POINTERS TO FUNCTIONS (3)

Program to *"sort run arguments with custom comparison function"* (continued).

```
      ...
      // Write content of sorted array linesPointers on console.
      writeLines(linesPointers, numLines);
      return 0; } // Normal exit.
   else { // Here, run arguments are less than 2, no text words to process...
      printf("ERROR: Nothing to process...\n"); fflush(stdout);
      return 1; } // Abnormal exit.
}

// Write input text lines on console (function definition).
void writeLines(char *linesPointers[], int numLines) {
   // Iteration to print text lines on console
   for(int i = 0; i < numLines; i++) { printf("%s\n", linesPointers[i]); }
   fflush(stdout);
}
...
```

POINTERS TO FUNCTIONS (4)

Program to *"sort run arguments with custom comparison function"* (continued).

```
...
// Sort (quicksort) input array of lines.
// Note: Recursive implementation, divide-by-2-and-conquer.
void quickSort(char *lines[], int left, int right, int (*comp)(void *v1, void *v2)) {
   if(left >= right) { return; } // If input array partition has <2 items, do nothing.
   else { // Here, input array partition has >2 items, quicksort.
      swap(lines, left, ((left + right) / 2));
      int last = left;
      for(int i = left + 1; i <= right; i++) {
         // Perform custom comparison of items, using pointer to function.
         if((*comp)(lines[i], lines[left]) < 0) { swap(lines, ++last, i); } }
      swap(lines, left, last);
      // Recursive calls, divide-by-2-and-conquer.
      quickSort(linesPointers, left, last - 1, comp);
      quickSort(linesPointers, last + 1, right, comp); }
}
...
```

POINTERS TO FUNCTIONS (5)

Program to *"sort run arguments with custom comparison function"* (continued).

```
...
// Swap v[i] with v[j].
void swap(char *v[], int i, int j) { char *temp = v[i]; v[i] = v[j]; v[j] = temp; }

// Compares input strings with a leading numeric value.
int numcmp(char *s1, char *s2) {
   // Converting strings to numbers.
   double v1 = atof(s1); // Converts an ASCII string into a floating-point value.
   double v2 = atof(s2); // Converts an ASCII string into a floating-point value.
   // Numeric comparison (3-way comparison).
   if(v1 < v2) { return -1; } // Return -1 if comparison is less-than.
   else if(v1 > v2) { return 1; } // Return 1 if comparison is greater-than.
   else { return 0; } // Return 0 if comparison is equal.
}
```

POINTERS TO FUNCTIONS (6)

The main elements of the program *"sort run arguments with custom comparison function"* are:

- The function names *"strcmp"* and *"numcmp"* are already pointers to functions in C.
  The & operator is unnecessary, function names (and array names) are already pointers.

- The sorting algorithm (function *"quickSort"*) is designed to process any data type.
  So the input pointer to function is a comparator taking 2 input pointers (*"void*"*).
  Any pointer can be cast to *"void*"* and back, without any loss of information.
  See the cast of the pointer to function in the calls to *"quickSort"*.

- The cast of the pointer to function in the calls to *"quickSort"*, needs parentheses!

```
(int (*comp)(void*, void*)) // Pointer to func taking 2 pointers and returning int.
(int *comp(void*, void*)) // Func taking 2 pointers and returning pointer to int.
```

COMPLICATED DECLARATIONS

In C, a declaration can be complicated, especially if it involves pointers to functions.
This is because declarations cannot be read left-to-right, and parentheses are over-used.
These are some examples of complicated declarations in C:

```
int *f(); // Function returning pointer to int.

int (*f)(); // Pointer to function returning int.

char **argv; // Pointer to pointer to char.

int (*d)[13]; // Pointer to an array of 13 items of type int.

int *d[13]; // Array of 13 items of type pointer to int.

char (*(*x())[])(); // Func returning pointer to array[] of pointers to func returning char.

char (*(*x[3])())[5]; // Array[3] of pointer to func returning pointer to array[5] of char.
```

STRUCTURES IN C

A structure is a collection of one or more variables, possibly of different types.

A structure groups together these variables under a single name for convenient handling.
Structures help organizing complicated data, so a group of variables can be treated as a unit.

Structures can be copied and assigned to, passed to functions, and returned by functions.
Automatic structures (local variables) and array of structures can also be initialized.

Note: Some languages call these groups of variables *"records"* (instead of *"structures"*).

BASICS OF STRUCTURES

The keyword *"struct"* introduces a structure declaration: a list of declarations in braces.

```c
// Structure declaration for variables representing a point in 2D.
struct point { // Note: Structure tag (point) is optional.
   float x; // X coordinate of the point in 2D.
   float y; // Y coordinate of the point in 2D.
};
```

The structure tag is an **optional** name that may follow the keyword *"struct"* (see *"point"*).
The structure tag names the structure and is a shorthand for the structure content.

The variables included in a structure are called members.
A structure member or a structure tag, and a variable can have the same name.

A structure declaration defines a type, and can be used as any other type (with tag or not).

BASIC OF STRUCTURES (2)

A structure declaration defines a type, and can be used as any other type (with tag or not).

```
// Structure declaration (no tag) followed by declaration of 2 variables of that type.
struct { float x; float y; } pA, pB;

struct point { float x; float y; }; // Structure declaration (with tag).
struct point pA, pB; // Variable declaration with structure type (need structure tag).
```

A structure can be initialized by following its definition with a list of initializers.
Structure initialization can also be done by using the structure member operator (.).

```
struct point pA = { 1.0, 2.0 }; // Structure initialization using list of initializers.

// Structure initialization using structure member operator (.).
struct point pA;
pA.x = 1.0;
pA.y = 2.0;
```

BASIC OF STRUCTURES (2)

A structure declaration defines a type, and can be used as any other type (with tag or not).

```c
// Structure declaration (no tag) followed by declaration of 2 variables of that type.
struct { float x; float y; } pA, pB;

struct point { float x; float y; }; // Structure declaration (with tag).
struct point pA, pB; // Variable declaration with structure type (need structure tag).
```

A structure can be initialized by following its definition with a list of initializers.
Structure initialization can also be done by using the structure member operator (.).

```c
struct point pA = { 1.0, 2.0 }; // Structure initialization using list of initializers.

// Structure initialization using structure member operator (.).
struct point pA;
pA.x = 1.0;
pA.y = 2.0;
```

BASIC OF STRUCTURES (3)

Structures can be nested. For example:

```
// Structure declaration for variables representing a point in 2D.
struct point { float x; float y; };

// Structure declaration for variables representing a rectangle in 2D.
struct rectangle {
   struct point corner1;
   struct point corner2;
   struct point corner3;
   struct point corner4;
};
```

Then, use the structure member operator (.) to access members of nested structures:

```
struct rectangle rA; // rA is a nested structure.
rA.corner1.x = 1.0; // Structure member operator (.) to access members of nested structure.
```

STRUCTURES AND FUNCTIONS

The only legal operations on a structure are:

- Copying it.
- Assigning to it as a unit.
- Taking its address using the pointer operator (&).
- Accessing its members using the structure member operator (.).
- Structures cannot be compared!

Copying and assigning include passing structures as arguments to functions, and using structures as return values from functions. There are at least 3 approaches:

- Pass structure members as arguments to a function, separately.
- Pass an entire structure as an argument to a function.
- Pass a pointer to a structure as an argument to a function.

POINTERS TO STRUCTURES

Pointers to structures are just like pointers to ordinary variables.

```
struct point { float x; float y; }; // Structure declaration.
struct point *pp; // pp is a pointer to a structure of type (struct point).
struct point p; // p is a structure of type (struct point).
pp = &p; // Assigning address of p to pointer pp.
float val = (*pp).x; // Parentheses needed because (.) has higher precedence than (*).
```

Pointers to structures are so frequently used that a shorthand is provided: (->).
If p points to a structure with m as its member, then: p->m is shorthand for (*p).

```
struct point { float x; float y; }; // Structure declaration.
struct point p; // p is a structure of type (struct point).
struct point *pp = &p; // pp is a pointer to a structure of type (struct point).
float valX = (*pp).x; // Parentheses needed because (.) has higher precedence than (*).
float valY = pp->y; // Shorthand operator (->) for pointers to structures.
```

OPERATOR SIZEOF

C provides a compile-time unary operator called *"sizeof"* to compute the byte-size.

These are examples of how to use this operator with either an *"object"* or a data type.

```
int a = 123; // Variable a of type int.
int byteSizeA = sizeof a ; // Using sizeof operator with a variable.
int byteSizeInt = sizeof( int ); // Using sizeof operator with a data type.
```

Strictly, *"sizeof"* returns an unsigned integer value of type *"size_t"* (defined in *"<stddef.h>"*).
An *"object"* can be a variable, an array, or a structure.
A data type can be a basic type (float), a structure type (struct point), or a pointer type (int*).

Do not assume that the size of a structure is the sum of the sizes of its members!
Alignment requirements for different objects could affect the structure size. For example:

```
struct mystruct { char c; int i; }; // Requires at least 5 (=1+4) bytes, maybe more...
int actualByteSize = sizeof( struct mystruct ); // Use sizeof to check the structure size.
```

ARRAYS OF STRUCTURES

Because a structure defines a new type, you can create arrays of structures. For example:

```
struct point { float x; float y; }; // Structure declaration.
struct point A[10]; // Array of 10 items of type (struct point).
struct point p = A[1]; // Copy second array item into variable p.
float valX = A[2].x; // Copy member x of third array item into variable valX.
```

As always, the byte-size of the array is its item-size times the byte-size of one of its items.

```
struct point { float x; float y; }; // Structure declaration.
struct point A[10]; // Array of 10 items of type (struct point).
int byteSizeA = 10 * sizeof( struct point ); // Using sizeof operator with data type.
struct point p; // Array of 10 items of type (struct point).
int byteSizeAItem = sizeof p; // Using sizeof operator with a variable.
```

SELF-REFERENTIAL STRUCTURES

It is illegal for a structure to contain an instance of itself!

```
// Definition of structure containing an instance of itself. Illegal!.
struct mylist { float item; struct mylist list; };
```

However, a structure is allowed to include a pointer to itself!

```
// Definition of structure containing a pointer to itself. Legal!.
struct mylist { float item; struct mylist *listp; };
```

USER-DEFINED DATA TYPE NAMES

In C, you can use the *"typedef"* statement to create new user-defined data type names.

```
typedef char *String; // Now String is a synonym for char*.
```

A data type name created in this way can be used exactly like any other data type name.

```
typedef char *String; // Now String is a synonym for char*.
String p; // Declaration of a variable of type String (char*).
p = (String) malloc(100); // Using String in a cast from void* to String (char*).
```

Note: Usually new data type names defined with *"typedef"* are **capitalized**.

USER-DEFINED DATA TYPE NAMES (2)

Besides purely aesthetic issues, there are 2 main reasons for using "*typedef*" statements:

- Parameterize a program against portability problems.

  You can use "*typedef*" statements for data types that may be machine-dependent, so that only the "*typedef*" statements need to change when a program is moved.

  For example: a common situation is to use "*typedef*" statements for various integer quantities, then select "*short*", "*int*", and "*long*" for each host machine.

  A type name like "size_t" is an example from the C standard library ("stdlib.h").

- Provide better documentation for a program.

  For example: a data type named "TreePointer" may be easier to understand.

UNIONS

A *"union"* is a variable that may store (at different times) *"objects"* of different types and sizes, with the C compiler keeping track of size and alignment requirements.

Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

The syntax to define a union is similar to the definition of a structure:

```
// Definition of a union.
union u_tag { int ival; float fval; char *sval; }; // Union can store 1 int/float/char*.
```

A variable of type union will be large enough to store the largest of their internal types (the specific size is implementation-dependent).

UNIONS (2)

Any type in a union may be assigned to a union variable, and then used in expressions.

Obviously, its usage must be consistent.

The type retrieved must be the type most recently stored.

Note: The programmer must keep track of which type is currently stored in a union variable. If an error occurs, the behavior is implementation-dependent.

Members of a union are accessed similarly to accessing structure members:

```
union u_tag { int ival; float fval; char *sval; }; // Definition of a union.
union u_tag u; // Variable u of type union u_tag.
union u_tag *up; // Variable up of type union u_tag *.
up = &u; // Initializing pointer to union.
int v1 = u.ival; // Using union member access operator (.).
int v2 = up->ival; // Using union member access operator (->).
```

UNIONS (3)

If effect, a union is a structure in which all members have offset 0 from the base in memory.

This structure (the union) is big enough to store the largest of its members.

Its alignment is appropriate for all of the types in the union.

The same operations are permitted on unions as on structures.

A union may only be initialized with a value of the type of its first member!

BIT-MASKS AND BIT-WISE OPERATORS

If saving memory is critical, we can to pack several *"objects"* into a single machine word.

A common example is packing a set of single-bit flags into a single variable.
A compact way to encode such information is a set of 1-bit flags in a single *"char"* or *"int"*.
This is usually done by defining a set of *"bit-masks"* corresponding to the relevant bit position.

A common way to implement *"bit-masks"* is by using macros (symbolic constants):

```
// Macros (symbolic constants) implementing bitmasks.
#define KEYWORD 01 // Bitmask for first bit (1 = 2^0).
#define EXTERNAL 02 // Bitmask for second bit (2 = 2^1).
#define STATIC 04 // Bitmask for third bit (4 = 2^2).
```

As an alternative, *"bit-masks"* can also be implemented using enumerations:

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 }; // Enumeration implementing bitmasks.
```

BIT-MASKS AND BIT-WISE OPERATORS (2)

Once a *“bit-mask”* has been created, its bits are manipulated by using bit-wise operators.

C provides 6 operators for bit manipulation:

- Bit-wise AND (&), binary operator.
- Bit-wise INCLUSIVE OR (|), binary operator.
- Bit-wise EXCLUSIVE OR (^), binary operator.
- LEFT SHIFT (<<), binary operator.
- RIGHT SHIFT (>>), binary operator.
- ONE’S COMPLEMENT (~), unary operator.

These operators only work on integer values (*“char”*, *“short”*, *“int”*, and *“long”*).

BIT-MASKS AND BIT-WISE OPERATORS (2)

The bit-wise AND operator (&) is often used to mask off some set of bits:

```
char n = n & 0177; // Set to 0 all but the low-order 7 bits of variable n.
// Note: 0177 in base 10, equals 01111111 in binary (base 2).
```

The bit-wise INCLUSIVE OR operator (|) is often used to turn on bits:

```
char n = n | 04; // Turn to 1 (on) low-order third bit of variable n.
// Note: 04 in base 10, equals 00000100 in binary (base 2).
```

The SHIFT operators shift their left operand by the number of bit positions given by their right operand (which must be positive):

```
char n = n << 2; // Shift left the value of n by 2 bit positions.
// Note: if n was 00000100 in binary, this shift updates n to be 00010000.
```

The unary operator (~) performs the ONE'S COMPLEMENT, inverting each bit:

```
char n = ~n; // Inverts bits of n (if n was 00000100 in binary, now n is 11111011).
```

BIT-FIELDS

As an alternative to bit-wise operators, in C you can define and access bit-fields directly.

A *"bit-field"* is a set of adjacent bits in a single storage unit that we will call a *"word"*.

The syntax of *"bit-field"* definition and access is based on structures.

```
// Definition of a structure storing a set of 3 1-bit fields.
struct flag_struct {
   unsigned int is_keyword : 1; // First bit-field, bit-size 1, type unsigned int.
   unsigned int is_extern : 1; // Second bit-field, bit-size 1, type unsigned int.
   unsigned int is_static : 1; // Third bit-field, bit-size 1, type unsigned int.
} flags; // Variable (flags) of type (struct flag_struct).
...
flags.is_extern = 1; // Turns on (1) the second bit-field in the variable (flags).
```

Note: Almost everything about bit-fields is implementation-dependent!

REFERENCES

- *“Wikipedia - C Programming Language”*

- *“GNU - The GNU C Reference Manual”*
- *“Microsoft - C Language Reference”*

- *“Nanyang Technological University – How To Install Eclipse for C/C++ Programming”*
- *“Eclipse – C/C++ Development Toolkit (CDT) User Guide”*

APPENDICES