

C Refresher

Team Emertxe



Chapter 1: Introduction



1.2 Language

- Simply put, a language is a stylized communication technique.
- The more varied vocabulary it has, the more expressive it becomes. Then, there is grammar to make meaningful communication statements out of it.
- Similarly, being more specific, a programming language is a stylized communication technique intended to be used for controlling the behavior of a machine (often a computer), by expressing ourselves to the machine.
- Like the natural languages, programming languages too, have syntactic rules (to form words) and semantic rules (to form sentences), used to define the meaning.

1.3 Programming Languages : Types

- Procedural
- Object Oriented
- Functional
- Logical
- And many more

1.4 Brief History

- Prior to C, most of the computer languages (such as Algol) were academic oriented, unrealistic and were generally defined by committees.
- Since such languages were designed having application domain in mind, they could not take the advantages of the underlying hardware and if done, were not portable or efficient under other systems.
- It was thought that a high-level language could never achieve the efficiency of assembly language.

Portable, efficient and easy to use language was a dream.

1.4 Brief History

- Prior to C, most of the computer languages (such as Algol) were academic oriented, unrealistic and were generally defined by committees.
- Since such languages were designed having application domain in mind, they could not take the advantages of the underlying hardware and if done, were not portable or efficient under other systems.
- It was thought that a high-level language could never achieve the efficiency of assembly language.
- C came as such a programming language for writing compilers and operating systems.

1.4 Brief History ...

- It was a revolutionary language and shook the computer world with its might. With just 32 keywords, C established itself in a very wide base of applications.
- It has lineage starting from CPL, (Combined Programming Language) a never implemented language.
- Martin Richards implemented BCPL as a modified version of CPL. Ken Thompson further refined BCPL to a language named as B.
- Later Dennis M. Ritchie added types to B and created a language, what we have as C, for rewriting the UNIX operating system.

1.4.1 The C Standard

- 'The C programming language' book served as a primary reference for C programmers and implementers alike for nearly a decade.
-
- However it didn't define C perfectly and there were many ambiguous parts in the language.
- As far as the library was concerned, only the C implementation in UNIX was close to the 'standard'.
- So many dialects existed for C and it was the time the language has to be standardized and it was done in 1989 with ANSI C standard.
- Nearly after a decade another standard, C9X, for C is available that provides many significant improvements over the previous 1989 ANSI C standard.

1.4.2 Important Characteristics

- C is considered as a middle level language.
- C can be considered as a pragmatic language.
- It is intended to be used by advanced programmers, for serious use, and not for novices and thus qualify less as an academic language for learning.
- Gives importance to compact code.
- It is widely available in various platforms from mainframes to palmtops and is known for its wide availability.

1.4.2 Important Characteristics(contd.)

- It is a general-purpose language, even though it is applied and used effectively in various specific domains.
- It is a free-formatted language (and not a strongly-typed language)
- Efficiency and portability are the important considerations.
- Library facilities play an important role

1.5 Keyword

- ✓ In programming, a keyword is a word that is reserved by a program because the word has a special meaning
- ✓ Keywords can be commands or parameters
- ✓ Every programming language has a set of keywords that cannot be used as variable names
- ✓ Keywords are sometimes called reserved names

1.5 C Keywords(contd...)

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- if
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while
- goto

Chapter 2: Basics Refresher



2.1 Data representation



Data representation

- ✓ Why bits?
 - Representing information as bits
 - Binary/Hexadecimal
 - Byte representations
- ✓ ANSI data storage
- ✓ Embedded specific data storage
- ✓ Floating point representation

Base 2



- ✓ Base 2 Number Representation
- ✓ Electronic Implementation
 - Easy to store
 - Reliably transmitted on noisy and inaccurate wires
 - Straightforward implementation of arithmetic functions

Byte Encoding

✓ Byte = 8 bits

- Binary: 00000000₂ -> 11111111₂
- Decimal: 010 -> 25510
- Hexadecimal: 0016 -> FF16

✓ Base 16 number representation

✓ Use characters '0' to '9' and 'A' to 'F'

✓ Write FA1D37B16 in C as 0xFA1D37B

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

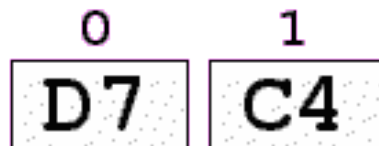
Byte ordering

- ✓ Big Endian: Sun's, Mac's are "Big Endian" machines
 - Least significant byte has highest address
- ✓ Little Endian: Alphas, PC's are "Little Endian" machines
 - Least significant byte has lowest address
- ✓ Example:

• Variable x has 2-byte representation D7C4
Storage of the value D7C4₁₆

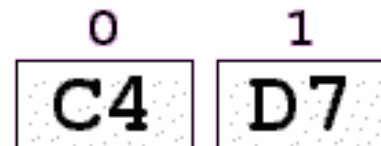
Big Endian

Motorola Processors:
68000, 68030, etc...



Little Endian

Intel Processors: 80386,
Pentium, etc...



ANSI data storage

- ✓ Integral Representation
 - char
 - short
 - int
 - long
 - long long
- ✓ Real Number Representation
 - float
 - double

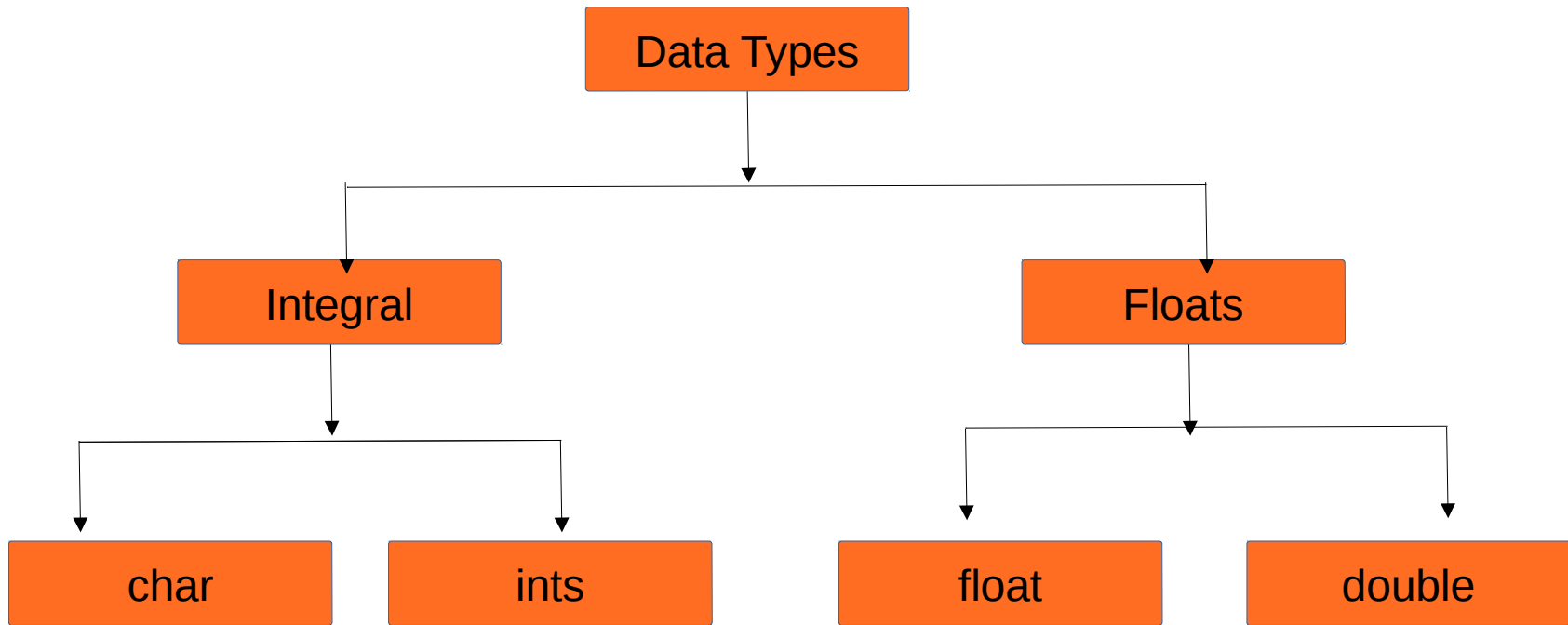
Embedded - Data storage

- ✓ Non-portability
- ✓ Solution: User typedefs
- ✓ Size utilizations
 - Compiler Dependency
 - Architecture Dependency
 - u8, s8, u16, s16, u32, s32, u64, s64
- ✓ Non-ANSI extensions
 - bit

2.2 Basic Data Types



2.2.1 Basic Data Types



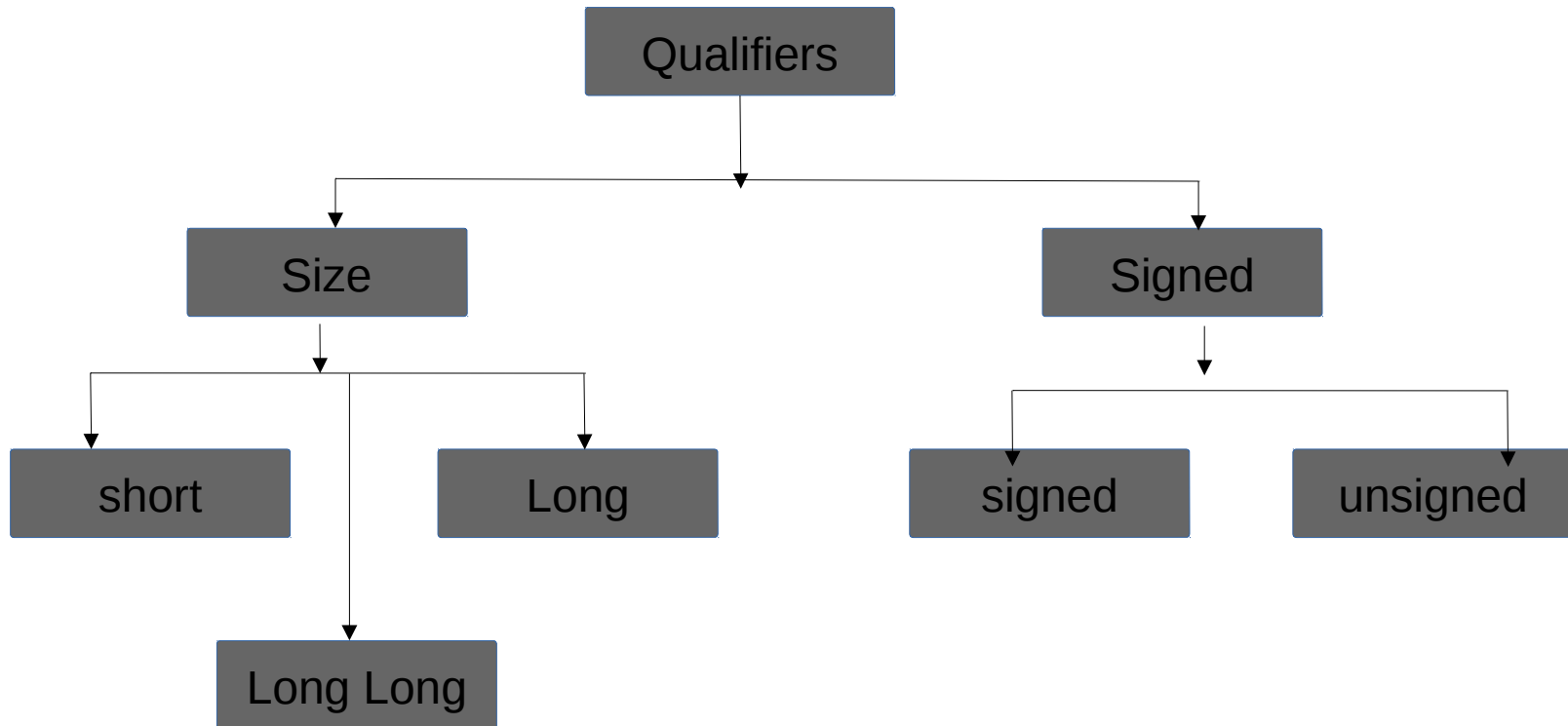
2.2.1 Sizes : Basic Data Types

- ✓ Type `int` is supposed to represent a machine's natural word size
- ✓ The size of character is always 1 byte
- ✓ $1 \text{ byte} = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$
- ✓ `float` = 4 bytes
- ✓ `double` = 8 bytes
- ✓ `pointer` = address word, mostly same as word
- ✓ `void` = 1 byte

2.3 Type, Variable & Function Qualifiers

A large, stylized arrow pointing to the right, composed of two overlapping shapes. The front shape is a dark purple chevron, and the back shape is a lighter purple chevron, creating a 3D effect. The arrow is positioned at the bottom of the slide, pointing towards the right edge.

2.3.1 Qualifiers



Const

- ✓ Variable cannot be changed
- ✓ Const used when function does not need to change a variable
- ✓ Attempting to change a const variable is a compiler error
- ✓ The addition of a 'const' qualifier indicates that the program may not modify the variable. Such variables may even be placed in read-only storage
- ✓ Since 'const' variables cannot change their value during runtime (at least not within the scope and context considered), they must be initialized at their point of definition

Example



- ✓ Example:

```
const int i = 5;
```

- ✓ An alternate form is also acceptable, since the order of type specifier and qualifiers does not matter:

```
int const i = 5;
```

- ✓ Order becomes important when composite types with pointers are used:

<pre>int * const cp = &i;</pre>	<pre>/* const pointer to int */</pre>
<pre>const int *ptci;</pre>	<pre>/* pointer to const int */</pre>
<pre>int const *ptci;</pre>	<pre>/* pointer to const int */</pre>

Volatile



- ✓ In embedded context its very common to read/write hardware registers, which might get changed by external influences
- ✓ Compiler may not aware of such possibilities and would optimize corresponding read/write. Volatile is exactly to stop that.
- ✓ Any object whose type includes the volatile type qualifier indicates that the object should not be subject to compiler optimizations altering references to, or modifications of, the object.

Example



- ✓ Example:

```
const int i = 5;
```

- ✓ An alternate form is also acceptable, since the order of type specifier and qualifiers does not matter:

```
int const i = 5;
```

- ✓ Order becomes important when composite types with pointers are used:

<pre>int * const cp = &i;</pre>	<pre>/* const pointer to int */</pre>
<pre>const int *ptci;</pre>	<pre>/* pointer to const int */</pre>
<pre>int const *ptci;</pre>	<pre>/* pointer to const int */</pre>

Register



- ✓ Register variables are a special case of automatic variables.
 - ✓ Automatic variables are allocated storage in the memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing in the CPU.
 - ✓ Registers – CPUs often have small amounts of storage within itself where data can be stored and accessed quickly
 - ✓ And hence, these would help better embedded performance. However, register is just a request to the compiler and may not be honored.
 - ✓ Be cautious in its usage!!!
-
- ✓ Example:
 - ✓ `register int x;`
 - `register char c;`

Inline



- ✓ During compilation, the C compiler automatically inlines small functions in order to reduce execution time (smart inlining).
- ✓ Inlining - The compiler inserts the function body at the place the function is called.
- ✓ With the inline keyword you force the compiler to inline the specified function
- ✓ Define inline functions in the same source module as in which you call the function
- ✓ Compiler only inlines a function in the module that contains the function definition.
- ✓ Example: `inline int max(int a, int b)`

Static



✓ Static variables

- Scope - The scope of variable is where the variable name can be seen
- Lifetime - The lifetime of a variable is the period over which it exists
- Scope of static variable is the file where it is defined
- Lifetime of static variable is lifetime of the program

Example



- ✓ Example:

```
const int i = 5;
```

- ✓ An alternate form is also acceptable, since the order of type specifier and qualifiers does not matter:

```
int const i = 5;
```

- ✓ Order becomes important when composite types with pointers are used:

<pre>int * const cp = &i;</pre>	<pre>/* const pointer to int */</pre>
<pre>const int *ptci;</pre>	<pre>/* pointer to const int */</pre>
<pre>int const *ptci;</pre>	<pre>/* pointer to const int */</pre>

Extern

- ✓ It is default storage class of all global variables
- ✓ When we use extern modifier with any variables it is only declaration
- ✓ Memory is not allocated for these variables

Example



- ✓ Example:

```
const int i = 5;
```

- ✓ An alternate form is also acceptable, since the order of type specifier and qualifiers does not matter:

```
int const i = 5;
```

- ✓ Order becomes important when composite types with pointers are used:

<pre>int * const cp = &i;</pre>	<pre>/* const pointer to int */</pre>
<pre>const int *ptci;</pre>	<pre>/* pointer to const int */</pre>
<pre>int const *ptci;</pre>	<pre>/* pointer to const int */</pre>

Operators



Bitwise operators

Bitwise Operators	Meaning
AND (&)	Logically AND all the bits in the two operands.
OR ()	Logically OR all the bits in the two operands.
XOR (^)	Logically XOR all the bits in the two operands.
Compliment (~)	Logically compliment all the bits in a operand.
Shift Left (<<)	Shift all the bits in the operand with n positions left and introduce zero from right.
Shift Right (>>)	Shift all the bits in the operand with n positions right and introduce zero from left.

How it works?

AND:

$$1001_2 \& 00112 = 00012$$

OR:

$$10012 \mid 00112 = 10112$$

Exclusive OR:

$$10012 \wedge 00112 = 10102$$

1's Complement:

$$\sim 000010102 = 111101012$$

Operator	Precedence
\sim	(H)
$\ll \gg$	\updownarrow
$\&$	
\wedge	
\mid	(L)


Bit wise operators are very powerful, extensively used in low level programming. This helps to deal with hardware (Ex: registers) efficiency.

Circuit logical operators



- ✓ These operators are similar to the & and | operators that are applied to Boolean type
- ✓ Have the ability to “short circuit” a calculation if the result is definitely known, this can improve efficiency
 - AND operator &&
 - If one operand is false, the result is false.
 - OR operator ||
 - If one operand is true, the result is true.

Hierarchy

Operators	Association	Precedence
() [] . -> ! ~ ++ -- + - * (Data Type) sizeof * / % + - << >> < <= > >= == != & ^ && ? : = += -= *= /= %= &= ^= = <<= >>= ,	Left to right. Right to left. Left to right. Left to right. Left to right. Left to right. Left to right. Left to right. Left to right. Left to right. Left to right. Left to right. Right to left Left to right.	(High)  (Low)

Promotion

long long double
⇕
long double
⇕
double
⇕
float
⇕
unsigned long long
⇕
(signed) long long
⇕
unsigned long
⇕
(signed) long
⇕
unsigned int
⇕
(signed) int
⇕
(unsigned short
⇕
(signed) short
⇕
unsigned char
⇕
signed char)

Hands-on!



Program Segments



Run time layout

- ✓ Run-time memory includes four (or more) segments
 - Text area: program text
 - Global data area: global & static variables
 - Allocated during whole run-time
- ✓ Stack: local variables & parameters
 - A stack entry for a functions
 - Allocated (pushed) - When entering a function
 - De-allocated (popped) - When the function returns
- ✓ Heap
 - Dynamic memory
 - Allocated by malloc()
 - De-allocated by free()

Details



- ✓ **Text Segment:** The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions.
- ✓ **Initialized Data Segment:** This segment contains global variables which are initialized by the programmer.
- ✓ **Uninitialized Data Segment:** Also named "BSS" (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute.

Details



- ✓ **The Stack:** The stack is a collection of stack frames which will be described in the next section. When a new frame needs to be added (as a result of a newly called function), the stack grows downward
- ✓ **The Heap:** Most dynamic memory, whether requested via C's malloc() . The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows upward

Hands-on!



Chapter 3: Functions



3.1 Why Functions?

- **Reusability**

- Once a function is defined, it can be used over and over again

- **Abstraction**

- In order to use a particular function you need to know the following things:
 - The name of the function;
 - What the function does;
 - What arguments you must give to the function; and
 - What kind of result the function returns.

3.2 Parameters, Arguments and Return



3.2 Parameters, Arguments and Return Values

- Parameters are also commonly referred to as arguments, though arguments are more properly thought of as the actual values or references assigned to the parameter variables when the subroutine is called at runtime.
- When discussing code that is calling into a subroutine, any values or references passed into the subroutine are the arguments, and the place in the code where these values or references are given is the parameter list.
- When discussing the code inside the subroutine definition, the variables in the subroutine's parameter list are the parameters, while the values of the parameters at runtime are the arguments.

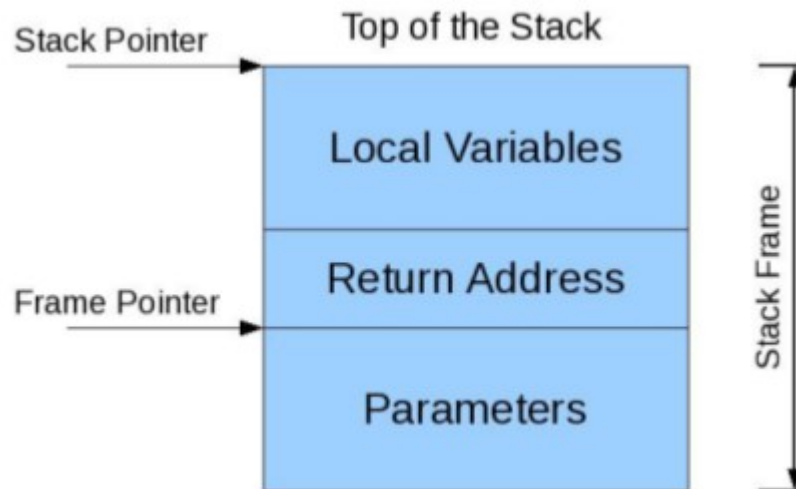
3.2 Parameters, Arguments and Return Values

▮ Example

```
1
2 //Function Declaration
3 int add(int a, int b);
4 int main()
5 {
6     int i, j, sum;
7     //Function call
8     sum = add(i, j);
9     printf("\nSum of %d and %d is %d\n", i, j, sum);
10    return 0;
11 }
12
13
14 //Function Definition
15 int add(int a, int b)
16 {
17     int val_to_return;
18     val_to_return = a + b;
19     return val_to_return;
20 }
```

3.2.1 Function and the Stack

Example



3.3 Procedures & Functions



3.3 Procedures & Functions

Procedure is a function which returns nothing, i.e. void

3.4 Various Passing Mechanisms



3.4 Various Passing Mechanisms

- Call by value
- Call by reference / address / location / variable /
- Copy-restore / Copy-in copy-out / Value-result
- Call by name

3.5 Ignoring Function's Return Value



3.5 Ignoring the function's return value

- In C tradition, you can choose to ignore to capture the return value from a method:

```
int i = get_an_integer();  
/*  
 * fetch the integer in i  
 * or  
 */  
get_an_integer();  
/* ignore the return value */
```

3.6 Returning the array from the function



3.6 Returning the array from the function

```
1 int *top2(int n, int array[])
2 int n, arr[10], *top;
3 int *top2(int n, int array[])
4 {
5     int biggest, second_biggest;
6     int big2[2];
7     /* Do calculation here */
8     big2[0] = biggest;
9     big2[1] = second_biggest;
10    return big2;
11
12 }
13 int main()
14 {
15     /* Read n & n elements in arr */
16     top = top2(n, arr);
17     printf("%d:%d\n", top[0], top[1]);
18     return 0;
19 }
```

3.7 Main & its argument

3.7 Main & its Arguments

`int main(void);`

`int main(int argc, char *argv[]);`

`int main(int argc, char *argv[], char *envp[]);`

```
1 #include <stdio.h>
2 int main(int argc, char *argv[], char *envp[])
3 {
4     int i;
5     char **env = envp;
6     printf("\n The count is %d:", argc);
7     for (i = 0; i < argc; i++)
8         printf("%s \n", argv[i]);
9     while (*env)
10         printf("%s \n", *env++);
11     return 0;
12 }
13 }
```

3.7.1 Three ways of taking input

- Through user interface
- Through command line arguments
- Through environment variables

3.8 Function type



3.8 Function type

- **The functions can be considered to be a type in C:**

- you can apply operators *, & to functions as if they are variables, and function definitions reserve a space (in code area of the program), function calls can participate in expressions as if they are variables and in that case, the type of the function is its return type.

- `/* function definition*/`
- `int foo() { return 0; }`

- `/* the function pointer can hold the function*/`
- `int (*fp)() = foo;`

- `/* using & for function is optional to take the address of the function*/`
- `fp = &foo;`

- `/*the value of i is 0; the type of the expression 10 * foo() is int*/`
- `int i = 10 * fp();` Through user interface

3.9 Variable argument functions



3.9 Variable Argument Functions

- The header: `#include <stdarg.h>`
- The type: `va_list ap;`
- The macros: `va_start(ap, last)` `va_arg(ap, type)` `va_end(ap)`

```
3 double calc_mean(int num, ...)
4 {
5     va_list ap;
6     double val;
7     int i;
8
9     va_start(ap, num);
10    val = 0;
11    for (i = 0; i < num; i++)
12    {
13        val += va_arg(ap, double);
14    }
15    va_end(ap);
16    return (val / num);
17 }
```

Chapter 4: Standard Input / Output



4.1: printf & scanf



4.1.1 The first parameter, format string

```
3 int main()
4 {
5     char *a = "Emertxe";
6     printf(a);
7 }
8
```

```
10 int main()
11 {
12     int a = 0;
13     char str[10];
14     scanf("%d%s", &a, str);
15     printf(str);
16     printf("%d%s", a, str);
17 }
```

4.1.2 Pointers with printf and scanf

Which of the following is a valid input parameter to scanf?

`&i, &a[i], a + i, &c, &str[i], str + i`
Is `&str \equiv &str[0] \equiv str`?

What's the output:

```
printf("%c", *("C is an Ocean" +5));  
printf("%c", *(&5["C is an Ocean"] -1));
```


4.1.3 Return values



Printf : Returns the number of characters printed.

Scanf : Returns the number of items successfully scanned.

4.1.4 Eating whitespaces by %d, %f, ...

scanf eats white spaces to be able to read any integer or real equivalent number.

4.1.4 Eating whitespaces by %d, %f, ...

scanf eats white spaces to be able to read any integer or real equivalent number.

4.3 Analyze the following loop

```
3 int main()
4 {
5     char ch;
6 }
7 scanf("%[^,]", &ch);
8 while (ch != 'e')
9 {
10     switch (ch)
11     {
12         case 'a':
13             break;
14         case 'b':
15             break;
16         default:
17             break;
18     }
19     printf("%c\n", ch);
20     scanf("%[^,]", &ch);
21 }
22 /*
23  * Input given is:
24  * a,b,c,d,e,f,g,h
25  */
```

4.3.1 Five reasons to flush the buffer

1. Buffer full
2. fflush
3. \n
4. Normal program termination
5. Read

4 Practice Set



4. Practice



1. Read the strings entered by the user and print the largest line among that.

1. Prerequisite

2. Objective

3. Algorithm Design

4. Dry run

5. Practical Implementation

Chapter 5: Files Input / Output



5.1 What is a file ?

File is a sequence of bytes.

5.2 Why files?



1. Persistent storage.
2. Theoretically unlimited size.
3. Flexibility of putting any data type into it.

5.3 The functions for the file operations

1. `fopen` - Open. `FILE *fopen(const char *path, const char *mode);`
2. `fwrite` - Write → Read Modify Write. `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
3. `fread` - Read. `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
4. `fclose` - Close. `int fclose(FILE *fp);`
5. `fseek` - Seek. `int fseek(FILE *stream, long offset, int whence);`
6. `feof` - End of File check. `int feof(FILE *stream);`

5.3.1 fgetc()

```
4 int main(int argc, char *argv[])
5 {
6     FILE *fp;
7     int ch;
8     if (argc == 1)
9     {
10         printf("Usage: %s <file_to_cat>\n");
11         return -1;
12     }
13
14
15     fp = fopen(argv[1], "r");
16     if (fp == NULL)
17     {
18         perror("fopen");
19         return -1;
20     }
21     while ((ch = fgetc(fp)) != EOF)
22     {
23         fputc(ch, stdout);
24     }
25     return 0;
26 }
```

5.3.2 Modes the file can be opened

r: Open text file for reading.

The stream is positioned at the beginning of the file.

r+: Open for reading and writing.

The stream is positioned at the beginning of the file.

w: Truncate file to zero length or create text file for writing.

The stream is positioned at the beginning of the file.

w+: Open for reading and writing.

The file is created if it does not exist, otherwise it is truncated.

The stream is positioned at the beginning of the file.

a: Open for appending (writing at end of file).

The file is created if it does not exist.

The stream is positioned at the end of the file.

a+: Open for reading and appending (writing at end of file).

The file is created if it does not exist.

The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

5.4 Practice



Practice - 1

Problem : Reverse a file

1. Prerequisite
2. Objective
3. Algorithm design
4. Dry run
5. Practical Implementation

Chapter 5 & 6: Pointers



6.1 : Strings



6.1 Strings

- C strings are low-level in nature (and bug-prone).
- C does not support strings as a data type - it is just an array of characters.
- But it has library support for strings (as in strstr, strcmp etc.) in literals.
- This leads to the close relationship between arrays and strings in C.

Output ?

```
6 if(sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
7     printf("WoW\n");
8 else
9     printf("Huh\n");
```

6.2 Initializing the Strings

Various Ways of Initialization

```
4 char a[5] = "Hello", b[] = "Hi", c[5] = "Hi";  
5 char str1[] = "Hi", str2[] = {'H', 'i'};  
6 printf("%s:%s:%s", a, b, c);  
7 printf("%s:%s", str1, str2);
```

6.3 Sizes of

```
4 char *str1 = "Hi";  
5 char str2[] = "Hi";  
6 printf("%d", sizeof(str1));  
7 printf("%d", sizeof(str2));
```

6.4 String Manipulations

- Strings are character arrays and
- an array name refers to an address - a pointer constant.
- So, there is a close relationship between arrays, string and pointers.
- Exploiting this makes string manipulation a very efficient one.

Example

```
5 int my_strlen(const char *s)
6 {
7     char *t = s;
8     while (*t++)
9         ;
10    return t-s-1;
11 }
```

Other Way

```
4 int my_strlen(const char *s)
5 {
6     int i = 0;
7     while (s[i] != '\0')
8     {
9         i++;
10    }
11    return i - 1;
12 }
```

DIY : Implement strcpy() function

6.4.1 Inconsistencies & Problems

Problem 1

```
5 char s1[10] = "string";
6 char s2[10];
7 s2 = "string";
8 /* error! only initialization is possible and not the */
9 /* assignment - because array assignment is not possible */
```

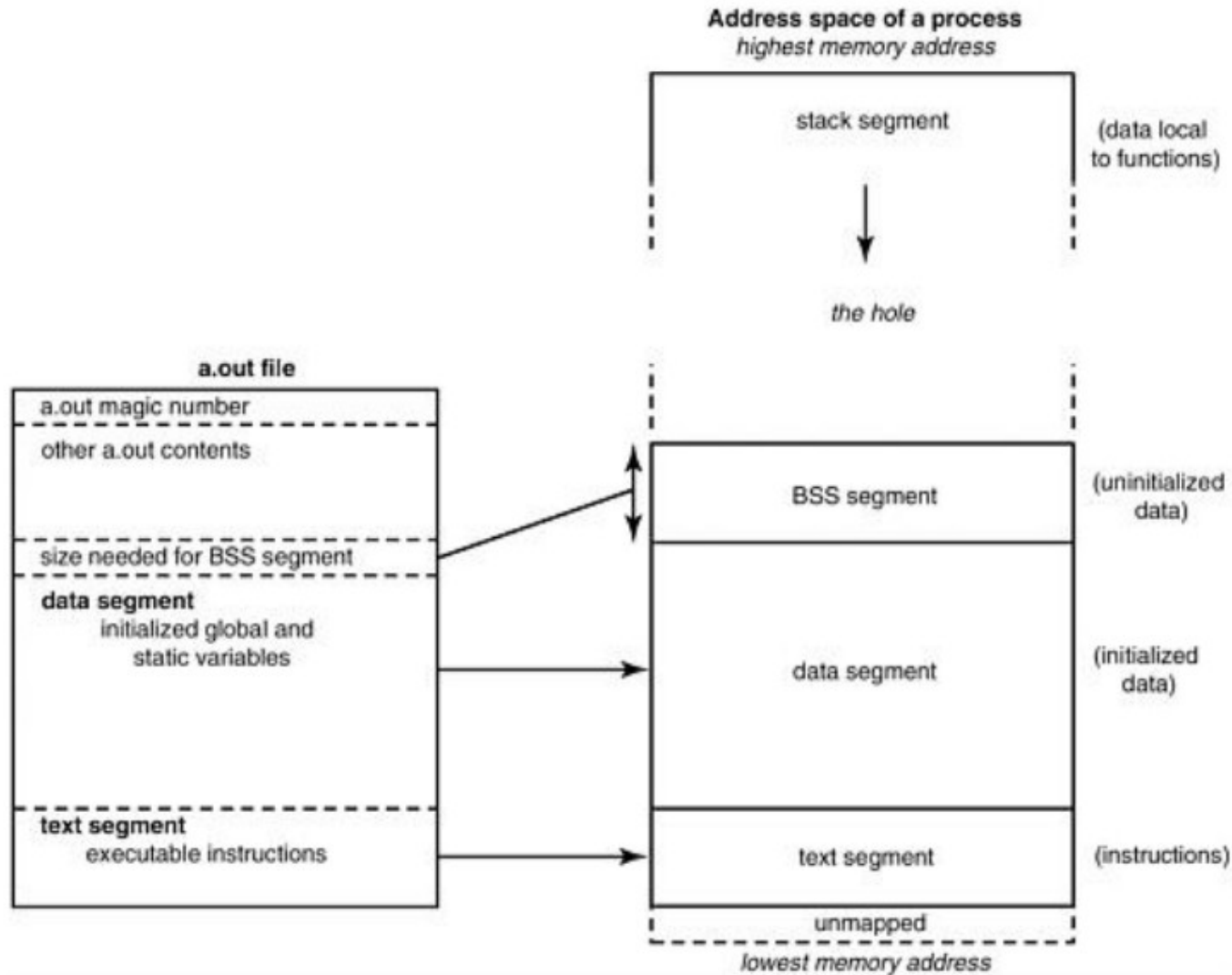
Problem 2

```
3
4 char *s3 = "string";
5 char *s4;
6 s4 = "string";
7 /* s3 is a pointer, so both initialization */
8 /*and assignment are possible */
9
10 s1[0] = 'S';
11 /* O.K. s1 now contains the string "String" */
12
13 s3[0] = 'S';
14 /* It points to a string literal - it is read only */
15 /* So, undefined behavior */
16 /* assume that sizeof pointer is 4 bytes */
17 printf(" %d %d %d ", sizeof(s1), sizeof(s3), sizeof("string"));
18 /* sizeof the array, pointer and string literal respectively.*/
19
20 printf(" %d %d %d ", strlen(s1), strlen(s3), strlen("string"));
```

6.4.1 Inconsistencies & Problems

- ✓ In C, the strings are always NULL terminated. This is often convenient but also bug prone.
- ✓ Consider the example of `strcpy()` - inside the function there is no way to check neither the argument is properly null terminated nor the target is capable enough to hold the source.
- ✓ In both cases, it leads to undefined behavior because it reads/writes past the array bounds.
- ✓ The serious disadvantage of this representation is to know the length of the string, traversal has to be made until the end of the string.
- ✓ Except such inconveniences, the C string implementation works out better in terms of efficiency and ease of implementation.

6.5 Program Segments



6.4.1 Inconsistencies & Problems

- ✓ In C, the strings are always NULL terminated. This is often convenient but also bug prone.
- ✓ Consider the example of `strcpy()` - inside the function there is no way to check neither the argument is properly null terminated nor the target is capable enough to hold the source.
- ✓ In both cases, it leads to undefined behavior because it reads/writes past the array bounds.
- ✓ The serious disadvantage of this representation is to know the length of the string, traversal has to be made until the end of the string.
- ✓ Except such inconveniences, the C string implementation works out better in terms of efficiency and ease of implementation.

6.5.1 Shared Strings

- In C, string constants (string literals) are shared as they cannot be modified.
- Consider the following code:

```
1
5 char *s1 = "string";
6 char *s2 = "string";
7 s1[0] = 'S';
8 printf("%s", s2);
```

- ✓ Since string literals are immutable in C, the compiler is free to store both the string literals in a single location and so can assign the same to both s1 and s2.
- ✓ Such sharing of string is an optimization technique done by the compiler to save space.

6.5.1 Shared Strings

It is easy to check if the two string literals are shared or not in your compiler/platform.

Consider the following code:

```
3 if(s1 == s2)
4 {
5     printf("Yes. shared strings");
6 }
7 /* or check it directly with the string constants */
8 if("string" == "string")
9 {
10     printf("Yes. shared strings");
11 }
```

Think on the common sense, that no two different string constants can be stored in memory.

6.6 Why pointers?

- ✓ To have C as a low level language being a high level language.
- ✓ To have the dynamic allocation mechanism.
- ✓ To achieve the similar results as of "pass by variable" parameter passing mechanism in function, by passing the reference.
- ✓ Returning more than one value in a function.

6.7 : Pointers & Seven rules



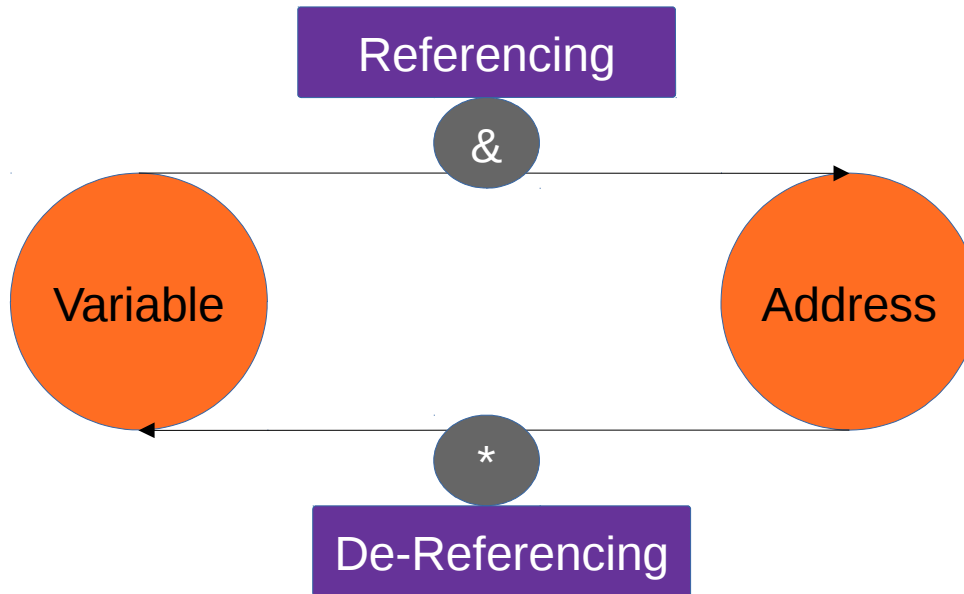


Rule #1:

Pointer as a integer variable

Rule #2:

Referencing & Dereferencing



Rule #3:

Type of a pointer

- Pointer of type $t \equiv t \text{ Pointer} \equiv (t *) \equiv$ A variable which contains an address, which when dereferenced becomes a variable of type t

- All pointers are of same size
- Pointers are defined indirectly

Rule #4:

Value of a pointer

Pointing means Containing,

i.e.,

Pointer pointing to a variable \equiv Pointer contains the address of the variable

Rule #5:

NULL pointer

- Pointer Value of zero \equiv Null Addr \equiv NULL pointer \equiv Pointing to nothing

Segmentation fault

```
3 int main()
4 {
5     int a[5], i;
6     printf("Enter even numbers\n");
7     for(i = 100; i <= 1000; i++)
8         scanf("%d", &a[i]);
9     printf("Entered even nos. are\n");
10    for(i = 1; i <= 1000; i++)
11        printf("%d", a[i]);
12    return 0;
13 }
```

Bus Error

```
3 int main()
4 {
5     char a[sizeof(int) + 1];
6     int *x, *y;
7     x = &a[0];
8     y = &a[1];
9     scanf("%d%d", x, y);
10 }
11
```

Array Interpretations



Two Interpretations:

- Original Variable
- Constant Pointer (Compiler only)

Rule: When (to interpret array variable as) what?

First variable, then pointer, then warning, then error

Rule #6:

Arithmetic Operations with Pointers & Arrays

- $\text{value}(p + i) \equiv \text{value}(p) + \text{value}(i) * \text{sizeof}(*p)$
- **Array → Collection of variables vs Constant pointer variable**
- `short sa[10];`
- `&sa` → Address of the array variable
- `sa[0]` → First element
- `&sa[0]` → Address of the first array element
- `sa` → Constant pointer variable
- **Arrays vs Pointers**
- Commutative use
- $(a + i) \equiv i + a \equiv \&a[i] \equiv \&i[a]$
- $*(a + i) \equiv *(i + a) \equiv a[i] \equiv i[a]$
- constant vs variable

Rule #7:

Static & Dynamic Allocation

- Static Allocation \equiv Named Allocation -
- Compiler's responsibility to manage it - Done internally by compiler, when variables are defined
- Dynamic Allocation \equiv Unnamed Allocation -
- User's responsibility to manage it - Done using malloc & free
- **Differences at program segment level**
- Defining variables (data & stack segment) vs Getting & giving it from the heap segment using malloc & free
- ```
int x, int *xp, *ip;
```
- ```
xp = &x;
```
- ```
ip = (int*)(malloc(sizeof(int)));
```



# Dynamic Memory Allocation



# Dynamic Memory Allocation

- In C functions for dynamic memory allocation functions are declared in the header file `<stdlib.h>`.
- In some implementations, it might also be provided in `<alloc.h>` or `<malloc.h>`.

- **malloc**
- **calloc**
- **realloc**
- **free**

# Malloc

- The malloc function allocates a memory block of size size from dynamic memory and returns pointer to that block if free space is available, otherwise it returns a null pointer.

- **Prototype**

- **`void *malloc(size_t size);`**

# Calloc

- The calloc function returns the memory (all initialized to zero)
- so may be handy to you if you want to make sure that the memory is properly initialized.
- calloc can be considered as to be internally implemented using malloc (for allocating the memory dynamically) and later initialize the memory block (with the function, say, memset()) to initialize it to zero.

## Prototype

▫ **void \*calloc(size\_t n, size\_t size);**

# Realloc



## **The function realloc has the following capabilities**

- 1. To allocate some memory (if p is null, and size is non-zero, then it is same as malloc(size)),
- 2. To extend the size of an existing dynamically allocated block (if size is bigger than the existing size of the block pointed by p),
- 3. To shrink the size of an existing dynamically allocated block (if size is smaller than the existing size of the block pointed by p),
- 4. To release memory (if size is 0 and p is not NULL then it acts like free(p)).

## **Prototype**

**void \*realloc(void \*ptr, size\_t size);**

# free

- The free function assumes that the argument given is a pointer to the memory that is to be freed and performs no check to verify that memory has already been allocated.
- 1. if free() is called on a null pointer, nothing happens.
- 2. if free() is called on pointer pointing to block other than the one allocated by dynamic allocation, it will lead to undefined behavior.
- 3. if free() is called with invalid argument that may collapse the memory management mechanism.
- 4. if free() is not called on the dynamically allocated memory block after its use, it will lead to memory leaks.

## Prototype

▫ **void free(void \*ptr);**

# Differences between a pointer and an array



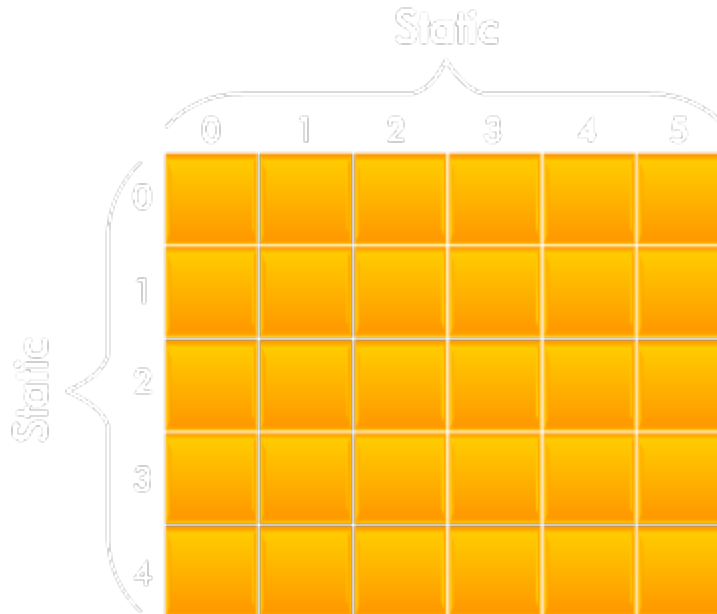
- Variable pointer vs Constant pointer
- sizeof a pointer and an array
- Initialization to point to correct location vs Correctly pointing
- `char *str = "str";` vs `char str[] = "str";`

# 2D Arrays

- ✓ Each Dimension could be static or Dynamic
- ✓ Various combinations for 2-D Arrays ( $2 \times 2 = 4$ )
  - C1: Both Static (Rectangular)
  - C2: First Static, Second Dynamic
  - C3: First Dynamic, Second Static
  - C4: Both Dynamic
- ✓ 2-D Arrays using a Single Level Pointer

# C1: Both static

- ✓ Rectangular array
- ✓ `int rec [5][6];`
- ✓ Takes totally  $5 * 6 * \text{sizeof}(\text{int})$  bytes





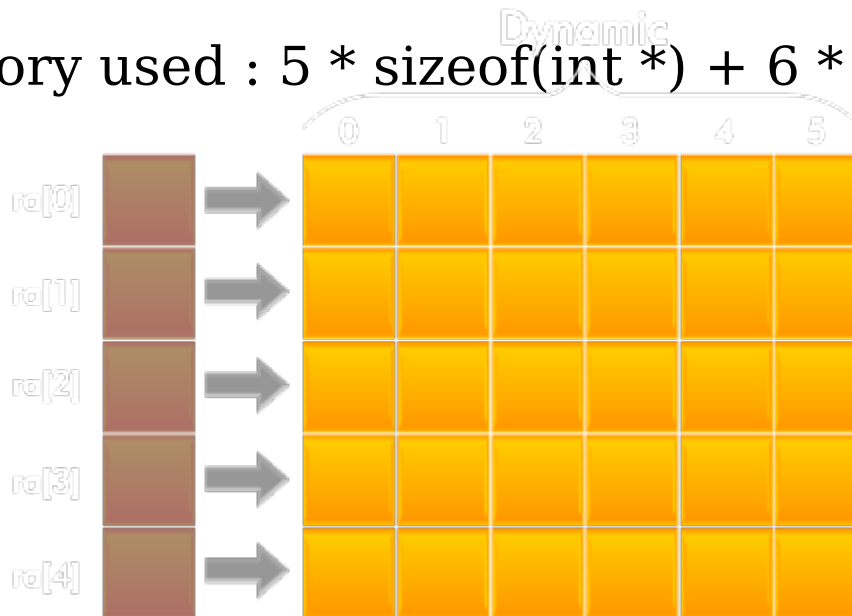
## C2: First static, Second dynamic

- ✓ One dimension static, one dynamic (Mix of Rectangular & Ragged)

```
int *ra[5];
for(i = 0; i < 5; i++)
```

```
 ra[i] = (int*) malloc(6 *
sizeof(int));
```

- ✓ Total memory used :  $5 * \text{sizeof(int *)} + 6 * 5 * \text{sizeof(int)}$  bytes



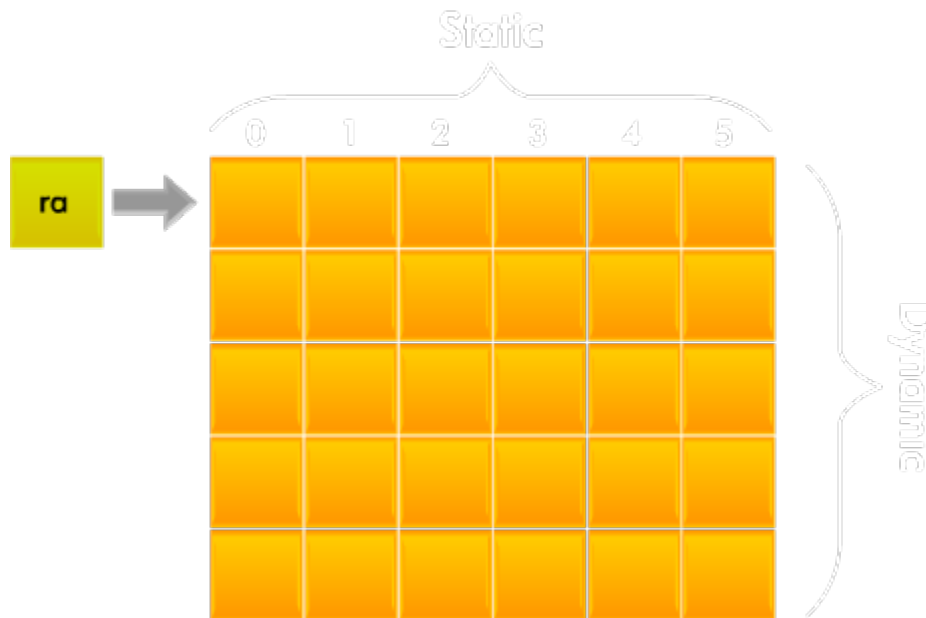
## C3: Second static, First dynamic

✓ One static, One dynamic

`int (*ra)[6];` (Pointer to array of 6 integer)

`ra = (int(*)[6]) malloc( 5 * sizeof(int[6]));`

✓ Total memory used :  $\text{sizeof(int *)} + 6 * 5 * \text{sizeof(int)}$  bytes



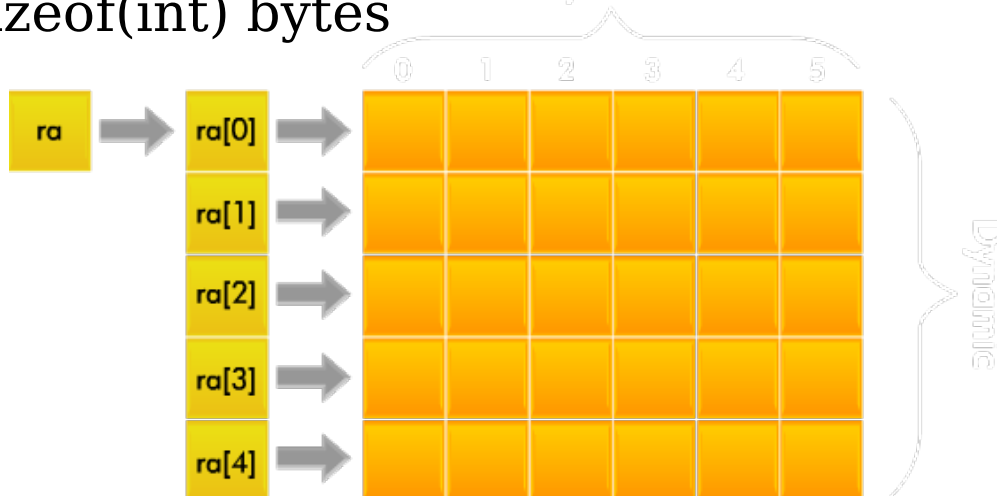
## C4: Both dynamic

### ✓ Ragged array

```
int **ra;
ra = (int **) malloc (5 * sizeof(int*));
for(i = 0; i < 5; i++)
 ra[i] = (int*) malloc(6 * sizeof(int));
```

✓ Takes  $5 * \text{sizeof(int*)}$  for first level of indirection

✓ Total memory used :  $1 * \text{sizeof(int **)}$  +  $5 * \text{sizeof(int *)}$  +  $5 * 6 * \text{sizeof(int)}$  bytes



## 6.9 Function Pointers



## 6.9.1 Why Function pointers



the solution to following requirements:

and independently and is standalone

used to iterate over a collection of

essentially asynchronous where there may be several objects that may be interes

and calling it later when required.



## 6.9.2 Function Name

### - The Second Interpretation

- C provides function pointers that are pretty low-level, efficient and
- direct way of providing support to callback functions.

Notes:

Only Rule #1 to Rule #4 are applicable for function pointers

## 6.9.3 Examples

```
void *bsearch(void *key, void *base, size_t num, size_t width,
int (*compare)(void *elem1, void *elem2));
```

```
4 int atexit(int (*)(void));
5 //You can register the functions to be called when the program exits.
6 //Consider:
7 int my_function ()
8 {
9 printf("Exiting the program \\n");
10 return 0;
11 }
12 int main()
13 {
14 printf("Inside main\\n");
15 atexit(my_function);
16 printf("About to quit\\n");
17 }
```

## 6.10 Practice





# Practice

- Write a function `read_int` to read an integer

# Chapter 7: Preprocessor



## 7.1 : What is preprocessing



# 7.1 What is preprocessing & When is it done?

- Preprocessor is a powerful tool with raw power.
- Preprocessor is often provided as a separate tool with the C compilers.
- After preprocessing the preprocessed output is sent to the C compiler for compilation.

# 7.1.1 Functionalities of Preprocessor

- The main functionalities of the preprocessor are:
- file inclusion (`#include`)
- Conditional compilation (`#ifdefs`)
- Textual replacement (`#defines`)
- Preprocessor is also responsible for removing the comments from source code (`//` and `/* */`)
- processing the line continuation character (`\` character)
- processing escape sequences (characters such as `\n`)
- processing of the trigraph sequences (such as `'??<'` character for `{`) etc

## **7.2 : Built-in Defines**



## 7.2 Built-in Defines

|                       |                                                                                      |
|-----------------------|--------------------------------------------------------------------------------------|
| <code>__FILE__</code> | Represents the current source file name in which it appears.                         |
| <code>__LINE__</code> | Represents the current line number during the preprocessing in the source file.      |
| <code>__DATE__</code> | Represents the current date during the preprocessing.                                |
| <code>__TIME__</code> | Represents the current time at that point of preprocessing.                          |
| <code>__STDC__</code> | This constant is defined to be true if the compiler conforms to ANSI/ISO C standard. |
| <code>__func__</code> | Represents the current function name in which it appears.                            |

Example    `printf("Error in %s @ %d on %s @ %s \n",  
          __FILE__, __LINE__, __DATE__, __TIME__);`

## **7.3 : The Preprocessor Directives**





## 7.3 Preprocessor Directives

- The C tradition is to have function declarations and type declarations in the header files and the function definitions in the source files.
- The preprocessor does textual replacement of the header file in the source file with the `#include` directive (while expanding necessary macros, doing conditional compilation etc as necessary).

## 7.3.1 #include

- Difference between `#include <filename>` & `#include "filename"`
- **Header vs Source File**
- All Declarations
- All Definitions except Typedefs
- Typedefs
- Defines
- Inline functions

## 7.3.2 #ifdef, #ifndef, #else, #endif

- In many cases, we need to have more than one version of the program and depending on the target platform, we may wish to compile accordingly.
- Conditional compilation supported by the preprocessor makes this easier.
- In any non-trivial implementations, it is usual to support debug and release versions and preprocessor comes handy.
- You can use conditional compilation to support that.
- An another common usage is to avoid multiple inclusion of headers.

## 7.3.3 #define, #undef

- Using `#define` directive is not limited to the conditional compilation alone
- It is used to provide symbolic constants as well as the function like macro expansion.
- The directives for conditional compilation are: `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, `#ifndef`, `#endif`.

# #define Constants

- Prior to ANSI C where `const` was not available, programmers just used the `#define` directive to have the functionality of the constants.
- The `#define` is not type safe and can be redefined again, defeating the whole purpose.
- Use `const`s and `enums` (for set of related values) instead of `#defines`:

```
3 int main()
4 {
5 #define MYCONSTANT 100
6 enum { enum_constant = 100; };
7 const int int_constant = 100;
8 printf("%d %d %d", MYCONSTANT enum_constant, int_constant);
9 }
```

Note that, with `#undef` directive, you can undefine a previously defined preprocessor constant or macro.

# #define Constants

- Will the following tiny code work!!!

```
3 #include <stdlib.h>
4 int main()
5 {
6 char *msg = "Hello World";
7 printf("%s", msg);
8 #include <stdio.h>
9 }
10 #include <stdio.h>
```

# #define Macros

- The `# define` directive can be used for function like macro definitions.
- What the main differences between the macros and functions?
- Consider the following macro definition:

```
3 #define max_macro(a, b) (a > b) ? a : b
4 inline int max_function (int a, int b)
5 {
6 return ((a > b) ? a : b);
7 }
```

## 7.3.4 #if, #else, #elif, #endif

- Check with the Jay, Code is very lengthy what to do?



## 7.3.5 #error, #line

- **C provides other constructs also like**
- `#error` to pass the error messages to the compiler
- `#line` to change the line number to be kept track while issuing such error messages.

## 7.3.6 #pragma

- The directive `#pragma` indicates that a particular feature is implementation dependent.
- If a compiler encounters an unknown option in `# pragma`, it is free to ignore it.

**Note :** These are compiler/assembler/linker dependent flags.

## 7.3.7 #, ##

- The preprocessor supports two operators useful for macro expansion:
- stringization("#")
- concatenation ("##")

```
1 #define STRINGIZE(string) #string
2 #define CONCATENATE(string1,string1) string1##string2
3 /* sample use: */
4 CONCATENATE(dou, ble) d;
5 /* this is same as declaring d as:*/
6 /* double d; */
7 printf("The int keyword is %s", STRINGIZE(int));
8 /* prints: the int keyword is int */
9
```

## 7.3.7 #, ##

- The preprocessor supports two operators useful for macro expansion:
- stringization("#")
- concatenation ("##")

```
1 #define STRINGIZE(string) #string
2 #define CONCATENATE(string1,string1) string1##string2
3 /* sample use: */
4 CONCATENATE(dou, ble) d;
5 /* this is same as declaring d as:*/
6 /* double d; */
7 printf("The int keyword is %s", STRINGIZE(int));
8 /* prints: the int keyword is int */
9
```

## **7.4 : Macro know-hows**



# Macros knows how

- Macros are not type-checked

```
1 #define STRINGIZE(string) #string
2 #define CONCATENATE(string1,string1) string1##string2
3 /* sample use: */
4 CONCATENATE(dou, ble) d;
5 /* this is same as declaring d as:*/
6 /* double d; */
7 printf("The int keyword is %s", STRINGIZE(int));
8 /* prints: the int keyword is int */
9
```

# Macros knows how

- Macros have side effects during textual replacement whereas
- functions does not have that since textual replacement is
- not done but a function call is made

```
3 int k = max_macro (i++, j);
4 /* we are in trouble as i++ is evaluated twice */
5 /* int k = (i++ > j) ? i++ : j */
6 int k = max_function (i++, j);
7 /* no problem, as it is not expanded, but a call to max_function is made */
8
```

# Macros knows how

- Macros might result in faster code as textual replacement is done and no function call overhead is involved.
- The function evaluates its arguments. A macro does textual replacement of its arguments, so it does not evaluate its arguments



# Macros knows how

- A function gets generated as a code and hence has an address (so you can use it for storing it in a function pointer).
- A preprocessor macro gets replaced with text replacement, so a macro becomes part of code in which it is used (it does not have an address by itself like a function, so you cannot use it for storing it in a function pointer).

```
typedef int (*fp)(int, int);
fp = max_function;
int k = fp(10, 20); /* ok, calls max_function */
fp = max_macro; /* error; unknown identifier max_macro */
int k = fp(10, 20);
```

## **7.5 : Practice - 1**

# Practice

- Define a macro `SIZEOF(x)`, where `x` is a variable, without using `sizeof` operator.
- 1. Prerequisite
- 2. Objective
- 3. Algorithm Design
- 4. Dry run
- 5. Practical Implementation

# Chapter 8: User Defined Datatypes

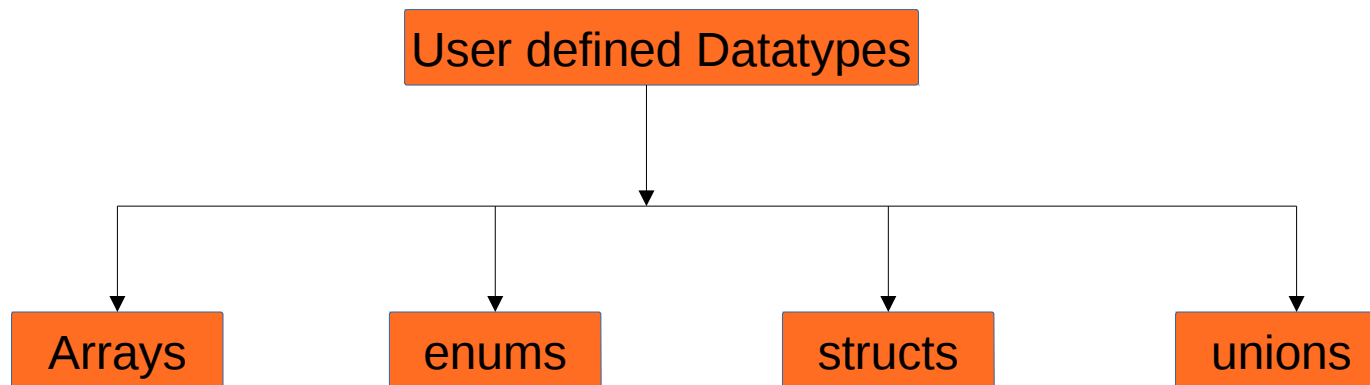


## **8.1 : Structures**



# 8.1 Why Structures ?

- Arrays, enums, structs, unions are the building blocks for the users to build their own type, as they need them.
- They can be built using primitive or aggregate types.



## 8.1 Various ways of defining a user-defined type

```
3 typedef int AgeType;
4 typedef float SalaryType;
5
6 typedef struct _Tag
7 {
8 char name[100];
9 AgeType age;
10 SalaryType salary;
11 } Tag;
```

```
14 typedef struct
15 {
16 char name[100];
17 AgeType age;
18 SalaryType salary;
19 } Tag;
20
21
```

```
21 struct _Tag
22 {
23 char name[100];
24 AgeType age;
25 SalaryType salary;
26 };
27 typedef Tag DB[100];
```

## 8.2 : Unions





## 8.2 Unions

able use in system programming, network programming and other related are  
coordinates in video memory can be addressed in DOS based machines as pair  
to represent the attribute describing the character followed by the actual char  
ish to use the pair of bytes as one unit or access as individual elements:

## 8.2 Unions

```
3 /* This is an implementation dependent code */
4 union VideoMemEntry
5 {
6 struct
7 {
8 unsigned char attr;
9 unsigned char value;
10 }entry;
11 short attrValue;
12 /* assume sizeof(short)==2 bytes */
13 }screen[25][80];
14 /* in a 25 * 80 text mode monitor */
15
16 screen[0][0].entry.attr = BOLD;
17 /* set the attribute of the character to BOLD */
18 screen[0][0].entry.value = 65;
19 /* the character to be displayed is ASCII character 'a' */
20 /* access them as individual bytes */
21 screen[0][0].attrValue = (BOLD<<8) + 65;
22 /*or set them together*/
```

## 8.2 Unions: Limitations

problems.

as it is easy to access wrong union members mistakenly and you have to keep

## 8.3 : `Sizeof()`

A decorative horizontal bar with a gradient from magenta to purple, ending in a double-lined arrow pointing to the right.

## 8.3 sizeof

```
struct example1
{
 char a[2];
 int x;
 char b[2];
};
```

sizeof(struct example1) = 12 (Assuming sizeof(int) == 4)

```
struct example2
{
 char a[2];
 char b[2];
 int x;
};
```

sizeof(struct example2) = 8 (Assuming sizeof(int) == 4)

## 8.4 why padding

- Ease of operation for compilers - Word aligned vs No padding
- Compiler dependent

## **8.5 : Initializing structures**



## **8.6 : Zero sized Array**





## 8.6 Zero sized Array

```
3 typedef struct
4 {
5 int n;
6 double val[0];
7 } Elements;
8 Elements *elements;
9 int i, n;
10 read(n);
11
12 elements = (Elements *) (malloc(sizeof(Elements) + n * sizeof(double)));
13
14 for(i = 0; i < n; i++)
15 read(elements->val[i]);
```

## **8.7 : Enumeration**



## 8.7 Enumeration

Enumeration specifies a set of named integral values and its members can be explicitly

```
3 enum color {black = 0, white = 1};
4 typedef enum
5 {
6 e_false,
7 e_true
8 } Boolean;
```

## 8.7 Enumeration

constraint about the values, they can be in any order and a same value can be repeated. Enumerations are part of the enclosing namespace and do not have a namespace of its own. This creates a problem in the surrounding namespace with an annoying problem that the enumerators must

```
3 enum color {black, white, orange, red, blue};
4 enum fruits {apple, orange, banana};
5 /* error: orange redefined */
6 int black;
7 /* error: black redefined */
```

## 8.7 Enumeration

created as integral values and this gives an added advantage they can take part

```
3 enum color {red = 1, green = 2, blue = 4};
4 int yellow = red + green;
5 /* now yellow = 3 */
6 int white = red + green + blue;
7 /* white = 7 */
```

Enumeration is useful in places where we need a closed set of named values (instead of having unrelated constant variables or preprocessor constants).

## **8.8 : Bit fields**



## 8.8 Bit fields

|       |       |    |      |   |
|-------|-------|----|------|---|
| 31:17 | 16:14 | 13 | 12:1 | 0 |
|-------|-------|----|------|---|

Field names: code(31:17), reset(16:14), enable(13), flags(12:1), priority(0)

## 8.8.1 Bit Operations

- Setting a field:
- Resetting a field:
- Extracting the value of a field:
- Putting a value into a field:



## 8.8.2 How with bit fields ?

```
5 struct
6 {
7 unsigned int code : 15;
8 unsigned int reset : 3;
9 unsigned int enable : 1;
10 unsigned int flags : 12;
11 unsigned int priority : 1;
12 }control;
```

## 8.8.3 why Bit fields?

- Ease of usage
- Example: `control.enable = 1;`

## 8.8.4 Ease vs Efficiency & Portability

## 8.8.5 Size considerations

### **Case Study I:**

- Increase enable to 2 bits
- Increase enable to 3 bits

### **Case Study II:**

- Replace unsigned int to unsigned long
- Replace unsigned int to unsigned short

## 8.8.6 Bit padding

Analogous to bytes and fields

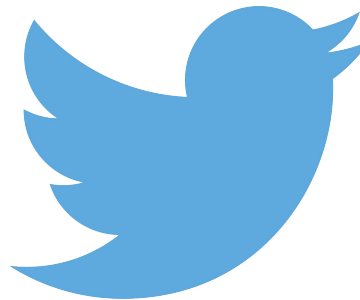
# Stay connected

**About us:** Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Emertxe Information Technologies,  
No-1, 9th Cross, 5th Main,  
Jayamahal Extension,  
Bangalore, Karnataka 560046  
T: +91 80 6562 9666  
E: [training@emertxe.com](mailto:training@emertxe.com)



<https://www.facebook.com/Emertxe>



<https://twitter.com/EmertxeTwitter>



<https://www.slideshare.net/EmertxeSlides>



THANK YOU