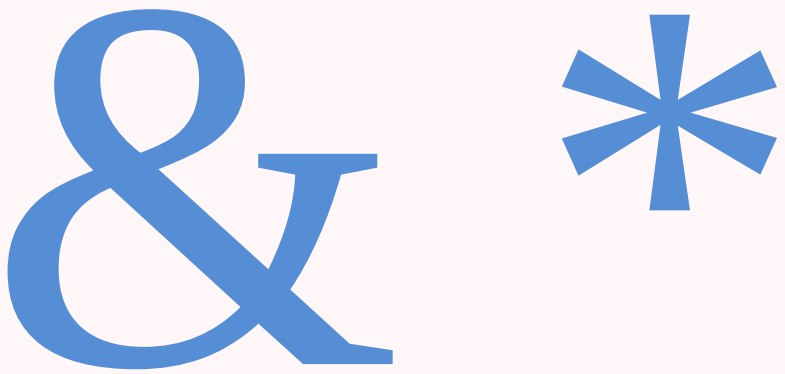
Tirgul2 - Agenda

- Pointers
- const
- C Strings
- Command line parameters
- Working with files
- Self reading about debugger

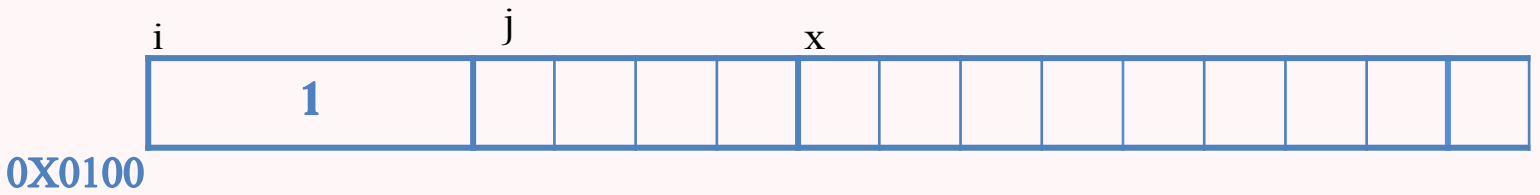
Pointers

- Data type for addresses
- (almost) all you need to know is:



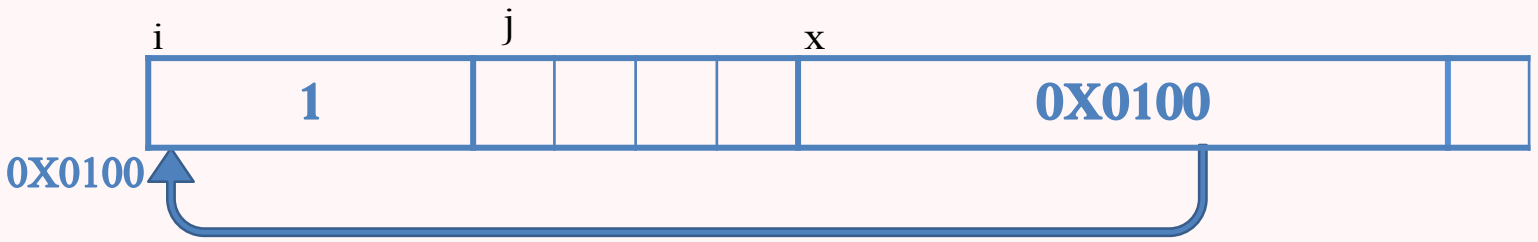
Pointers

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    …
```



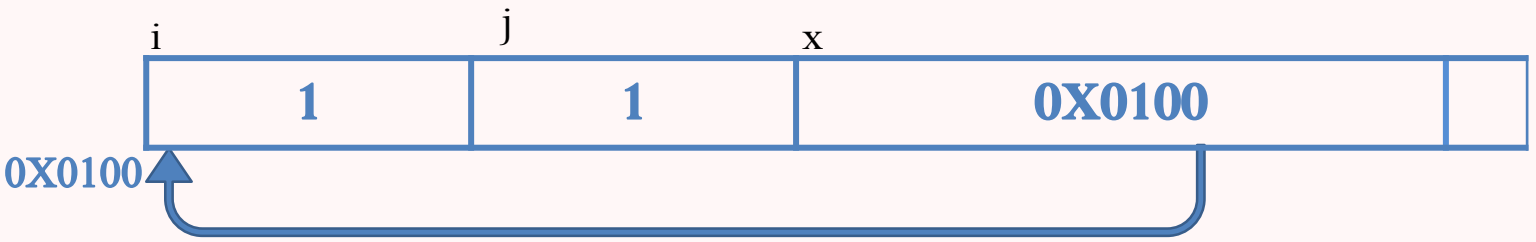
Pointers

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    …
```



Pointers

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    …
```



Pointers – brief summary

```
int main()
{
    int i;
    int *p;
    ...
```

- Which of the following is relevant?
  - &i
  - *i
  - &p
  - *p

C's "const"

- C's "const" is a qualifier that can be applied to the declaration of any variable to specify its value will not be changed.

- Example:

```
const double E = 2.71828;
E= 3.14;                    // compilation error!
```

Const and User Defined Types

```
typedef struct Complex
{
    int _img;
    int _real;
}Complex;

Complex const COMP1 = {1, 2};  // ok, copying values
Complex COMP2 = COMP1;  // ok, copying value by value

COMP1 = COMP2;                    // illegal!
COMP1._img = 3;                   // illegal!
```

- All the members of a const variable are immutable!

C's "const"

- **Const protects his left side, unless there is nothing to his left and only then it protects his right side.**

- Example**:**

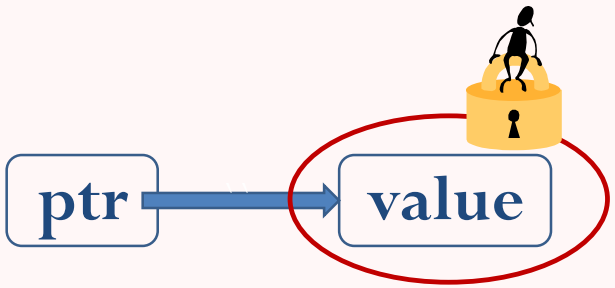
```
const int arr[] = {1,2};
arr[0] = 1;                    // compilation error!
```

C's "const"

- Do not confuse what the "const" declaration "protects"!

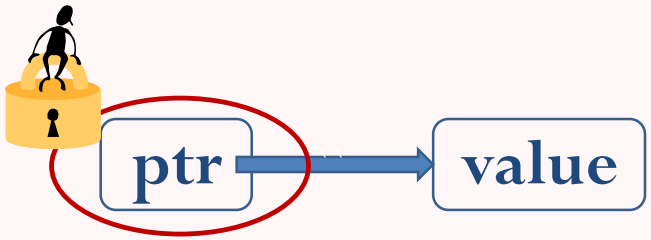
  - A pointer to a **const variable**:

```
int arr[] = {1,2,3};
int const * p = arr;
p[1] = 1;              // illegal!
*(p+1) = 1;          // same. illegal!
p = NULL;              // legal!
```



  - A **const pointer** to a variable:

```
int arr[] = {1,2,3};
int* const const_p = arr;
const_p[1] = 0;        // legal!
const_p = NULL;        // illegal!
```



C's "const"

- How about *arr* itself?

```
int arr[] = {1,2,3};
int* const const_p = arr;

arr = const_p;      // compilation error!
```

Const and Pointer's Syntax

- (2) and (3) are synonyms in C to a *pointer to a const int*

```
(1)int * const p = arr;
(2)const int * p = arr;
(3)int const * p = arr;
```

C's "const"

- C's "const" can be cast away

```
const int arr[] = {1,2};
int* arr_ptr = (int*)arr;

arr_ptr[0] = 3;      // compilation - ok!
                     // but might give a run-time error
```

- Helps to find errors.
- Doesn't protects from evil changes.

“Const” Usage

- The const declaration can (and should!) be used in the definition of a function’s arguments, to indicate it would not change them:

```
int strlen(const char []);
```

- Why use? (This is not a recommendation but a **must**)
  - clearer code
  - avoids errors
  - part of the interfaces you define!

- We will see more of “const” meaning and usage when we get to C++

Strings in C

- Java:
  - Char: is 2 bytes (Unicode)
  - String: an object which behaves as a primitive type (an immutable object, passed by value)
- C:
  - char: usually 1 byte. An Integer.
  - string: an array of characters.

```
char* txt1 = "text";

char txt2[] = "text";
char txt3[] = {'t','e','x','t','\0'};
```

C Strings

- Strings are always terminated by a *null character*, (a character with integer value 0).

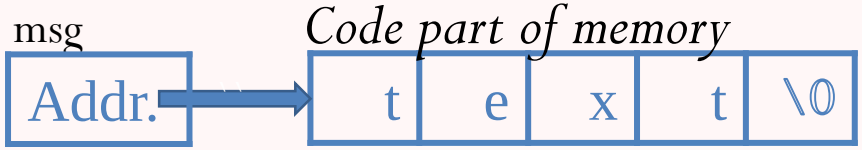
```
char* text = "string";
// means text[5] = g and text[6] = \0
// 7 chars are allocated!
```

- There is no way to enforce it automatically when you create your own strings, so:
  - remember it's there
  - allocate memory for it
  - specify it when you initialize char by char

C's string literals ("")

- When working with char*, C's string literals ("") are written in the code part of the memory.
  Thus, you can't change them!

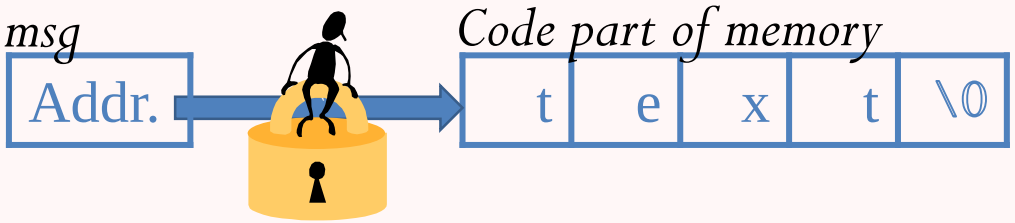
```
char* msg = "text";
msg[0] = 'w';                    // seg fault!
```



C's string literals ("") with const

- So, what we do is:

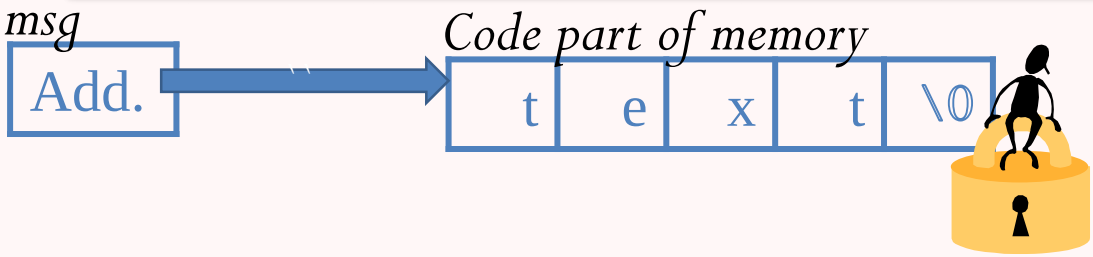
```
char const * msg = "text";
msg[0] = 't';          // compile error!
                       // better!
```



C's string literals ("")

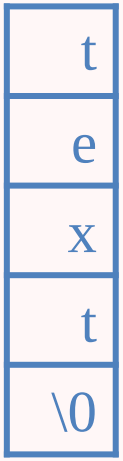
- Note the difference:

```
char* msg = "text";
// msg is a pointer that points to a memory
that is in the code part
```




```
char msg2[] = "text";
// msg2 is an array of chars that
are on the stack
```

Stack part of memory



C's string literals ("")

- Now we understand why:

```
char* msg = "text";
char msg2[] = "text";

msg[0]= 'n';      // seg fault - trying to change
                     what is written in the code
                     part of the memory
msg2[0]= 'n';     // ok - changing what is written
                     in the stack part of the memory
```

C Strings Manipulation

- To manipulate a single character use the functions defined in *ctype.h*

```
#include <ctype.h>
char c = ‘A’;
isalpha(c); isupper(c); islower(c);   …
```

- Manipulation of Strings is done by including the *string.h* header file

```
// copy a string
char* strcpy(char * dest, const char* src);
// append a string
char* strcat(char * dest, const char* src);
```

C Strings Manipulation (2)

```
// compare two strings.
// when str1 < str2 lexicographically return < 0
// when str1 > str2 lexicographically return > 0
// when identical return 0
int strcmp(const char * str1, const char* str2);

// return strings length, not including the \0!!
size_t strlen(const char * str);

// Other functions:
strncpy(),strncat(),strncmp()   …
```

- NOTE :
  - All C library functions assumes the usages of '\0' and enough storage space.
  - No boundary checks! You are responsible.
  - http://opengroup.org/onlinepubs/007908799/xsh/string.h.html

C Strings Examples

```
char txt1[] = "text";
char* txt2 = "text";

int i = strlen(txt1); // i = 4, same for strlen(txt2)

txt1[0] = ’n’;          // now txt1="next“
*txt2 = ’n’;            // illegal! “text” is in the code
                        // segment
txt2 = txt1;            // legal. now txt2 points to the
                        // same string.
txt1 = txt2;            // illegal!

if (! (strcmp(txt2,"next")) // after the legal commands -
                            // This condition is now true
{
   ...
```

C Strings Functions

- An “array” version of strcpy():

```
void strcpy(char * dest, const char* src)
{
   int i = 0;
   while ((dest[i] = src[i])!= '\0'))
   i++;
}
```

- A “pointers” version of strcpy():

```
void strcpy(char * dest, const char* src)
{
   while ((*dest = *src)!= '\0'))
   {
      dest++;
      src++;
   }
}
```

C Strings Functions (2)

- An experienced C programmer would write:

```
void strcpy(char * dest,const char* src)
{
   while ((*dest++ = *src++)!= '\0'));
}
```

- Actually the comparison against \0 is redundant:

```
void strcpy(char * dest,const char* src)
{
   while (*dest++ = *src++);
}
```

- **Style note**: Unlike K&R book, we do NOT encourage you to write such code. However, you should be able to read and understand it. The language features are used in any case.

Command-line arguments

```
int main(int argc, char* argv[])
{
      printf(“%s %d %s \n”, “you entered”, argc, “arguments”);
      printf(“%s: %s\n”, “the zero arg is the program name”, argv[0]);
      printf(“%s: %s\n”, “the first argument is”, argv[1]);
      printf(“%s: %s\n”, “the second argument is, argv[2]);

      int  i;
      for (i = 0; i < argc; ++i)
            printf(“arg num %d is %s\n”, i, argv[i]);
      …
}
```

File I/O

- File I/O is mostly similar to stdin & stdout I/O
- Most I/O functions we encountered have a “file” counterpart which receives a FILE pointer (handle)
- Examples:
  - getchar(void) fgetc(FILE*)
  - scanf(const char *,...) fscanf(FILE*, const char*,...)
  - printf(const char *,...) fprintf(FILE*, const char*,...)
- The standard streams (stdin, stdout, stderr) are also of FILE* type
- See related man pages: fprintf, fscanf, etc.

Opening and closing files

```
FILE* fp;
fp = fopen(filename, "r");

//fopen – returns a FILE pointer. Otherwise, NULL is returned and // the global variable errno is set to indicate the error.


if (fp == NULL)
{
   fprintf(stderr, “Cannot open the file");
}

     ...//Do stuff with file

fclose(fp);  //don’t forget to close the file when done
```

File I/O

- Read also about Binary File I/O
  - fread, fwrite – read/writes “raw” memory

Debugger

- To see how the program runs
  - flow
  - value of variables
- Breakpoints
- Break on expressions change
- Stack Trace: very helpful for seg faults!

GDB – Debugger for GCC

- **http://www.gnu.org/s/gdb/**
- allows you to see what is going on `inside' another program while it executes

- Break points and conditional breakpoint
- Examining
- Ad hoc changes
- Stepping
- Inspecting crashes

GDB – Debugger for GCC

- For debug use –g flag:
  - gcc –Wall **–g** hello.c –o hello
- Run gdb:
  - gdb hello
- Basic commands:
  - run, next, step, break, condition, continue, where, backtrace
- Many tutorials, for example:
  - http://www.cs.cmu.edu/~gilpin/tutorial/
  - http://ace.cs.ohiou.edu/~bhumphre/gdb.html

Debugging 101

1. “Define” the bug --- reproduce it
2. Use debugger (and other tools e.g. valgrind, debug prints)
3. Don’t panic --- think!
4. Divide & Conquer