Programming with C Advanced

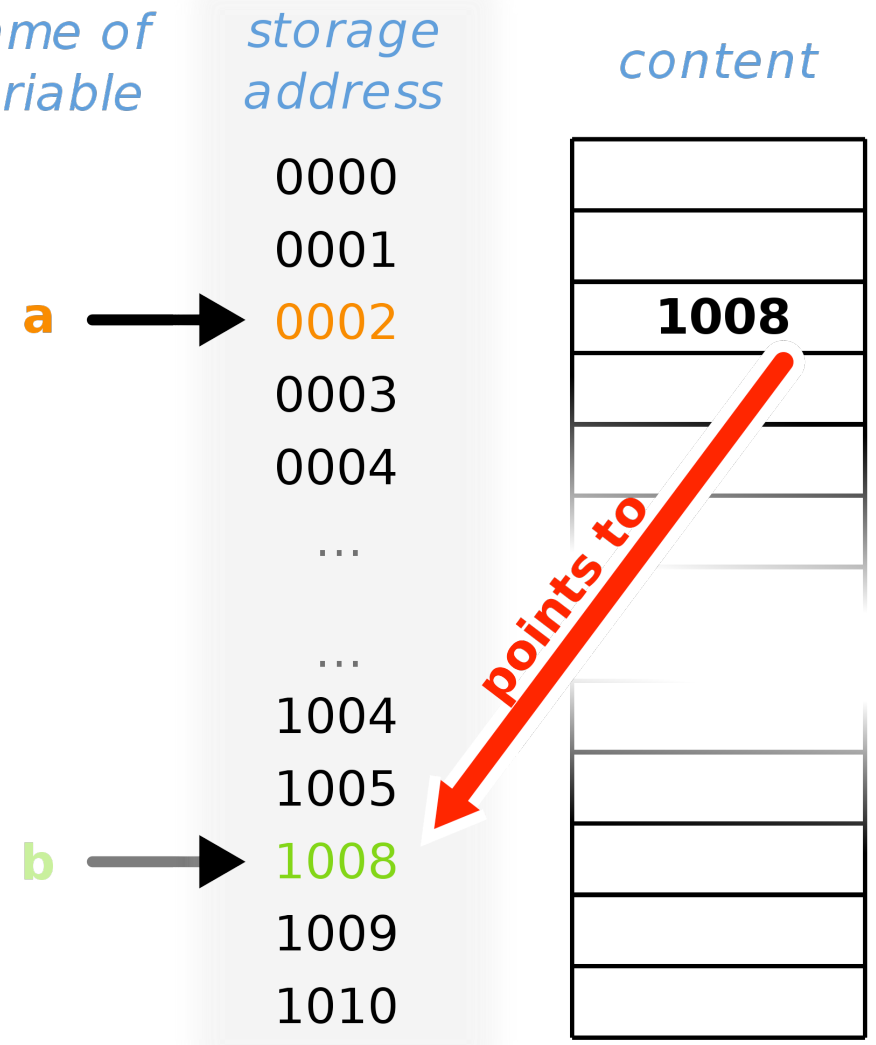
Vania Marangozova-Martin@imag.fr

M1 MOSIG

2015-2016

Addresses and Pointers: Notions

- A variable is stored in memory
  - it has a memory address
- A pointer is a variable whose value is the address of another variable



Addresses and Pointers: Syntax

```
int a = 10;       //an integer variable

int *pointer_a;   //a pointer to an integer variable

pointer_a = &a;   //take as value the address of a
```

```
scanf("%d", &a)   //enter the value of a from the keyboard
```

```
int **b;          //a pointer to a pointer to int
float ***c;       //a pointer to a pointer to a pointer of float

typedef struct {
    double double_member;
   char string_member[25];
} mystruct_t;

mystruct_t *m;
```

Addresses and Pointers: Syntax (2)

```
int a = 10;       //an integer variable

int *pointer_a;  //a pointer to an integer variable

pointer_a = &a;  //take as value the address of a

printf("%d\n", *pointer_a);    //access the variable
                               //pointed by pointer_a


*pointer_a = 20;               //change the pointed variable



//Indeed *pointer_a = *&a = a
```

Pointers and the NULL value

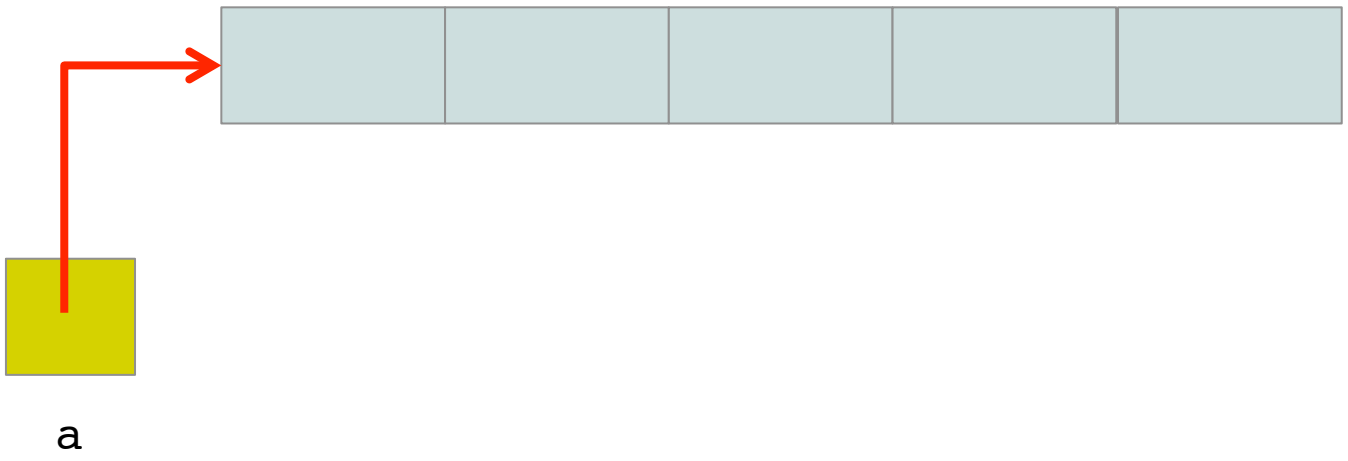
- NULL = invalid address

```
int a = 10;       //an integer variable

int *pointer_a = NULL;

printf("%d\n", *pointer_a);    //access the variable
                               //pointed by pointer_a

> ./a.out
> Segmentation fault: 11
```

Pointers and Arrays

- The name of an array is a pointer to its first element

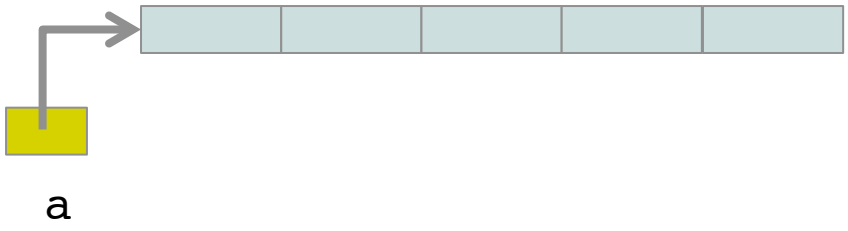
```
float a[10];
```



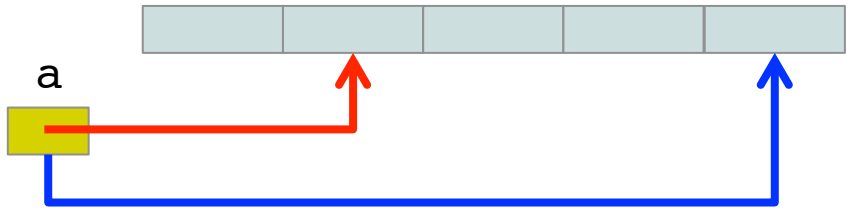
Pointer Arithmetics

- `a[0]` stored at address `a`
- `a[1]` stored at `a+sizeof(element of a)`
- `a[2]` stored at `a+2*sizeof(element of a)`

```
float a[5];
```




```
//advance the size of the
//pointed type
b = a; b++;
c = a + 4;
```



Pointer Arithmetics

```
uint32_t a = 10;    //an integer variable

uint32_t *pointer_a = &a;

//BEWARE, these are not the same!
printf("%d\n", sizeof(a));
printf("%d\n", sizeof(pointer_a));

//on a 64-bit machine
[~/Enseignement/Langage_C/auto-test] ./test_p2
4
8
```

Parameter Passing

- If you want to change the value of a variable in a function, you need to pass its address

PARAMETER by VALUE

```
int vglob;

int plus1(int x) {
   return x+1;
}

void main(void) {
   vglob = plus1(vglob);
}
```

PARAMETER by REFERENCE

```
int vglob;

int increment(int *x) {
   *x = *x + 1;
}

void main(void) {
   increment(&vglob);
}
```

Pointers, Parameters and `main()` (1)

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

  printf("The program is \"%s\".\n",argv[0]);
  printf("Nb arguments %d\n", argc);

  if (argc > 1){
     for (count = 1; count < argc; count++) {
        printf("argv[%d] = %s\n", count, argv[count]);
     }
  } else {
     printf("The command had no other arguments.\n");
  }
  return 0;
}
```

Pointers, Parameters and `main()` (2)

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

  printf("The program is \"%s\".\n",argv[0]);
  printf("Nb arguments %d\n", argc);

  if (argc > 1){
     for (count = 1; count < argc; count++) {
        printf("argv[%d] = %s\n", count, argv[count]);
     }
  } els
     pr
  }
  retur
}
```

```
> ./mainarg 1 2 abc toto 12.0
The program is "./mainarg".
Nb arguments 6
argv[1] = 1
argv[2] = 2
argv[3] = abc
argv[4] = toto
argv[5] = 12.0
```

Pointers, Parameters and `main()` (3)

```
int main (int argc, char *argv[]) {
  int p1;
  double p2;
  long p3;

  printf("The program is \"%s\".\n",argv[0]);
  printf("Nb arguments %d\n", argc);

  if (argc == 4){
    p1 = atoi(argv[1]);
    p2 = atof(argv[2]);
    p3 = atol(argv[3]);
  } else {
    printf("Usage %s int double long\n", argv[0]);
  }
  return 0;
}
```

Pointers and Structures

```
struct polCoord {
   float r;
   float theta;
};

struct polCoord p1,p3;
struct polCoord *p2 = &p1;

*p2 = (struct polCoord){1.0 , 0.0};

p3.r =        p2 -> r;
p3.theta =    p2 -> theta;
```

Back to Arrays

- Predefined size

```
#define SIZE_ARR 10

void init(unsigned int nb,
          int t[]) {
    int i;
    for (i=0; i<nb; i++)
        t[i] = 2*i;
}

void main(void) {
    int myarr[SIZE_ARR];
    init(SIZE_ARR, myarr);
}
```

- Size not known statically

```
void init(    unsigned int nb,
              int t[]) {
    int i;
    for (i=0; i<nb; i++)
        t[i] = 2*i;
}
…
void main(void) {
    unsigned int nb;
    int *myarr;
    scanf("Enter size", &nb)
    myarr = malloc(nb*sizeof(int));
    init(nb, myarr);
}
```

Back to Arrays

- Predefined size

```
#define SIZE_ARR 10

void init(unsigned int nb,
          int t[]) {
    int i;
    for (i=0; i<n
        t[i
}

id main(void) {
    int myarr[SIZE_ARR];
    init(SIZE_ARR, myarr);
}
```

Static memory allocation

- Size not known statically

```
void init(    unsigned int nb,
              int t[]) {
    int i;
    for (i=0; i<nb; i++)
        t[i] = 2*i;
}
…
void main(void) {
    unsigned int nb;
    int *myarr;
    scanf("Enter size", &nb)
    myarr = malloc(nb*sizeof(int));
    init(nb, myarr);
}
```

Back to Arrays

- Predefined size

```
#define SIZE_ARR 10

void init(unsigned int nb,
          int t[]) {
    int i;
    for (i=0; i<n
        t[i
}

id main(void) {
    int myarr[SIZE_ARR];
    init(SIZE_ARR, myarr);
}
```

Static memory allocation

- Size not known statically

```
void init(    unsigned int nb,
              int t[]) {
    int i;
    for (i=0; i<nb; i++)
        t[i] = 2*i;
}
…
vo                    ) {
    ned int nb;
    int *myarr;
    scanf("Enter size", &nb)
    myarr = malloc(nb*sizeof(int));
    init(nb, myarr);
}
```

Dynamic memory allocation

Memory Management

```
#define LINES 4
#define COLUMNS 5

int main(int argc, char** argv) {
    int ** matrix; //two-dimensional array
    matrix = malloc(LINES * sizeof(*matrix));
    for (i = 0; i < LINES; i++) {
         matrix[i] = malloc(sizeof(**matrix) * COLUMNS);
    }
    …
    for(i = 0; i < LINES; i++) {
         free(matrix[i]);
    }
    free(matrix);
}
```

On Memory Management

- For detecting memory problems
  - valgrind

The `void*` Pointer

- A pointer that can point to any type
  - no arithmetics
  - genericity

```
void memzero(void *ptr, size_t len)
{
    for(; len>0; len--) {
        *(char *)ptr = 0;       //type casting
    }
}
```

Type Casting

- Type casting is a way to convert a variable from one data type to another data type.

```
main() {
   int sum = 17;
   int count = 5;
   double mean;

   mean = sum / count;
   printf("M: %f\n", mean );

}
```

```
> 3.000000
```

```
main() {
   int sum = 17;
   int count = 5;
   double mean;

   mean = (double) sum / count;
   printf("M: %f\n", mean );

}
```

```
> 3.400000
```

Type Casting: Another Example

```
#define NB_CHARS 128

int main(int argc, char** argv) {
    char *M = malloc(NB_CHARS+sizeof(int));

    int *nb_p = (int*)M;
    char* start = M+sizeof(int);

    *nb_p = NB_CHARS;

    *start = 'a';
    *(start+1) = 'b';
    *(start+2)='c';
    *(start+3)='\0';

    *nb_p=*nb_p-3;
    printf("%d %s\n", *nb_p, start);
}
```

Pointer to Function

```
int func (int a, int b) {
    printf("\n a = %d\n",a);
    printf("\n b = %d\n",b);
    return 0;
}

int main(void) {
    int(*fptr)(int,int); // Function pointer

    fptr = func; // Assign address to function pointer

    func(2,3);
    fptr(2,3);

    return 0;
}
```

Debugging

- When your program does not work
  - printf is a very limited means to find out about what is wrong
  - gdb is your friend
  - you are provided with a gdb tutorial for the lab

Versioning

- It is a good practice to backup and keep the modification history of your code
  - in case of a crash
  - in case of some curious buggy behavior that appeared out of nowhere
  - for collaborative work
- SVN and GIT
  - track changes
  - status of files
  - revert to a previous state
  - diff

References

- GIT
  - https://git-scm.com/
  - and many tutorials on the web