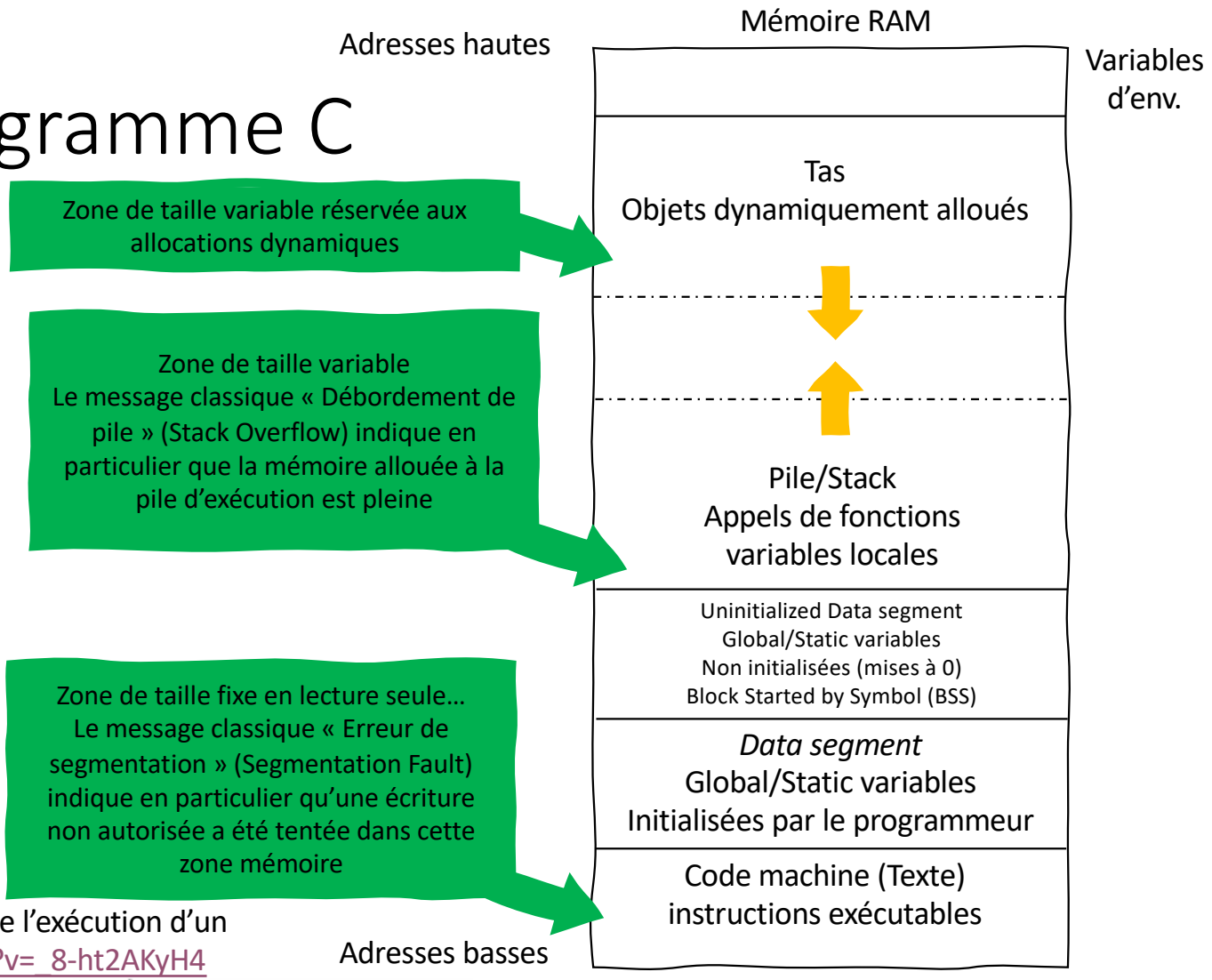
Exécution mémoire (RAM) d'un programme C La pile et le tas !

- Pile (LIFO) : espace mémoire de travail pour un thread d'exécution
  - A l'appel d'une fonction, réservation d'un bloc en sommet de pile pour les variables locales
  - Lors du retour le bloc est libéré (mise à jour pointeur sommet de pile)
- Tas : mémoire réservée à l'allocation dynamique
  - Allocation et désallocation libre

Exécution d'un programme C

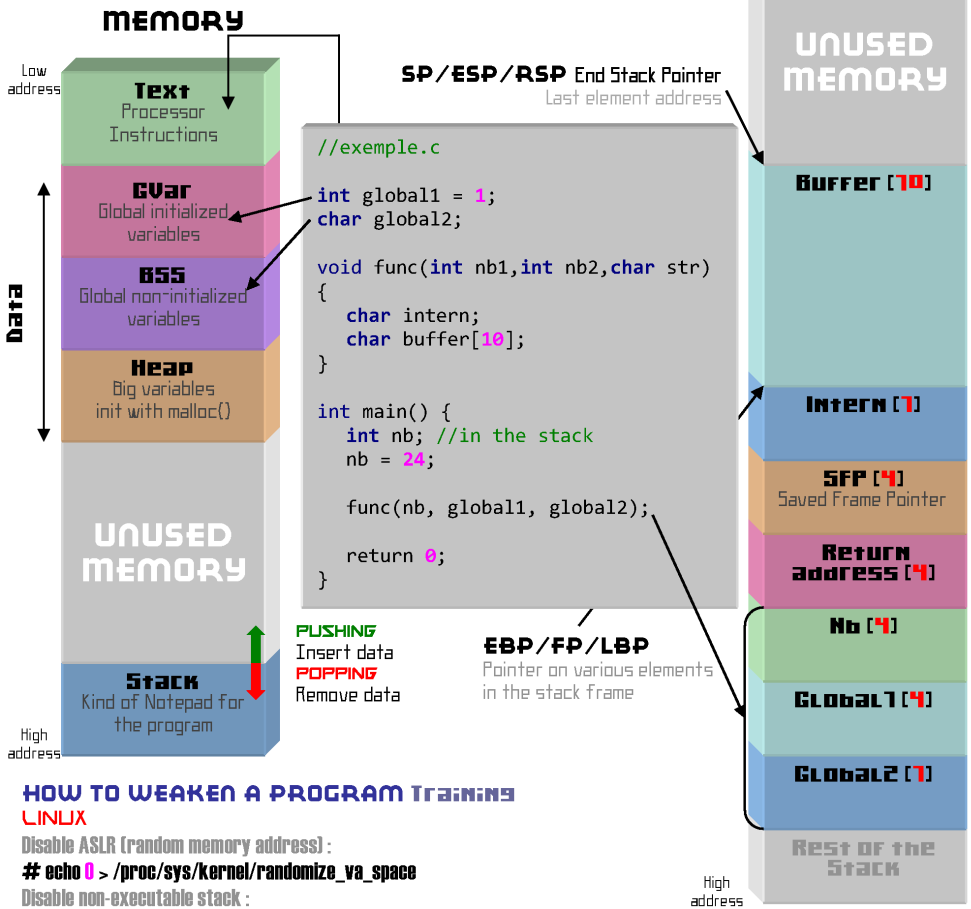
La taille de la pile d'appel (d'exécution) dépend de plein de facteurs le langage de programmation, l'architecture de la machine, machine architecture, quantité de mémoire disponible, etc. Si le programme essaye d'utiliser plus d'espace que ce qui est disponible dans la pile (ce qui est un buffer overflow) la pile est en débordement…



Une explication des concepts de pile et de tas lors de l'exécution d'un programme en C https://www.youtube.com/watch?v=_8-ht2AKyH4

Memory Segmentation Cheat Sheet

0x0ff.info production, inspired by bases-hacking.org



How to weaken a program Training

Linux

Disable ASLR (random memory address) :

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

Disable non-executable stack :

```
$ gcc -z execstack ...
```

Disable stack protector :

```
$ gcc -fno-stack-protector ...
```

Force 32-bits compilation mode :

```
$ gcc -m32 ...
```

Global registers

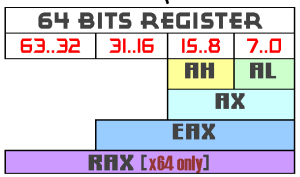
Used for general purpose

X86

AL/AH/AX/EAX/RAX
BL/BH/BX/EBX/RBX
CL/CH/CX/ECX/RCX
DL/DH/DX/EDX/RDX

ARM

r0-r12



Index pointers

Use for Strings operations

X86

SI/ESI/RSI : Source index
DI/EDI/RDI : Destination index

Instruction pointer

The current instruction address.

X86

IP/EIP/RIP

ARM

PC

Segment registers

Use to easily reade/write to memory

X86

CS : Code	ES : Extra data #1
DS : Data	FS : Extra data #2
SS : Stack	GS : Extra data #3

Memory alignment

Data must be aligned on 4,8,16... Bytes, depending on your system.

EXAMPLE

There is an 3 bytes long empty gap between intern and SFP.

Buffer overflow

when input is longer than the allocated memory space.

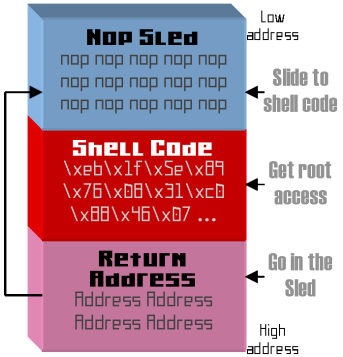
[Stack based]

Smart overwrite of return address

EXAMPLE

Put a 22 bytes long string and overwrite intern Return address.

Exploit anatomy



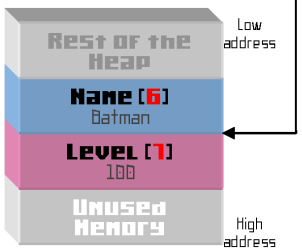
[Heap based]

Smart overwrite of others variables like file name.

EXAMPLE

You have to enter a 6 letters name in the character builder of a game. You enter Batman\x64, to overwrite level variable and get max stats !

HEAP



Les erreurs communes dans la gestion de la mémoire en C

Cf. Exercice TD identification erreur gestion de mémoire

1. Fuites mémoire (*memory leaks*) si on oublie de libérer via free la mémoire allouée
   Peut prendre plus ou moins de temps…
2. Mémoire insuffisante pour réussir l'allocation, malloc retourne NULL
   Toujours vérifier que malloc retourne un pointeur != NULL
3. Non initialisation de la mémoire allouée via malloc
   Si vous voulez une initialisation à 0 utilisez calloc (plus lent)
4. Non allocation mémoire préalablement à une tentative d'accès en lecture ou écriture
   SEGMENTATION FAULT
5. Non libération de la mémoire allouée
   Dès que vous écrivez malloc ou calloc écrivez le free correspondant
6. Pointeurs pendant (*dangling pointer*), tentative d'accès à de la mémoire déjà libérée via un pointeur qui ne pointe plus sur de la mémoire allouée
7. Désallocation mémoire multiple
   Toujours valuer un pointeur libéré à NULL car free ne fait rien sur un pointeur NULL

Les erreurs/maladresses communes dans la gestion de la mémoire en C

Cf. Exercice TD identification erreur gestion de mémoire

8. Tentative de libération de mémoire non allouée dynamiquement
   SEGMENTATION FAULT
9. Utiliser une allocation dynamique (plus lente) de tableau, alors qu'une allocation statique (int tab[TAILLE]) aurait suffi
   C'est le cas si vous n'avez pas besoin du tableau déclaré en dehors de votre fonction
10. Utiliser sizeof sur un tableau alloué dynamiquement pour déterminer sa taille
    Fonctionne avec un tableau statique, mais pas dynamique, dans ce cas c'est la taille du pointeur qui est retournée
11. Toujours utiliser le pointeur ayant servi à l'allocation dynamique pour la libération de la mémoire
    Travailler sur une copie de ce pointeur afin de ne jamais perdre l'adresse originale de début du bloc alloué.

Pointeur générique void*

- Pointeur de type indéfini, permettant de manier l'adresse mémoire d'un objet dont on ignore le type.
- Toutes les opération d'accès en lecture ou écriture nécessiteront un cast explicite

```
int un_int = 49;
void* pt = &un_int;
printf("un_int contient %d",*((int*)pt));
//printf("Essayez sans cast cela fait une erreur à la compilation ! %d", *pt);
return 0;
```

Nous reviendrons sur le sujet…

Exercice

- Déclarez deux entiers et trois pointeurs sur void et échangez le contenu des deux entiers initialisés respectivement à 10 et 20 de façon qu'à la fin de l'exécution les valeurs aient été échangées.
- Vous n'avez le droit de manipuler les entiers que via les pointeurs et jamais directement.

Solution

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

int main(int argc, char const *argv[]){
    int a = 10;
    int b = 20;
    void * pt_a = &a;
    void * pt_b = &b;
    void * pt_temp = NULL;
    printf("Les deux entiers au départ sont %d et %d\n",*(int*)pt_a,*(int*)pt_b);
    //Permutation utilisant les pointeurs
    pt_temp = (int*)malloc(sizeof(int));
    *(int*)pt_temp = *(int*)pt_a;
    *(int*)pt_a = *(int*)pt_b;
    *(int*)pt_b = *(int*)pt_temp;
    printf("Les deux entiers après permutation sont %d et %d\n",*(int*)pt_a,*(int*)pt_b);
     return 0;
}
```

Pointeurs et constantes

- Pointeur variable sur un objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- Pointeur constant sur un objet variable

```
int *const pt_const_sur_int;
```

- Pointeur constant sur un objet constant

```
const int * const pt_const_sur_int_const;
```

Pointeur variable sur objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- L'objet pointé devient constant donc non modifiable par le pointeur même s'il n'est pas constant à l'origine.
- Le pointeur peut changer de valeur et pointer sur un autre objet.

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

int main(){
    const int val = 100; //objet constant
    int v2 = 0; //objet non constant
    const int *p = &val; //pointeur de const int, non constant
    *p += 10; // erreur, objet pointé supposé constant
    p = &v2;// p non constant, peut varier

    *p += 10; // erreur, objet pointé supposé constant

    printf("val : %d et *p : %d\n", val, *p);

    return 0;
}
```

Pour chaque instruction ci-contre déterminez s'il y a erreur ou non et justifiez votre réponse.

Variation… Que se passe-t-il ici ?

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

int main(){
    const int val = 100;      //objet constant
    int *p = &val;      //pointeur sur int ordinaire
    *p += 10;     // pas d'erreur, pourtant objet pointé supposé constant

    printf("val : %d et *p : %d\n", val, *p);

    return 0;
}
```

Le type du pointeur prime sur celui de l'objet pointé !

Pointeur constant sur objet variable ou non

```
int value0=10;
int value1=60;
const int valueconst=100;

int *const pt_const_sur_int=&value0;//pointeur constant sur int
int *const pt_const_sur_int_const=&valueconst; //pointeur constant sur un int
                                                //constant
pt_const_sur_int = &value1; //erreur pt_const_sur_int constant

*pt_const_sur_int_const += 10;//OK objet pointé devient modifiable, même
                              //si constant à l'origine

printf("val : %d et *p : %d\n", valueconst, *pt_const_sur_int);
```

Le type du pointeur prime sur celui de l'objet pointé et détermine le fonctionnement du pointeur !

Pointeur constant sur objet constant

```
int value0=10;
int value1=60;
const int * const pt_const_sur_const_int=&value0;
pt_const_sur_const_int=&value1; //Erreur car pointeur constant

*pt_const_sur_const_int += 10; //Erreur pointeur sur un const int


printf("val : %d et *p : %d\n", value1, *pt_const_sur_const_int);
```

Ici L'objet pointé, qu'il soit constant ou pas, et pointeur sont considérés constants et ne peuvent changer de valeur après l'affectation de déclaration.

Opérations et opérateurs sur les pointeurs

- Les pointeurs ont pour valeur des adresses mémoires
- Plusieurs opérations sont possibles sur les **pointeurs de même type**
  - L'addition avec un entier
  - La soustraction avec un entier
  - La différence entre deux pointeurs
  - La comparaison entre pointeurs via opérateurs ==, !=, <, >, etc.

https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/

Arithmétique des pointeurs, addition, soustration d'un entier

Les pointeurs sont des adresses mémoire, les opérations autorisées sur les adresses mémoire sont :

- L'addition (ou la soustraction) d'un entier à un pointeur : `pointeur + unentier` (ou `pointeur – unentier`).
  - Le résultat de `pointeur + unentier` est une nouvelle adresse, décallée de `unentier` cases.
  - *Attention* : La taille d'une "case" (en nombre d'octets) dépend du type du pointeur, c'est-à-dire de la taille mémoire de l'objet pointé.

Ainsi :

- Si `pointeur` est de type int *, alors `pointeur + N` décalle la case mémoire de N cases, et ajoutera donc (N*4) octets à la valeur de pointeur, puisqu'un int occupe 4 octets en mémoire;
- Si `pointeur` est de type char *, alors `pointeur + N` décalle la case mémoire de N cases, et ajoutera donc (N*1) octets à la valeur de pointeur, puisqu'un code char occupe 1 octet;
- Si `pointeur` est de type double *, alors `pointeur + N` décalle la case mémoire de N cases, et ajoutera donc (N*8) octets à la valeur de pointeur, puisqu'un double occupe 8 octets;

https://chamilo.grenoble-inp.fr/courses/ENSE3A13EMAPPSIC/document/resources/Fiches/1XX-Arithmetique-de-pointeurs.html

Arithmétique des pointeurs
Différence entre deux pointeurs

- Possible qu'entre pointeurs du même type.
- La valeur retournée sera égale au nombre de cases (conformément à la taille mémoire des objets pointés) existants entre les deux pointeurs.

Comparaison entre pointeurs

- Il n’est possible que de comparer des pointeurs sur le même type.
- On peut alors savoir si :
  - Les pointeurs pointent sur la même adresse mémoire ==
  - Les pointeurs pointent sur des adresses mémoire différentes !=
  - L’un des pointeur pointe « plus loin » en mémoire que l’autre

Les fonctions

Mode de passage d'arguments ?

…

Comment renvoyer plusieurs résultats ?

Nombre variable d'arguments ?

Fonctions ou Macros ?

UNIVERSITÀ DI CORSICA
PASQUALE PAOLI

Qu’est-ce qu’une fonction ?

- Unité de traitement fondamentale du C
  - Permet la **factorisation** et donc la **généralisation** de code qui se répète
  - Consiste en la **paramétrisation** d’un bloc de code
- Bloc d’instructions doté d’un nom et :
  - D’un mécanisme d’entrée de valeurs : les paramètres
  - D’un mécanisme de sortie de valeur : **la** valeur de retour

En C passage d'arguments par valeur

- Cette règle ne souffre aucune exception !

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

void modif(int a, int b){
     a = 10;
     b = 15;
     printf("Dans la fonction modif a=%d, b=%d\n",a,b);
}

int main(int argc, char const *argv[]){
     int a=0, b=0;
     modif(a,b);
     printf("Dans le main, après appel à modif a=%d, b=%d\n",a,b);
     return 0;
}
```

Qu'est-ce qui est affiché à l'exécution ?

Implications du passage d'arguments par valeur

- La valeur de l'expression passée en paramètre est copiée dans une variable locale
- Les modifications internes à la fonction portant sur les variables paramètres ne sont pas répercutées à l'extérieur de la fonction puisque on travaille sur des copies locales

Exercice Swap, cas d'école…

Ecrivez une fonction swap_int qui prend en paramètre deux pointeurs sur des entiers et qui échange le contenu des deux variables de telle sorte que la valeur de la variable 1 se retrouve dans la variable 2 et vis-versa.

Vous invoquerez cette fonction depuis un main et afficherez les contenus des deux variables avant et après appel à la fonction swap.

Solution

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

void swap(int* pt_a, int* pt_b){
    int temp = *pt_a;
    *pt_a = *pt_b;
    *pt_b=temp;
}

int main(int argc, char const *argv[]){
    int a = 10;
    int b = 20;
    printf("Les deux entiers au départ sont %d et %d\n",a, b);
    //Permutation via appel à Swap
    swap(&a,&b);
    printf("Les deux entiers après permutation sont %d et %d\n",a, b);
    return 0;
}
```

Passage par référence et pointeurs

- Si on passe en paramètre à une fonction l'adresse mémoire d'une variable il lui sera possible le lire et écrire dans cet espace mémoire et ainsi de modifier le contenu de cette variable
- L'adresse mémoire d'un objet est une **référence**
- Les variables qui prennent comme valeur des références sont des **pointeurs**
- Ce qui est passé en paramètre c'est **la valeur de l'adresse**

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

int main(int argc, char const *argv[]){
    int intlu = 0;
    scanf("%d", &intlu); //On passe la référence à intlu, cad son adresse
    printf("La valeur lue est %d", intlu);
}
```

Fonction empoisonnement…

Ecrivez une fonction `empoisonner` prenant deux paramètres entiers, permettant de soumettre un personnage – représenté par un nombre entier de points de vie – à un poison – représenté également par un entier. Lorsque le poison agis, il fait perdre des points de vie au personnage (point de vie du personnage - points de nocivité du poison), et perd également de la nocivité (décrémentation des points de nocivité du poisson de 1 à chaque empoisonnement).

Quand le poison est totalement épuisé, il retrouve par magie de la force, sans toutefois jamais dépasser 20 !

Proposez un code pour la fonction empoisonnement

```
Avant empoisonnement Faustus a 35 points de vie et la cigue est au niveau 15
Après empoisonnement Faustus a 20 points de vie et la cigue est au niveau 14
```