ADDRESS AND POINTER

C Programming

université de BORDEAUX

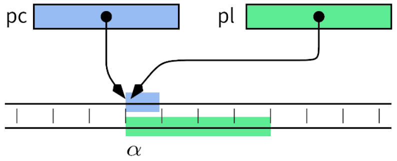
FROM ADRESSES TO POINTERS

- ▷ memory address: start of a memory area (`unsigned integer` α)
- ▷ reference: address of typed data (α, type)
- ▷ pointer: variable containing a reference (pointer of type)

FROM ADRESSES TO POINTERS

- ▷ memory address: start of a memory area (`unsigned integer` α)
- ▷ reference: address of typed data (α, type)
- ▷ pointer: variable containing a reference (pointer of type)

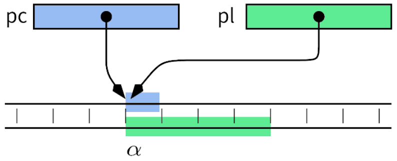
- ▷ example
  - ▷ `pc` : pointer to `char`
  - ▷ `pl` : pointer to `long`



- ▷ so, both pointers `pc` and `pl` contain
  - ▷ the same address α
  - ▷ but different references (α, char) and (α, long)

FROM ADRESSES TO POINTERS

- ▷ memory address: start of a memory area (`unsigned integer` α)
- ▷ reference: address of typed data (α, type)
- ▷ pointer: variable containing a reference (pointer of type)
- ▷ example
  - ▷ `pc` : pointer to `char`
  - ▷ `pl` : pointer to `long`



- ▷ so, both pointers `pc` and `pl` contain
  - ▷ the same address α
  - ▷ but different references (α, char) and (α, long)

- ▷ all pointers have the same size (architecture dependent)

DECLARATION AND USE OF POINTERS

- ▷ Pointer p to a type T : `T *p`
- ▷ May (and should) be initialized when declared
- ▷ NULL : null pointer of value 0L (`<sdtlib.h>`)
- ▷ unvalid address : *NULL → Segmentation fault

▷ Thus, it is always a good practice to test whether the value of a pointer is NULL

```
if(p!=NULL){
    ...
}
```

DECLARATION AND USE OF POINTERS

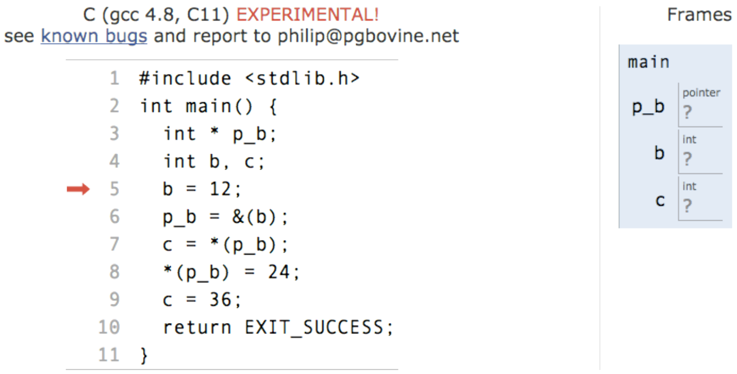
▷ Related operators :

*	get an address content (dereferencing)
&	get a variable address

```
char *s = NULL;
int v = 0;
int *pointer_to_v = &v;
int n = *pointer_to_v + 1;

*pointer_to_v = 256;
```

DECLARATION AND USE OF POINTERS



C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
#include <stdlib.h>
int main() {
  int * p_b;
  int b, c;
  b = 12;
  p_b = &(b);
  c = *(p_b);
  *(p_b) = 24;
  c = 36;
  return EXIT_SUCCESS;
}
```

DECLARATION AND USE OF POINTERS

- ▷ Pointers are usefull for **accessing an address**
  - ▷ **read** the content of a memory location
  - ▷ **write** data to a memory location
  - ▷ transmit a reference to a function **call by reference**

AN EXAMPLE OF CALL BY REFERENCE : SWAP

▷ Write a function that **swaps the contents** of two variables

```
int main(int argc, char *argv[])
{
    int a = 42;
    int b = 24;
    printf("Before swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
.......
    printf("After swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
    return EXIT_SUCCESS;
}
```

AN EXAMPLE OF CALL BY REFERENCE : SWAP

▷ Write a function that **swaps the contents** of two variables

```
/* not working */
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main(int argc, char *argv[])
{
    int a = 42;
    int b = 24;
    printf("Before swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
    /* not working */
    swap(a, b);
    printf("After swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
    return EXIT_SUCCESS;
}
```

AN EXAMPLE OF CALL BY REFERENCE : SWAP

▷ Write a function that **swaps the contents** of two variables

```
/* thing to do */
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main(int argc, char *argv[])
{
    int a = 42;
    int b = 24;
    printf("Before swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
    /* thing to do */
    swap(&a, &b);
    printf("After swap %d(%p) %d(%p)\n",
                        a, &a, b, &b);
    return EXIT_SUCCESS;
}
```

POINTER ARITHMETIC

▷ ⚠Any arithmetic operation performed over a pointer will take into account the size of the target data type

```
type x,k;
int j=2;
type * p_x = &(x);
k=*(p_x+j);
// k = content at address &(x)+j*sizeof(type)
```

▷ Interest ?

▷ Why is it possible ?

ABOUT POINTERS AND ARRAYS

- ▷ An array identifier v is a constant of value a reference

$$v \simeq \&(v[0]) \simeq \&v$$

- ▷ A pointer p is a variable containing a reference ($p \neq \&p$)
- ▷ pointers and array both are references operators * and [] both are dereferencing operators
- ▷ `x[i]` $\iff$ `*(x + i)`

ABOUT POINTERS AND ARRAYS

```
int v[] = { 1, 2, 3, 5, 7 };
int *pv = v;
printf("v[2]       = %d\n", v[2]);
printf("*(pv + 2) = %d\n", *(pv + 2));
printf("*(v + 2)  = %d\n", *(v + 2));
printf("pv[2]      = %d\n", pv[2]);
```

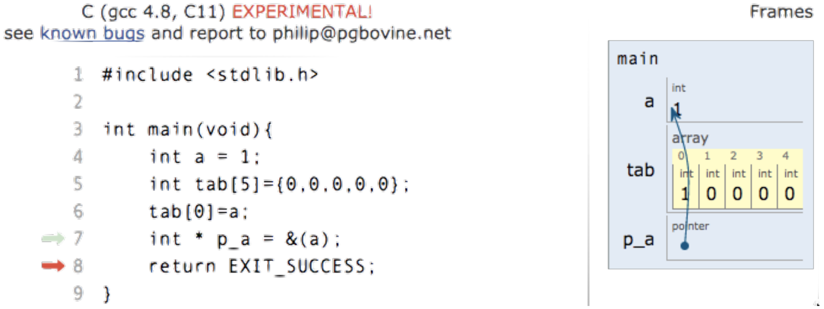
```
v[2]      = 3
*(pv + 2) = 3
*(v + 2)  = 3
pv[2]     = 3
```

```
printf("v       = %p\n", v);
printf("&(v[0]) = %p\n", &(v[0]));
printf("pv      = %p\n", pv);
```

```
v       = 0x7ffe976ed4e0
&(v[0]) = 0x7ffe976ed4e0
pv      = 0x7ffe976ed4e0
```

REMINDER ON ARRAYS

- ▷ ⚠The allocated memory space isn't the same: `tab` corresponds to an address where 5 integers can be stored; `p_a` corresponds to an address where a single address can be stored

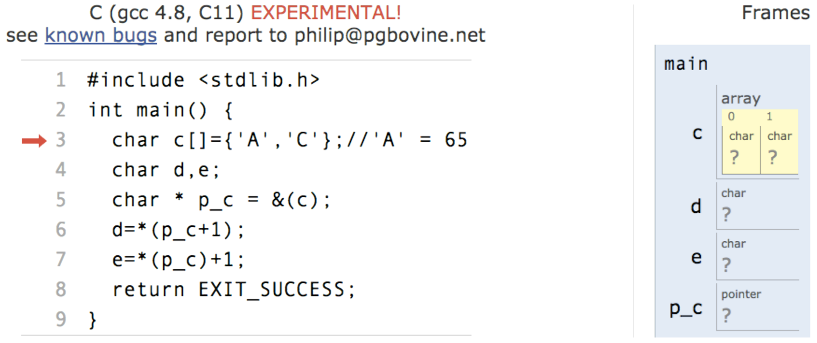


C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
#include <stdlib.h>

int main(void){
     int a = 1;
     int tab[5]={0,0,0,0,0};
     tab[0]=a;
     int * p_a = &(a);
     return EXIT_SUCCESS;
}
```

POINTER ARITHMETIC



C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
#include <stdlib.h>
int main() {
  char c[]={'A','C'};//'A' = 65
  char d,e;
  char * p_c = &(c);
  d=*(p_c+1);
  e=*(p_c)+1;
  return EXIT_SUCCESS;
}
```

STRINGS - CHAR[]

- ▷ Not a valid data type, but rather a storage standard
  - ▷ Array of characters ended by a null character `'\0'`
- ▷ Surrounded by double quotes
  - ▷ `char msg[]="Welcome";`
  - ▷ `char msg[]={'W','e','l','c','o','m','e','\0'};`

MAIN CHARACTERISTICS

▷ The empty string `""` corresponds to an array whose first element is `'\0'`

▷ The length of a string corresponds to the number of characters preceding `'\0'`

▷ Accessing to the nth character

```
char s[]="wxyz";
char c=s[2]; //c='y'
```

REMINDER ON STRINGS

▷ Thus `char tab[]` and `char *tab` are similar ... ⚠

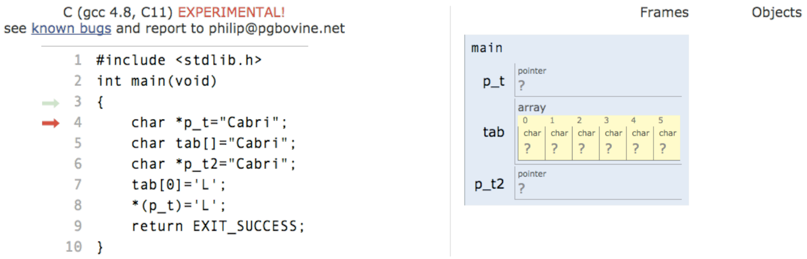
```
int main (void)
{
 char *tab="Cabri";
 tab[0]='L';
 return EXIT_SUCCESS;
}
```

```
int main (void)
{
 char tab[]="Cabri";
 tab[0]='L';
 return EXIT_SUCCESS;
}
```

▷ In the first case, `"Cabri"` is a constant string stored in memory (in a write protected segment since it is common to all the program) and `tab` "only" stores its address

▷ In the second case, it is the content of the string that is copied character by character into `tab`. The string is not stored elsewhere

REMINDER ON STRINGS



C (gcc 4.8, C11) EXPERIMENTAL!
see known bugs and report to philip@pgbovine.net

```
#include <stdlib.h>
int main(void)
{
    char *p_t="Cabri";
    char tab[]="Cabri";
    char *p_t2="Cabri";
    tab[0]='L';
    *(p_t)='L';
    return EXIT_SUCCESS;
}
```

POLYMORPHIC POINTER : VOID *

- ▷ Reminder : all pointers have the same size
- ▷ Type `void *`
  - ▷ represents any pointer
  - ▷ is compatible with all types of pointers
- ▷ Usefull to store or copy memory areas
- ▷ But does not allow any arithmetic operation
  - ▷ if p is such a pointer : `void *p = NULL;`
  - ▷ `*p` is illegal (no information on the pointed object)

POLYMORPHIC POINTER : VOID *

- ▷ Reminder : all pointers have the same size
- ▷ Type `void *`
  - ▷ represents any pointer
  - ▷ is compatible with all types of pointers
- ▷ Usefull to store or copy memory areas

```
void *memcpy(void* dst, void* src, size_t bytes)
{
    char *s = src;
    char *d = dst;
    while (bytes --)
        *d++ = *s++;
    return dst;
}
```