

COMP26020 Programming Languages and Paradigms Part 1: C Programming

---

# Pointers Applications

# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Arguments are passed by copy in C!

```
void add_one(int param) {  
    param++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // prints 20  
    add_one(x);  
    printf("after call, x is %d\n", x); // prints 20  
  
    return 0;  
}
```

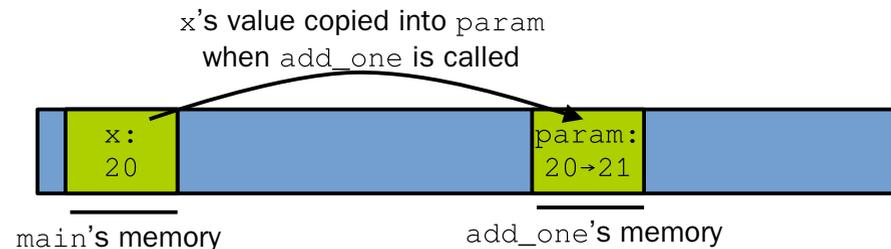
[10-pointers-applications/params-copy.c](#)  

# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Arguments are passed by copy in C!

```
void add_one(int param) {  
    param++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // prints 20  
    add_one(x);  
    printf("after call, x is %d\n", x); // prints 20  
  
    return 0;  
}
```

[10-pointers-applications/params-copy.c](https://github.com/pierrelolivier/10-pointers-applications/tree/master/params-copy.c) 



# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Use a pointer argument to pass the address as parameter

```
void add_one(int *param) {  
    (*param)++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // print 20  
    add_one(&x);  
    printf("after call, x is %d\n", x); // print 21  
  
    return 0;  
}
```

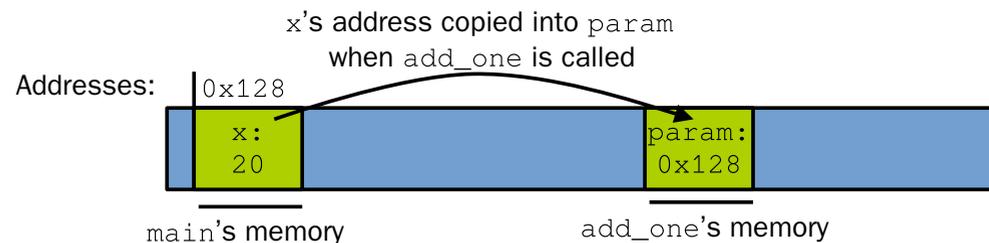
[10-pointers-applications/params-pointer.c](#)  

# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Use a pointer argument to pass the address as parameter

```
void add_one(int *param) {  
    (*param)++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // print 20  
    add_one(&x);  
    printf("after call, x is %d\n", x); // print 21  
  
    return 0;  
}
```

[10-pointers-applications/params-pointer.c](#)



# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Use a pointer argument to pass the address as parameter

```
void add_one(int *param) {  
    (*param)++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // print 20  
    add_one(&x);  
    printf("after call, x is %d\n", x); // print 21  
  
    return 0;  
}
```

[10-pointers-applications/params-pointer.c](#)



# Allow a Function to Access the Calling Context

- Want a function to 'change its parameters'?
  - Use a pointer argument to pass the address as parameter

```
void add_one(int *param) {  
    (*param)++;  
}  
  
int main(int argc, char **argv) {  
    int x = 20;  
  
    printf("before call, x is %d\n", x); // print 20  
    add_one(&x);  
    printf("after call, x is %d\n", x); // print 21  
  
    return 0;  
}
```

[10-pointers-applications/params-pointer.c](#)



# Allow a Function to Access the Calling Context

- Want a function to 'return' more than a single value?

```
// we want this function to return 3 things: the product and division of n1 by n2,  
// as well as an error code in case division is impossible  
int multiply_and_divide(int n1, int n2, int *product, int *division) {  
    if(n2 == 0) return -1;    // Can't divide if n2 is 0  
  
    *product = n1 * n2;  
    *division = n1 / n2;  
    return 0;  
}  
  
int main(int argc, char **argv) {  
    int p, d, a = 10, b = 5;  
    if(multiply_and_divide(a, b, &p, &d) == 0) {  
        printf("10*5 = %d\n", p); printf("10/5 = %d\n", d);  
    }  
}
```

[10-pointers-applications/params-pointers.c](#) 

# Allow a Function to Access the Calling Context

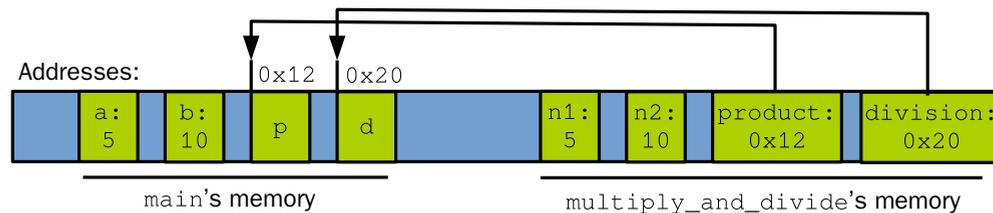
- Want a function to 'return' more than a single value?

```
// we want this function to return 3 things: the product and division of n1 by n2,
// as well as an error code in case division is impossible
int multiply_and_divide(int n1, int n2, int *product, int *division) {
    if(n2 == 0) return -1;    // Can't divide if n2 is 0

    *product = n1 * n2;
    *division = n1 / n2;
    return 0;
}

int main(int argc, char **argv) {
    int p, d, a = 10, b = 5;
    if(multiply_and_divide(a, b, &p, &d) == 0) {
        printf("10*5 = %d\n", p); printf("10/5 = %d\n", d);
    }
}
```

[10-pointers-applications/params-pointers.c](https://github.com/PierreOlivier10/10-pointers-applications/tree/master/params-pointers.c)

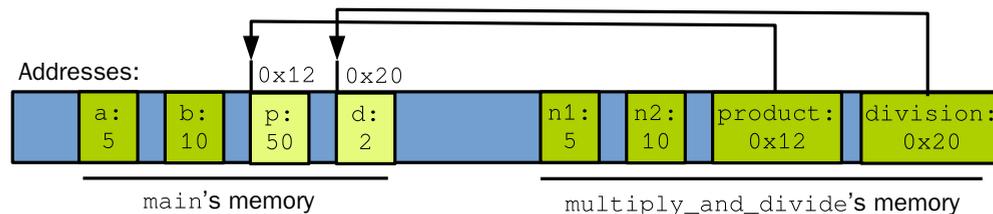


# Allow a Function to Access the Calling Context

- Want a function to 'return' more than a single value?

```
// we want this function to return 3 things: the product and division of n1 by n2,  
// as well as an error code in case division is impossible  
int multiply_and_divide(int n1, int n2, int *product, int *division) {  
    if(n2 == 0) return -1;    // Can't divide if n2 is 0  
  
    *product = n1 * n2;  
    *division = n1 / n2;  
    return 0;  
}  
  
int main(int argc, char **argv) {  
    int p, d, a = 10, b = 5;  
    if(multiply_and_divide(a, b, &p, &d) == 0) {  
        printf("10*5 = %d\n", p); printf("10/5 = %d\n", d);  
    }  
}
```

[10-pointers-applications/params-pointers.c](https://github.com/PierreOlivier10/10-pointers-applications/tree/master/params-pointers.c)



# Efficient Function Calls with Large Data Structures

- Assume we want to write a function updating a large struct variable

```
typedef struct {
    // lots of large (8 bytes) fields:
    double a; double b; double c; double d; double e; double f;
} large_struct;

large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!
    s.a += 42.0;
    return s;
}

int main(int argc, char **argv) {
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
    large_struct y = f(x);
    printf("y.a: %f\n", y.a);
}
```

[10-pointers-applications/large-param-copy.c](#) 

# Efficient Function Calls with Large Data Structures

- Assume we want to write a function updating a large struct variable

```
typedef struct {  
    // lots of large (8 bytes) fields:  
    double a; double b; double c; double d; double e; double f;  
} large_struct;  
  
large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!  
    s.a += 42.0;  
    return s;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};  
    large_struct y = f(x);  
    printf("y.a: %f\n", y.a);  
}
```

[10-pointers-applications/large-param-copy.c](#) 

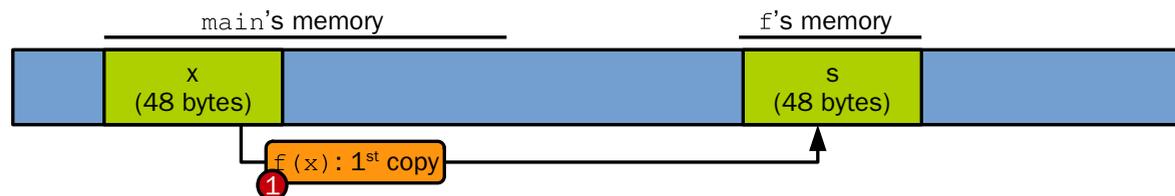


# Efficient Function Calls with Large Data Structures

- Assume we want to write a function updating a large struct variable

```
typedef struct {  
    // lots of large (8 bytes) fields:  
    double a; double b; double c; double d; double e; double f;  
} large_struct;  
  
large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!  
    s.a += 42.0;  
    return s;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};  
    large_struct y = f(x);  
    printf("y.a: %f\n", y.a);  
}
```

[10-pointers-applications/large-param-copy.c](https://github.com/ericniebler/10-pointers-applications/tree/master/large-param-copy.c)

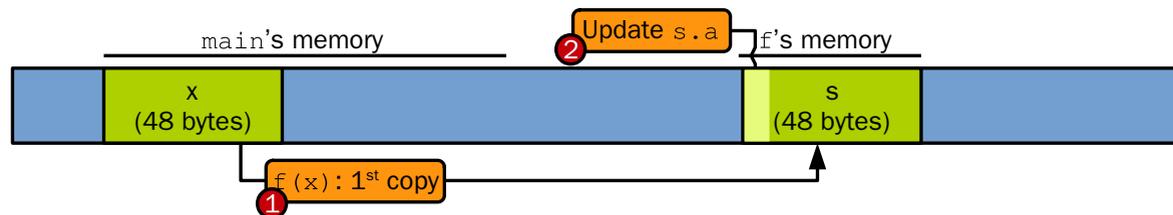


# Efficient Function Calls with Large Data Structures

- Assume we want to write a function updating a large struct variable

```
typedef struct {  
    // lots of large (8 bytes) fields:  
    double a; double b; double c; double d; double e; double f;  
} large_struct;  
  
large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!  
    s.a += 42.0;  
    return s;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};  
    large_struct y = f(x);  
    printf("y.a: %f\n", y.a);  
}
```

[10-pointers-applications/large-param-copy.c](https://github.com/ericniebler/10-pointers-applications/tree/master/large-param-copy.c)

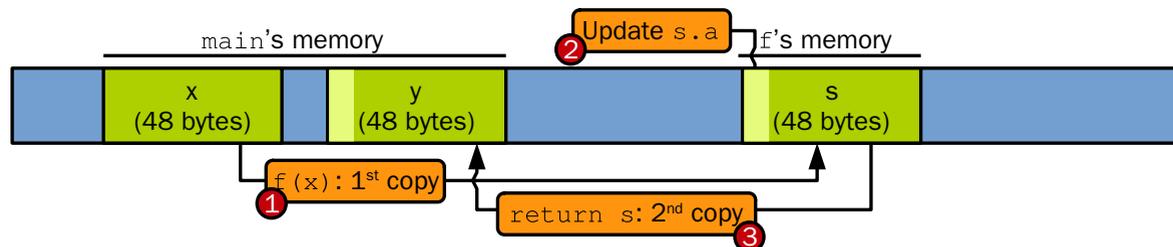


# Efficient Function Calls with Large Data Structures

- Assume we want to write a function updating a large struct variable

```
typedef struct {  
    // lots of large (8 bytes) fields:  
    double a; double b; double c; double d; double e; double f;  
} large_struct;  
  
large_struct f(large_struct s) { // very inefficient in terms of performance and memory usage!  
    s.a += 42.0;  
    return s;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};  
    large_struct y = f(x);  
    printf("y.a: %f\n", y.a);  
}
```

[10-pointers-applications/large-param-copy.c](https://github.com/10-pointers-applications/large-param-copy.c)



```

typedef struct {
    double a; double b;
    double c; double d;
    double e; double f;
} large_struct;

large_struct f(large_struct s) {
    s.a += 42.0;
    return s;
}

int main(int argc, char **argv) {
    large_struct x = {1, 2, 3, 4, 5, 6};
    large_struct y = f(x);
    printf("y.a: %f\n", y.a);
    return 0;
}

```

- Code decompiled with `objdump`  
`--source`
- **Huge performance loss and memory footprint increase!**

```

int main(int argc, char **argv) { /* ... */
    large_struct y = f(x);
11e9: 48 8d 45 a0 lea    -0x60(%rbp),%rax
11ed: ff 75 f8     pushq  -0x8(%rbp)
11f0: ff 75 f0     pushq  -0x10(%rbp)
11f3: ff 75 e8     pushq  -0x18(%rbp)
11f6: ff 75 e0     pushq  -0x20(%rbp)
11f9: ff 75 d8     pushq  -0x28(%rbp)
11fc: ff 75 d0     pushq  -0x30(%rbp)
11ff: 48 89 c7     mov    %rax,%rdi
1202: e8 2e ff ff callq  1135 <f>
1207: 48 83 c4 30 add    $0x30,%rsp
    /* ... */
}

```

```

large_struct f(large_struct s) {
    /* ... */
    return s;
1153: 48 8b 45 f8 mov    -0x8(%rbp),%rax
1157: 48 8b 55 10 mov    0x10(%rbp),%rdx
115b: 48 8b 4d 18 mov    0x18(%rbp),%rcx
115f: 48 89 10     mov    %rdx,(%rax)
1162: 48 89 48 08 mov    %rcx,0x8(%rax)
1166: 48 8b 55 20 mov    0x20(%rbp),%rdx
116a: 48 8b 4d 28 mov    0x28(%rbp),%rcx
116e: 48 89 50 10 mov    %rdx,0x10(%rax)
1172: 48 89 48 18 mov    %rcx,0x18(%rax)
1176: 48 8b 55 30 mov    0x30(%rbp),%rdx
117a: 48 8b 4d 38 mov    0x38(%rbp),%rcx
117e: 48 89 50 20 mov    %rdx,0x20(%rax)
1182: 48 89 48 28 mov    %rcx,0x28(%rax)
}

```

# Efficient Function Calls with Large Data Structures

- With a pointer: maintain a single copy of the variable

```
typedef struct { double a; double b; double c; double d; double e; double f;} large_struct;

void f(large_struct *s) { // now takes a pointer parameter
    (*s).a += 42.0;      // dereference to access x
    return;
}

int main(int argc, char **argv) {
    large_struct x = {1, 2, 3, 4, 5, 6};
    f(&x);              // pass x's address
    printf("x.a: %f\n", x.a);
    return 0;
}
```

[10-pointers-applications/large-param-pointer.c](https://github.com/olivier-pierre/10-pointers-applications/blob/master/large-param-pointer.c) 

# Efficient Function Calls with Large Data Structures

- With a pointer: maintain a single copy of the variable

```
typedef struct { double a; double b; double c; double d; double e; double f;} large_struct;  
  
void f(large_struct *s) { // now takes a pointer parameter  
    (*s).a += 42.0;      // dereference to access x  
    return;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1, 2, 3, 4, 5, 6};  
    f(&x);              // pass x's address  
    printf("x.a: %f\n", x.a);  
    return 0;  
}
```

[10-pointers-applications/large-param-pointer.c](https://github.com/10-pointers-applications/large-param-pointer.c)

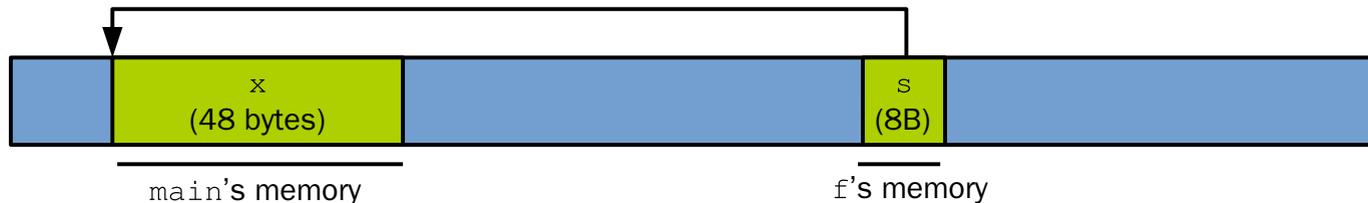


# Efficient Function Calls with Large Data Structures

- With a pointer: maintain a single copy of the variable

```
typedef struct { double a; double b; double c; double d; double e; double f;} large_struct;  
  
void f(large_struct *s) { // now takes a pointer parameter  
    (*s).a += 42.0;      // dereference to access x  
    return;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1, 2, 3, 4, 5, 6};  
    f(&x);              // pass x's address  
    printf("x.a: %f\n", x.a);  
    return 0;  
}
```

[10-pointers-applications/large-param-pointer.c](https://10-pointers-applications/large-param-pointer.c)



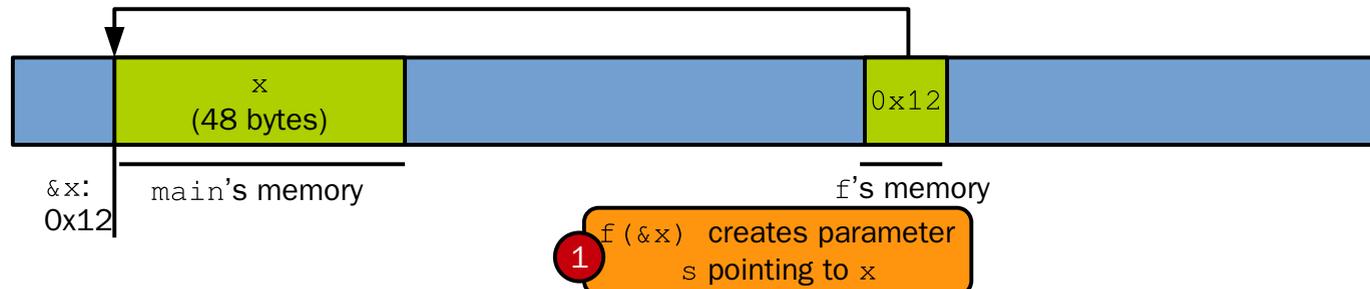
1 f(&x) creates parameter s pointing to x

# Efficient Function Calls with Large Data Structures

- With a pointer: maintain a single copy of the variable

```
typedef struct { double a; double b; double c; double d; double e; double f;} large_struct;  
  
void f(large_struct *s) { // now takes a pointer parameter  
    (*s).a += 42.0;      // dereference to access x  
    return;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1, 2, 3, 4, 5, 6};  
    f(&x);              // pass x's address  
    printf("x.a: %f\n", x.a);  
    return 0;  
}
```

[10-pointers-applications/large-param-pointer.c](https://10-pointers-applications/large-param-pointer.c)

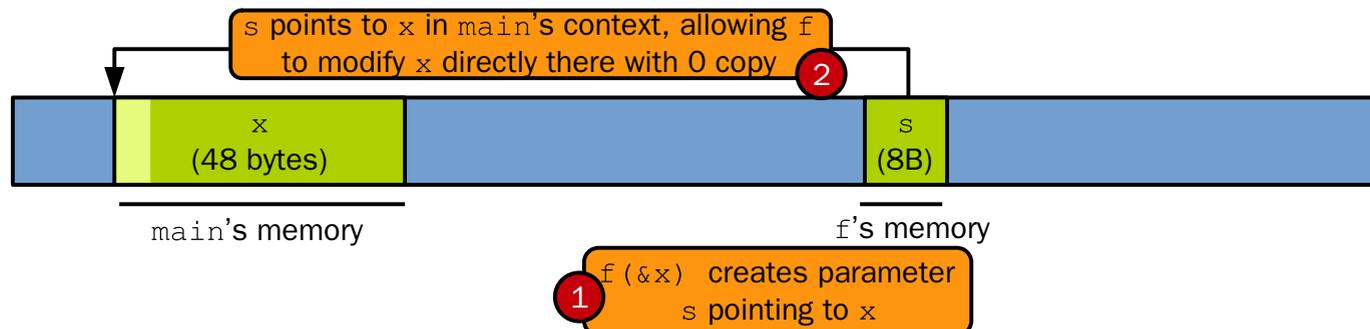


# Efficient Function Calls with Large Data Structures

- With a pointer: maintain a single copy of the variable

```
typedef struct { double a; double b; double c; double d; double e; double f;} large_struct;  
  
void f(large_struct *s) { // now takes a pointer parameter  
    (*s).a += 42.0;      // dereference to access x  
    return;  
}  
  
int main(int argc, char **argv) {  
    large_struct x = {1, 2, 3, 4, 5, 6};  
    f(&x);              // pass x's address  
    printf("x.a: %f\n", x.a);  
    return 0;  
}
```

[10-pointers-applications/large-param-pointer.c](https://github.com/10-pointers-applications/large-param-pointer.c)



```

typedef struct {
    double a; double b;
    double c; double d;
    double e; double f;
} large_struct;

void f(large_struct *s) {
    (*s).a += 42.0;
    return;
}

int main(int argc, char **argv) {
    large_struct x = {1, 2, 3, 4, 5, 6};
    f(&x);
    printf("x.a: %f\n", x.a);
    return 0;
}

```

- **With pointers we get code that is much faster and memory efficient**

```

int main(int argc, char **argv) {
    /* ... */

    f(&x);
    11b9: 48 8d 45 d0  lea -0x30(%rbp),%rax
    11bd: 48 89 c7      mov %rax,%rdi
    11c0: e8 70 ff ff  callq 1135 <f>

    /* ... */
}

```

```

void f(large_struct *s) {
    /* ... */

    return;
    1159: 90          nop

}
    115a: 5d          pop    %rbp
    115b: c3          retq

```

# Misc. Pointers-related Topics

# C Arrays are Pointers

- Under the hood arrays are pointers
- To pass an array as parameter or return an array, use a pointer of the array's type

```
void negate_int_array(int *ptr, int size) { // function taking pointer as parameter
    for(int i=0; i<size; i++)             // also need the size to iterate properly
        ptr[i] = -ptr[i];                // use square brackets like a standard array
}

int main(int argc, char **argv) {
    int array[] = {1, 2, 3, 4, 5, 6, 7};

    negate_int_array(array, 7); // to get the pointer just use the array's name
    for(int i=0; i<7; i++)
        printf("array[%d] = %d\n", i, array[i]);

    return 0;
}
```

[10-pointers-applications/arrays-pointers.c](#) 

# C Arrays are Pointers

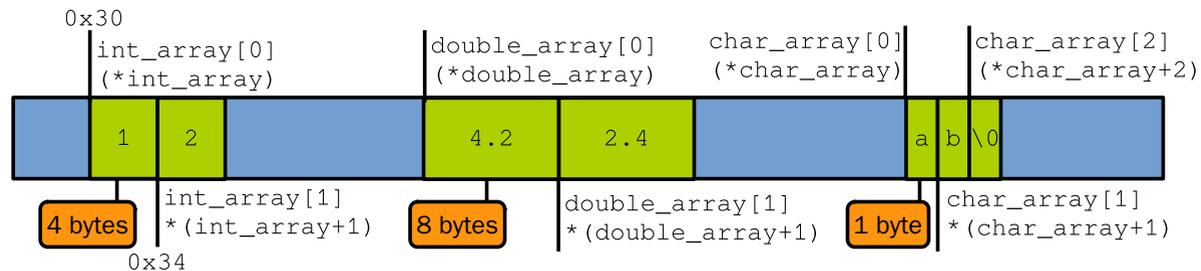
```
int int_array[2] = {1, 2};
double double_array[2] = {4.2, 2.4};
char char_array[] = "ab";

printf("int_array[0]      = %d\n", int_array[0]);
printf("int_array[1]      = %d\n", int_array[1]);
printf("*(int_array+0)     = %d\n", *(int_array+0)); // pointer arithmetic!
printf("*(int_array+1)     = %d\n", *(int_array+1)); // +1 means + sizeof(array type) bytes

printf("double_array[0]    = %f\n", double_array[0]);
printf("double_array[1]    = %f\n", double_array[1]);
printf("*(double_array+0)   = %f\n", *(double_array+0));
printf("*(double_array+1)   = %f\n", *(double_array+1));

printf("char_array[0]        = %c\n", char_array[0]);
printf("char_array[1]        = %c\n", char_array[1]);
printf("*(char_array+0)      = %c\n", *(char_array+0));
printf("*(char_array+1)      = %c\n", *(char_array+1));
```

[10-pointers-applications/arrays-pointers-indexing.c](#)



# Pointers and Structures

```
typedef struct {
    int int_member1;
    int int_member2;
    int *ptr_member;
} my_struct;

my_struct ms = /* ... declaration and initialisation of ms omitted for space reasons */

my_struct *p = &ms;

(*p).int_member1 = 1; // don't forget the parentheses! . takes precedence over *
p->int_member2 = 2; // s->x is a shortcut for (*s).x

p->ptr_member = &(p->int_member2);

printf("p->int_member1 = %d\n", p->int_member1);
printf("p->int_member2 = %d\n", p->int_member2);
printf("p->ptr_member = %p\n", p->ptr_member);
printf("*(p->ptr_member) = %d\n", *(p->ptr_member));
```

[10-pointers-applications/pointers-structs.c](#) 

# Pointers and Structures

```
typedef struct {
    int int_member1;
    int int_member2;
    int *ptr_member;
} my_struct;

my_struct ms = /* ... declaration and initialisation of ms omitted for space reasons */

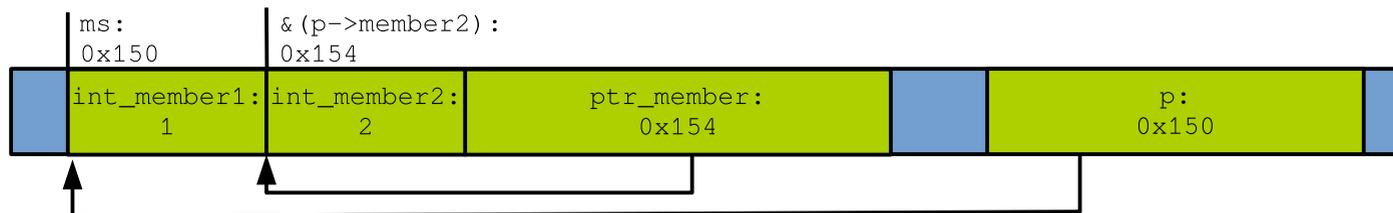
my_struct *p = &ms;

(*p).int_member1 = 1; // don't forget the parentheses! . takes precedence over *
p->int_member2 = 2; // s->x is a shortcut for (*s).x

p->ptr_member = &(p->int_member2);

printf("p->int_member1 = %d\n", p->int_member1);
printf("p->int_member2 = %d\n", p->int_member2);
printf("p->ptr_member = %p\n", p->ptr_member);
printf("*(p->ptr_member) = %d\n", *(p->ptr_member));
```

[10-pointers-applications/pointers-structs.c](#)



# Pointer Chains

- A pointer is a variable and has itself an address, i.e. a location in memory
  - So it can be pointed to!

```
int value = 42;           // integer
int *ptr1 = &value;      // pointer of integer
```

```
printf("ptr1: %p, *ptr1: %d\n", ptr1, *ptr1);
```

```
//
```

[10-pointers-applications/pointer-chains.c](https://github.com/10-101/10-pointers-applications/blob/master/pointer-chains.c)



# Pointer Chains

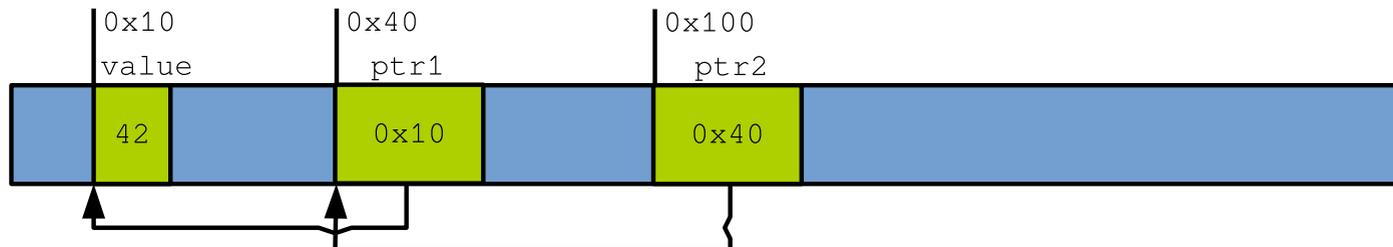
- A pointer is a variable and has itself an address, i.e. a location in memory
  - So it can be pointed to!

```
int value = 42;           // integer
int *ptr1 = &value;      // pointer of integer
int **ptr2 = &ptr1;      // pointer of pointer of integer

printf("ptr1: %p, *ptr1: %d\n", ptr1, *ptr1);
printf("ptr2: %p, *ptr2: %p, **ptr2: %d\n", ptr2, *ptr2, **ptr2);

//
```

[10-pointers-applications/pointer-chains.c](https://github.com/PierreOlivier/10-pointers-applications/blob/master/pointer-chains.c)



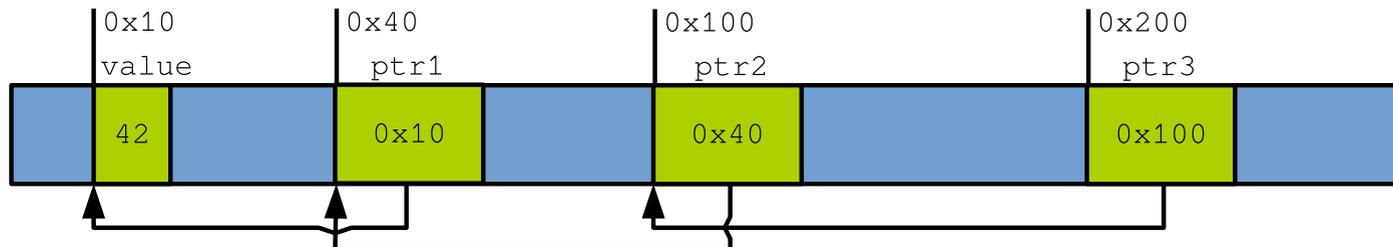
# Pointer Chains

- A pointer is a variable and has itself an address, i.e. a location in memory
  - So it can be pointed to!

```
int value = 42;           // integer
int *ptr1 = &value;      // pointer of integer
int **ptr2 = &ptr1;      // pointer of pointer of integer
int ***ptr3 = &ptr2;    // pointer of pointer of pointer of integer
```

```
printf("ptr1: %p, *ptr1: %d\n", ptr1, *ptr1);
printf("ptr2: %p, *ptr2: %p, **ptr2: %d\n", ptr2, *ptr2, **ptr2);
printf("ptr3: %p, *ptr3: %p, **ptr3: %p, ***ptr3: %d\n", ptr3, *ptr3,
      **ptr3, ***ptr3);
```

[10-pointers-applications/pointer-chains.c](https://github.com/olivier-pierre/10-pointers-applications/tree/master/pointer-chains.c)



# Summary

- Pointers use cases
    - Modify a function calling context
    - For it to 'return' more than a single value
    - Avoid costly data copy on function calls
  - Arrays and data structures relationship with pointers
  - Pointer chains
- 

Feedback form: <https://bit.ly/2WZ60QG>

