Modern C on Sea

Dawid Zalewski

Off we go!

a.k.a. What is it all about?

- Modern C
- Modern == (C >= C99)
- For (seasoned) C++ developers
- Who learned C from K&R (and never updated 😳)
- Or who never learned C
- Or who heard something about VLA, compound literals, etc... and got curious
- ...
- For anyone who wants to get up-to-date with modern C

The flight plan!

a.k.a. What will we do?

- Initialization & designated initializers
- Variable length arrays
- Passing pointers and arrays to functions
- Compound literals
- Flexible array members
- Anonymous class="inline-key">structs and class="inline-key">unions
- Don't be afraid of values
- Generic selection (macro magic)
- ...
- With a lots of rehearshal and best practices in between

C for C++ programmers

in three slides

C for C++ programmers

in three slides

Functions that don't take arguments have an explicit `void` parameter

```
1  int no_args( void ){
2
3    return 42;
4
5  }
```

`auto` exists in C and specifies automatic storage duration

```
1  int auto_vars( int n ){
2
3    auto int squared = n * n;
4    auto min_one = squared - 1;
5    return min_one;
6
7  }
```

side note: variables with no type are implicitly `int`

C for C++ programmers

in three slides

Casting pointer types to and from `void*` is not needed

```c
void func(void){

  double* arr = malloc(sizeof(double[N]));

}
```

For Boolean values the type `_Bool` can be used

```c
#include <stdbool.h>

int get_num( _Bool random ){
  return random? rand() : 42;
}

bool random = false;
int num = get_num(random);
```

C for C++ programmers

in three slides

Type punning in C is supported through `union` types

```c
union dbl_bytes{
  double number;
  unsigned char bytes[sizeof(double)];
};

int main(void){
  union dbl_bytes db = {42.0};
  for (size_t i=0; i < sizeof(db.bytes); ++i)
    printf("0x%02x ", db.bytes[i]);

}
```

In C++ the size of a `char` literal is 1

In C it is: `sizeof('B') == sizeof(int)`

	C	C++
`sizeof(int)`	4	4
`sizeof(42)`	4	4
`sizeof(char)`	1	1
`sizeof('B')`	4	1

Initialization

Let's start with a (semi-)quiz!

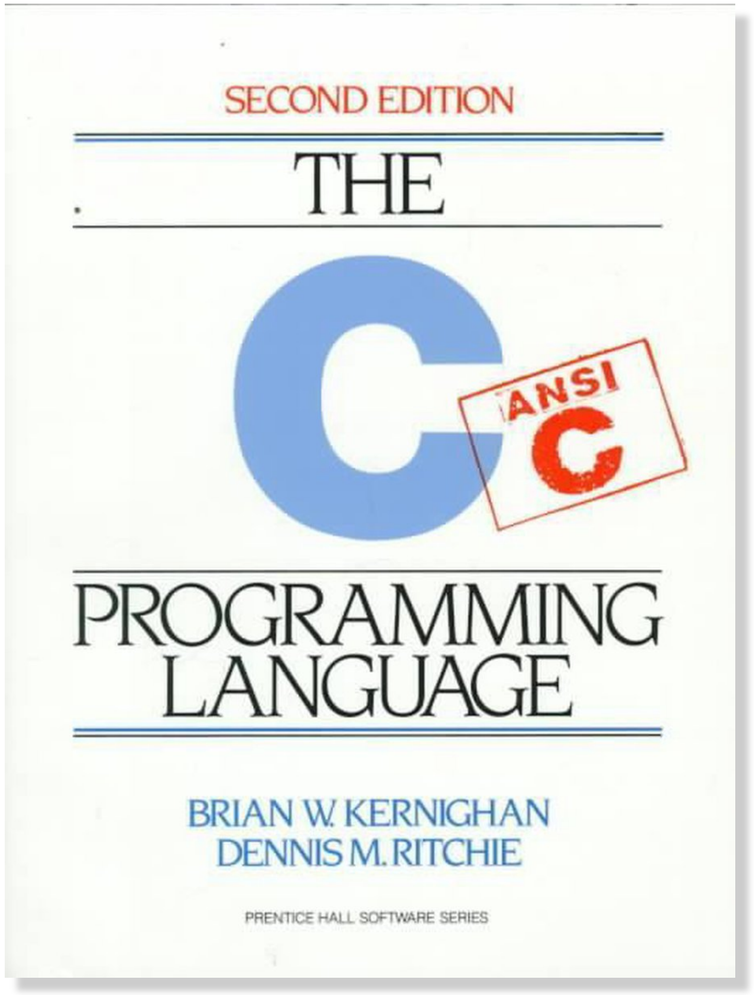
Exhibit #1

```
1  typedef struct point {
2    int x;
3    int y;
4  } point_s;
5
6  point_s makepoint(int x, int y) {
7    point_s temp;
8    temp.x = x;
9    temp.y = y;
10   return temp;
11 }
12
13 point_s p = makepoint(42, 24);
```

Do you ever write code like this?

Where does this piece come from?

The source of all C!



How to initialize everything

K&R C

```
 1 typedef struct point {
 2   int x;
 3   int y;
 4 } point_S;
 5
 6 point_s makepoint(int x, int y) {
 7   point_s temp;
 8   temp.x = x;
 9   temp.y = y;
10   return temp;
11 }
12
13 point_s p = makepoint(42, 24);
```

Up-to-date C

```
 1 typedef struct point {
 2   int x;
 3   int y;
 4 } point_s;
 5
 6 point_s makepoint(int x, int y) {
 7
 8   point_s tmp = {x, y};
 9   return tmp;
10
11 }
12
13 point_s p = makepoint(42, 24);
```

How to initialize everything

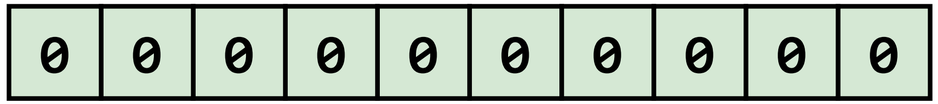
In C a variable of any type can be initialized using: { }

```c
struct data {
  const char* str;
  int value;
};

// array contains two elements
int array[] = {42, 24};

// numbers contains 10 elements, the first three are: 2, 3, 5
double numbers[10] = {2, 3, 5};

// my_data.str == "alice"
// my_data.value = 123
struct data my_data = {"alice", 123};
```

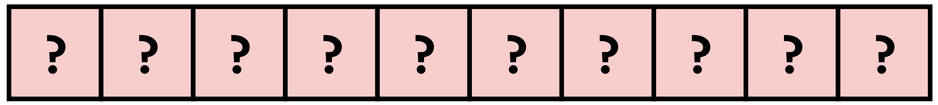
How to initialize an array

```
double numbers[10];
```

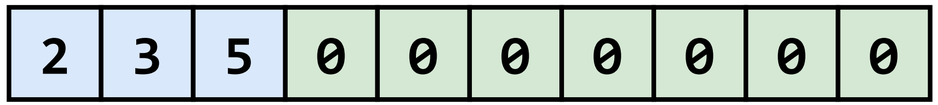
static storage



automatic storage




```
double numbers[10] = { 2, 3, 5 };
```



How to initialize everything

In C a variable of any type can be initialized using: `{ }`

Really of any type:

```
1 double pi = { 3.1415927 };
```

How to default-initialize everything

There's also a shortcut that ***default-initializes*** any type

```
Type name = { 0 };
```

It works for scalars, arrays, `struct`s and `union`s

```
struct data {
  const char* str;
  int value;
};

double value = { 0 };
double numbers[42] = { 0 };
struct data my_data = { 0 };
```

Default-initialization rules

Default-initialization makes sense for objects with *automatic storage duration*

For scalars it just initializes a variable with 0

```
1 double value = { 0 };     // value == 0
2 void *ptr = { 0 };        // ptr == NULL
3 const char* str = { 0 }; // str == ""
```

For arrays it sets all the items to 0

```
1 double numbers[42] = { 0 }; // numbers contains 42 0's
2 int values[] = { 0 };       // values is an array with 1 zero
```

Default initialization rules

Default-initialization makes sense for objects with *automatic storage duration*

For `struct`s it zeroes all the members

```
 1 struct data {
 2   const char* str;
 3   int value;
 4 };
 5
 6 struct data my_data = { 0 };
 7
 8 /*
 9 * my_data.str == ""
10 * my_data.value == 0
11 */
```

`struct data dt;`

`const char* str;`	?	?	?	?	?	?	?	?
`int value;`	?	?	?	?	?	?	?	?

`struct data dt = { 0 };`

`const char* str;`	0	0	0	0	0	0	0	0
`int value;`	0	0	0	0	0	0	0	0

padding bytes

Default initialization rules

Default-initialization makes sense for objects with *automatic storage duration*

This works also for nested types

```
1  struct data {
2    const char* str;
3    int value;
4  };
5
6  struct data my_data_array[42] = { 0 };
7
8  /*
9  * my_data_array[0..41].str == ""
10 * my_data_array.value[0..41] == 0
11 */
```

What if we want to be selective?

```
1 typedef struct driver {
2   const char* name;
3   unsigned char data[16];
4   uint16_t flags;
5   uint16_t flags_ex;
6   status_f status;
7 } driver_s;
```

In 95% of cases the `data`, `flags` and `flags_ex` fields are zeroed.

But:

```
drivers_s my_serial = {"serial", {0}, 0, 0, &status_serial};
```

Ouch, do you see those `{0}, 0, 0`?

Designated initializers to the rescue!

A designated initializer takes the form:

```
{ .field_name = value }
```

To initialize only the `name` and `status` fields:

```
1 typedef struct driver {
2   const char* name;
3   unsigned char data[16];
4   uint16_t flags;
5   uint16_t flags_ex;
6   status_f status;
7 } driver_s;
8
9 driver_s my_serial = {.name="serial", .status=&status_serial};
```

All the remaining fields are zero-initialized

Designated initializers: order

The order of designated initializers can be arbitrary:

```
 1 typedef struct driver {
 2   const char* name;
 3   unsigned char data[16];
 4   uint16_t flags;
 5   uint16_t flags_ex;
 6   status_f status;
 7 } driver_s;
 8
 9 driver_s my_serial = {.status=&status_serial, .name="serial"};
10 driver_s my_serial = {.name="serial", .status=&status_serial};
```

Designated initializers: mixing

The *positional* and designated initializers can be freely mixed

```
1 typedef struct driver {
2   const char* name;
3   unsigned char data[16];
4   uint16_t flags;
5   uint16_t flags_ex;
6   status_f status;
7 } driver_s;
8
9 driver_s my_serial = {"serial", .status=&status_serial};
```

But make sure that you know what you are doing...

Designated initializers: mixing

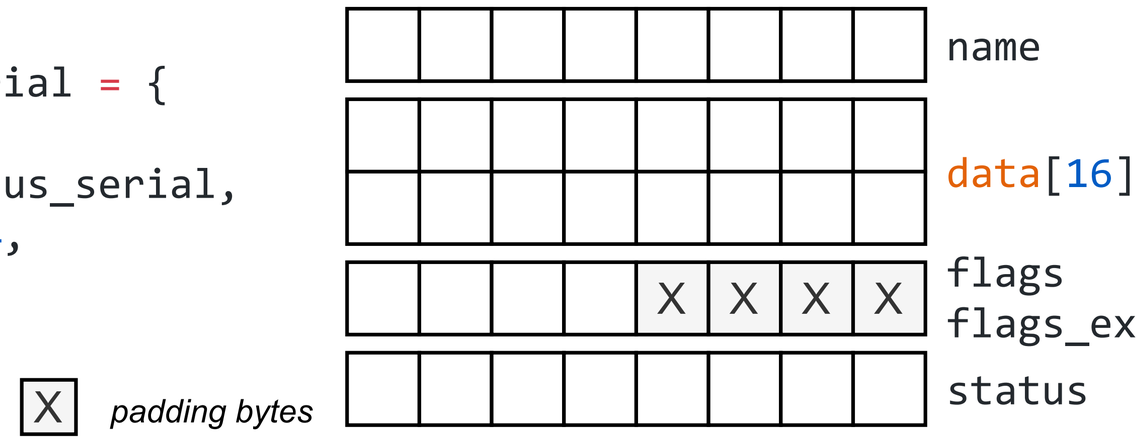
```
1 typedef struct driver {
2   const char* name;
3   unsigned char data[16];
4   uint16_t flags;
5   uint16_t flags_ex;
6   status_f status;
7 } driver_s;
8
9 driver_s my_serial = {"serial", .status=&status_serial, .flags=0x1234, 0x4321};
```

This ↑ is equivalent to ↓

```
driver_s my_serial = {
  .serial = "serial",
  .data = { 0 },
  .flags = 0x1234,
  .flags_ex = 0x4321,
  .status = &status_serial
};
```

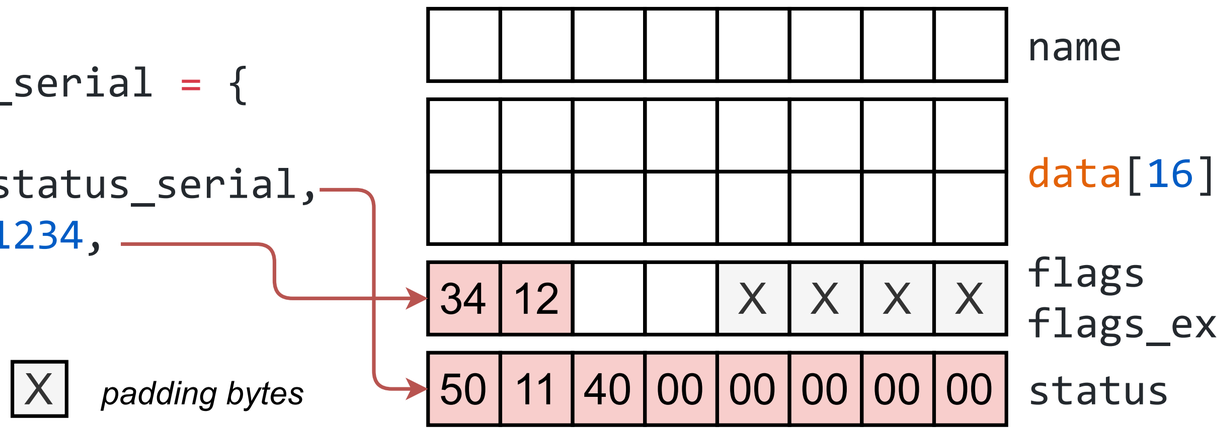
Designated initializers: mixing

```
driver_s my_serial = {
  "serial",
  .status=&status_serial,
  .flags=0x1234,
  0x4321
};
```

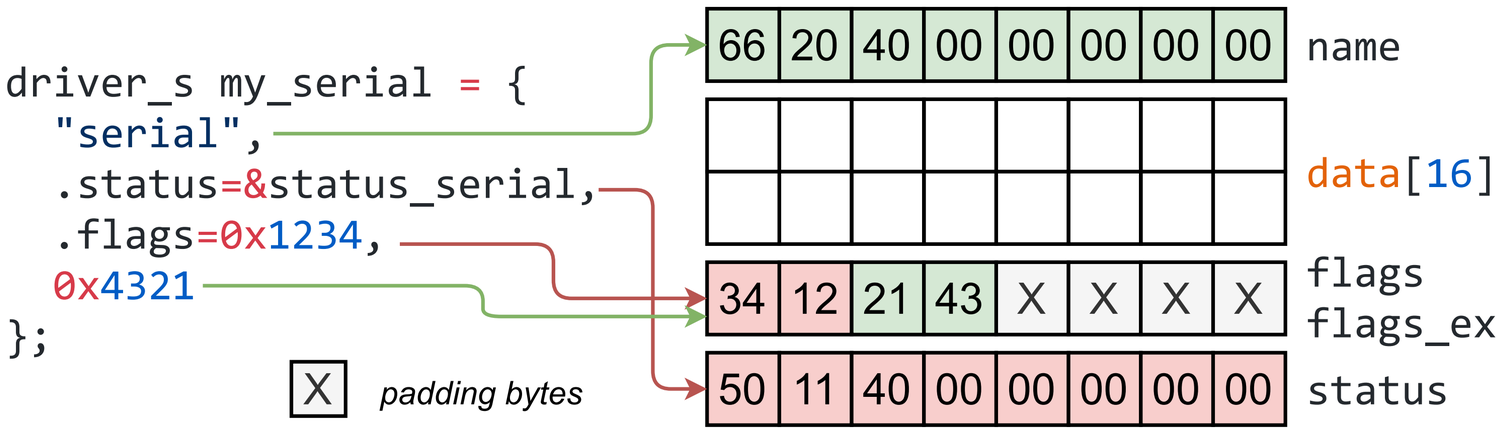


Designated initializers: mixing

```
driver_s my_serial = {
  "serial",
  .status=&status_serial,
  .flags=0x1234,
  0x4321
};
```

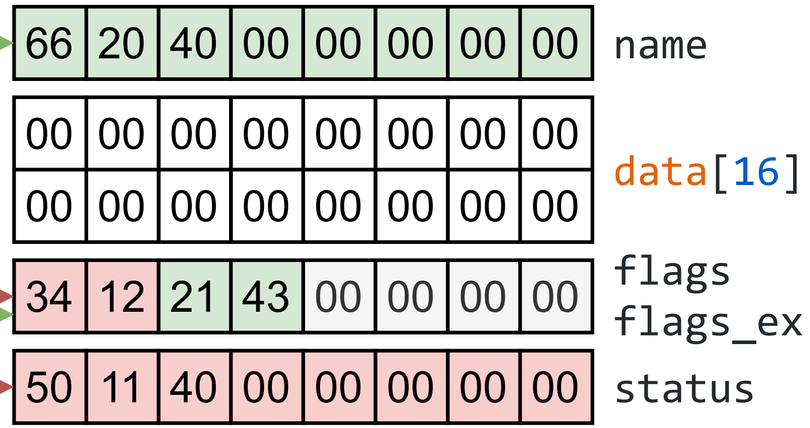


Designated initializers: mixing



Designated initializers: mixing

```
driver_s my_serial = {
  "serial",
  .status=&status_serial,
  .flags=0x1234,
  0x4321
};
```



Designated initializers: nested structures

Multiple nesting styles are supported

```
 1 typedef struct flags {
 2   uint16_t standard;
 3   uint16_t extended;
 4 } flags_s;
 5
 6 typedef struct driver {
 7   const char* name;
 8   unsigned char data[16];
 9   flags_s flags;
10   status_f status;
11 } driver_s;
```

Nested positional

```
driver_s my_serial = {
  "serial",
  .status= &status_serial,
  .flags= { 0x1234, 0x4321 }
};
```

Nested designated

```
driver_s my_serial = {
  "serial",
  .status= &status_serial,
  .flags={ .extended=0x4321 }
};
```

⚡ *Super nested* ⚡

```
driver_s my_serial = {
  "serial",
  .status= &status_serial,
  .flags.extended = 0x4321
};
```

Designated initializers: arrays

Arrays support designated initializers by subscript (index)

```
1 double numbers[42] = { [1]=1, [5]=5, [10]=55};
```

As with `structs`, items not initialized directly are *zero-initialized*

```
1 double numbers[42] = { [1]=1, [5]=5, [10]=55};
```

The snippet above ↑ is equivalent to ↓

```
1 // numbers at indices > 10 are set to 0
2 double numbers[42] = { 0, 1, 0, 0, 0, 5, 0, 0, 0, 0, 55 };
```

Designated initializers: arrays

As with `struct`s initialization order is arbitrary

```
1 double numbers[42] = { [10]=55, [5]=5, [1]=1 };
2
3 double numbers[42] = { [5]=5, [10]=55, [1]=1 };
```

And positional and designated initializers can be freely mixed

```
1 double numbers[42] = {0, [10]=55, 89, [5]=5, 8, [1]=1, 1, 2};
```

The snippet above ↑ is equivalent to ↓

```
1 // numbers at indices > 11 are set to 0
2 double numbers[42] = { 0, 1, 1, 2, 0, 5, 8, 0, 0, 0, 55, 89 };
```

Array of unknown size

For arrays with unknown size, the largest designator subscript is used to determine it

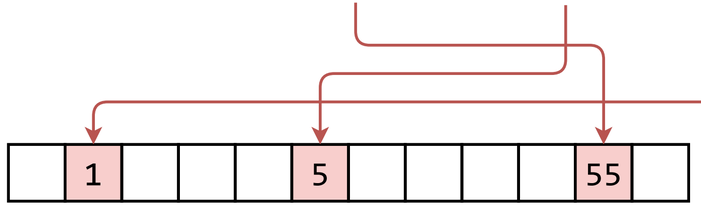
```
1 double numbers[] = {0, [10]=55, 89, [5]=5, 8, [1]=1, 1, 2};
```

`numbers` has type `double[12]`

Array of unknown size

For arrays with unknown size, the largest designator subscript is used to determine it

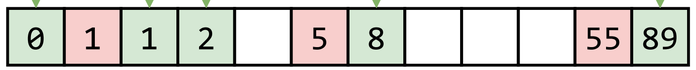
```
double numbers[] = {0, [10]=55, 89, [5]=5, 8, [1]=1, 1, 2};
```



Array of unknown size

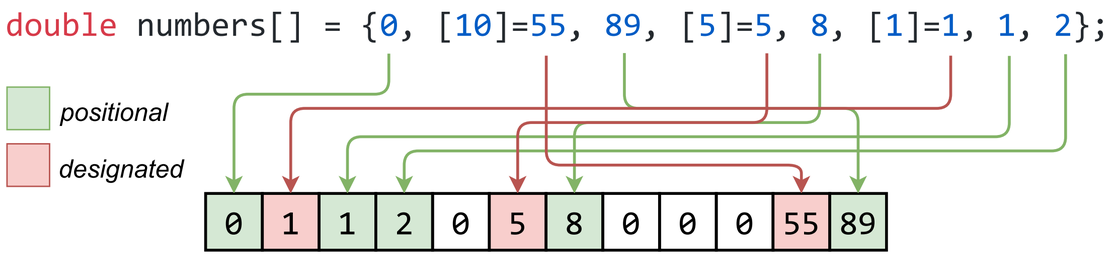
For arrays with unknown size, the largest designator subscript is used to determine it

```
double numbers[] = {0, [10]=55, 89, [5]=5, 8, [1]=1, 1, 2};
```



Array of unknown size

For arrays with unknown size, the largest designator subscript is used to determine it



Designated initializers C vs. C++

C++ is not even half that flexible as C

What	C	C++
`driver_s my_serial = { .flags = {0x4321}, .name="" };`	✓	✗ nope, out of order
`driver_s my_serial = { .flags = {0x4321}, 0 };`	✓	✗ nope, mixed
`driver_s my_serial = { .flags.extended = 0x4321 };`	✓	✗ nope, nesting
`double numbers[] = {[10] = 55};`	✓	✗ nope: no array designated initializers

Arrays

What about arrays?

With constant known size

```
1 #define A_LOT (100)
2
3 int numbers[42];
4 struct tm dates[A_LOT];
5 unsigned char view[sizeof(double[4])];
```

With dynamic size

```
1 size_t len = rand();
2 size_t sz = sizeof(double);
3
4 double* roulette_1 = malloc(len *  sz);
5 double* roulette_2 = calloc(len, sz);
```

**You should use `calloc` - it zeroes the allocated block*

What about constant known size?

Q: Does `SIZE` name a ***constant known size*** ?

```
1 static const size_t SIZE = 100;
2
3 void func(){
4
5   double numbers[SIZE];
6
7 }
```

A: No

So why does this code compile?

Variable Length Arrays (VLA)

Variable Length Arrays are arrays whose size is **not known** at compile time.

They can only appear at block scope and in function prototypes

```c
size_t size = 100;

// nope - file scope
double oh_no[size];

// ok - function prototype
void filter(size_t n, double arr[n], size_t len){

  // ok - block scope
  double buffer[len];

}
```

VLAs - initialization

Remember this slide?

How to initialize everything

In C a variable of any type can be initialized using: { }

Really of any type:

```
1 double pi = { 3.1415927 };
```

VLAs - initialization

That wasn't, technically speaking, the absolute truth...

VLAs are the only types that cannot be initialized using the brace-enclosed list

```
1 void filter(size_t n, double arr[n], size_t len){
2
3   // error - won't compile
4   double buffer[len] = { arr[0], arr[1], arr[2] };
5
6   // error - won't compile either
7   double another_buffer[len] = { 0 };
8
9 }
```

VLAs - initialization

VLAs need to be initialized after declaration.

```
1 void filter(size_t n, double arr[n], size_t len){
2
3   double buffer[len];
4
5   for (size_t i = 0; i < len; ++i)
6     buffer[i] = arr[i];
7
8 }
```

VLAs - controversies

*USING VLA'S IS ACTIVELY STUPID! It generates much more code, and much **slower** code (and more fragile code), than just using a fixed (array) size would have done.*

Linus Torvalds, 7 March 2018

VLAs - the overhead

The static size variant

```
1 int test_known_size(){
2   int arr[10];
3
4   init(10, arr);
5   return sum(10, arr);
6 }
```

The VLA variant

```
1 int test_vla(size_t n){
2   int arr[n];
3
4   init(n, arr);
5   return sum(n, arr);
6 }
```

VLAs - the overhead

The static size variant

```
 1  test_known_size:
 2    push    rbp
 3    mov     rbp, rsp
 4    sub     rsp, 48
 5
 6    ; init & sum calls
 7
 8    add     rsp, 48
 9    pop     rbp
10    ret
```

`clang -std=c11 -O0`

The VLA variant

```
 1  test_vla:
 2    push    rbp
 3    mov     rbp, rsp
 4    mov     rax, rsp
 5    lea     rdx, [4*rdi + 15]
 6    and     rdx, -16
 7    mov     rcx, rax
 8    sub     rcx, rdx
 9    mov     rsp, rcx
10
11    ; init & sum calls
12
13    mov     rsp, rbp
14    pop     rbp
15    ret
```

`clang -std=c11 -O1`

VLAs - the overhead

The static size variant

```
1 int test_known_size(){
2   int arr[10];
3
4   init(10, arr);
5   return sum(10, arr);
6 }
```

The VLA variant

```
1 int test_vla(size_t n){
2   int arr[n];
3
4   init(n, arr);
5   return sum(n, arr);
6 }
```

VLA vs. modern optimizers

Compiler: `clang -O3` (trunk)

```
 1 void init(size_t n, int *arr){
 2   for (size_t i = 0; i < n; ++i)
 3   arr[i] = (i+1) * (i+1);
 4 }
 5
 6 #define TEST_VLA (0)
 7
 8 int main(void){
 9   int n = 10;
10   return TEST_VLA?
11   test_vla(n) :
12   test_known_size();
13 }
```

With `#define TEST_VLA (0)`

```
1 main:
2   mov     eax, 385
3   ret
```

With `#define TEST_VLA (1)`

```
1 main:
2   mov     eax, 385
3   ret
```

VLAs - why the critique?

VLAs are (almost always) allocated on the stack

This can potentially overflow the stack

Solution?

```
1  #define MAX_FILTER_LEN (128)
2
3  status_e filter(size_t n, double *arr, size_t len){
4    if (len > MAX_FILTER_LEN)
5      return status_fail;
6
7    double buffer[len];
8    for (size_t i=0; i < len; ++i)
9      buffer[i] = arr[i];
10   // ...
11 }
```

VLAs - why the critique?

VLAs are (almost always) allocated on the stack

This can potentially overflow the stack

Solution?

```
1  #define MAX_FILTER_LEN (128)
2
3  status_e filter(size_t n, double *arr, size_t len){
4    if (len > MAX_FILTER_LEN)
5      return status_fail;
6
7    double buffer[MAX_FILTER_LEN];
8    for (size_t i=0; i < len; ++i)
9      buffer[i] = arr[i];
10   // ...
11 }
```

VLAs - why the critique?

- VLAs are a poor replacement of *normal* arrays
- Usually it's well known upfront what the maximum size can be
- It's easier and better (performance) to just use this know maximum size
  - No stack overflow danger
  - No extra checks
  - No bookkeeping

- ...those are also the reasons for no VLAs in C++

VLAs as arguments to functions

The VLA syntax can be also used to pass arrays to functions!

```
1  #define MAX_FILTER_LEN (128)
2
3  status_e filter(size_t n, double *arr, size_t len){
4    if (len > MAX_FILTER_LEN)
5      return status_fail;
6
7    double buffer[MAX_FILTER_LEN];
8    for (size_t i=0; i < len; ++i)
9      buffer[i] = arr[i];
10   // ...
11 }
```

VLAs as arguments to functions

The VLA syntax can be also used to pass arrays to functions!

```c
#define MAX_FILTER_LEN (128)

status_e filter(size_t n, double arr[n], size_t len){
  if (len > MAX_FILTER_LEN)
    return status_fail;

  double buffer[MAX_FILTER_LEN];
  for (size_t i=0; i < len; ++i)
    buffer[i] = arr[i];
  // ...
}
```

The variable length argument (`size_t n`) must come before the VLA

VLAs as arguments to functions

This can be used to pass *normal* arrays

```
1  int use_vla(size_t n, int numbers[n]){
2    return numbers[n/2];
3  }
4
5  int main(void){
6    int numbers[] = {1, 2, 3, 4, 5};
7    return use_vla(5, numbers);
8  }
```

VLAs as arguments to functions

And, **if very lucky**, will detect bugs

```
1  int use_vla(size_t n, int numbers[n]){
2    return numbers[n/2];
3  }
4
5  int main(void){
6    int numbers[] = {1, 2, 3, 4, 5};
7    return use_vla(6, numbers);
8  }
```

```
[GCC 11] warning: 'use_vla' accessing 48 bytes in a region of size 40 [-Wstringop-overflow=]
```

VLAs as arguments to functions

Or to pass *dynamically allocated* arrays

```c
int use_vla(size_t n, int numbers[n]){
  return numbers[n/2];
}

int main(void){
  int* pnumbers = malloc(sizeof(int[5]));
  // init pnumbers
  return use_vla(5, pnumbers);
}
```

With no additional (even if rare) benefits 😢

VLAs as arguments to functions

Unless that's a multidimensional array...

```c
int use_vla(size_t n, size_t m, int numbers[n][m]){
  return numbers[n/2][m/2];
}

int main(void){
  int *pnumbers = malloc(sizeof(int[4][4]));
  // init pnumbers
  return use_vla(4, 4, pnumbers);
}
```

```
[GCC 11 / Clang 12] warning: incompatible pointer types passing 'int *' to parameter of type 'int (*)[*]'
```

Pointers to arrays (short rehearshal)

A pointer to array is declared as `'Type (*p)[size]'`

```
1 struct tm (*pNumbers)[10];
2
3 struct tm (*pDates)[n_dates];
4
5 double (*pArr2D)[rows][cols];
```

VLAs as arguments to functions

Pointers to VLAs offer some type-safety benefits

```c
int use_vla(size_t n, size_t m, int numbers[n][m]){
  return numbers[n/2][m/2];
}

int main(void){
  int (*pnumbers)[4][4] = malloc(sizeof(int[4][4]));
  // init pnumbers
  return use_vla(4, 4, *pnumbers);
}
```

No warnings: passing an array to a function that accepts arrays

VLAs and pointers to VLAs

VLAs are useful when used as arguments to functions

But there is more to the story...

Dynamic multidimensional arrays

Let's create a ***dynamic*** multidimensional array...

```
1 double* kernel_gauss_create(size_t sz){
2   double *pK = malloc(sizeof(double[sz*sz]));
3   // ...
4   return pK;
5 }
```

0.004	0.015	0.026	0.015	0.004
0.015	0.059	0.095	0.059	0.015
0.026	0.095	0.150	0.095	0.026
0.015	0.059	0.095	0.059	0.015
0.004	0.015	0.026	0.015	0.004

Dynamic multidimensional arrays

...subscripting into a ***dynamic*** multidimensional array is an ugly business

```
1  double* kernel_gauss_create(size_t sz){
2
3    double *pK = malloc(sizeof(double[sz*sz]));
4    if (!pK) return pK;
5
6    for (size_t i=0; i < sz; ++i)
7      for (size_t j=0; j < sz; ++j)
8        *(pK + i*sizeof(double[sz]) + j) = _gcoeff(sz, i, j);
9
10   return pK;
11 }
```

Dynamic multidimensional arrays

Subscripting into a ***dynamic*** multidimensional array is an ugly business

```
1  double* kernel_gauss_create(size_t sz){
2
3    double *pK = malloc(sizeof(double[sz*sz]));
4    if (!pK) return pK;
5
6    for (size_t i=0; i < sz; ++i)
7      for (size_t j=0; j < sz; ++j)
8        pK[i + sizeof(double[sz]) + j] = = _gcoeff(sz, i, j);
9
10   return pK;
11 }
```

Dynamically allocated VLA arrays

The pointer-to-array declaration can contain variable size!

```
1  double* kernel_gauss_create(size_t sz){
2
3    double (*pK)[sz][sz] = malloc(sizeof(*pK));
4    if (!pK) return pK;
5
6    for (size_t i=0; i < sz; ++i)
7      for (size_t j=0; j < sz; ++j)
8        (*pK)[i][j] = _gcoeff(sz, i, j);
9
10   return pK;
11 }
```

Dynamically allocated VLA arrays

If you don't like `*pK`...

```
1  double* kernel_gauss_create(size_t sz){
2
3    double (*pK)[sz] = malloc(sizeof(*pK) * sz);
4    if (!pK) return pK;
5
6    for (size_t i=0; i < sz; ++i)
7      for (size_t j=0; j < sz; ++j)
8        pK[i][j] = _gcoeff(sz, i, j);
9
10   return pK;
11 }
```

Credit for this one goes to Chris Wellons (nullprogram.com)

VLAs on one slide

What	Syntax	Advice
VLA variables	`Type arr[var_sz];`	Don't use prefer arrays with known size
VLA arguments	`void f(size_t n, Type arr[n])`	Use instead of pointer arguments communicates intent better
Pointers to VLAs	`Type (*pArr)[var_n][var_m]`	Use instead of pointers easier to read & write

Since we already talk about arrays...
...and arrays decay to pointers

Arrays, pointers and functions

Q: How to declare a function that takes an array or a pointer to object(s)?

A: If it's a collection of objects and the (dynamic) size is **unknown** then:

```
1 void sum( size_t n, double numbers[n] ){
2   double sum = 0;
3   for(size_t i = 0; i < n; ++i )
4     sum += numbers[i];
5   return sum;
6 }
```

VLA array syntax

Pointers to single objects

Q: How to declare a function that takes an array or a pointer to object(s)?

A: If it's conceptually a single object and it **can be `NULL`**:

```
1 time_t time( time_t *arg ){
2   time_t result = /* whatever `time` does to get the current time */ ;
3
4   if (arg)
5     *arg = result;
6
7   return result;
8 }
```

Just use a pointer

Pointers to single objects

Q: How to declare a function that takes an array or a pointer to object(s)?

A: If it's conceptually a single object and it **mustn't be `NULL`**:

```
1 size_t strlen(const char str[static 1]){
2   size_t len = 0;
3   while( *str++ != '\0')
4     ++len;
5   return len;
6 }
```

Use `[static 1]` array notation

Arrays of known size

Q: How to declare a function that takes an array or a pointer to object(s)?

A: If it's a collection of objects and the size is **known**:

```
1  pair_s solve_quadratic(double coeffs[static 3]){
2    double d = coeffs[1] * coeffs[1] - 4 * coeffs[0] * coeffs[2];
3    d = sqrt(d);
4
5    double x1 = (-coeffs[1] - d) / (2 * coeffs[0]);
6    double x2 = (-coeffs[1] + d) / (2 * coeffs[0]);
7
8    pair_s result = {.first=x1, .second=x2 };
9    return result;
10 }
```

Use `[static CONST_KNOWN_SIZE]` array notation

Array parameters with static sizes

```
void func( Type array[static CONST_KNOWN_SIZE] );
```

- This syntax is only allowed in function prototypes
- It only works for sizes that are constant integer expressions
- It communicates the intent: `func` expects at least `CONST_KNOWN_SIZE` objects
- Compilers will often be able to check and emit a warning

Array parameters with static sizes

Compilers will *sometimes* be able to check and emit a warning

```
1 pair_s solve_quadratic(double coeffs[static 3]);
2
3 double c0[] = {1, 2};
4 double* c1 = malloc(sizeof(double[2]));
5
6 pair_s r = solve_quadratic(c0);
7 pair_s r = solve_quadratic(c1);
```

These both trigger `-Wstringop-overflow` under GCC

Array parameters with static sizes

Compilers will *sometimes* be able to check and emit a warning

```
1 size_t strlen(const char str[static 1]);
2
3 const char* str = NULL;
4 size_t len = strlen(str);
```

Under GCC triggers

"argument 1 to 'char[static 1]' is null where non-null expected [-Wnonnull]"

Arrays, pointers and functions

What	How	Result
A single object, can be null	`void func(Type* obj);`	`func` is responsible for checking `obj`
A single object, shouldn't be null	`void func(Type obj[static 1]);`	`func` is responsible for checking `obj` Compilers might emit a warning
A collection of objects, size is known	`void func(Type arr[static SZ]);`	`SZ` is the minimum size Compilers might emit a warning
A collection of objects, size is unknown	`void func(size_t n, Type arr[n]);`	Communicates the intent clearly

Compound literals

Back to brace-enclosed initializers

K&R C

```
 1 typedef struct point {
 2   int x;
 3   int y;
 4 } point_S;
 5
 6 point_s makepoint(int x, int y) {
 7   point_s temp;
 8   temp.x = x;
 9   temp.y = y;
10   return temp;
11 }
12
13 point_s p = makepoint(42, 24);
```

Modestly up-to-date C

```
 1 typedef struct point {
 2   int x;
 3   int y;
 4 } point_s;
 5
 6 point_s makepoint(int x, int y) {
 7
 8   point_s tmp = {.x=x, .y=y};
 9   return tmp;
10
11 }
12
13 point_s p = makepoint(42, 24);
```

Back to brace-enclosed initializers

K&R C

```c
typedef struct point {
  int x;
  int y;
} point_S;

point_s makepoint(int x, int y) {
  point_s temp;
  temp.x = x;
  temp.y = y;
  return temp;
}

point_s p = makepoint(42, 24);
```

Modern C

```c
typedef struct point {
  int x;
  int y;
} point_s;

point_s makepoint(int x, int y) {

  return (point_s) {.x=x, .y=y};

}

point_s p = makepoint(42, 24);
```

This is known as ***compound literals***

Compound literals 101

Compound literals are constructed with

```
(Type){ /* initializer list */ }
```

This creates an ***unnamed object*** of type `Type`

Compound literals 101

The unnamed objects have:

- ***static*** storage (for file-scope compound literals)

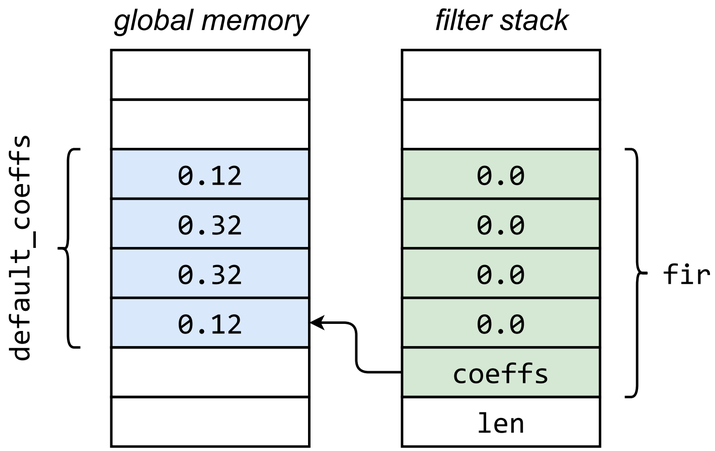
```c
static const double* default_coeffs = (double[]){0.12, 0.32, 0.32, 0.12};

void filter(...){}
```

- ***automatic*** storage (for block-scope compound literals)

```c
typedef struct fir4{
  const double* coeffs;
  double buffer[4];
} fir4_s;


void filter(size_t n, double arr[n]){
  fir4_s fir = { ... };
  // ...
  fir = (fir4_s){.coeffs=default_coeffs, .buffer={0}};
  // ...
}
```

Compound literals 101 (storage)

```c
static const double* default_coeffs =
  (double[]){0.12, 0.32, 0.32, 0.12};

void filter(size_t n, double arr[n]){
  size_t len = n * 2;
  fir4_s fir = { ... };
  // ...
  fir = (fir4_s){.coeffs=default_coeffs, .buffer={0}};
  // ...
}
```



Compound literals 101

The unnamed object is an ***lvalue*** (its address can be taken)

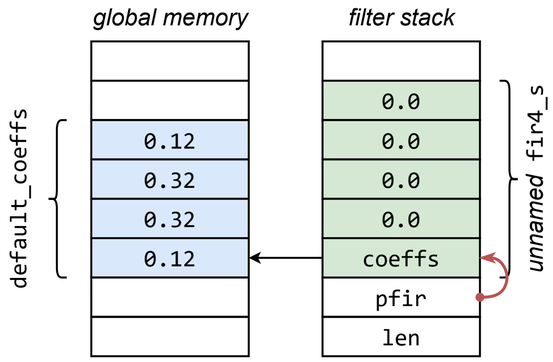
```
Type* ptr = &(Type){ /* initializer list */}
```

This enables surprising applications

```
time_t time = mktime( &(struct tm){ .tm_year=2021, .tm_mon=6, .tm_mday=1, .tm_isdst=-1 } );
```

Compound literals 101 (storage)

```c
static const double* default_coeffs =
  (double[]){0.12, 0.32, 0.32, 0.12};

void filter(size_t n, double arr[n]){
  size_t len = n * 2;

  fir4_s* pfir = &(fir4_s){.coeffs=default_coeffs, .buffer={0}};
  // ...
}
```



Compound literals vs. C++

Compound literals are not supported in ISO C++

They shouldn't be mistaken with C++ temporaries

```c
time_t time = mktime( &(struct tm){
   .tm_year=2021,
   .tm_mon=6,
   .tm_mday=1,
   .tm_isdst=-1 }
);
```

Valid C - taking an address of a compound literal

```cpp
std::time_t time = std::mktime( &std::tm{
   .tm_mday=1,
   .tm_mon=6,
   .tm_year=2021,
   .tm_isdst=-1 }
);
```

Invalid C++ - taking an address of an rvalue (temporary)

Compound literals for initialization

Given a dynamic array structure

```
1 typedef struct array {
2   double* data;
3   size_t capacity;
4   size_t count;
5 } array_s;
```

Write an initialization function

```
array_s* array_init(array_s* pa, size_t capacity);
```

Compound literals for initialization

```c
typedef struct array {
  double* data;
  size_t capacity;
  size_t count;
} array_s;
```

Good ol' C

```c
array_s* array_init(array_s* pa, size_t capacity){

  memset(pa, 0, sizeof(*pa));
  pa->data = calloc(capacity, sizeof(*pa->data));


  if (pa->data)
    pa->capacity = capacity;

  return pa;
}
```

Compound literals for initialization

```c
typedef struct array {
  double* data;
  size_t capacity;
  size_t count;
} array_s;
```

Modern C

```c
array_s* array_init(array_s* pa, size_t capacity){

  *pa = (array_s){
    .data = calloc(capacity, sizeof(*pa->data))
  };

  if (pa->data)
    pa->capacity = capacity;

  return pa;
}
```

Compound literal + designated initializers

(`pa->count` and `pa->capacity` are zeroed automatically)

Compound literals for initialization

```c
typedef struct array {
  double* data;
  size_t capacity;
  size_t count;
} array_s;
```

Modern C

```c
array_s* array_init(array_s* pa, size_t capacity){

  *pa = (array_s){
    .data = calloc(capacity, sizeof(*pa->data))
  };

  if (pa->data)
    pa->capacity = capacity;

  return pa;
}
```

The parameter list also needs fixing

(`pa` shouldn't be `NULL`, should it?)

Compound literals for initialization

```c
typedef struct array {
  double* data;
  size_t capacity;
  size_t count;
} array_s;
```

Modern C

```c
array_s* array_init(array_s pa[static 1], size_t capacity){

  *pa = (array_s){
    .data = calloc(capacity, sizeof(*pa->data))
  };

  if (pa->data)
    pa->capacity = capacity;

  return pa;
}
```

Compound literal + designated initializers

`pa` argument checked against `NULL`

Compound literals for initialization

Filling a (dynamic) array with an ad-hoc sequence

```
 1 void fill(size_t n, long target[n], size_t m, long source[m]){
 2   assert(n >= m);
 3   memcpy(target, source, sizeof(long[m]));
 4 }
 5
 6 #define ARR_SZ (100)
 7
 8 int main(void){
 9   long* numbers = malloc(sizeof(long[ARR_SZ]));
10   fill(ARR_SZ, numbers, 3, (long[]){0, 1, 1});
11 }
```

Modern C style

A key-value pair

```
1 typedef struct kv_pair{
2   char* key;
3   long value;
4 } kv_pair_s;
```

And its usual alloc/init function

```
1  kv_pair_s* mk_pair(const char* key, long val){
2
3    kv_pair_s* kv = malloc(sizeof(*kv));
4    if (kv){
5      kv->key = malloc(strlen(key) + 1);
6      if (kv->key){
7        strcpy(kv->key, key);
8        kv->value = val;
9      }
10     else{
11       free(kv);
12       kv = NULL;
13     }
14   }
15   return kv;
16 }
```

Modern C style

A key-value pair

```
1 typedef struct kv_pair{
2   char* key;
3   long value;
4 } kv_pair_s;
```

And its modern alloc/init function

```
1  kv_pair_s* mk_pair_m(const char key[static 1], long val){
2
3    kv_pair_s* kv = malloc(sizeof(*kv));
4    if (kv){
5      *kv = (kv_pair_s){
6        .key = malloc(strlen(key) + 1),
7        .value = val
8      };
9      if (kv->key){
10       strcpy(kv->key, key);
11     }
12     else{
13       free(kv);
14       kv = NULL;
15     }
16   }
17   return kv;
18 }
```

Modern C style

The best thing: both versions generate the same assembly

Clang 12.0 (30 instructions)

```
1   mk_pair:
2     push    r15
3     push    r14
4     push    rbx
5     mov     r14, rsi
6     mov     r15, rdi
7     mov     edi, 16
8     call    malloc
9     test    rax, rax
10    je      .LBB0_4
11    mov     rbx, rax
12    mov     rdi, r15
13    call    strlen
14    lea     rdi, [rax + 1]
15    call    malloc
16    mov     qword ptr [rbx], rax
17    test    rax, rax
18    je      .LBB0_3
19    mov     rdi, rax
20    mov     rsi, r15
```

GCC 11.1 (36 instructions)

```
1  mk_pair:
2    push    r13
3    push    r12
4    push    rbp
5    mov     rbp, rdi
6    mov     edi, 16
7    push    rbx
8    mov     rbx, rsi
9    sub     rsp, 8
10   call    malloc
11   mov     r12, rax
12   test    rax, rax
13   je      .L1
14   mov     rdi, rbp
15   call    strlen
16   lea     r13, [rax+1]
17   mov     rdi, r13
18   call    malloc
19   mov     QWORD PTR [r12], rax
20   mov     rdi, rax
```

Compound literals as function arguments

```
image_s* blur(image_s* img, size_t width, blur_type type, int cwh, _Bool in_place);
```

A bit ugly, isn't it?😓

Function arguments

```
1 typedef struct blur_params {
2   size_t width;
3   blur_type type;
4   int compute_hw;
5   _Bool in_place;
6 } blur_params_s;
7
8 image_s* blur(image_s img[static 1], blur_params_s params);
```

Much better!

```
1 blur_params_s params = { .width=64, .type=box, .compute_hw=0, .in_place=true };
2 blur(img, params);
```

But there's more!

The `struct` argument can be created in-place

```
1 image_s* blur(blur_params_s img[static 1], blur_params_s params);
2
3 blur(img, (blur_params_s){ .width=64, .type=box, .compute_hw=0, .in_place=true });
```

Both passing by value and by pointer are ok (let the optimizer do its work)

```
1 image_s* blur(image_s img[static 1], const blur_params_s* params);
2
3 blur(img, &(blur_params_s){ .width=64, .type=box, .compute_hw=0, .in_place=true });
```

Structs == default arguments

Some arguments can be even skipped - because compound `struct` literals

```
1 image_s* blur(image_s img[static 1], blur_params_s params);
2
3 enum blur_type{ box, gauss };
4 blur(img, (blur_params_s){ .width=64, .type=box, .compute_hw=0, .in_place=true });
5
6 blur(img, (blur_params_s){ .width=64, .in_place=true });
```

The omitted arguments will be zero-initialized

Structs == default arguments

Add a little macro magic to the mix

```
blur( &img, .width=64, .in_place=true );
```

Not that much magic actually...

```
1 image_s* _blur(image_s* img, blur_params_s params){
2 #define blur(img, ...) _blur((img), (blur_params_s){__VA_ARGS__})
```

Structs == default arguments

```
1 image_s* _blur(image_s img[static 1], blur_params_s params);
2 #define blur(img, ...) _blur((img), (blur_params_s){__VA_ARGS__})
3
4 blur( &img, .width=64, .in_place=true );
```

- `blur` macro takes an `img`
- and a variable number of arguments hidden behind `(...)`
- they are expanded in `_blur` function call with `__VA_ARGS__`
- during expansion a `blur_params_s` object is created in-place using those arguments

Arbitrary default values

With little effort arbitrary default values are possible

```
1 enum blur_type{ box, gauss };
2
3 image_s* _blur(image_s img[static 1], blur_params_s params);
4 #define blur(img, ...) _blur((img), (blur_params_s){.width=32, .type=gauss, __VA_ARGS__})
5
6 blur( &img, .width=64, .in_place=true );
```

- value of `width` is overridden from default `32` to `64`
- `type` defaults to `gauss`
- this will trigger an `initializer override warning`

Arbitrary default values

The warning can be disabled temporarily

```
1  image_s* _blur(image_s img[static 1], blur_params_s params);
2  #define DEF_ARGS_ON \
3    _Pragma("GCC diagnostic push") \
4    _Pragma("GCC diagnostic ignored \"-Woverride-init\"")
5
6  #define DEF_ARGS_OFF \
7    _Pragma("GCC diagnostic pop")
8
9  #define blur(img, ...) \
10   DEF_ARGS_ON \
11   _blur((img), (blur_params_s){.width=64, .type=gauss, __VA_ARGS__}) \
12   DEF_ARGS_OFF
13
14 blur( &img, .width=64, .in_place=true );
```

The `_Pragma` operator was introduced in C99 to allow `#pragma`s in macro expansion

Compound literals

- Create unnamed objects with static or automatic storage duration
- Are *lvalues* - you can safely take an address
- Can be used to initialize `struct`s and arrays in-place
  (compilers optimize them away, if needed)
- Can be used as function arguments (either by-value or by-pointer)
- Can enable features like default function arguments

Since we are talking about `structs`...

...and there is more to them

Let's make a string

The string

```
1 typedef struct string{
2   size_t sz_arr;
3   size_t length;
4   char* arr;
5 } string_s;
```

Its initialization

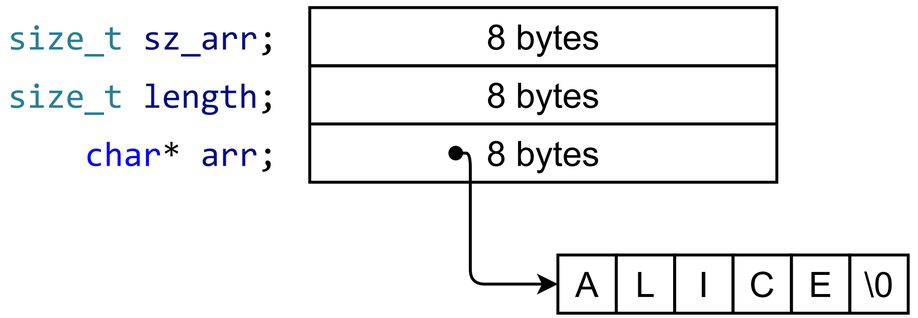
```
 1 string_s* mk_string(const char str[static 1]){
 2   string_s* p = malloc(sizeof(*p));
 3   if (p){
 4     size_t sz = strlen(str);
 5     *p = (string_s){
 6       .sz_arr = sz + 1,
 7       .length = sz,
 8       .arr = malloc(sz + 1)
 9     };
10     if (p->arr){
11       memcpy(p->arr, str, sz +1);
12     }
13     else{
14       free(p);
15       p = NULL;
16     }
```

Let’s make a string

The string

```
1 typedef struct string{
2   size_t sz_arr;
3   size_t length;
4   char* arr;
5 } string_s;
6
7 // sizeof(string_s) == 24
```

It's memory layout



- 2 x `malloc`
- At least 8 bytes lost (on 64 bit arch.)

Flexible array members

The string

```
1 typedef struct string{
2   size_t sz_arr;
3   size_t length;
4   char arr[];
5 } string_s;
6
7 // sizeof(string_s) == 16
```

- `struct string` has now a flexible array member `arr`
- a flexible array member must be the last member of a `struct`
- `struct string` has an ***incomplete type***
- it cannot be a member of another `struct`

Flexible array members

The string

```
1 typedef struct string{
2   size_t sz_arr;
3   size_t length;
4   char arr[];
5 } string_s;
```

```
#define ARR_SZ (100)

string_s names[ARR_SZ];
```

✗ arrays are fobidden

```
struct person{
  string_s name;
  int id;
};
```

✗ cannot be a member of another struct

```
string_s str = {
  .sz_arr = 0,
  .length = 0,
  .arr = "alice"};
```

✗ direct initialization of the flexible array member

Side note on arrays as parameters

Arrays of structures with flexible array members == 

```c
typedef struct string{
  size_t sz_arr;
  size_t length;
  char arr[];
} string_s;

void prn_string(const string_s str[static 1]){
  printf("%ld, %ld, %s\n", str->sz_arr, str->length, str->arr);
}
```

A less portable solution

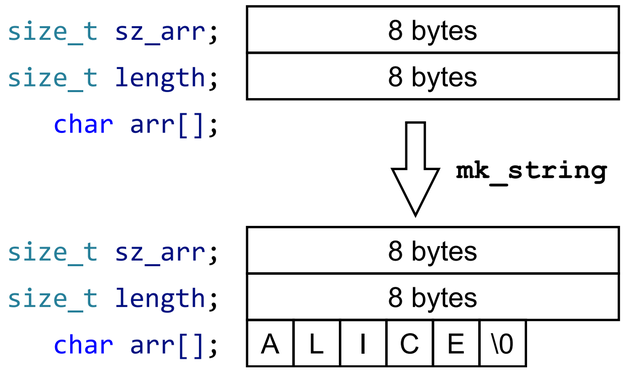
```c
__attribute__((nonnull))
void prn_string(const string_s* str){
  printf("%ld, %ld, %s\n", str->sz_arr, str->length, str->arr);
}
```

Back to the string

The string

```
 1 typedef struct string{
 2   size_t sz_arr;
 3   size_t length;
 4   char arr[];
 5 } string_s;
 6
 7 string_s* mk_string(const char str[static 1]){
 8   size_t sz = strlen(str);
 9   string_s* p = malloc(sizeof(*p) + sizeof(char[sz + 1]));
10   if (p){
11     *p = (string_s){.sz_arr = sz  + 1, .length = sz};
12     memcpy(p->arr, str, sz + 1);
13   }
14   return p;
15 }
```

It's memory layout



- Only 1 x `malloc`
- No wasted space (and only 1 `malloc` space overhead)

One last struct thing

Going anonymous

Embedded software engineering is all about repetition and abbreviations...

```
 1 enum { data, test, fetch };
 2 typedef uint8_t payload_type;
 3 typedef uint8_t test_type;
 4
 5 struct data_payload  { uint8_t bytes[4]; };
 6 struct test_payload  { test_type tt; uint16_t flags; };
 7 struct fetch_payload { uint16_t address; uint16_t length; };
 8
 9 typedef struct packet {
10   payload_type pt;
11   union {
12     struct data_payload pdd;
13     struct test_payload pdt;
14     struct fetch_payload pdf;
15   } payload;
16 } packet_s;
17
18 void test_fetch(void){
19   packet_s test_mem = { .pt=test, .payload.pdt = {.tt=2, .flags=0x4321 } };
20   packet_s test_res = { .pt=fetch, .payload.pdf = { address=0x0100, length=64 } };
```

Going anonymous

Embedded software engineering is all about repetition and abbreviations...

```
 1 enum { data, test, fetch };
 2 typedef uint8_t payload_type;
 3 typedef uint8_t test_type;
 4
 5 struct data_payload { uint8_t bytes[4]; };
 6 struct test_payload { test_type tt; uint16_t flags; };
 7 struct fetch_payload { uint16_t address; uint16_t length; };
 8
 9 typedef struct packet {
10   payload_type pt;
11   union {
12     struct data_payload pdd;
13     struct test_payload pdt;
14     struct fetch_payload pdf;
15   }; // <-- SEE! No name here!
16 } packet_s;
17
18 void test_fetch(void){
19   packet_s test_mem = { .pt=test, .pdt = {.tt=2, .flags=0x4321 } };
20   packet_s test_res = { .pt=fetch, .pdf = {.address=0x0100, .length=64 } };
```

Going anonymous

Embedded software engineering is all about digging deeper...

```
1  typedef struct packet {
2    payload_type pt;
3    union{
4      struct { uint8_t bytes[4]; }; // No...
5      struct { test_type tt; uint16_t flags; }; // ...name...
6      struct { uint16_t address; uint16_t length; };  // ...here!
7    }; // <-- And still none here.
8  } packet_s;
9
10 void test_fetch(void){
11   packet_s test_mem = { .pt=test, .tt=2, .flags=0x4321 };
12   packet_s test_res = { .pt=fetch, .address=0x0100, .length=64 };
13 }
```

- Members of anonymous `struct`s and `union`s are directly accessible in the enclosing type
- This applies recursively

Don’t be afraid of values

Returning values from functions

Dynamically allocated variant

```
1  string_s* mk_string(const char str[static 1]){
2
3    string_s* p = malloc(sizeof(*p));
4    if (p){
5      size_t sz = strlen(str);
6      *p = (string_s){
7        .sz_arr = sz + 1,
8        .length = sz,
9        .arr = malloc(sz + 1)
10     };
11     if (p->arr){
12       memcpy(p->arr, str, sz +1);
13     }
14     else{
15       free(p);
16       p = NULL;
17     }
18   }
19   return p;
20 }
```

Value-on-stack variant

```
1  string_s mk_string(const char str[static 1]){
2
3    size_t sz = strlen(str);
4
5    string_s s = {
6        .sz_arr = sz + 1,
7        .length = sz,
8        .arr = malloc(sz + 1)
9      };
10
11   if (s.arr){
12     memcpy(s.arr, str, sz +1);
13   }
14   else{
15     s = (string_s){0};
16   }
17
18   return s;
19 }
```

Returning values from functions

Let's start with the dynamically allocated structure

```
1 void func_ptr(){
2   string_s* ps = mk_string("Alice and Bob learn C!");
3 }
```

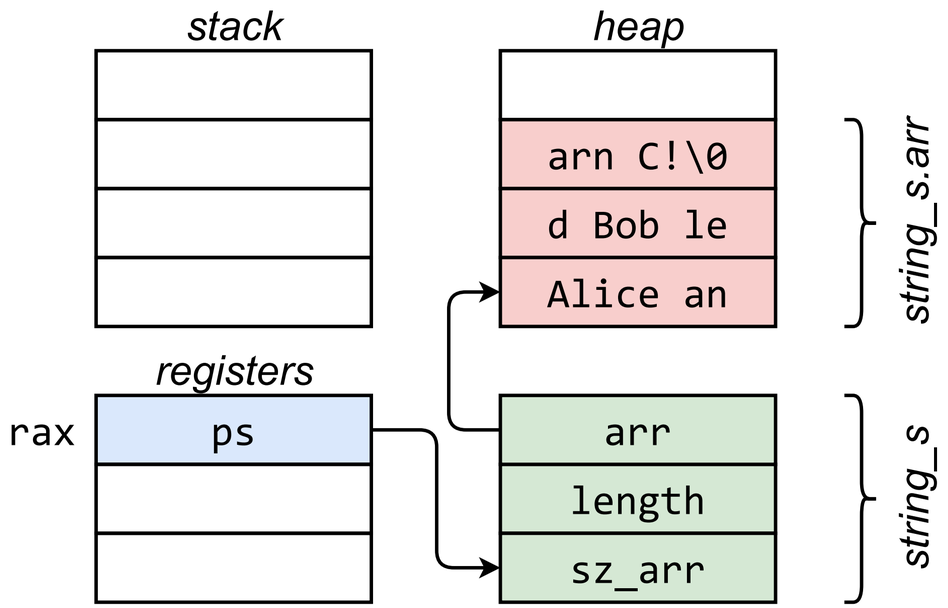
`clang 12` or `gcc 11.1` with `-O1`

Returning values from functions

Generated assembly

```
1  func_ptr:
2    push    rax
3    mov     edi, offset .L.str
4    call    mk_string
5    pop     rax
6    ret
```

Memory after the call



Returning values from functions

Return by value variant

```
1 void func_val(){
2   string_s s = mk_string("Alice and Bob learn C!");
3 }
```

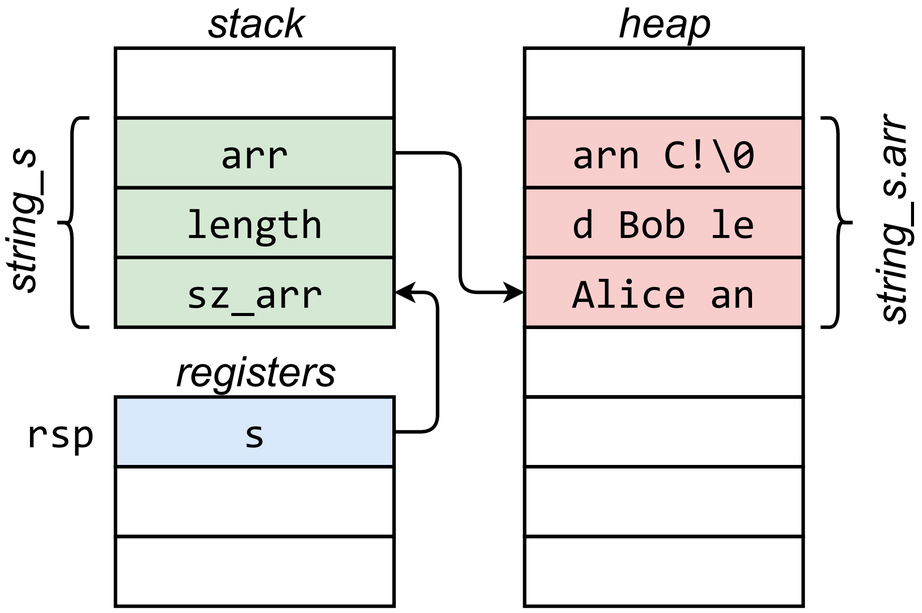
`clang 12` or `gcc 11.1` with `-O1`

Returning values from functions

Generated assembly

```
1  func_val:
2    sub     rsp, 24    ; make space for string_s
3    mov     rdi, rsp   ; pass stack pointer
4    mov     esi, offset .L.str
5    call    mk_string
6    add     rsp, 24    ; roll-back stack ptr
7    ret
```

Memory after the call

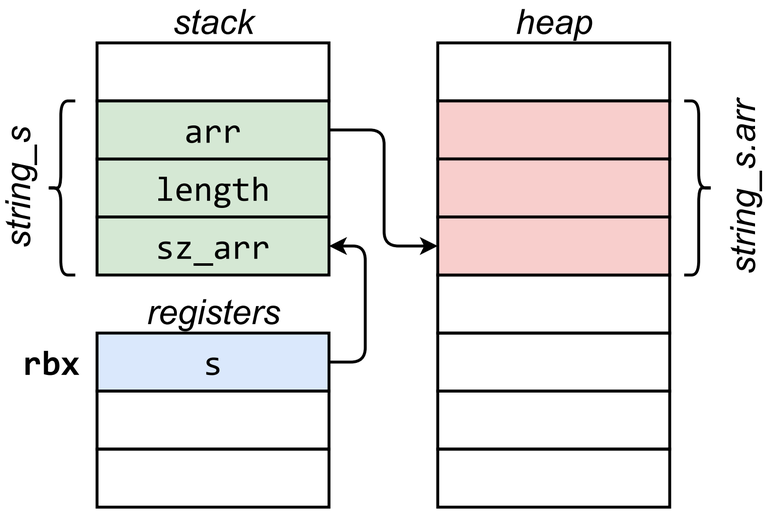


Returning values from functions

Generated assembly

```
 1  mk_string:
 2    ; bookkeeping
 3    mov    rbx, rdi   ; beginning of the stack
 4    mov    rdi, rsi   ; address of string
 5    call   strlen     ; rax <- strlen(str)
 6    lea    r15, [rax + 1]             ; r15 <- sz + 1
 7    mov    qword ptr [rbx], r15       ; .sz_arr = sz + 1
 8    mov    qword ptr [rbx + 8], rax   ; .length = sz
 9    mov    rdi, r15
10    call   malloc     ; rax <- malloc(sz + 1)
11    mov    qword ptr [rbx + 16], rax ; .arr = rax
```

Memory during the call



`string_s` is created directly on the stack at its target location

Passing values to functions

Pass by pointer

```
1 void func_ptr(const string_s* ps){
2   printf("%s", ps->arr);
3 }
```

Pass by value

```
1 void func_val(string_s s){
2   printf("%s", s.arr);
3 }
```

Passing values to functions

Pass by pointer

```
1  func_ptr:
2    push     rax
3    mov      rsi, qword ptr [rdi + 16]
4    mov      edi, offset .L.str.1
5    xor      eax, eax
6    call     printf
7    pop      rax
8    ret
```

Address of `string_s` is passed in `rdi`

Pass by value

```
1  func_val:
2    push     rax
3    mov      rsi, qword ptr [rsp + 32]
4    mov      edi, offset .L.str.1
5    xor      eax, eax
6    call     printf
7    pop      rax
8    ret
```

`string_s` is located at `rsp + 16`

Passing values to functions

Pass by value

```
1  func_val:
2    push    rax
3    mov     rsi, qword ptr [rsp + 32]
4    mov     edi, offset .L.str.1
5    xor     eax, eax
6    call    printf
7    pop     rax
8    ret
```

`string_s` is located at `rsp + 16`

Calling function

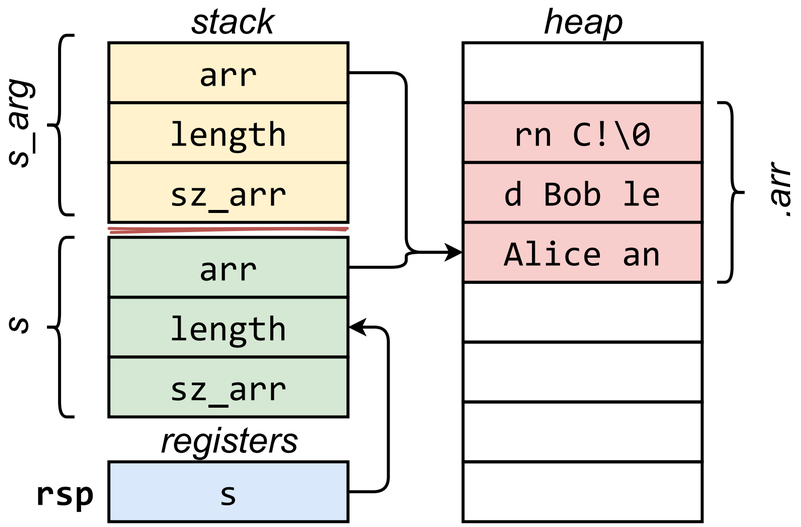
```
1  call_by_val:
2    ; ...
3    call    mk_string ; create a string
4    ; ...
5    mov     rax, qword ptr [rsp + 48] ; copy:(
6    mov     qword ptr [rsp + 16], rax
7    movups  xmm0, xmmword ptr [rsp + 32]
8    movups  xmmword ptr [rsp], xmm0
9    ; ...
10   call    func_val
```

(2 x `mov` + 2 x `movups`)

Passing values to functions

```
1 void func_val(string_s s_arg){
2   printf("%s", s_arg.arr);
3 }
4
5 void call_by_val(void){
6   string_s s = mk_string("Alice and Bob learn C!");
7   func_val(s);
8 }
```

A copy of the `string_s` object is made on the stack



Passing values to functions

Pass by value

```
1  func_val:
2    push     rax
3    mov      rsi, qword ptr [rsp + 32]
4    mov      edi, offset .L.str.1
5    xor      eax, eax
6    call     printf
7    pop      rax
8    ret
```

Calling function

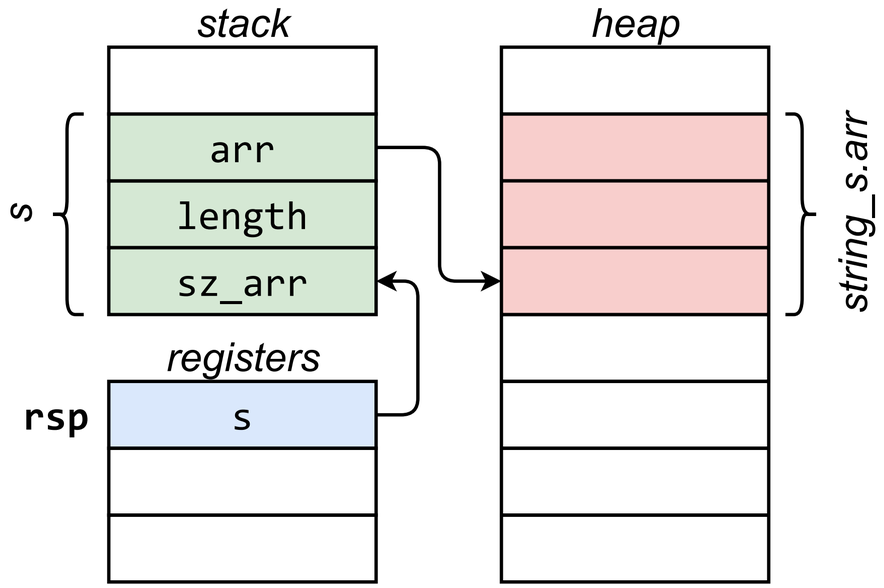
```
1  call_by_val:
2    ; ...
3    call     mk_string ; create a string
4    ; ...
5    mov      rax, qword ptr [rsp + 48] ; copy:(
6    mov      qword ptr [rsp + 16], rax
7    movups   xmm0, xmmword ptr [rsp + 32]
8    movups   xmmword ptr [rsp], xmm0
9    ; ...
10   call     func_val
```

...but this was compiled with `-O1`

Passing values to functions

Pass by value (`-O3`)

```
1  call_by_val:
2    sub     rsp, 40
3    mov     esi, offset .L.str.2
4    mov     rdi, rsp
5    call    mk_string ; create string
6    mov     rsi, qword ptr [rsp + 16] ; no copy!
7    mov     edi, offset .L.str.1
8    mov     eax, 0
9    call    printf
10   add     rsp, 40
11   ret
```



`func_val(string_s s_arg)` is inlined

`clang` does the same + aggressively inlines `mk_string`

Passing and returning values

- Return by value if you don't need dynamic memory
- With copy elision objects are constructed in-place
- Passing by value often makes sense
- Optimizing compilers will remove unnecessary copies...
  ...or even whole function calls
- Check the assembly or measure

Function overloading in C

aka *generic selection*

How to overload a function in C

Two simple data structures

```c
typedef struct Point2D {
  double x, y;
} Point2D_s;

typedef struct Circle {
  Point2D_s center;
  double radius;
} Circle_s;

typedef struct Rectangle {
  Point2D_s center;
  double w, h;
} Rectangle_s;
```

That need scaling

```c
void scale(Circle_s* c, double scale){

  c->radius *= scale;

}

void scale(Rectangle_s* r, double scale){

  r->w *= scale;
  r->h *= scale;

}
```

A ***no-go*** in C

How to overload a function in C

Two simple data structures

```c
typedef struct Point2D {
  double x, y;
} Point2D_s;

typedef struct Circle {
  Point2D_s center;
  double radius;
} Circle_s;

typedef struct Rectangle {
  Point2D_s center;
  double w, h;
} Rectangle_s;
```

That need scaling

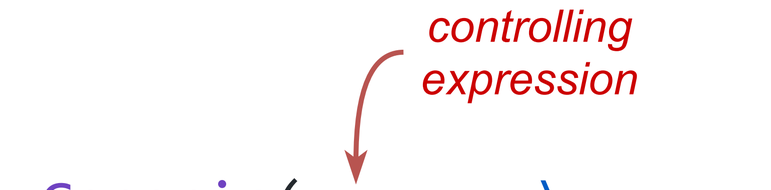
```c
void scale_circ(Circle_s* c, double scale){

  c->radius *= scale;

}

void scale_rect(Rectangle_s* r, double scale){

  r->w *= scale;
  r->h *= scale;

}
```

Different function names - not so handy... 🤔

Say `Hello` to generic selection

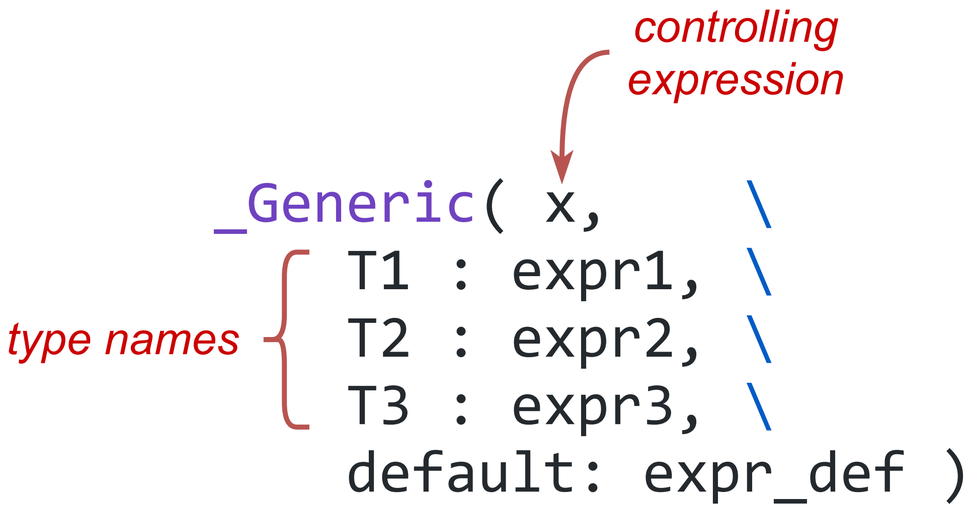
```
_Generic( x,     \
    T1 : expr1, \
    T2 : expr2, \
    T3 : expr3, \
    default: expr_def )
```

Say `Hello` to generic selection

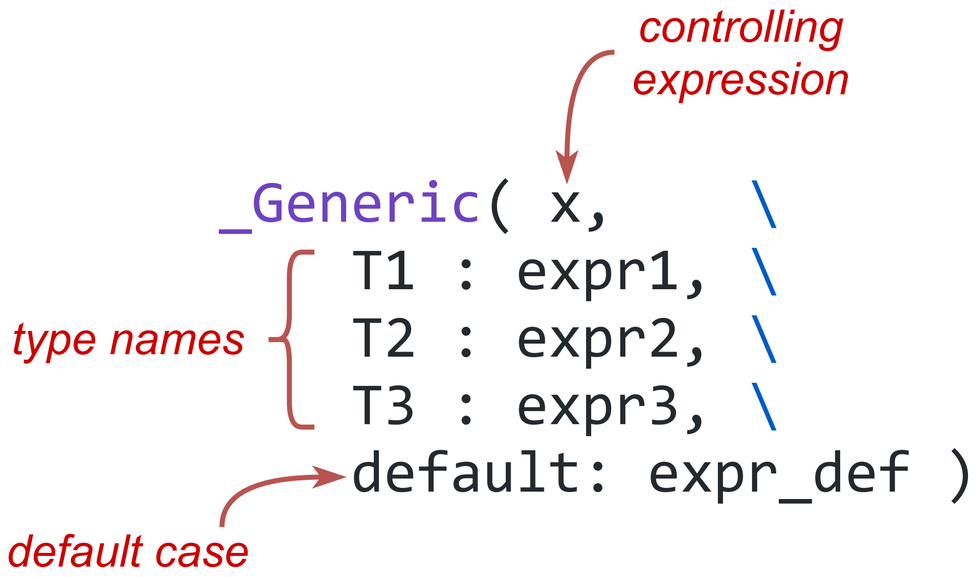



```
_Generic( x,     \
    T1 : expr1, \
    T2 : expr2, \
    T3 : expr3, \
    default: expr_def )
```

Say `Hello` to generic selection



Say `Hello` to generic selection



Say `Hello` to generic selection

Generic selection for function overloading

```c
void scale_circ(Circle_s* c, double scale);
void scale_rect(Rectangle_s* r, double scale);

#define scale(obj, scale          \
  _Generic( (obj),                \
    Rectangle_s* : scale_rect,    \
    Circle_s* : scale_circ        \
  )((obj), (scale))

void func(){
  Rectangle rect;
  Circle circ;

  scale(&rect, 5.3);
  scale(&circ, 3.5);
}
```

Say `Hello` to generic selection

So far so good, but what about...
...overloading on number of parameter

```
1  void scale_circ_1p(Circle_s* c, double scale);
2
3  void scale_rect_1p(Rectangle_s* r, double scale);
4  void scale_rect_2p(Rectangle_s* r, double w_scale, double h_scale);
5
6  void func(){
7    Rectangle rect1;
8    Rectangle rect2;
9
10   scale(&rect1, 5.3);
11   scale(&rect2, 5.3, 3.5)
12 }
```

Overloading on number of parameters

The bad news is: `_Generic` doesn't work with macros for expressions

The good news is:

```
1  #define scale2p(obj, ...)                \
2    _Generic( (obj),                       \
3      Rectangle_s* : scale_rect_2p         \
4    )((obj), __VA_ARGS__)
5
6  #define scale1p(obj, ...)                \
7    _Generic( (obj),                       \
8      Rectangle_s* : scale_rect_1p,        \
9      Circle_s* : scale_circ_1p            \
10   )((obj), __VA_ARGS__)
```

`` `...` `` stands for variable number of arguments

they are *passed* on with `__VA_ARGS__`

Overloading on number of parameters

This allows for a classic trick

```
#define scale2p(obj, ...)          \
  _Generic( (obj),                 \
    Rectangle_s* : scale_rect_2p   \
  )((obj), __VA_ARGS__)

#define scale1p(obj, ...)          \
  _Generic( (obj),                 \
    Rectangle_s* : scale_rect_1p,  \
    Circle_s* : scale_circ_1p      \
  )((obj), __VA_ARGS__)

#define INVOKE(_1, _2, _3, NAME, ...) NAME
#define scale(...) INVOKE(__VA_ARGS__, scale2p, scale1p,)(__VA_ARGS__)
```