REVE1

Programming in C

```
int a = 5;
const b = 7;
int c;

int foo(int a, int b)
{
  int e, f, g;
  char *m = malloc(10);
}
```

Module Outline

- Program Structure
- Language Elements
- Modules
- Variables and Memory
- Parameter Passing
- The C Standard Library
- Module Summary

Python vs C

```python
def binary_search( data, N, value):


  lo, hi = 0, N-1

  while lo < hi:
    mid = (lo + hi ) // 2

    if data[mid] < value:
      lo = mid + 1
    else:
      hi = mid


  if hi == lo and data[lo] == value:
    return lo
  else:
    return -1
```

```c
int binary_search(int *data, int N,
                  int value)
{
  int lo = 0, hi = N-1;

  while (lo < hi) {
    size_t mid = (lo + hi) / 2;

    if (data[mid] < value)
      lo = mid + 1;
    else
      hi = mid;
  }

  if ((hi == lo) && (data[lo] == value))
    return lo;
  else
    return -1;
}
```

Why C / C++?

- **Ubiquity**
  - operating systems
  - device drivers
  - embedded computing, edge devices, IoT
  - virtual machines of higher-level languages written in C/C++
    - CPython
    - JavaScript engines, e.g. V8
- **Power**
  - direct access to memory
  - expressive, yet terse
- **Speed**
  https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

Why C / C++?

With great power comes great responsibility.

- Aka "easy to shoot yourself in the foot"
- Writing good, standards-compliant code is **not hard** and will make your life **much easier**.
- Bad code is **not a question of the language**
  - well, maybe
  - C can also be made unreadable

HOCHSCHULE LUZERN

```c
#include <stdio.h>

void main(void)
{
  printf("Hello, world!\n");
}
```

Program Structure

Hello, world!

- **Our first C program**

```c
#include <stdio.h>

void main(void)
{
  printf("Hello, world!\n");
}
```

```
$ gcc -o hello hello.c
$ ./hello
Hello, world!
$
```


HOCHSCHULE
LUZERN

General Structure

```c
#include <stdio.h>

#define LIMIT 50

int A = 0;
const int B = 1;

int fib(int n)
{
  if (n > 1) return fib(n-1)+fib(n-2);
  else return 1;
}

int main(int argc, char *argv[])
{
  int n = argc;

  printf("Hello, world!\n");
  printf(“Fib(%d) = %d\n“, n, fib(n));

  return 0;
}
```

include files

preprocessor defines

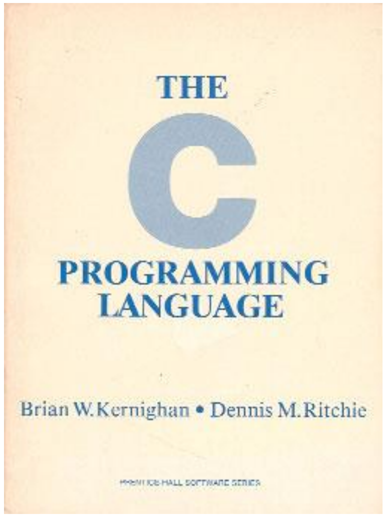
global variables

function definitions

local variables

main function definition

HOCHSCHULE LUZERN



Language Elements

Language Elements

- **Block delimiter: { … }**
  - must be properly nested
- **Statement end marker: ;**
  - after every statement
- **Comments**
  - single line: // …
  - multi-line: /* … */
  - multi-line > single line
  - nesting of same-level comments not supported

```
/*
 * language elements
 */
#include <stdio.h>

// this is the main function
void main(void)
{
  /* not needed
  // some variables
  int i,j;
  */
  printf("Hello, world!\n");
}
```

Language Elements

- **Identifiers**

  [a-zA-Z_][a-zA-Z0-9_]*

  “a letter or underscore followed by
  an arbitrary number of letters, digits,
  or underscore.”

  - identifiers are *case-sensitive*
    - HSLU, Hslu, hsLU, hlsu
      are all separate variables

- **Whitespace**
  - spaces, tabs, newlines
  - is ignored
  - no forced indentation

```
/*
 * language elements
 */
#include <stdio.h>

// this is the main function
void main    (     void
){
  // some variables
  int _i,j123__5;
  unsigned char x;
printf("Hello, world!\n");
}
```

Language Elements

- **Keywords**
  - reserved
  - i.e., cannot be used as identifiers

auto	if	enum
char	else	struct
short	switch	union
int	case	typedef
long	for	extern
float	do	static
double	while	sizeof
void	break	
register	continue	
volatile	default	
signed	goto	
unsigned		

Language Elements

- **Sequence of statements**
  - separated by semicolon

```
a = a + 1;
b = b + a;
c = b + a; d=c-1;e
=     1-
b;
```

- **Parallel statements**
  - separated by comma

```
void foo(int a, int b, int c)
{
  int d, e, f;

  d = a+b, e = b+c, f = a+c;
  …
```

Language Elements

- **Operators**
  - arithmetic: +, -, *, /, %
  - bitwise: &, |, ~, ^, <<, >>
  - logical: &&, ||, !
  - relational: <, <=, ==, !=, =>, >
  - assignment: =
  - address operator: &
  - dereferencing operator: *

```c
void foo(int a, int b)
{
  int c, d;
  int *p;

  c = a + b;
  d = c - b;
  c = a * b / c;
  d = d % a;
  a = c & d | a;
  a = a && b;
  a = a <= b;
  p = &c;
  d = *p & a ^ (b << (c!=d));
}
```

Language Elements

- **Operators**
  - short form assignment:
    a = a ● b → a ●= b

```
void foo(int a, int b)
{
  int c, d;

  c = c + b;
  c += b;

  c = c >> 2;
  c >>= 2;

  c = c & a;
  c &= a;


}
```


HOCHSCHULE
LUZERN

Language Elements

- **Operators**
  - in-place increment: ++, --
  - use in expression
  - position dependent

```
void foo(int a, int b)
{
  int c, d;

  c = c + 1;
  c++;

  a = a + b++;



  a = a + ++b;


}
```


HOCHSCHULE LUZERN

Language Elements

- **Expressions**
  - mathematical expressions made up from operations
  - be careful with precedence
    - tip: always use parentheses
  - more details on this later

```
void foo(int a, int b)
{
  int c, d;

  c = (a + b) * b;


  c = a + b >> 2;
  c = (a + b) >> 2;
  c = a + (b >> 2);




}
```


HOCHSCHULE LUZERN

Language Elements

- **Assignments**
  - form: LHS = RHS
    [LHS/RHS stand for
    “left/right-hand side”]
    - LHS designates a memory address
    - RHS is an expression
  - subtle difference when using variable names as LHS vs RHS

```
void foo(int a, int b)
{
  int c, d;

  c = (a + b) * b;

  2 = a + b;


  a = a + 1;
```

Language Elements

- **if-then-else**
  - if (<cond>) true_statement;
    [else false_statement;]
  - condition <cond> *must* be enclosed in parentheses
  - *one* statement after cond and else
  - use a statement block if more than one statement required
  - nested if constructs possible
    - else matches closest if

```
void foo(int a, int b)
{
  int c, d;

  if (a > b) c = a-b;
  if (b > a) c = b-a;
  else c = a-b;
  if (2*a == b) {
    b = b*2;
    a = a+1;
  } else
    c = 3;
    d = 4;

   …
```


HOCHSCHULE
LUZERN

Language Elements

- **Condition evaluation**
  - <cond> is an expression evaluated to true/false
    - true = “value is not 0”
    - false = “value is 0”
    - if (a) identical to if (a != 0)
    - true/false not defined in C!
      - use 0, 1 instead
      - define true/false
      - careful, however!
        → not recommended

```
void foo(int a, int b)
{
  int c, d;

  if (a)      c = a-b;
  if (a != 0) c = a-b:

  if (!b)     c = b-a;
  if (b == 0) c = b-a;

void bar(int a, int b)
{
  #define true  1
  #define false 0

  a = true; if (a) …
  if (a == true) …
```


HOCHSCHULE LUZERN

Language Elements

- **Condition evaluation**
  - source of error with accidental use of assignment operator (=) instead of equality relation operator (==)
    - if (a = b) …
    - identical to
      a = b; if (a != 0) …
    - if you must use it, enclose assignment in parentheses and spell the condition out to clarify your intention
    - useful construct,
      but use with care

```
void foo(int a, int b)
{
  int c, d;

  if (a = b) c = 0;





  if ((a = b) != 0) c = 0




  if ((c = open("f.txt")) < 0)
    printf("Can’t open file");
```


HOCHSCHULE LUZERN

Language Elements

- **Condition evaluation**
  - pointers are memory addresses
  - as such, they are interpreted as a number in conditions
  - NULL is identical to 0

```
void foo(int *a, int *b)
{
  int c;

  if (a != NULL) c = *a;
  else c = 0;

  if (a != 0)    c = *a;
  else c = 0;

  if (a)         c = *a;
  else c = 0;




}
```


HOCHSCHULE
LUZERN

Language Elements

- **The conditional operator "?"**
  - in programming, vertical space is precious
  - like so many other constructs in C, the conditional operator allows us to write more compact code
  - the only ternary operator
  - avoid nesting

```
void foo(int *a, int *b)
{
  int c;

  if (a != NULL) c = *a;
  else c = 0;

  if (a != 0)    c = *a;
  else c = 0;

  if (a)         c = *a;
  else c = 0;

  c = a ? *a : 0;

  c=a?*a?*a:b?*b?*b:0:1:2;
}
```


HOCHSCHULE LUZERN

Language Elements

- **Loops**
  - while (<cond>) statement;
  - do statement; while (cond);
  - nesting possible

```
void foo(int *a, int *b)
{
  int c, i;

  while (a && *a) {
    c += *a;
  }
  while (a && *a) c += *a;

  do {
    c += *a;
  } while (a && *a);




}
```


HOCHSCHULE LUZERN

Language Elements

- **Loops**
  - for (init; test; update) statement;
  - nesting possible

```
void foo(int a, int N)
{
  int c=0, i, j;

  for (i=0; i<N; i++) {
    c += a;
  }

  for (i=0,c=7; i<N; i++,c--)
    c += a;

  for (i=0; i<N-1; i++) {
    for (j=i+1; j<N; j++) {
      c += i^j;
    }
  }
}
```

Language Elements

- **Loop control**
  - break
    break out of the loop at any point in the loop body
  - continue
    skip rest of loop body and direct go to the condition evaluation

```
void foo(int *a, int N)
{
  int c=0, i;

  while (i < N) {
    c += a[i];
    if (c > N) break;
    i++;
  }

  for (i=0; i<N; i++) {
    if (i % 2) continue;
    c += a[i];
  }


}
```


HOCHSCHULE LUZERN

Language Elements

- **A word on goto**
  - do not use goto
  - unless you really, really, REALLY know what you are doing

```
void foo(int a, int N)
{
  int c=0, i=0;

  while (i < N) {
    c += a;
    i++;
  }

  c = 0, i = 0;
test:
  if (i >= N) goto end;
  c += a;
  i++;
  goto test:
end:
}
```

Language Elements

- **A word on goto**
  - sometimes used in low-level code to avoid deeply nested if constructs
  - the general rule still applies: do not use goto

```
void foo(int a, int N) {
  if (a) {
    if (b) {
      if (c) {
        do something;
      } else undo stuff;
    } else undo stuff;
  } else undo stuff;
}
```

```
void foo(int a, int N) {
  if (!a) goto error;
  if (!b) goto error;
  if (!c) goto error;
  do something;
  return;
error:
   undo stuff;
}
```

Language Elements

- **Scalar data types**
  - integer data types
    [[un]signed]
    char, short [int], int,
    long [int], long long [int]
    - default: signed
  - floating point data types
    float, double, long double
  - pointers
    *

    read right to left, careful with space
    int *<variable>

    * → int: pointer to int

```
void foo(int a, int N)
{
  char c;
  unsigned char d;
  short int s;
  unsigned short us;
  int i;
  unsigned int u;
  signed long int l;
  unsigned long long ll;
  float f;
  double d;
  long double ld;

  int *a;
  char ***c;
}
```


HOCHSCHULE LUZERN

Language Elements

- **Composite data types: arrays**
  - sequence of elements of a certain data type
  - zero-indexed
    int a[10] → a[0]..a[9]
  - no range checks
  - multi-dimensional arrays
  - multi-level arrays
    array of pointers to some other data type

    "10 pointers to an array of chars"
    "array of pointers to an array of chars"

```
void foo(void)
{
  int a[10];

  a[1] = 5;


  int pixel[1024][768];
  pixel[0][0] = red;


  char *strings[10];
  char **strings;




}
```


HOCHSCHULE LUZERN

Language Elements

- **Composite data types: arrays**
  - in fact, C does not “know” arrays
  - [] is "syntactic sugar" for convenience
  - **arrays are pointers**
  - no range checks
    int *a could be
    - a pointer to a single int or
    - a pointer to a sequence of ints
    - compiler and runtime do not know!
  - the two definitions are identical

```
void foo(void)
{
  int a[10];

  a[1] = 5;
}

void foo(int *a, int b[])
{


}


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

Language Elements

- **Strings**
  - strings are arrays of characters
  - strings are defined with double quotes (")
  - ASCII character encoding
  - end of string: \0 (null) character (arithmetic value 0)
  - special characters need to be quoted
    - \n newline
    - \t tab
    - \" double-quote character
    - \0 null character
    - …

```
int main(int argc, char *argv[]) {
  printf("Hello\n");

  char str[128];
  snprintf(str, sizeof(str),
           "%d arguments\n",
           argc);

  return EXIT_SUCCESS;
}
```


HOCHSCHULE LUZERN

Language Elements

- **Composite data types: structs**
  - group data into a structure
  - variable declaration with an anonymous struct
  - named struct
  - define new type from named struct

```
struct {
  char  name[32];
  char  age;
  short sex;
  int   phonenumber;
} person;

struct person {
  …
};
struct person p;

typedef struct _person {
  …
} person;
person p;
```


HOCHSCHULE LUZERN

Language Elements

- **Composite data types: structs**
  - often used for linked lists
  - referencing struct members from a pointer to a struct

    `X->Y` is shorthand for `(*X).Y`

```
typedef struct _list {
  int key;
  int value;
  struct _list *next;
} list;

int search(list *l, int key)
{
  while (l && l->key != key) {
    l = l->next;
  }

  return l ? l->value : -1;
}
```

Language Elements

- **Composite data types: unions**
  - put several data values at the same memory address
  - biggest member determines total size of struct
  - useful to bypass the type system

```c
union flint {
  int i;
  float f;
};


void print_fhex(float f)
{
  union {
    int i;
    float f;
  } if;

  if.f = f;
  printf("%04x\n", if.i);
}
```


HOCHSCHULE LUZERN

Language Elements

- **typedef**
  - syntactic sugar
  - allows use of simpler type names
    - particularly useful for composite types
  - all types except the basic scalar types are defined using typedef
    - size_t
    - ssize_t
    - …

```
$ echo "#include <stdlib.h>" | \
    gcc -E - | \
    grep -E "typedef .* ssize_t;"
```

```
typedef unsigned int   uint32;
typedef signed int      int32;
typedef unsigned short uint16;
typedef signed short    int16;
typedef unsigned char  uint8;
typedef signed char     int8;

int32 a,b,c;
```


HOCHSCHULE LUZERN

Language Elements

- **Functions**
  - encapsulation of functionality

    <rettype> <name>(<paramlist>)
    {
      <body>
    }

  - no return value: void
  - no parameters: void
  - return type can be composite data type

```
int fact(int n)
{
  if (n > 1) {
    return n*fact(n-1);
  } else {
    return 1;
  }
}

void foo(void) {…}

list make(int k, int v)
{
  list l;
  l.key = k; l.value = v;
  return l;
}
```


HOCHSCHULE LUZERN

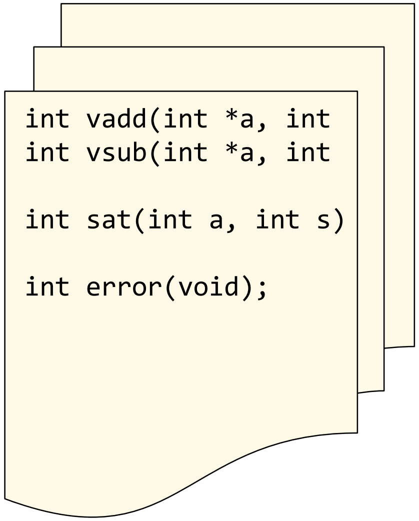
Language Elements

- **Function pointers**
  - variable definition

    <rettype> (*<name>)(<paramlist>);

  - a function pointer can point to a function of the same type
  - allows implementation of (manual) polymorphism

```
void iam_a_bike(void) {
  printf("I am a bike.\n");
}

void iam_a_car(void) {
  printf("I am a car.\n");
}

struct vehicle {
  void (*iam)(void);
  int  wheels;
}

void init(struct vehicle *v, int w) {
  v->wheels = w;
  if (w <= 2) w->iam = iam_a_bike;
  else w->iam = iam_a_car;
}

void print(struct vehicle *v) {
  v->iam();
  printf("I have %d wheels.\n", v->wheels);
}

void main(void) {
  struct vehicle v;
  init(&v, 2);
  print(&v);
}
```


int vadd(int *a, int
int vsub(int *a, int
int sat(int a, int s)
int error(void);

Operators

C Operators

- Assignment: =
- Arithmetic: +, -, *, /, %
- Bitwise: &, |, ~, ^, <<, >>
- Logical: &&, ||, !
- Relational: <, <=, ==, !=, =>, >
- Memory:
  - Reference: &
  - Dereference: *

```
void foo(int a, int b)
{
  int c, d;
  int *p;

  c = a + b;
  d = c - b;
  c = a * b / c;
  d = d % a;
  a = c & d | a;
  a = a && b;
  a = a <= b;
  p = &c;
  d = *p & a ^ (b << (c!=d));
}
```

C Operator Precedence

Precedence	Operator	Description	Associativity
1 (highest)	++ -- () [] . ->	In-place increment/decrement Function call Array access Structure/union access Structure/union access	Left-to-right
2	+ - ! ~ (type) * &	Unary plus/minus Logical NOT, bitwise NOT Type cast Dereference Address-of	Right-to-left
3	* / %	Multiplication, division, remainder	Left to right
4	+ -	Addition, subtraction	
5	<< >>	Shift left, shift right	
6	< <= > >=	Relational operators	

C Operator Precedence

Precedence	Operator	Description	Associativity
7	== !=	Relational operators	Left to right
8	&	Bitwise AND	
9	^	Bitwise XOR	
10	|	Bitwise OR	
11	&&	Logical AND	
12	||	Logical OR	
13	? :	Ternary conditional	Right-to-left
14	= ●=	Assignment Assignment by operator (*=, +=, …)	
15 (lowest)	,	Comma	Left-to-right

HOCHSCHULE LUZERN

Bit-Level Operations in C

- **Operations &, |, ~, ^, <<, >> available in C**
  - Valid for any "integral" data type
    - `char, short, int, long, long long`

- Examples (char data type)
  - `~0x41` → `0xBE`
    - `~`01000001_2 → 10111110_2
  - `~0x00` → `0xFF`
    - `~`00000000_2 → 11111111_2
  - `0x69 & 0x55` → `0x41`
    - 01101001_2 & 01010101_2 → 01000001_2
  - `0x69 | 0x55` → `0x7D`
    - 01101001_2 | 01010101_2 → 01111101_2

Logic Operations in C

- **Contrast to logical operators**
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination

- Examples (char data type)
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`

  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

HOCHSCHULE LUZERN

Mixing Signed and Unsigned Data Types

```c
// kernel memory region holding user-accessible data
#define KSIZE 1024
char kbuf[KSIZE];

// copy up to maxlen bytes from kernel region to user buffer
int copy_from_kernel(void *user_dest, int maxlen) {
    // byte count len is minimum of buffer size and maxlen
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

// libc memcpy() definition
void* memcpy(void *dest, const void *src, size_t n);
```

```c
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

determine type of size_t:

```
$ echo "#include <stdio.h>" | \
    gcc -E - | \
    grep -E "typedef .* size_t;"
```

Mixing Signed and Unsigned Data Types

- **Constants**
  - *By default are considered to be signed integers*
  - Unsigned with "U" suffix
    - 0U, 4294967259U

- **Casting**
  - **Explicit casting** between signed and unsigned same as U2T and T2U
    - int tx, ty;
    - unsigned ux, uy;
    - tx = (int) ux;
    - uy = (unsigned) ty;
  - **Implicit casting** occurs via assignments and procedure calls
    - tx = ux;
    - uy = ty;

Casting Surprises

- **Expression evaluation**
  - If there is a mix of unsigned and signed in a single expression, **signed values are implicitly cast to unsigned**
  - Including comparison operations <, >, ==, <=, >=
  - Examples for W = 32: TMIN = -2,147,483,648 , TMAX = 2,147,483,647

Constant1	Constant2	Relation	Evaluation
`0`	`0U`	`==`	`unsigned`
`-1`	`0`		
`-1`	`0U`		
`2147483647`	`-2147483647-1`		
`2147483647U`	`-2147483647-1`		
`-1`	`-2`		
`-1`	`(unsigned)-2`		
`2147483647`	`2147483648U`		
`2147483647`	`(int)2147483648U`		

HOCHSCHULE LUZERN

Should I Use Unsigned At All?

- ***Don't*** **use just because number nonnegative**

  - Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

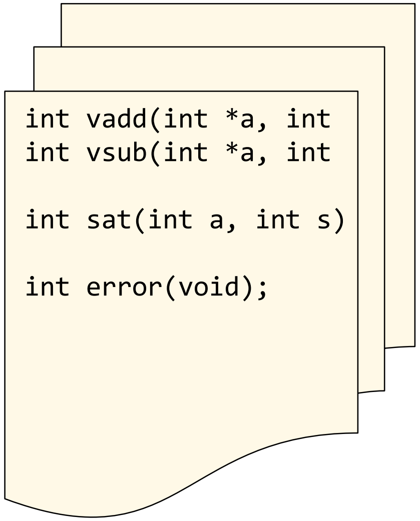
  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  …
```


HOCHSCHULE LUZERN

Should I Use Unsigned At All?

- ***Do*** **use**
  - When performing modular arithmetic
    - Multiprecision arithmetic
  - When using bits to represent sets
    - Logical right shift, no sign extension


int vadd(int *a, int
int vsub(int *a, int
int sat(int a, int s)
int error(void);

Modules

Modules

- **Defining a module**
  - module = basic compilation unit
  - group elements (data, functions, typedefs, …) into logical modules
  - a module is simply a C file with some definitions in it
  - compile using gcc's -c option

```c
int add(int a, int b)
{
  return a+b;
}

int sub(int a, int b)
{
  return a-b;
}

int mul(int a, int b)
{
  return a*b;
}
```


HOCHSCHULE
LUZERN

Modules

- **Using a module**
  - use functionality defined in an other module
  - compile warning: undefined use of add
  - information about defined types stored in object file
  - declare external function to avoid compiler warning

```c
int foo(int a, int b)
{
  return add(a, b);
}
```

```c
int add(int a, int b);

int foo(int a, int b)
{
  return add(a, b);
}
```

Modules

- **Header files**
  - declaring external functions in each file we use them is not particularly practical
  - write header file for module that provides a module's declarations
  - include header file in file using the module

```c
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
```

`intops.h`

```c
#include <intops.h>

int foo(int a, int b)
{
  return add(a, b);
}
```

Modules

- **Header files**
  - prevent header files from begin included twice
  - documentation should go into the header file
  - i.e. Doxygen-style comments

```
#ifndef __intops_h__
#define __intops_h__

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);

#endif // __intops_h__
```

`intops.h`

HOCHSCHULE LUZERN

alias	adr	mem
	28	
c	24	?
b	20	7
a	16	5
	12	
	8	
	4	
	0	

Variables and Memory

Variables and Memory

- **Definition**

  <type> <name>

- **Two consequences**
  1. allocates memory somewhere in the process' memory space to hold the data (size of allocated memory = size of type)
  2. creates a label <name> that allows us to conveniently reference this data

Variables and Memory

- **Example**

```
int a = 5, b = 7, c;
```

- compiler allocates
  - a to address 16
  - b to address 20
  - c to address 24
  - usually, but not guaranteed to be sequential

- **variable names are aliases for memory addresses**

```
… = a + …   →   … = mem[16] + …

a = …       →   mem[16] = …
```

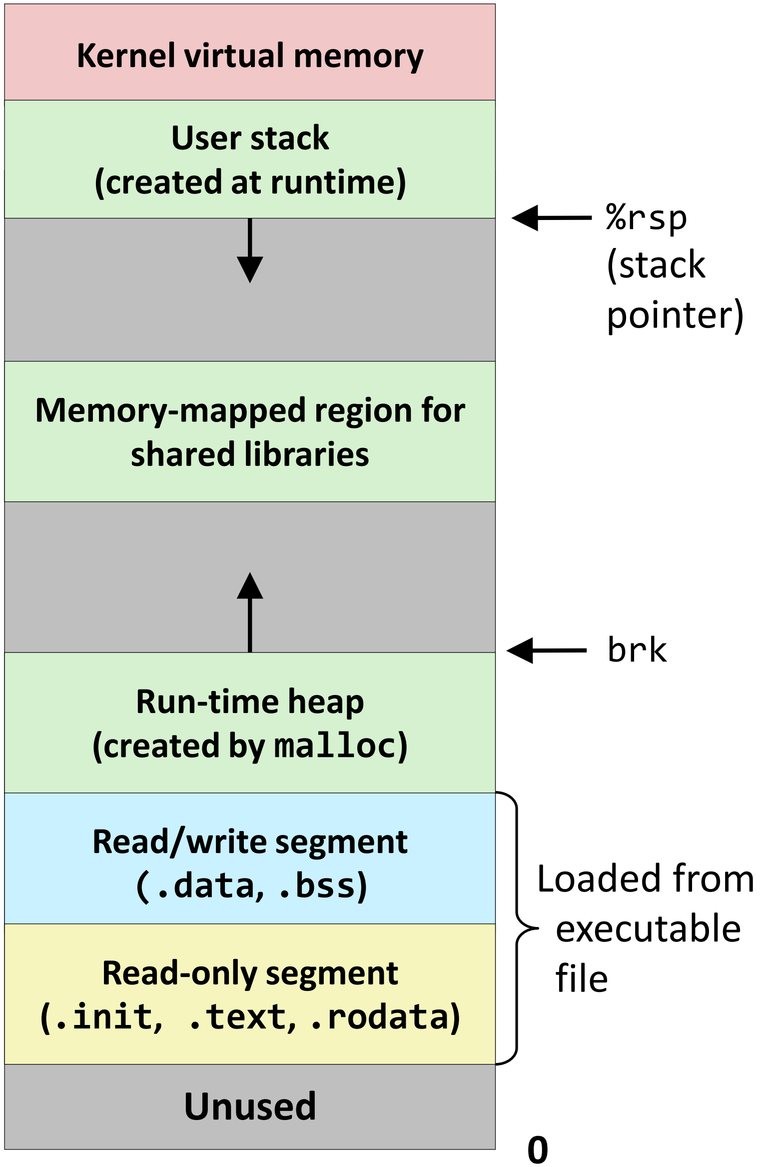
alias	adr	mem
	28	
c	24	?
b	20	7
a	16	5
	12	
	8	
	4	
	0	

HOCHSCHULE LUZERN

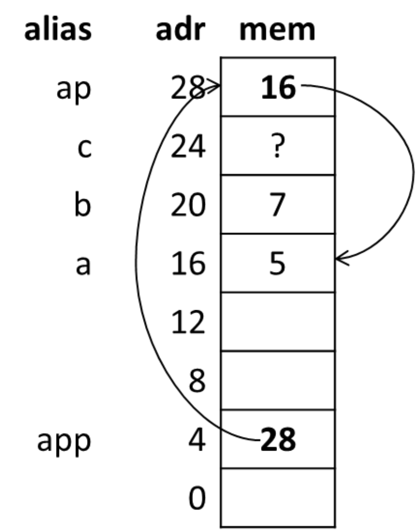
Variables and Memory

- **Where is the data allocated in memory?**
  - global variables
    - data/rodata/bss section
  - local variables
    - user stack
  - dynamically allocated data (later)
    - heap

```
int a = 5;
const b = 7;
int c;

int foo(int a, int b)
{
  int e, f, g;
  char *m = malloc(10);
}
```



HOCHSCHULE LUZERN



Pointers and Memory Allocation

HOCHSCHULE LUZERN

Pointers

- Variable names are aliases to memory addresses
  - we cannot know what the address is directly
- **Pointers are explicit memory addresses**
  - if we need to know where in memory an address is
- Pointers are also used to refer to arrays

HOCHSCHULE LUZERN

Pointers

- **Example**

```
int a = 5, b = 7, c;
int *ap = &a;
```

  - compiler allocates
    - a,b,c at addresses 16,20,24
    - ap at address 28
      - *ap itself is allocated somewhere in memory*

alias	adr	mem
ap	28	**16**
c	24	?
b	20	7
a	16	5
	12	
	8	
	4	
	0	

HOCHSCHULE
LUZERN

Pointers

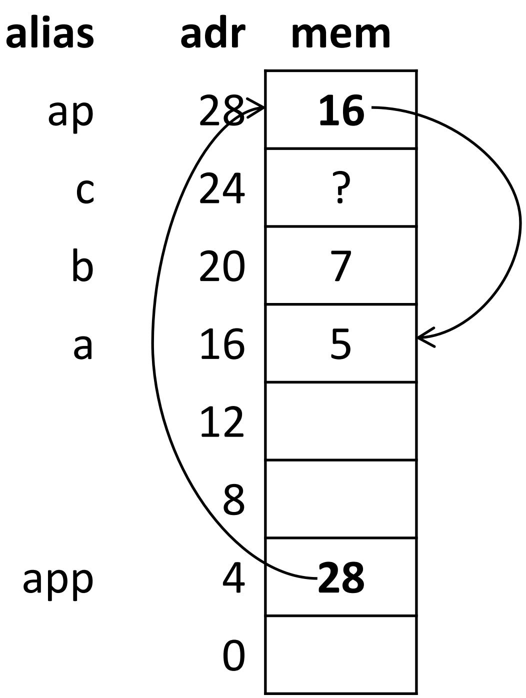
- **Example**

```
int a = 5, b = 7, c;
int *ap = &a;
```

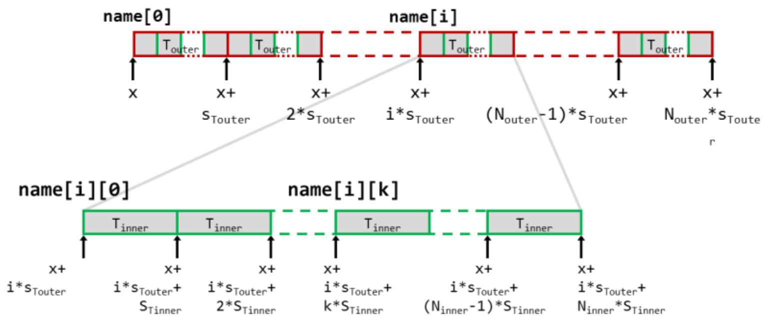
- compiler allocates
  - a,b,c at addresses 16,20,24
  - ap at address 28
    - *ap itself is allocated somewhere in memory*

```
…
int **app = &ap;
```

- compiler allocates
  - app at address 4



HOCHSCHULE LUZERN



Composite Data Structures

HOCHSCHULE LUZERN

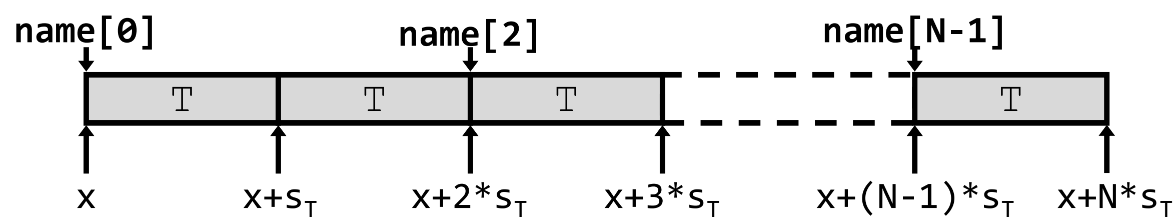
Arrays

- **Declaration**

```
<T> name[<N>]
```

- **Size**
  - one element: $s_T = \texttt{sizeof(T)}$
  - entire array: $s_A = N * s_T$

- **Memory layout**



- **Address of i-th element**

$$adr_i = name + i*s_T$$

HOCHSCHULE LUZERN

Multidimensional Arrays

- **formed when <type> is an array type**

$$\texttt{<}T_{outer}\texttt{> name[<}N_{outer}\texttt{>]}$$

$$\texttt{<}T_{outer}\texttt{> = <}T_{inner}\texttt{> …[<}N_{inner}\texttt{>]}$$

combined notation

$$\texttt{<}T_{inner}\texttt{> <name>[}N_{outer}\texttt{][}N_{inner}\texttt{]}$$

- **Size**
  - one element: $s_{T_{outer}} = \texttt{sizeof}(T_{outer}) = N_{inner} * s_{T_{inner}}$
  - entire array: $s_{A_{outer}} = N_{outer} * s_{T_{outer}} = N_{outer} * N_{inner} * s_{T_{inner}}$

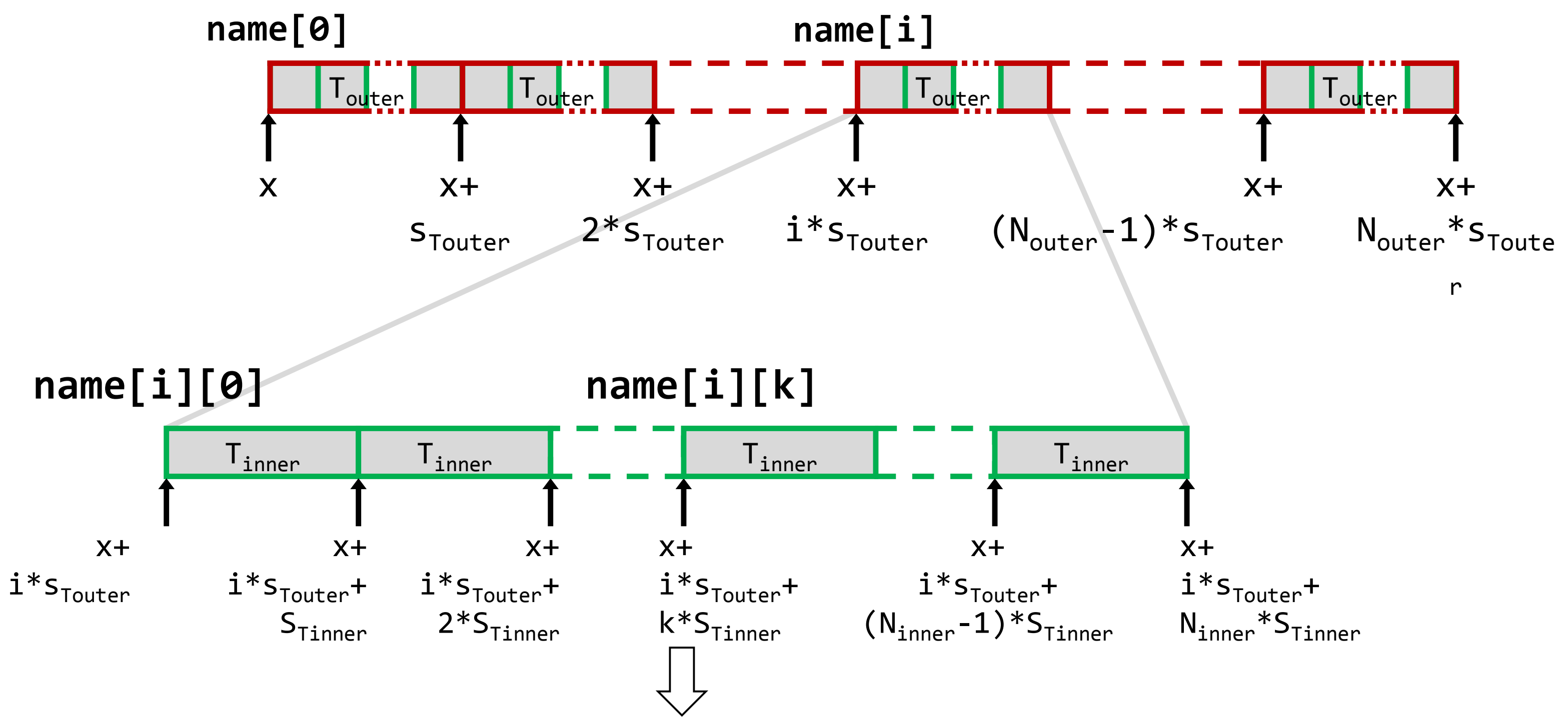
Multidimensional Arrays

$\langle T_{outer} \rangle\ \mathtt{name}[\langle N_{outer} \rangle]$

$\langle T_{outer} \rangle = \langle T_{inner} \rangle \ldots [\langle N_{inner} \rangle]$

$\langle T_{inner} \rangle\ \langle \mathtt{name} \rangle [N_{outer}][N_{inner}]$

- Memory layout (“row-major” layout)



- Address of element [i,k]
  - note how N_{outer} is not needed to compute name[i][k]

$x + i*N_{inner}*S_{Tinner} + k*S_{Tinner} \Rightarrow \mathtt{name} + S_{Tinner}*(k + N_{inner}*i)$

HOCHSCHULE LUZERN

Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\texttt{<}T_{base}\texttt{> <name>}[N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Size**
  - entire array

$$s_{D_n} = N_{D_n} * s_{T_{Dn-1}} = N_{D_n} * N_{D_{n-1}} * s_{T_{Dn-2}} = \ldots = \mathbf{N_{D_n} * N_{D_{n-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

  - subdimension k (n ≥ k ≥ 1)

$$s_{D_k} = N_{D_k} * s_{T_{Dk-1}} = \ldots = \mathbf{N_{D_k} * N_{D_{k-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

HOCHSCHULE LUZERN

Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\texttt{<}T_{base}\texttt{> <name>}[N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Address**
  - $\texttt{name}[i_{D_n}][i_{D_{n-1}}]\ldots[i_{D_2}][i_{D_1}]$

$$\mathbf{name} + s_{T_{base}} * (i_{D_1} + N_{D_1} * (i_{D_2} + N_{D_2} * (\ldots + N_{D_{n-1}} * i_{D_n})\ldots))$$

  - again, note how $\mathbf{N_{D_n}}$ is not required for the address computation

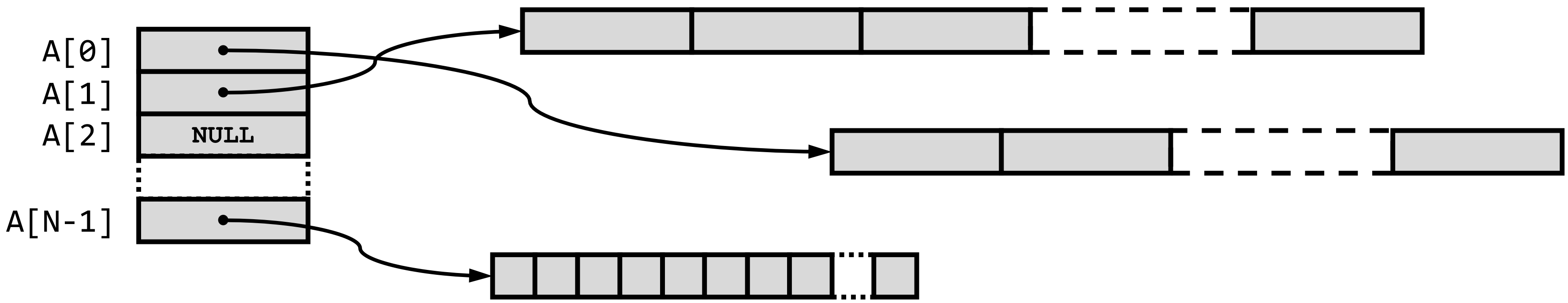
    → this is why you can declare arrays with an open outermost dimension such as

```
int A[][N][K];
```

Multilevel Arrays

- A multilevel array is simply an array of pointers to another array

```
int *A[N]
```



- pointed-to arrays can have different sizes
- elements in the pointer array can be NULL
- one extra memory access per pointer indirection
- both arrays, pointer array and pointed-to arrays can be multidimensional
- address calculation separate
  pointer array: element = pointer; pointed-to array: any type

Pointer Arithmetic

- C uses “weird” pointer arithmetic

```
int A[10];
int *a = A, *ep = A[10];

for (i=0; i<10; i++)
  printf(“%d”, A[i]);

while (a < ep) {
  printf(“%d”, *a++);
}
```

  - a++ increments the pointer value by the size of the base element (sizeof(int) = 4)
    a = 20, 24, 28, 32, …,60 (stop)

alias		**adr**	**mem**
ep		64	**60**
a		60	**20**
	A[9]	56	
	…		
	A[4]	36	
	A[3]	32	
	A[2]	28	
	A[1]	24	
A	A[0]	20	

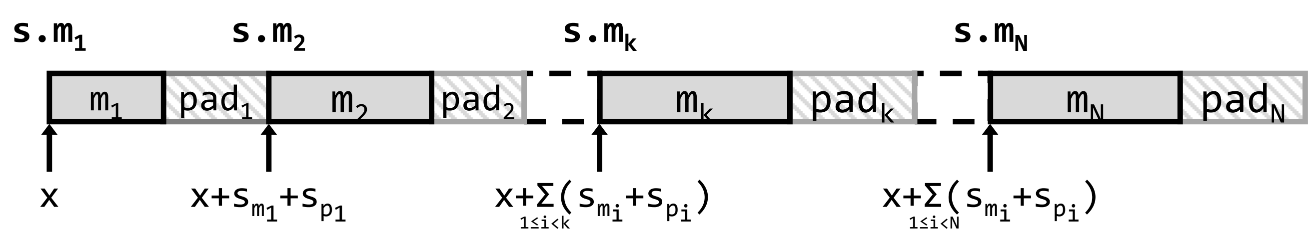
HOCHSCHULE LUZERN

Structures

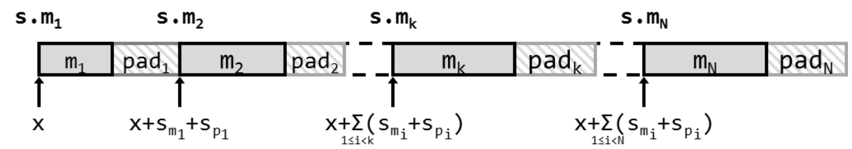
- **Declaration**

```
struct name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i in-order, non-overlapping, and properly aligned



Structures



- **Address of k-th member**
  - start of struct plus sum of sizes of all 1..k-1 members and paddings

$$adr_{m_k} = x+\sum_{1\le i<k}(s_{m_i}+s_{p_i})$$

- **Size of struct**

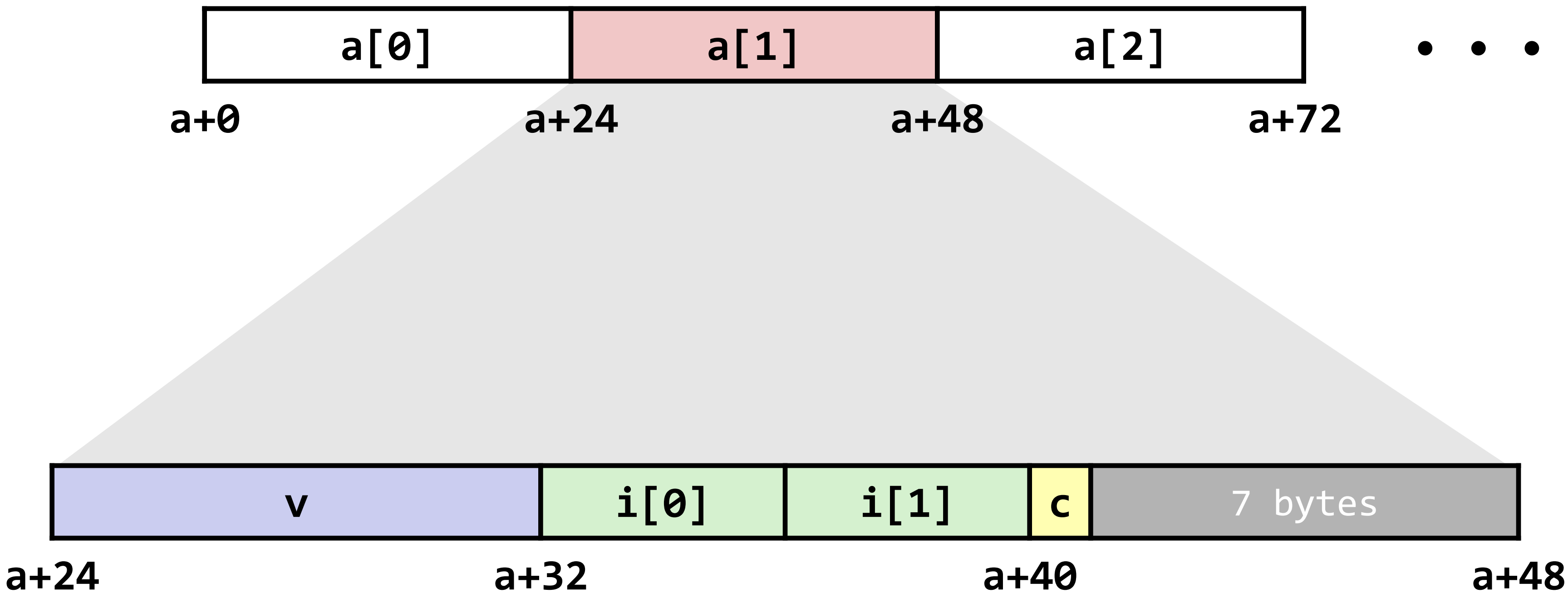
$$s_s = \sum_{1\le i\le N}(s_{m_i}+s_{p_i})$$

  - the last padding (pad_N) is chosen such that the size of the struct is the next multiple of the biggest alignment requirement of any of its members
    - this comes in handy when declaring arrays of structs (all elements of the array will be automatically aligned)

Example: Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

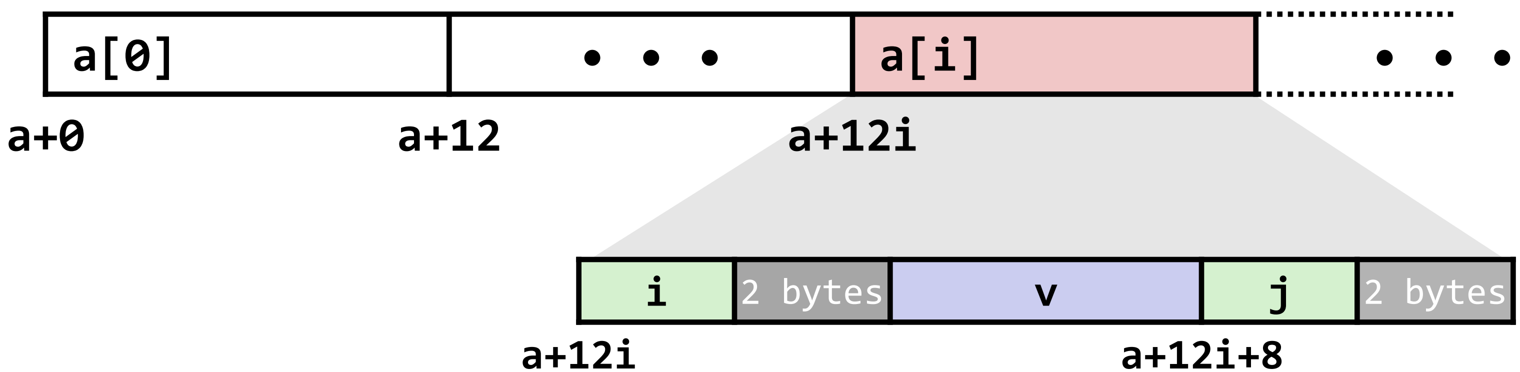
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Example: Accessing Array Elements

- Compute array offset 12i
  - sizeof(S3), including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```



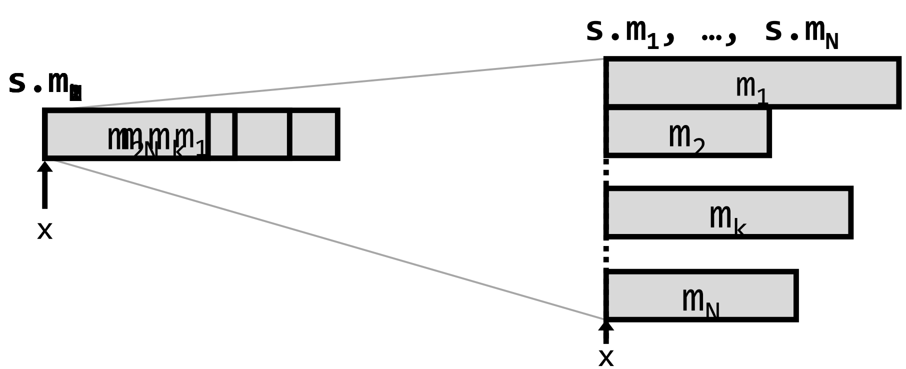

```
short get_j(int idx)
{
  return a[idx].j;
}
```

Unions

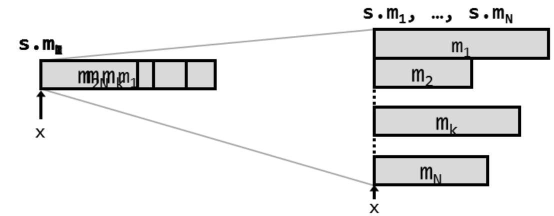
- **Declaration**

```
union name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i, and properly aligned
  - all members *are located at offset 0 and overlap in memory*



Unions



Alignment

- union alignment = maximum alignment requirement of any of its members

Address of k-th member

- start of union

$$adr_{m_k} = x$$

Size of union

$$s_u = \max_{1 \le i \le N}(s_{m_i})$$

HOCHSCHULE LUZERN

Example: Using Unions to Access Bit Patterns

- Task: print bit pattern of a floating point number

- Try 1:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  return (unsigned int)f;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
```

convert1.c

The assembly reveals

```
get_bitpattern:
  subl    $20, %esp
  fnstcw  14(%esp)
  movzwl  14(%esp), %eax
  flds    24(%esp)
  movb    $12, %ah
  movw    %ax, 12(%esp)
  fldcw   12(%esp)
  fistpq  (%esp)      # convert float to int
  fldcw   14(%esp)
  movl    (%esp), %eax
  addl    $20, %esp
  ret
```

Indeed, this is a conversion float→int!

Output looks suspiciously wrong:

```
$ gcc -m32 -O3 -o convert1 convert1.c
$ ./convert1
3.1415927410 = 00000000000000000000000000000011b = 0x00000003
```


HOCHSCHULE LUZERN

Example: Using Unions to Access Bit Patterns

- Try 2:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  union {
    float f;
    unsigned int u;
  } fu;
  fu.f = f;
  return fu.u;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
```

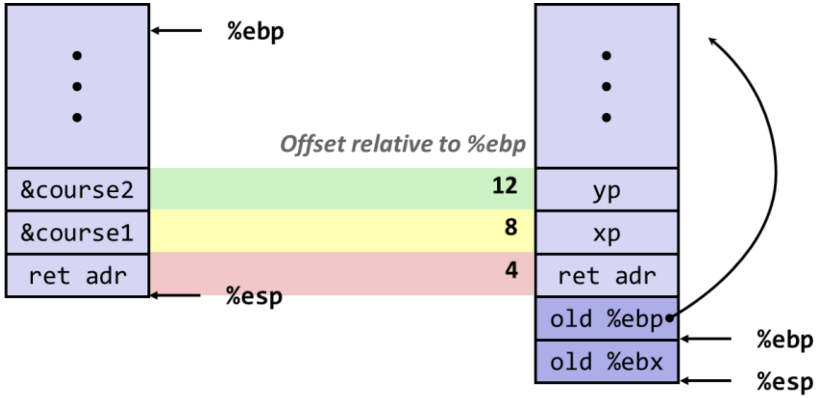
convert2.c

This is what we want

```
get_bitpattern:
  movl    4(%esp), %eax
  ret
```

Output:

```
$ gcc -m32 -O3 -o convert2 convert2.c
$ ./convert2
3.1415927410 = 01000000010010010000111111011011b = 0x40490fdb
```



Parameter Passing

HOCHSCHULE LUZERN

Parameter Passing

- Caller vs callee
  - caller
    the function calling another function
  - callee
    the function being called
- Caller can pass arguments to a callee
- Two ways
  - pass by value
  - pass by reference

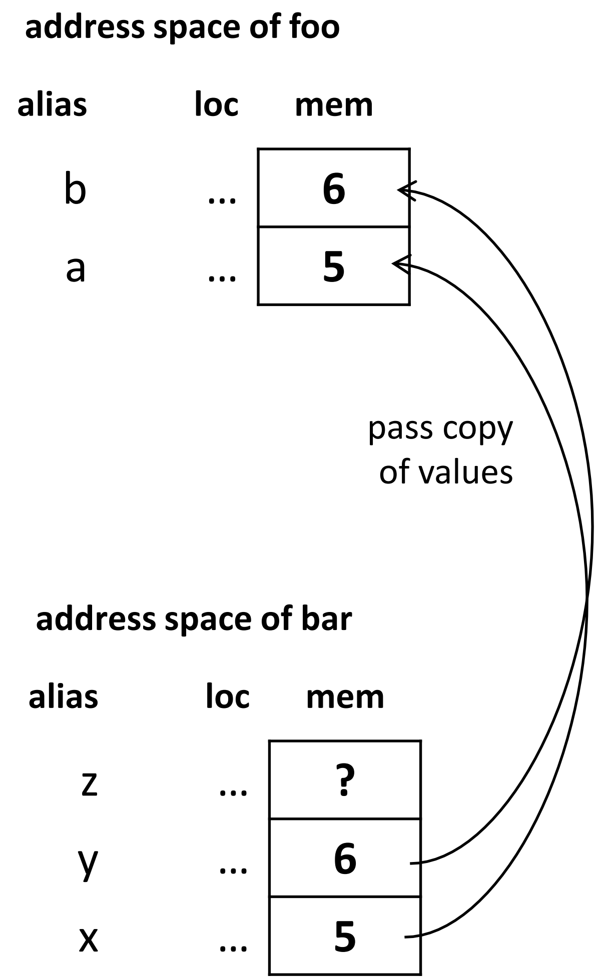
```c
int foo(int a, int b)
{
  a = 2*a;
  return a + b;
}

int bar(void)
{
  int x = 5, y = 6, z;

  z = foo(x, y);
}
```


HOCHSCHULE LUZERN

Parameter Passing

- **Pass by value**
  - pass a copy of the value to the callee
  - modifications to arguments local to callee
  - scalar types are passed by value

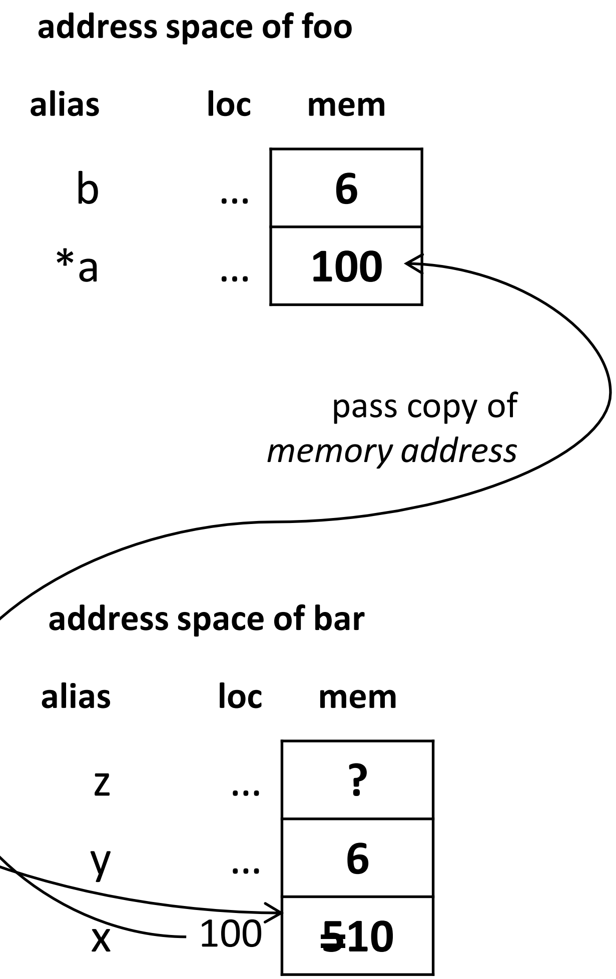
```
int foo(int a, int b)
{
  a = 2 * a;
  return a + b;
}

int bar(void)
{
  int x = 5, y = 6, z;

  z = foo(x, y);
}
```



HOCHSCHULE LUZERN

Parameter Passing

- **Pass by reference**
  - instead of passing a copy of the value, a **copy of the value's memory address** is passed to the callee
  - modifications to arguments global

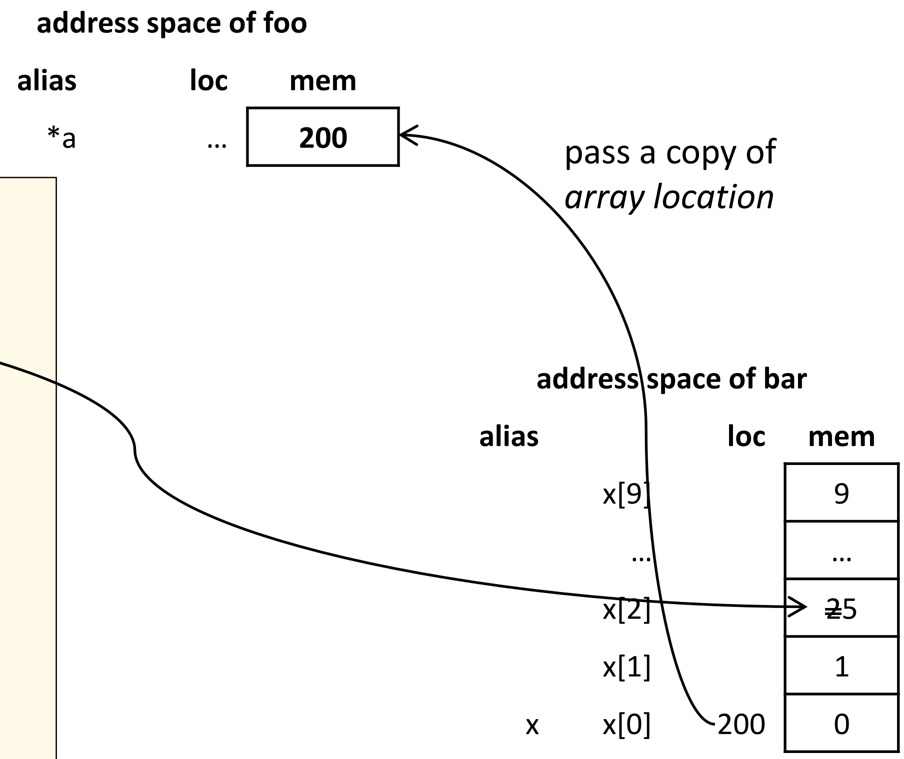



```
int foo(int *a, int b)
{
  *a = 2 * *a;
  return *a + b;
}

int bar(void)
{
  int x = 5, y = 6, z;

  z = foo(&x, y);
}
```

Parameter Passing

- **Pass by reference**
  - arrays are always passed by reference
  - makes sense, we know that arrays are simply pointers

```
int foo(int *a)
{
  a[2] = 5;
}

int bar(void)
{
  int x[10] = {0,1,2,3,4,5,6,7,8,9};

  printf("%d", x[2]); // prints 2
  foo(x);
  printf("%d", x[2]); // prints 5
}
```

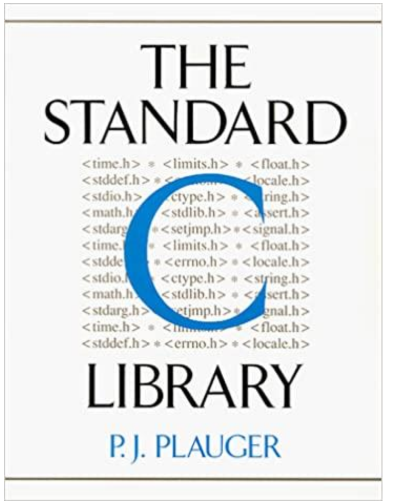


HOCHSCHULE LUZERN

Parameter Passing

- **Command line parameters**
  - command line parameters are passed to main
    - argc: number of arguments
    - argv[]: array of pointers to char (=strings)
    - argv[0] is the program name (i.e., argc >= 1)

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

  for (i=0; i<argc; i++) {
    printf("argv[%d] = '%s'\n",
           i, argv[i]);
  }

  return 0;
}
```



The C Standard Library

The C Standard Library

- **C comes with a standardized library**
  - wide range of functions
  - system calls (integration with OS) defined by POSIX standard

  - string manipulation
  - input/output
  - file access
  - dynamic memory management

  - C header files are typically found in
    /usr/include/*

  - code in static/dynamic libraries
    /usr/lib64/libc.a,so

HOCHSCHULE LUZERN

The C Standard Library

- **C comes with a standardized library**
  - use man pages to learn about functions

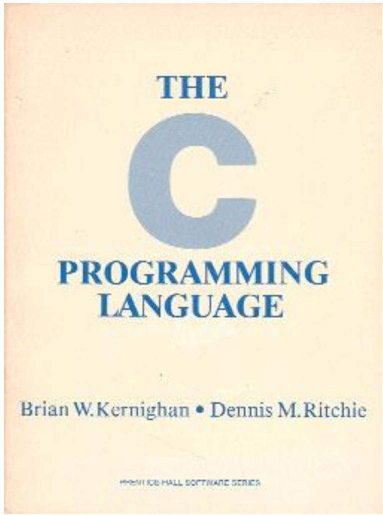
    $ man <section> <function>

  - Sections?

    $ man man

  - Don't know the function name?

    Wikipedia and
    DuckDuckGo are your friends



Module Summary

Programming in C

- **Properties of C**
  - Procedural language (no classes)
  - Supports basic set of control flow structures and operators
  - Strongly typed, although the type system allows easy circumvention
  - Composite data types such as arrays, structs, and unions
    - strings are null-terminated character arrays
  - Modules allow for separate compilation

- **Variables and Pointers**
  - Variables are aliases for actual memory addresses
  - Pointers are variables that contain another memory address as their data
  - Parameters can be passed by value (copy of data) or by reference (pointer to data)

- **C Standard Library**
  - Standardized extensive support from low- to high-level functionality

HOCHSCHULE LUZERN