P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #11

Pointers

memory address, dereferencing, pointer arithmetic,
dynamic memory management

Variables (again)

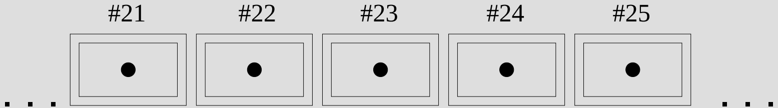
Aspects

In an imperative language four aspects make up a variable:

1. the memory area serving as storage container for the data stored in the variable
2. the address of the memory area
3. the data stored in the memory area
4. the variable's name to access the data in the memory area

What is a Pointer?

- Computer memory can be imagined as a sequence of drawers, that store your data



- Pointers are variables for storing memory addresses (drawer numbers).
- They allow us to locate our data in memory.

Pointer Definition

- to declare a pointer variable one uses the * token
- we note that a pointer variable always is associated with a data type, i.e. it can only store addresses for objects of that fixed type
- thus, `iPtr` is called a pointer to int or briefly an int pointer; and so on
- spaces around the * do not matter; though `int* iPtr` more clearly expresses that `iPtr` is an int pointer

```
int* iPtr;
double *dPtr;
float * fPtr;
```

```
// careful with lists:

short s, *sPtr;
// s (short)
// sPtr (short pointer)

int* iPtr1, iPtr2;
// iPtr (int pointer)
// iPtr2 (int)
```

Obtaining & Storing Addresses

- In order to assign a value to a pointer, we need a way to find the memory address of an object.
- For this there is the address operator & .
- We can also copy the value from one pointer into another one.

```
double dVal;
double *dPtr = &dVal;
int *iPtr1, *iPtr2;
int iVal;
iPtr1 = &iVal;

iPtr2 = iPtr1;
iPtr2 = dPtr;   // dubious as pointers have different type!
```

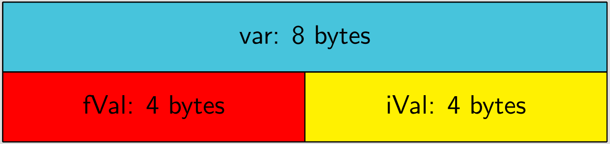
- After the assignment `iPtr1 = &iVal;` we say that `iPtr1` points to `iVal`.

Addresses (1/2)

- Most data types occupy more than a single word. So what is the address of a variable?
- Let us examine the following example:

```
typedef struct { float fVal; int iVal; } compound;
compound var;
```

- In memory `var` will (typically) occupy eight bytes



Addresses (2/2)

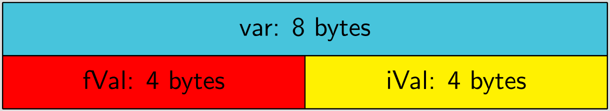
- this code

```
compound* cPtr = &var;
float* fPtr = &var.fVal;
int* iPtr = &var.iVal;
printf( "cPtr = %p\n", cPtr );
printf( "fPtr = %p\n", fPtr );
printf( "iPtr = %p\n", iPtr );
```

- might print

```
cPtr = 0x7ffc23fc2bb0
fPtr = 0x7ffc23fc2bb0
iPtr = 0x7ffc23fc2bb4
```

(address details are runtime dependent; distance between fPtr and iPtr is not)



- an object's address is always that of its first byte

Dereferencing

- The `*` operator allows to access the memory location a pointer points to.
- This is called dereferencing the pointer.

```
int a = 3, b = 0;
int* ptr;
ptr = &a;
b = *ptr;
*ptr = 5;
printf( "a=%d, b=%d\n", a, b );
```

- Dereferencing allows to read or write data from that memory address.
- Code above will print `a=5, b=3`.

Null Pointer

- What will happen, when we run this code?

```
int* ptr;
*ptr = 5;
```

Null Pointer

- What will happen, when we run this code?

```
int* ptr;
*ptr = 5;
```

- Most likely this:

```
Segmentation fault (core dumped)
```

The pointer variable `ptr` was not initialised. Thus, it will point to some arbitrary memory address!

Null Pointer

- What will happen, when we run this code?

```
int* ptr;
*ptr = 5;
```

- Most likely this:

```
Segmentation fault (core dumped)
```

The pointer variable `ptr` was not initialised. Thus, it will point to some arbitrary memory address!

- How can we find out if a pointer is unitialised?

Null Pointer

- What will happen, when we run this code?

```
int* ptr;
*ptr = 5;
```

- Most likely this:

```
Segmentation fault (core dumped)
```

The pointer variable `ptr` was not initialised. Thus, it will point to some arbitrary memory address!

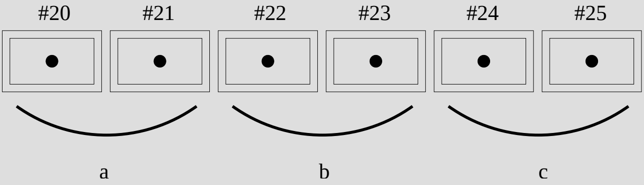
- How can we find out if a pointer is unitialised?
- We initialise it to an invalid, but well defined value. C provides the macro `NULL` for this purpose:

```
double* dPtr = NULL; // good style
```

Pointer Arithmetics

```
short int a, b, c, *ptr;
```

Assume that a short integer occupies two drawers and that `a`, `b` and `c` are stored one after the other



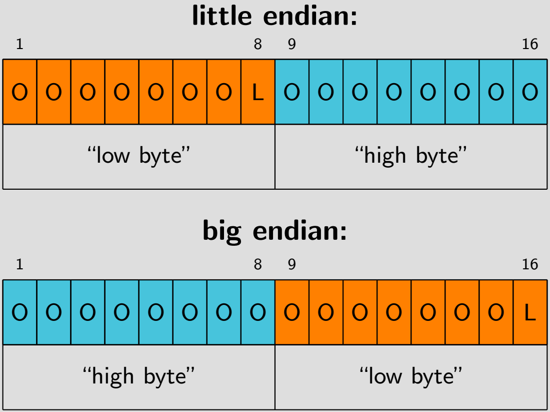

```
ptr = &a;           /* ptr -> drawer #20 */
ptr = ptr + 2;      /* ptr -> drawer #24 */
a = *ptr;           /* a = c */
```

Example: Endianess Test (1/2)

```
#include <stdio.h>

// for uint*_t types
#include <stdint.h>

int main() {

  uint16_t i = 1u;
  uint8_t* c = (uint8_t*)&i;

  if( *c )
    printf( "little endian\n" );
  else
    printf( "big endian\n" );

}
```

- Code on the left will report the machine's endianess.
- Works as follows:
  1. define a 2-byte unsigned integer variable and initialise it to 1
  2. get its (starting) address and cast it to a 1-byte unsigned integer pointer
  3. dereference pointer and check value

Example: Endianess Test (2/2)



code accesses value stored in first byte;

if value of bit pattern in non-zero we have a little endian system

Example: Taming the Panda

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

int main() {

  char v1[] = "Eats, shoots and leaves";
  //           01234567890123456789012
  //                     ^         ^
  char v2[ strlen( v1 ) ];

  printf( "'%s'\n", v1 );  // prints 'Eats, shoots and leaves'!

  strncpy( v2, v1, 4 );
  strcpy( &v2[4], &v1[5] );
  // or: strcpy( &v2[0]+4, &v1[0]+5 );
  // or: strcpy( v2+4, v1+5 );

  printf( "'%s'\n", v2 );  // prints 'Eats shoots and leaves'!
}
```

Another Example

Examine the following codelet:

```
1   int a;
2   int* x;
3   int** y;
4   y = &x;
5   x = &a;
6   **y = 1;
7   *x = a + **y;
8   a = *x + **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes . . .

. . . and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #3

int	**	y	$\otimes$	1012
int	*	x	$\otimes$	1004
int		a	$\otimes$	1000

$\otimes$ = garbage

Another Example

Examine the following codelet:

```
1  int a;
2  int* x;
3  int** y;
4  y = &x;
5  x = &a;
6  **y = 1;
7  *x = a + **y;
8  a = *x + **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes ...

... and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #4

int	**	y	1004	1016
int	*	x	$\otimes$	1004
int		a	$\otimes$	1000

$\otimes$ = garbage

Another Example

Examine the following codelet:

```
1  int a;
2  int* x;
3  int** y;
4  y = &x;
5  x = &a;
6  **y = 1;
7  *x = a + **y;
8  a = *x + **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes ...

... and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #5

int	**	y	1004	1012
int	*	x	1000	1004
int		a	$\otimes$	1000

$\otimes$ = garbage

Another Example

Examine the following codelet:

```
1  int a;
2  int* x;
3  int** y;
4  y = &x;
5  x = &a;
6  **y = 1;
7  *x = a + **y;
8  a = *x + **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes ...

... and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #6

int	**	y	1004	1012
int	*	x	1000	1004
int		a	1	1000

$\otimes$ = garbage

Another Example

Examine the following codelet:

```
1   int  a;
2   int*  x;
3   int**  y;
4   y  =  &x;
5   x  =  &a;
6   **y  =  1;
7   *x  =  a  +  **y;
8   a  =  *x  +  **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes . . .

. . . and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #7

int	**	y	1004	1012
int	*	x	1000	1004
int		a	2	1000

$\otimes$ = garbage

Another Example

Examine the following codelet:

```
1 int a;
2 int* x;
3 int** y;
4 y = &x;
5 x = &a;
6 **y = 1;
7 *x = a + **y;
8 a = *x + **y;
```

We assume that an int occupies 4 bytes and a pointer 8 bytes . . .

. . . and that the variables are residing in memory one after the other starting at address 1,000.

memory occupation: after line #8

int	**	y	1004	1012
int	*	x	1000	1004
int		a	4	1000

$\otimes$ = garbage

Member Access “Syntactic Sugar”

```
typedef struct {
  double x;
  int m;
} compound;

int main( void ) {

  compound cVar;
  cVar.x = 0.0;
  cVar.m = 1;

  compound* cPtr = &cVar;
  (*cPtr).m = 3;
  // *cPtr.m = 3;  <- error!
  cPtr->x = 2.0;
}
```

What are Pointers Good for?

- Causing confusion ;-)
- Dynamic memory management
- Changing variables from within subprograms
- Linked lists, tree data structures, ...

Dynamic Memory Management

- In many applications the amount of memory required is not known a priorily at compile time (e.g. importing an unstructured mesh)
- Dynamic memory management means that memory is requested by the program at run time (from the OS).
- Pointers are closely related to dynamic memory management.
- Routines that allocate memory return pointers to the starting address of the allocated memory block.

Library Functions from `stdlib.h`

```
void* malloc(size_t size);
```

`malloc()` allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared!

```
void* calloc(size_t nmemb, size_t size);
```

`calloc()` allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero!

```
void free(void* ptr);
```

`free` frees the memory space pointed to by ptr, which must have been returned by a previous call to `malloc()` or `calloc()`.

Datatype `void`

- C has a special `void` datatype.
- It is incomplete, i.e. we cannot declare variables of type `void`.
- However, we can have void pointers
  - no pointer arithmetic!‡
  - no dereferencing!
- We will encounter `void` again in the unit on functions.

‡ GCC will allow this as language extension

```c
#include <stdlib.h>

int main() {

  // not allowed!
  // void freeVar;

  void* freePtr;

  double x = 2.0;
  freePtr = (void*) &x;

  double y;

  // not allowed!
  // y = *freePtr;

  y = *(double*)freePtr;

}
```

Example: Allocate 1D Array (1/2)

Allocate memory for a vector of four doubles $(v_1, \ldots, v_4)^T$:

```
double *v = NULL;
v = (double*) calloc( 4, sizeof(double) );
```

Memory:	v_1	v_2	v_3	v_4
Access:	`*(v+0)`	`*(v+1)`	`*(v+2)`	`*(v+3)`
Shorter:	`v[0]`	`v[1]`	`v[2]`	`v[3]`

equivalent: $\boxed{*(v+i) = v[i]}$

Example: Allocate 1D Array (2/2)

- With dynamic arrays we are not fixed to starting arrays at index 0!

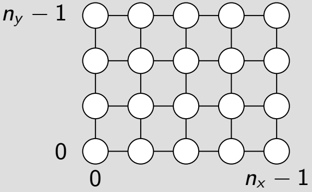
```
double *v = NULL;
v = (double*) calloc( 4, sizeof(double) );
v--;    /* equiv to v = v - 1; */
```

- Now `v` points to a position `sizeof(double)` bytes in front of our allocated memory block!
- `v[1]` now points to v_1 (by pointer arithmetics).
- Technique is (nearly) legal, but never try to access `v[0]`
(But: out-of-bounds accesses are always a bad idea!)

```
#include<stdio.h>  // by Chr. Feichtinger
#include<stdlib.h>

int main() {
  int i, n;
  char* buffer = NULL;

  printf( "How long should the string be?\n" );
  scanf( "%d", &i );

  buffer = (char*)malloc( sizeof(char) * (i+1) );
  if( buffer == NULL ) {
    fprintf( stderr, "Memory allocation failed!\n" );
    exit(1);
  }
  for( n=0; n<i; ++n )
    buffer[n] = rand()%26 + 'a';

  buffer[i] = '\0';
  printf( "Random string: %s\n", buffer );

  free( buffer ); return 0; }
```

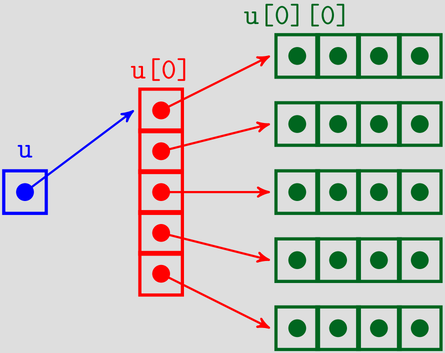
Dynamic Allocation of 2D Arrays

- Let u be a function "living" on a 2D grid: $\boxed{u_{i,j} := u(x_i, y_j)}$



- We want to dynamically allocate memory to store the values of u.
- There are several ways to do this. We will examine three of them.

Variant #1




```
// u is of type pointer to pointer
// to double
double** u = NULL;

// allocate memory for holding
// addresses of mesh columns
u = (double**)
   malloc( nx * sizeof(double*) );

// allocate memory for entries
// in mesh columns
for( int i = 0; i < nx; i++ ) {
  u[i] = (double*)
   calloc( ny, sizeof(double) );
}
```

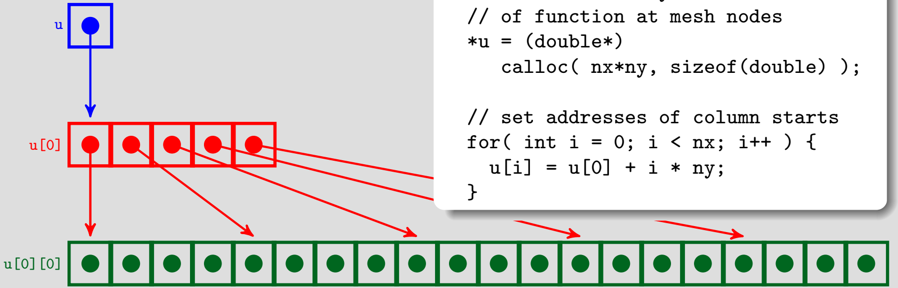
We can access values as: $u_{i,j} \equiv$ `u[i][j] = *(*(u+i)+j)`

Variant #2

same as variant #1, but values of *u* are stored in a contiguous memory area

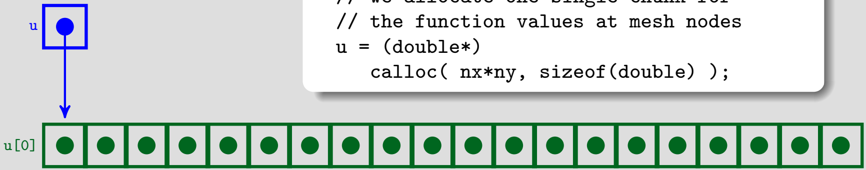
access again via `u[i][j]`

```
// u is of type pointer to pointer
// to double
double** u = NULL;

// allocate memory for holding
// addresses of columns starts
u = (double**)
   malloc( nx * sizeof(double*) );

// allocate memory for values
// of function at mesh nodes
*u = (double*)
   calloc( nx*ny, sizeof(double) );

// set addresses of column starts
for( int i = 0; i < nx; i++ ) {
  u[i] = u[0] + i * ny;
}
```



Variant #3

```
// u now is of type pointer to double
double* u = NULL;

// we allocate one single chunk for
// the function values at mesh nodes
u = (double*)
   calloc( nx*ny, sizeof(double) );
```



- similar to variant #2 we store all values of the function u consecutively in a contiguous memory area
- however, we skip the intermediate layer with the addresses of the column starting positions
- need to do address computation ourselves: $u_{i,j} \equiv$ `u[i*ny+j]`

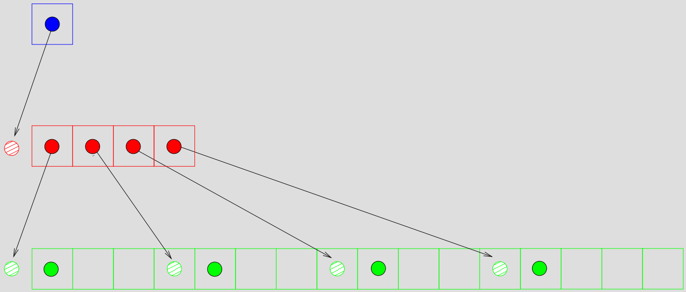
Comparison

- Pointer/address calculations for variants (1) and (2) are basically as expensive as the mapping in (3).
- Type (1) and (2) require additional storage for the pointer vectors.
  (1) also demands more internal administration work for malloc than (2)
- Number of memory accesses to get value of $u_{i,j}$
  - Type (1): three memory accesses
  - Type (2): three memory accesses
  - Type (3): two memory accesses
- With (1) there is no control, where the data are placed in memory (paging and caching effects!).

Variant (2) seems a good compromise between efficiency and usability.

Adding Offsets

As with the 1D array allocation, we can also add offsets to our pointers to e.g. get one-based indexing for a matrix:



sizeof: static vs. dynamic

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

int main( void ) {

  int staticVec[10] = {0};
  int* dynamicVec = (int*)calloc( 10, sizeof(int) );

  printf( "sizeof(staticVec)  = %2lu\n", sizeof(staticVec)  );
  printf( "sizeof(dynamicVec) = %2lu\n", sizeof(dynamicVec) );

}
```

sizeof: static vs. dynamic

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

int main( void ) {

  int staticVec[10] = {0};
  int* dynamicVec = (int*)calloc( 10, sizeof(int) );

  printf( "sizeof(staticVec)  = %2lu\n", sizeof(staticVec)  );
  printf( "sizeof(dynamicVec) = %2lu\n", sizeof(dynamicVec) );

}
```

```
sizeof(staticVec) = 40
sizeof(dynamicVec) = 8
```

sizeof: static vs. dynamic

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

int main( void ) {

  int staticVec[10] = {0};
  int* dynamicVec = (int*)calloc( 10, sizeof(int) );

  printf( "sizeof(staticVec)  = %2lu\n", sizeof(staticVec)  );
  printf( "sizeof(dynamicVec) = %2lu\n", sizeof(dynamicVec) );

}
```

```
    sizeof(staticVec) = 40
    sizeof(dynamicVec) = 8     <-- size of a pointer variable!
```