Why should you think about memory allocation strategies?
Global vs local variables
Heap memory
Free-lists
Alloca and arenas
Struct layout for reduced size
Why memory allocation strategies are important

- It affects your productivity — some strategies are more convenient than others
- Some are dangerous or don’t work in certain situations
- Sometimes they affect the performance of your program
The address of a global variable

- A local variable is usually put in a register and hence has no address in memory.
- The address of a struct member accessed through a pointer is the content of the pointer plus a constant.
- The constant specifies how many bytes from the start of the struct the variable is located within the struct.
- The address of a global variable is, almost always, a 32 bit constant.
Note that RISC processors cannot put a 32 bit constant into a register in one instruction.

Since the instruction itself is 32 bits — a 32 bit constant in an instruction would leave zero bits to tell the machine what to do.

32 bit constants are put in a register using two instructions.

First 16 bits which are put at the upper half of the word and then 16 more bits are put at the lower half.

Note: floating point constants such as $\pi$ in $a = r * r * 3.14$ are loaded from memory using two instructions. For most optimising compilers, a floating point constant is like a readonly global variable.
Primitive data types have alignment requirements — usually equal to their size, which is called \textbf{natural alignment}.

A surrounding struct gets the alignment from its members.

Sizes become: struct s: 32 bytes, struct t: 24 bytes. See next page.

```c
struct s {
    double a;
    char b;
    double c;
    char d;
};

struct t {
    double a;
    double c;
    char b;
    char d;
};
```
You might have thought that struct t on the previous slide would have size 18 bytes.

Consider then the following: struct t x[100];

Assume the size of struct t is 18 bytes.

The double variables in x[0] would be properly aligned.

But the double variables in eg x[1] would not be.

Therefore the compiler puts padding at the end of a struct to make its size a multiple of its alignment requirements.
Cost of accessing scalar variables

- A scalar variable is a pointer or of arithmetic type.
- On modern machines (i.e., not PCs with X86), a variable must first be loaded to a register before it can be used in an arithmetic operation.

<table>
<thead>
<tr>
<th>Kind of variable</th>
<th>Example</th>
<th>Instructions needed for loading it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>x</td>
<td>None since it usually is allocated to a register</td>
</tr>
<tr>
<td>Struct member</td>
<td>p-&gt;x</td>
<td>One LOAD using the pointer</td>
</tr>
<tr>
<td>Global or static</td>
<td>x</td>
<td>Two, see below in simplified Power assembler</td>
</tr>
</tbody>
</table>

- addis r3,r0,ha16(x) // add immediate and shift: 16 upper bits
- lwz r3,lo16(x)(r3) // load: 16 lower bits of address + R3
Global variables

- May make your program harder to read and modify. **Avoid them.**
- However, for large matrices, if you can know the size of the input, global variables are best because they help the compiler produce the fastest code.
- Flags which are parameters to your program often are global variables. They are set once at program start and then never changes so using global variables for those does not make your code more complicated to understand.
- If you later want to multithread your program to exploit a multiprocessor, then accesses to global variables often must be protected with locks. However, that is not studied in this course.
Local variables

- When possible, use them because they are the easiest for the compiler to optimise.
- Don’t think they get a default initial value. They don’t in C.
- Local variables disappear when the function returns. Consequences:
  - It is meaningless to return the address of a local variable (advanced trick: do just that if you want to know whether the stack grows towards lower or higher addresses — not recommended because it triggers undefined behaviour but it can be useful (eg to implement alloca)).
  - If you use a VLA (ie variable-length array), which is a local variable, you cannot return the address of the VLA either! Use the VLA in the function \( f \) which allocated it or in functions called by \( f \).
  - You can return a struct because its value is copied.
Size constraints of local variables

- Too big local variables can cause your program to crash, especially if somebody later runs it on a multiprocessor.
- On a uniprocessor there is by default one thread running and it gets all the stack memory area — usually as much as it wishes.
- On a multiprocessor, however, by default the stack space for a thread is more limited.
Heap memory: calling malloc

- When you use `new` in Java or C++ or `malloc/calloc` in C, you request memory to be allocated from the so called **heap**.

- You get an address from malloc and the address is valid until the address is deallocated using `free`, which may be done in a function other than the one which called malloc.

- The heap memory is a data structure which can be organised in different ways and can grow in size when the application needs more memory.

- Calling malloc takes time because malloc must find a suitable address to give to you. Malloc must also make some book keeping to keep track of which data already has been allocated, and how much you got.
Heap memory: calling free

- Calling free also takes time because free must find out whether the freed memory can be merged with other memory to create a bigger "piece" in the heap.
- Forgetting to call free results in a memory leak.
- It is good practice to do `free(p); p = NULL;`
- It is important to know, of course, which part of your program "owns" memory allocated from malloc so that it is absolutely clear who should free that memory.
- Suppose you malloc memory and then give the memory to another part of the program. Who should deallocate (ie free) that memory? You decide, but do decide.
How can free know what to do?

- When you call free, you give it a pointer as the parameter.
- But how can free know what it should do with that address?
- The word located just before the address malloc gives you is used by malloc to store information which free can use to find out what to do.
- Therefore free gets completely confused (and usually crashes) if you return an address which had not been allocated by malloc.
Examples with malloc

- Sometimes one see code such as the following:

  ```c
  int* x = malloc(sizeof(int));
  ```

- Since using malloc/free takes time, one should avoid calling malloc for so small types.
When malloc cannot find any free memory, it returns NULL.

```c
list_t* p = malloc(sizeof(list_t));
if (p == NULL) {
    fprintf(stderr, "out of memory\n");
    exit(1);
}
```

Writing this everywhere you allocate memory might be tedious, increase code size, and reduce readability.

Write instead a constructor `new_list()` which does it.
You might want to avoid calling malloc directly:

```c
void* xmalloc(size_t size)
{
    void* p = malloc(size);

    if (p == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    } else
    return p;
}
```
If some of your data structures can be written down to disk, or you are for other reasons able to free memory, you can do:

```c
void* xmalloc(size_t size)
{
    void* p = malloc(size);
    if (p == NULL) {
        find_memory();
        p = malloc(size);
    }

    if (p == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }

    return p;
}
```
Free-lists

- Suppose you have a data structure, eg a tree or a list and you often perform the following:
  - 1. allocate some nodes
  - 2. work with the nodes
  - 3. deallocate the nodes

- You might want to avoid calling malloc and free. How?
- Don't free the nodes but rather keep them in a pool, or free-list, for future use.
typedef struct list_ts list_t;
struct list_t {
    list_t* succ;
    list_t* pred;
    void* data;
};

static list_t* xfree_list;

list_t* new_list(void* data)
{
    list_t* p;

    if (xfree_list != NULL) {
        p = xfree_list;
        xfree_list = xfree_list->succ;
    } else
    {
        p = xmalloc(sizeof(list_t));
    }
    ...
}
Example

- Assume the list is circular so (*head)->pred is the last node.
- The xfree_list only uses succ.

```c
static list_t* xfree_list;

void free_list(list_t** head)
{
    list_t* h = *head;

    if (h == NULL)
        return;
    h->pred->succ = xfree_list;
    xfree_list = h;
    *head = NULL;
}
```
We should let `find_memory` know about functions `deallocate_X_free_list`, with one such function for each data structure `X` for which you have a free-list.

For instance: use `deallocate_list_free_list` to force deallocation of all nodes on the free-list.

When `xmalloc` cannot find memory it can call `find_memory`.

This approach lets you avoid some `malloc/free` while not running out of memory due to the free-lists (since that memory will be reclaimed when needed).

You should still call `deallocate_list_free_list` so that your program does not use much more memory than needed. Again an engineering trade-off.
Alloca

- Alloca is not standard and is not supported by some systems. Not so easily implemented since it typically needs assembler code.
- Alloca is much faster than malloc and does not need free.
- Alloca uses the memory just after the current function’s part of the stack (i.e., where it saves local variables and the return address).
- Essentially what a call `p = alloca(n);` does is simply to subtract `n` from the stack pointer.
- When returning from a function, as you know, the stack pointer is adjusted to what it was before the function was called, which means the memory allocated with alloca is automatically deallocated.
int f(int n)
{
    int i;

    for (i = 0; i < 1000; i++) {
        int a[n];
        int* b = alloca(n * sizeof(int));
        // use A and B.
    }
}

- The variable-length array is allocated and deallocated in each loop iteration
- The memory from alloca is deallocated when the function returns.
An arena is a data structure which provides memory allocation somewhat similar to alloca.

Arenas are not provided by the language but they can be easily implemented.

An arena has basically three functions in its interface:

1. `arena_t* new_arena();` Create a new arena.
2. `void free_alloc(arena_t* arena);` Deallocate the arena.
3. `void* alloc(arena_t* arena, size_t size);` Allocate memory.

Memory allocated with `alloc` is not freed until the whole arena is freed.

Thus, the memory survives function return.
Advantages of arenas

- Allocating memory from an arena is as fast as from alloca, ie much faster than from malloc.
- You don’t have to deallocate individual objects so it’s easier and faster to use.
- The hidden word used by malloc/free (for free to understand what to do with a returned pointer) is not needed in arenas (since there is no free for each object) and thus arenas can save some memory especially for small objects:
  - Allocating small objects such as short strings with malloc is usually regarded as a bad idea because it wastes memory due to some overhead for each object.
  - Allocating small objects from an arena makes perfect sense since no memory is wasted.
Disadvantages of arenas

- An arena must own some memory, say some kilobytes or some megabytes depending on the application and needs.
- If only a small fraction of the memory allocated to the arena is actually allocated from the arena, the arena still occupies all memory allocated to it.
- Deallocation of an arena must, of course, wait until your program is no longer interested in any memory allocated from the arena.
Use Multiple Arenas

- If you use only one arena, you may have to wait until program termination to reclaim that memory since some odd objects still use it.
- Partition your memory allocation needs into short-term and long-term.
- Short-term means objects that you know can be thrown away at an exact source code line which you will reach relatively soon — such as return from the current function.
- Then, use one arena for those short-term objects.
- Use another arena for objects which will be needed for the rest of the program.
- Since they cannot be thrown away before the end, it makes sense to put them together in one arena.
#define ARENA_SIZE (1024 * 1024)
typedef struct arena_t arena_t;
struct arena_t {
    char* current;
    char buffer[ARENA_SIZE]
};

arena_t* new_arena(void) {
    arena_t* arena;
    arena = calloc(1, sizeof(arena_t));
    arena->current = &arena->buffer[0];
    return arena;
}
void* alloc(arena_t* arena, size_t size)
{
    size_t remain; // how much memory is left.
    void* data; // return this to user.

    remain = ARENA_SIZE - (arena->current - &arena->buffer[0]);
    if (size > remain)
        return NULL;

    data = arena->current;
    arena->current += size;
    return data;
}
Limitations of version 0

- Problem: instead of returning NULL, it would be better if the arena can fix more memory.

- Solution: let the arena request more memory from malloc and use some form of list where the arena structs are put so they can be freed later on.

- Problem: Alignment. Alignment means that eg an int of size four bytes must be located at an address which is a multiple of four.

- Solution: Add a new parameter to alloc which specifies the alignment requirement, and increment the current pointer sufficiently. See also next page.
More about alignment

- We need to adjust the current pointer with respect to alignment.
- Pointer arithmetic only permits the following cases:
  1. pointer + integer
  2. pointer - integer
  3. integer + pointer
  4. pointer - pointer
- Recall, for add and subtract, the integer is multiplied by the size of what the pointer points to.
- For pointer difference, the difference is divided by that size.
- We need the following to increment current to be a multiple of align:

```c
// eg align = 8, current = 1, skip = 7
skip = align - current % align; // but invalid type for %
current = current + skip;
```
New in C99: uintptr_t

- Traditionally, the solution has been the non-portable
  ```c
  int a = (int)arena->current;
  ```
- This does not work if a pointer is bigger than an int.
- We can use `long long` instead but that is slower than `int` and `int` might have worked...
- C99 introduced the typedef name `uintptr_t` which is an unsigned integer big enough to hold any void pointer (and hence any data pointer).
- The following works:

```c
#include <stdint.h>

void* a;
void* b = a;
uintptr_t c = (uintptr_t)a;
a = (void*)c;
a == b; // certain to be true.
```
If we know that `align` is a power of two, we can instead of:

```c
// eg align = 8, current = 1, skip = 7
skip = align - current % align; // but invalid type for %
current = current + skip;
```

If we know that `align` is a power of two, we can instead of:

```c
current = current + align - 1; // add max remainder
current = current & ~(align - 1); // remove remainder
```
#include <stdint.h>

void* alloc(arena_t* arena, size_t size, size_t align)
{
    int a;
    uintptr_t p = (uintptr_t)arena->current;
    void* data;

    p = arena->current;
    p += align - 1;
    p &= ~(align - 1);
    arena->current = (char*)p;
    data = arena->current;
    arena->current += size;
    return data;
}