Contents of Lecture 3

- Repetition of matrices

  ```c
  double a[3][4];
  double* b;
  double** c;
  ```

- Terminology
- Linkage
- Types
- Conversions
Suppose we declare `double a[3][4]` as a global matrix.
The compiler decides in which address the matrix will start.
Since the matrix is global the address is a known number.

<table>
<thead>
<tr>
<th>address</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+0</td>
<td>a[0][0]</td>
</tr>
<tr>
<td>a+8</td>
<td>a[0][1]</td>
</tr>
<tr>
<td>a+16</td>
<td>a[0][2]</td>
</tr>
<tr>
<td>a+24</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>a+32</td>
<td>a[1][0]</td>
</tr>
<tr>
<td>a+40</td>
<td>a[1][1]</td>
</tr>
<tr>
<td>a+48</td>
<td>a[1][2]</td>
</tr>
<tr>
<td>a+56</td>
<td>a[1][3]</td>
</tr>
<tr>
<td>a+64</td>
<td>a[2][0]</td>
</tr>
<tr>
<td>a+72</td>
<td>a[2][1]</td>
</tr>
<tr>
<td>a+80</td>
<td>a[2][2]</td>
</tr>
<tr>
<td>a+88</td>
<td>a[2][3]</td>
</tr>
</tbody>
</table>
We will now see what happens when we access the matrix.

```c
double a[3][4];
double* p;
double x;
```

To do $x = a[i][j]$ the compiler will produce code for:

```c
// Find address of a[i][j]:
p = a + (i * 4 + j) * sizeof(double);

// Read from memory:
x = *p;
```
Creating a matrix with one call to calloc

double* b;  // 3 x 4 matrix
double* p;
double x;

b = calloc(3 * 4, sizeof(double));
x = b[i * 4 + j];

- Code for reading element (i,j):

  // Find address of b[i * 4 + j] :
  p = b + (i * 4 + j) * sizeof(double);

  // Read from memory:
  x = *p;

- The differences are:
  - The value of b comes from calloc.
  - We must do i \times 4 ourselves.
Creating a matrix with $m + 1$ calls to calloc

double** c;

c = calloc(3, sizeof(double));
for (i = 0; i < 3; ++i)
    c[i] = calloc(4, sizeof(double));
x = c[i][j];
Accessing the matrix

To do \( x = c[i][j] \) the compiler will produce code for:

```c
double** p;
double* q;
double* r;
double x;

// Find address of \( c[i] \);
p = c + i * sizeof(double*); // sizeof a pointer

// Read address of row \( i \):
q = *p;

// Find address of \( q[j] \), i.e. \( c[i][j] \):
r = q + j * sizeof(double); // sizeof a double

// Read from memory:
x = *r;
```
### Comments on `free`

```c
int* a;
int* b;
a = malloc(sizeof(int));
b = a;
free(a);
*a = 12;  // wrong.
a;  // wrong.
b;  // wrong.
```

- After you have freed an object, any mention of that object is wrong, and the behavior is undefined. Anything is permitted to happen according to the C standard.
#include <stddef.h>

size_t length(list_t* head)
{
    size_t count;
    list_t* p;

    if (head == NULL)
        return 0;
    p = head;
    do {
        count += 1;
        p = p->succ;
    } while (p != head);
    return count;
}
A comment on whether to use list_t or not

- We set the data pointer to an object which we want to put in a list.
- Now, why don’t we just add the succ and pred variables to the object type directly?
- That would avoid the waste of allocating memory for the void* data.
- The answer is that if there is one obvious list that our objects should be put in, then that is a good idea, since it avoids calling malloc/free when dealing with lists.
- The list_t should be used when eg an object should be put in two lists at the same time.
- Another aspect is that it may be more convenient to waste that small amount of memory and just use the list_t type. Make your own priorities.

Jonas Skeppstedt (js@cs.lth.se)
Strings in C

- Strings are adjacent characters terminated with a 0.
- ’’C is fun’’ is a string and consists of 9 bytes.
- Eg char v[10] can hold a string.
- Eg char* s can point to a string — but it is no string.
- If we also do s = malloc(10); it is still no string.
- However, s points to memory which can hold a string.
- If we now do s = ’’C is fun’’; — what will happen?
- The memory allocated by malloc is lost.
To make a copy of a string, we can use the following function.

```c
char* copy_string(char* s)
{
    int length;
    char* t;

    length = strlen(s);
    t = malloc(length + 1); // why + 1 ???
    strcpy(t, s);
    return t;
}
```
The type `size_t` is an unsigned integer of some suitable size, and `const` means this function promises not to modify what `s` points to.

```c
size_t strlen(const char* s) {
    size_t length = 0;
    while (*s != 0) {       // have we reached the zero?
        length += 1;        // one more char found.
        s += 1;             // step to the next character.
    }
    return length;
}
```
size_t strlen(const char* s)
{
    char* s0 = s;
    while (*s != 0)
    {
        s += 1;
    }
    return s - s0;  // length is difference in addresses
An implementation refers to a C compiler and a C Standard Library.

Implementation defined behavior is behavior not specified by the ISO C Standard, but rather by the implementation.

- Examples include the range of the different types.
- Whether signed shift is logical or arithmetic:

  ```c
  int   a;
  int   b;
  
  b = a >> 2;  // arithmetic in Java
  b = a >>> 2; // logical in Java — invalid C!
  ```

- The shifts above remove the two least significant bits.
- Logical shift copies in zeroes at the most significant position.
- Arithmetic shift copies in the value of the most significant bit.
- For a positive a we have $a >> 3 = a / 8$, due to $2^3 = 8$.
- For a negative a — see pages 114 — 116.
Unspecified and undefined behavior

- The implementation must document how it deals with implementation defined behavior.
- For **unspecified behavior** the implementation is free to do as it wishes among a set of reasonable alternatives.
  - For instance:
    - the order of evaluation of parameters to a function
    - the order of evaluation of operands of binary arithmetic operators (but not && and ||).
- For **undefined behavior** anything may happen.
  - For instance:
    - Reading or writing through a null pointer.
    - Overflow of signed integers.
    - Divide by zero.
Character sets

- C99 uses implementation defined multibyte characters sequences and wide characters.
- C11 uses UTF-8 multibyte character sequences and wide characters with support for Unicode.
- A wide character is 16 or 32 bits wide and makes all characters this size which usually is a waste of space but makes processing easier.
- UTF-8 was invented by the person who invented UNIX and is an extension to ASCII — Ken Thompson at Bell Labs.
- A non-ASCII Unicode character such as Ö can be encoded by a few bytes in UTF-8.
- When using multibyte characters instead of wide characters only the non-ASCII characters need multiple bytes.
Scopes of identifiers

- **File scope**

- **Function scope** — only labels have function scope so label names must be unique in a function.

  ```c
  void f(void)
  {
    L:    goto L;
  }
  ```

- **Function prototype scope** — don’t declare new types in a prototype since the type will be useless elsewhere:

  ```c
  void f(struct s { int a; } s);
  ```

- **Block scope**
An identifier can have external, internal or no linkage.

Linkage should not be confused with storage duration but is somewhat related.

With linkage is meant that an identifier can be mentioned multiple times while referring to the same function or variable.

A variable or function at file scope declared with `static` has internal linkage.

Internal linkage is very useful to avoid conflicts between different files.

You may want a function `initialize` in several C files. Declare them with `static`:

```c
static void initialize(void)
{
    /* ... */
}
```

Without `static` there can only be one identifier `initialize`. 
Identifiers with linkage can be declared multiple times

- Declaring the same identifier multiple times in the same file results only in one variable:

  ```
  int a; // external linkage.
  int a; // external linkage.
  extern int b; // external linkage.
  extern int b; // external linkage.
  static int c; // internal linkage.
  static int c; // internal linkage.
  ```

- An identifier with external linkage must be unique in the entire program.
- An identifier with internal linkage must be unique in the file.
Identifiers with no linkage

- At block scope only identifiers declared with `extern` have linkage.
- We get a compilation error if we redeclare an identifier at the same scope and without linkage:

```c
int a;

int main(void)
{
    int b;
    int b; // invalid.
    static int c; // no linkage.
    static int c; // no linkage and invalid.
    extern int a; // external linkage.

    return 0;
}
```

- The `a` in `main` refers to the global variable.
Storage duration of objects

- By **storage duration** is meant the lifetime of an object.
- An ”object” is some data such as a variable or an object allocated with `malloc`.

- Local variables on the stack have **automatic storage duration** and disappear when the function returns.
- All variables with linkage have **static storage duration** and exist from the beginning to the end of the program’s execution.
- In C11 there is also **thread storage duration** for variables declared with `_Thread_local` (and possibly static or extern) and exist from the beginning to the end of the thread’s execution.
- Data allocated from the heap has **dynamic storage duration** and exists until it is freed.
- C11 also introduced **temporary lifetime** which says that a returned value must exist for the duration of the full expression in which it was created — for details see the book.
There are three main kinds of types:

- Function types
- Object types
- Incomplete object types

```c
typedef struct list_t list_t;
extern int a[];
```

- `void` is an incomplete object type which can never be completed.
- Except for `void`, incomplete object types can later be completed.
Object types

- The object types are divided into:
  - Scalar types — pointers and arithmetic types (i.e. numbers)
  - Aggregate types — arrays and structs
Compatible types

- The basic rules is that two types are compatible if they are the same type.
- We can only assign pointers to compatible types and the null pointer.
- In C++ we can write 0 for the null pointer but in C we must use NULL (since there are machine which use a different value for the null pointer).

```c
char* cp;
signed char* sp;
unsigned char* up;
unsigned int* p;

p = NULL;    // valid
p = up;      // invalid
cp = sp;     // invalid
cp = up;     // invalid
```

- What about structs?
Compatibility of different structs

- Recall a translation unit is a C file being compiled with all its included files.
- C uses **name equivalence** for structs declared in the same translation unit.
- The two structs below are not compatible types.

  ```c
  struct s { int a, b; } *p, *q;
  struct t { int a, b; } *r;
  
  p = q;  // valid: pointers to the same type.
  r = p;  // invalid: pointers to different types.
  ```

- C uses **structural equivalence** for structs declared in different translation units, for which two different declarations of the same struct are compatible despite being in different translation units.
A **cast** is an explicit type conversion, which sometimes are needed. For instance, if we want to copy the value of a pointer to an integer, we use the type `intptr_t` and convert the pointer using a cast:

```c
#include <stdint.h>

intptr_t a;
int b;
void* p;

a = (intptr_t)p; // converts a pointer to an integer type.
b = (int)p;     // wrong type: may not work.
```

- This is one of the rare cases we should use a cast.
- We will see later why we might want to do this.
- When we use casts with pointers we are playing with dangerous tools.
Type aliasing and the ANSI C Aliasing Rules

```
struct s { int a, b; } *p, *q;
struct t { int a, b; } *r;
int x;

r->a = 1;
p = (struct s*)r;
p->a = 2;
x = r->a;  // x may become 1.
```

- To help compilers optimize C code, they are allowed to assume that pointers to incompatible types cannot point to the same data.
- Exceptions to this rule are if one of the types is a pointer to `void`, pointer to a `char` type, or the same integer type but with different sign.
- The last is called corresponding integer types.
Integer types which only differ in sign are called corresponding integer types:

```c
signed int* p;
unsigned int* q;
int x;

*q = 1;
p = (signed int*)q;
*p = 2;
x = *q; // x must become 2.
```
Integer promotions

- For integer types smaller than an int, operands of arithmetic operators and function call arguments are converted to an int or unsigned int (if an int cannot hold all values of the original type).
- Consider:

  ```c
  unsigned char a = 1;
  unsigned char b = 2;
  unsigned char c;
  
  c = a + b;
  ```

- Each of `a` and `b` is converted to an int before adding and then the result is converted to an unsigned char in the assignment.
- This means that:

  ```c
  sizeof a < sizeof +a
  sizeof(unsigned char) < sizeof(int)
  ```

- We will see some effects of this next...
Hex numbers and the bitwise complement operator

- A number on base 16 is called a hex number and is written with 0x as a prefix:
  
  ```c
  unsigned char a = 0xf0;
  ```

- This stores 11110000 in `a` (assuming it is 8 bits wide).

- The ~ operator changes each 1 to 0 and 0 to 1 (called bitwise complement) — of the promoted operand:

  ```c
  unsigned char a = 0xf0;

  printf("%x\n", ~a);
  ```

- Assuming a 32-bit `int`, how many `f` does it print?
First 11110000 is promoted to become:
0000000000000000000000011110000 i.e. 24 leading zeroes.

Bitwise complement results in 11111111111111111111100001111

Printing this as a hex number results in ffffffff0f. i.e. seven f.
Another quiz

- Assume 255 is the largest value of an unsigned char.
- What does the following print?

```c
unsigned char a;
int b;

a = 255;
b = a + 1;

printf("%d\n", b);
```
The a is promoted to an int and 1 is added to 255 so the output becomes 256.

However, if we then do:

```c
a = b;
printf("%u\n", a);
```

The output will be 0.

Assume an unsigned integer type is $n$ bits wide.

In the conversion at the assignment, the value stored in an unsigned integer always is $x \mod 2^n$, where $x$ is the value of the expression.

For signed integers, if the value cannot be represented it is implementation defined what happens but usually as many bits of the expression that fits are stored.

There is no overflow at a conversion — they can occur in expressions and will be discussed in Lecture 5.