What is meant by ”declarations” ???

- A declaration associates an identifier with one of
  - variable,
  - function, or
  - type.
- For example:

```
int*     a;                          int   *a, *b, *c;
int*     b;                          typedef int integer;
int     *c;
typedef int integer;
```

- One declaration can have multiple variables and functions.

Beware of declarators

same

```
int*    a;   | int*     a, b, c; // only a is a pointer!
int*    b;   |
int    *c;   |

int     a, f(int, int), v[10]; // valid C but uncommon
```

Names spaces

- Variables, types (typedef names) and functions belong to the same name space.
- Struct tags are in a separate namespace.

```
typedef int a;
a           a;        // error: redeclaration of a
struct x { int y; } z; // tag x, field y, variable z
struct s { int s; } s; // OK.
```

går ej!

- Labels are in a separate name space as well:

```
void f()
{
        goto f;                 // not a function call!
f:      printf("hello\n");      // target of the goto.
}
```

Declaring vs defining global variables

- **Declaring** a global variable means to inform the compiler that a certain variable exists with a type and a name.
- It does not mean asking the compiler to reserve memory for it.
- **Defining** a global variable means declaring the variable and telling the compiler to reserve memory for it.
- If we want to define a global variable and thus making sure its address is "here", we have to give it an initial value:

```
int a = 12;               // correct
extern int b = 12;        // not portable
int c;                    // problematic since 2020!
```

New implementation defined behavior in gcc and clang

- If we just want to declare a global variable but **not** reserving memory for it, we use `extern`:

```
extern int a;
```

- If we want to declare a global variable and don't mind reserving memory for it, we **could** skip `extern`:

```
int a;
```

- This is no longer portable with gcc and clang: but use `gcc -fcommon`
- Instead seeing this, they *define* a (reserve memory for it)
  - Since gcc 10.1 released in May 2020
  - With clang 11.0 released in October 2020
- Skipping `extern` had the effect that if no other file in the program the variable has been defined with an initial value, then the compiler will reserve memory for it.

Wrong declaration 1

```
file a.c
---------

extern int x;

int main()
{
        printf("x = %d\n", x);
        return 0;
}
```

```
file b.c
---------

extern int x;
```

- The compiler will complain that `x` is not defined (anywhere).

Wrong declaration 2

```
file a.c
---------

int x = 1;

int main()
{
}
```

```
file b.c
---------

int x = 1;
```

- The compiler will complain that x is defined multiple times.

Wrong declaration 3 (but worked 1972-2019!)

```
file a.c                             file b.c
---------                            ---------


int x;                               int x;


int main()
{
}
```

Doesn't work anymore

- Before recent changes in implementation defined behavior to gcc and clang
  - The compiler (actually, the linker) would reserve memory for `x` somewhere but not in memory corresponding to these files.
  - Global variables without `extern` were put in the ELF section COMMON.
- With recent changes to gcc and clang
  - The compiler puts the variables in the ELF section BSS
  - BSS means block started by symbol and the name is complete nonsense and a historical rest

Correct declaration 1

```
file a.c
---------

int x = 1;

int main()
{
        printf("x = %d\n", x);
        return 0;
}
```

```
file b.c
---------

extern int x;
```

- The compiler will reserve memory for `x` in file ’’`a.c`’’.

Correct declaration 2

```
file a.c
---------

int x;

int main()
{
        printf("x = %d\n", x);
        return 0;
}
```

```
file b.c
---------

extern int x;
```

- The compiler will reserve memory for x somewhere but not in these files.

Correct declaration 3

```
file a.c                 file b.c                 file globals.h
--------                 --------                 --------------

#include "globals.h"     #include "globals.h"     extern int x;

int x;

int main()
{
        // use x         // use x
}
```

- Declare all global variables of an application in a file ''`globals.h`'' and include it when needed.
- The same applies to abstract data types such as lists. Put the type and function declarations in a header file.
- Sometimes only the typedef and function declarations are put in the public header file and the struct declarations in a private header file. More about that later.

Variable length array: VLA

- A local array (with automatic storage duration, i.e. allocated on the stack) can have a non-constant size:

```
void f(size_t n)
{
        int     a[n];

        /* ... */
}
```

- When the execution comes to the declaration it evaluates and remembers the size of `n` and the compiler must allocate memory for the array.
- This memory is allocated on the stack simply by changing the stack pointer.
- Thus the programmer does not have to (and cannot) deallocate that memory.
- The memory is automatically deallocated when the function returns.

Local array only

- Since the memory for a VLA is allocated from the stack, only local arrays can have a non-constant size.
- VLA's are different from flexible array members.
- A VLA must be an ordinary identifier and not for instance a struct member.
- VLA's were introduced in C99 and are very useful.

Restrictions of VLA's

- Unfortunately:
  - VLA's has become optional in C11, but since `gcc` and `clang` supports them most compilers will also.
  - It's impossible to know whether the allocation succeeded or not.
- Use with care and **never** for a size supplied as program input — otherwise a security risk.
- Commercial compilers use it (or a similar approach called `alloca`) to improve speed.
- For instance allocating an array of object pointers can make iteration through a data structure simpler and faster — forget it in a nuclear power plant though.

Automatic storage duration (allocated on stack) makes faster allocation and deallocation than e.g., malloc, and good cache locality, dynamic sizing

May be limited to stack size, and can lead to overflow.

VLA's and normal arrays

- A VLA is organized in memory the same way as other arrays are.
- The only difference really is that the compiler must produce code to remember the array sizes.

```
void f(int a[3][4]);

void g(size_t m, size_t n)
{
        int     b[m][n];

        f(b); // OK if m == 3 and n == 4.
}
```

VLA parameters

- A VLA can be a parameter:

```
void f(size_t m, size_t n, int a[m][n])
{
        /* ... */
}

int b[10000][20000];
```

Allocated on the static memory and not on the Stack

```
void g(void)
{
        f(10000, 20000, b); // OK.
}
```

- This is not dangerous in any way since the matrix was not allocated on the stack.

It may be dangerous to send in a VLA allocated on the stack as a function parameter

A function prototype with VLA

- Consider `f` again:

```
void f(size_t m, size_t n, int a[m][n])
{
        /* ... */
}
```

- How can we declare `f` in a header file?
- We can use:

```
void f(size_t m, size_t n, int a[m][n]);
```

- But if we don't want to write `m` and `n`?
- Recall: we can write prototypes without parameter names:

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

- Therefore there is new syntax and we can write:

```
void f(size_t, size_t, int [*][*]);
```

- Can we remove one or both of the stars?

Recall array parameters become pointer parameters

- An array parameter becomes a pointer parameter and we can therefore skip the size:

```
void f(size_t, size_t, int [][*]);
```

- Or:

```
void f(size_t, size_t, int (*)[*]);
```

Since array parameter becomes a pointer parameter in a function argument

- But that does not win a prize for beautiful C code.

Variably modified types

- A type with a VLA is called a variably modified type.
- We can declare a pointer to a VLA:

```
void f(size_t n)
{
        int     (*p)[n];

        /* ... */
}
```

Precedence and associativity

- All operators are ordered according to their **precedence**.
- For instance $*$ has higher precedence than $+$.
- The **associativity** specifies in which order multiple operators with the same precedence should be evaluated.
- The binary operators are left-associative.
- For instance `a - b - c` is evaluated as `(a - b) - c`.
- The unary, the assignment operators, and the conditional expression are right-associative.
- For instance `a = b = c` means `a = (b = c)`.
- `a += b += c` means `a += (b += c)`.
- If initially `a = 1`, `b = 10` and `c = 100`, then the above results in `b = 110` and `a = 111`.
- Thus the value of `b += c` is the new value of `b`.

More examples

- What is the value of:

$$1 \texttt{<<} \underbrace{2 + 3}_{=5}$$

1<<5

=> 35

More examples

- The value on the previous slide is:

```
1 << (2 + 3)
1 << 5
32
```

Evaluation of an expression

- Operands smaller than `int` are converted either to `int` or `unsigned int`.
- The usual arithmetic conversions determine the type of the result of an operation.
- For instance:

```
→ unsigned char    a = 1;
→ unsigned short   b = 2;
→ double           c;
                int

  c = a / b;  = double
  a = c + 1;  = double
```

- First both a and b are converted to `int`.
- The type of the quotient is `int` which is then converted to `double`.
- The type of the sum is `double`.

Assignment

- At an assignment the value is converted to the type of the modified variable.
- In the previous example:

```
unsigned char   a = 1;
unsigned short  b = 2;
double          c;

c = a / b;
```

- What can we do to get `0.5` assigned to `c`?
- What about:

```
c = (double)(a / b);      // No effect!
```

- One of the operands must be converted to `double`!

```
c = (double)a / b;        // OK.  c = 0.5
```

Assignment conversions

- Recall that a pointer to void can point to any **data** object (but not to functions).

```
char*          cp;
signed char*   scp;
int*           ip;
long*          lp;
void*          vp;

vp = ip;   // OK. void pointer always ok.
ip = vp;   // OK. void pointer always ok.
ip = lp;   // Always wrong to mix non-void-pointer types.
cp = scp;  // No: char and signed char are different types.
```

Exceptional conditions

- Note the difference between the following:
  - the value of an operation cannot be represented in the type of the expression
  - the value assigned to a variable cannot be represented in the type of the variable
- In the former case we have an overflow which can result in undefined behavior and a crash.
- In the latter case the implementation must document what happens — for integers usually as many bits that fit are stored.

```
unsigned char    a;
signed char      b;
float            c;
a = 0xfff; 255
b = 0xfff; -1
c = 1e100; INFINITY
```

A value assigned to a variable cannot be represented in the variable type, instead the value is usually the amount of bits that can be fit.

- What are the values of a, b, and c?

Values

- `a = 255`
- `b = -1`
- `c = INFINITY`
- The macro `INFINITY` is defined in `<math.h>`

Another quiz

- What is the value of the following:

```
unsigned char   uc = 255;

uc + 1
```

$\Rightarrow uc = 256$

Since uc is promoted to int drought addition, the number becomes 256 and not 0.

Overflow

- The value in the previous slide is 256 since uc is integer promoted to the type int and then two operands of type int are added.
- What about:

```
#include <limits.h>

int a = INT_MAX;

a + 1;
```

= overflow

The result

- In the previous slide, the type of the result is `int` but the sum cannot be represented in that type.
- For signed integers, an overflow triggers undefined behavior.
- For unsigned integers, an overflow ”wraps around”, i.e. all unsigned arithmetic is performed modulo one greater than the maximum value of the type:

```
unsigned int    a = UINT_MAX;

a + 1; // zero
```

- For floating point on most machines the result becomes `INFINITY`.

Struct parameter

- Given:

```
typedef struct {
        double  x;
        double  y;
} point_t;
```

New type definition

```
void print(point_t p);
```

- Assume we have calculated x and y and want to print them as a point:

```
point_t         tmp;

tmp.x = x;
tmp.y = y;
```

Doesn’t work, since cast does not work for aggregate types

```
print(tmp);
```

We cannot use a cast instead

- We cannot make a cast for an aggregate type.
- Aggregate types are structs/unions and arrays.
- What should we do?
- As above or using the C99 compound literal.
- Compound literals were first used in Ken Thompson's C compiler for the Plan9 operating system — recall Ken Thompson invented UNIX (and UTF-8 and many other things).

Compound literals

- Compound literals look like casts (explicit conversions) but they are different.
- One purpose of compound literals is to make it possible to create constants for structs:

```
(point_t) { 1.23, 4.56 };
```

Directly create point_t structs and create constants

- We can pass it to the print function:

```
int main(void)
{
        print((point_t) { 1.23, 4.56 });
}
```

- So what is a compound literal really and how it is implemented in C compilers?

Details of compound literals

- A compound literal is simply an anonymous variable initialized using special syntax.
- Since it's a normal object, we can take its address:

```
void print(point_t*);

int main(void)
{
        print(&(point_t) { 1.23, 4.56 });
}
```

- We can also use designated initializers:

```
        print(&(point_t) { .x = 1.23, .y = 4.56 });
```

Compound literals for other types

- We can use compound literals for other types as well:

```
int*    p = (int[]){ 1, 2, 3 };
int     a = (int){ 1 };
```

- There is no purpose to use compound literals for scalar types, however.

NAN and relational and equality expressions

- Recall: NAN stands for not-a-number and is the value of expressions which are not mathematically defined such as:

$$\boxed{0/0 \quad \infty/\infty \quad \infty - \infty}$$

- Floating point comparisons with NAN are always false.
- Thus we should not change comparisons such as

```
if (a < b)
        printf("case 1\n");
else
        printf("case 2\n");
```

- into:

```
if (a >= b)
        printf("case 2\n");
else
        printf("case 1\n");
```

Avoid!! Will always result in false

Pointers and relational and equality expressions

- The relational expressions are: `< <= > >=`
- The equality expressions are: `== !=`
- Pointers can be compared in relational expressions only if they point to the same array object.
- For relational expressions, scalar variables are treated as arrays with one element.
- The compiler must ensure that the first byte after the array is a valid address.
- Any valid pointers to compatible types can be compared in equality expressions.

Valid optimization of array references

```
double  a[N];

for (i = 0; i < N; ++i)
        x += a[i];
```

Optimised

```
double*      p = a;
double*      end = &a[N];
```

Loop with pointers instead

```
while (p < end)
        x += *p++;
```

- Don't do this by hand, instead use the command: `cc -O2`
- Do this only if you are not allowed to use compiler optimizations.
- In the course Optimizing Compilers it is taught how this and other optimizations are implemented.

Invalid optimization of array references

```
double  a[N];

for (i = N-1; i >= 0; --i)
        x += a[i];
```

```
double*   p = &a[N];

while (--p >= a)
       x += *p;
```

!

Will loop until p == a[-1], which triggers undefined behavior

- In the last iteration `p == a[-1]` in the comparison.
- The compiler is not required to make that address valid.
- The code to the right triggers undefined behavior.

Modifications of variables

- A **sequence point**, for example a semicolon, is used in C to determine when side effects have been performed.
- The most important side effect is the modification of a variable.
- A variable may only be modified once between two sequence points.
- The following are invalid:

```
a = a = 1;
b = ++b;
++c * c--;
```

- In addition, a variable may not be read after a modification before the next sequence point. Therefore also wrong:

```
b = (a = 1) + (a * 2);
2          1
```

error if (a = 1) is evaluated first

- The code is invalid if the left operand of the add is evaluated first — which it may be since the evaluation order is unspecified.

Comma expression

- A comma expression can be used when multiple variables should be initialized in a for loop:

```
for (i = 0, p = list; p != NULL; p = p->next)
        /* ... */
```

- In a comma expression first the left operand is evaluated, and then the right operand.
- There is a sequence point between the evaluations of the operands.
- The value of a comma expression is the value of the right operand.
- To use a comma expression in an argument list, it must be enclosed in parentheses:

```
printf("%d\n", (1, 2)); // prints 2
```

”The value of comma expression is the value of the right operand”

Alignment of pointers

- Recall: if a type has an alignment *b* it means objects of that type should have an address that is a multiple of *b*.
- The operator `_Alignof` takes a type name and gives the alignment of that type represented as `size_t`.
- Including `<stdalign.h>` we can write `alignof` instead:

```
printf("%zu\n", alignof(double));
```

- Suppose now we allocate 20 bytes and wish to store an object of type `double` there:

```
char            data[20];
double*         p;

p = (double*)data;
*p = x;                    // No — probably not aligned!
```

Not aligned

- First warning: this violates ANSI C aliasing rule but is sometimes used.

Aligning a pointer

- We need to add a number to p so that its value becomes a multiple of 8.
- One attempt is:

```
unsigned        a = (unsigned)p; // wrong type.
unsigned        r = a % 8;

if (r != 0)
        a += 8 - r;
p = (double*)a; // might not work.
```

- If $p = 15$ then $r = 7$ and we add 1 to a.
- An alternative is to calculate: $(p + 7)/8 * 8$ — then we don't need to branch.
  $(15 + 7)/8 * 8 = 22/8 * 8 = 2 * 8 = 16.$
- Remainder and division are expensive however.

Using bitwise operators

- We can write $p + 7$ as $x * 8 + y$, where $0 \leq y \leq 7$.
- The purpose of dividing and multiplying is to get rid of y.
- How can we do that faster than using division and multiplication?
- Bitwise operators are useful for this.
- Dividing and multiplying by 8 is equivalent to clearing the bits which contribute to y.
- $p + 7 = 15 + 7 = 22 = 16 + 4 + 2 = 10110_2$
- The value of the bitwise complement operator is the operand with every bit inverted.
- That is, we should do a bitwise and with the bitwise complement of 111_2, in C: ~7:

```
unsigned          a = (unsigned)p; // still wrong type.

a = (a + 7) & ~7;
p = (double*)a;
```

The uintptr_t

- Since a pointer may be 64 bits and an unsigned int only 16 bits the above code is wrong.
- We should use the type uintptr_t defined in the header file <stdint.h>.

Integer divide and remainder

- The value of $-5/3$ is -1 but that was not always certain!
- In ANSI C (i.e. before C99) the rounding mode was implementation defined.
- Since C99, ISO C follows FORTRAN which rounds towards zero (i.e. truncation).
- The % operator computes the remainder of a/b as $r = a - a/b * b$
- What is printed by the following program?

```
int a = 5;
int b = -3;
int q = a / b;  ≈-1
int r = a - q * b; = 2
assert(a % b == r);
printf("q = %d, r = %d\n", q, r);
```

Integer divide and remainder

Output from the previous program is: `q = -1, r = 2`

- What about the following program?

```
int a = -5;
int b = 3;
int q = a / b; ≈-1
int r = a - q * b; -2
assert(a % b == r);
printf("q = %d, r = %d\n", q, r);
```

Signed versus unsigned integer arithmetic

Output from the previous program is: `q = -1, r = -2`

- In general, the value of `a % b` has the sign of `a` — also if both are negative.
- For unsigned integers, $r \geq 0$.

```
unsigned int a = 5;
unsigned int b = 3;
unsigned int q = a / b;
unsigned int r = a - q * b;
assert(a % b == r);
printf("q = %d, r = %d\n", q, r);
```

Arithmetic with both signed and unsigned integers

- Suppose we have `a + b` and the operands have different types.
- Then the **usual arithmetic conversions** apply.
- Each arithmentic type has a rank and `long double` has highest rank among real types (as opposed to imaginary or complex types) down to `char` and `_Bool`.
- Two types are **corresponding** if they only differ in signed versus unsigned.
- If one operand has e.g. type `signed int` and the other `unsigned int` then the signed operand is converted to the corresponding unsigned type, i.e. `unsigned int`.
- Recall: all unsigned arithmetic is performed modulo 2^N where N is the number of bits in the type

Examples

- $-3+5=2$ — both have type `signed int`
- $-3+5U = (2^{32}-3)+5U = 2u$ — if `UINT_MAX = ` $2^{32}-1$.
- $-3/5U =?$
- $-3\%5U =?$

Examples

- $-3/5U = 858993458$
- $-3\%5U = 3$
- $3U/-5$ $=?$
- $3U\%-5$ $=?$

Examples

- $3U/-5=0$
- $3U\%-5=3$
- What is printed by the program below?

```
if (-5 > 3U)
        puts("PASS");
else
        puts("FAIL");
```

More about usual arithmetic conversions

- Recall: if the operands only differ in signed/unsigned then the signed is converted to the corresponding unsigned type.
- I should say: the rank of the type of the operand with signed type...
- But ”simplify” it as: the rank of the signed type...
- Now: if the rank of the signed type is higher than the rank of the unsigned type and the signed type can represent all values of the unsigned type, then the operand with unsigned type is converted to the type of the signed type:

```
if (-5LL > 3U)
        puts("FAIL");
else
        puts("PASS");
```

- Since the type `signed long long` can represent all values of type `unsigned int` the conversion is to `signed long long`.

Another example with usual arithmetic conversions

- Finally: if the rank of the signed type is higher than the rank of the unsigned type but the signed type cannot represent all values of the unsigned type, then the operand with signed type is converted to its corresponding type:

```
// assume sizeof(signed long) == sizeof(unsigned int)
if (-5L > 3U)
        puts("PASS");
else
        puts("FAIL");
```

- If the type `signed long` cannot represent all values of type `unsigned int` the `-5L` is converted to `unsigned long` (and becomes a big number...).
- Warning: the C standard does not specify the output of the last two examples since their behavior depends on the sizes of the integer types!
- Summary: avoid comparing signed and unsigned types — especially not corresponding types.