Contents of Lecture 5: Expressions

- Precedence and associativity
- Generic selections
- Arrays vs pointers
- Compound literals
- Alignment
- Divide and remainder
- Relational and equality expressions

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 << 2 + 3
```

More examples

- The value on the previous slide is:

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

Evaluation of an expression

- Recall that operands smaller than `int` are converted either to `int` or `unsigned int`.
- Also recall from that 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;

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

- 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
```

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 we have an 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;
b = 0xfff;
c = 1e100;
```

- 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
```

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;
```

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;

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;

print(tmp);
```

We cannot use a cast instead

- For scalar parameters, we can either pass the scalar value, or if there is no type for the parameter (e.g. as for `printf`).
- 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 };
```

- 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

- Floating point comparisons with NaN are always false.
- Recall: NaN stands for not-a-number and is the value of expressions which are not mathematically defined such as:

$$0/0 \quad \infty/\infty \quad \infty - \infty$$

- Thus we should not change comparisons such as

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

- into:

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

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];
```

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

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 EDA230 Optimizing Compilers it is taught how this and other optimizations are implemented.

Invalid optimization of array references

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

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

- 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);
```

- 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

jitem 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
```

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!
```

- What can we do?

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>`.
- Sometimes, such as when allocating memory buffers for the Cell processor we need to align pointers like we did above.
- If we use a compiler with support for VLA's such as `gcc` we can allocate memory for a matrix as in Example 1.4.13 — as we saw in Lecture 4.
- If we use a compiler without VLA support we can do as in Example 1.4.12.

Example 1.4.12

- The goals with Example 1.4.12 are:
  - to allocate memory with only one call to `malloc`
  - to be able to use matrix syntax: `a[i][j]`
  - to be portable to an ANSI C compiler or C11 compiler without VLA support
- The function `alloc` allocates memory for a matrix with an element size specified by `block` which must be a power of 2.
- We will not go into the details — it is essentially exactly what we just saw with a value of `block` being 8.

The C preprocessor

- Predefined macros
- Macro replacement
- Conditional inclusion
- Source file inclusion
- Line control
- Error directive
- Pragma directive
- Null directive
- Predefined macro names
- Pragma operator

Predefined macros: useful standard macros

- **__FILE__** expands to the source file name.
- **__LINE__** expands to the current line number.
- **__DATE__** expands to the date of translation.
- **__TIME__** expands to the time of translation.
- **__STDC__** expands to 1 if the implementation is conforming.
- **__STDC_HOSTED__** expands to 1 if the implementation is hosted, and to 0 if it is free-standing.
- **__STDC_VERSION__** expands to **199901L**.

Predefined macros: implementation-defined

- **__STDC_IEC_559__** expands to 1 if IEC 60559/IEEE 754 is supported (except complex arithmetic).
- **__STDC_IEC_559_COMPLEX__** expands to 1 if complex arithmetic in IEC 60559/IEEE 754 is supported.
- **__STDC_ISO_10646__** expands to an integer **yyyymmL** to indicate which values of **wchar_t** are supported.
- If a predefined macro is undefined then behavior is undefined.

Defining macros

```
#define obj (a)    a+1
#define bad(a)     a+1      // Wrong.
#define good(a)    (a+1)    // Use parentheses.

obj(3)          =>    (a) a+1(3)
bad(3)*10       =>    3+1*10
good(3)*10      =>    (3+1)*10
(good)(3)*10    =>    (good)(3)*10
```

- No whitespace between macro name and left parenthesis in function-like macro.
- A function-like macro not followed by left parenthesis is not expanded.

Conditional inclusion

```
#define DEBUG

#ifdef DEBUG
        printf("here we go: %s %d\n", __FILE__, __LINE__);
#endif

#ifndef DEBUG
#endif

#if expr1
#elif expr2
#elif expr3
#else
#endif
```

More directives

```
#define DEBUG 12
#define DEBUG 13   // invalid: cannot redefine a macro
#undef DEBUG
#define DEBUG 13   // OK. undefined first

#line 9999 "a.c"   // will set __LINE__ and __FILE__

#ifndef __STDC__
#error this will not with a pre-ANSI compiler!
#endif

#pragma directive from user to compiler
_Pragma("directive from user to compiler")
```

operator "stringizer"

- Operator # must precede a macro parameter and it expands to a string.

```
#define xstr(a) #a
#define str(b)  xstr(b)
#define c       12

xstr(c)        => "c"
str(c)         => "12"

#define fatal(expr) {                                  \
        fprintf(stderr, "%s line %d in \"%s\": "       \
        "fatal error %s = %d\n", __FILE__,             \
        __LINE__, __func__, #expr, expr); exit(1); }
int x = 2;
fatal(x); => a.c line 15 in "main": fatal error x = 2
```

operator

- Operator ## concatenates the tokens to the left and right.

```
#define name(id, type) id##type

name(x,int) => xint

#define a   x ## y
#define xy  12
#int b = a;        // initializes b to 12;
```

__VA_ARGS__

- Sometimes it is convenient to a have a variable number of arguments to a function-like macro, eg when using printf.
- Without **__VA_ARGS__**, the number of arguments must match the number of parameters.

Variable number of arguments in macros

```
#ifdef DEBUG
#define pr(...) fprintf(stderr, __VA_ARGS__);
#else
#define pr(...)  /* do nothing. */
#endif

int x = 1, y = 3;

pr("x = %d, y = %d\n", x, y);   => x = 1, y = 3
```

Macros can improve performance

- Since macros are expanded in the called function they eliminate the overhead of calling functions.
- Macros can cause problems however:

```
#define square(a)       a*a

x = 100 / square(10) => 100 / 10 * 10
```

- Use parentheses:

```
#define square(a)       ((a)*(a))

y = square(cos(x))      // valid but slow
z = square(++y)         // wrong
```

- Now the `cos` function is called twice!
- Modifying `y` twice is wrong.

Macros with statements

- Suppose we write want to swap the values of two variables using a macro:

```
#define SWAP(a, b)      tmp = a; a = b; b = tmp;

if (a < b)
        SWAP(a, b);
```

- What happens?
- How about:

```
#define SWAP(a, b)      { int tmp = a; a = b; b = tmp; }

if (a < b)
        SWAP(a, b);
else
        printf("syntax error!\n");
```

- A compound statement cannot be followed by a semicolon.

Using do-while loops

- We can do as follows:

```
#define SWAP(a, b)  do { int tmp = a; a = b; b = tmp; } while (0)
```

- This macro will solve both of the previous problems.