Contents of Lecture 6

Three versions of a program

```
void f(int* a)
{
        *a = 12;
}

int g(void)
{
        int     b;

        f(&b);

        return b + 1;
}

int c;

int main(void)
{
        c = g();
}
```

Using a macro

```
#define F(var, value)   ((var) = (value))

int g(void)
{
        int     b;

        F(b, 12);

        return b + 1;
}

int c;

int main(void)
{
        c = g();
}
```

Using an inline function

```
inline void f(int* a)
{
        *a = 12;
}

int g(void)
{
        int     b;

        f(&b);
        return b + 1;
}

int c;
int (*fp)(void) = g;

struct s {
        int     (*open)(const char* name, int mode);
        int     (*close)(const char* name, int mode);
};
```

Conclusion

- After inspecting the assembler code we can draw some conclusions.
- The program will be faster if small frequently called functions are inlined somehow.
- The `gcc` compiler can handle the inlining for us — so we don't need to use neither a macro nor the `inline` specifier.
- When should we use macros or the `inline` specifier?
- If we use a poor compiler, or if we don't know which compiler a customer will use, then it might make sense to use either macro or `inline`.
- If we share our C code as open source code, however, it's a fairly good assumption that `gcc` will be used to compile it so simply add optimization to the `CFLAGS` variable in `makefile`.

Two courses about writing fast C code

- In EDAF15 Algorithm implementation we study techniques to write faster C code:
  - How modern processors work and how that can be exploited by C programmers.
  - How to measure and analyze the performance and what happens in the processor that affects the performance.
- This is covered in Chapters 3, 4, 6, and 14-16 in the book.
- In EDAN25 Multicore programming we study multicore hardware and how to write fast multithreaded C programs (and to some extent Java and Scala programs).
- This is covered in Section 7.18 and Chapters 5, 17 and 18 in the book.

So this course is about understanding C

- Therefore we will not discuss in any more detail why inlining might be good.
- Instead, we will now focus on when it is not permitted to use inlining and why.

Linkage and inline functions

- Recall: external linkage means an identifier is accessible from other files.
- A function with internal linkage, i.e. declared with `static` can always be inlined but functions with external linkage have restrictions:
    - An inline function with external linkage may not define modifiable data with static storage duration.
    - An inline function with external linkage may not reference any identifier with internal linkage.
- What do these mean and why do we need these restrictions?

First restriction

```
extern inline int f(void)
{
        static const int a[] = { 1, 2, 3 };     // OK.
        static int x;                           // Invalid.
        int y = 0;                              // OK.

        return ++y * ++x;
}
```

- Restriction: an inline function with external linkage is not allowed to declare modifiable data with static storage duration.
- Since copies of `f` inlined in different files will use different instances of `x` this is forbidden.
- The constant array and modifiable variable with automatic storage duration are OK.

Second restriction

```
static int g(void)
{
        return 1;
}

static int a;

extern inline int f(void)
{
        a = 1;          // Invalid.
        return g();     // Invalid.
}
```

- Restriction: an inline function with external linkage is not allowed to access any identifier with internal linkage.
- When `f` is inlined in some file, it will use the available function g or variable a but then different files can have different functions g, and a.

A warning

- The gcc compiler supported the inline function specifier before it was added to the C standard.
- Unfortunately, gcc uses slightly non-standard semantics for inline.
- A simple rule which works both in ISO C and with gcc is to declare inline functions in header files such as:

```
#ifndef max_h
#define max_h

static inline int max(int a, int b)
{
        return a >= b ? a : b;
}

#endif
```

- Read Section 9.5.1 for details about the incompatibility — I will not ask about it in the exam, however.

Chapter 11: Statements

- Labeled statements
- Compound statement
- Expression and null statements
- Selection statements
- Iteration statements
- Jump statements

Labeled statements

- Labels — i.e. targets of `goto` statements.
- Integer constant `case` statements in a `switch`.
- The `default` statement which a switch will jump to if no case matches.

Compound statement

- A compound statement, a block, can contain a sequence of statements and declarations.
- For instance:

```
int main(void)
{
        int     a;
        a = 1;
        int     b;
        b = 2;
}
```

- Mixing declarations and statements comes from C++ where some objects declared as local variables need this.
- In C there is no need to do this, and the main reason is simply (to quote a top engineer at Sony Mobile) that: ”**it’s ugly**”.

First declarations and then statements

- The following is cleaner:

```
int main(void)
{
        int     a;
        int     b;

        a = 1;
        b = 2;
}
```

Expression and null statements

- Most statements are expression statements, including assignments.
- A null statement does nothing and consists only of a semicolon.
- Null statements are used at end of blocks to avoid syntax errors:

```
int main(void)
{
        /* ... */
        if (p == NULL)
                goto cleanup;

        /* ... */

cleanup:
        ;
}
```

Selection statements: `if` and `switch`

- The controlling expression in a `switch` must be an integer.
- If there are initializations in the compound block of a `switch` they are not executed:

```
switch (a) {
        int     b = 10;

case 1:
        printf("a is one\n");
        a = b;  // invalid. b not defined.
                // falls through to case 2.

case 2: printf("a is two\n");
        break;

default:
        printf("hello from default\n");
}
```

Iteration statements

- Three loops: `for`, `while`, and `do-while`.
- A `for`-loop can have a declaration statement:

```
for (int i = 0; i < N; ++i)
        f(i);
```

- This was partly introduced to C due to C++ already had it and partly due to a false assumption that optimizing compilers would be helped by having the declaration close to the `for`-loop, which is nonsense.

New in C11: exact rules for optimizing away loops

- Consider the following loop:

```
int                i;
unsigned           b = 0;

for (i = 1; i; b += 1)
        ;
abort();
```

- Previously there were no rules regarding whether compilers are allowed to optimize away loops which never terminate and do not affect output by themselves.
- C11 says compilers may optimize away loops if they do not access atomic or `volatile` objects, perform I/O, or have a constant nonzero termination condition, e.g. `while (1) { }` must stay.

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;
int r = a - q * b;
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;
int r = a - q * b;
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.