

Contents of Lecture 9

- The C Preprocessor
- Statements
- Portable C

e.g., macros and other preprocessors to control the C compiler

Preprocessing directives

- **Predefined macros** Macros which provide info about compilation environment, compiler version, operating system etc. `__FILE__`, or `__DATE__`
- **Macro replacement** `#define`, to create symbolic names for expressions, constants, or code snippets
- **Conditional inclusion** `#if`, `#ifdef` `#else` `#elif`, enable or disable portions of code based on certain conditions.
- **Source file inclusion** `#include`, to include contents of another file in the current source file. For code reuse or external libraries
- **Line control** `#line` allows to control the line number and file name info that is reported during compilation. Useful for error messages or debugging
- **Error directive** `#error` generates compilation error with a specified error message.
- **Pragma directive** Provides a way to give special instructions to the compiler e.g., optimisation settings and platform-specific configurations
- **Null directive** `#`, empty, when a statement is required but no action is desired
- **Predefined macro names** Predefined names by the compiler to provide info about the environment,
- **Pragma operator** `#pragma`, allows inclusion of compiler-specific directives to control various aspects of compilation, such as optimisation settings or platform-specific configurations.

Predefined macros: 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 `yyymmmL` 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
#define good(a)          (a+1)
```

```
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 fencing-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 1
#define DEBUG 0    // invalid: cannot redefine a macro
#undef DEBUG
#define DEBUG 0    // OK. undefined first
```

```
#line 124 "a.scala"    // will set __LINE__ and __FILE__
```

 Check If defined

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

```
#pragma directive from user to compiler
#pragma ("directive from user to compiler")    // equivalent
```

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); => prog-015.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 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.

If we don't want the number of arguments to 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)  
  
if (a < b)  
    SWAP(a, b);
```

```
tmp = a; a = b; b = tmp;
```

Semi-colon doesn't work,
instead use do-while loops

- 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 For compound statements

- 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.

An alternative to macros: inline functions

- Inlining a function means copying the statements of a function into the calling function instead of doing the call.
- This can be done automatically by good compilers and should not be done by programmers — in my opinion at least.
- With C99 the keyword `inline` was introduced to C which can be used to give the compiler a hint that it might be a good idea to inline a function.
- Since good compilers can inline parts of a function automatically — and even copy rarely used parts of a function to some other place in memory it is much better to let the compiler take care of this.
- Use `inline` only if you use a poor compiler.

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 int x;
    static const int a[] = { 1, 2, 3 };

    return ++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 is OK.

Second restriction

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

```
extern inline int f(void)
{
    return g();
}
```

- 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` but then different files can have different functions `g`.

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 used if no case matches.

```
void f(void)
{
    for (...) {
        for (...) {
            for (...) {
                if (...)
                    goto fail;
            }
        }
    }

    return;

fail: /* clean up disaster. */ ;

}
```

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.

First declarations and then statements

- The following is cleaner in my opinion.

```
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 fail;

    /* ... */

fail:
    ;
}
```


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) // OK to remove this  
    ;  
abort();
```

```
for (;;) ;           // must remain in C11  
while (1) ;          // must remain in C11
```

- 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.

Writing Portable C Code

- Avoid undefined behavior.
- Write code with implementation-defined or unspecified behavior only when doing so cannot affect the observable behavior of your program.
- Avoid platform-specific system calls — stick to the Standard C library if possible.
- Do not exceed minimum compiler limits, eg number of parameters etc (this is mostly for machine-generated C).
- Appendix J of the C Standard has information on portability issues. Most of them are concerned with the Standard C library.

Examples of Unspecified Behavior 1(2)

- Whether string literals share memory.
- The order in which the operands of eg add are evaluated (discussed before).
- Whether `f()` or `g()` is called first in: `fun(f(), g())`.
- Whether `errno` is a macro or identifier with external linkage.
- The order in which `#` and `##` are evaluated during macro expansion.
- Which of two elements which compare equal is matched by `bsearch`.
- The order of two elements which compare equal when sorted by `qsort` (no surprise).

Examples of Unspecified Behavior 2(2)

- The resulting value at an overflow when converting a floating-point value to an integer.
- Whether the conversion of a non-integer floating point value to an integer raises the "inexact" exception.
- The order of side-effects during initialization, eg it is not specified whether `f()` or `g()` will be called first below:

```
int main()
{
    int a[] = { f(), g() };
}
```

Examples of Undefined Behavior 1(2)

- A "shall" or "shall not" requirement which appears outside a *constraint* is violated.
- A file ends in a comment `/* comment`.
- An identifier is first declared as `extern` and later as `static`.
- An invalid pointer is used:

```
int* fun()
{
    int    a;

    return &a;    // This pointer must not be used.
}
```

Examples of Undefined Behavior 2(2)

- Conversion to or from an integer which cannot be represented (also for conversion from floating-point to an unsigned).
- When a program attempts to modify a string literal:

```
char* s = "hello, world";  
s[0] = 'H'; // may crash.
```

- When an object is modified multiple times between two sequence points:

```
i = ++i + i++;
```

- / or % with the second operand being zero.

Examples of Implementation-Defined Behavior

- The number of bits in a `char`.
- Whether a `char` is signed or unsigned.
- How integer numbers are represented: not necessarily two's complement (but most of the world assumes that so you should too).
- Where to search for `#include <header.h>` files. In UNIX, use the switch `-I dir` to look in the directory `dir`.
- Endianness. Check on which format the data is stored when reading binary data using `fread`.