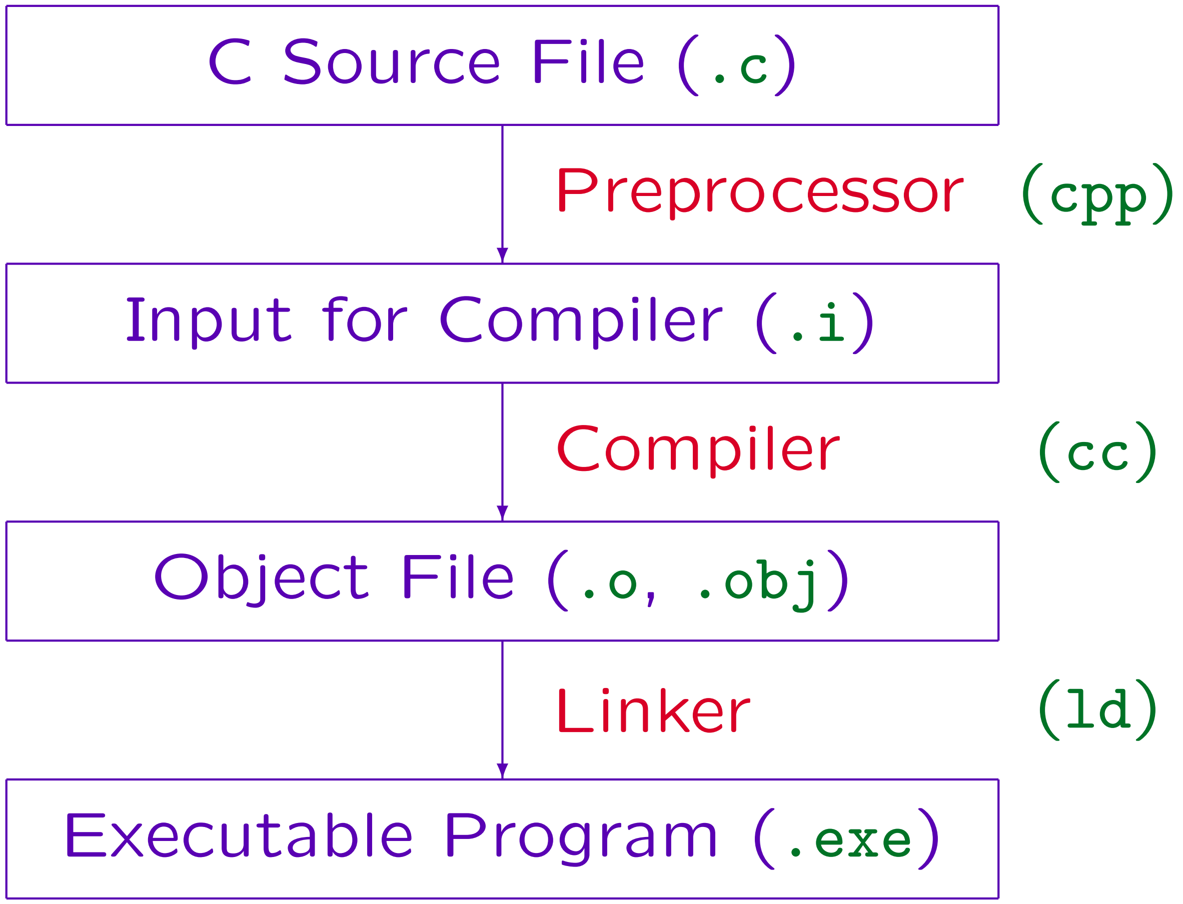
Chapter 5: C Syntax III (The Preprocessor)

References:

- Brian W. Kernighan / Dennis M. Ritchie:
  The C Programming Language, 2nd Ed.
  Prentice-Hall, 1988.
- Samuel P. Harbison / Guy L. Steele Jr.:
  C — A Reference Manual, 4th Ed.
  Prentice-Hall, 1995.
- Online Documentation of Microsoft Visual C++ 6.0 (Standard Edition):
  MSDN Library: Visual Studio 6.0 release.

Overview

Introduction (1)



Introduction (2)

- The C source file is first translated by the "C Preprocessor" (or precompiler) into a kind of "core C" (consisting of the constructs shown so far).
- The C compiler itself sees only the result of the precompilation stage. It translates the program into object code.

  Object code is machine code, but with additional information for the linker. The output of the compiler contains only machine code for the functions in the source program, machine code for library functions (or functions defined in other modules) must still be added.

Introduction (3)

- Today, the compiler and the preprocessor are usually combined in one program.

  At least under UNIX, there exists also a separate preprocessor called "`cpp`". This can also be used for preprocessing extended versions of other languages, not only C.

- However, the compiler usually has an option to do only the preprocessing and save the output in a file (normally with the extension "`.i`").

  Under Microsoft C++, this option is "`/P`". It seems that there is no special checkbox for it, but one can add it under "Project→Settings" in the "Project Options" field. No object file is produced as long as this option is set.

Introduction (4)

- The linker produces the executable program: It combines modules and adds code from libraries.
- By default, the C compiler automatically calls the linker for the generated object file (and the standard C library).

  Thus, one might not notice that compiler and linker are actually two separate programs. The linker can also link object files produced by compilers for other languages.
- When one writes programs consisting of several, separately compiled source files, the distinction of compiler and linker becomes more obvious.

Preprocessor Commands (1)

- Preprocessor commands begin with a "#" in the first column and extend to the end of the line.

  Actually, only old-style C requires that the "#" is the first character of the line. ANSI C (new style C) permits that it is preceded by blanks and tabulator characters. ANSI C and some older compilers also permit white space between the "#" and the preprocessor command (e.g. "include"). However, it has remained common practice to write the "#" in the first column.

- Example:

```
#include <stdio.h>
```

- Note that preprocessor commands do not end in a semicolon ";".

Preprocessor Commands (2)

- C has a general “line splicing” mechanism: If one writes a backslash “\” as the last character of a line, the backslash together with the line end are removed.

  This happens before tokens (lexical units) are built, so one can use it in the middle of a token: E.g. it was earlier used in long string constants (old-style C did not merge adjacent string constants).

- In this way, preprocessor commands can extend over several lines, if each line except the last ends in a backslash “\”.

Preprocessor Commands (3)

- The preprocessor understands the following commands ("directives"):
  - ◇ `#include`
  - ◇ `#define`, `#undef`
  - ◇ `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
  - ◇ `#line`

    This command is actually processed in the compiler, not (only) in the preprocessor (it sets the file name and line number for error messages).
  - ◇ `#error`
  - ◇ `#pragma`

Preprocessor Commands (4)

- Preprocessor commands are evaluated after tokenization.

  However, there are small differences in what constitutes a token: The "`<>`" file name after "`#include`" is a single token for the preprocessor. Characters that would be invalid for the compiler are accepted as tokens by the preprocessor. Between the defined macro name and the parameter-opening "`(`" no white space is allowed in the preprocessor. Otherwise tokens of preprocessor and compiler are the same.

- Therefore, one can use C comments on preprocessor lines.

  They are removed before the preprocessor sees the input. Also a line break inside a comment does not end the preprocessor command: The scanner replaces comments by a single space.

Preprocessor Commands (5)

- The syntax of preprocessor commands is relatively independent of the syntax of the rest of the language.
- E.g. macro definitions remain in effect until the end of the file, even if they were defined inside a function.

  The preprocessor does not understand the C syntax of functions, therefore it has no notion of "local commands".

Overview

Include Files (1)

- If the preprocessor sees e.g. the line

  ```
  #include <stdio.h>
  ```

  it replaces this line by the contents of the file "`stdio.h`" that is located in some system directory.

  E.g. "`C:\Program Files\Microsoft Visual Studio\VC98\Include`" for Microsoft Visual C++ or "`/usr/include`" under UNIX. There is a compiler/preprocessor option to define additional directories where include files are searched. In MS Visual C++, one can define them under "Project→Settings→C/C++→Precompiler" (or directly with `/I"C:\myinclude"`).

- Files like `stdio.h` are called include files or header files (or simply "headers").

Include Files (2)

- One can also write

  ```
  #include "stdio.h"
  ```

- In this case the preprocessor will first search the current directory for a file called “`stdio.h`”.
- If it does not find it there, it will search the system include directories.

  So this variant will also work, but it takes a little longer. One should use `#include <xyz.h>` for system include files and `#include "xyz.h"` for one’s own include files.

Include Files (3)

- Of course, file names can contain directories:

  ```
  #include "../headers/x.h"
  ```

  The syntax of file names is implementation-defined. Even the Microsoft Visual C++ compiler understands “/” as delimiter between directory and file name. After “`#include`”, the standard leaves the effect of “\” undefined, so one really should use “/” or a compiler option that extends the search path for include files.

- A final form (not in old-style C) is

  ```
  #include FILE
  ```

  where `FILE` is a macro that is replaced by one of the previous forms.

Include Files (4)

- Header files usually contain
  - ◇ Declarations of functions or global variables defined in one module and used in another module.

    "Module" means "translation unit" (source file). The defining module can also be contained in a library.
  - ◇ Declarations of types used in several modules.
  - ◇ Declarations of macros (see below).
- It is possible to put any C code into header files.

  However, function definitions (with the body) in include files are uncommon and poor style. Include files are usually included in several modules. Function definitions must be processed only once.

Include Files (5)

- Usually, the "`#include`" commands are written near the top of the source file (before declarations and function definitions).

  Maybe, for this reason include files are called "headers".

- Include files can (and often do) include other files.
- Of course, an include file cannot directly or indirectly include itself, or the compiler may get into an infinite loop.

  The nesting depth for include files is probably limited in most compilers, so the compiler will soon discover this error. At least 8 nesting levels are guaranteed by the ISO standard.

Overview

Parameterless Macros (1)

- If the preprocessor sees e.g. the line

  ```
  #define MAX_LINE_SIZE 80
  ```

  it will replace the token "`MAX_LINE_SIZE`" in the remaining input by "`80`".

- E.g. one can define an array as

  ```
  char input[MAX_LINE_SIZE+1];
  ```

- The compiler will really see the declaration:

  ```
  char input[80+1];
  ```

Parameterless Macros (2)

- In this way, the array size is a constant expression that the compiler can evaluate.

  C does not permit `const`-variables in constant expressions. In C++, this is possible. In general, one of the design goals of C++ was to remove the need for using "`#define`".

- Note that `MAX_LINE_SIZE` is only replaced if it forms a token in the source program: It is not replaced in string constants or comments and not replaced if it is only a part of a larger identifier.

Parameterless Macros (3)

- The replacement text for the macro can be any sequence of tokens, e.g. the following definition is legal (gives no error message):

```
#define MAX_LINE_SIZE = 80;    /* ERROR */
```

- However, the array declaration will then look to the compiler as

```
char input[= 80;+1];
```

  This of course gives a syntax error.

  But the source program contains an array declaration that looks completely right: `char input[MAX_LINE_SIZE+1];`

Parameterless Macros (4)

- It is common practice to use identifiers consisting only of capital letters as macro names.

  Variable, function, and type names are usually all lower case.

- Good programs should be easy to change: Therefore constants that are not absolutely fixed should appear only in a single place.

  Otherwise there can easily be inconsistent changes: The constant is changed in some places, but not in all. Some style guides require that no numbers except `0` and `1` appear anywhere in a program except on the right side of a `#define`.

Parameterless Macros (5)

- The replacement text can also be empty:

```
#define CHANGED_BY_SB
```

- This will remove the symbol from the program.

  This works like a comment. But such tricks are bad style, because they confuse other programmers who have to read the program.

- Such macros are often used as flags (boolean values) for conditional compilation: One can test with `#ifdef` whether a macro is defined (see below).

  In this case, the actual replacement text is not important and one simply leaves it empty.

Parameterless Macros (6)

- Once a symbol was replaced in the program ("expanded"), the replacement text is rescanned for other symbols that must be replaced.

  The compiler checks for infinite loops: A symbol that was already explanded is not expanded again if it appears directly or indirectly in its own replacement text.

- So the following replaces both N and M by 5, and the sequence of the two lines is not important:

```
#define N 5
#define M N
```

Parameterless Macros (7)

- It is illegal to redefine a macro that is already defined, unless the definition is exactly the same as the already existing definition.
- However, one can delete the definition of e.g. the macro `N` with

  ```
  #undef N
  ```

- After that, one can redefine it (but having a macro with two different values is bad style).

Macros with Parameters (1)

- Macros can have parameters, e.g.

```
#define max(N, M)    N < M ? M : N
```

- E.g. if the program contains

```
j = max(i,0);
```

the compiler will really see

```
j = i < 0 ? 0 : i;
```

- So in contrast to procedures, the "macro body" is inserted at each "macro call".

  This corresponds to a procedure call that is "unfolded".

Macros with Parameters (2)

- Using a macro is more efficient (with respect to CPU time) than using a real procedure: The overhead for the procedure call is not needed.

  For a procedure call, the parameters and the return address must be pushed on the stack, then a jump to the start address of the procedure is required, then several registers are saved on the stack. For the return, all these things must be undone.

- The generated machine code might be slightly larger if the macro is often called.

  However, if the replacement text is small, the generated machine instructions may actually require less space than the procedure call.

Macros with Parameters (3)

- However, macros are not without problems.

  There can be surprises. Good languages should minimize surprises.

- E.g. if the call of `max` looks as follows:

  ```
  j = max(i++, 0);
  ```

  this will be translated to:

  ```
  j = i++ < 0 ? 0 : i++;
  ```

  But now if `i` is not negative, it will be incremented two times. This differs from a real function call.

  Since one might actually not know whether something is a procedure or a function, it is better to avoid side effects in function calls.

Macros with Parameters (4)

- Suppose we define:

```
#define double(X)    X * 2
```

If the macro is called as follows:

```
n = double(i + 1);
```

The result performs not as expected:

```
n = i + 1 * 2;
```

Since `*` binds stronger than `+`, only `1` will be doubled, not `i + 1`.

Macros with Parameters (5)

- This problem is easy to solve: In the macro definition, one should always put parentheses around the parameters and around the whole expression:

```
#define double(X)   ((X) * 2)
```

- The definition of `max` should really look as follows:

```
#define max(N, M)   ((N)<(M) ? (M) : (N))
```

  There is no good solution for the other problem with `max`: Macros that access parameters more than once (or never) behave different than functions if arguments contain side effects.

Macros with Parameters (6)

- The replacement text can be any sequence of tokens, not only an expression:

```
#define assert(C)    if(!(C)) error()
```

- This leads to a surprise in this context (why?):

```
if(isupper(c))
        assert(c=='A'||...||c=='F');
else
        assert(isdigit(c));
```

- If `c` an upper case letter, one gets an error message (even for `A...F`). Otherwise nothing is checked.

Macros with Parameters (7)

- Also the following definition does not help:

```
#define assert(C)   { if(!(C)) error(); }
```

- Then one gets a syntax error for the else:

```
if(isupper(c))
        assert(c=='A'||...||c=='F');
else
        ...
```

- The problem is that the ";" after the assert becomes a null statement after the block `{...}`. Then one cannot add an `else`.

Macros with Parameters (8)

- The following is a correct definition of `assert`:

  ```
  #define assert(C) \
          ((void) ((C) || error()))
  ```

  If the replacement text for a macro is an expression, it behaves at least syntactically similar to a function call.

- A complete `if ... else ...` would also work in the above examples, but is slightly worse than the previous solution (it cannot be used in expressions):

  ```
  #define assert(C) \
          if(!(C)) error() \
          else (void) 0
  ```

Macros with Parameters (9)

- An arbitrary block (including declarations) can be quite safely packed into a `do`-loop:

  ```
  #define assert(C) \
          do { ... any code ... } while 0
  ```

  This executes the block exactly once.

  The declared variable names must be distinct from variables used in the arguments, or there are new surprises.

- Developing macros can be a bit tricky (one has to be careful).

  But, as far as I know, the above examples show everything that can go wrong. After you understood them, there should be no more surprises.

Macros with Parameters (10)

- But macros are a powerful tool. They have proven effective in many successful C projects.
- Having functions without the price of a function call improves the programming style: More code is encapsulated in functions/macros.

  Users of a data type should treat the data structure as a "black box" and access it only via functions/methods/macros that form the official interface ("abstract data type"). E.g. suppose that a linked list type is defined as

  ```
  typedef struct list_s { int data; struct list_s next; } list_t;
  ```

  Even if the user of the linked list can do an insertion in a few statements, he/she should call the official `insert` procedure, since later the implementation might be changed, e.g. by adding a "`back`" link.

Macros with Parameters (11)

- Because of the above problems with macros, C++ has "inline functions". They behave exactly like normal functions, but the compiler translates them by inserting the body for the function call.
- Inline functions and const-declarations in C++ eliminate most, but not all uses of `#define`.

  Macros are still needed when the arguments are not expressions, as e.g. in `MY_MALLOC(TYPE)`, or when one uses e.g. `#PAR` to turn a macro argument into a string (see below), or `__FILE__` and `__LINE__` to access the current source file name and line number (see below).

Syntax Details (1)

- In the macro declaration, there may be no space between the "`define`" and the opening "(".
- The following defines a parameterless macro:

```
#define max (N, M)    N < M ? M : N
```

- The macro call on slide 5-26 is replaced by

```
j = (N, M) N < M ? M : N (i,0);
```

- In calls of macros with parameters, spaces are allowed between the macro name and the "(".

Syntax Details (2)

- The actual arguments for a macro parameter can be any sequence of tokens.

  They do not have to be a syntactically well-formed expressions.

- However, commas must be put into parentheses.

  Otherwise the preprocessor assumes that they separate arguments. Of course, a comma in string or character constants is also no problem.

- Any macros in the arguments of a macro call are only substituted after the arguments of this macro call are collected.

  Therefore, it is e.g. no problem if an argument expands further to something that contains a comma.

Syntax Details (3)

- If the occurrence of a parameter in the replacement text (in the `#define`) is immediately preceded by “`#`” the actual argument will be put into string quotes.
- E.g. this macro can be used to print an error message if a pointer variable is NIL:

```
#define CHECK_PTR(P) \
        ((void)((P)||error(#P "is nil!")))
```

- The call `CHECK_PTR(list);` is then expanded to

```
((void)((list)||error("list" "is nil!")));
```

Syntax Details (4)

- The replacement text for a macro may also contain the special symbol "##".
- Then, immediately after the actual arguments are replaced for the formal parameters, the token before and after the "##" are merged into a single token by deleting any white space between them.

  The two special cases "#" and "##" are evaluated when macros in the arguments are not yet expanded (and they normally will not be expanded afterwards, although some further substitution of the token that results from "##" is possible due to the rescanning of the substitution result). Macros in normal arguments are replaced before the arguments are inserted in the replacement text.

Predefined Macros (1)

- `__LINE__`: Current line number.

  I.e. the macro is replaced by the line number in the source file where the macro call occurs. It is an integer constant.

- `__FILE__`: Current file name.
- `__DATE__`: Compilation date (e.g. `"Dec 20, 2001"`).
- `__TIME__`: Compilation time (e.g. `"15:45:00"`).
- `__STDC__`: Defined for ANSI/ISO C compilers.

  It has the value `1` if it is defined. There is also `__STDC_VERSION__` that is defined (and has the value `199409L`) if the compiler conforms to Amendment 1 of ISO C (then it has the type `wchar_t`).

Predefined Macros (2)

- E.g. the assert-macro can put the file name, line number and condition into the error message:

```
extern void __assert(const char *,
                     const char *, int);
#define assert(C) \
(void)((C)|| \
       (__assert(#C, __FILE__, __LINE__), \
        0))
```

This definition is from “assert.h”, which is part of the standard library. See below for the possibility to remove the assertions from the program with conditional compilation. The function __assert prints a message of the form: “Assertion failed: C, file F, line L” and then calls abort to terminate the program.

Predefined Macros (3)

- Every compiler defines some additional macros to identify itself for the purpose of conditional compilation (see below).
- Microsoft Visual C++ defines e.g.
  - ◇ `_WIN32`: Always defined.
  - ◇ `_MSC_VER`: C compiler version (1200 for 6.0).
  - ◇ `_M_IX86`: Defined for Intel x86 Processors.

    The value 500 means that code for a Pentium processor is generated, 300 means 386. `_M_ALPHA` is defined when code for DEC Alpha processors is being generated. There are also corresponding symbols for PowerPC and MIPS platforms.

Predefined Macros (4)

- The gcc compiler defines e.g. "`__unix__`" on UNIX systems.
- For C++ source files, the macro "`__cplusplus`" is defined.

  This is used to write header files that work with C and with C++ compilers.
- Macros can also be defined in the compiler options (compiler settings dialog box).

  In this way, some parameters that are different from installation to installation (e.g. the directory in which the program data will be stored) do not have to be written into the source files.

Overview

Conditional Compilation (1)

- Large software projects often need several versions of the program code, e.g.:
  - ◇ There is a debugging version and a version that is distributed to the customers.
  - ◇ There is a UNIX version and a Windows version.
- The C preprocessor makes it possible to compile part of the program code only when certain conditions are met.

  In this way, the same source file can contain different versions.

Conditional Compilation (2)

- Example:

```
#ifdef DEBUG
    printf("i = %d\n", i);
#endif
```

- If the macro `DEBUG` is not defined, the compiler will ignore all lines between the `#ifdef` and the corresponding `#endif`.

- Since it is only important whether the macro is defined, it usually has an empty replacement text:

```
#define DEBUG
```

Conditional Compilation (3)

- For the debugging version, `DEBUG` must be defined before the `#ifdef`, e.g. in a common header file, or as a compiler option.
- One can switch to the non-debug version by putting the `#define` line into a comment:

```
/* #define DEBUG */
```

It might be nicer to put this symbol into a compiler option in order to avoid changing the source file. In MS VC++, this is done under `Project`→`Settings`→`C/C++`→`General`. The input field "`Preprocessor Definitions`" contains a comma-separated list of defined macros. One can also use "`=`" to assign a value to the macro. Under UNIX, the compiler option is "`-DDEBUG`" or in general "`-D`⟨`Name`⟩`=`⟨`Value`⟩".

Conditional Compilation (4)

- `#ifndef` ("if not defined") ignores program code if the symbol is defined.
- `#if` can be used with a constant expression:

```
#if defined(_MSC_VER) && _MSC_VER >= 1200
        ... /* VC++ 6.0 and higher */
#else
        ... /* Version for other compilers */
#endif
```

- Of course, `#else` can also be used with `#ifdef` and `#ifndef`.

Conditional Compilation (5)

- The constant expression after `#if` is quite restricted. Since it is not evaluated by the C compiler itself, it can only contain integer literals/constants, arithmetic and logic operators, and the special condition `defined(Identifier)`.

  Of course, it can contain macros: Macros in the `#if`-condition are normally replaced before the condition is evaluated. The condition cannot contain the `sizeof` operator, type casts, or enumeration constants. All arithmetic is done with long or unsigned long values, even if one writes "`1`" it is implicitly taken as "`1L`". All identifiers that remain after macro expansion are replaced by "`0L`". Therefore, the above condition does not give a syntax error when `_MSC_VER` is not defined. Actually, the test `defined(_MSC_VER)` is not even necessary.

Conditional Compilation (6)

- "`#ifdef X`" is an abbreviation for "`#if defined(X)`" and "`#ifndef X`" stands for "`#if !defined(X)`".

  "`defined`" is new in ANSI C, it was not contained in old-style C.

- "`#ifdef`", "`#ifndef`", and "`#if`" can be nested.

  When the compiler skips source lines it does not simply stop at the next "`#endif`", but counts the number of open if's in order to find the matching "`#endif`".

- For "`#if`", there is also "`#elif`" ("else if") to distinguish more than two cases.

  If one used "`#else`" and "`#if`" instead, one would have to put as many "`#endif`" at the end as there were "`#if`".

Applications (1)

- Include files often make sure that they are processed only once in a compiler run.
- E.g. we might develop a file “list.h” that defines a type for linked lists of integers:

```
#ifndef LIST_INCLUDED
#define LIST_INCLUDED
typedef struct list_s { ... } *list_t;
...
#endif /* LIST_INCLUDED */
```

When the file is read for the first time, `LIST_INCLUDED` is not defined, and the file contents is processed. But then `LIST_INCLUDED` is defined so that the `ifndef` will be false if the file is included again.

Applications (2)

- Header files can include other header files that define needed types.
- E.g. suppose that we implement stacks as linked lists. Then "stack.h" could look as follows:

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED
#include "list.h"
typedef list_t stack_t;
...
#endif /* STACK_INCLUDED */
```

Applications (3)

- In this way, the user of “stack.h” does not have to include “list.h” first, but if he/she does (or some other header file does), it is also no problem.

  C permits that macros, variables, functions, and types are redeclared if the redeclaration is exactly the same as the original declaration. However, structure, union, and enumeration tags cannot be redeclared (unless the previous declaration was incomplete). It also does not help not to specify a tag, since then the type will by definition be new.

- Of course, if the same file is included multiple times, the compilation takes slightly longer.

  This is today not a big problem, since the computers are fast.

Applications (4)

- Macros can have alternative definitions. E.g. the standard include file "assert.h" contains the following definition:

```
#ifdef NDEBUG
#define assert(C) ((void)0)
#else
#define assert(C) \
(void)((C)|| \
       (__assert(#C, __FILE__, __LINE__), \
        0))
#endif
```

Applications (5)

- If the macro "`NDEBUG`" is defined when "`assert.h`" is included, calls to "`assert`" are simply removed from the program.

  They are replaced by "`((void)0)`", but a good compiler will produce no code for this. One could replace "`assert(C)`" simply by the empty token sequence, but then (1) syntax errors such as missing semicolons after the `assert` are not caught, (2) in the rare case that `assert` is used as part of a comma expression, one would get a syntax error.

- In this way, assertions help to find errors during debugging, but cost no runtime in the final product.

  Some important computer scientist (Hoare?) has said that this is like leaving the safety belt off in a car once you finished the driving school.

Applications (6)

- Many C include files are today made C++ compatible by adding a frame like this one:

```
#ifdef __cplusplus
extern "C" {
#endif
... /* Function Declarations */
#ifdef __cplusplus
}
#endif
```

- Since C++ has a different calling convention, functions compiled by a C compiler must be marked.

Applications (7)

- `extern "C" { ... }` is C++ syntax.

  A C compiler would produce a syntax error. But because of the conditional compilation, the C compiler does not see this part.

- An alternative is to define a macro "`EXTERN`" that is then put in front of every function declaration:

```
#ifdef __cplusplus
#define EXTERN extern "C"
#else
#define EXTERN extern
#endif
```

Overview

Setting Line Numbers (1)

- Sometimes C programs are produced by other programs (generators) from some specification.

  E.g. lex/flex, yacc/bison, precompilers for Embedded SQL.

- The specification may contain C code, which the generator simply copies to the output. If that C code contains errors, the programmer will get an error message when compiling the generated program.
- Then the programmer has to look into the generated program, find the corresponding place in the original specification, and correct the error there.

Setting Line Numbers (2)

- In order to simplify this, the generator may put lines of the following form into the generated C program:

```
#line 25 "scanner.l"
```

- If an error appears in the immediately following line, the C compiler will print an error message that refers to line 25 in "`scanner.l`", and not to the real line (e.g. 1000) in the real input file (e.g. `lex.yy.c`).

  Also the macros `__LINE__` and `__FILE__` are set accordingly.

Setting Line Numbers (3)

- It is not explicitly mentioned in the C Reference, but it seems that (some?) C compilers afterwards count from the given line number. E.g. if there is an error in the second but next line, the error message will refer to line 26 in `scanner.l`.

  This can be confusing. Of course, the generator expects that there are no errors in its generated code but if e.g. the interfacing of the specified code and the generated code does not work, one should get a message that refers to the generated file. The generator should switch back by a `#line` directive that mentions the generated file name and the real line number.

Setting Line Numbers (4)

- One can leave the file name out in the `#line` directive, then the compiler continues to use the last file name.
- Macros are expanded in the arguments to `#line`.
- Note that `#line` looks like a preprocessor directive, but it is really evaluated by the compiler.

  If the preprocessor is a separate program (which is today very seldom), it can also use `#line` commands to mark C code from include files. Of course, the compiler should report errors in include files correctly.

Error Messages

- One can also generate compiler error messages, e.g. in order to detect invalid settings for macros:

```
#if ELEMENTS > 32
#error "ELEMENTS can be at most 32!"
/* The set is implemented as a bitmap */
#endif
```

- The parameter of `#error` can be any sequence of tokens. It is printed as part of the error message.

  Macros are expanded. The `#error` directive is new in ANSI C.

Pragma (1)

- With `#pragma`, one can pass information to the compiler that depends on the specific compiler.
- E.g., it might be possible to
  - ◇ Selectively turn off warnings.
  - ◇ Set compiler options.
  - ◇ Request specific optimizations.
- E.g. in MS VC++, the following directive switches off warning 4514 (inline function not used):

```
#pragma warning(disable: 4514)
```

Pragma (2)

- E.g. in MS VC++, this pragma prints a message during the compilation (not an error or warning):

```
#pragma message("CGI version selected!")
```

- The C standard does not define the syntax of the `#pragma` arguments — this depends on the specific compiler. However, a compiler must ignore any `#pragma` specification that it does not understand.

  Still, it is possible that to compilers use the same syntax with different meanings. Therefore, it is better style to put the `#pragma` into an `#ifdef` that checks the compiler.

Other Features

- It is legal to write a "#" without anything following it, such lines are simply ignored ("Null Directive").
- In case one uses C on a computer that does not support the full ASCII character set, one can encode certain special characters as sequences of characters starting with "??" ("Trigraph Sequences"):

`??=`	`#`	`??(`	`[`	`??<`	`{`
`??/`	`\`	`??)`	`]`	`??>`	`}`
`??'`	`^`	`??!`	`|`	`??-`	`~`