(a) The Preprocessor

The '#include' Statement

- The `#include` statement includes another file into the current source
- This is usually being used for *header files*
- Where the preprocessor looks for it depends on the *quotes*
- System headers: `#include <stdio.h>`
  $\longrightarrow$ Search system directories
- Own headers: `#include "myheader.h"`
  $\longrightarrow$ Search the current directory
- But how to include a header which is neither in a system nor the current directory?
- The "*-I*" switch tells the compiler to search additional directories for (system and own) headers

Example:

```
% cat module.c
#include <myheader.h>
int main (void) { return 0; }

% gcc -I/home/herbert/include -c module.c
```

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

For All Companies placement Material

@placementclasses

For CAT Exam Preparation Material

@cat_classes

For GATE Exam Preparation Material

@gate_classes

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C

@c_examples

C++

@cpp_examples

Java

@java_examples0

Python

@python_examples

The '#define' Statement

- The `#define` statement allows to define macros
- Example: A name for a number: `#define MAXLEN 256`
- After this statement the preprocessor will *textually* replace all occurrances of `MAXLEN` by 256
- By convention macros are *all-uppercase*

Example:

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 256

void read_line (char *buf)
{
  int s;
  fgets (buf, MAXLEN, stdin);
  s = strlen (buf);
  buf[s-1] = ( buf[s-1] == '\n'? '\0': buf[s-1] );
}

int main (void)
{
  char inbuf[MAXLEN];
  read_line (inbuf);
  return 0;
}
```

'#define' vs. 'typedef'

- Some people use `#define` like this: `#define INT int`
  ⟶ So '`INT x, y;`' expands to '`int x, y;`'
- That's fine, but what about this: `#define PINT int*`
  ⟶ Here '`PINT x, y;`' expands to '`int* x, y;`'!!!
- Better solution: `typedef int* pint;`
- *'`#define`' is no replacement for '`typedef`'!*

Conditional Statements

- We can use the preprocessor to control what the compiler sees
- Therefore we use these: `#if`, `#elif`, `#else` and `#endif`
- The syntax is different from normal C-code
- The `defined` keyword allows to test for existing macros

Example:

```
#if defined(DEBUG)
printf ("The value of c is: %c\n", c);
#endif
```

We can use the compiler to manually define `DEBUG` instead of doing it in our source code:

```
% gcc -DDEBUG -c module.c
```

Portability Issues

- Macros are useful when writing code for different platforms
- Type definitions and path delimiters may be different
- Critical definitions should be put into a single header
- Errors can be handled with the `#error` statement

Example:

```
/*  portability header, compile with:       *\
\*    -DWIN32, -DMSDOS, -DLINUX or -DSUNOS  */

#if defined(WIN32) || defined(MSDOS)
#define PATHSEP '\\'
#else
#define PATHSEP '/'
#endif

#if defined(WIN32) || defined(LINUX)
typedef int int32;
#elif defined (SUNOS)
typedef short int32;
#elif defined (MSDOS)
typedef long int32;
#else
#error "Unsupported operating system!"
#endif
```

(b) Macros

Function-like Macros

- Macros can look like functions: `#define INT(a) (int)(a)`
- Difference to functions: the code gets *copied* into the source instead of entering a function
- Advantage: may be faster than real functions
- Disadvantage: lets the code grow, also side effects

Macro Pitfalls

- Bear in mind that macros do just *textual substitution*
- This can lead to ugly bugs.
- Example:
  - `#define POW2(a) (a*a)`
  - Good: `POW2(2)` $\longrightarrow$ `(2*2)`
  - Bad: `POW2(1+1)` $\longrightarrow$ `(1+1*1+1)`
  - Solution: `#define POW2(a) ((a)*(a))`
- Debugging macros: Run the preprocessor on the source and look at the result:

  ```
  gcc -E source.c
  ```