



The C Preprocessor

```
#define exprprintf(expr, \
    /* optional msg... */) \
    _exprprintf(#expr, \
    (int) (expr), \
    "%s: %d\n" msg)

#define __exprprintf(str_expr, \
    expr, fmt, args...) \
    printf(fmt, \
    str_expr, expr, ##args)
```

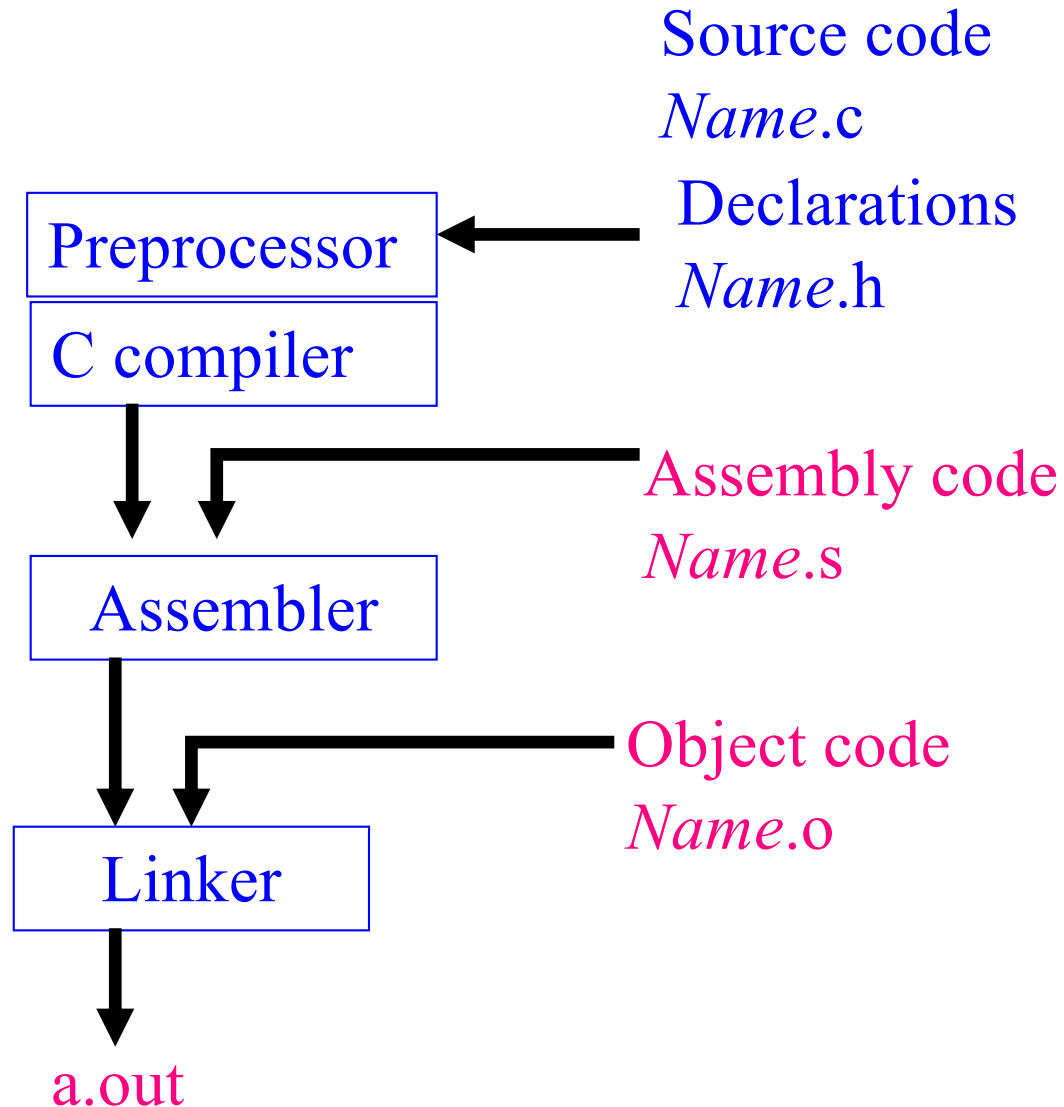


Object Code Files

- the C compiler can produce an *object code* version of a .c file that is machine language but not linked with other parts of your program
- these *object code* files end in .o

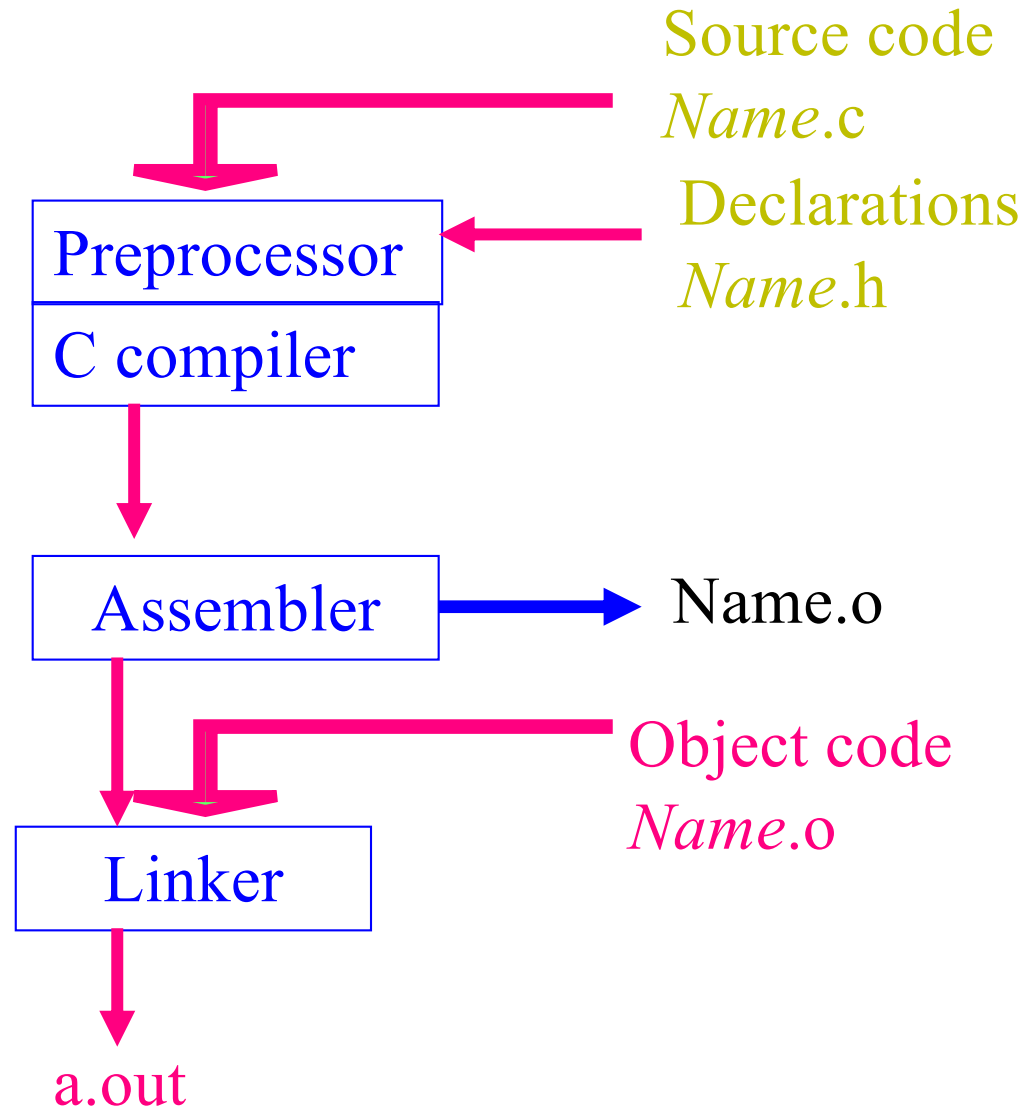


Compiling multi-file programs





Compiling multi-file programs





Compiling and Linking

- the C compiler can be instructed to produce the .o file from the .c using the -c flag, eg:

```
gcc -c util.c
```

- several .c or .o files can be combined to produce an executable program:

```
gcc myprog.c util.o -o myprog
```



- the *object code* files are linked together to form the final executable program
- after changing a .c file we only need to recompile the affected file into its object code (.o) form and then relink all the .o files to produce the executable



Preprocessor commands

- preprocessor commands are lines starting with #

eg #include

- The C preprocessor interprets these lines



Including Text from other files

- The `#include` statement is used to include text from another file into your program file at that point
- C programs typically consist of many source code files that each contain a small number of functions
- functions work on common data structures and so need declarations of the data structure to be included in each file



Including Text from other files

- rather than copy the declarations into every file (error prone!) we can use *include files*
- Useful for:
 - externs
 - typedefs
 - struct definitions
- can even nest the included files



Two Files: before

myprog.c

```
/* My Program */  
#include "decs.h"  
  
int main(int argc,  
         char *argv[])  
{  
    ...  
}
```

decs.h

```
/* Declarations */  
  
extern int count;  
  
struct employee  
{  
    ...  
}
```



After preprocessor:

```
extern int count;
```

```
struct employee  
{  
  ...  
}
```

```
int main(int argc,  
         char *argv[])  
{  
  ...  
}
```

Included text



- By convention, the names of included files end in ".h"
- So called “header” files because they tend to be included near the head of the program file



- Why shouldn't you include actual code?
- Why should you include relevant header files rather than simply have them in the code?



Defined Symbols

- An identifier symbol can be given a value by the preprocessor

```
#define LINES 100
```

- The preprocessor will replace the identifier LINES with the string 100 ***whenever*** it finds it in the program



Before preprocessor

myprog.c

```
/* My Program */  
#include "decs.h"  
char page[LINES]  
int main(int argc,  
         char *argv[])  
{  
    ...  
}
```

decs.h

```
/* Declarations */  
  
#define LINES 100
```



After preprocessor

Included text

Symbol
replaced

```
char page[100]
```

```
int main(int argc,  
        char *argv[])
```

```
{
```

```
...
```

```
}
```




Any replacement string

- The replacement string can be any string of characters:

```
#define LINES 5*10*20
```



Before preprocessor

myprog.c

```
/* My Program */  
#include "decs.h"  
char page[LINES]  
int main(int argc,  
         char *argv[])  
{  
    ...  
}
```

decs.h

```
/* Declarations */  
  
#define LINES 5*10*20
```



After preprocessor

Symbol
replaced

```
char page[5*10*20]

int main(int argc,
          char *argv[])
{
    ...
}
```



Warning!

- The replacement string can be any string of characters and replaces the symbol exactly

```
#define LINES 10+10
```



Before preprocessor

myprog.c

```
/* My Program */  
#include "decs.h"  
char page[LINES]  
int main(int argc,  
        char *argv[])  
{  
    pagesize = LINES * 5;  
}
```

decs.h

```
/* Declarations */  
  
#define LINES 10+10
```



After preprocessor

Symbol
replaced

```
char page[10+10]
```

```
int main(int argc,  
        char *argv[])
```

```
{
```

```
    pagesize = 10+10 * 5;
```

```
}
```

Important Tip!

- Always bracket expressions in defined symbols:

```
#define LINES (10+10)
```



Defined symbols: macros with parameters

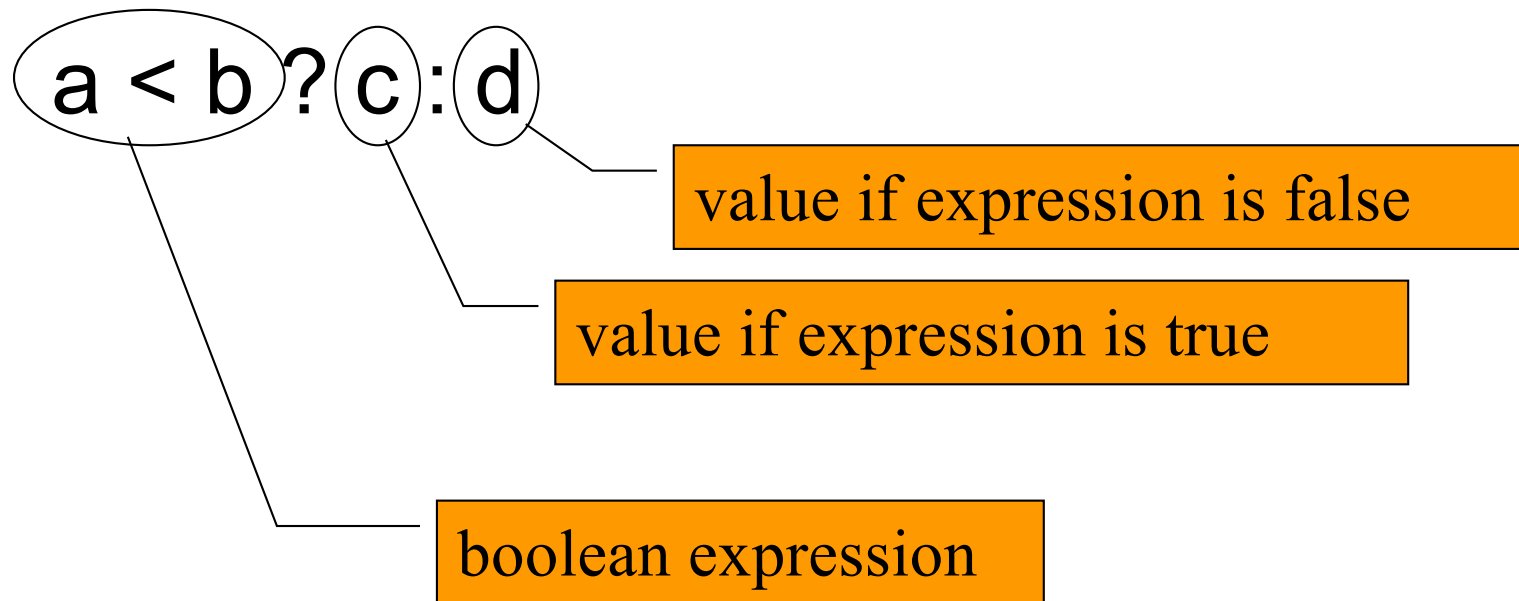
- A macro looks like a function with parameters
- A macro is processed by the preprocessor: replacing symbols in the body by parameters

```
#define min(a,b) ((a) < (b) ? (a):(b))
```




Note!

Ternary operator ?:



Before preprocessor

myprog.c

```
/* My Program */  
#include "decs.h"  
int main(int argc,  
         char *argv[])  
{  
    y = min(size, 100)  
}
```

decs.h

```
/* Declarations */  
  
#define min(a,b) ((a)<(b)?(a):(b))
```

After preprocessor

```
/* My Program */  
#include "decs.h"  
int main(int argc,  
        char *argv[])  
{  
    y = ((size)<(100)?(size):(100))  
}
```

macro call
replaced



Danger!

Beware of side-effects

```
y = min(a++,b) /* before */
```

```
y = ((a++)<(b) ? (a++):(b) /* after */
```

Danger!

Beware of side-effects

```
y = min(a++,b) /* before */
```

```
y = ((a++)<(b) ? (a++):(b) /* after */
```

a is incremented twice



General form of macro

#define *identifier(identifier,.....) token-string*



How can you tell a function from a macro?

examples:

if (isupper(ch)) ...

if (ch = getchar())...

if (ch = getc(**stdin**)) ...

if (ch = fgetc(**stdin**)) ...



How can you tell a function from a macro?

stdio.h

```
/* The C standard explicitly says this is a  
macro, so we always do the optimization for it.  
*/
```

```
#define getc(_fp) _IO_getc (_fp)
```

libio.h

```
extern int _IO_getc (_IO_FILE *__fp);
```


End of Segment

The C Preprocessor:

conditional inclusion



Conditional inclusion

- the preprocessor allows you to select text to be included or not
- very useful for debugging: include debug printouts or not - controlled by preprocessor command



Conditional inclusion

`#ifdef`

`#if`

`#ifndef`

`#else`

`#elif`

`#undef`

`#endif`

Conditional inclusion

#ifdef tests if a preprocessor symbol is defined

eg

```
#define DEBUG
```

```
#ifdef DEBUG
```

```
    printf("loop counter = %d\n", count);
```

```
#endif
```

no need to
give a value

Conditional inclusion

`#ifndef` tests if a preprocessor symbol is *NOT* defined

eg

```
#ifndef FASTLINK
    .... /* code for slow links */
#endif
```

Conditional inclusion

`#if` allows more complex expressions to be used

eg

```
#if WINDOWWIDTH > 600
.... /* code for wide windows */
#endif
```

Conditional inclusion

Both `#ifdef` and `#if` can have an `#else`
eg

```
#ifdef DEBUG
    .... /* Debugging version */
#else
    ... /* production version */
#endif
```


Conditional inclusion

`#if` can have an `#elif`

eg

```
#if WIDTH > 600
    .... /* wide version */
#elif WIDTH > 400
    ... /* medium version */
#else
    ... /* narrow version */
#endif
```



Before preprocessor

```
#include "declarations.h"
int main(int argc, char *argv[])
{
    #ifdef DEBUG
        printf("MyProg (debug version)\n");
    #else
        printf("MyProg (production version)\n");
    #endif
    return 0;
}
```

declarations.h

```
#define DEBUG
```



After preprocessor

```
int main(int argc, char *argv[])  
{  
  
    printf("MyProg (debug version)\n");  
  
}
```

Controlling the preprocessor from the gcc command

`gcc -DWIDTH=600 prog.c`

has the same effect as

`#define WIDTH 600`

at the beginning of the program

`#define` or `#undef` **within** the program
overrides the command line setting

gcc -Didentifier

is equivalent to

#define identifier

multiple -D arguments can be used:

gcc -DWIDTH=600 -DTEST prog.c



Useful for debugging

gcc -DEBUG prog.c

prog.c:

```
#ifdef EBUG
#define DEBUG(m) printf("debug: %s\n", (m))
#else
#define DEBUG(m) /* null statement */
#endif

...
    DEBUG("called proc fn");
...
```

Alternatives

```
#ifdef DEBUG
```

```
    printf(...)
```

```
#endif
```

compared with

```
enum {DEBUG = 0}
```

```
...
```

```
if (DEBUG)
```

```
    printf(...)
```


Pre-defined Symbols

- the preprocessor defines several symbols automatically
- the most useful of these are:

`__LINE__` contains the current line number at any point

`__FILE__` contains the name of the current program file



When would you need them?

- LINE ?
- FILE ?



Debug example revisited

gcc -DEBUG prog.c

continuation
indicator

prog.c:

```
#ifdef EBUG
#define DEBUG(m) \
    printf("debug: %s at line %d in file %s\n", \
           (m), __LINE__, __FILE__)
#else
#define DEBUG(m) /* null statement */
#endif

...
    DEBUG("called doit function");
...
```

#include revisited

- the normal form of #include has a file name in double quotes - this specifies a relative or absolute path for the file
- if the file name is enclosed in <> the file is searched for in /usr/include
- the preprocessor can be instructed to look in other directories using the -I directory flag
- this allows you to have your own directory for include files

Example

```
#include <defs.h>
```

- the preprocessor will look in
 /usr/include for the defs.h file
- if we use the command
 gcc -I/home/john/include myprog.c
the preprocessor will look in
 /home/john/include for the file

Caution!

```
#define IF          if(  
#define THEN      )  
#define BEGIN     {  
#define END       }
```

```
IF a == 1 THEN  
BEGIN  
    dothis()  
    dothat();  
END
```

A new language!
Unreadable for the
next programmer.



Preprocessor as a tool

- `gcc -E`
- runs just the preprocessor
- can be used as a tool exploiting
 - `#define` call by name
 - `#ifdef` for conditional generation



Example: hack templates

- Generate text in different forms as required:
 - `#include` parts
 - `#define` to replace parts



Role of preprocessor

- lots of it around, especially for
 - machine-dependencies
 - OS versions
- pretty hard to read code with lots of conditional compilation through it
- how to debug the conditional compilations
preprocessor “code”?

- the preprocessor is very useful for configuring programs
 - different versions
 - debugging
- not found in Java
- image: <https://packagecontrol.io/packages/C%20Improved>

End of Segment