

CSE 374 Lecture 16

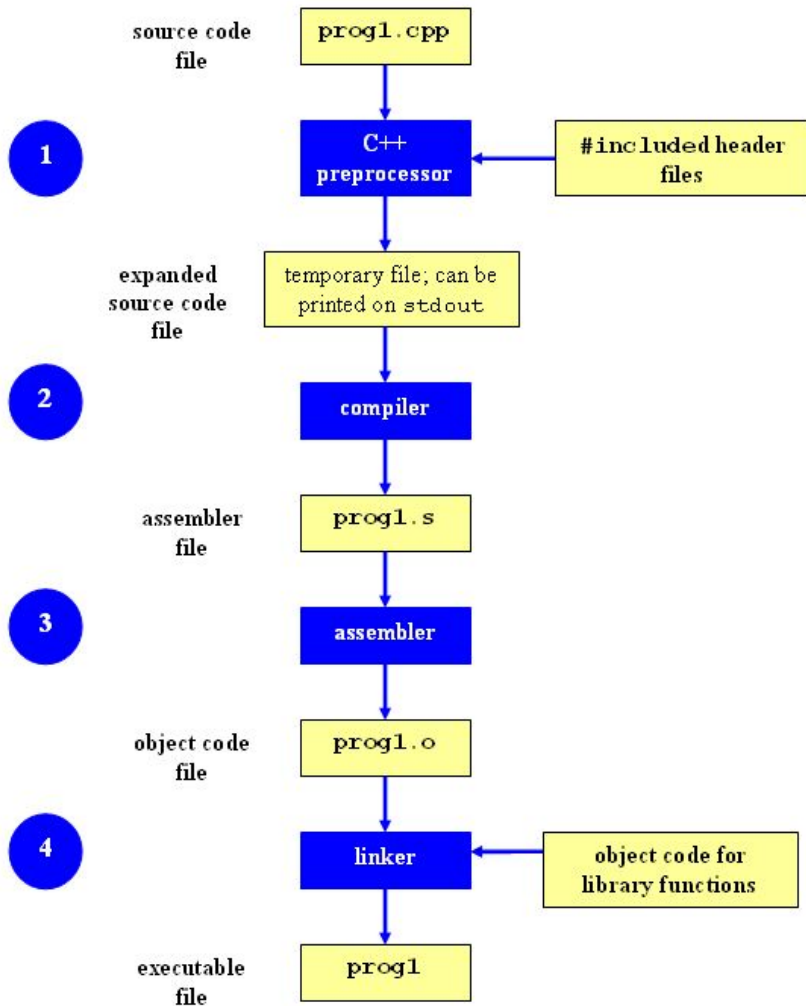
Week 6: More preprocessor, Multiple Files



Compiling in more detail

Compilation process is actually
multi-step

Multi-file compilation requires
knowing more details



Stop after the preprocessor and store the preprocessed C file in file.pp

```
$ gcc -E file.c > file.pp
```

Stop after the compiler and store the assembly code in file.s

```
$ gcc -S file.c
```

Stop after the assembler and store the machine code in file.o

```
$ gcc -c file.c
```

Preprocessor Review

The preprocessor rewrites code before the compiler gets it.

Has multiple roles:

Include header files

Define Constants

Define Macros

Conditional Compilation

(and header files)

```
#include <stdlib.h>
```

```
#include <userfile.h>
```

Header files

Always use '.h',

Headers include function, struct,
constant declarations

Never include function implementations

Never include '.c'

```
$gcc -I : look in specific  
directories
```

Symbolic Constants & Macros

- Creates TOKEN to represent more text
- Preprocessor:
 - ◆ Replaces all matching TOKENS in rest of file
 - ◆ Knows where words start and end
 - ◆ Has no notion of scope (not the compiler)
- Can shadow another #define
- Use #undef to remove

Constants:

```
#define SYMBOLIC_CONSTANT value
#define NOT_PI 22/7
#define VERSION 3.14
#define FEET_PER_MILE 5280
#define MAX_LINE_SIZE 5000
```

Macros

Replace all matching “calls” with “body”
but with text of arguments where the
parameters are (just string substitution)

Gotchas (understand why!) ->

Macros DO NOT avoid performance
overhead of a function call (maybe true in
1975, not now)

Macros CAN BE more flexible though
(type-inspecific)

```
#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)
double twice(double x) {
    return x+x; }
```

```
y=3;
```

```
z=4;
```

```
w=TWICE_AWFUL(y+z); [y+z*2]
```

```
z=TWICE_BAD(++y); [++y + ++y]
```

```
z=TWICE_BAD(y++); [y++ + y++]
```

Macros: debugging

*Remember - it's just
pure string
replacement.*

```
#define TWICE_AWFUL(x) x*2

int main(int argc, char **argv) {
    int x = 1;
    int y = 2;

    // This gives 5 instead of 6
    printf("Twice(1+2) is 6, but %d\n",
        TWICE_AWFUL(x+y));

    ...
}
```

Macros: debugging

*Remember - it's just
pure string
replacement.*

```
#define TWICE_AWFUL(x) x*2

int main(int argc, char **argv) {
    int x = 1;
    int y = 2;

    // This gives 5 instead of 6
    printf("Twice(1+2) is 6, but %d\n",
        x+y*2;

    ...
}
```

Justifiable Macros

Parameterized macros are generally to be avoided (use functions)

There are things functions cannot do:

```
#define NEW_T(t, howmany) ((t*)malloc((howmany)*sizeof(t))
```

```
#define PRINT(x) printf("%s:%d %s\n", __FILE__, __LINE__, x)
```

Be very careful with syntax if you do use them

Conditional Compilation

```
#ifdef FOO
// only compiled if FOO is defined
#endif
```

```
#ifndef FOO
// only compiled if NOT FOO
#endif
```

```
#if FOO > 2
// only compiled if FOO > 2
#endif
```

```
// use DBG_PRINT for debug-printing
#ifdef DEBUG
#define DBG_PRINT(x) printf("%s",x)
#else
```

```
// replace with nothing
#define DBG_PRINT(x)
#endif
```

```
DBG_PRINT("hello world!\n");
```

```
$ gcc -D DEBUG foo.c
// or with #define
```

#ifndef: header file inclusion

```
#ifndef FOO_H
```

```
#define FOO_H
```

and end it with:

```
#endif
```

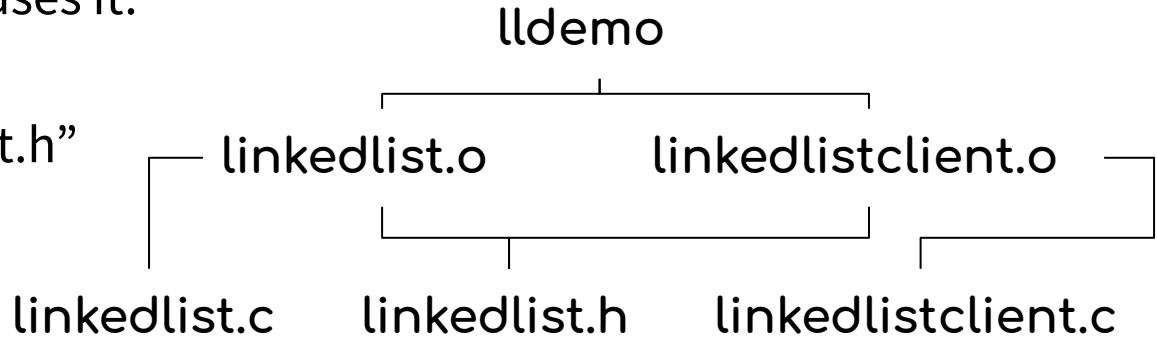
- Assuming nobody else defines SOME_HEADER_H (convention)
 - first #include "some_header.h" will do the define and include the rest of the file
 - second and later will skip everything
- More efficient than copying the prototypes over and over again
- In presence of circular includes, necessary to avoid “creating” an infinitely large result of preprocessing

Linked List Continued

- One set of code to define linked list:
 - `Linkedlist.h`
 - `Linkedlist.c`
- Another piece of code uses it:
 - `Linkedlistclient.c`
 - `#include "linkedlist.h"`

Compile with

```
$gcc -o lldemo linkedlist.c  
linkedlistclient.c
```



Dependency Tree - *helps decide what to do*

Each target T is dependent on one or more sources S

If any S is newer than T, remake T.

Recursive: If a source is also a target for other sources, must also evaluate its dependencies and possibly remake

Directed-acyclic-graph
(cycles make no sense)

