C Programming Essentials

Unit 3: Basic Data Structures

CHAPTER 7: PRE-PROCESSING

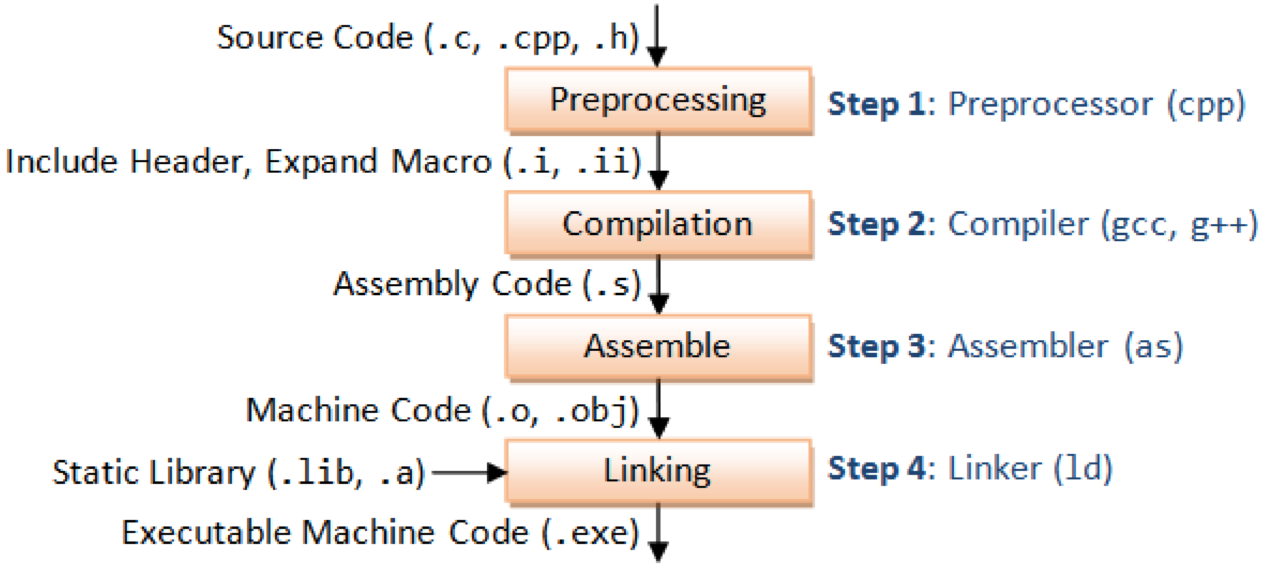
DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

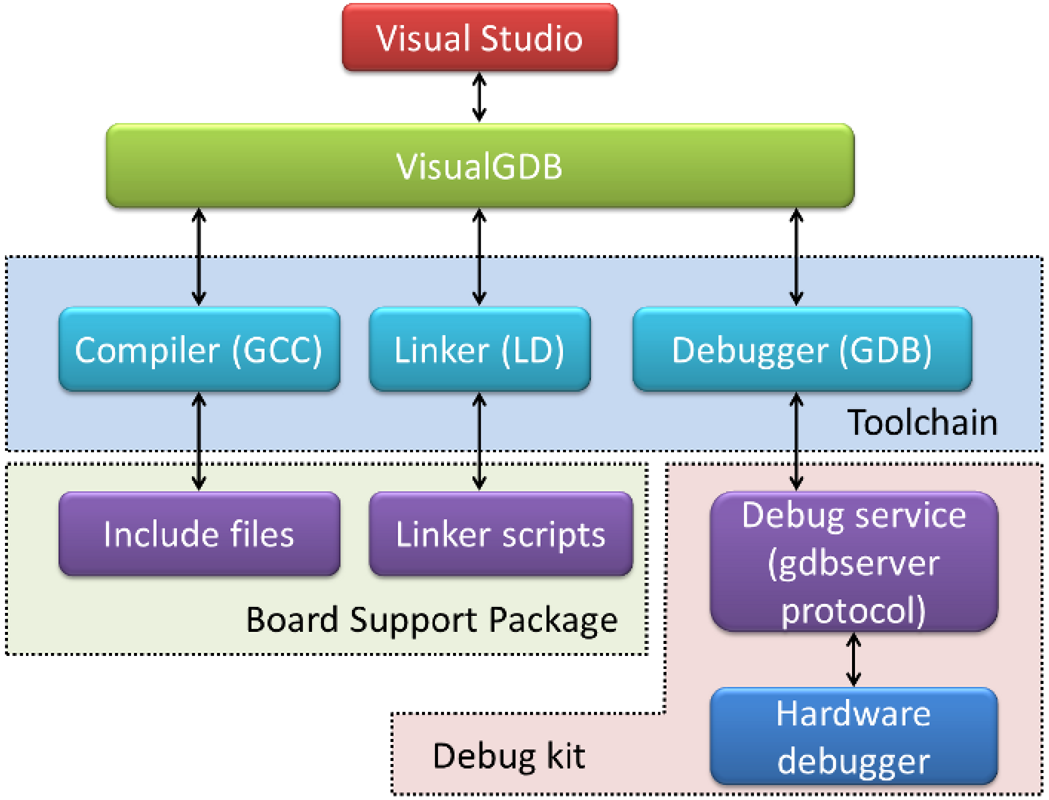
Preprocessing

GNU Tool Set



Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

GDB
The GNU Project Debugger

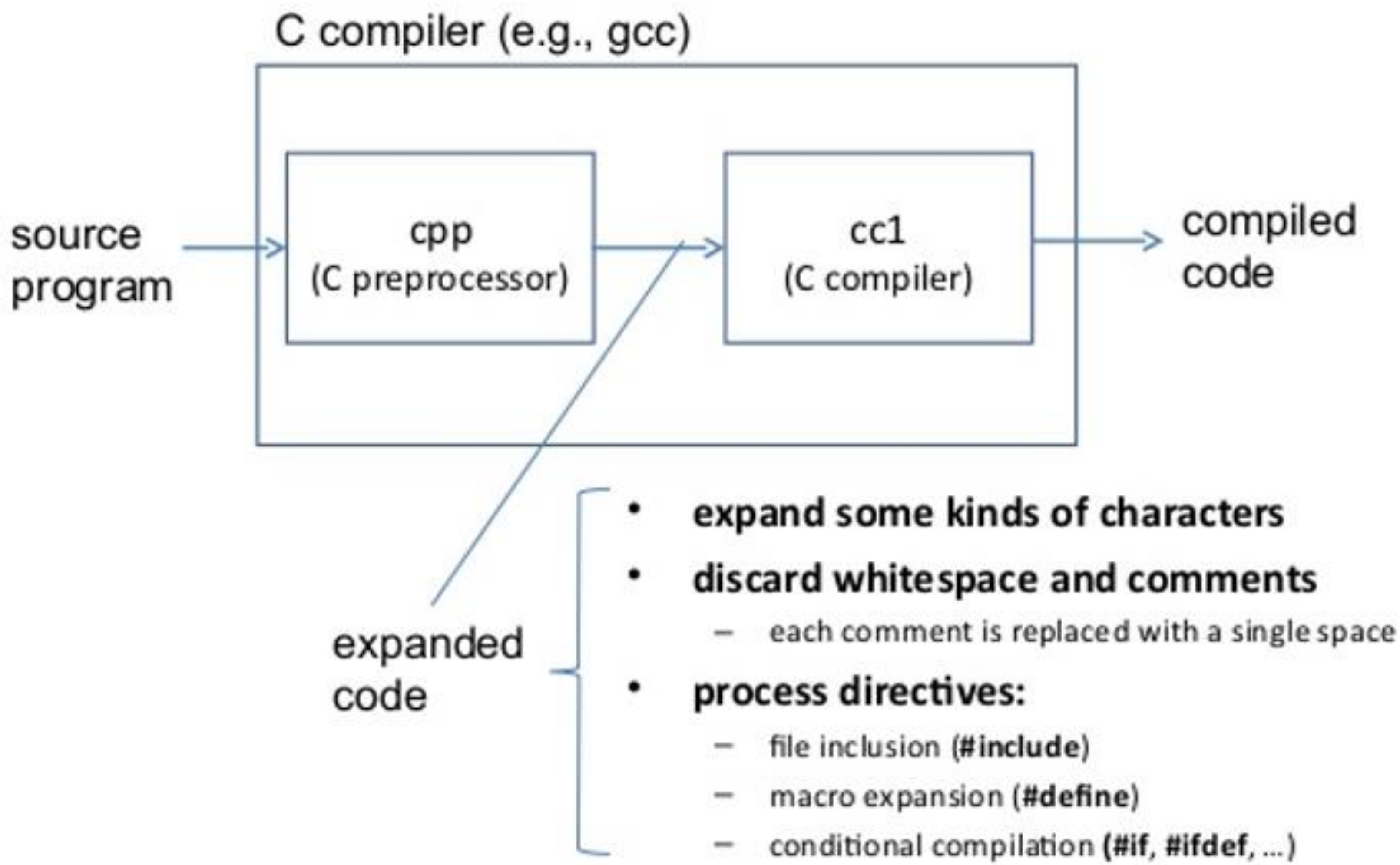


Visual Studio and GCC+LD+GDB

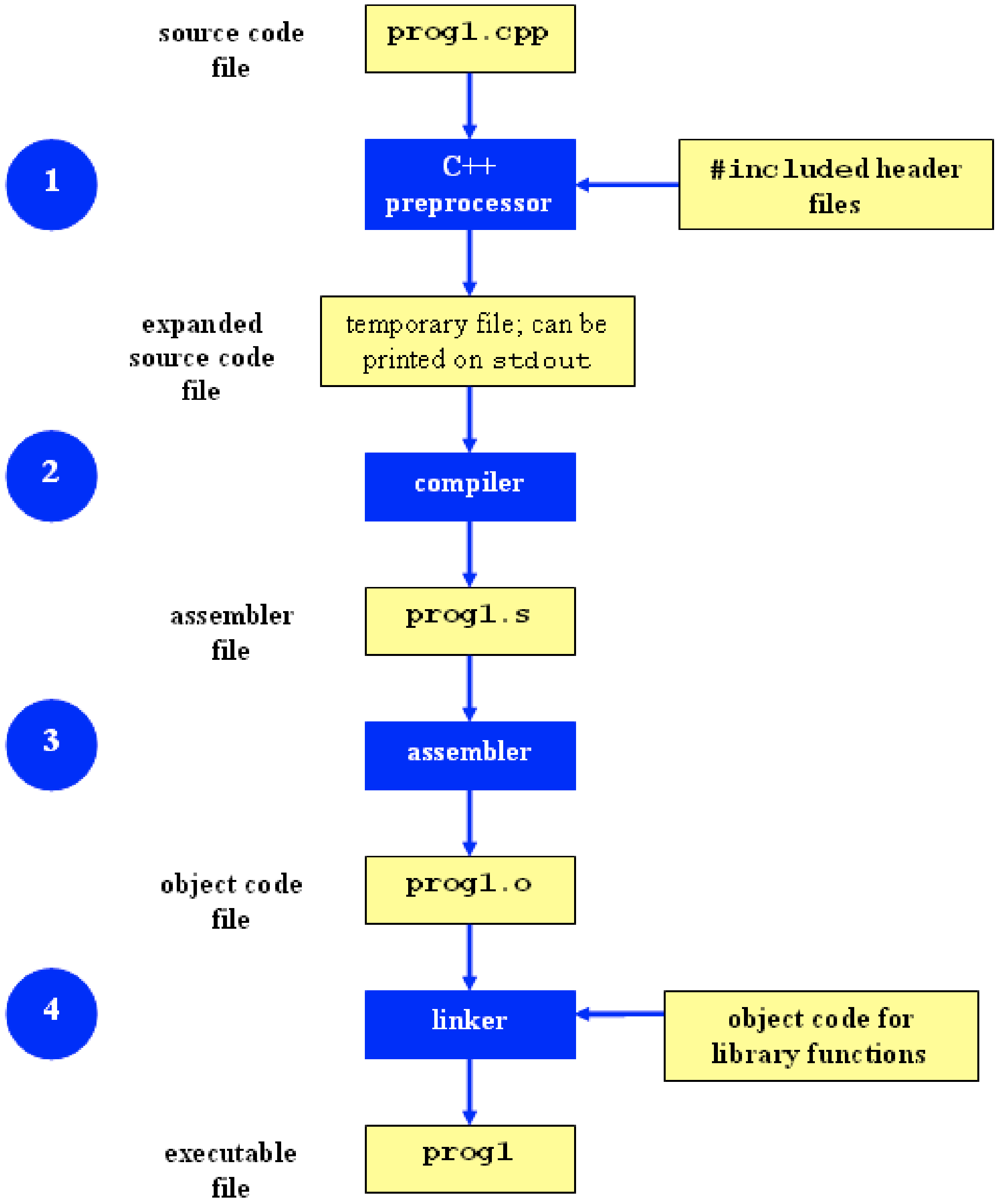
Overview

- The C preprocessor, often known as **cpp**, is a macro processor that is used automatically by the C compiler to transform your program before compilation.
- It is called a **macro processor** because it allows you to define macros, which are brief abbreviations for longer constructs.
- The C preprocessor is intended to be used only with C, C++, and Objective-C source code.

The C preprocessor and its role



- expand some kinds of characters
- discard whitespace and comments
  - each comment is replaced with a single space
- process directives:
  - file inclusion (**#include**)
  - macro expansion (**#define**)
  - conditional compilation (**#if**, **#ifdef**, ...)



Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named prog1.cpp and you execute the compile command

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

The compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

gcc and g++'s options are similar. The same flow can also be used for C language.

Character sets

- Source code character set processing in C and related languages is rather complicated. The C standard discusses two character sets, but there are really at least four.
- The files input to CPP might be in any character set at all. CPP's very first action, before it even looks for line boundaries, is to convert the file into the character set it uses for internal processing. That set is what the C standard calls the source character set. It must be isomorphic with ISO 10646, also known as Unicode. **CPP uses the UTF-8 encoding of Unicode.**
- The character sets are specified using the -finput-charset= option.
- All preprocessing work is carried out in the source character set. If you request textual output from the preprocessor with the -E option, it will be in UTF-8.

GCC Steps and Partial Building Results

By using appropriate compiler options, we can stop this process at any stage.

1. To stop the process after the preprocessor step, you can use the **-E** option:

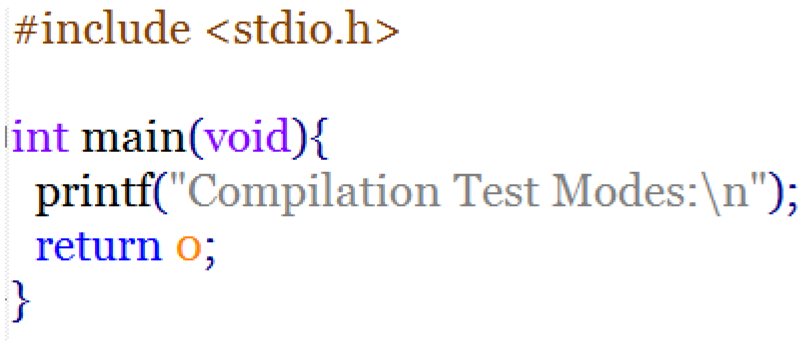
   -
     ```
     g++ -Wall -std=c++11 -E prog1.cpp > prog1.e
     ```
   - The expanded source code file will be printed on standard output (the screen by default); you can redirect the output to a file if you wish. Note that the expanded source code file is often incredibly large - a 20 line source code file can easily produce an expanded file of 20,000 lines or more, depending on which header files were included.
2. To stop the process after the compile step, you can use the -S option:

   -
     ```
     g++ -Wall -std=c++11 –O2 -S prog1.cpp
     ```
   - By default, the assembler code for a source file named filename.cpp will be placed in a file named **filename.s**.   // no re-direction needed
3. To stop the process after the assembly step, you can use the -c option:

   -
     ```
     g++ -Wall -std=c++11 -c prog1.cpp
     ```
   - By default, the assembler code for a source file named filename.cpp will be placed in a file named **filename.o**.
4. To complete the whole compilation process use –o **executable_filename**

GCC Partial Results

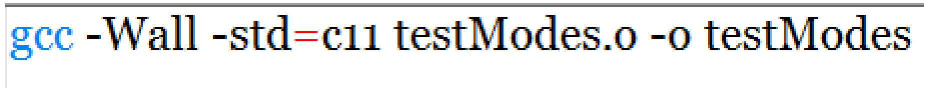
Go gcc!!!



```c
#include <stdio.h>

int main(void){
  printf("Compilation Test Modes:\n");
  return 0;
}
```

buildexe2.bat -> testModes.exe



```
gcc -Wall -std=c11 testModes.o -o testModes
```

buildasm.bat (generate a.exe, testModes.o and re-directed to testModes.asm)

```
gcc -Wall -std=c11 -g -c testModes.c
objdump -d -M intel -S testModes.o > testModes.asm
```

buildasm2.bat -> testModes.s

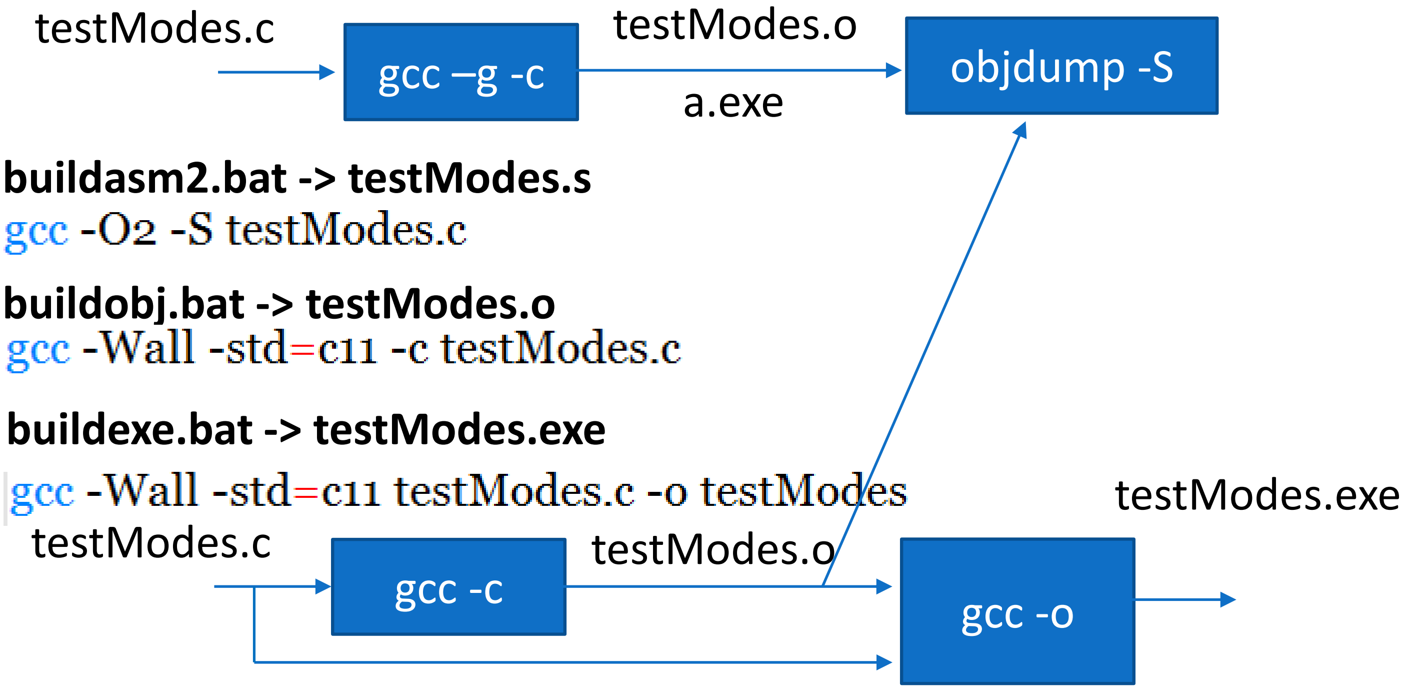
```
gcc -O2 -S testModes.c
```

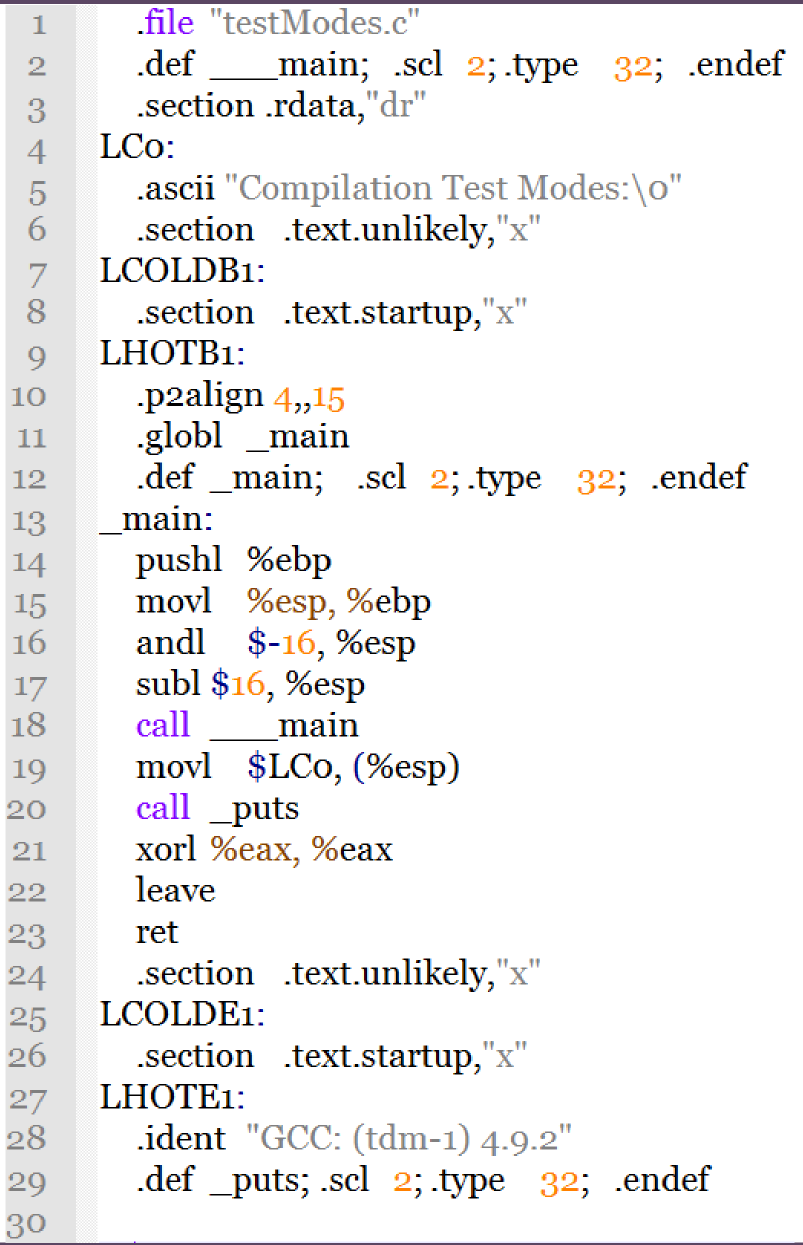
buildobj.bat -> testModes.o

```
gcc -Wall -std=c11 -c testModes.c
```

buildexe.bat -> testModes.exe

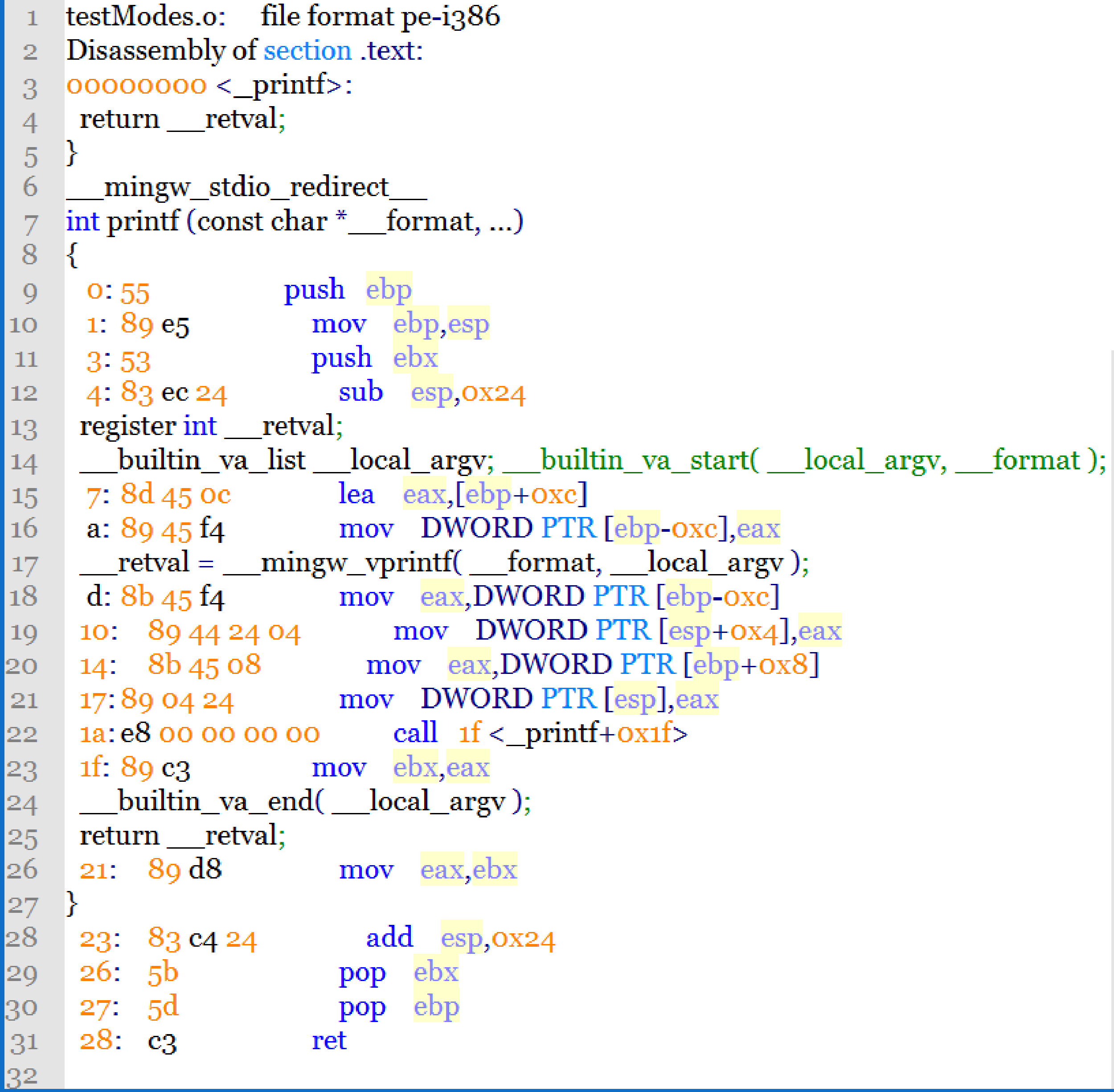
```
gcc -Wall -std=c11 testModes.c -o testModes
```



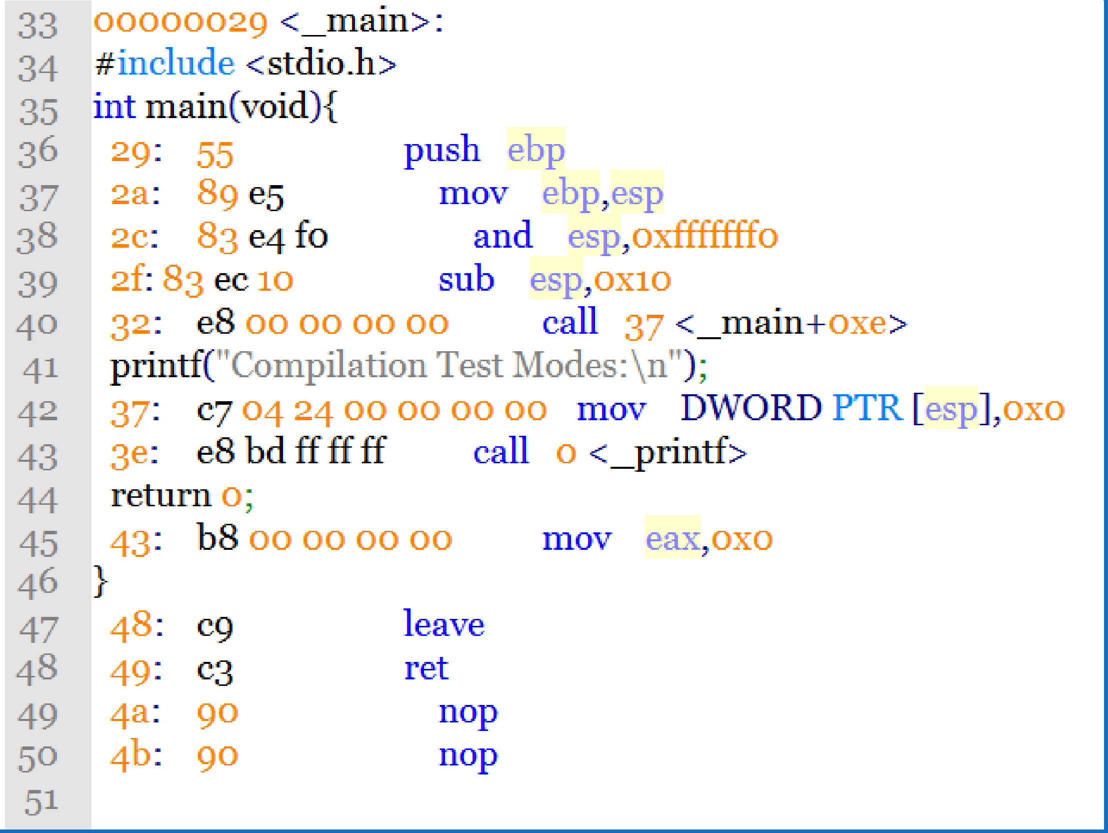


```
	.file	"testModes.c"
	.def	___main;	.scl	2;	.type	32;	.endef
	.section .rdata,"dr"
LC0:
	.ascii "Compilation Test Modes:\0"
	.section	.text.unlikely,"x"
LCOLDB1:
	.section	.text.startup,"x"
LHOTB1:
	.p2align 4,,15
	.globl	_main
	.def	_main;	.scl	2;	.type	32;	.endef
_main:
	pushl	%ebp
	movl	%esp, %ebp
	andl	$-16, %esp
	subl	$16, %esp
	call	___main
	movl	$LC0, (%esp)
	call	_puts
	xorl	%eax, %eax
	leave
	ret
	.section	.text.unlikely,"x"
LCOLDE1:
	.section	.text.startup,"x"
LHOTE1:
	.ident	"GCC: (tdm-1) 4.9.2"
	.def	_puts;	.scl	2;	.type	32;	.endef
```

testModes.asm
In report style.



```
testModes.o:     file format pe-i386
Disassembly of section .text:
00000000 <_printf>:
 return __retval;
}
__mingw_stdio_redirect__
int printf (const char *__format, ...)
{
  0: 55                push   ebp
  1:  89 e5              mov    ebp,esp
  3: 53                  push   ebx
  4: 83 ec 24              sub    esp,0x24
 register int __retval;
 __builtin_va_list __local_argv; __builtin_va_start( __local_argv, __format );
  7: 8d 45 0c            lea    eax,[ebp+0xc]
  a: 89 45 f4            mov   DWORD PTR [ebp-0xc],eax
 __retval = __mingw_vprintf( __format, __local_argv );
  d: 8b 45 f4            mov   eax,DWORD PTR [ebp-0xc]
 10:   89 44 24 04          mov   DWORD PTR [esp+0x4],eax
 14:   8b 45 08          mov   eax,DWORD PTR [ebp+0x8]
 17: 89 04 24            mov   DWORD PTR [esp],eax
 1a: e8 00 00 00 00        call  1f <_printf+0x1f>
 1f: 89 c3            mov   ebx,eax
 __builtin_va_end( __local_argv );
 return __retval;
 21:   89 d8             mov   eax,ebx
}
 23:   83 c4 24           add   esp,0x24
 26:   5b               pop   ebx
 27:   5d               pop   ebp
 28:   c3             ret
```



```
00000029 <_main>:
#include <stdio.h>
int main(void){
 29:   55              push  ebp
 2a:   89 e5              mov   ebp,esp
 2c:   83 e4 f0              and   esp,0xfffffff0
 2f: 83 ec 10            sub   esp,0x10
 32:   e8 00 00 00 00        call  37 <_main+0xe>
 printf("Compilation Test Modes:\n");
 37:   c7 04 24 00 00 00 00  mov   DWORD PTR [esp],0x0
 3e:   e8 bd ff ff ff        call  0 <_printf>
 return 0;
 43:   b8 00 00 00 00        mov   eax,0x0
}
 48:   c9              leave
 49:   c3              ret
 4a:   90                nop
 4b:   90                nop
```

LECTURE 2

Macro

C Preprocessor

Modifies C code "to save typing"

- Define constants
- Define macros
- Include files
- Other parameters (time of compilation...)
- Conditional Compilation

Macro Definition and Expansion

Object Like:

#define <identifier> <replacement token list>

Example:

```
#define PI 3.14159
```

Function Like:

#define <identifier>(<parameter list>) <replacement token list>

Example:

```
#define RADTODEG(x) ((x) * 57.29578)
```

Preprocessor constants (I) Constants for

- Define a symbolic constant like so

```
#define   PI    3.141526
```

- Better version

```
#define   PI    ( 3.141526 )
```

- Use the symbolic constant

```
circle_length = 2 * PI * radius ;
```

Preprocessor constants (2)

Constants used as #define switch

Check if constant defined (#ifdef)

```
#define  VERBOSE

. . .

#ifdef  VERBOSE

          printf("I am extremely glad to see you !\n");

#else

          printf("Hi !\n");

#endif
```

Preprocessor Macros

Replacement of Text

Parameterized Macros:

Similar to function calls. Symbolic parameters !

```
        #define SQUARE( x )       x * x
```

Better version:

```
        #define SQUARE( x )       ((x) * (x))
```

Usage: *What will be the output for each version?*

```
 int x
 x = SQUARE (  1 + 2 + 3 );
 ⇔ (1+2+3*1+2+3) =???
 printf( " x = %d \n", x ); is x=11?, or is it, x=36?
```

How do you fix it to generate 36? → ((1+2+3) * (1+2+3))

Macro Definition and Expansion

Function Like: Be careful!

Example:

```
#define MAC1(x) (x * 57.29578)
```

will expand MAC1(a + b)

to (a + b * 57.29578)

```
#define MIN(a,b) ((a)>(b)?(b):(a))
```

What happens when called as

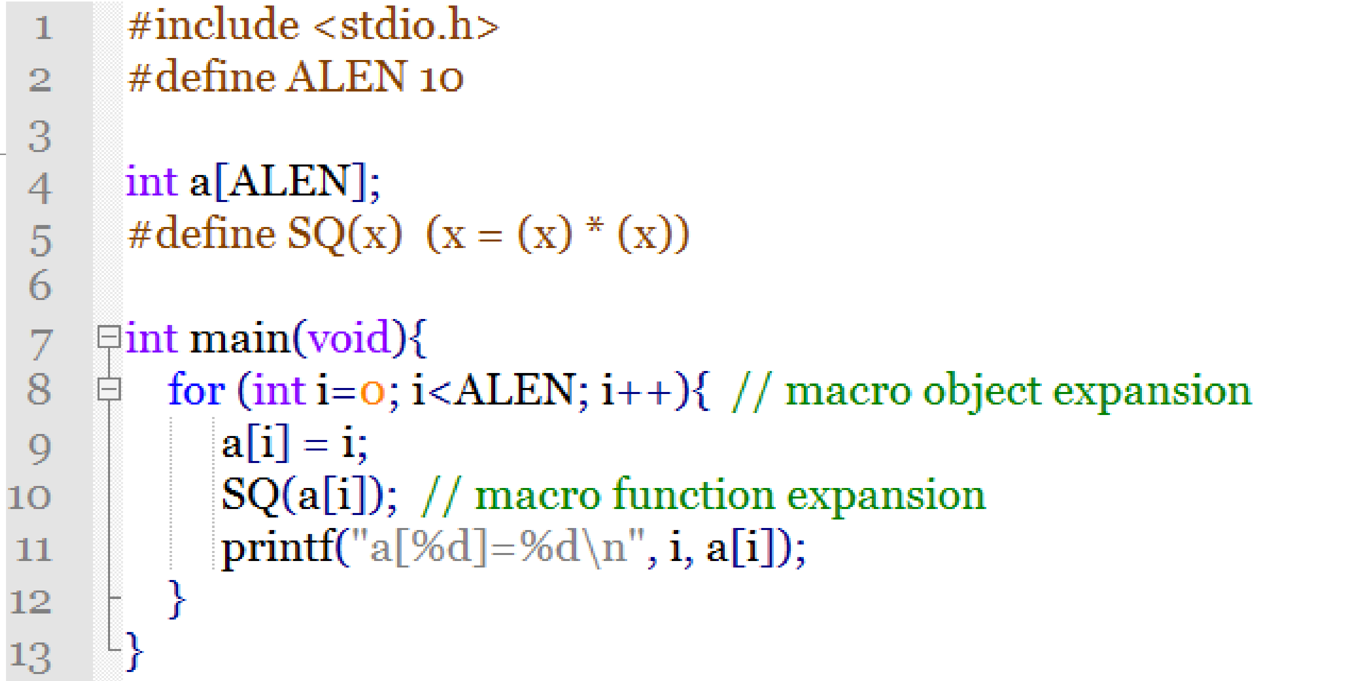
MIN(++firstnum, secondnum) ?

firstnum will be incremented twice. It is not functional call. It is macro exapansion.

Demo Program:

macros.c

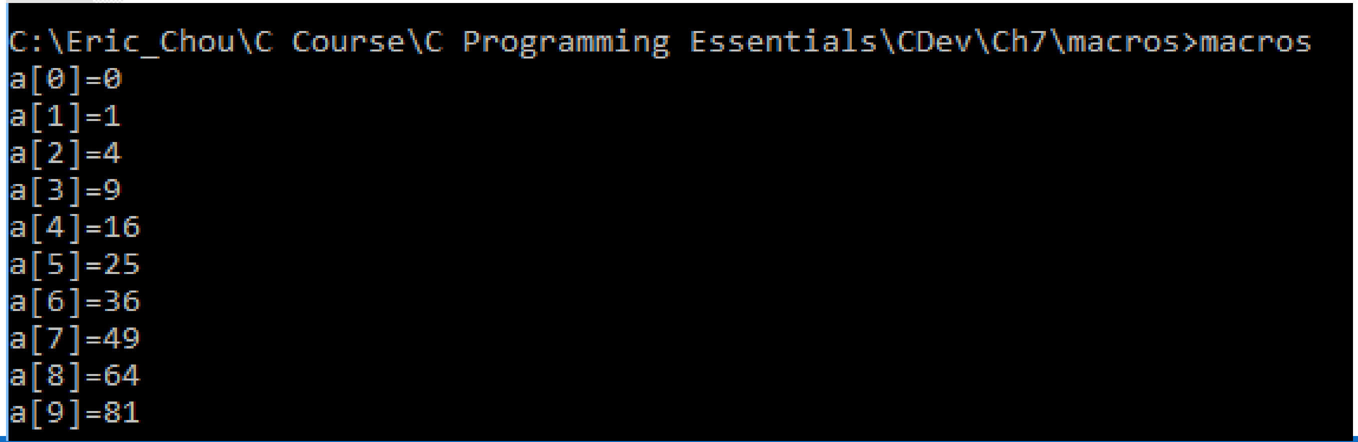
Go gcc!!!



```c
#include <stdio.h>
#define ALEN 10

int a[ALEN];
#define SQ(x)  (x = (x) * (x))

int main(void){
   for (int i=0; i<ALEN; i++){  // macro object expansion
      a[i] = i;
      SQ(a[i]);  // macro function expansion
      printf("a[%d]=%d\n", i, a[i]);
   }
}
```



```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch7\macros>macros
a[0]=0
a[1]=1
a[2]=4
a[3]=9
a[4]=16
a[5]=25
a[6]=36
a[7]=49
a[8]=64
a[9]=81
```

Getting Fancy with Macros

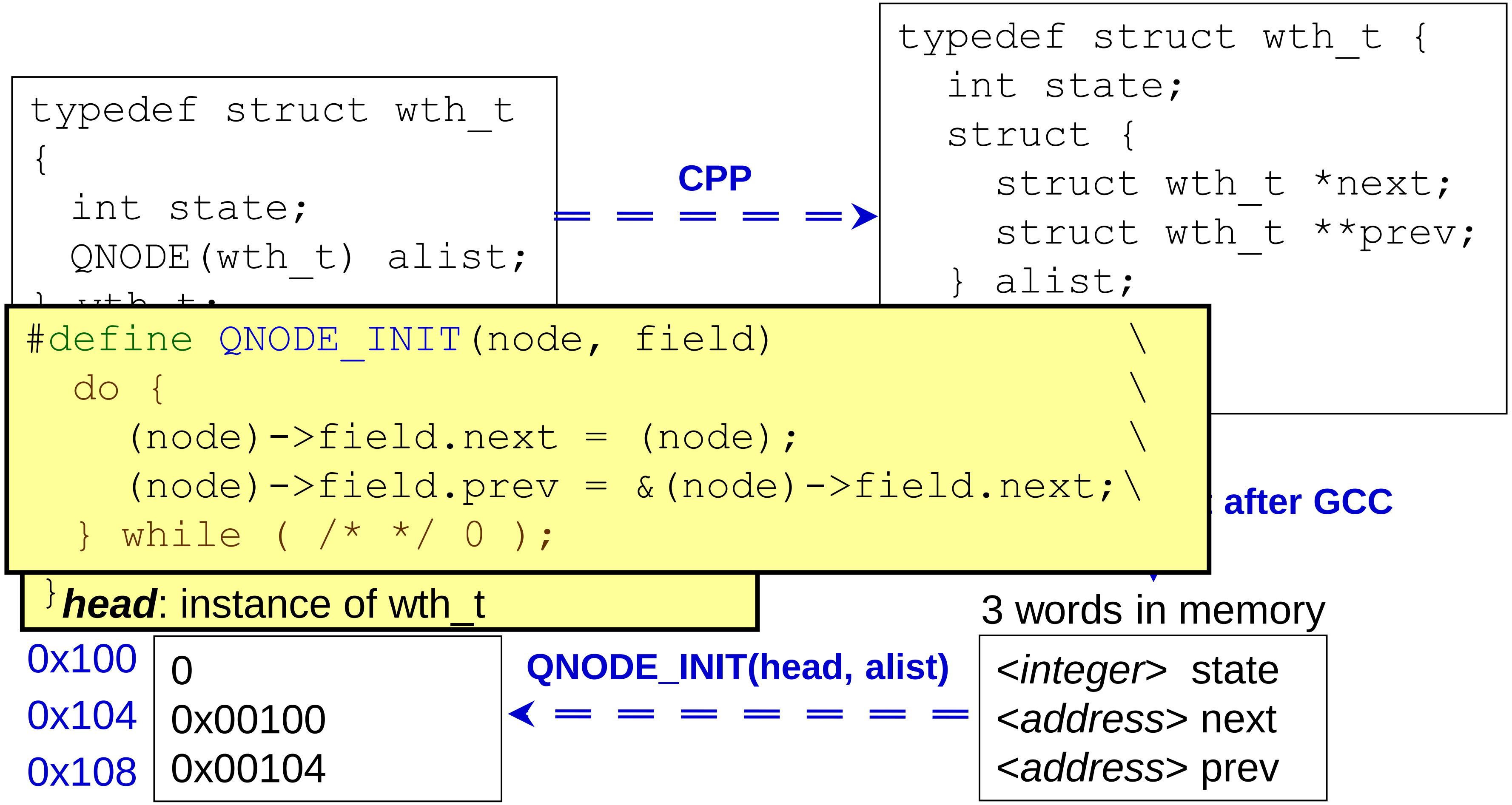
```
#define QNODE(type)    \
struct {               \
  struct type *next;   \
  struct type **prev;  \
}
#define QNODE_INIT(node, field)   \
  do {                            \
    (node)->field.next = (node);  \
    (node)->field.prev =          \
           &(node)->field.next;   \
  } while ( /* */ 0 );
#define QFIRST(head, field) \
        ((head)->field.next)
#define QNEXT(node, field) \
        ((node)->field.next)
#define QEMPTY(head, field) \
        ((head)->field.next == (head))
#define QFOREACH(head, var, field) \
  for ((var) = (head)->field.next; \
       (var) != (head);            \
       (var) = (var)->field.next)
```

```
#define QINSERT_BEFORE(loc, node, field) \
  do {                                 \
    *(loc)->field.prev = (node);       \
    (node)->field.prev =               \
           (loc)->field.prev;          \
    (loc)->field.prev  =               \
           &((node)->field.next);      \
    (node)->field.next = (loc);        \
  } while (/* */0)

#define QINSERT_AFTER(loc, node, field)         \
  do {                                        \
    ((loc)->field.next)->field.prev =         \
           &(node)->field.next;               \
    (node)->field.next = (loc)->field.next;   \
    (loc)->field.next = (node);               \
    (node)->field.prev = &(loc)->field.next;  \
  } while ( /* */ 0)

#define QREMOVE(node, field)                    \
  do {                                        \
    *((node)->field.prev) = (node)->field.next;\
    ((node)->field.next)->field.prev =        \
            (node)->field.prev;               \
    (node)->field.next = (node);              \
    (node)->field.prev = &((node)->field.next);\
  } while ( /* */ 0)
```

After Preprocessing and Compiling



```
typedef struct wth_t
{
  int state;
  QNODE(wth_t) alist;
} wth_t;
```

CPP

```
typedef struct wth_t {
  int state;
  struct {
    struct wth_t *next;
    struct wth_t **prev;
  } alist;
```

```
#define QNODE_INIT(node, field)                   \
  do {                                            \
    (node)->field.next = (node);                  \
    (node)->field.prev = &(node)->field.next;\
  } while ( /* */ 0 );
```

after GCC

}***head***: instance of wth_t

0x100	0
0x104	0x00100
0x108	0x00104

QNODE_INIT(head, alist)

3 words in memory

<integer> state
<address> next
<address> prev

Preprocessor: Macros

Using macros as functions, exercise caution:

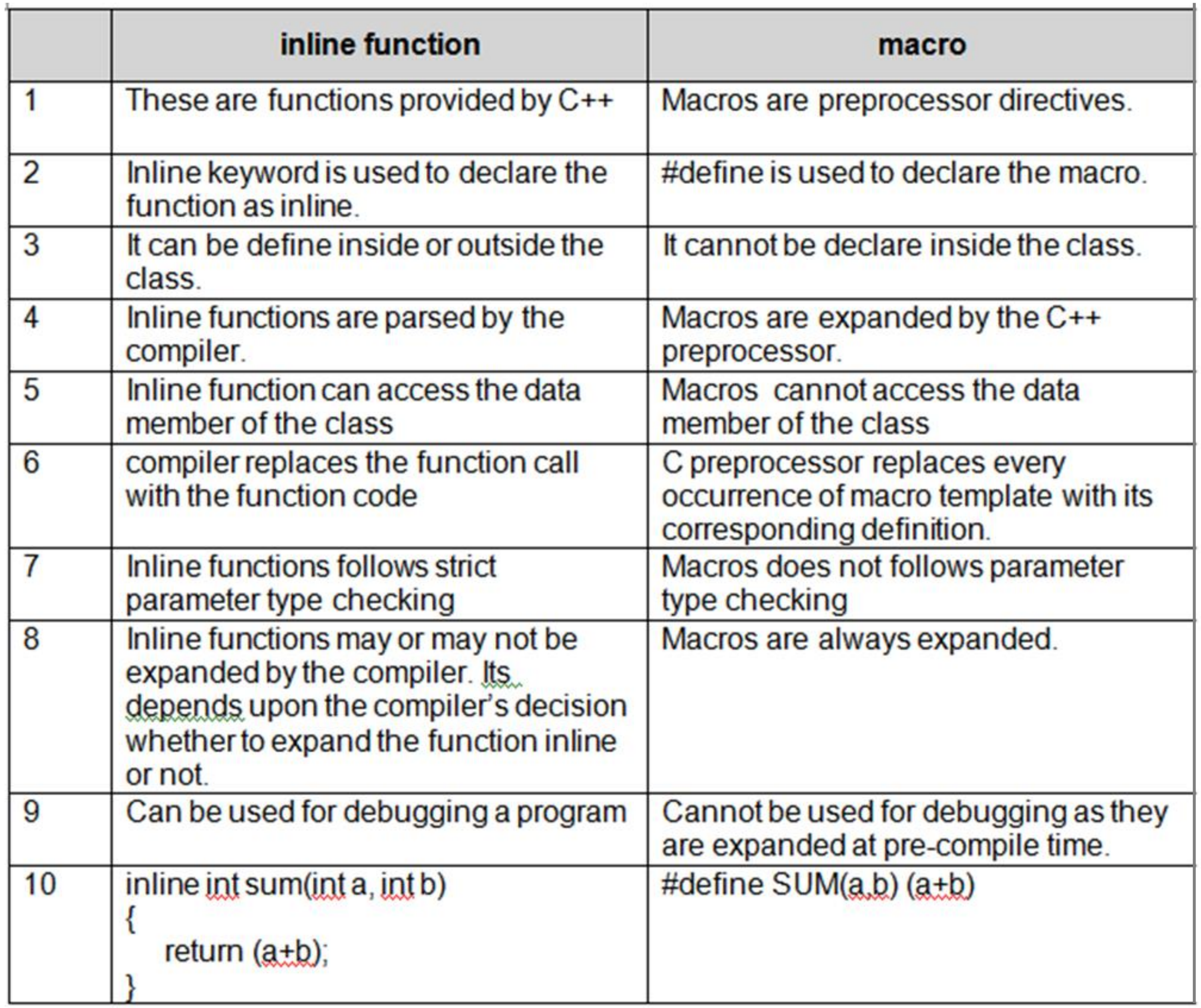
- flawed example: `#define mymult(a,b) a*b`
  - Source: `k = mymult(i-1, j+5);`
  - Post preprocessing: `k = i – 1 * j + 5;`
- better: `#define mymult(a,b) (a)*(b)`
  - Source: `k = mymult(i-1, j+5);`
  - Post preprocessing: `k = (i – 1)*(j + 5);`

Be careful of side effects, for example what if we did the following

- Macro: `#define mysq(a) (a)*(a)`
- flawed usage:
  - Source: `k = mysq(i++)`
  - Post preprocessing: `k = (i++)*(i++)`

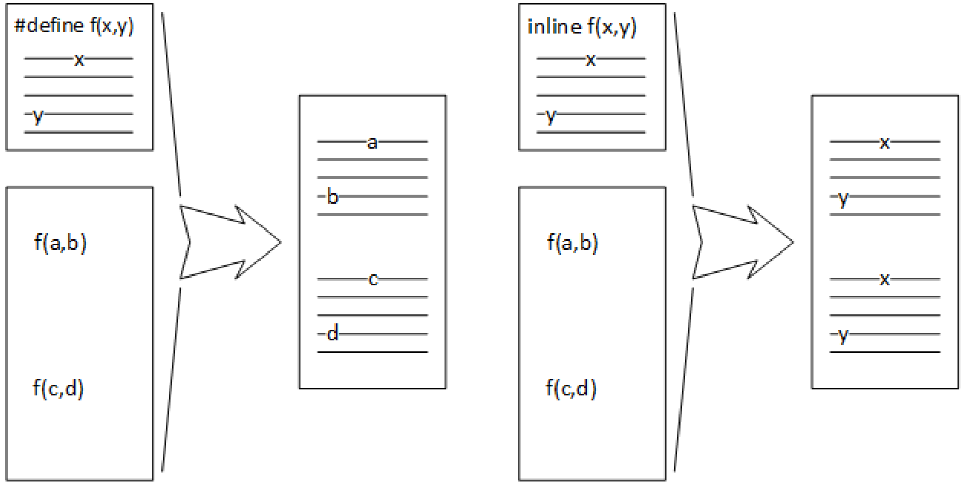
Alternative is to use inline’ed functions

- `inline int mysq(int a) {return a*a};`
- `mysq(i++)` works as expected in this case.



	inline function	macro
1	These are functions provided by C++	Macros are preprocessor directives.
2	Inline keyword is used to declare the function as inline.	#define is used to declare the macro.
3	It can be define inside or outside the class.	It cannot be declare inside the class.
4	Inline functions are parsed by the compiler.	Macros are expanded by the C++ preprocessor.
5	Inline function can access the data member of the class	Macros cannot access the data member of the class
6	compiler replaces the function call with the function code	C preprocessor replaces every occurrence of macro template with its corresponding definition.
7	Inline functions follows strict parameter type checking	Macros does not follows parameter type checking
8	Inline functions may or may not be expanded by the compiler. Its depends upon the compiler's decision whether to expand the function inline or not.	Macros are always expanded.
9	Can be used for debugging a program	Cannot be used for debugging as they are expanded at pre-compile time.

10	inline int sum(int a, int b)
	{
	    return (a+b);
	}
		#define SUM(a,b) (a+b)

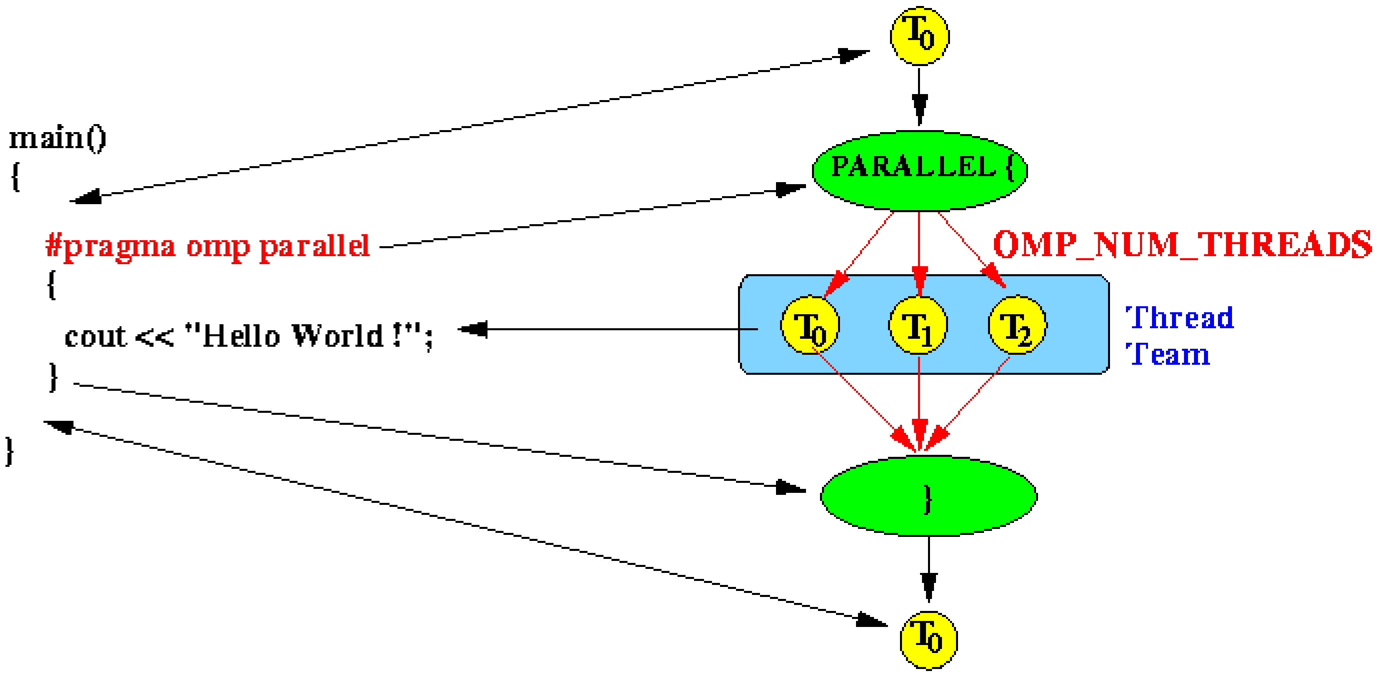


(a)

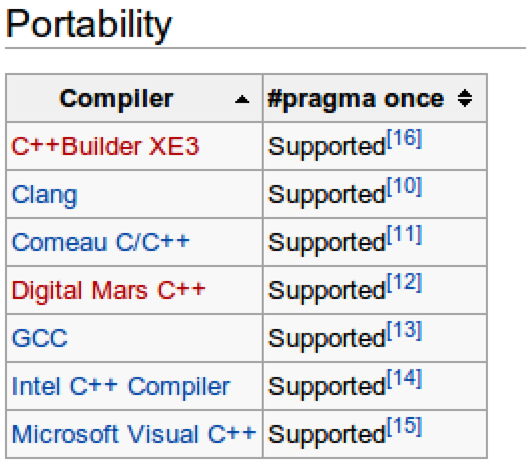
(b)

#pragma for expansion of compiler-dependent code (Super Macro)

#pragma – this directive is for inserting compiler-dependent commands into a file.



```
main()
{
   #pragma omp parallel
   {
    cout << "Hello World !";
   }
}
```



Portability

Compiler ▲	#pragma once ♦
C++Builder XE3	Supported[16]
Clang	Supported[10]
Comeau C/C++	Supported[11]
Digital Mars C++	Supported[12]
GCC	Supported[13]
Intel C++ Compiler	Supported[14]
Microsoft Visual C++	Supported[15]

LECTURE 3

#include for File Inclusion

#include

- **Specifies that the preprocessor should read in the contents of the specified file**
  - usually used to read in type definitions, prototypes, etc.
  - proceeds recursively
    - #includes in the included file are read in as well
- **Two forms:**
  - #include <*filename*>
    - searches for filename from a predefined list of directories
    - the list can be extended via "**gcc –I *dir***"
  - #include "*filename*"
- looks for *filename* specified as a relative or absolute path

Header files

- Usually define function prototypes, user defined types and global variables.

- Avoid including twice

```
int x;              /* included from myHeader.h */
int x;              /* included from myHeader.h */
```

- Standard header file header

```
#ifndef MyHeaderFile_H
#define MyHeaderFile_H
...     /* header file contents goes here  */
#endif
```

Source and Header files

Just as in C++, place related code within the same module (i.e. file).

Header files (`*.h`) export interface definitions

- function prototypes, data types, macros, inline functions and other common declarations

Do not place source code (i.e. definitions) in the header file with a few exceptions.

- inline'd code
- class definitions
- const definitions

C preprocessor (`cpp`) is used to insert common definitions into source files

There are other cool things you can do with the preprocessor

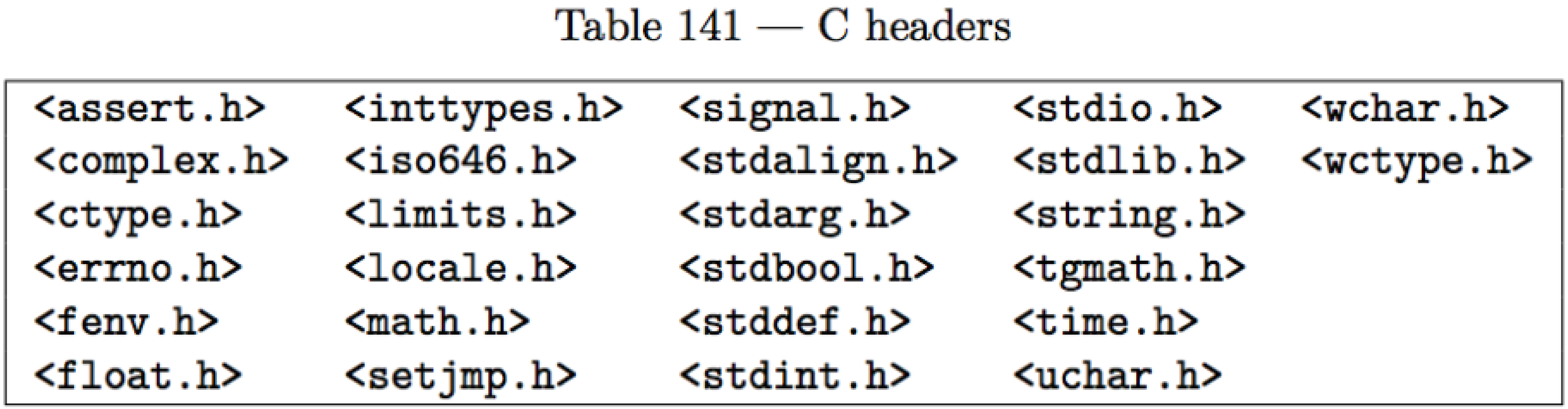


Table 141 — C headers

<assert.h>	<inttypes.h>	<signal.h>	<stdio.h>	<wchar.h>
<complex.h>	<iso646.h>	<stdalign.h>	<stdlib.h>	<wctype.h>
<ctype.h>	<limits.h>	<stdarg.h>	<string.h>	
<errno.h>	<locale.h>	<stdbool.h>	<tgmath.h>	
<fenv.h>	<math.h>	<stddef.h>	<time.h>	
<float.h>	<setjmp.h>	<stdint.h>	<uchar.h>	

Standard Library

at /usr/include or C:\Program Files (x86)\CodeBlocks\MinGW\include

C Standard Header Files you may want to use

Standard Headers you should know about:

- `stdio.h` – file and console (also a file) IO: *`perror, printf, open, close, read, write, scanf`*, etc.
- `stdlib.h` - common utility functions: *`malloc, calloc, strtol, atoi`*, etc
- `string.h` - string and byte manipulation: *`strlen, strcpy, strcat, memcpy, memset`*, etc.
- `ctype.h` – character types: *`isalnum, isprint, isupport, tolower,`* etc.
- `errno.h` – defines *`errno`* used for reporting system errors
- `math.h` – math functions: *`ceil, exp, floor, sqrt`*, etc.
- `signal.h` – signal handling facility: *`raise, signal`*, etc
- `stdint.h` – standard integer: *`intN_t, uintN_t`*, etc
- `time.h` – time related facility: *`asctime, clock, time_t,`* etc.

A Simple C Program

Create example file: `try.c`

Compile using gcc:
`gcc –o try try.c`

The standard C library *libc* is included automatically

Execute program `./try`

Note, I always specify an absolute path

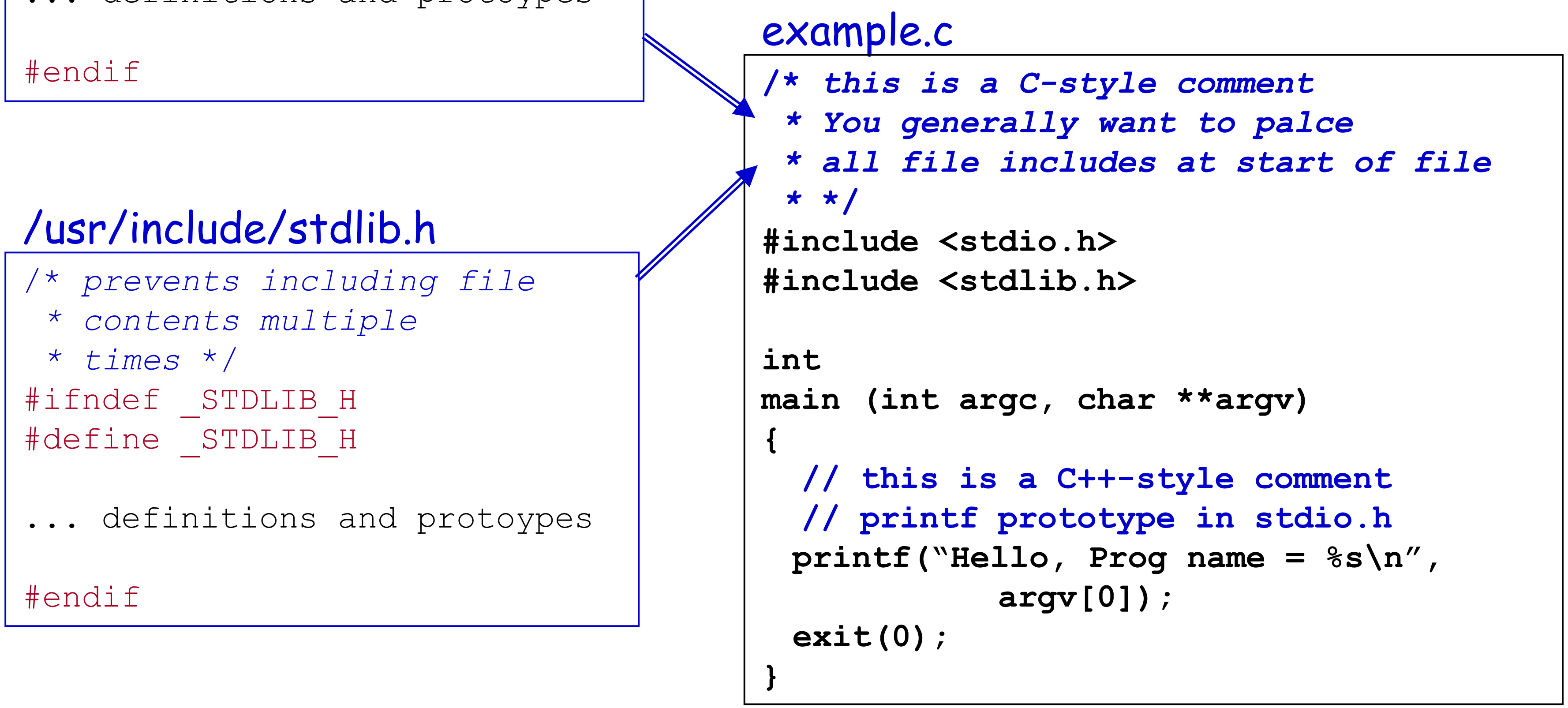
Normal termination:
`void` **`exit`**`(int status);`

- calls functions registered with `at exit()`
- flush output streams
- close all open streams
- return status value and control to host environment

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf(“Hello World\n”);
   exit(0);
}
```

#include directs the preprocessor to “include” the contents of the file at this point in the source file.
#define directs preprocessor to define macros.



/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

example.c

```
/* this is a C-style comment
 * You generally want to palce
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
 printf(“Hello, Prog name = %s\n”,
            argv[0]);
 exit(0);
}
```

Demo Program:

include package

Go gcc!!!

User Defined Library

Build Application

build.bat

```
gcc -I ./I -c test.c
gcc test.o ./I/a.o -o test
```

test.c

```
#include <stdio.h>
#include <a.h>
extern int x;
int main(void){
  f();
  x++;
  printf("x=%d in f()\n", x);
  return 0;
}
```

Build I Library

buildI.bat

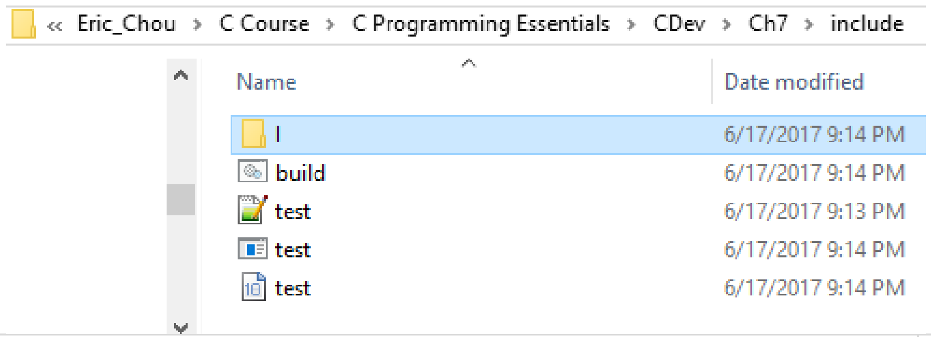
```
gcc -c a.c
```

a.h

```
#ifndef INX
#define INX
extern int f();
#endif
```

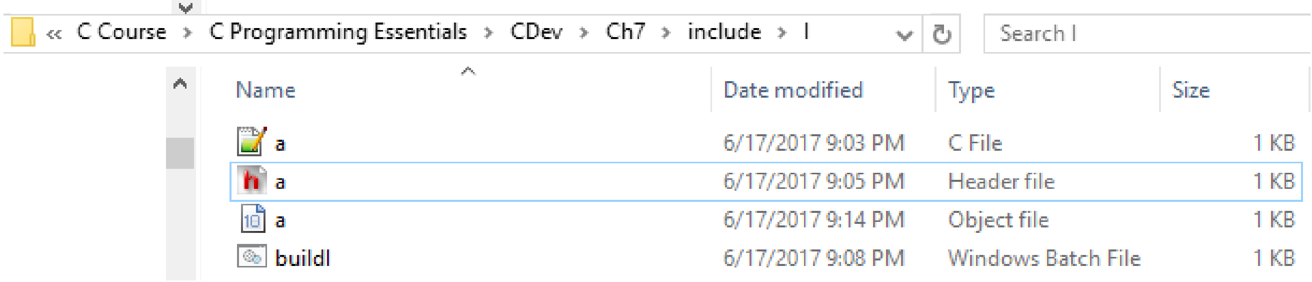
a.c

```
#include <stdio.h>
int x;
int f(){
 x=0;
 x++;
 printf("In a.c f() \n");
 return x;
}
```



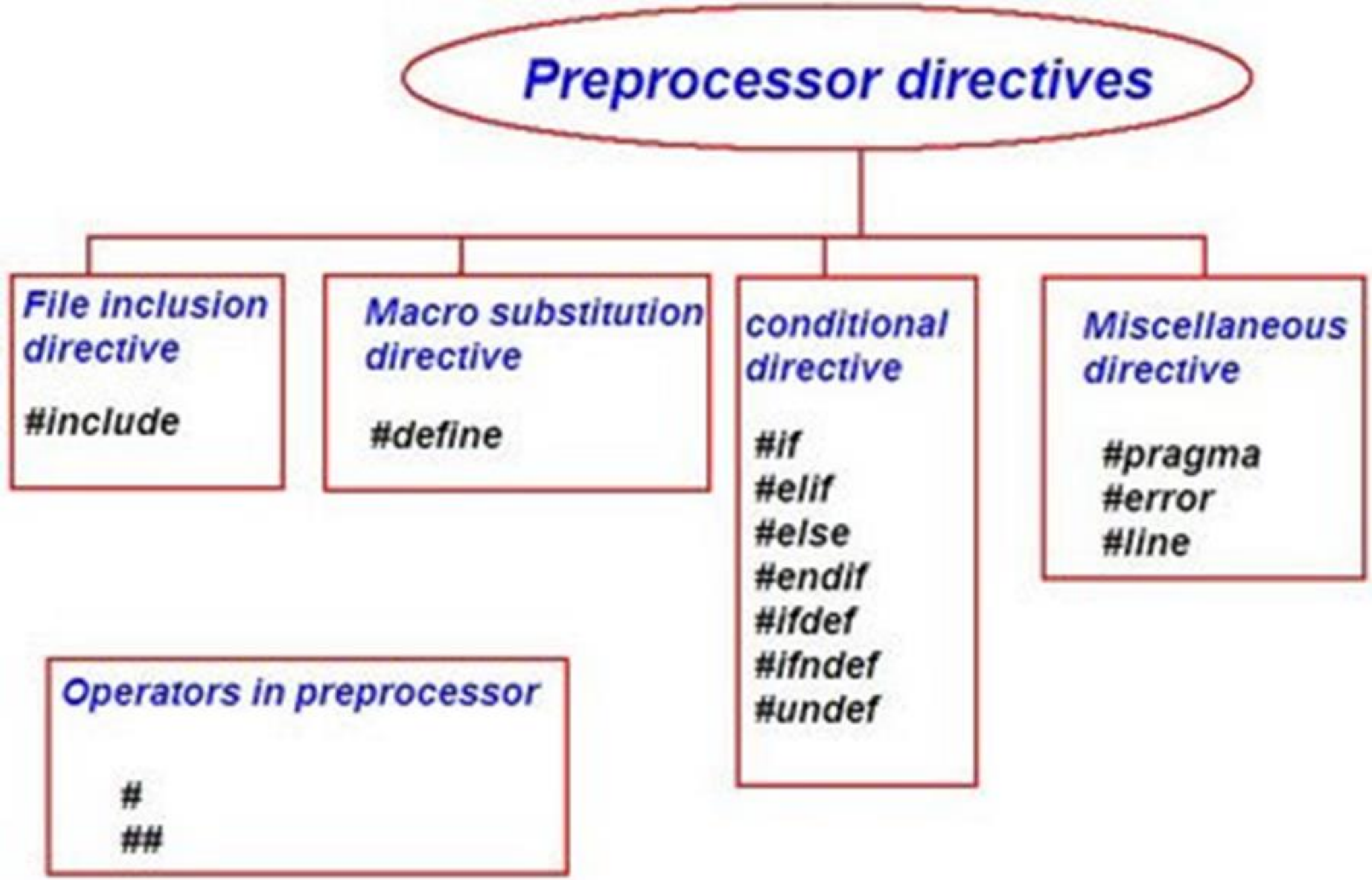
« Eric_Chou › C Course › C Programming Essentials › CDev › Ch7 › include

Name	Date modified
I	6/17/2017 9:14 PM
build	6/17/2017 9:14 PM
test	6/17/2017 9:13 PM
test	6/17/2017 9:14 PM
test	6/17/2017 9:14 PM



« C Course › C Programming Essentials › CDev › Ch7 › include › I

Name	Date modified	Type	Size
a	6/17/2017 9:03 PM	C File	1 KB
a	6/17/2017 9:05 PM	Header file	1 KB
a	6/17/2017 9:14 PM	Object file	1 KB
buildI	6/17/2017 9:08 PM	Windows Batch File	1 KB



Preprocessor directives

File inclusion directive

#include

Macro substitution directive

#define

conditional directive

#if
#elif
#else
#endif
#ifdef
#ifndef
#undef

Miscellaneous directive

#pragma
#error
#line

Operators in preprocessor

#
##

The Preprocessor

The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

Commands begin with a ‘#’. Abbreviated list:

`#define` : defines a macro

`#undef` : removes a macro definition

`#include` : insert text from file

`#if` : conditional based on value of expression

`#ifdef` : conditional based on whether macro defined

`#ifndef` : conditional based on whether macro is not defined

`#else` : alternative

`#elif` : conditional alternative

`defined()` : preprocessor function: 1 if name defined, else 0

```
#if defined(__NetBSD__)
```

LECTURE 4

Conditional Compilation

Preprocessor: Conditional Compilation

- Its generally better to use inline'ed functions
- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts

```
#define DEFAULT_SAMPLES   100
#ifdef __linux static inline int64_t gettime(void) {...}
#elif defined(sun) static inline int64_t gettime(void) \
                   {return (int64_t)gethrtime()}
#else static inline int64_t gettime(void)   \
        {... gettimeofday()...}
#endif
```

Conditional Compilation

The #if, #ifdef, #ifndef, #else, #elif and #endif directives can be used for conditional compilation.

Example 1 （Conditional Compilation on Different Platform）

```
#define __WINDOWS__
#ifdef __WINDOWS__
  #include <windows.h>
#else
  #include <unistd.h>
#endif
```

Conditional Compilation

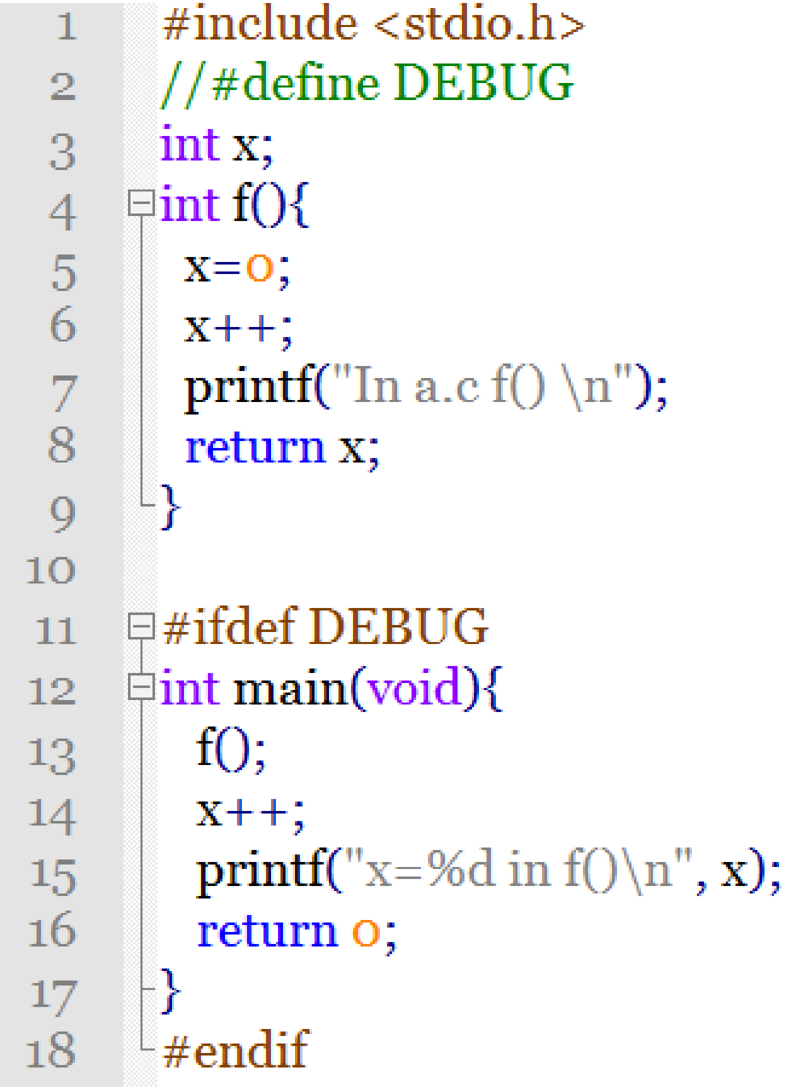
Example 2 (Conditional Compilation for Debug Mode)

```
#define DEBUG
#ifdef DEBUG
  printf("trace message");
#endif
```

Demo Program:

a.c

Go gcc!!!



```
#include <stdio.h>
//#define DEBUG
int x;
int f(){
  x=0;
  x++;
  printf("In a.c f() \n");
  return x;
}

#ifdef DEBUG
int main(void){
   f();
   x++;
   printf("x=%d in f()\n", x);
   return 0;
}
#endif
```

With DEBUG:

1. Under I directory to use buildDEBUG.
2. Run a program

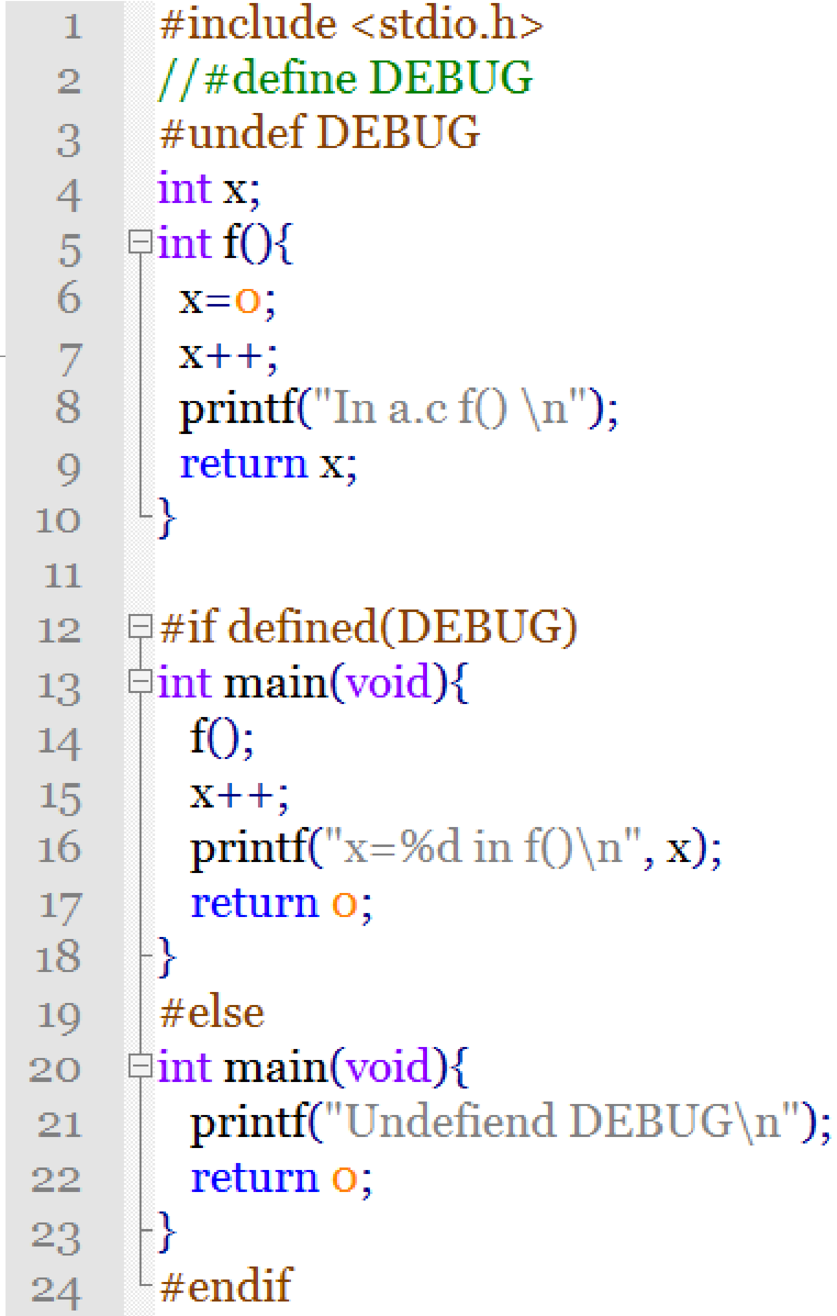
Without DEBUG:

1. Under I directory, use buildI.bat like include project
2. Under debugMode directory, use build.bat
3. Run test program

Demo Program:

b.c

Go gcc!!!



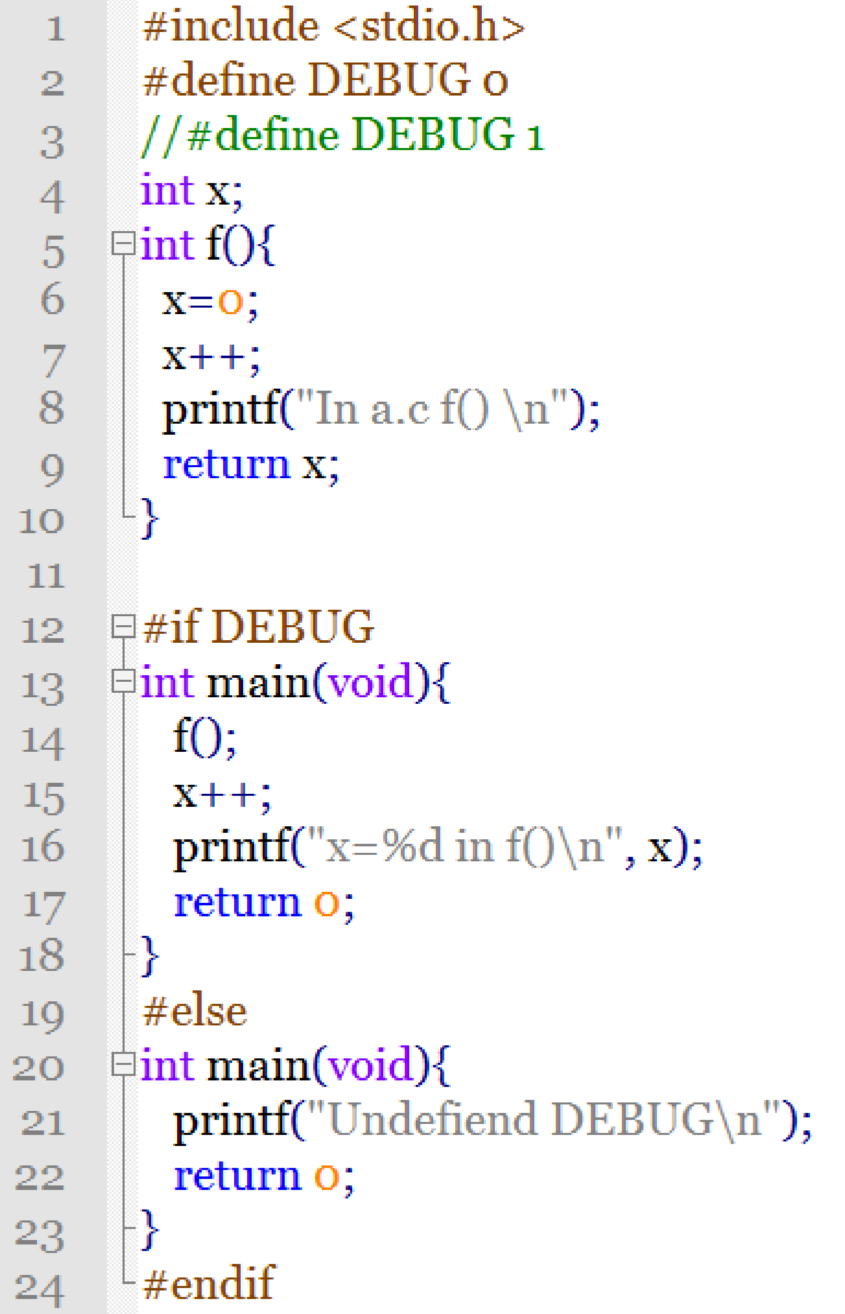
```
#include <stdio.h>
//#define DEBUG
#undef DEBUG
int x;
int f(){
 x=0;
 x++;
 printf("In a.c f() \n");
 return x;
}

#if defined(DEBUG)
int main(void){
  f();
  x++;
  printf("x=%d in f()\n", x);
  return 0;
}
#else
int main(void){
  printf("Undefiend DEBUG\n");
  return 0;
}
#endif
```

Demo Program:

d.c

Go gcc!!!



```c
#include <stdio.h>
#define DEBUG 0
//#define DEBUG 1
int x;
int f(){
  x=0;
  x++;
  printf("In a.c f() \n");
  return x;
}

#if DEBUG
int main(void){
  f();
  x++;
  printf("x=%d in f()\n", x);
  return 0;
}
#else
int main(void){
  printf("Undefiend DEBUG\n");
  return 0;
}
#endif
```

Multi File Programs

Why?

- As the file grows, compilation time tends to grow, and for each little change, the whole program has to be re-compiled.
- It is very hard, if not impossible, that several people will work on the same project together in this manner.
- Managing your code becomes harder. Backing out erroneous changes becomes nearly impossible.

Solution

- split the source code into multiple files, each containing a set of closely-related functions

Multi File Programs

Option 1

- Say Program broken up into main.c A.c and B.c
- If we define a function (or a variable) in one file, and try to access them from a second file, declare them as external symbols in that second file. This is done using the C "extern" keyword.
- Compile as:
  gcc main.c A.c B.c -o prog

Multi File Programs

Option 2

- Use header files to define variables and function prototypes
- Use #ifndef _*headerfile name* #define _ *headerfile name* and #endif to encapsulate the code in each Header file
- Compile only the modified files as:

```
gcc -c main.cc
gcc -c A.c
gcc -c B.c
```

And then link as

```
gcc main.o A.o B.o -o prog
```

Multi File Programs

Which is better Option 1 or Option 2?

- Re-usability
- Debuggability
- Convenience
- Project scale