Preprocessing

Advanced C

Preprocessor

- One of the step performed before compilation
- Is a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- Instructions given to preprocessor are called preprocessor directives and they begin with “#” symbol
- Few advantages of using preprocessor directives would be,
  - Easy Development
  - Readability
  - Portability

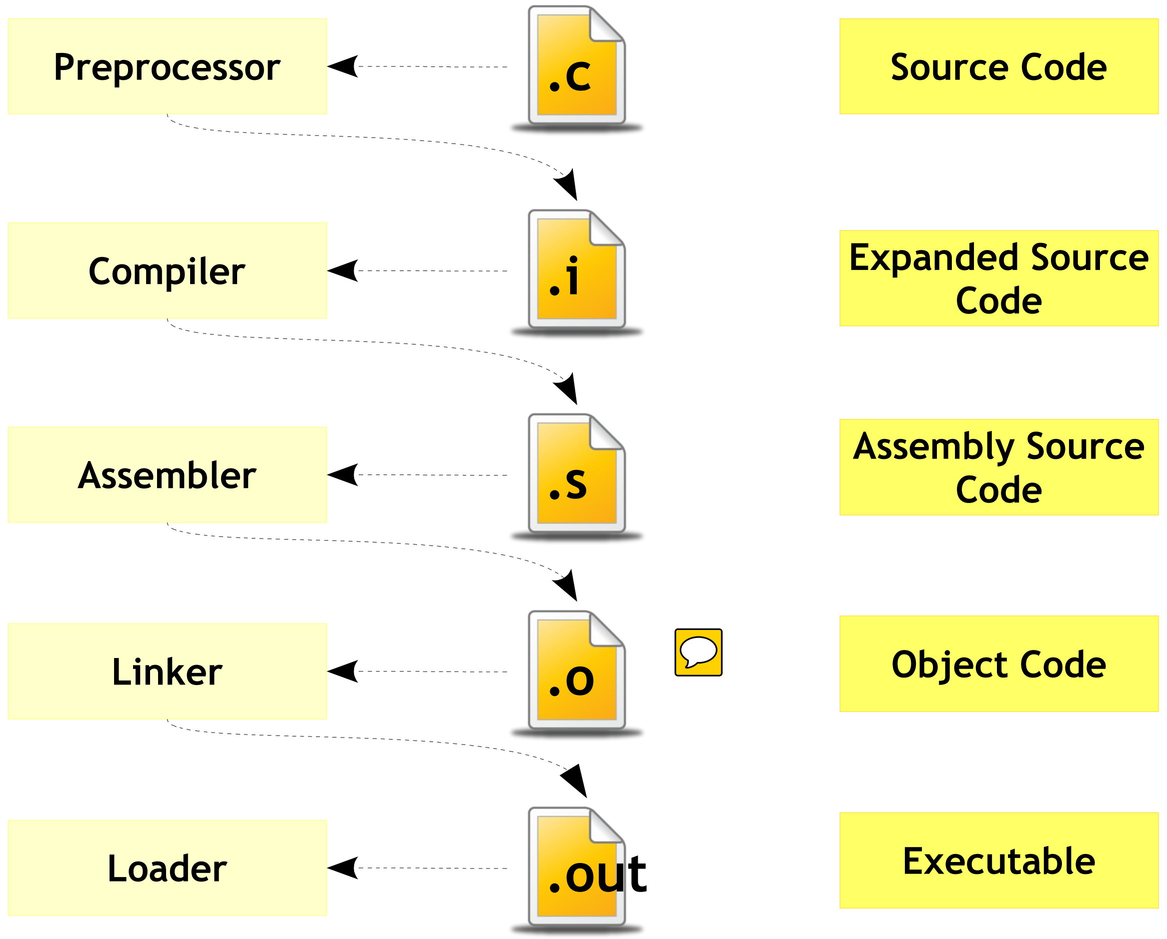
Advanced C

Preprocessor – Compilation Stages

- Before we proceed with preprocessor directive let's try to understand the stages involved in compilation
- Some major steps involved in compilation are
  - Preprocessing (Textual replacement)
  - Compilation (Syntax and Semantic rules checking)
  - Assembly (Generate object file(s))
  - Linking (Resolve linkages)
- The next slide provide the flow of these stages

Advanced C

Preprocessor – Compilation Stages



```
user@user:~] gcc -E file.c
```

```
user@user:~] gcc -S file.c
```

```
user@user:~] gcc -c file.c
```

```
user@user:~] gcc file.c -o file.out
```

```
user@user:~]gcc -save-temps file.c #would generate all intermediate files
```

Advanced C

Preprocessor – Compilation Steps

.c

```
user@user:~] gcc -E file.c -o file.i
```

.i

```
user@user:~] gcc -S file.i -o file.s
```

.s

```
user@user:~] as file.s -o file.o
```

Bit complex step

.o

```
user@user:~] ld file.o -o file.out <LIBRARY PATH>
```

.out

```
user@user:~]./file.out
```

Advanced C

Preprocessor – Compilation Steps

.c

```
user@user:~] gcc -E file.c -o file.i
```

.i

```
user@user:~] gcc -S file.i -o file.s
```

.s

```
user@user:~] gcc file.s -o file.out
```

.out

```
user@user:~]./file.out
```

Advanced C

Preprocessor – Directives

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #else
- #endif
- #if
- #else
- #elif
- #error
- #warning
- #line
- #pragma
- #
- ##

Advanced C

Preprocessor – Header Files

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- Has to be included using C preprocessing directive ‘**#include**'
- Header files serve two purposes.
  - Declare the interfaces to parts of the operating system by supplying the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program.

Advanced C

Preprocessor – Header Files vs Source Files

VS

- Declarations
- Sharable/reusable
  - #defines
  - Datatypes
- Used by more than 1 file

- Function and variable definitions
- Non sharable/reusable
  - #defines
  - Datatypes

Advanced C

Preprocessor – Header Files - Syntax

Syntax

```
#include <file.h>
```

- System header files
- It searches for a file named ***file*** in a standard list of system directories

Syntax

```
#include “file.h”
```

- Local (your) header files
- It searches for a file named ***file*** first in the directory containing the current file, then in the quote directories and then the same directories used for <file>

Advanced C

Preprocessor – Header Files - Operation

file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

file2.h

```
char *test(void);
```

file1.c

```
int num;

#include "file2.h"

int main()
{
    puts(test());

    return 0;
}
```

```
int num;

char *test(void);

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E file1.c file2.c
```

Advanced C

Preprocessor – Header Files – Search Path

file2.c

```c
char *test(void)
{
    static char *str = “Hello”;

    return str;
}
```

file2.h

```c
char *test(void);
```

file1.c

```c
int num;

#include “file2.h”

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E file1.c file2.c
```

Advanced C

Preprocessor - Header Files - Search Path

file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

file1.c

```
int num;

#include <file2.h>

int main()
{
    puts(test());

    return 0;
}
```

file2.h

```
char *test(void);
```

Compile as

```
user@user:~] gcc -E file1.c file2.c -I .
```

Advanced C

Preprocessor – Header Files – Search Path

- On a normal Unix system GCC by default will look for headers requested with #include <file> in:
  - /usr/local/include
  - libdir/gcc/target/version/include
  - /usr/target/include
  - /usr/include
- You can add to this list with the -I <*dir*> command-line option

Get it as

```
user@user:~] gcc -print-prog-name=cc1 -v #would search the path info
```

Advanced C

Preprocessor – Macro – Object-Like

- An object-like macro is a simple identifier which will be replaced by a code fragment
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants

Syntax

```
#define SYMBOLIC_NAME      CONSTANTS
```

Example

```
#define BUFFER_SIZE        1024
```

Advanced C

Preprocessor - Macro - Object-Like

main.c

```c
#define SIZE    1024
#define MSG     "Enter a string"

int main()
{
    char array[SIZE];

    printf("%s\n", MSG);
    fgets(array, SIZE, stdin);

    printf("%s\n", array);

    return 0;
}
```

Compile as

```
user@user:~] gcc -E main.c -o main.i
```

main.i

```c
# 1 "main.c"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"



int main()
{
 char array[1024];

 printf("%s\n", "Enter a string");
 fgets(array, 1024, stdin);

 printf("%s\n", array);

 return 0;
}
```

ΣMERTXE

Advanced C

Preprocessor – Macro – Standard Predefined

- Several object-like macros are predefined; you use them without supplying their definitions.
- Standard are specified by the relevant language standards, so they are available with all compilers that implement those standards

Example

```
#include <stdio.h>

int main()
{
    printf(“Program: \“%s\” ”, __FILE__);
    printf(“was compiled on %s at %s. ”, __DATE__, __TIME__);
    printf(“This print is from Function: \”%s\”, __func__);
    printf(“at line %d\n”, __LINE__);

    return 0;
}
```

Advanced C

Preprocessor – Macro – Arguments

- Function-like macros can take arguments, just like true functions
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like

Syntax

```
#define MACRO(ARGUMENT(S))             (EXPRESSION WITH ARGUMENT(S))
```

EMERTXE

Advanced C

Preprocessor - Macro - Arguments

main.c

```
#include <stdio.h>

#define SET_BIT(num, pos)     num | (1 << pos)

int main()
{
    printf("%d\n", 2 * SET_BIT(0, 2));

    return 0;
}
```

main.i

```
int main()
{
 printf("%d\n", 2 * 0 | (1 << 2));

 return 0;
}
```

ΣMERTXE

Advanced C

Preprocessor - Macro - Arguments

main.c

```c
#include <stdio.h>

#define SET_BIT(num, pos)      (num | (1 << pos))

int main()
{
    printf("%d\n", 2 * SET_BIT(0, 2));

    return 0;
}
```

main.i

```c
int main()
{
 printf("%d\n", 2 * (0 | (1 << 2)));

 return 0;
}
```

Advanced C

Preprocessor – Macro – Arguments - DIY

- WAM to find the sum of two nos
- Write macros to get, set and clear Nth bit in an integer
- WAM to swap a nibble in a byte

Advanced C

Preprocessor – Macro – Multiple Lines

- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
- This could be done to achieve readability
- When the macro is expanded, however, it will all come out on one line

Advanced C

Preprocessor - Macro - Multiple Lines

main.c

```
#include <stdio.h>

#define SWAP(a, b)              \
    int temp = a;               \
    a = b;                      \
    b = temp;                   \

int main()
{
    int n1 = 10, n2= 20;

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    return 0;
}
```

main.i

```
int main()
{
 int n1 = 10, n2= 20;

 int temp = n1;n1 = n2;n2 = temp;
 printf("%d %d\n", n1, n2);

 int temp = n1;n1 = n2;n2 = temp;
 printf("%d %d\n", n1, n2);

 return 0;
}
```

Advanced C

Preprocessor – Macro – Multiple Lines

main.c

```c
#include <stdio.h>

#define SWAP(a, b)                      \
{                                       \
    int temp = a;                       \
    a = b;                              \
    b = temp;                           \
}

int main()
{
    int n1 = 10, n2= 20;

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    return 0;
}
```

main.i

```c
int main()
{
 int n1 = 10, n2= 20;

 {int temp = n1;n1 = n2;n2 = temp;}
 printf("%d %d\n", n1, n2);

 {int temp = n1;n1 = n2;n2 = temp;}
 printf("%d %d\n", n1, n2);

 return 0;
}
```

EMERTXE

Advanced C

Preprocessor – Macro – Multiple Lines - DIY

- WAM to swap any two numbers of basic type using temporary variable

Advanced C

Preprocessor – Macro vs Function

Function

```c
#include <stdio.h>

int set_bt(int n, int p)
{
    return (n | (1 << p));
}

int main()
{
    printf("%d\n", 2 * set_bt(0, 2));
    printf("%d\n", 4 * set_bt(0, 2));

    return 0;
}
```

- Lose time optimization on context switching
- Stack creation overhead
- Space optimized on repeated call
- Compiled by the compiler
- Not recommended for smaller operation

Macro

```c
#include <stdio.h>

#define set_bt(n, p)    (n | (1 << p))




int main()
{
    printf("%d\n", 2 * set_bt(0, 2));
    printf("%d\n", 4 * set_bt(0, 2));

    return 0;
}
```

- Good time optimization since no context switching
- No stack creation overhead
- Time optimized on repeated call
- Preprocessed by the compiler
- Recommended for smaller operation

Advanced C

Preprocessor – Macro – Stringification

Example

```c
#include <stdio.h>

#define WARN_IF(EXP)                                          \
do                                                            \
{                                                             \
    x--;                                                      \
    if (EXP)                                                  \
    {                                                         \
        fprintf(stderr, "Warning: " #EXP "\n");               \
    }                                                         \
} while (x);

int main()
{
    int x = 5;

    WARN_IF(x == 0);

    return 0;
}
```

- You can convert a macro argument into a string constant by adding #

Advanced C

Preprocessor – Conditional Compilation

- A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler
- Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator
- A conditional in the C preprocessor resembles in some ways an if statement in C with the only difference being it happens in compile time
- Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Advanced C

Preprocessor – Conditional Compilation

- There are three general reasons to use a conditional.
  - A program may need to use different code depending on the machine or operating system it is to run on
  - You may want to be able to compile the same source file into two different programs, like one for debug and other as final
  - A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference

Advanced C

Preprocessor – Header Files – Once-Only

- If a header file happens to be included twice, the compiler will process its contents twice causing an error
- E.g. when the compiler sees the same structure definition twice
- This can be avoided like

Syntax

```
#ifndef NAME
#define NAME

/* The entire file is protected */

#endif
```

Advanced C

Preprocessor - Header Files - Once-Only

main.c

```
#include "main.h"
#include "main.h"

int main()
{
    struct UserInfo p = {420, "Tingu"};

    return 0;
}
```

main.h

```
struct UserInfo
{
    int id;
    char name[30];
};
```

- Note that the main.h is included 2 times which would lead to redefinition of the structure UserInfo

Advanced C

Preprocessor - Header Files - Once-Only

main.c

```c
#include “main.h”
#include “main.h”

int main()
{
    struct UserInfo p = {420, “Tingu”};

    return 0;
}
```

main.h

```c
#ifndef MAIN_H
#define MAIN_H

struct UserInfo
{
    int id;
    char name[30];
};

#endif
```

- The multiple inclusion is protected by the **#ifndef** preprocessor directive

Advanced C

Preprocessor – Header Files – Once-Only

main.c

```
#include "main.h"
#include "main.h"

int main()
{
    struct UserInfo p = {420, "Tingu"};

    return 0;
}
```

main.h

```
#pragma once

struct UserInfo
{
    int id;
    char name[30];
};
```

- The other way to do this would be **#pragma once** directive
- This is not portable

Advanced C

Preprocessor – Conditional Compilation - ifdef

Syntax

```
#ifdef MACRO

/* Controlled Text */

#endif
```

Example

```
#include <stdio.h>

#define METHOD1

int main()
{
#ifdef METHOD1
    puts("Hello World");
#else
    printf("Hello World");
#endif

    return 0;
}
```

Advanced C

Preprocessor – Conditional Compilation - ifndef

Syntax

```
#ifndef MACRO

/* Controlled Text */

#endif
```

Example

```
#include <stdio.h>

#undef METHOD1

int main()
{
#ifndef METHOD1
    puts("Hello World");
#else
    printf("Hello World");
#endif

    return 0;
}
```

ΣMERTXE

Advanced C

Preprocessor – Conditional Compilation - defined

Syntax

```
#if defined condition

/* Controlled Text */

#endif
```

Example

```
#include <stdio.h>

#define METHOD1

int main()
{
#if defined (METHOD1)
    puts("Hello World");
#endif
#if defined (METHOD2)
    printf("Hello World");
#endif
#if defined (METHOD1) && defined (METHOD2)
    puts("Hello World");
    printf("Hello World");
#endif

    return 0;
}
```

Advanced C

Preprocessor – Conditional Compilation - if

Syntax

```
#if expression

/* Controlled Text */

#endif
```

Example

```
#include <stdio.h>

#define METHOD 1

int main()
{
#if METHOD == 1
    puts("Hello World");
#endif
#if METHOD == 2
    printf("Hello World");
#endif

    return 0;
}
```

Advanced C

Preprocessor – Conditional Compilation - else

Syntax

```
#if expression

/* Controlled Text if true */

#else

/* Controlled Text if false */

#endif
```

Example

```
#include <stdio.h>

#define METHOD

int main()
{
#if METHOD == 1
    puts("Hello World");
#else
    printf("Hello World");
#endif

    return 0;
}
```

Advanced C

Preprocessor – Conditional Compilation - elif

Syntax

```
#if expression1

/* Controlled Text*/

#elif  expression2

/* Controlled Text */

#else

/* Controlled Text */

#endif
```

Example

```
#include <stdio.h>

#define METHOD  1

int main()
{
    char msg[] = "Hello World";

#if METHOD == 1
    puts(msg);
#elif METHOD == 2
    printf("%s\n", msg);
#else
    int i;
    for (i = 0; i < 12; i++)
    {
        putchar(msg[i]);
    }
#endif

    return 0;
}
```

Advanced C

Preprocessor - Cond... Com... - CL Option

main.c

```c
#include <stdio.h>

int main()
{
    int x = 10, y = 20;

#ifdef SPACE_OPTIMIZED
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
    printf("Selected Space Optimization\n");
#else
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("Selected Time Optimization\n");
#endif

    return 0;
}
```

Compile as

```
user@user:~] gcc main.c -D SPACE_OPTIMIZED
```

ΣMERTXE

Advanced C

Preprocessor – Cond... Com... – Deleted Code

Syntax

```
#if 0

/* Deleted code while compiling */
/* Can be used for nested code comments */
/* Avoid for general comments */
/* Don't write lines like these!! with '

#endif
```

ΣMERTXE

Advanced C

Preprocessor – Diagnostic

- The directive **#error** causes the preprocessor to report a fatal error. The tokens forming the rest of the line following **#error** are used as the error message
- The directive **#warning** is like **#error**, but causes the preprocessor to issue a warning and continue preprocessing. The tokens following **#warning** are used as the warning message

Advanced C

Preprocessor – Diagnostic - #warning

main.c

```
#include <stdio.h>

#if defined DEBUG_PRINT
#warning “Debug print enabled”
#endif

int main()
{
    int sum, num1, num2;

    printf(“Enter 2 numbers: ”);
    scanf(“%d %d”, &num1, &num2);

#ifdef DEBUG_PRINT
    printf(“The entered values are %d %d\n”, num1, num2);
#endif

    sum = num1 + num2;
    printf(“The sum is %d\n”, sum);

    return 0;
}
```

Advanced C

Preprocessor – Diagnostic - #error

main.c

```
#include <stdio.h>

#if defined (STATIC) || defined (DYNAMIC)
#define SIZE    100
#else
#error "Memory not allocated!! Use -D STATIC or DYNAMIC while compiling"
#endif

int main()
{
#if defined STATIC
    char buffer[SIZE];
#elif defined DYNAMIC
    char *buffer = malloc(SIZE * sizeof(char));
#endif

#if defined (STATIC) || defined (DYNAMIC)
    fgets(buffer, SIZE, stdin);
    printf("%s\n", buffer);
#endif

    return 0;
}
```

Advanced C

Preprocessor – Diagnostic - #line

- Also known as preprocessor line control directive
- #line directive can be used to alter the line number and filename
- The line number will start from the set value, from the #line is encountered with the provided name

Example

```
#include <stdio.h>

int main()
{
#line 100 "project tuntun"
    printf("This is from file %s at line %d \n", __FILE__, __LINE__);

    return 0;
}
```