Compilation Process in C

Preprocessing, Compilation, Assembly, and Linking

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Program 1: Basic Preprocessor Macros

```c
#include <stdio.h>
#define PI 3.14159
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define DEBUG
#ifdef DEBUG
  #define LOG(msg) printf("DEBUG: %s\n", msg)
#else
  #define LOG(msg)
#endif
int main() {
  printf("PI = %.5f\n", PI);
  printf("SQUARE(5) = %d\n", SQUARE(5));
  printf("MAX(10, 20) = %d\n", MAX(10, 20));
  LOG("Program started");
  int r = 3;
  double area = PI * SQUARE(r);
  printf("Area = %.2f\n", area);
  LOG("Program ended");
  return 0;
}
```

```
PI = 3.14159
SQUARE(5) = 25
MAX(10, 20) = 20
DEBUG: Program started
Area = 28.27
DEBUG: Program ended
```

Macros are text replacements done by the preprocessor before compilation. Use parentheses to avoid operator precedence issues.

Program 2: Viewing Preprocessor Output

```c
#include <stdio.h>
#define DOUBLE(x) ((x) * 2)
#define GREET "Hello, World!"
int main() {
  int a = 5;
  int b = DOUBLE(a);
  printf("%s\n", GREET);
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Compile with: `gcc -E prog.c -o prog.i`
The `-E` flag stops after preprocessing.

```
# Preprocessed output (prog.i excerpt):
# 1 "prog.c"
# 1 "<built-in>"
...
# 1 "/usr/include/stdio.h" 1 3 4
...
int main() {
  int a = 5;
  int b = ((a) * 2);
  printf("%s\n", "Hello, World!");
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Notice macros are expanded and header files are included inline.

Program 3: Conditional Compilation

```c
#include <stdio.h>
#define VERSION 2
int main() {
#if VERSION == 1
  printf("Running version 1\n");
  printf("Basic features only\n");
#elif VERSION == 2
  printf("Running version 2\n");
  printf("Enhanced features\n");
#else
  printf("Unknown version\n");
#endif
#ifdef __linux__
  printf("Linux OS\n");
#elif defined(__APPLE__)
  printf("macOS\n");
#elif defined(_WIN32)
  printf("Windows OS\n");
#endif
  return 0;
}
```

```
Running version 2
Enhanced features
macOS
```

Conditional compilation allows platform-specific or version-specific code. The preprocessor includes only the relevant sections.

Program 4: Predefined Macros

```c
#include <stdio.h>
int main() {
  printf("File: %s\n", __FILE__);
  printf("Line: %d\n", __LINE__);
  printf("Date: %s\n", __DATE__);
  printf("Time: %s\n", __TIME__);
  printf("Function: %s\n", __func__);
#ifdef __STDC__
  printf("Standard C: YES\n");
#endif
#ifdef __STDC_VERSION__
  printf("C Version: %ldL\n", __STDC_VERSION__);
#endif
  printf("Line again: %d\n", __LINE__);
  return 0;
}
```

```
File: prog.c
Line: 5
Date: Jan 17 2026
Time: 10:30:45
Function: main
Standard C: YES
C Version: 201112L
Line again: 15
```

Predefined macros provide compilation context information useful for debugging and logging.

Program 5: Include Guards

```c
// myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H
#define MAX_SIZE 100
typedef struct {
  int x, y;
} Point;
void print_point(Point p);
#endif
// prog.c
#include <stdio.h>
#include "myheader.h"
#include "myheader.h"
void print_point(Point p) {
  printf("(%d, %d)\n", p.x, p.y);
}
int main() {
  Point p = {5, 10};
  print_point(p);
  printf("MAX_SIZE: %d\n", MAX_SIZE);
  return 0;
}
```

```
(5, 10)
MAX_SIZE: 100
```

Include guards prevent multiple inclusion of the same header file. Without them, you'd get redefinition errors. The pattern is: `#ifndef HEADER_H`, `#define HEADER_H`, content, `#endif`.

Program 6: Stringification and Token Pasting

```c
#include <stdio.h>
#define STRINGIFY(x) #x
#define TOSTRING(x) STRINGIFY(x)
#define CONCAT(a, b) a##b
#define VARNAME(prefix, num) prefix##num
int main() {
  printf("%s\n", STRINGIFY(Hello World));
  printf("%s\n", TOSTRING(100 + 200));
  int CONCAT(my, Var) = 42;
  printf("myVar = %d\n", myVar);
  int VARNAME(value, 1) = 10;
  int VARNAME(value, 2) = 20;
  printf("value1 = %d\n", value1);
  printf("value2 = %d\n", value2);
  printf("Line: %s\n", TOSTRING(__LINE__));
  return 0;
}
```

```
Hello World
100 + 200
myVar = 42
value1 = 10
value2 = 20
Line: 16
```

The # operator stringifies (converts to string), and ## pastes tokens together. These are powerful preprocessor features.

Program 7: Compilation to Assembly

```c
// prog.c
int add(int a, int b) {
  return a + b;
}
int main() {
  int result = add(5, 3);
  return result;
}
```

Compile with: `gcc -S prog.c`
This generates `prog.s` (assembly code).

```
# Assembly output (prog.s excerpt):
_add:
  pushq %rbp
  movq  %rsp, %rbp
  movl  %edi, -4(%rbp)
  movl  %esi, -8(%rbp)
  movl  -4(%rbp), %edx
  movl  -8(%rbp), %eax
  addl  %edx, %eax
  popq  %rbp
  retq
_main:
  pushq %rbp
  movq  %rsp, %rbp
  movl  $3, %esi
  movl  $5, %edi
  callq _add
  ...
```

Program 8: Object File Creation

```c
// prog.c
#include <stdio.h>
int square(int n) {
  return n * n;
}
int main() {
  int x = 5;
  printf("Square: %d\n", square(x));
  return 0;
}
```

Compile to object file: `gcc -c prog.c`
This creates `prog.o` (binary object file).
View symbols: `nm prog.o`

```
# nm prog.o output:
0000000000000000 T _main
0000000000000030 T _square
                 U _printf

T = defined in text section
U = undefined (external reference)

# After linking:
$ gcc prog.c -o prog
$ ./prog
Square: 25
```

Object files contain machine code but aren't executable yet. They need linking.

Program 9: Separate Compilation

```c
// math_ops.h
#ifndef MATH_OPS_H
#define MATH_OPS_H
int add(int a, int b);
int multiply(int a, int b);
#endif
// math_ops.c
#include "math_ops.h"
int add(int a, int b) {
  return a + b;
}
int multiply(int a, int b) {
  return a * b;
}
// main.c
#include <stdio.h>
#include "math_ops.h"
int main() {
  printf("5 + 3 = %d\n", add(5, 3));
  printf("5 * 3 = %d\n", multiply(5, 3));
  return 0;
}
```

```
# Compile separately:
$ gcc -c math_ops.c
$ gcc -c main.c
$ gcc math_ops.o main.o -o prog
$ ./prog
5 + 3 = 8
5 * 3 = 15

# Or in one command:
$ gcc math_ops.c main.c -o prog
```

Separate compilation allows modular development. Only changed files need recompilation, saving build time for large projects.

Program 10: Static Libraries

```c
// lib_utils.c
#include <stdio.h>
void greet(const char *name) {
  printf("Hello, %s!\n", name);
}
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
// main.c
#include <stdio.h>
void greet(const char *name);
int factorial(int n);
int main() {
  greet("Alice");
  printf("5! = %d\n", factorial(5));
  return 0;
}
```

```
# Create static library:
$ gcc -c lib_utils.c
$ ar rcs libutils.a lib_utils.o
$ gcc main.c -L. -lutils -o prog
$ ./prog
Hello, Alice!
5! = 120

# ar: archive command
# r: replace/add files
# c: create archive
# s: create index
```

Static libraries (.a) are archives of object files. Code is copied into the executable at link time.

Program 11: Dynamic Libraries (Shared Objects)

```c
// libshared.c
#include <stdio.h>
void show_message() {
  printf("From shared library\n");
}
int compute(int x) {
  return x * x + 10;
}
// main.c
#include <stdio.h>
void show_message();
int compute(int x);
int main() {
  show_message();
  printf("Result: %d\n", compute(5));
  return 0;
}
```

```
# Create shared library (Linux):
$ gcc -fPIC -c libshared.c
$ gcc -shared -o libshared.so libshared.o
$ gcc main.c -L. -lshared -o prog
$ LD_LIBRARY_PATH=. ./prog
From shared library
Result: 35

# macOS uses .dylib instead of .so
# -fPIC: Position Independent Code
# -shared: Create shared library
```

Shared libraries are loaded at runtime, saving disk space and allowing updates without recompilation.

Program 12: Linker Symbols and nm

```c
// prog.c
#include <stdio.h>
int global_var = 42;
static int static_var = 10;
extern int extern_var;
void public_func() {
  printf("Public function\n");
}
static void private_func() {
  printf("Private function\n");
}
int main() {
  public_func();
  private_func();
  printf("global_var: %d\n", global_var);
  return 0;
}
```

```
$ gcc -c prog.c
$ nm prog.o
0000000000000020 T _main
0000000000000000 T _public_func
0000000000000010 t _private_func
0000000000000000 D _global_var
0000000000000004 d _static_var
                 U _extern_var
                 U _printf

T/t = text (code), T=global, t=local
D/d = data, D=global, d=local
U = undefined (needs linking)
```

The nm tool displays symbol tables from object files, showing which symbols are exported.

Program 13: Linking Multiple Object Files

```c
// file1.c
int shared_data = 100;
int get_shared() {
  return shared_data;
}
// file2.c
extern int shared_data;
void modify_shared(int val) {
  shared_data += val;
}
// main.c
#include <stdio.h>
extern int shared_data;
int get_shared();
void modify_shared(int val);
int main() {
  printf("Initial: %d\n", get_shared());
  modify_shared(50);
  printf("After modify: %d\n", shared_data);
  return 0;
}
```

```
$ gcc -c file1.c file2.c main.c
$ gcc file1.o file2.o main.o -o prog
$ ./prog
Initial: 100
After modify: 150

# The linker resolves:
# - main.c needs get_shared() -> found in file1.o
# - main.c needs modify_shared() -> found in file2.o
# - file2.c needs shared_data -> found in file1.o
```

The linker resolves external references by matching symbols across object files.

Program 14: Weak Symbols

```c
// default.c
#include <stdio.h>
__attribute__((weak))
void custom_handler() {
  printf("Default handler\n");
}
void process() {
  custom_handler();
}
// main1.c (use default)
void process();
int main() {
  process();
  return 0;
}
// main2.c (override)
#include <stdio.h>
void custom_handler() {
  printf("Custom handler\n");
}
void process();
int main() {
  process();
  return 0;
}
```

```
$ gcc default.c main1.c -o prog1
$ ./prog1
Default handler

$ gcc default.c main2.c -o prog2
$ ./prog2
Custom handler
```

Weak symbols can be overridden by strong symbols during linking. Useful for providing default implementations that can be customized.

Program 15: Optimization Levels

```c
// opt_test.c
#include <stdio.h>
int sum_array(int *arr, int n) {
  int total = 0;
  for (int i = 0; i < n; i++) {
    total += arr[i];
  }
  return total;
}
int main() {
  int arr[1000];
  for (int i = 0; i < 1000; i++) {
    arr[i] = i;
  }
  int result = sum_array(arr, 1000);
  printf("Sum: %d\n", result);
  return 0;
}
```

```
# Compile with different optimization:
$ gcc -O0 opt_test.c -o prog_O0
$ gcc -O1 opt_test.c -o prog_O1
$ gcc -O2 opt_test.c -o prog_O2
$ gcc -O3 opt_test.c -o prog_O3
$ ls -lh prog_O*
-rwxr-xr-x prog_O0  (largest, slowest)
-rwxr-xr-x prog_O1
-rwxr-xr-x prog_O2
-rwxr-xr-x prog_O3  (smallest, fastest)

-O0: No optimization (default, debug-friendly)
-O1: Basic optimization
-O2: Recommended for release
-O3: Aggressive optimization
-Os: Optimize for size
```

Program 16: Debug Symbols

```c
// debug_test.c
#include <stdio.h>
int buggy_divide(int a, int b) {
  return a / b;
}
int main() {
  int x = 10;
  int y = 0;
  int result = buggy_divide(x, y);
  printf("Result: %d\n", result);
  return 0;
}
```

Compile with debug info: `gcc -g debug_test.c`

Use debugger: `gdb ./a.out`

```
$ gcc -g debug_test.c -o prog
$ gdb ./prog
(gdb) run
Program received signal SIGFPE

(gdb) backtrace
#0  buggy_divide (a=10, b=0)
    at debug_test.c:4
#1  main () at debug_test.c:9

(gdb) print a
$1 = 10
(gdb) print b
$2 = 0

# -g includes source line info
# Allows breakpoints, stepping, variable inspection
```

Program 17: Undefined References

```c
// main.c
#include <stdio.h>
extern void missing_function();
extern int missing_var;
void defined_function() {
  printf("I'm defined!\n");
}
int main() {
  defined_function();
  missing_function();
  printf("Var: %d\n", missing_var);
  return 0;
}
```

Try to compile: `gcc main.c`

```
$ gcc main.c
Undefined symbols for architecture x86_64:
  "_missing_function", referenced from:
      _main in main-xxxx.o
  "_missing_var", referenced from:
      _main in main-xxxx.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed

# Compilation succeeds (generates main.o)
# Linking fails (can't find symbols)
```

The linker reports undefined references when it can't find symbol definitions in any object file or library.

Program 18: Multiple Definition Errors

```c
// file1.c
int global_value = 10;
void func1() {
}
// file2.c
int global_value = 20;
void func1() {
}
// main.c
#include <stdio.h>
extern int global_value;
void func1();
int main() {
  func1();
  printf("%d\n", global_value);
  return 0;
}
```

```
$ gcc file1.c file2.c main.c
duplicate symbol '_global_value' in:
    file1.o
    file2.o
duplicate symbol '_func1' in:
    file1.o
    file2.o
ld: 2 duplicate symbols for architecture x86_64

# Fix: Make one static or extern
// file1.c: int global_value = 10;
// file2.c: extern int global_value;
```

Multiple definitions cause linker errors. Use static for file-local symbols or extern for shared symbols.

Program 19: Link Order Matters

```c
// liba.c
#include <stdio.h>
void func_b();
void func_a() {
  printf("Function A\n");
  func_b();
}
// libb.c
#include <stdio.h>
void func_b() {
  printf("Function B\n");
}
// main.c
void func_a();
int main() {
  func_a();
  return 0;
}
```

```
$ gcc -c liba.c libb.c main.c
$ ar rcs liba.a liba.o
$ ar rcs libb.a libb.o

# Wrong order - fails:
$ gcc main.o -L. -lina -libb
undefined reference to 'func_b'

# Correct order - succeeds:
$ gcc main.o -L. -liba -libb -o prog
$ ./prog
Function A
Function B

# Libraries are searched left-to-right
# Dependencies should come after dependents
```

Program 20: Complete Build Process

```c
// config.h
#define VERSION "1.0"
#define MAX_USERS 100
// utils.c
#include <stdio.h>
#include "config.h"
void show_config() {
  printf("Version: %s\n", VERSION);
  printf("Max users: %d\n", MAX_USERS);
}
// main.c
#include <stdio.h>
#include "config.h"
void show_config();
int main() {
  printf("Starting app v%s\n", VERSION);
  show_config();
  printf("Initialized!\n");
  return 0;
}
```

```
# Full build process:
$ gcc -E main.c -o main.i      # Preprocess
$ gcc -S main.i -o main.s      # Compile to assembly
$ gcc -c main.s -o main.o      # Assemble to object
$ gcc -c utils.c -o utils.o    # Compile utils
$ gcc main.o utils.o -o app    # Link
$ ./app
Starting app v1.0
Version: 1.0
Max users: 100
Initialized!

# Or simply:
$ gcc main.c utils.c -o app

# Each step transforms code toward executable:
# .c -> .i -> .s -> .o -> executable
```