C Preprocessor

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Object-like Macros

Program 1:

```c
#include <stdio.h>
#define PI 3.14159
#define MAX 100
#define MESSAGE "Hello, World!"
int main() {
  printf("PI: %f\n", PI);
  printf("MAX: %d\n", MAX);
  printf("%s\n", MESSAGE);
  double area = PI * 5 * 5;
  printf("Area: %f\n", area);
  return 0;
}
```

Output:

```
PI: 3.141590
MAX: 100
Hello, World!
Area: 78.539750
```

Note:

```
Object-like macros are simple
text replacements. Preprocessor
replaces PI with 3.14159 before
compilation.
```

Function-like Macros

Program 2:

```c
#include <stdio.h>
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
int main() {
  int n = 5;
  printf("Square: %d\n", SQUARE(n));
  printf("Max: %d\n", MAX(10, 20));
  printf("Min: %d\n", MIN(10, 20));
  printf("Square expr: %d\n",
    SQUARE(3 + 2));
  return 0;
}
```

Output:

```
Square: 25
Max: 20
Min: 10
Square expr: 25
```

Note:

Parentheses around parameters and whole expression prevent precedence issues. SQUARE(3+2) expands to ((3+2) * (3+2)) = 25, not 3+2*3+2.

Macro Pitfalls

Program 3:

```c
#include <stdio.h>
#define SQUARE_BAD(x) x * x
#define SQUARE_GOOD(x) ((x) * (x))
#define INCREMENT(x) ((x)++)
int main() {
  int a = 5;
  printf("Bad: %d\n", SQUARE_BAD(a + 1));
  printf("Good: %d\n", SQUARE_GOOD(a + 1));
  int b = 5;
  int c = INCREMENT(b);
  printf("b: %d, c: %d\n", b, c);
  return 0;
}
```

Output:

```
Bad: 11
Good: 36
b: 6, c: 5
```

Note:

```
SQUARE_BAD(a+1) expands to a+1*a+1
= 5+1*5+1 = 11 (wrong!)
SQUARE_GOOD(a+1) expands to
((a+1)*(a+1)) = 36 (correct!)
Side effects can cause issues.
```

Stringification Operator

Program 4:

```c
#include <stdio.h>
#define PRINT_VAR(x) \
  printf(#x " = %d\n", x)
#define TO_STRING(x) #x
int main() {
  int age = 25;
  int count = 100;
  PRINT_VAR(age);
  PRINT_VAR(count);
  printf("String: %s\n", TO_STRING(hello));
  printf("Expr: %s\n",
    TO_STRING(5 + 3));
  return 0;
}
```

Output:

```
age = 25
count = 100
String: hello
Expr: 5 + 3
```

Note:

```
# operator converts parameter to
string literal. #x becomes "x".
Useful for debugging and logging.
```

Token Pasting Operator

Program 5:

```c
#include <stdio.h>
#define CONCAT(a, b) a##b
#define VAR_NAME(prefix , num) prefix##num
int main() {
  int xy = 100;
  int value1 = 10;
  int value2 = 20;
  printf("%d\n", CONCAT(x, y));
  printf("%d\n", VAR_NAME(value , 1));
  printf("%d\n", VAR_NAME(value , 2));
  return 0;
}
```

Output:

```
100
10
20
```

Note:

```
## operator pastes tokens together.
a##b becomes ab.
CONCAT(x, y) becomes xy.
VAR_NAME(value , 1) becomes value1.
```

Conditional Compilation - ifdef

Program 6:

```c
#include <stdio.h>
#define DEBUG
int main() {
#ifdef DEBUG
  printf("Debug mode enabled\n");
#endif
#ifndef RELEASE
  printf("Not in release mode\n");
#endif
#ifdef FEATURE_X
  printf("Feature X enabled\n");
#else
  printf("Feature X disabled\n");
#endif
  printf("Program running\n");
  return 0;
}
```

Output:

```
Debug mode enabled
Not in release mode
Feature X disabled
Program running
```

Note:

```
#ifdef checks if macro is defined.
#ifndef checks if not defined.
Code included or excluded based
on macro definitions.
```

Conditional Compilation - if defined

Program 7:

```c
#include <stdio.h>
#define FEATURE_A
#define FEATURE_B
int main() {
#if defined(FEATURE_A) && defined(FEATURE_B)
  printf("Both features enabled\n");
#elif defined(FEATURE_A)
  printf("Only A enabled\n");
#elif defined(FEATURE_B)
  printf("Only B enabled\n");
#else
  printf("No features enabled\n");
#endif
  return 0;
}
```

Output:

```
Both features enabled
```

Note:

```
defined() operator checks if macro
exists. Can combine with logical
operators (&&, ||, !).
#elif provides else-if functionality.
```

Conditional Compilation - Numeric

Program 8:

```c
#include <stdio.h>
#define VERSION 3
int main() {
#if VERSION == 1
  printf("Version 1 code\n");
#elif VERSION == 2
  printf("Version 2 code\n");
#elif VERSION >= 3
  printf("Version 3+ code\n");
#else
  printf("Unknown version\n");
#endif
  printf("Version: %d\n", VERSION);
  return 0;
}
```

Output:

```
Version 3+ code
Version: 3
```

Note:

```
#if can evaluate constant integer
expressions. Supports comparison
operators (==, !=, <, >, <=, >=).
Useful for version control.
```

Predefined Macros

Program 9:

```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__);
#ifdef __STDC__
  printf("Standard C: Yes\n");
#endif
  printf("Line: %d\n", __LINE__);
  return 0;
}
```

Output:

```
File: program.c
Line: 4
Date: Jan 16 2026
Time: 20:15:30
Standard C: Yes
Line: 11
```

Note:

```
Predefined macros provide
compilation context.
__LINE__ updates dynamically.
Useful for debugging and logging.
```

Macro Undef

Program 10:

```c
#include <stdio.h>
#define MAX 100
int main() {
  printf("MAX: %d\n", MAX);
#undef MAX
#define MAX 200
  printf("MAX: %d\n", MAX);
#undef MAX
#ifdef MAX
  printf("MAX defined\n");
#else
  printf("MAX not defined\n");
#endif
  return 0;
}
```

Output:

```
MAX: 100
MAX: 200
MAX not defined
```

Note:

```
#undef removes macro definition.
Can redefine macro after #undef.
Useful to prevent conflicts with
library macros.
```

Multiline Macros

Program 11:

```c
#include <stdio.h>
#define SWAP(a, b, type) \
  do { \
    type temp = a; \
    a = b; \
    b = temp; \
  } while(0)
int main() {
  int x = 10, y = 20;
  printf("Before: x=%d, y=%d\n", x, y);
  SWAP(x, y, int);
  printf("After: x=%d, y=%d\n", x, y);
  return 0;
}
```

Output:

```
Before: x=10, y=20
After: x=20, y=10
```

Note:

Backslash continues macro to next line. do-while(0) ensures macro acts like single statement in all contexts (if, else, etc.).

Variadic Macros

Program 12:

```c
#include <stdio.h>
#define LOG(fmt, ...) \
  printf("[LOG] " fmt "\n", __VA_ARGS__)
#define DEBUG_PRINT(fmt, ...) \
  printf("%s:%d " fmt "\n", \
    __FILE__, __LINE__, ##__VA_ARGS__)
int main() {
  LOG("Value: %d", 42);
  LOG("x=%d, y=%d", 10, 20);
  DEBUG_PRINT("Starting");
  DEBUG_PRINT("Count: %d", 5);
  return 0;
}
```

Output:

```
[LOG] Value: 42
[LOG] x=10, y=20
program.c:9 Starting
program.c:10 Count: 5
```

Note:

```
... accepts variable arguments.
__VA_ARGS__ expands to all args.
## before __VA_ARGS__ removes
comma if no arguments provided.
```

Error and Warning Directives

Program 13:

```c
#include <stdio.h>
#define MIN_VERSION 2
#define CURRENT_VERSION 3
#if CURRENT_VERSION < MIN_VERSION
  #error "Version too old"
#endif
#ifndef PLATFORM
  #warning "Platform not defined"
#endif
int main() {
  printf("Compilation successful\n");
  printf("Version: %d\n", CURRENT_VERSION);
  return 0;
}
```

Output:

```
warning: Platform not defined
Compilation successful
Version: 3
```

Note:

```
#error stops compilation with
message. #warning shows warning
but continues. Useful for enforcing
requirements at compile time.
```

Pragma Directive

Program 14:

```c
#include <stdio.h>
#pragma message("Compiling program...")
#pragma pack(push, 1)
struct Packed {
  char c;
  int i;
  char d;
};
#pragma pack(pop)
struct Normal {
  char c;
  int i;
  char d;
};
int main() {
  printf("Packed: %lu\n",
    sizeof(struct Packed));
  printf("Normal: %lu\n",
    sizeof(struct Normal));
  return 0;
}
```

Output:

```
Compiling program...
Packed: 6
Normal: 12
```

Note:

```
#pragma provides compiler-specific
directives. pack(1) removes padding.
Normal struct has padding for
alignment. Compiler-dependent.
```

Include Guard Pattern

myheader.h:

```c
#ifndef MYHEADER_H
#define MYHEADER_H
#define CONSTANT 42
int add(int a, int b);
#endif
```

Program 15:

```c
#include <stdio.h>
#include "myheader.h"
#include "myheader.h"
int add(int a, int b) {
  return a + b;
}
int main() {
  printf("Constant: %d\n", CONSTANT);
  printf("Sum: %d\n", add(5, 3));
  return 0;
}
```

Output:

```
Constant: 42
Sum: 8
```

Note:

```
Include guards prevent multiple
inclusion. Header included twice
but content processed once.
Standard pattern for all headers.
```

Macro Debugging

Program 16:

```c
#include <stdio.h>
#define DEBUG
#ifdef DEBUG
  #define DBG(x) printf("DEBUG: " #x \
    " = %d at line %d\n", x, __LINE__)
#else
  #define DBG(x)
#endif
int main() {
  int count = 10;
  int total = 50;
  DBG(count);
  DBG(total);
  DBG(count + total);
  return 0;
}
```

Output:

```
DEBUG: count = 10 at line 12
DEBUG: total = 50 at line 13
DEBUG: count + total = 60 at line 14
```

Note:

```
Conditional debug macro. Enabled
when DEBUG defined, disabled
otherwise. No runtime overhead
when disabled.
```

Assert Macro

Program 17:

```c
#include <stdio.h>
#define ASSERT(cond) \
  if (!(cond)) { \
    printf("Assertion failed: " #cond \
      " at %s:%d\n", __FILE__, __LINE__); \
    return 1; \
  }
int divide(int a, int b) {
  ASSERT(b != 0);
  return a / b;
}
int main() {
  printf("10/2 = %d\n", divide(10, 2));
  printf("10/0 = %d\n", divide(10, 0));
  printf("Done\n");
  return 0;
}
```

Output:

```
10/2 = 5
Assertion failed: b != 0
at program.c:9
```

Note:

Custom assert macro for runtime checks. Shows condition, file, and line on failure. Returns early to prevent undefined behavior.

Platform-Specific Code

Program 18:

```c
#include <stdio.h>
int main() {
#ifdef _WIN32
  printf("Windows platform\n");
  const char *separator = "\\";
#elif defined(__linux__)
  printf("Linux platform\n");
  const char *separator = "/";
#elif defined(__APPLE__)
  printf("macOS platform\n");
  const char *separator = "/";
#else
  printf("Unknown platform\n");
  const char *separator = "/";
#endif
  printf("Separator: %s\n", separator);
  return 0;
}
```

Output (macOS):

```
macOS platform
Separator: /
```

Note:

```
Platform detection using predefined
macros. Different code compiled for
different platforms. Write once,
compile anywhere.
```

Compiler-Specific Features

Program 19:

```c
#include <stdio.h>
int main() {
#ifdef __GNUC__
  printf("GCC version: %d.%d.%d\n",
    __GNUC__, __GNUC_MINOR__,
    __GNUC_PATCHLEVEL__);
#endif
#ifdef __clang__
  printf("Clang version: %d.%d.%d\n",
    __clang_major__, __clang_minor__,
    __clang_patchlevel__);
#endif
#ifdef _MSC_VER
  printf("MSVC version: %d\n", _MSC_VER);
#endif
  return 0;
}
```

Output (GCC):

```
GCC version: 11.2.0
```

Output (Clang):

```
Clang version: 13.0.0
```

Note:

```
Compiler detection using predefined
macros. Access compiler version.
Enable compiler-specific features.
```

Build Configuration

Program 20:

```c
#include <stdio.h>
#ifndef BUILD_TYPE
  #define BUILD_TYPE "unknown"
#endif
#ifndef OPTIMIZATION_LEVEL
  #define OPTIMIZATION_LEVEL 0
#endif
int main() {
  printf("Build: %s\n", BUILD_TYPE);
  printf("Optimization: %d\n",
    OPTIMIZATION_LEVEL);
#if OPTIMIZATION_LEVEL >= 2
  printf("High optimization enabled\n");
#else
  printf("Low optimization\n");
#endif
  return 0;
}
```

Output:

```
Build: unknown
Optimization: 0
Low optimization
```

Compile with flags:

```
gcc -DBUILD_TYPE=\"release\" \
  -DOPTIMIZATION_LEVEL=3 prog.c

Build: release
Optimization: 3
High optimization enabled
```

Note:

```
Macros from command line with -D.
Configure build without changing
source code.
```