Multifile Compilation

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Basic Header File

math_ops.h:

```c
int add(int a, int b);
int subtract(int a, int b);
```

math_ops.c:

```c
#include "math_ops.h"
int add(int a, int b) {
  return a + b;
}
int subtract(int a, int b) {
  return a - b;
}
```

main.c:

```c
#include <stdio.h>
#include "math_ops.h"
int main() {
  int x = 10, y = 5;
  printf("Add: %d\n", add(x, y));
  printf("Subtract: %d\n", subtract(x, y));
  return 0;
}
```

Output:

```
Add: 15
Subtract: 5
```

Compile:

```
gcc -c math_ops.c
gcc -c main.c
gcc math_ops.o main.o -o prog
./prog
```

Include Guards

utils.h:

```c
#ifndef UTILS_H
#define UTILS_H
void print_message();
#endif
```

utils.c:

```c
#include <stdio.h>
#include "utils.h"
void print_message() {
  printf("Hello from utils!\n");
}
```

main.c:

```c
#include <stdio.h>
#include "utils.h"
#include "utils.h"
int main() {
  print_message();
  return 0;
}
```

Output:

```
Hello from utils!
```

Note:

```
Include guards prevent multiple
inclusion errors. Header included
twice but compiled once.
```

Extern Variables

globals.h:

```c
#ifndef GLOBALS_H
#define GLOBALS_H
extern int counter;
extern char name[];
#endif
```

globals.c:

```c
int counter = 0;
char name[] = "Global";
```

main.c:

```c
#include <stdio.h>
#include "globals.h"
int main() {
  printf("Counter: %d\n", counter);
  printf("Name: %s\n", name);
  counter = 42;
  printf("Counter: %d\n", counter);
  return 0;
}
```

Output:

```
Counter: 0
Name: Global
Counter: 42
```

Note:

```
extern declares variable defined
in another file. Definition in
globals.c, declaration in globals.h
```

Static Functions

helper.c:

```c
#include <stdio.h>
static void internal_func() {
  printf("Internal helper\n");
}
void public_func() {
  printf("Public function\n");
  internal_func();
}
```

helper.h:

```c
#ifndef HELPER_H
#define HELPER_H
void public_func();
#endif
```

main.c:

```c
#include "helper.h"
int main() {
  public_func();
  return 0;
}
```

Output:

```
Public function
Internal helper
```

Note:

```
static makes function visible
only within its file. Cannot be
called from other files.
```

Static Variables

counter.h:

```c
#ifndef COUNTER_H
#define COUNTER_H
void increment();
int get_count();
#endif
```

counter.c:

```c
static int count = 0;
void increment() {
  count++;
}
int get_count() {
  return count;
}
```

main.c:

```c
#include <stdio.h>
#include "counter.h"
int main() {
  printf("Count: %d\n", get_count());
  increment();
  increment();
  printf("Count: %d\n", get_count());
  return 0;
}
```

Output:

```
Count: 0
Count: 2
```

Note:

```
static variable has file scope.
Only accessible within counter.c
through provided functions.
```

Multiple Source Files

calc.h:

```
#ifndef CALC_H
#define CALC_H
int multiply(int a, int b);
int divide(int a, int b);
#endif
```

calc.c:

```
#include "calc.h"
int multiply(int a, int b) {
  return a * b;
}
int divide(int a, int b) {
  if (b != 0) return a / b;
  return 0;
}
```

main.c:

```
#include <stdio.h>
#include "calc.h"
int main() {
  printf("Multiply: %d\n", multiply(6, 7));
  printf("Divide: %d\n", divide(20, 4));
  return 0;
}
```

¡function$_c$*alls* $>$ **Output:**

```
Multiply: 42
Divide: 5
```

Compile:

```
gcc -c calc.c -o calc.o
gcc -c main.c -o main.o
gcc calc.o main.o -o program
```

Struct in Header

point.h:

```c
#ifndef POINT_H
#define POINT_H
typedef struct {
  int x;
  int y;
} Point;
Point create_point(int x, int y);
void print_point(Point p);
#endif
```

point.c:

```c
#include <stdio.h>
#include "point.h"
Point create_point(int x, int y) {
  Point p = {x, y};
  return p;
}
void print_point(Point p) {
  printf("(%d, %d)\n", p.x, p.y);
}
```

main.c:

```c
#include "point.h"
int main() {
  Point p1 = create_point(10, 20);
  print_point(p1);
  return 0;
}
```

Output:

```
(10, 20)
```

Note:

```
Struct definition in header allows
use in all files that include it.
Functions operate on the struct.
```

Circular Dependencies

a.h:

```c
#ifndef A_H
#define A_H
typedef struct B B;
typedef struct A {
  int value;
  B *b_ptr;
} A;
void print_a(A *a);
#endif
```

b.h:

```c
#ifndef B_H
#define B_H
typedef struct A A;
typedef struct B {
  int value;
  A *a_ptr;
} B;
void print_b(B *b);
#endif
```

main.c:

```c
#include <stdio.h>
#include "a.h"
#include "b.h"
int main() {
  A a = {10, NULL};
  B b = {20, &a};
  printf("A: %d, B: %d\n", a.value, b.value);
```

Output:

```
A: 10, B: 20
```

Note:

```
Forward declarations break circular
dependencies. Each header declares
the other's struct as incomplete.
```

Const in Header

constants.h:

```c
#ifndef CONSTANTS_H
#define CONSTANTS_H
extern const int MAX_SIZE;
extern const double PI;
#endif
```

constants.c:

```c
const int MAX_SIZE = 100;
const double PI = 3.14159;
```

main.c:

```c
#include <stdio.h>
#include "constants.h"
int main() {
  printf("Max: %d\n", MAX_SIZE);
  printf("PI: %.5f\n", PI);
  return 0;
}
```

Output:

```
Max: 100
PI: 3.14159
```

Note:

```
Const variables declared extern
in header, defined in source.
Shared across files as constants.
```

Enum in Header

status.h:

```c
#ifndef STATUS_H
#define STATUS_H
typedef enum {
  SUCCESS,
  ERROR,
  PENDING
} Status;
const char* status_string(Status s);
#endif
```

status.c:

```c
#include "status.h"
const char* status_string(Status s) {
  switch(s) {
    case SUCCESS: return "Success";
    case ERROR: return "Error";
    case PENDING: return "Pending";
    default: return "Unknown";
  }
}
```

main.c:

```c
#include <stdio.h>
#include "status.h"
int main() {
  Status s = SUCCESS;
  printf("Status: %s\n", status_string(s));
  return 0;
}
```

Output:

```
Status: Success
```

Note:

```
Enum definition in header makes
it available to all files. Helper
function converts to string.
```

Function Pointers in Header

callback.h:

```c
#ifndef CALLBACK_H
#define CALLBACK_H
typedef void (*Callback)(int);
void register_callback(Callback cb);
void trigger();
#endif
```

callback.c:

```c
#include "callback.h"
static Callback callback = NULL;
void register_callback(Callback cb) {
  callback = cb;
}
void trigger() {
  if (callback) callback(42);
}
```

main.c:

```c
#include <stdio.h>
#include "callback.h"
void my_callback(int value) {
  printf("Callback: %d\n", value);
}
int main() {
  register_callback(my_callback);
  trigger();
  return 0;
}
```

Output:

```
Callback: 42
```

Note:

```
Function pointer typedef in header.
Callback stored statically and
invoked when triggered.
```

Inline Functions in Header

math.h:

```c
#ifndef MATH_H
#define MATH_H
static inline int square(int x) {
  return x * x;
}
static inline int cube(int x) {
  return x * x * x;
}
#endif
```

main.c:

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

Output:

```
Square: 25
Cube: 125
```

Note:

```
Inline functions in header are
expanded at call site. Use static
inline to avoid multiple definition.
```

Library with Multiple Files

string_utils.h:

```c
#ifndef STRING_UTILS_H
#define STRING_UTILS_H
int str_len(const char *s);
void str_upper(char *s);
#endif
```

string_utils.c:

```c
#include "string_utils.h"
int str_len(const char *s) {
  int len = 0;
  while (s[len]) len++;
  return len;
}
void str_upper(char *s) {
  int i;
  for (i = 0; s[i]; i++) {
    if (s[i] >= 'a' && s[i] <= 'z')
      s[i] -= 32;
  }
}
```

main.c:

```c
#include <stdio.h>
#include "string_utils.h"
int main() {
  char str[] = "hello";
  printf("Length: %d\n", str_len(str));
  str_upper(str);
  printf("Upper: %s\n", str);
```

Output:

```
Length: 5
Upper: HELLO
```

Compile:

```
gcc -c string_utils.c
gcc -c main.c
gcc string_utils.o main.o -o prog
```

Nested Headers

base.h:

```c
#ifndef BASE_H
#define BASE_H
typedef struct {
  int id;
} Base;
#endif
```

derived.h:

```c
#ifndef DERIVED_H
#define DERIVED_H
#include "base.h"
typedef struct {
  Base base;
  char name[20];
} Derived;
#endif
```

main.c:

```c
#include <stdio.h>
#include <string.h>
#include "derived.h"
int main() {
  Derived d;
  d.base.id = 100;
  strcpy(d.name, "Test");
  printf("ID: %d, Name: %s\n",
    d.base.id, d.name);
  return 0;
}
```

Output:

```
ID: 100, Name: Test
```

Note:

```
Headers can include other headers.
Include guards prevent multiple
inclusion of base.h.
```

Multiple Object Files

add.c:

```c
int add(int a, int b) {
  return a + b;
}
```

sub.c:

```c
int sub(int a, int b) {
  return a - b;
}
```

ops.h:

```c
#ifndef OPS_H
#define OPS_H
int add(int a, int b);
int sub(int a, int b);
#endif
```

main.c:

```c
#include <stdio.h>
#include "ops.h"
int main() {
  printf("Add: %d\n", add(8, 3));
  printf("Sub: %d\n", sub(8, 3));
  return 0;
}
```

Output:

```
Add: 11
Sub: 5
```

Compile:

```
gcc -c add.c
gcc -c sub.c
gcc -c main.c
gcc add.o sub.o main.o -o prog
```

Global Array Sharing

data.h:

```c
#ifndef DATA_H
#define DATA_H
extern int data[5];
void init_data();
void print_data();
#endif
```

data.c:

```c
#include <stdio.h>
#include "data.h"
int data[5];
void init_data() {
  int i;
  for (i = 0; i < 5; i++)
    data[i] = i * 10;
}
void print_data() {
  int i;
  for (i = 0; i < 5; i++)
    printf("%d ", data[i]);
  printf("\n");
}
```

main.c:

```c
#include "data.h"
int main() {
  init_data();
  print_data();
  return 0;
```

Output:

```
0 10 20 30 40
```

Note:

```
Global array defined in data.c,
declared extern in data.h.
Accessible from any file including
the header.
```

Opaque Pointers

handle.h:

```c
#ifndef HANDLE_H
#define HANDLE_H
typedef struct Handle Handle;
Handle* create_handle(int val);
void set_value(Handle *h, int val);
int get_value(Handle *h);
void destroy_handle(Handle *h);
#endif
```

handle.c:

```c
#include <stdlib.h>
#include "handle.h"
struct Handle {
  int value;
};
Handle* create_handle(int val) {
  Handle *h = malloc(sizeof(Handle));
  h->value = val;
  return h;
}
void set_value(Handle *h, int val) {
  h->value = val;
}
int get_value(Handle *h) {
  return h->value;
}
void destroy_handle(Handle *h) {
  free(h);
}
```

main.c:

```c
#include <stdio.h>
#include "handle.h"
int main() {
  Handle *h = create_handle(100);
  printf("Value: %d\n", get_value(h));
  set_value(h, 200);
  printf("Value: %d\n", get_value(h));
  destroy_handle(h);
  return 0;
}
```

Output:

```
Value: 100
Value: 200
```

Note:

```
Opaque pointer hides implementation.
Struct details only in .c file.
Encapsulation in C.
```

Conditional Compilation in Header

debug.h:

```c
#ifndef DEBUG_H
#define DEBUG_H
#include <stdio.h>
#ifdef DEBUG
  #define LOG(msg) printf("DEBUG: %s\n", msg)
#else
  #define LOG(msg)
#endif
#endif
```

main.c:

```c
#define DEBUG
#include "debug.h"
int main() {
  LOG("Starting program");
  printf("Hello\n");
  LOG("Ending program");
  return 0;
}
```

Output:

```
DEBUG: Starting program
Hello
DEBUG: Ending program
```

Without DEBUG:

```
Hello
```

Note:

```
Conditional macros in headers.
LOG expands to printf when DEBUG
defined, nothing otherwise.
```

Version Management

version.h:

```c
#ifndef VERSION_H
#define VERSION_H
#define MAJOR 1
#define MINOR 2
#define PATCH 3
extern const char* get_version();
#endif
```

version.c:

```c
#include <stdio.h>
#include "version.h"
static char version[20];
const char* get_version() {
  sprintf(version, "%d.%d.%d",
    MAJOR, MINOR, PATCH);
  return version;
}
```

main.c:

```c
#include <stdio.h>
#include "version.h"
int main() {
  printf("Version: %s\n", get_version());
  return 0;
}
```

Output:

```
Version: 1.2.3
```

Note:

```
Version numbers as macros in header.
Function formats version string.
Central version management.
```

Module Initialization

module.h:

```c
#ifndef MODULE_H
#define MODULE_H
void module_init();
void module_cleanup();
void module_work();
#endif
```

module.c:

```c
#include <stdio.h>
#include "module.h"
static int initialized = 0;
void module_init() {
  if (!initialized) {
    printf("Module initialized\n");
    initialized = 1;
  }
}
void module_cleanup() {
  printf("Module cleaned up\n");
  initialized = 0;
}
void module_work() {
  if (initialized)
    printf("Module working\n");
}
```

main.c:

```c
#include "module.h"
int main() {
  module_init();
  module_work();
  module_cleanup();
  return 0;
}
```

Output:

```
Module initialized
Module working
Module cleaned up
```

Note:

```
Module pattern with init/cleanup.
Static flag tracks initialization.
Common pattern in C libraries.
```