C Programming - Deck 21

Enumerations and Unions

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Enumerations (enum)

- User-defined data type for named integer constants
- Makes code more readable and maintainable
- Syntax: `enum name {value1, value2, ...};`
- Default values start at 0 and increment by 1
- Can assign custom values
- Internally stored as integers
- Used for states, days, months, status codes, etc.
- Better than using magic numbers

Program 1: Basic Enum Declaration

```c
#include <stdio.h>
enum Day {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
};
int main() {
  enum Day today = WEDNESDAY;
  printf("Today is day: %d\n", today);
  printf("SUNDAY = %d\n", SUNDAY);
  printf("MONDAY = %d\n", MONDAY);
  printf("SATURDAY = %d\n", SATURDAY);
  return 0;
}
```

Output:

```
Today is day: 3
SUNDAY = 0
MONDAY = 1
SATURDAY = 6
```

Default values: 0, 1, 2, 3, ...

Program 2: Enum with Custom Values

```c
#include <stdio.h>
enum ErrorCode {
  SUCCESS = 0,
  FILE_NOT_FOUND = 404,
  SERVER_ERROR = 500,
  TIMEOUT = 408
};
int main() {
  enum ErrorCode status = FILE_NOT_FOUND;
  printf("Status: %d\n", status);
  printf("SUCCESS = %d\n", SUCCESS);
  printf("SERVER_ERROR = %d\n", SERVER_ERROR);
  printf("TIMEOUT = %d\n", TIMEOUT);
  return 0;
}
```

Output:

```
Status: 404
SUCCESS = 0
SERVER_ERROR = 500
TIMEOUT = 408
```

Can assign any integer values

Program 3: Enum with Partial Custom Values

```c
#include <stdio.h>
enum Month {
  JAN = 1,
  FEB,
  MAR,
  APR,
  MAY,
  JUN,
  JUL,
  AUG,
  SEP,
  OCT,
  NOV,
  DEC
};
int main() {
  printf("JAN = %d\n", JAN);
  printf("FEB = %d\n", FEB);
  printf("DEC = %d\n", DEC);
  return 0;
}
```

Output:

```
JAN = 1
FEB = 2
DEC = 12
```

After custom value, auto-increment continues

Program 4: Switch Case with Enum

```c
#include <stdio.h>
enum Color { RED, GREEN, BLUE, YELLOW };
int main() {
  enum Color c = GREEN;
  switch(c) {
    case RED:
      printf("Red color\n");
      break;
    case GREEN:
      printf("Green color\n");
      break;
    case BLUE:
      printf("Blue color\n");
      break;
    case YELLOW:
      printf("Yellow color\n");
      break;
  }
  return 0;
}
```

Output:

```
Green color
```

Enums make switch statements readable

Program 5: Enum for State Machine

```c
#include <stdio.h>
enum State { IDLE, RUNNING, PAUSED, STOPPED };
int main() {
  enum State current = IDLE;
  printf("Initial state: %d\n", current);
  current = RUNNING;
  printf("Changed to: %d\n", current);
  current = PAUSED;
  printf("Changed to: %d\n", current);
  current = STOPPED;
  printf("Final state: %d\n", current);
  return 0;
}
```

Output:

```
Initial state: 0
Changed to: 1
Changed to: 2
Final state: 3
```

Enums perfect for state management

Unions

- User-defined data type like structure
- All members share the same memory location
- Only one member can hold value at a time
- Size equals size of largest member
- Syntax: `union name {type1 m1; type2 m2; ...};`
- Memory efficient when only one member used at a time
- Useful for type punning and variant types
- Writing to one member overwrites others

Program 6: Basic Union Declaration

```c
#include <stdio.h>
union Data {
  int i;
  float f;
  char c;
};
int main() {
  union Data d;
  d.i = 10;
  printf("d.i = %d\n", d.i);
  d.f = 3.14;
  printf("d.f = %.2f\n", d.f);
  printf("d.i now = %d (garbage)\n", d.i);
  d.c = 'A';
  printf("d.c = %c\n", d.c);
  printf("d.f now = %.2f (garbage)\n", d.f);
  return 0;
}
```

Output:

```
d.i = 10
d.f = 3.14
d.i now = 1078523331 (garbage)
d.c = A
d.f now = 0.00 (garbage)
```

Only one member valid at a time

Program 7: Union vs Struct Size

```c
#include <stdio.h>
struct StructData {
  int i;
  float f;
  char c;
};
union UnionData {
  int i;
  float f;
  char c;
};
int main() {
  printf("Struct size: %lu bytes\n",
    sizeof(struct StructData));
  printf("Union size: %lu bytes\n",
    sizeof(union UnionData));
  printf("int: %lu, float: %lu, char: %lu\n",
    sizeof(int), sizeof(float), sizeof(char));
  return 0;
}
```

Output:

```
Struct size: 12 bytes
Union size: 4 bytes
int: 4, float: 4, char: 1
```

Union takes size of largest member

Program 8: Union Memory Layout

```c
#include <stdio.h>
union Data {
  int i;
  float f;
  char c;
};
int main() {
  union Data d;
  printf("Address of union: %p\n", (void*)&d);
  printf("Address of d.i: %p\n", (void*)&d.i);
  printf("Address of d.f: %p\n", (void*)&d.f);
  printf("Address of d.c: %p\n", (void*)&d.c);
  printf("All members share same address\n");
  return 0;
}
```

Output:

```
Address of union: 0x7ffeeb3c4a10
Address of d.i: 0x7ffeeb3c4a10
Address of d.f: 0x7ffeeb3c4a10
Address of d.c: 0x7ffeeb3c4a10
All members share same address
```

All members at same memory location

Program 9: Union for Type Conversion

```c
#include <stdio.h>
union Convert {
  int i;
  char bytes[4];
};
int main() {
  union Convert c;
  c.i = 0x12345678;
  printf("Integer: 0x%X\n", c.i);
  printf("Bytes: ");
  int i;
  for (i = 0; i < 4; i++) {
    printf("0x%02X ", (unsigned char)c.bytes[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Integer: 0x12345678
Bytes: 0x78 0x56 0x34 0x12
```

Little-endian byte order visible

Program 10: Tagged Union (Discriminated Union)

```c
#include <stdio.h>
enum Type { INT, FLOAT, CHAR };
struct Tagged {
  enum Type type;
  union {
    int i;
    float f;
    char c;
  } data;
};
int main() {
  struct Tagged t1 = {INT, {.i = 42}};
  struct Tagged t2 = {FLOAT, {.f = 3.14}};
  if (t1.type == INT)
    printf("t1: %d\n", t1.data.i);
  if (t2.type == FLOAT)
    printf("t2: %.2f\n", t2.data.f);
  return 0;
}
```

Output:

```
t1: 42
t2: 3.14
```

Tag indicates which member is valid

Program 11: Union in Array

```c
#include <stdio.h>
union Number {
  int i;
  float f;
};
int main() {
  union Number arr[3];
  arr[0].i = 10;
  arr[1].f = 3.14;
  arr[2].i = 20;
  printf("arr[0].i = %d\n", arr[0].i);
  printf("arr[1].f = %.2f\n", arr[1].f);
  printf("arr[2].i = %d\n", arr[2].i);
  printf("Array size: %lu bytes\n", sizeof(arr));
  return 0;
}
```

Output:

```
arr[0].i = 10
arr[1].f = 3.14
arr[2].i = 20
Array size: 12 bytes
```

Array of unions: 3 * 4 bytes = 12

Program 12: Enum for Menu System

```c
#include <stdio.h>
enum Menu { ADD=1, SUB, MUL, DIV, EXIT };
int main() {
  int choice;
  printf("1. Add\n2. Subtract\n3. Multiply\n")
  printf("4. Divide\n5. Exit\nChoice: ");
  scanf("%d", &choice);
  switch(choice) {
    case ADD: printf("Addition\n"); break;
    case SUB: printf("Subtraction\n"); break;
    case MUL: printf("Multiplication\n"); break;
    case DIV: printf("Division\n"); break;
    case EXIT: printf("Exiting\n"); break;
    default: printf("Invalid\n");
  }
  return 0;
}
```

Output:

```
1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Choice: 3
Multiplication
```

- Enums for menu choices

Program 13: Nested Union in Structure

```c
#include <stdio.h>
struct Employee {
  char name[20];
  union {
    int hourly_rate;
    int monthly_salary;
  } pay;
  int is_hourly;
};
int main() {
  struct Employee e1 = {"Alice", {.hourly_rate=50}, 1};
  struct Employee e2 = {"Bob", {.monthly_salary=5000}, 0};
  printf("%s: ", e1.name);
  if (e1.is_hourly)
    printf("$%d/hour\n", e1.pay.hourly_rate);
  printf("%s: ", e2.name);
  if (!e2.is_hourly)
    printf("$%d/month\n", e2.pay.monthly_salary);
  return 0;
}
```

Output:

```
Alice: $50/hour
Bob: $5000/month
```

Union saves space for variant data

Program 14: Enum with typedef

```c
#include <stdio.h>
typedef enum {
  NORTH,
  SOUTH,
  EAST,
  WEST
} Direction;
int main() {
  Direction d = NORTH;
  printf("Direction: %d\n", d);
  d = EAST;
  printf("Changed to: %d\n", d);
  printf("No need to write 'enum'\n");
  return 0;
}
```

Output:

```
Direction: 0
Changed to: 2
No need to write 'enum'
```

typedef simplifies usage

Program 15: Union with Different Sizes

```c
#include <stdio.h>
union Mixed {
  char c;
  short s;
  int i;
  long l;
  double d;
};
int main() {
  union Mixed m;
  printf("Union size: %lu bytes\n", sizeof(m));
  printf("char: %lu, short: %lu\n",
    sizeof(char), sizeof(short));
  printf("int: %lu, long: %lu\n",
    sizeof(int), sizeof(long));
  printf("double: %lu\n", sizeof(double));
  printf("Union size = largest member\n");
  return 0;
}
```

Output:

```
Union size: 8 bytes
char: 1, short: 2
int: 4, long: 8
double: 8
Union size = largest member
```

Size is max(long, double) = 8

Program 16: Bit Fields in Union

```c
#include <stdio.h>
union Flags {
  unsigned int all;
  struct {
    unsigned int flag1 : 1;
    unsigned int flag2 : 1;
    unsigned int flag3 : 1;
    unsigned int reserved : 29;
  } bits;
};
int main() {
  union Flags f;
  f.all = 0;
  f.bits.flag1 = 1;
  f.bits.flag3 = 1;
  printf("all = %u\n", f.all);
  printf("Binary: flag3=%u flag2=%u flag1=%u\n",
    f.bits.flag3, f.bits.flag2, f.bits.flag1);
  return 0;
}
```

Output:

```
all = 5
Binary: flag3=1 flag2=0 flag1=1
```

Access individual bits or whole value

Program 17: Enum for Return Codes

```c
#include <stdio.h>
typedef enum {
  OK = 0,
  ERROR_NULL_PTR = -1,
  ERROR_NO_MEMORY = -2,
  ERROR_INVALID = -3
} Status;
Status processData(int *data) {
  if (data == NULL)
    return ERROR_NULL_PTR;
  return OK;
}
int main() {
  Status s = processData(NULL);
  if (s == OK)
    printf("Success\n");
  else
    printf("Error code: %d\n", s);
  return 0;
}
```

Output:

```
Error code: -1
```

Meaningful error codes with enum

Program 18: Union for IP Address

```c
#include <stdio.h>
union IPAddress {
  unsigned int addr;
  unsigned char octets[4];
};
int main() {
  union IPAddress ip;
  ip.octets[0] = 192;
  ip.octets[1] = 168;
  ip.octets[2] = 1;
  ip.octets[3] = 1;
  printf("IP: %u.%u.%u.%u\n",
    ip.octets[0], ip.octets[1],
    ip.octets[2], ip.octets[3]);
  printf("As integer: %u\n", ip.addr);
  return 0;
}
```

Output:

```
IP: 192.168.1.1
As integer: 16885952
```

Access as octets or full integer

Program 19: Enum Boolean Type

```c
#include <stdio.h>
typedef enum { FALSE = 0, TRUE = 1 } Bool;
Bool isEven(int n) {
  return (n % 2 == 0) ? TRUE : FALSE;
}
int main() {
  int num = 10;
  Bool result = isEven(num);
  if (result == TRUE)
    printf("%d is even\n", num);
  else
    printf("%d is odd\n", num);
  num = 7;
  if (isEven(num))
    printf("%d is even\n", num);
  else
    printf("%d is odd\n", num);
  return 0;
}
```

Output:

```
10 is even
7 is odd
```

Custom boolean type before C99

Program 20: Complex Tagged Union Example

```c
#include <stdio.h>
enum DataType { TYPE_INT, TYPE_FLOAT, TYPE_STR };
struct Variant {
  enum DataType type;
  union {
    int i;
    float f;
    char *s;
  } value;
};
void printVariant(struct Variant v) {
  switch(v.type) {
    case TYPE_INT: printf("Int: %d\n", v.value.i); break;
    case TYPE_FLOAT: printf("Float: %.2f\n", v.value.f); break;
    case TYPE_STR: printf("String: %s\n", v.value.s); break;
  }
}
int main() {
  struct Variant v1={TYPE_INT, {.i=42}};
  struct Variant v2={TYPE_FLOAT, {.f=3.14}};
  struct Variant v3={TYPE_STR, {.s="Hello"}};
  printVariant(v1); printVariant(v2); printVariant(v3);
  return 0;
}
```

Output:

```
Int: 42
Float: 3.14
String: Hello
```

Variant type for different data

When to Use Enum

- Defining named constants (days, months, states)
- Menu options and user choices
- Error codes and status values
- State machine states
- Configuration flags
- Direction or position constants
- Makes code self-documenting
- Better than magic numbers

When to Use Union

- Only one member needed at a time
- Memory-constrained environments
- Type punning (viewing data as different types)
- Implementing variant types
- Hardware register access
- Protocol message parsing
- Saving memory in large arrays
- Always use with a tag to track active member

Key Takeaways

- **Enum**: Named integer constants for readability
- Enum values auto-increment from 0 or custom start
- **Union**: Members share same memory location
- Union size = size of largest member
- Only one union member valid at a time
- Tagged union combines enum and union safely
- Enum improves code maintainability
- Union saves memory but requires care
- Both make code more expressive
- Understanding both is important for systems programming