Typedef and Const

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Basic Typedef

Program 1:

```c
#include <stdio.h>
typedef int Integer;
typedef float Real;
typedef char Character;
int main() {
  Integer x = 10;
  Real y = 3.14;
  Character c = 'A';
  printf("Integer: %d\n", x);
  printf("Real: %.2f\n", y);
  printf("Character: %c\n", c);
  return 0;
}
```

Output:

```
Integer: 10
Real: 3.14
Character: A
```

Note:

```
typedef creates type aliases. Makes
code more readable and portable.
Integer is now synonym for int.
No runtime overhead.
```

Typedef with Struct

Program 2:

```c
#include <stdio.h>
typedef struct {
  int x;
  int y;
} Point;
typedef struct {
  Point top_left;
  Point bottom_right;
} Rectangle;
int main() {
  Point p = {10, 20};
  Rectangle r = {{0, 0}, {100, 50}};
  printf("Point: (%d, %d)\n", p.x, p.y);
  printf("Rectangle: (%d,%d) to (%d,%d)\n",
    r.top_left.x, r.top_left.y,
    r.bottom_right.x, r.bottom_right.y);
  return 0;
}
```

Output:

```
Point: (10, 20)
Rectangle: (0,0) to (100,50)
```

Note:

```
typedef with struct eliminates need
for "struct" keyword. Cleaner syntax.
Point instead of struct Point.
Common C idiom.
```

Typedef with Pointers

Program 3:

```c
#include <stdio.h>
#include <stdlib.h>
typedef int* IntPtr;
typedef char* String;
int main() {
  IntPtr p = malloc(sizeof(int));
  *p = 42;
  printf("Value: %d\n", *p);
  String s = "Hello";
  printf("String: %s\n", s);
  free(p);
  return 0;
}
```

Output:

```
Value: 42
String: Hello
```

Note:

```
typedef for pointer types. IntPtr
hides pointer syntax. String is
common typedef for char*.
Be careful: can hide complexity.
```

Typedef with Arrays

Program 4:

```c
#include <stdio.h>
typedef int Vector[3];
typedef char Name[50];
int main() {
  Vector v = {1, 2, 3};
  Name name = "Alice";
  int i;
  printf("Vector: ");
  for (i = 0; i < 3; i++) {
    printf("%d ", v[i]);
  }
  printf("\nName: %s\n", name);
  return 0;
}
```

Output:

```
Vector: 1 2 3
Name: Alice
```

Note:

```
typedef for fixed-size arrays.
Vector is array of 3 ints. Name is
array of 50 chars. Size part of type.
Ensures consistent dimensions.
```

Typedef with Function Pointers

Program 5:

```c
#include <stdio.h>
typedef int (*BinaryOp)(int, int);
int add(int a, int b) { return a + b; }
int multiply(int a, int b) {
  return a * b;
}
void apply(int x, int y, BinaryOp op) {
  printf("Result: %d\n", op(x, y));
}
int main() {
  BinaryOp op1 = add;
  BinaryOp op2 = multiply;
  apply(5, 3, op1);
  apply(5, 3, op2);
  return 0;
}
```

Output:

```
Result: 8
Result: 15
```

Note:

typedef makes function pointers readable. BinaryOp instead of int (*)(int, int). Essential for clean callback code.

Typedef with Enum

Program 6:

```c
#include <stdio.h>
typedef enum {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY
} Day;
const char* day_name(Day d) {
  const char* names[] = {
    "Mon", "Tue", "Wed", "Thu",
    "Fri", "Sat", "Sun"
  };
  return names[d];
}
int main() {
  Day today = WEDNESDAY;
  printf("Today: %s\n", day_name(today));
  return 0;
}
```

Output:

```
Today: Wed
```

Note:

```
typedef with enum eliminates "enum"
keyword. Day instead of enum Day.
Type-safe named constants.
Self-documenting code.
```

Const Variables

Program 7:

```c
#include <stdio.h>
int main() {
  const int MAX = 100;
  const float PI = 3.14159;
  printf("MAX: %d\n", MAX);
  printf("PI: %.5f\n", PI);
  return 0;
}
```

Output:

```
MAX: 100
PI: 3.14159
```

Note:

```
const makes variable read-only.
Compiler enforces immutability.
Better than #define for type safety.
Scope rules apply.
```

Const Pointers

Program 8:

```c
#include <stdio.h>
int main() {
  int x = 10, y = 20;
  const int *p1 = &x;
  int *const p2 = &x;
  const int *const p3 = &x;
  printf("*p1: %d\n", *p1);
  p1 = &y;
  printf("*p1: %d\n", *p1);
  *p2 = 30;
  printf("*p2: %d\n", *p2);
  printf("*p3: %d\n", *p3);
  return 0;
}
```

Output:

```
*p1: 10
*p1: 20
*p2: 30
*p3: 10
```

Note:

```
const int *p: pointer to const int
  (can change pointer, not data)
int *const p: const pointer to int
  (can change data, not pointer)
const int *const: both const
```

Const Function Parameters

Program 9:

```c
#include <stdio.h>
#include <string.h>
void print_string(const char *s) {
  printf("%s\n", s);
}
int string_length(const char *s) {
  int len = 0;
  while (*s++) len++;
  return len;
}
int main() {
  const char *msg = "Hello";
  print_string(msg);
  printf("Length: %d\n",
    string_length(msg));
  return 0;
}
```

Output:

```
Hello
Length: 5
```

Note:

```
const parameters promise not to
modify data. Documents intent.
Enables compiler optimizations.
Common for string functions.
```

Const Arrays

Program 10:

```c
#include <stdio.h>
int main() {
  const int primes[] = {2, 3, 5, 7, 11};
  const char* days[] = {
    "Mon", "Tue", "Wed", "Thu", "Fri"
  };
  int i;
  printf("Primes: ");
  for (i = 0; i < 5; i++) {
    printf("%d ", primes[i]);
  }
  printf("\nDays: ");
  for (i = 0; i < 5; i++) {
    printf("%s ", days[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Primes: 2 3 5 7 11
Days: Mon Tue Wed Thu Fri
```

Note:

```
const array elements cannot be
modified. Compiler error if attempted.
Prevents accidental changes to
lookup tables or constants.
```

Const Struct Members

Program 11:

```c
#include <stdio.h>
typedef struct {
  const int id;
  char name[20];
} Student;
int main() {
  Student s = {101, "Alice"};
  printf("ID: %d\n", s.id);
  printf("Name: %s\n", s.name);
  s.name[0] = 'B';
  printf("Name: %s\n", s.name);
  return 0;
}
```

Output:

```
ID: 101
Name: Alice
Name: Blice
```

Note:

```
const struct member is read-only.
Must be initialized. Cannot change
after creation. Useful for immutable
identifiers.
```

Const Return Values

Program 12:

```c
#include <stdio.h>
const char* get_message() {
  return "Hello, World!";
}
const int* get_primes() {
  static const int primes[] = {
    2, 3, 5, 7, 11
  };
  return primes;
}
int main() {
  const char* msg = get_message();
  const int* p = get_primes();
  printf("%s\n", msg);
  printf("First prime: %d\n", p[0]);
  return 0;
}
```

Output:

```
Hello, World!
First prime: 2
```

Note:

```
const return type prevents caller
from modifying returned data.
Protects internal data structures.
Documents function contract.
```

Typedef vs Define

Program 13:

```c
#include <stdio.h>
#define INT_DEFINE int
typedef int INT_TYPEDEF;
#define PTR_DEFINE int*
typedef int* PTR_TYPEDEF;
int main() {
  INT_DEFINE a, b;
  INT_TYPEDEF c, d;
  PTR_DEFINE p1, p2;
  PTR_TYPEDEF p3, p4;
  printf("sizeof(a): %lu\n", sizeof(a));
  printf("sizeof(p1): %lu\n", sizeof(p1));
  printf("sizeof(p2): %lu\n", sizeof(p2));
  printf("sizeof(p3): %lu\n", sizeof(p3));
  printf("sizeof(p4): %lu\n", sizeof(p4));
  return 0;
}
```

Output:

```
sizeof(a): 4
sizeof(p1): 8
sizeof(p2): 4
sizeof(p3): 8
sizeof(p4): 8
```

Note:

```
#define is text replacement.
PTR_DEFINE p1, p2 becomes int* p1, p2
(p1 is pointer, p2 is int!)
typedef is true type alias.
PTR_TYPEDEF makes both pointers.
```

Complex Typedef

Program 14:

```c
#include <stdio.h>
typedef int (*FuncArray[5])(int, int);
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int divide(int a, int b) {
  return b ? a / b : 0;
}
int mod(int a, int b) {
  return b ? a % b : 0;
}
int main() {
  FuncArray ops = {add, sub, mul,
    divide, mod};
  int i;
  for (i = 0; i < 5; i++) {
    printf("%d ", ops[i](10, 3));
  }
  printf("\n");
  return 0;
}
```

Output:

```
13 7 30 3 1
```

Note:

```
Complex typedef: array of function
pointers. FuncArray is array of 5
function pointers. Each takes two
ints, returns int. Simplifies syntax.
```

Typedef for Portability

Program 15:

```c
#include <stdio.h>
#include <stdint.h>
typedef uint8_t  Byte;
typedef uint16_t Word;
typedef uint32_t DWord;
typedef int32_t  SignedInt;
int main() {
  Byte b = 255;
  Word w = 65535;
  DWord d = 4294967295U;
  SignedInt si = -123456;
  printf("Byte: %u\n", b);
  printf("Word: %u\n", w);
  printf("DWord: %u\n", d);
  printf("SignedInt: %d\n", si);
  return 0;
}
```

Output:

```
Byte: 255
Word: 65535
DWord: 4294967295
SignedInt: -123456
```

Note:

typedef for platform-independent types. Fixed-size types from stdint.h. Ensures portability across systems. Common in embedded programming.

Const Correctness Example

Program 16:

```c
#include <stdio.h>
void modify_array(int *arr, int n) {
  int i;
  for (i = 0; i < n; i++) {
    arr[i] *= 2;
  }
}
void print_array(const int *arr, int n) {
  int i;
  for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}
int main() {
  int nums[] = {1, 2, 3, 4, 5};
  print_array(nums, 5);
  modify_array(nums, 5);
  print_array(nums, 5);
  return 0;
}
```

Output:

```
1 2 3 4 5
2 4 6 8 10
```

Note:

```
const correctness: print_array uses
const to promise no modification.
modify_array uses non-const to show
it changes data. Self-documenting.
```

Typedef with Union

Program 17:

```c
#include <stdio.h>
typedef union {
  int i;
  float f;
  char c[4];
} Data;
int main() {
  Data d;
  d.i = 0x41424344;
  printf("As int: %d\n", d.i);
  printf("As float: %f\n", d.f);
  printf("As chars: %c%c%c%c\n",
    d.c[0], d.c[1], d.c[2], d.c[3]);
  return 0;
}
```

Output:

```
As int: 1094861636
As float: 808.765625
As chars: DCBA
```

Note:

typedef with union. Data can hold int, float, or char array. Same memory interpreted differently. Type punning example.

Nested Typedef

Program 18:

```c
#include <stdio.h>
typedef struct {
  int x;
  int y;
} Point;
typedef struct {
  Point center;
  int radius;
} Circle;
typedef struct {
  Circle circles[3];
  int count;
} Drawing;
int main() {
  Drawing d = {
    {{{10, 20}, 5}, {{30, 40}, 10},
     {{50, 60}, 15}},
    3
  };
  int i;
  for (i = 0; i < d.count; i++) {
    printf("Circle %d: center(%d,%d) r=%d\n",
      i, d.circles[i].center.x,
      d.circles[i].center.y,
      d.circles[i].radius);
  }
  return 0;
}
```

Output:

```
Circle 0: center(10,20) r=5
Circle 1: center(30,40) r=10
Circle 2: center(50,60) r=15
```

Note:

Nested typedef structures. Point used in Circle, Circle used in Drawing. Clean hierarchical types. Reusable components.

Const with Typedef

Program 19:

```c
#include <stdio.h>
typedef const int ConstInt;
typedef const char* ConstString;
typedef struct {
  ConstInt id;
  char name[20];
} Record;
int main() {
  ConstInt max = 100;
  ConstString msg = "Hello";
  Record r = {1, "Alice"};
  printf("Max: %d\n", max);
  printf("Message: %s\n", msg);
  printf("Record: %d, %s\n", r.id,
    r.name);
  return 0;
}
```

Output:

```
Max: 100
Message: Hello
Record: 1, Alice
```

Note:

Combine typedef with const. ConstInt is always const int. ConstString is pointer to const char. Enforces immutability at type level.

Real World Example

Program 20:

```c
#include <stdio.h>
typedef unsigned int Size;
typedef const char* String;
typedef enum { FALSE, TRUE } Bool;
typedef struct {
  String name;
  Size age;
  Bool is_student;
} Person;
void print_person(const Person *p) {
  printf("Name: %s\n", p->name);
  printf("Age: %u\n", p->age);
  printf("Student: %s\n",
    p->is_student ? "Yes" : "No");
}
int main() {
  Person people[] = {
    {"Alice", 25, TRUE},
    {"Bob", 30, FALSE}
  };
  Size i;
  for (i = 0; i < 2; i++) {
    print_person(&people[i]);
    printf("---\n");
  }
  return 0;
}
```

Output:

```
Name: Alice
Age: 25
Student: Yes
---
Name: Bob
Age: 30
Student: No
---
```

Note:

```
Complete example combining typedef
for Size, String, Bool, struct.
Const parameters for safety.
Clean, self-documenting code.
```