Function Pointers

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Basic Function Pointer

Program 1:

```c
#include <stdio.h>
int add(int a, int b) {
  return a + b;
}
int main() {
  int (*func_ptr)(int, int);
  func_ptr = &add;
  int result = func_ptr(5, 3);
  printf("Result: %d\n", result);
  result = (*func_ptr)(10, 20);
  printf("Result: %d\n", result);
  return 0;
}
```

Output:

```
Result: 8
Result: 30
```

Note:

```
Function pointer syntax:
return_type (*ptr_name)(params);
Can call with func_ptr() or
(*func_ptr)(). Both work. & optional
when assigning function address.
```

Function Pointer as Parameter

Program 2:

```c
#include <stdio.h>
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int multiply(int a, int b) {
  return a * b;
}
void compute(int x, int y,
  int (*op)(int, int)) {
  printf("Result: %d\n", op(x, y));
}
int main() {
  compute(10, 5, add);
  compute(10, 5, sub);
  compute(10, 5, multiply);
  return 0;
}
```

Output:

```
Result: 15
Result: 5
Result: 50
```

Note:

Function pointers enable passing functions as arguments. Different operations with same function. Callback pattern foundation.

Array of Function Pointers

Program 3:

```c
#include <stdio.h>
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 main() {
  int (*ops[4])(int, int) = {
    add, sub, mul, divide
  };
  char* names[] = {"+", "-", "*", "/"};
  int i;
  for (i = 0; i < 4; i++) {
    printf("10 %s 5 = %d\n",
      names[i], ops[i](10, 5));
  }
  return 0;
}
```

Output:

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
```

Note:

```
Array of function pointers.
Syntax: return_type (*arr[])(params)
Index into array to select function.
Useful for dispatch tables.
```

Calculator with Function Pointers

Program 4:

```c
#include <stdio.h>
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 main() {
  int (*ops[4])(int, int) = {
    add, sub, mul, divide
  };
  int choice, a, b;
  printf("0:+ 1:- 2:* 3:/\n");
  printf("Enter choice: ");
  scanf("%d", &choice);
  printf("Enter two numbers: ");
  scanf("%d %d", &a, &b);
  if (choice >= 0 && choice < 4) {
    printf("Result: %d\n",
      ops[choice](a, b));
  }
  return 0;
}
```

Output:

```
0:+ 1:- 2:* 3:/
Enter choice: 2
Enter two numbers: 7 8
Result: 56
```

Note:

```
Menu-driven calculator using
function pointer array. User choice
selects which function to call.
Cleaner than switch statement.
```

Callback Functions

Program 5:

```c
#include <stdio.h>
void process_array(int arr[], int n,
  void (*callback)(int)) {
  int i;
  for (i = 0; i < n; i++) {
    callback(arr[i]);
  }
}
void print_square(int x) {
  printf("%d ", x * x);
}
void print_double(int x) {
  printf("%d ", x * 2);
}
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  printf("Squares: ");
  process_array(arr, 5, print_square);
  printf("\nDoubles: ");
  process_array(arr, 5, print_double);
  printf("\n");
  return 0;
}
```

Output:

```
Squares: 1 4 9 16 25
Doubles: 2 4 6 8 10
```

Note:

```
Callback pattern: function accepts
another function to call back.
Same processing logic, different
actions via callback.
```

Returning Function Pointers

Program 6:

```c
#include <stdio.h>
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int (*get_operation(char op))(int, int) {
  if (op == '+') return add;
  if (op == '-') return sub;
  return NULL;
}
int main() {
  int (*op)(int, int);
  op = get_operation('+');
  if (op) {
    printf("10 + 5 = %d\n", op(10, 5));
  }
  op = get_operation('-');
  if (op) {
    printf("10 - 5 = %d\n", op(10, 5));
  }
  return 0;
}
```

Output:

```
10 + 5 = 15
10 - 5 = 5
```

Note:

```
Function can return function pointer.
Syntax: return_type (*func())(params)
Factory pattern: select and return
appropriate function.
```

Typedef with Function Pointers

Program 7:

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

Output:

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

Note:

typedef makes function pointer syntax cleaner. Operation is now a type name. Much more readable than raw function pointer syntax.

qsort with Function Pointers

Program 8:

```c
#include <stdio.h>
#include <stdlib.h>
int compare_asc(const void *a,
  const void *b) {
  return (*(int*)a - *(int*)b);
}
int compare_desc(const void *a,
  const void *b) {
  return (*(int*)b - *(int*)a);
}
int main() {
  int arr[] = {5, 2, 8, 1, 9};
  int i, n = 5;
  qsort(arr, n, sizeof(int), compare_asc);
  printf("Ascending: ");
  for (i = 0; i < n; i++)
    printf("%d ", arr[i]);
  qsort(arr, n, sizeof(int),
    compare_desc);
  printf("\nDescending: ");
  for (i = 0; i < n; i++)
    printf("%d ", arr[i]);
  printf("\n");
  return 0;
}
```

Output:

```
Ascending: 1 2 5 8 9
Descending: 9 8 5 2 1
```

Note:

qsort accepts comparison function pointer. void* for generic sorting. Compare function returns negative, zero, or positive for <, ==, >.

Sorting Structs with qsort

Program 9:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
  char name[20];
  int age;
} Person;
int compare_age(const void *a,
  const void *b) {
  return ((Person*)a)->age -
    ((Person*)b)->age;
}
int main() {
  Person p[] = {
    {"Alice", 25}, {"Bob", 20},
    {"Charlie", 30}
  };
  int i, n = 3;
  qsort(p, n, sizeof(Person),
    compare_age);
  for (i = 0; i < n; i++) {
    printf("%s: %d\n", p[i].name,
      p[i].age);
  }
  return 0;
}
```

Output:

```
Bob: 20
Alice: 25
Charlie: 30
```

Note:

```
qsort works with any data type.
Cast void* to appropriate type in
compare function. Access struct
members for comparison.
```

Function Pointer in Struct

Program 10:

```c
#include <stdio.h>
typedef struct {
  int (*add)(int, int);
  int (*multiply)(int, int);
} Calculator;
int add(int a, int b) { return a + b; }
int multiply(int a, int b) {
  return a * b;
}
int main() {
  Calculator calc;
  calc.add = add;
  calc.multiply = multiply;
  printf("Add: %d\n", calc.add(5, 3));
  printf("Multiply: %d\n",
    calc.multiply(5, 3));
  return 0;
}
```

Output:

```
Add: 8
Multiply: 15
```

Note:

```
Function pointers as struct members.
Object-oriented style in C.
Struct bundles data and operations.
Foundation for virtual functions.
```

State Machine with Function Pointers

Program 11:

```c
#include <stdio.h>
typedef void (*State)();
void state_idle() {
  printf("IDLE state\n");
}
void state_running() {
  printf("RUNNING state\n");
}
void state_stopped() {
  printf("STOPPED state\n");
}
int main() {
  State states[] = {
    state_idle, state_running,
    state_stopped
  };
  int current = 0;
  int i;
  for (i = 0; i < 3; i++) {
    states[current]();
    current = (current + 1) % 3;
  }
  return 0;
}
```

Output:

```
IDLE state
RUNNING state
STOPPED state
```

Note:

```
State machine using function pointer
array. Each state is a function.
Transition by changing index.
Common embedded systems pattern.
```

Filter with Predicate Function

Program 12:

```c
#include <stdio.h>
int is_even(int x) { return x % 2 == 0; }
int is_positive(int x) { return x > 0; }
void filter(int arr[], int n,
  int (*predicate)(int)) {
  int i;
  for (i = 0; i < n; i++) {
    if (predicate(arr[i])) {
      printf("%d ", arr[i]);
    }
  }
  printf("\n");
}
int main() {
  int arr[] = {-2, 3, -5, 4, 6, -1};
  printf("Even: ");
  filter(arr, 6, is_even);
  printf("Positive: ");
  filter(arr, 6, is_positive);
  return 0;
}
```

Output:

```
Even: -2 4 6
Positive: 3 4 6
```

Note:

```
Predicate function returns boolean.
Filter calls predicate to decide
inclusion. Functional programming
pattern in C.
```

Map Function

Program 13:

```c
#include <stdio.h>
int square(int x) { return x * x; }
int cube(int x) { return x * x * x; }
void map(int arr[], int n,
  int (*transform)(int)) {
  int i;
  for (i = 0; i < n; i++) {
    printf("%d ", transform(arr[i]));
  }
  printf("\n");
}
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  printf("Squares: ");
  map(arr, 5, square);
  printf("Cubes: ");
  map(arr, 5, cube);
  return 0;
}
```

Output:

```
Squares: 1 4 9 16 25
Cubes: 1 8 27 64 125
```

Note:

```
Map applies transformation function
to each element. Higher-order
function pattern. Transform logic
separated from iteration.
```

Reduce Function

Program 14:

```c
#include <stdio.h>
int add(int a, int b) { return a + b; }
int multiply(int a, int b) {
  return a * b;
}
int max(int a, int b) {
  return a > b ? a : b;
}
int reduce(int arr[], int n, int init,
  int (*op)(int, int)) {
  int i, result = init;
  for (i = 0; i < n; i++) {
    result = op(result, arr[i]);
  }
  return result;
}
int main() {
  int arr[] = {1, 2, 3, 4, 5};
  printf("Sum: %d\n",
    reduce(arr, 5, 0, add));
  printf("Product: %d\n",
    reduce(arr, 5, 1, multiply));
  printf("Max: %d\n",
    reduce(arr, 5, arr[0], max));
  return 0;
}
```

Output:

```
Sum: 15
Product: 120
Max: 5
```

Note:

```
Reduce combines all elements using
binary operation. Accumulates result.
Different operations: sum, product,
max, etc. Fold pattern.
```

Event Handler Pattern

Program 15:

```c
#include <stdio.h>
typedef void (*EventHandler)(int);
void on_click(int x) {
  printf("Click at %d\n", x);
}
void on_hover(int x) {
  printf("Hover at %d\n", x);
}
void trigger_event(int pos,
  EventHandler handler) {
  if (handler) {
    handler(pos);
  }
}
int main() {
  trigger_event(10, on_click);
  trigger_event(20, on_hover);
  trigger_event(30, NULL);
  return 0;
}
```

Output:

```
Click at 10
Hover at 20
```

Note:

```
Event handler pattern using function
pointers. Register handlers for
events. Trigger calls handler.
GUI programming foundation.
```

Command Pattern

Program 16:

```c
#include <stdio.h>
typedef void (*Command)();
void save() { printf("Saving...\n"); }
void load() { printf("Loading...\n"); }
void quit() { printf("Quitting...\n"); }
typedef struct {
  char name[10];
  Command cmd;
} MenuItem;
int main() {
  MenuItem menu[] = {
    {"Save", save},
    {"Load", load},
    {"Quit", quit}
  };
  int i;
  for (i = 0; i < 3; i++) {
    printf("%d. %s\n", i+1, menu[i].name);
  }
  printf("Choice: 1\n");
  menu[0].cmd();
  return 0;
}
```

Output:

```
1. Save
2. Load
3. Quit
Choice: 1
Saving...
```

Note:

```
Command pattern: bind name to
action. Menu system using struct
with function pointer. Extensible
without switch statements.
```

Strategy Pattern

Program 17:

```c
#include <stdio.h>
typedef int (*SortStrategy)(int, int);
int ascending(int a, int b) {
  return a > b;
}
int descending(int a, int b) {
  return a < b;
}
void sort(int arr[], int n,
  SortStrategy cmp) {
  int i, j;
  for (i = 0; i < n-1; i++) {
    for (j = 0; j < n-i-1; j++) {
      if (cmp(arr[j], arr[j+1])) {
        int temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
}
int main() {
  int arr[] = {5, 2, 8, 1, 9};
  int i;
  sort(arr, 5, ascending);
  for (i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");
```

Output:

```
1 2 5 8 9
```

Note:

```
Strategy pattern: algorithm selected
at runtime via function pointer.
Same sort code, different comparison
strategies. Bubble sort here.
```

Function Composition

Program 18:

```c
#include <stdio.h>
int add_five(int x) { return x + 5; }
int multiply_two(int x) { return x * 2; }
int square(int x) { return x * x; }
int compose(int x,
  int (*f)(int), int (*g)(int)) {
  return f(g(x));
}
int main() {
  int x = 3;
  int result;
  result = compose(x, multiply_two,
    add_five);
  printf("(3+5)*2 = %d\n", result);
  result = compose(x, square,
    multiply_two);
  printf("(3*2)^2 = %d\n", result);
  return 0;
}
```

Output:

```
(3+5)*2 = 16
(3*2)^2 = 36
```

Note:

```
Function composition: combine
functions. compose(f, g, x) returns
f(g(x)). Build complex operations
from simple ones.
```

Plugin System

Program 19:

```c
#include <stdio.h>
typedef void (*Plugin)(const char*);
void plugin_upper(const char *s) {
  printf("UPPER: ");
  while (*s) {
    printf("%c", (*s >= 'a' && *s <= 'z')
      ? *s - 32 : *s);
    s++;
  }
  printf("\n");
}
void plugin_reverse(const char *s) {
  int len = 0;
  const char *p = s;
  while (*p++) len++;
  printf("REVERSE: ");
  while (len--) printf("%c", s[len]);
  printf("\n");
}
void execute_plugin(Plugin p,
  const char *data) {
  p(data);
}
int main() {
  execute_plugin(plugin_upper, "hello");
  execute_plugin(plugin_reverse, "world");
  return 0;
}
```

Output:

```
UPPER: HELLO
REVERSE: dlrow
```

Note:

Plugin architecture: load and execute different plugins via function pointers. Extensible system without recompilation.

Observer Pattern

Program 20:

```c
#include <stdio.h>
typedef void (*Observer)(int);
Observer observers[5];
int observer_count = 0;
void register_observer(Observer obs) {
  if (observer_count < 5) {
    observers[observer_count++] = obs;
  }
}
void notify_all(int value) {
  int i;
  for (i = 0; i < observer_count; i++) {
    observers[i](value);
  }
}
void logger(int v) {
  printf("Log: value=%d\n", v);
}
void alert(int v) {
  if (v > 100)
    printf("Alert: high value!\n");
}
int main() {
  register_observer(logger);
  register_observer(alert);
  notify_all(50);
  notify_all(150);
  return 0;
}
```

Output:

```
Log: value=50
Log: value=150
Alert: high value!
```

Note:

```
Observer pattern: register callbacks
that get notified on events. All
observers called when state changes.
Publish-subscribe pattern.
```