Function Pointers to implement a branch table

• Functions in a program occupy memory.
• The name of the function is a pointer to its first byte of memory.
• Pointer variables may hold the address of a function and be used to invoke the function indirectly.
• A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
Function Pointers

• Consider functions *main*, *fun1*, and *fun2*
Declaring a pointer to a function

The general syntax:

```
return_type (*fp_name)(parameter-list)
```

Examples:

```
void (*f1)();

double (*f2)(int, double);
```
Initializing a pointer to a function

A pointer to a function must be initialized to point to something before it can be used:

```c
void fun1();
void (*pf) () = &fun1;  // the & is optional, but it doesn't hurt

double fun2(int, float);
double (*f2) (int, float) = fun2;

OR

void fun1();
void (*f1) ();

double fun2(int, double);
double (*f2) (int, double);

f1 = fun1;
f2 = fun2;
```
Calling the function

After the function has been declared and initialized, there are three ways to call the function:

fun1();       // calls the function fun1 by name
(*f1) ();     // apply indirection to f1
f1();         // indirection is not needed, because the
              // compiler wants a pointer to the function

double x;
x = fun2(7, 5.5);
x = (*f2) (7, 5.5);
x = f2(7, 5.5)
Example

/* fnptr.c */
int adder(int a, int b) {
    return(a + b);
}
main() {
    int (*ptrf)(int, int);   // declare pointer to function
    int sum;
    ptrf = adder;           // point it to adder (note no & is needed, but it
                             // doesn’t hurt)
    sum = ptrf(3, 4);       // invoke it. (or use sum = (*ptrf)(3,4)
                            // parens required.

    printf("sum = %d \n", sum);
}

./a.out
sum = 7
Uses for function pointers

- to pass a function pointer as an argument in a function call.

```c
/** Function pointers as parameters to functions **/
#include <stdio.h>

int add(int a, int b);
int multiply(int a, int b);
int compute(int (*fnp)(int, int), int, int);

int main()
{
    int answer;
    printf("\npassing a function pointer\n");
    answer = compute(add, 6, 4);
    printf("sum = %d\n", answer);
    answer = compute(multiply, 6, 4);
    printf("product = %d\n", answer);
    return 0;
}
```
Uses for function pointers

/* This function computes a result based on the function passed as an argument. */
int compute(int (*fn)(int a, int b), int x, int y)
{
    int answer;
    answer = (*fn)(x, y);

    return answer;
}
/**
   * This function returns the sum of its arguments
   ***/

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

/**
   * This function returns the product of its arguments
   ***/
int multiply(int a, int b)
{
    return (a * b);
}
Function Pointers to implement a branch table

• jump table (branch table), is an array of pointers to functions. see Calculator Example, page 360 of *Pointers on C*

```c
switch(operator)  {
  case ADD:
    result = add(op1, op2)
    break;
  case SUB:
    result = sub(op1, op2)
    break;
  case MUL:
    result = mul(op1, op2)
    break;
  case DIV:
    result = div(op1, op2);
    break;
  ...
```

For a hundred or so operators, the switch statement will become extremely large

```c
double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);
...
double (*op_func []) (double, double) = {add, sub, mul, div, ...};

double result = op_func[operator] (d1, d2);
```