Text Illustrations

To Accompany

Embedded Systems: A Contemporary Design Tool

James K. Peckol, Univ. of Washington
ISBN: 978-0-471-72180-2

Chapter 7 – The Software Side – Part 2. Pointers and Functions

```
// we are working with byte sized pieces in this example

unsigned char a = 0xF3;        // a = 1111 0011 – note this is not a negative number
unsigned char b = 0x54;        // b = 0101 0100 – note this is not a positive number

unsigned char c = a & b;       // c gets a AND b
                               // a 1111 0011
                               // b 0101 0100
                               // c 0101 0000

unsigned char d = a | b;       // d gets a OR b
                               // a 1111 0011
                               // b 0101 0100
                               // d 1111 0111

unsigned char e = a ^ b;       // e gets a XOR b
                               // a 1111 0011
                               // b 0101 0100
                               // e 1010 0111

unsigned char f = ~a;          // f gets ~a
                               // a 1111 0011
                               // f   0000 1100
```

fig_07_00

```
unsigned char testPattern0 = 0x40;          // testPattern0 = 0010 0000
unsigned char setPattern0 = 0x8;            // setPattern0 = 0000 1000

// assume portShadow holds 1010 0110
if (portShadow & testPattern0)              // if bit 5 is set, the AND will give a nonzero result
{
        // set bit3 reset bit 5 and update portShadow
        // portShadow = (1010 0110 & ~(0010 0000)) | (0000 1000)
        // portShadow = (1010 0110 & 1101 1111) | (0000 1000)
        // portShadow = 1000 1110

        portShadow = (portShadow & ~testPattern0 ) | setPattern0;
        setPort(portShadow);
}
```

fig_07_01

```
unsigned char a = 0x3;                                  // a = 0000 0011
unsigned char b = 0xC5;                                 // b = 1100 0101
printf (" a shifted 4 places left is %x\", a << 4);     // prints…0011 0000
printf (" b shifted 2 places right is %x\", b >> 2);    // prints …0011 0001
printf (" a is %x\", a );                               // prints…0000 0011
printf (" b is %x\", b );                               // prints …1100 0101
```

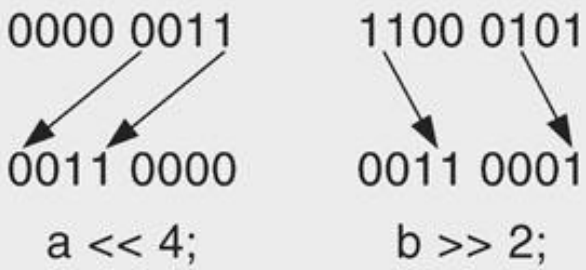


fig_07_02

```
unsigned char bitPattern0 = 0x1;              // bitPattern0 = 0000 0001

// assume portShadow holds 1010 0110
if (portShadow & (bitPattern0 << 5)           // if bit 5 is set, the AND will give a nonzero result
{
        // set bit3 reset bit 5 and update portShadow
        // portShadow = (1010 0110 & ~(0010 0000)) | (0000 1000)
        // portShadow = (1010 0110 & 1101 1111) | (0000 1000)
        // portShadow = 1000 1110

        portShadow = (portShadow & ~(bitPattern0 << 5) ) | (bitPattern0 << 3);
        setPort(portShadow);
}
```

fig_07_03

```
unsigned char getPort(void);                    // port access function prototype

unsigned char testPattern0 = 0x1A;              // testPattern0 = 0001 1010
unsigned char mask = 0x1E;                      // mask = 0001 1110
unsigned char portData = 0x0;                   // working variable

// assume port holds 1101 1011
portData = getPort();                           // read the port

if !((portShadow & mask) ^ testPattern0)        // will give a zero result if pattern present
{
     printf( “pattern present \n”;
}
```

fig_07_04

Multiply by x where x is 2, 4, 8, …

result = number << x;

Divide by y where y is 2, 4, 8, …

result = number >> y;

```
Multiply by x where x is a simple number such as 5, 6, 9, 10, 12,…
result = (number << 2) + number;              // multiply by 5, …0101
result = (number << 2) + (number << 1);   // multiply by 6, …0110
result = (number << 3) + number;              // multiply by 9, …1001
result = (number << 3) + (number << 1);   // multiply by 10, …1010
result = (number << 3) + (number << 2);   // multiply by 12, …1100
```

fig_07_06

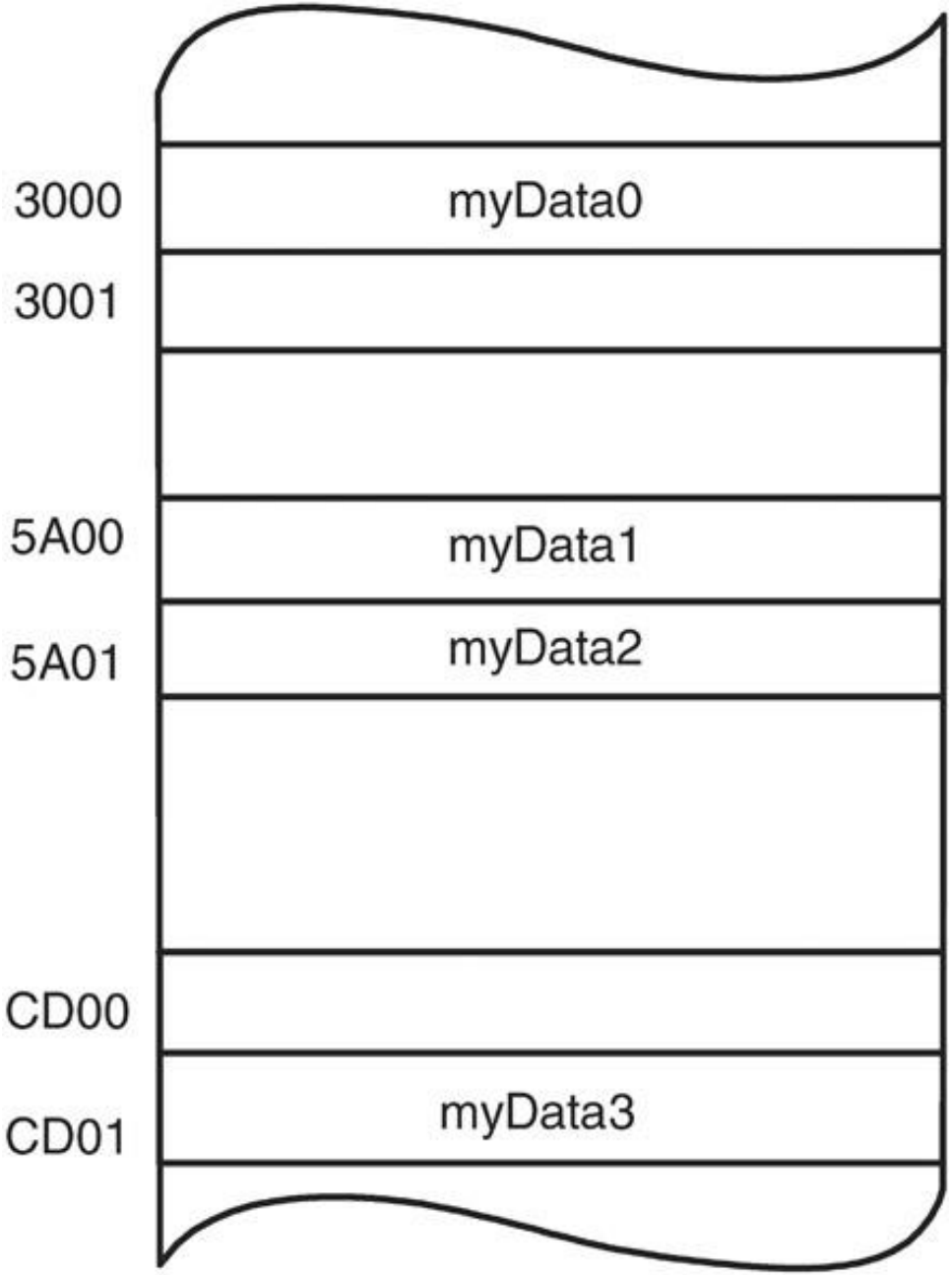

3000
myData0
3001
5A00
myData1
5A01
myData2
CD00
CD01
myData3

fig_07_07

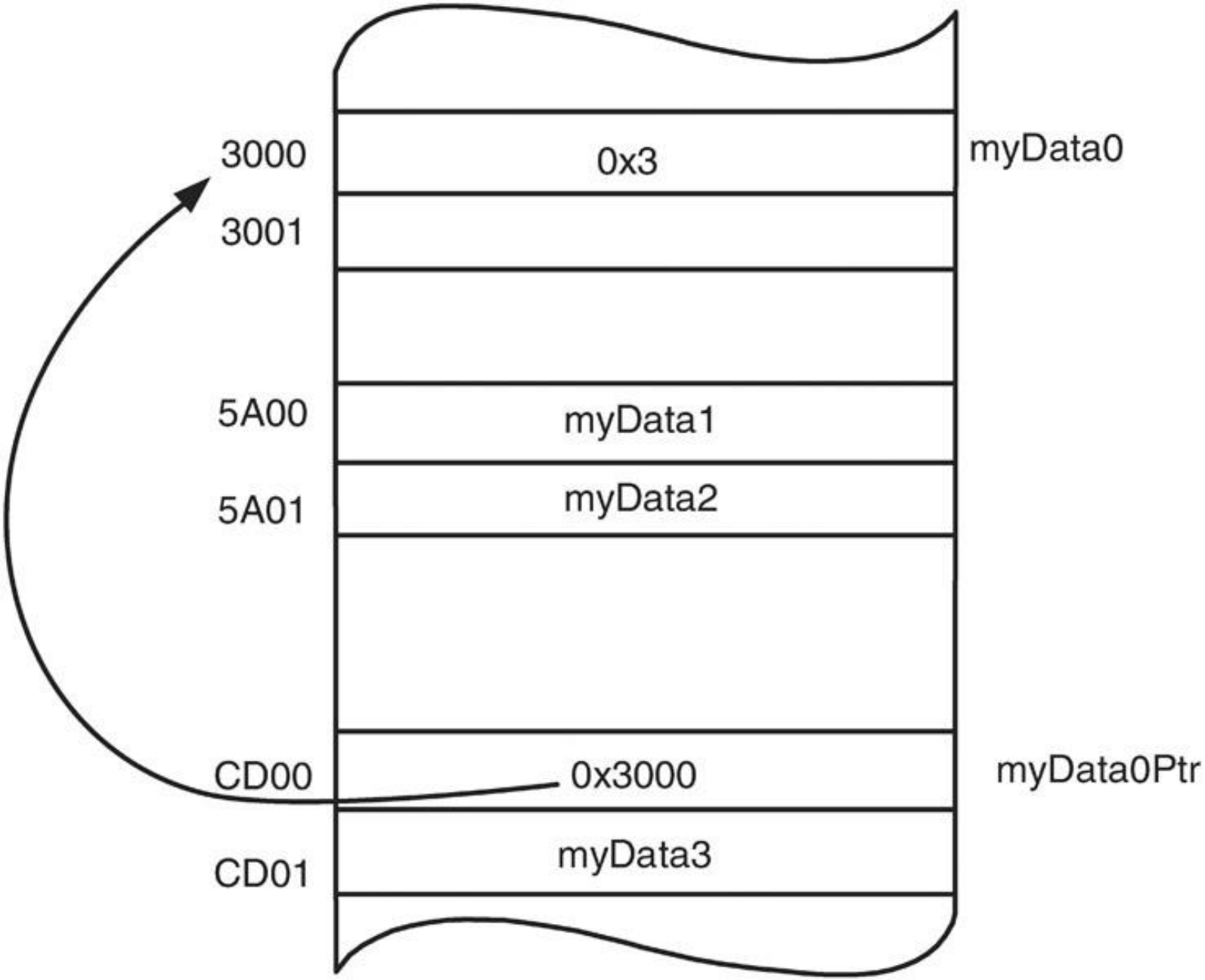
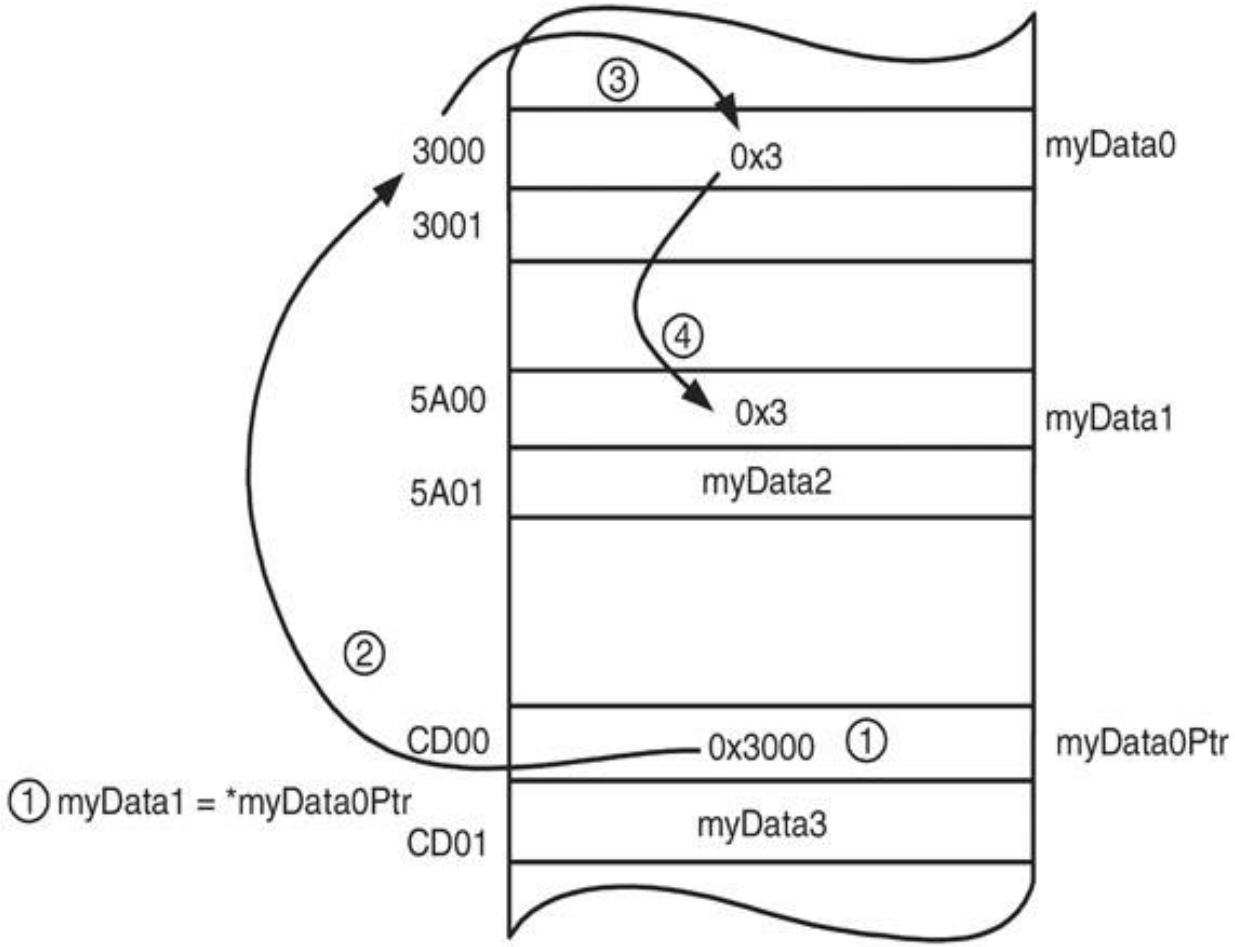

3000
0x3
myData0
3001
5A00
myData1
5A01
myData2
CD00
0x3000
myData0Ptr
CD01
myData3

fig_07_08



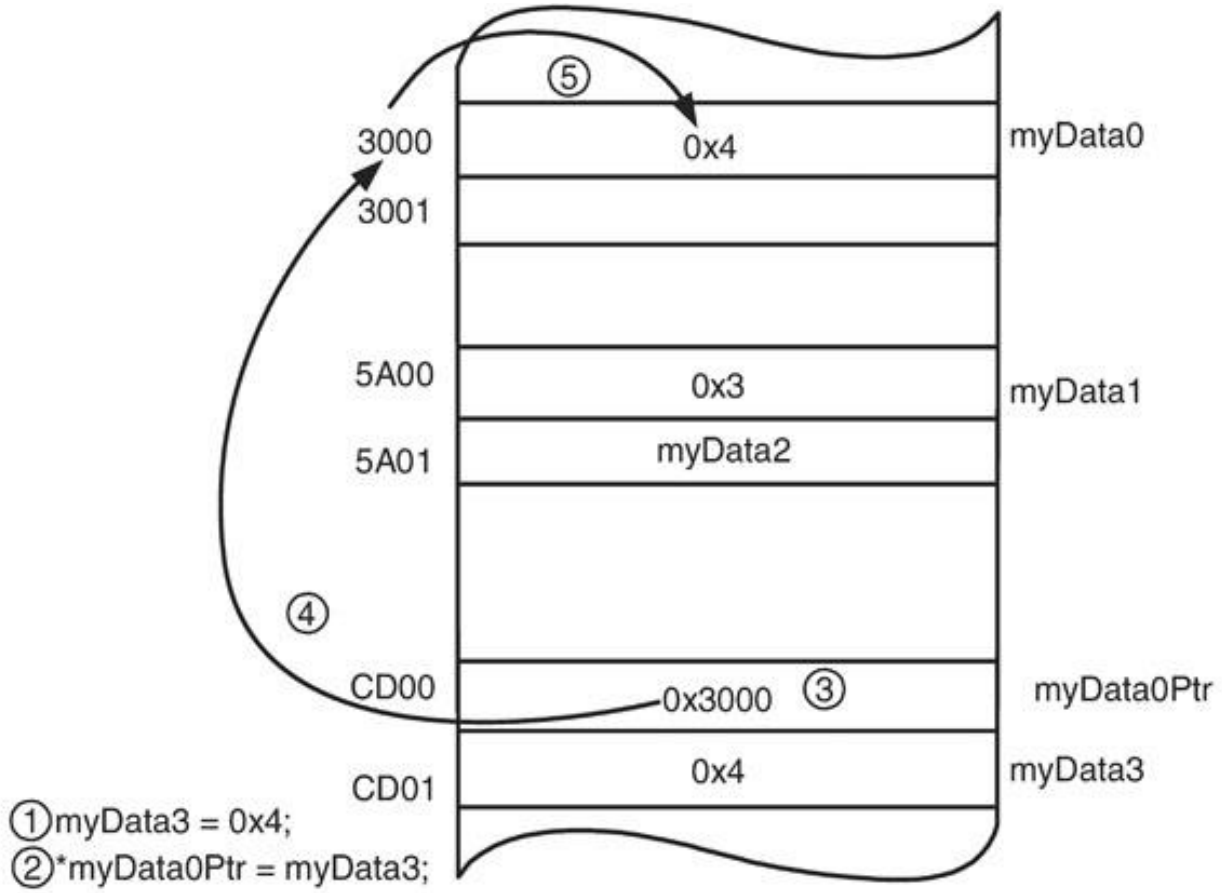
① Get the value contained in the pointer variable *myData0Ptr*, 0x3000.

② Go to the address (0x3000) specified, pointed, or referred to by that value.

③ Get the value of the variable, *myData0*, at memory address 0x3000 – this will be the value 0x3.

④ Assign that value (0x3) to the variable *myData1*.

fig_07_09



① Assign the value 0x4 to the variable *myData*3.

② Get the value contained in the variable *myData3* – 0x4.

③ Get the value of the pointer variable *myData0Ptr*, 0x3000.

④ Go to the address (0x3000) specified or pointed to by that value.

⑤ Assign the value contained in the variable, *myData3* to the variable at memory address 0x3000, *myData0*—this will be the value 0x4.

fig_07_10

```
/*
 * A First Look at Pointers
 */
#include <stdio.h>
void main(void)
{
        int myData0 = 0x3;
        int myData1 = 0;
        int myData2 = 0;
        int myData3 = 0;

        int *myData0Ptr = &myData0;             // myData0Ptr is a pointer to int
                                                // initialized to point to myData0
        myData1 = *myData0Ptr;                  // myData1 now contains the value 3

        printf ("The value of myData1 is: %d\n", *myData0Ptr);
        myData3 = 0x4;                          // myData3 now contains the value 4
        *myData0Ptr = myData3;                  // myData0 now contains the value 4 as well

        printf ("The value of myData3 is: %d\n", *myData0Ptr);
        return;
}
```

fig_07_11

```
int aVal = *myPtr++;
```

Evaluation

1. myPtr will be dereferenced and will return 0x3000 because * is higher precedence than ++.
2. 0x3 will be assigned to aVal.
3. myPtr will be incremented by the size of one integer to 0x3002.

```
int aVal = *(myPtr++);
```

Evaluation

1. The operation inside the parentheses will be evaluated first. myPtr will be evaluated as 0x3000 and this will be the return value from the operation.
2. Before the return, myPtr will be incremented by the size of one integer to 0x3002.
3. The value 0x3000 is returned—the value *before* the increment.
4. 0x3 will be assigned to aVal because this is the value stored at memory location 0x3000.

fig_07_12a

```
int aVal = *myPtr+1;
```

Evaluation

1. myPtr will be evaluated as 0x3000 and dereferenced.
2. The value at memory location 0x3000 will be returned.
3. 1 will be added to the value returned from memory location 0x3000 to yield 0x4.
4. 0x4 will be assigned to aVal.

```
int aVal = *(myPtr+1);
```

Evaluation

1. myPtr will be evaluated as 0x3000.
2. 1 will be added to 0x3000 to give 0x3002.
3. The value at memory location 0x3002 will be returned and assigned to aVal.

char myChar = ‘a’;

Object is Constant		*Pointer is Constant*
char	*	ptr = &myChar
const char	*	ptr = &myChar
char	*	const ptr = &myChar
const char	*	const ptr = &myChar

```
#include <stdio.h>

void main(void)
{
	// declare some working variables

	const char myChar0 = 'a';
	char myChar1 = 'b';
	const char* ptr0 = &myChar0;
	char* const ptr1 = &myChar1;

	// *ptr0 = 'c';		// illegal ptr0 points to a constant
	*ptr1 = 'd';		// ok, the pointer not the object is const

	ptr0 = &myChar0;	// ok, the object is const not the pointer
	// ptr1 = &myChar1;	// illegal, the pointer is const
	return;
}
```

fig_07_14

Syntax

returnType functionName (arg_0, arg_1…arg_{n-1})

{

body

}

```
int multiply(int first, int second)
{
    // this is the function body
    return first * second;
}
```

```
#include <stdio.h>
// function prototype
int computeArea (length, width);
void main(void)
{
        // declare and initialize some variables
        int    length =10;
        int    width=20;
        int    area=0;

        area = computeArea(length, width);   //   this is the function call

        printf("the area is: %d\n", area);        //   displays 200
        return;
}

int computeArea(int first, int second)
{
        int    answer;
        answer =first *second;
        return answer;
}
```

fig_07_16

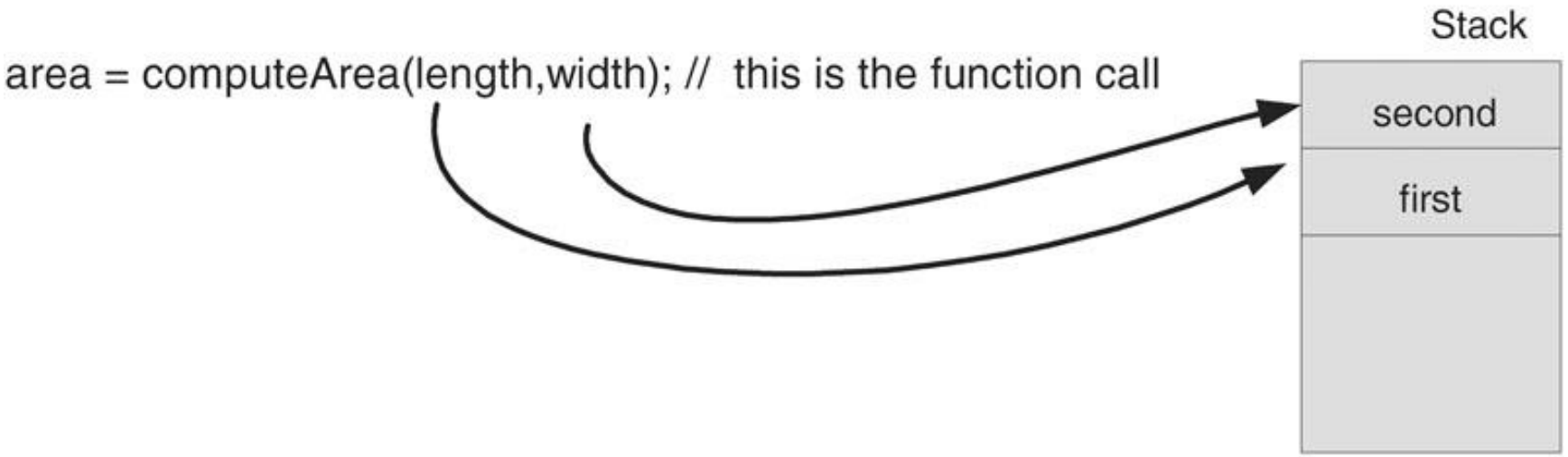

area = computeArea(length,width); // this is the function call
Stack
second
first

fig_07_17

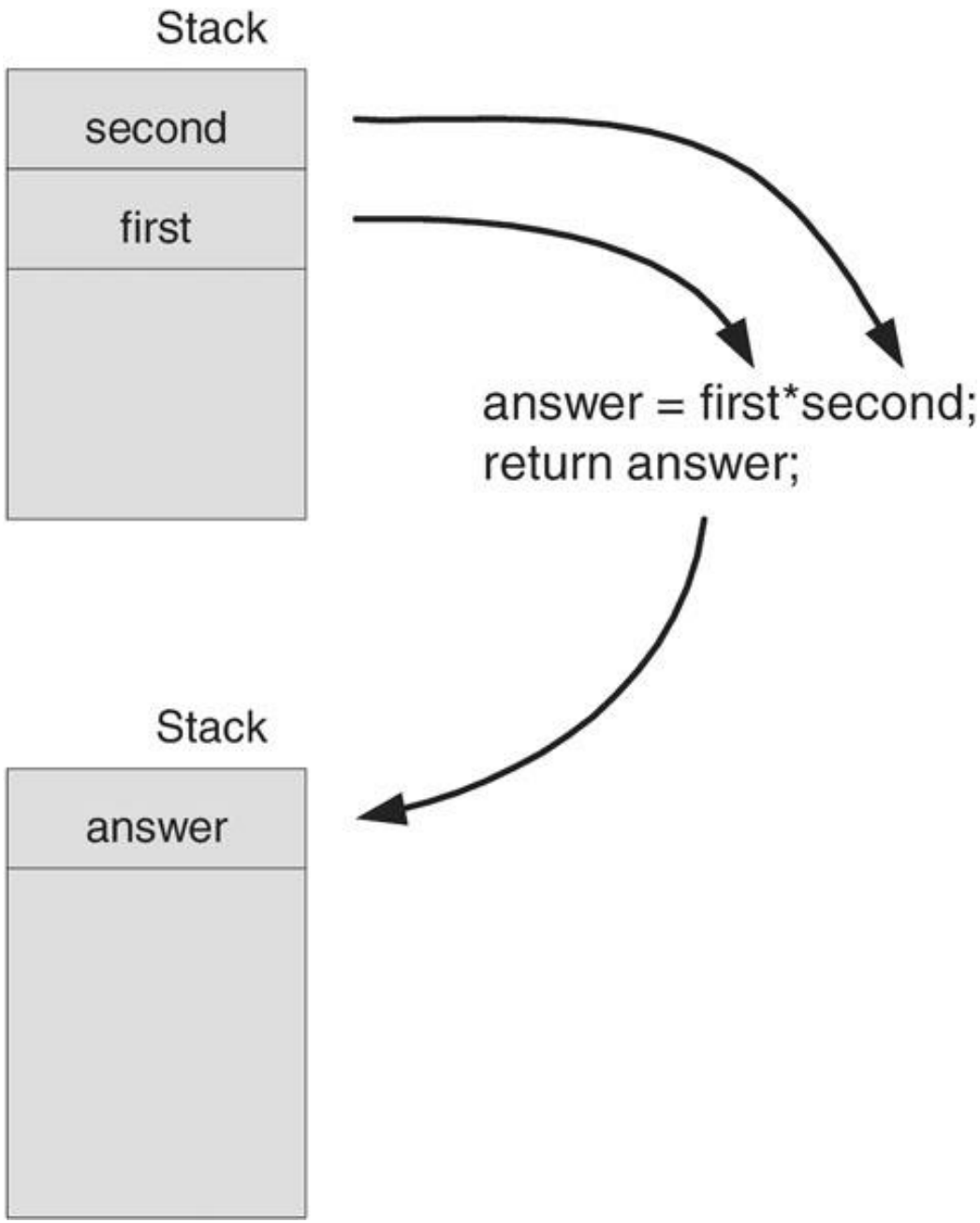

Stack
second
first
answer = first*second;
return answer;
Stack
answer

fig_07_18

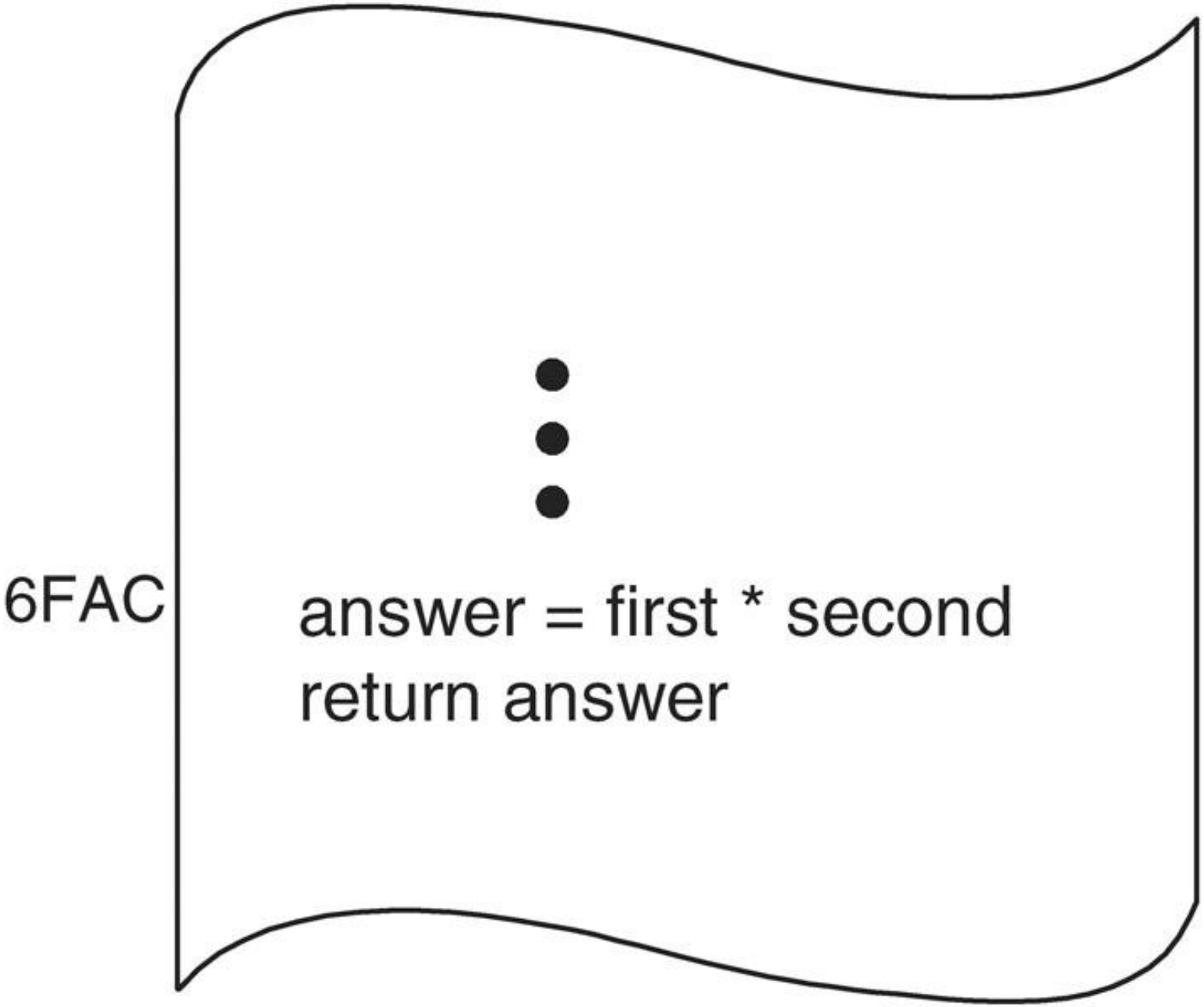


fig_07_19

```
#include <stdio.h>
/*
	Demonstrate pass and return by value in C
*/
int myFunction(int aValue);
void main(void)
{
	// declare and initialize some working variables
	int myValue = 5;
	int aReturnVal = 0;
	myFunction(myValue);

	// will show myValue as 5...no change
	printf("main(): myValue is: %i\n", myValue);

	// will show aReturnValue as 0...no change
	printf("main(): aReturnVal is: %i\n", aReturnVal);

	// by assigning to aReturnValue, we copy the returned value
	aReturnVal = myFunction(myValue);

	// will show aReturnValue as 9
	printf("main(): aReturnVal is: %i\n", aReturnVal);
	return;
}
int myFunction(int myValue)
{
	// declare and initialize a working variable
	int aReturnValue = 0;
	// change the value of the input parameter
	// this change will not appear in main
	myValue = myValue + 4;

	// will show myValue as 9
	printf("myFunction: aValue is: %i\n", myValue);
	aReturnValue = myValue;
	return aReturnValue;
}
```

fig_07_20

```
#include <stdio.h>
/*
	Demonstrate pass by reference in C
*/
void myFunction(int* aValuePtr);

void main(void)
{
	// declare and initialize a working variable
	int myValue = 5;
	// pass in the address of the data
	myFunction(&myValue);
	// will show myValue as 9...the original has been changed through the pointer
	printf("main(): myValue is: %i\n", myValue);
	return;
}

void myFunction(int* myValuePtr)
{
	// change the value of the input parameter this change will appear in main
	*myValuePtr = *myValuePtr + 4;
	// will show myValue as 9
	printf("myFunction(): myValue is: %i\n", *myValuePtr);
	return;
}
```

fig_07_21

```c
// staticFunct0.c

#include <stdio.h>
// make the function name available
// in this file
extern void myFunct0(void);

// this name will not be available
extern void myFunct1(void);
void main (void)
{
      myFunct0();

// results in compile error -
// the function name is not visible
      myFunct1();
      return;
}
```

```c
// containFunct0.c

#include <stdio.h>
// function prototypes
void myFunct0(void);

// function not visible outside of this file
// …remove static to make visible
static void myFunct1(void);

// define the functions
void myFunct0(void)
{
        int x = 3;
        printf("x is %i\n", x);
        return;
}
// remove static to make visible
static void myFunct1(void)
{
        int y = 4;
        printf("y is %i\n", y);
        return;
}
```

fig_07_22

```
/*
 * Function Name with Signature
 * Short Description of intent/purpose of the function
 * Input Parameters with short description and range of legal values
 * Return Values with short description and range of legal values
 * Side effects of the Function—What it might change that could affect other parts of the program.
 * Invariants—Things the function should not change
 * Revision History—Identify who, when, and what changes have been made to the function.
 * Citation of Code Source or Reference if developed by another author
 */
```

fig_07_23

Syntax

return type (* functionPointer) (<arg_0, arg_1...arg_n>)

arg list may be empty

syntax

(* functionPointer) (<arg_0, arg_1...arg_n>)

or

functionPointer (<arg_0, arg_1...arg_n>)

arg list may be empty

```
unsigned int anInt = 3;                    // declare some working variables
unsigned char aChar = 'a';

int (* intFunctPtr) ();                    // declare a function pointer
double (*doubleFunctPtr)(int, char);       // declare another function pointer

int myFunction(void);                      // declare a function
double yourFunction (int, char)            // declare another function

intFunctPtr = myFunction;                  // point to the first function
doubleFunctPtr = yourFunction;             // point to the second function

 (*intFunctPtr)();                         // dereference the first pointer
 (*doubleFunctPtr)(anInt, aChar );         // dereference the second pointer
```

fig_07_26

```
//     Pointers to Functions used as Function Arguments
#include <stdio.h>

// function prototypes
int add(int a1, int a2);
int sub(int a1, int a2);

// myFunction has a three parameters,
// a pointer to a function taking 2 ints as arguments and the argument values, and returning an int.

int myFunction (int (*fPtr)(int, int), int, int);

void main(void)
{
      // declare some working variables
      int sum, diff;

      // Declare fPtr as a pointer to a function taking 2 ints as arguments and returning an int
      int (*fPtr)(int a1, int a2);

      // assign fPtr to point to the add function
      fPtr = add;                                   // fPtr points to the function add
      sum = myFunction(fPtr, 2, 3);                 // pass fPtr to myFunction()
      printf ("The sum is: %d\n", sum);             // prints The sum is: 5

      // assign fPtr to point to the sub function
      fPtr = sub;                                   // fPtr points to the function sub
      diff = myFunction(fPtr, 5, 2);                // pass fPtr to myFunction()
      printf ("The difference is: %d\n", diff);     // prints The difference is: 3

      return;
}
// perform requested binary computation and return result
int myFunction (int (*fPtr)(int a1, int a2), int aVar0, int aVar1)
{
      // variables a1 and a2 are placeholders - they are not used
      // dereference the pointer and return value
      return (fPtr(aVar0, aVar1));
}

// add two integers and return their sum
int add(int a1, int a2)
{
      return (a1+a2);
}

// subtract two integers and return their difference
int sub(int a1, int a2)
{
      return (a1-a2);
}
```

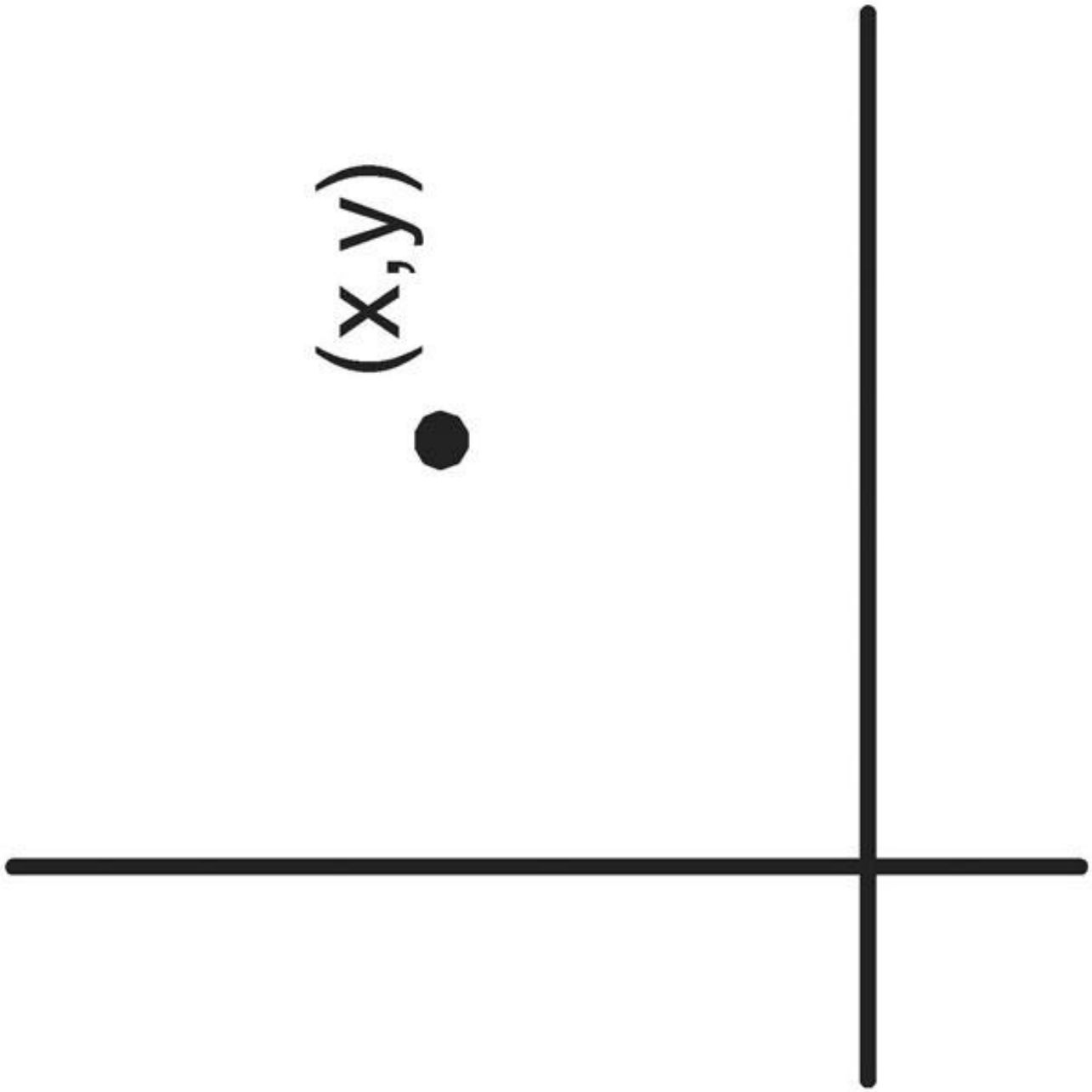
fig_07_27

```
// Declare fPtr as a pointer to a function taking 2 ints as arguments and returning an int
int (*fPtr)(int a1, int a2);

// assign fPtr to point to the add function
fPtr = add;                          // fPtr points to the function add
sum = myFunction(fPtr, 2, 3);        // pass fPtr to myFunction()
printf ("The sum is: %d\n", sum);    // prints The sum is: 5
```

```
// assign fPtr to point to the sub function
fPtr = sub;                                   // fPtr points to the function sub
diff = myFunction(fPtr, 5, 2);                // pass fPtr to myFunction()
printf ("The difference is: %d\n", diff);     // prints The difference is: 3
```

fig_07_30



syntax

```
struct StructTag
{
    struct body;
};
struct StructTag anInstance;
```

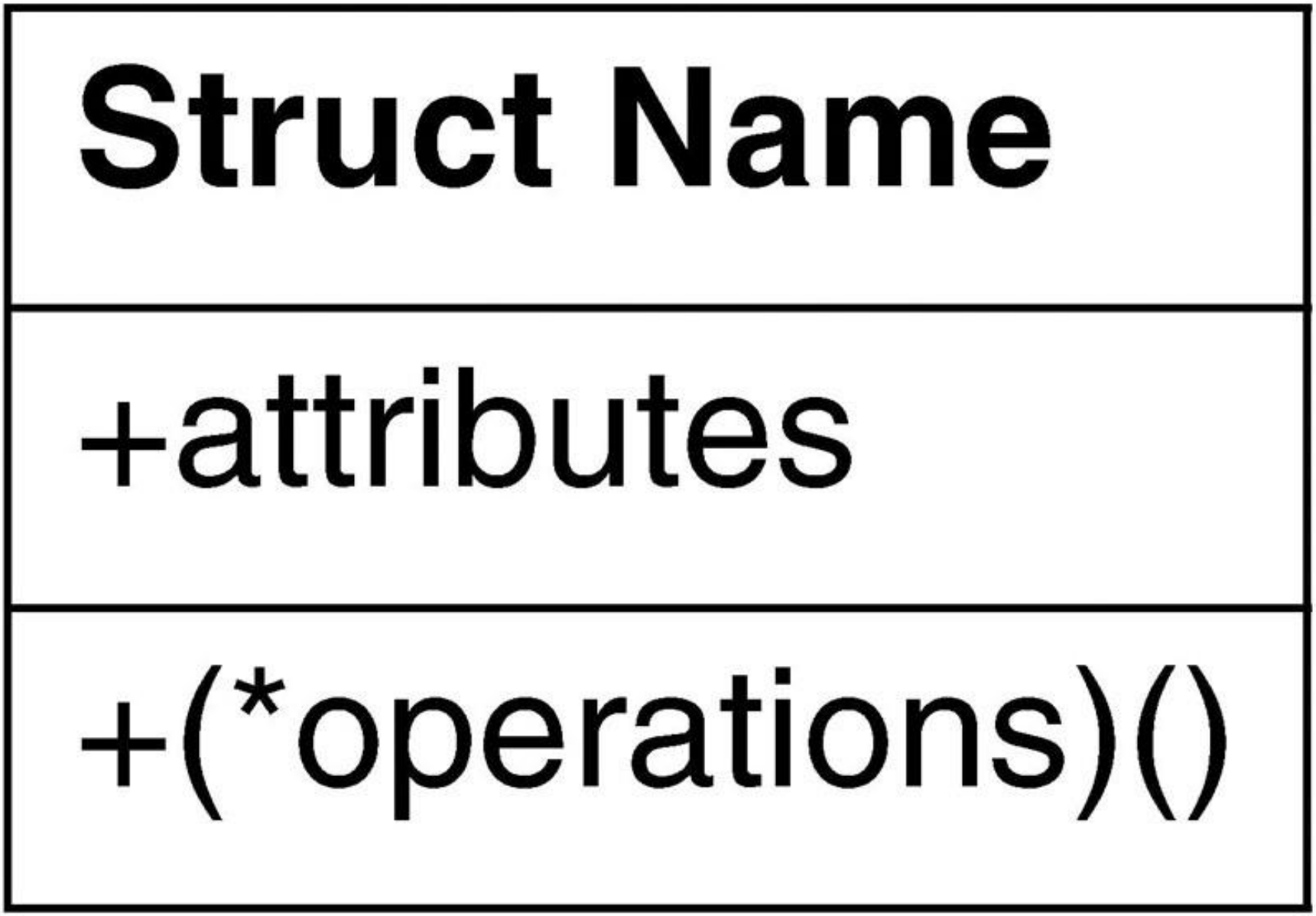
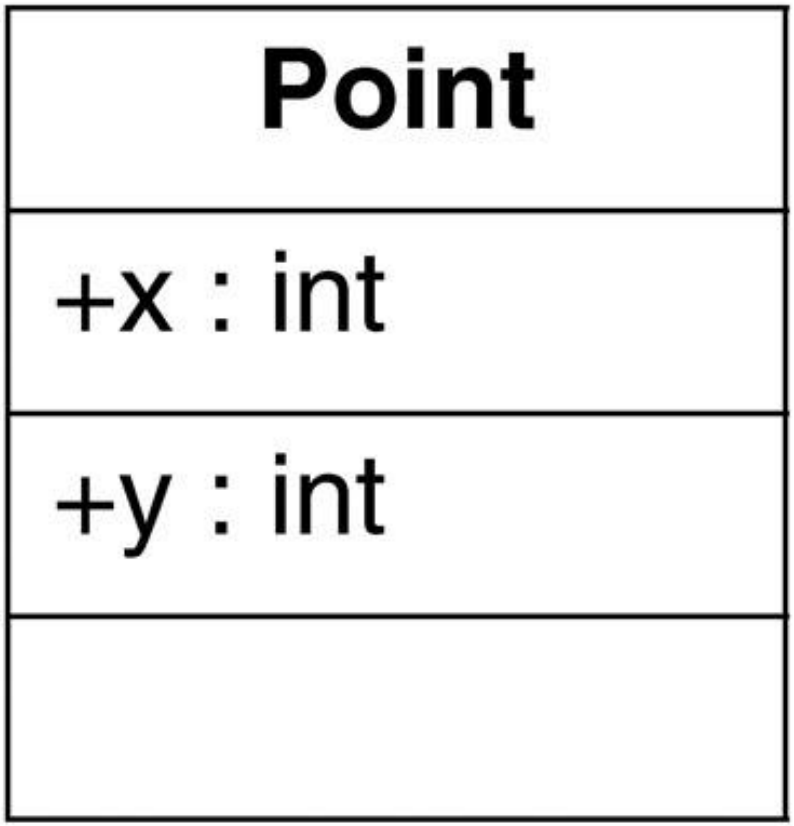


fig_07_32




```
struct Point
{
    int x;
    int y;
};
```

fig_07_33

```
typedef struct
{
    int x;
    int y;
} Point;
```

fig_07_34

syntax

```
struct StructTag
{
        struct body;
};

struct StructTag anInstance = (initializer list);
```

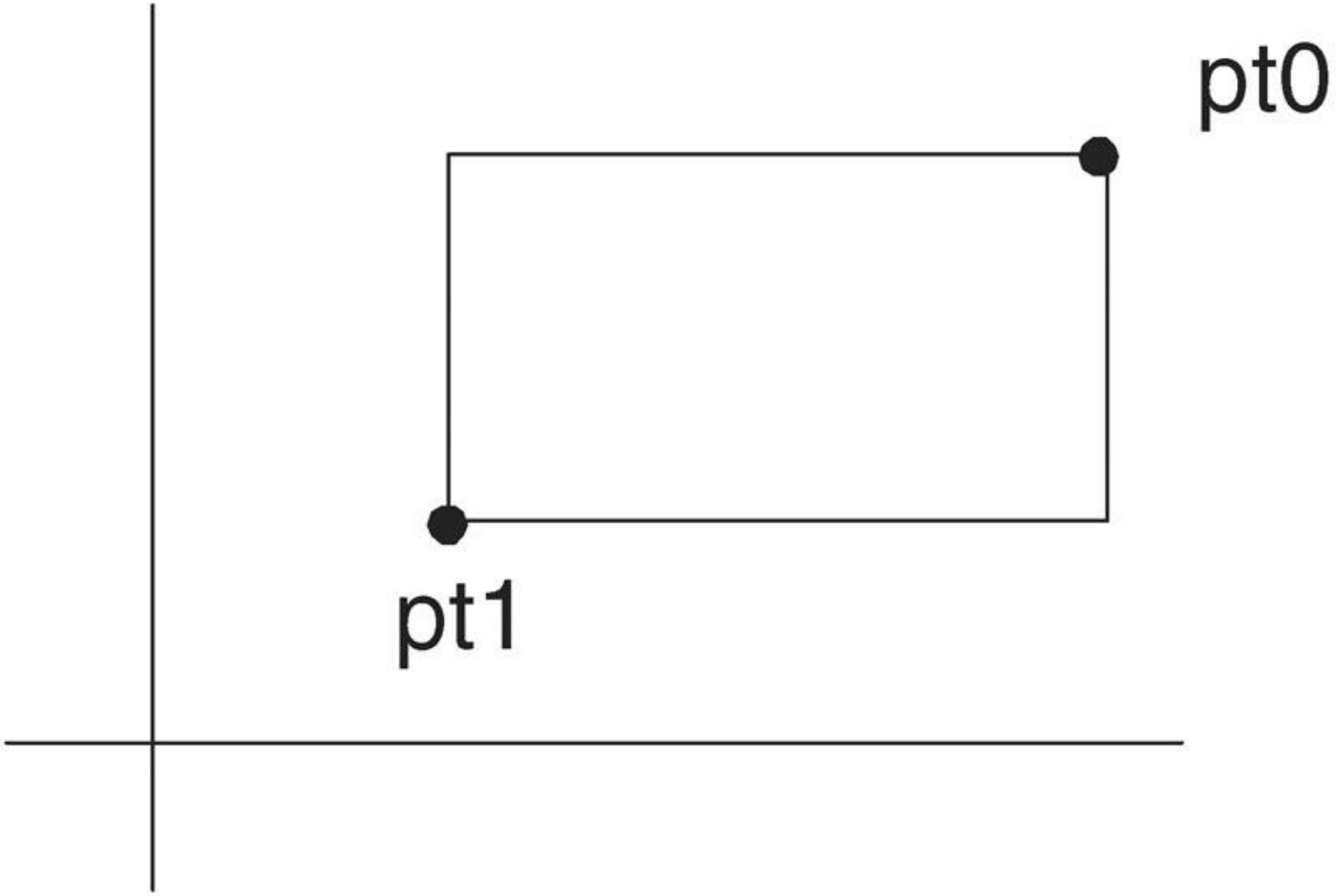


fig_07_36

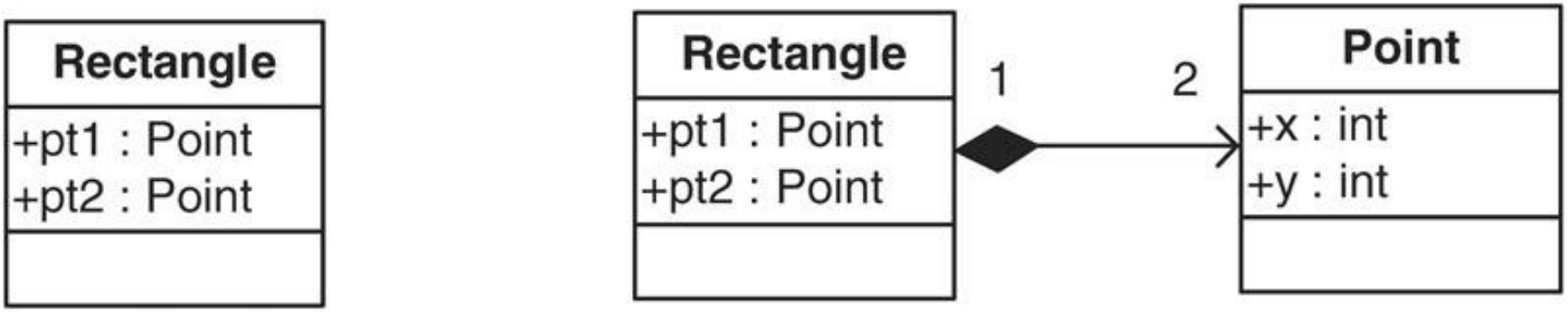
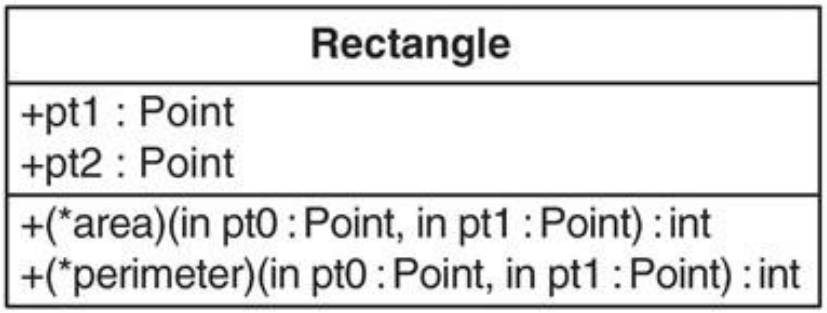

Rectangle
+pt1 : Point
+pt2 : Point
Rectangle
+pt1 : Point
+pt2 : Point
1
2
Point
+x : int
+y : int

fig_07_37

```
typedef struct
{
        int x;
        int y;
} Point;
```

```
typedef struct
{
        Point pt1;
        Point pt2;
} Rectangle;
```

fig_07_38




```
typedef struct
{
    Point pt1;
    Point pt2;
    int (*area)(Point pt0, Point pt1);
    int (*perimeter) (Point pt0, Point pt1);
} Rectangle;
```

```
typedef struct
{
    int x;
    int y;
} Point;

typedef struct
{
    Point pt1;
    Point pt2;
    int (*area)(Point pt0, Point pt1);
    int (*perimeter) (Point pt0, Point pt1);
} Rectangle;
int computeArea(Point pt0, Point pt1);
int computePerimeter(Point pt0, Point pt1);
```

fig_07_40

```
#include "rect.h"
// Rectangle area function
int computeArea(Point pt0, Point pt1)
{
        int area = (pt1.x - pt0.x) * (pt1.y - pt0.y);
        return area;
}

// Rectangle perimeter function
int computePerimeter(Point pt0, Point pt1)
{
        int perimeter = 2*(pt1.x - pt0.x) + 2*(pt1.y - pt0.y);
        return perimeter;
}
```

```
#include <stdio.h>
// bring in struct definitions and function prototypes
#include "rect.h"

void main (void)
{
        // declare and instance of Rectangle
        Rectangle myRectangle;
        // declare some working variables
        int myArea = 0;
        int myPerimeter = 0;

        // assign values to instance data members
        myRectangle.pt1.x = 5;
        myRectangle.pt1.y = 10;
        myRectangle.pt2.x = 10;
        myRectangle.pt2.y = 20;

        // assign values to instance function pointers
        myRectangle.area = computeArea;
        myRectangle.perimeter = computePerimeter;

        // compute the area and perimeter
        myArea = myRectangle.area(myRectangle.pt1, myRectangle.pt2);
        myPerimeter = myRectangle.perimeter(myRectangle.pt1, myRectangle.pt2);

        printf("the area and perimeter are: %i, %i\n", myArea, myPerimeter);
        return;
}
```

fig_07_42

```
// Passing Structures to Functions

#include <stdio.h>
// declare the struct
typedef struct
{
      int aVar0;
      int* aVar1Ptr;
}Data;

//    Declare the function prototype
void funct0(Data aBlock);
void funct1 (Data* aBlock);

void main(void)
{
      Data myData;

      // Declare and define a variable
      int varData0 = 20;

      // assign values to the struct data members
      myData.aVar0 = 10;
      myData.aVar1Ptr = &varData0;

      // Will print on execution:
      // The variables values are: 10, 20

      // Pass the struct to the function by
      // value then by reference
      funct0(myData);
      funct1(&myData);

      return;
}
```

```
void funct0(Data aBlock)
{
      // Retrieve the data from the struct
      // Using the member selector
      printf ("The variables values are: ");
      printf ("%i, %i\n",    aBlock.aVar0,
                             *(aBlock.aVar1Ptr));

      return;
}

void funct1(Data* aBlockPtr)
{
      // Retrieve the data from the struct
      // Using the pointer to member selector

      printf ("The variables values are: ");
      printf ("%i, %i\n",     aBlockPtr->aVar0,
                              *(aBlockPtr->aVar1Ptr));

      return;
}
```

fig_07_43

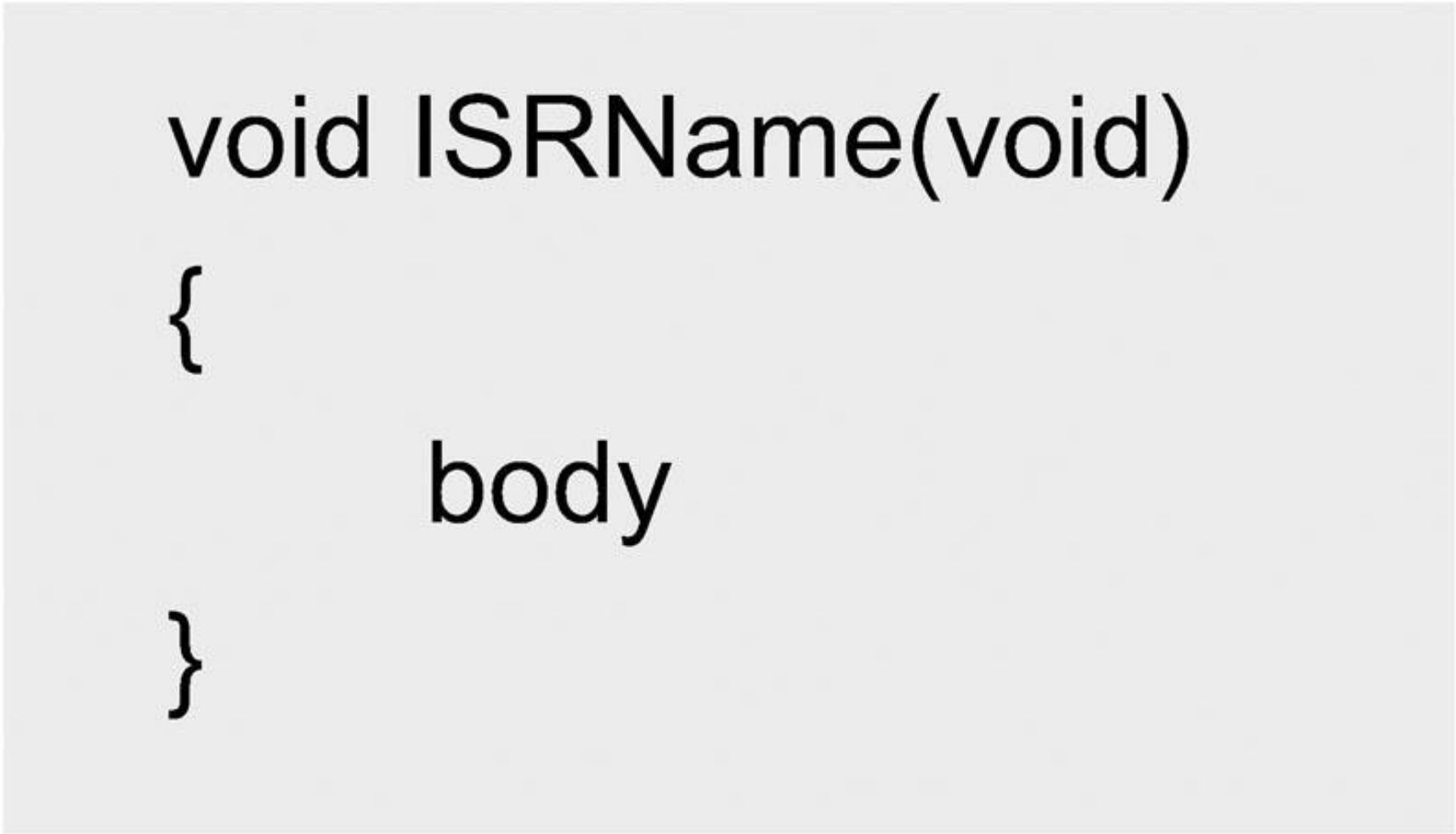


fig_07_44

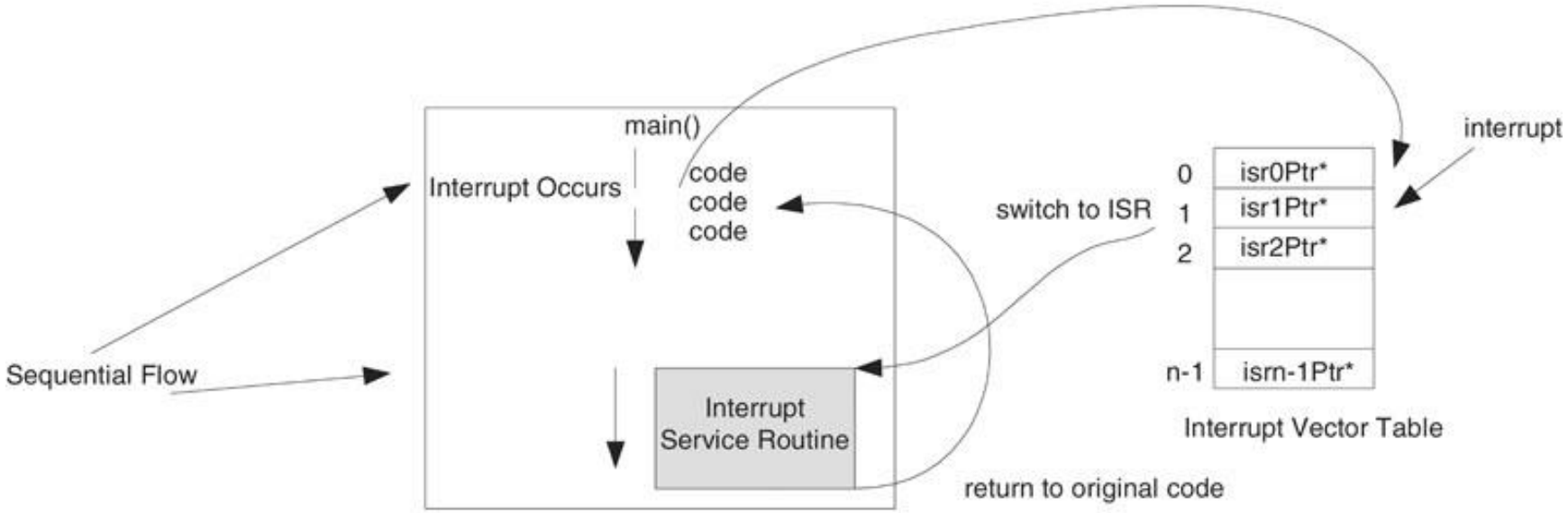

main()
Interrupt Occurs
code
code
code
Sequential Flow
Interrupt
Service Routine
switch to ISR
interrupt
0
1
2
n-1
isr0Ptr*
isr1Ptr*
isr2Ptr*
isrn-1Ptr*
Interrupt Vector Table
return to original code

fig_07_45

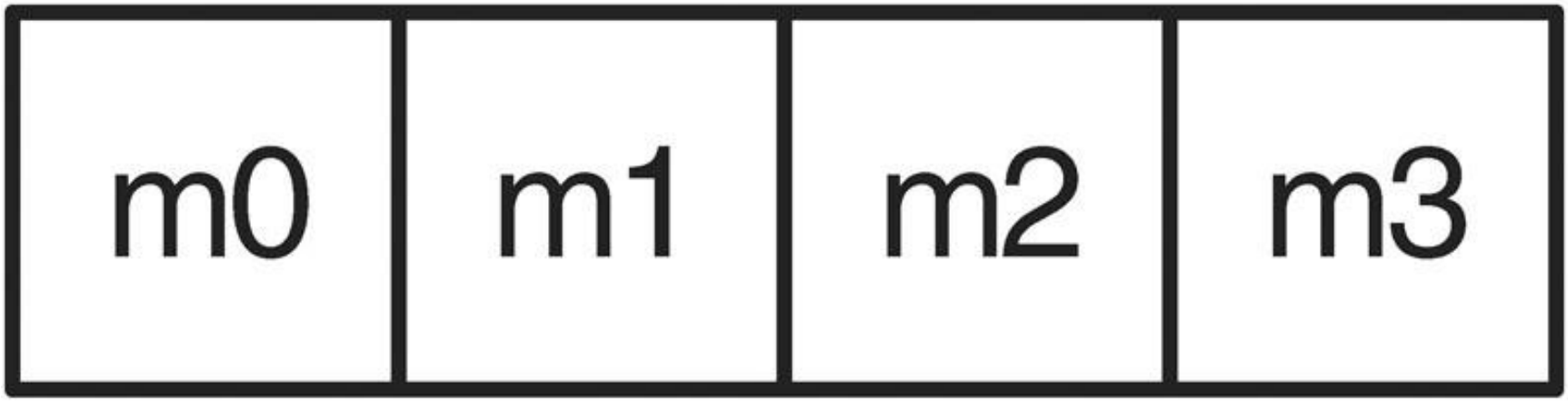


fig_07_46

Table 7.0 Bitwise Operators

	Operator	Meaning	Description
Shift			
	>>	Logical shift right	Operand shifted positions are filled with 0's
	<<	Logical shift left	Operand shifted positions are filled with 0's
Logical			
	&	Bitwise AND	
	|	Bitwise inclusive OR	
	^	Bitwise exclusive OR	
	~	Bitwise negation	