C Programming for Embedded Systems

Computer Layers
Low-level hardware to high-level software
(4GL: “domain-specific”, report-driven, e.g.)

Embedded systems programming:

--likely to be at lower levels (assembly language, C), require excellent knowledge of pointers, e.g.

--may not be “room” for elaborate data structures

--safety and security concerns require more attention to programming details, e.g., stack management

--less opportunity for “automatically generated code”. e.g., Netbeans

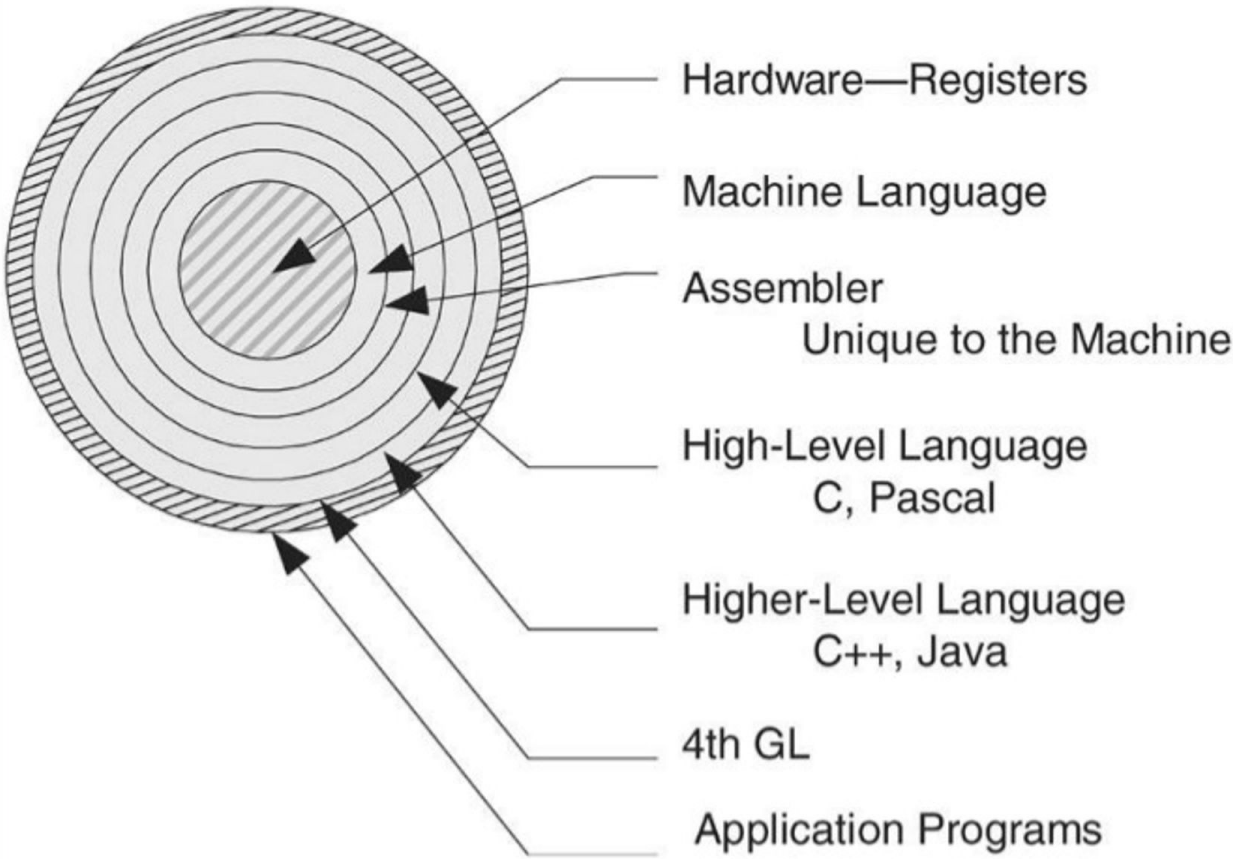


fig_06_00

From source code to executable program:
Compilation:

Preprocessing step:

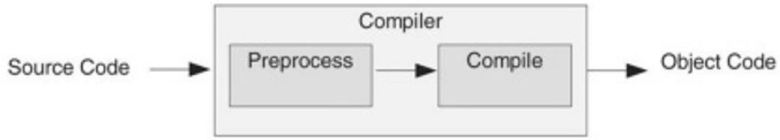


fig_06_01

Compiling or assembling:

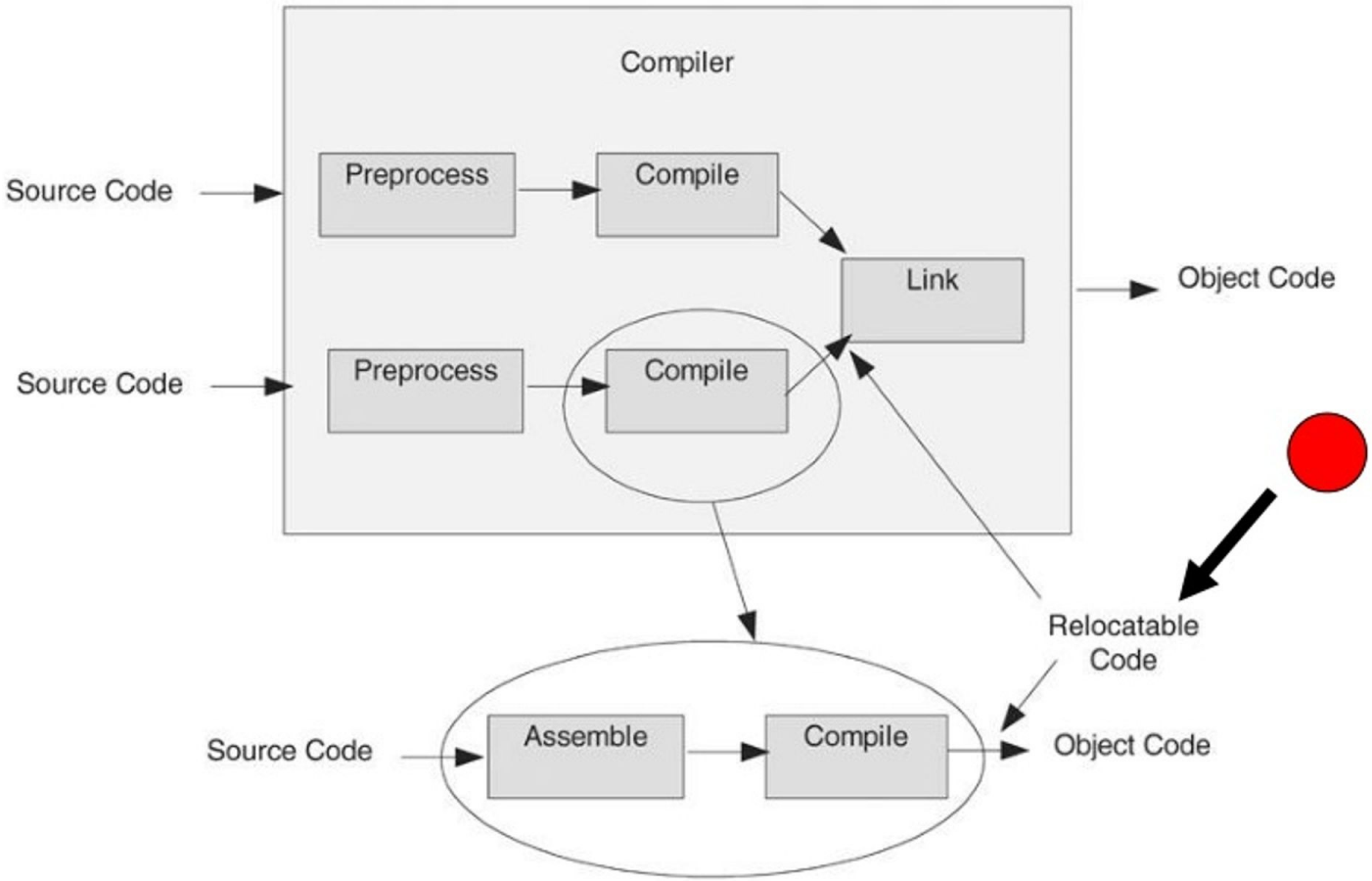


fig_06_03

The entire process:

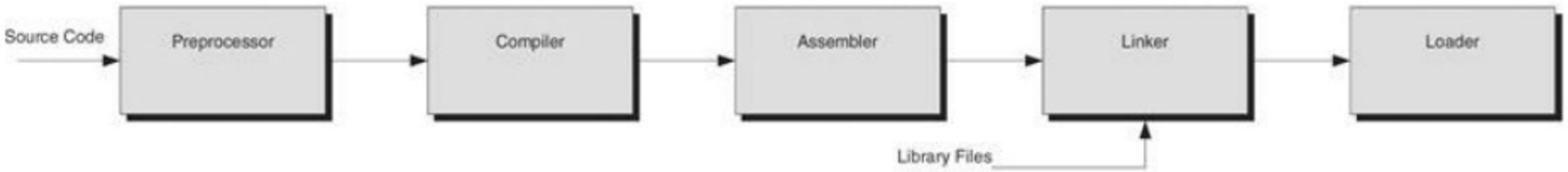
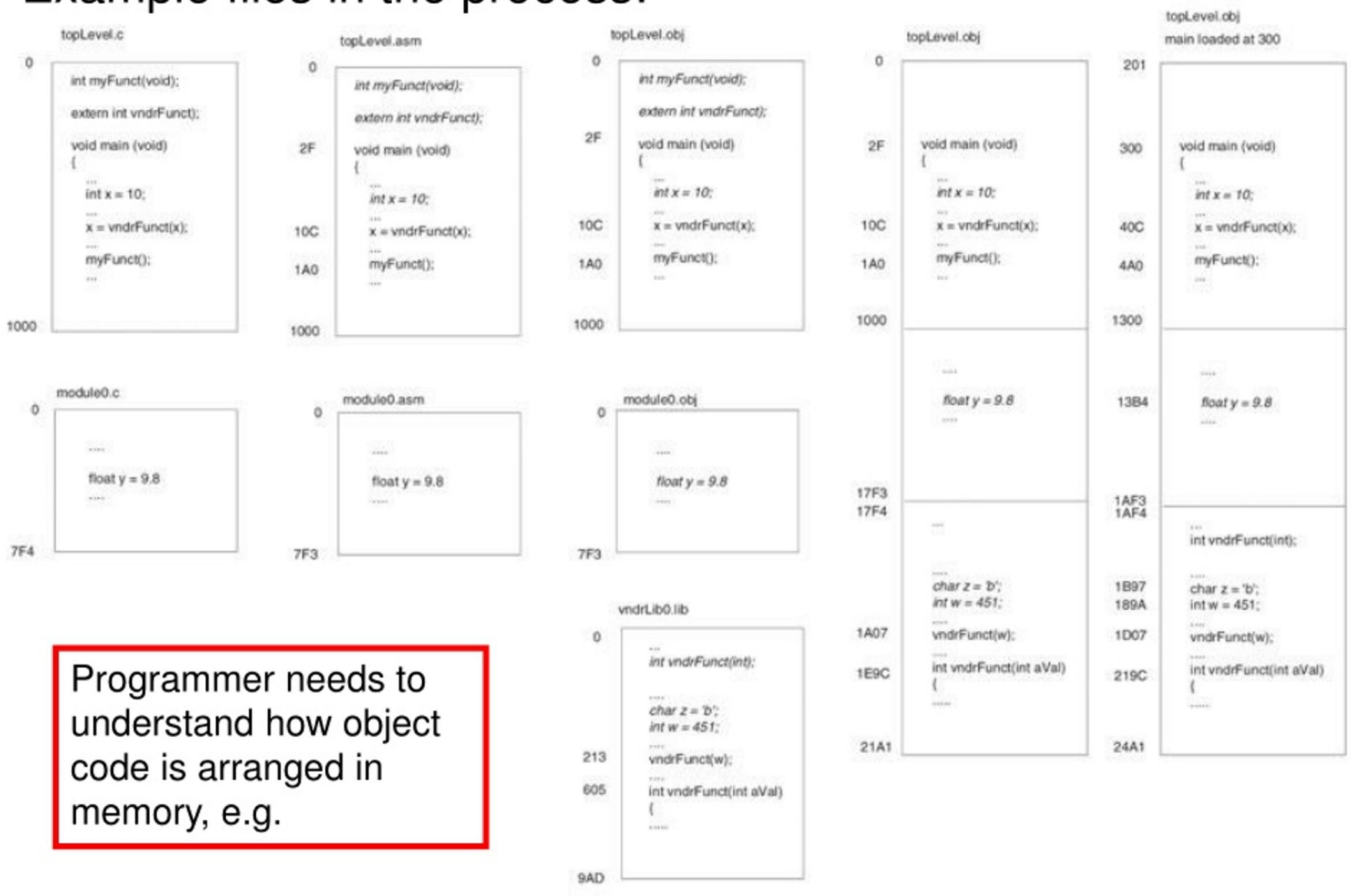


fig_06_04

Example files in the process:



Programmer needs to understand how object code is arranged in memory, e.g.

fig_06_05

Important features of embedded code:

Performance

Robustness (response to failures)

Security conscious

Ease of change

Style—***simple***, understandable

example: flag = x != 0 && ! y/x < 0
how is this evaluated?
in embedded systems:
use parentheses!!!!!!!

Important to understand details of data storage:

-- safety and security require higher sensitivity to error conditions such as overflow, e.g.
--need to understand internal data formats and any changes needed to interface with I/O devices

Example: C integral types: what is size in YOUR system?

char

short

int

long

(signed or unsigned)

Unsigned integer—range and format

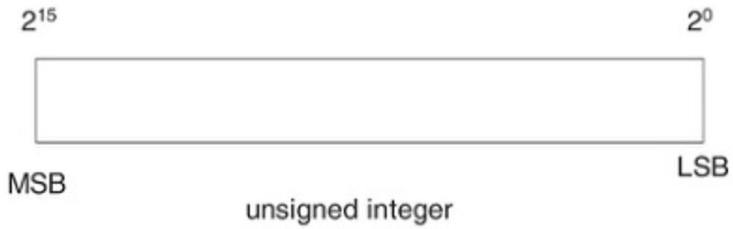


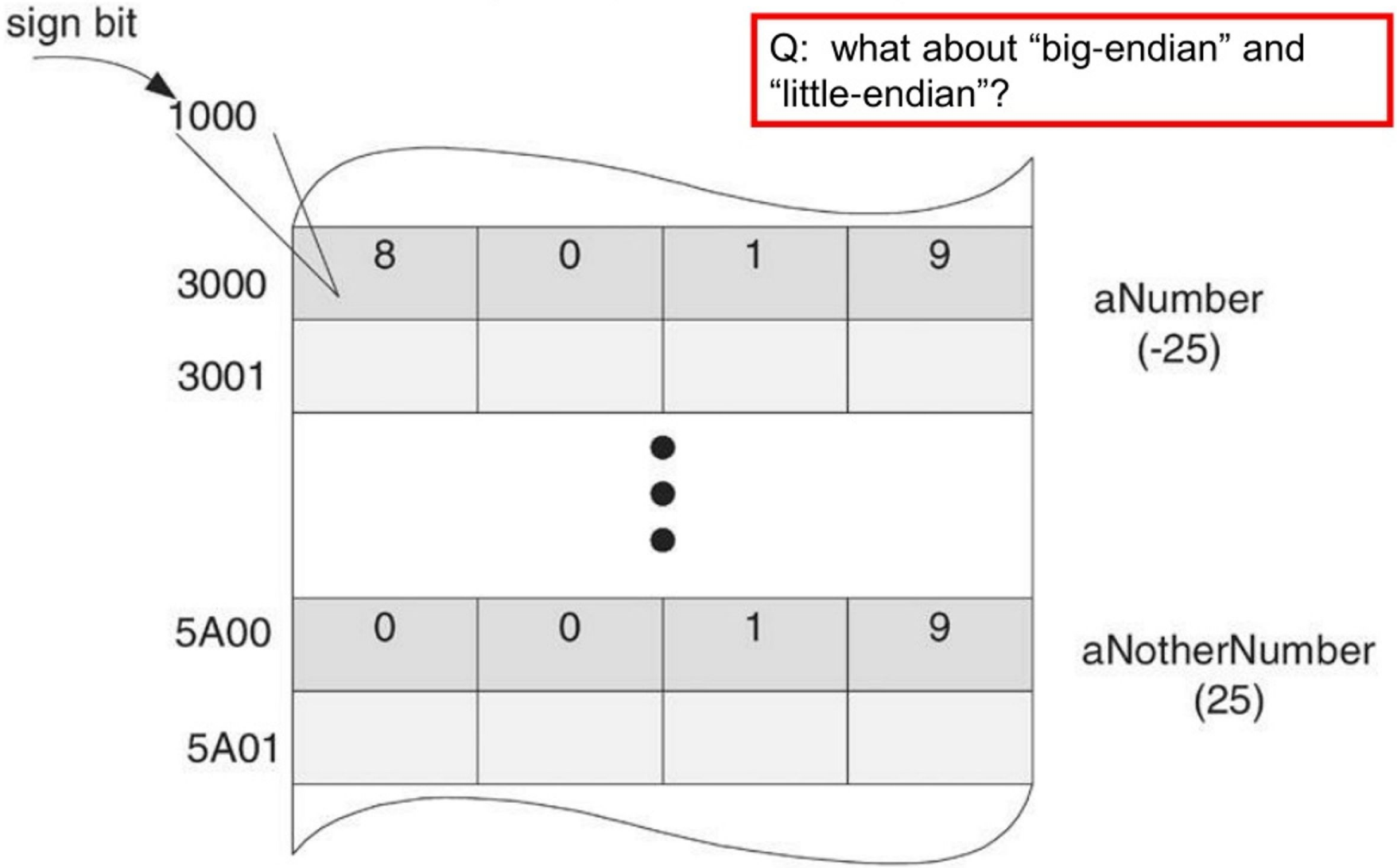
fig_06_08

Signed integer (1's complement)—range & form

What about 2's complement?

fig_06_10

Numbers in memory—sign in most sig “nibble”



Q: what about “big-endian” and “little-endian”?

fig_06_12

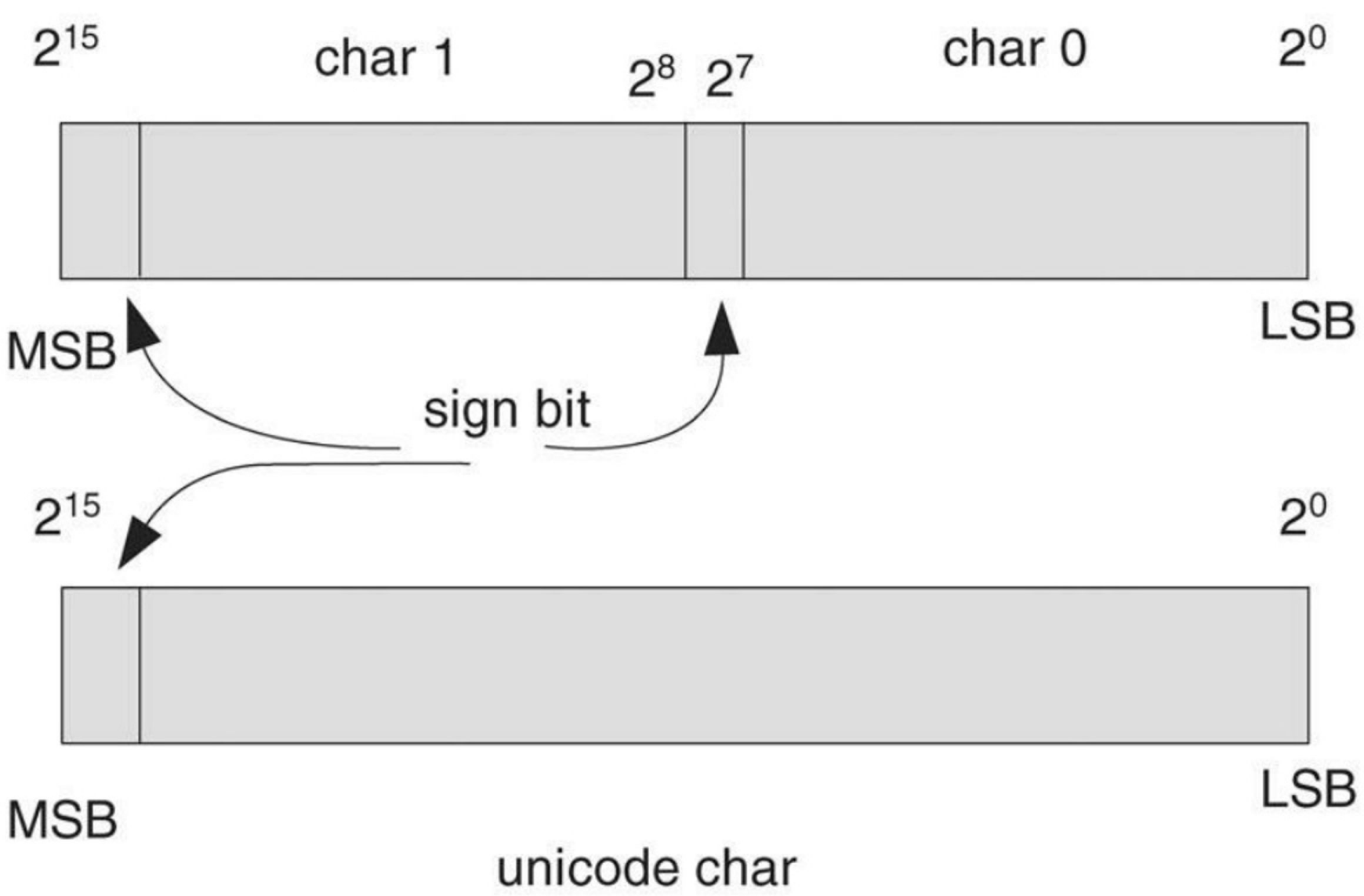

Character data
2^{15}
char 1
2^8 2^7
char 0
2^0
MSB
LSB
sign bit
2^{15}
2^0
LSB
MSB
unicode char

fig_06_14

Floating point data (32 bit)

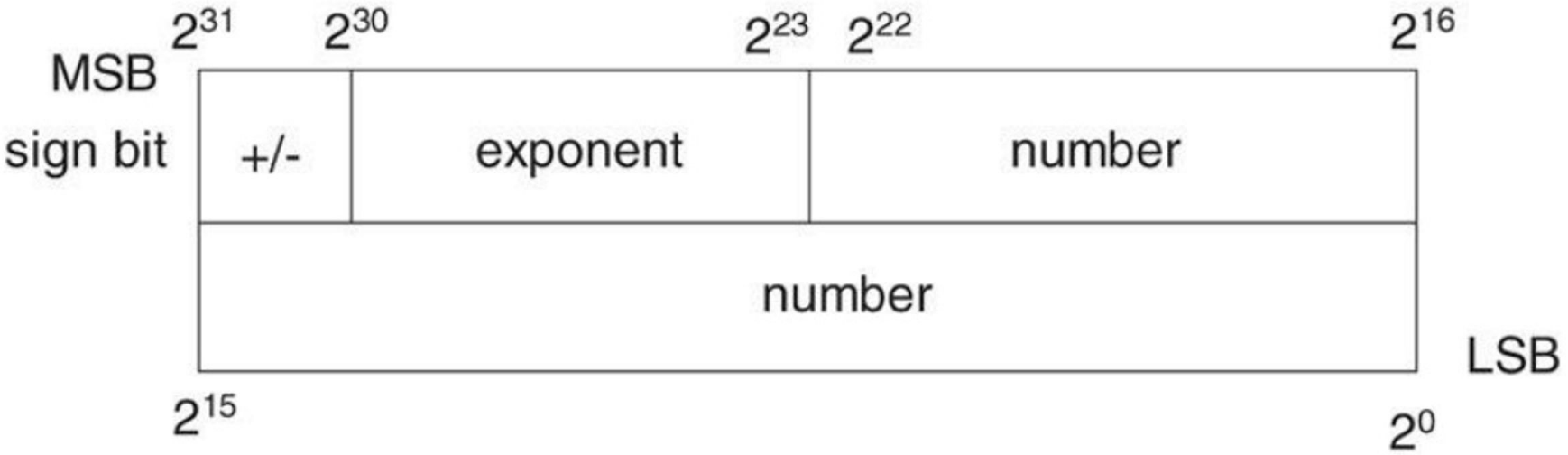


fig_06_15

Exponent: “excess notation”

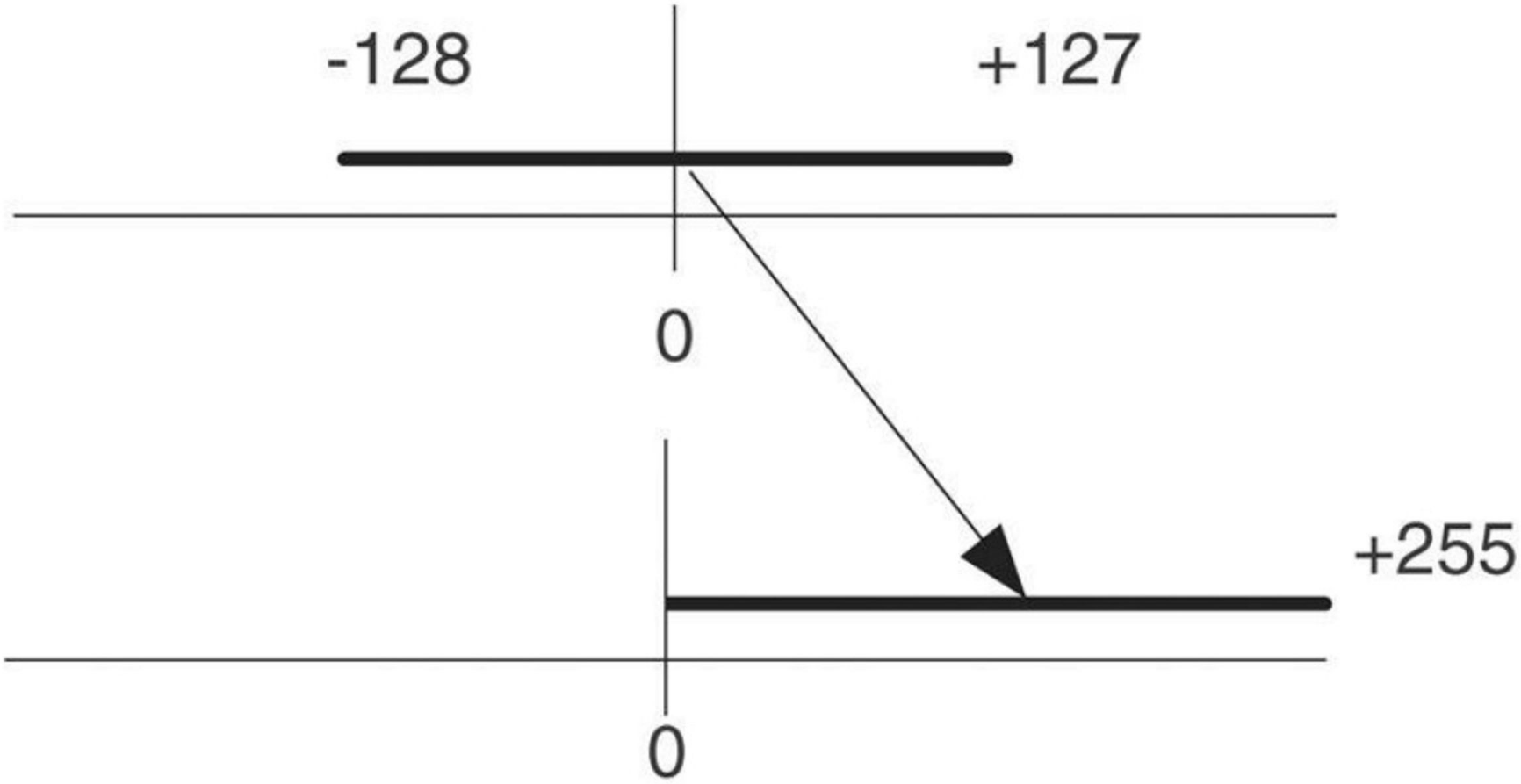


fig_06_16

Example program:

```
#include <stdio.h>
// Execution on a 32 bit machine
void main(void)
{
        // declare some variables
        short myShort = 2;
        int myInt = 3;
        myInt = myShort;
        printf("myInt %i\n", myInt);              // prints 2

        myInt = 3;
        myShort = myInt;
        printf("myShort %i\n", myShort);          // prints 3

        myInt = 32767;
        myShort = myInt;
        printf("myShort %i\n", myShort);          // prints 32767

        myInt = 100000;
        myShort = myInt;
        printf("myShort %i\n", myShort);          // prints -31072
        return;
}
```

fig_06_17

Variable designations and visibility: scope

- Program or Global
- Local
- File

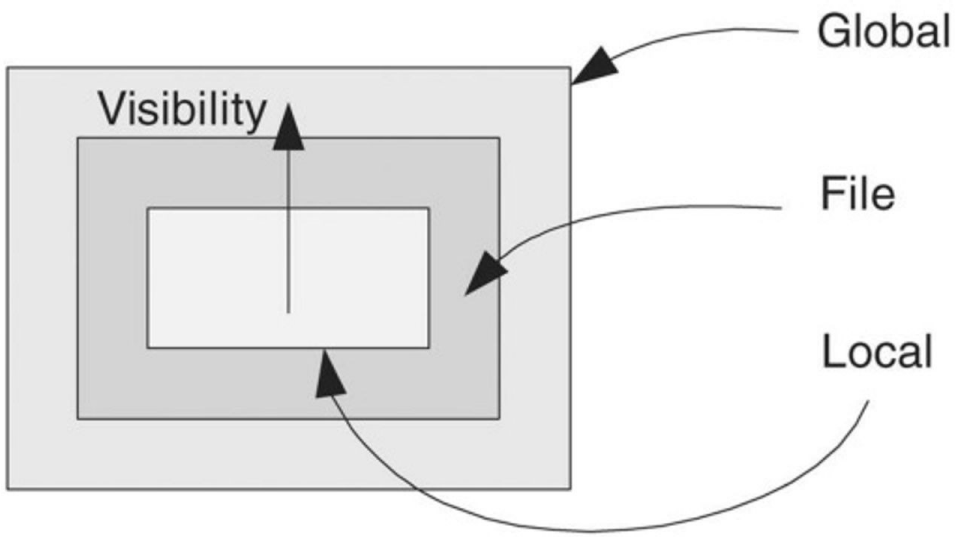


fig_06_18

Storage classes

- auto
- extern
- static
- register
- typedef
- volatile

fig_06_20

C storage classes—when and why?

--auto

--extern

--static

--register

--typedef

--volatile

Useful explanations:

auto, register, static, extern:
http://itee.uq.edu.au/~comp2303/Leslie_C_ref/C/CONCEPT/storage_class.html

typedef:
http://itee.uq.edu.au/~comp2303/Leslie_C_ref/C/SYNTAX/typedef.html

Keyword volatile: important in embedded programming: Ex: from http://en.wikipedia.org/wiki/Volatile_variable
In C, and consequently C++, the volatile keyword was intended to[1]
- allow access to memory mapped devices
- allow uses of variables between setjmp and longjmp
- allow uses of sig_atomic_t variables in signal handlers.

Operations on volatile variables are not atomic, nor do they establish a proper happens-before relationship for threading. This is according to the relevant standards (C, C++, POSIX, WIN32), and this is the matter of fact for the vast majority of current implementations. The volatile keyword is thus basically worthless as a portable threading construct.[2][3][4][5][6]

Example of memory-mapped I/O in C: In this example, the code sets the value stored in foo to 0. It then starts to poll the value repeatedly until it changes to 255:

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

An optimizing compiler will notice that no other code can possibly change the value stored in foo, and will assume that it will remain equal to 0 at all times. The compiler will therefore replace the function body with an infinite loop similar to this:

```
void bar_optimized(void) {
    foo = 0;

    while (true)
```

However, foo might represent a location that can be changed by other elements of the computer system at any time, such as a hardware register of a device connected to the CPU. The above code would never detect such a change; without the volatile keyword, the compiler assumes that the current program is the only part of the system that could change the value (which is by far the most common situation).

To prevent the compiler from optimizing code as above, the volatile keyword is used:

```
static volatile int foo;

void bar (void) {
    foo = 0;
     while (foo != 255)
        ;
}
```

With this modification the loop condition will not be optimized away, and the system will detect the change when it occurs. However, it is usually overkill to mark the variable volatile as that disables the compiler from optimizing any accesses of that variable instead of the ones that could be problematic. Instead, it is a better idea to cast to volatile where it is needed:

In C:

```
static int foo;

void bar (void) {
    foo = 0;
     while (*(volatile int *)&foo != 255)
        ;
}
```

Example:

```
// staticData0.c
#include <stdio.h>

// make the function name available in this file
extern unsigned int myData0;
extern unsigned int myData1;

void main (void)
{
        printf ("myData0 is: %i \n", myData0);

// results in compile error - the variable name
// is not visible
// printf ("myData1 is: %i \n", myData1);
        return;
}
```

```
// containData0.c

#include <stdio.h>

unsigned int myData0 = 3;

static unsigned int myData1 = 4;
```

Separate compilations:

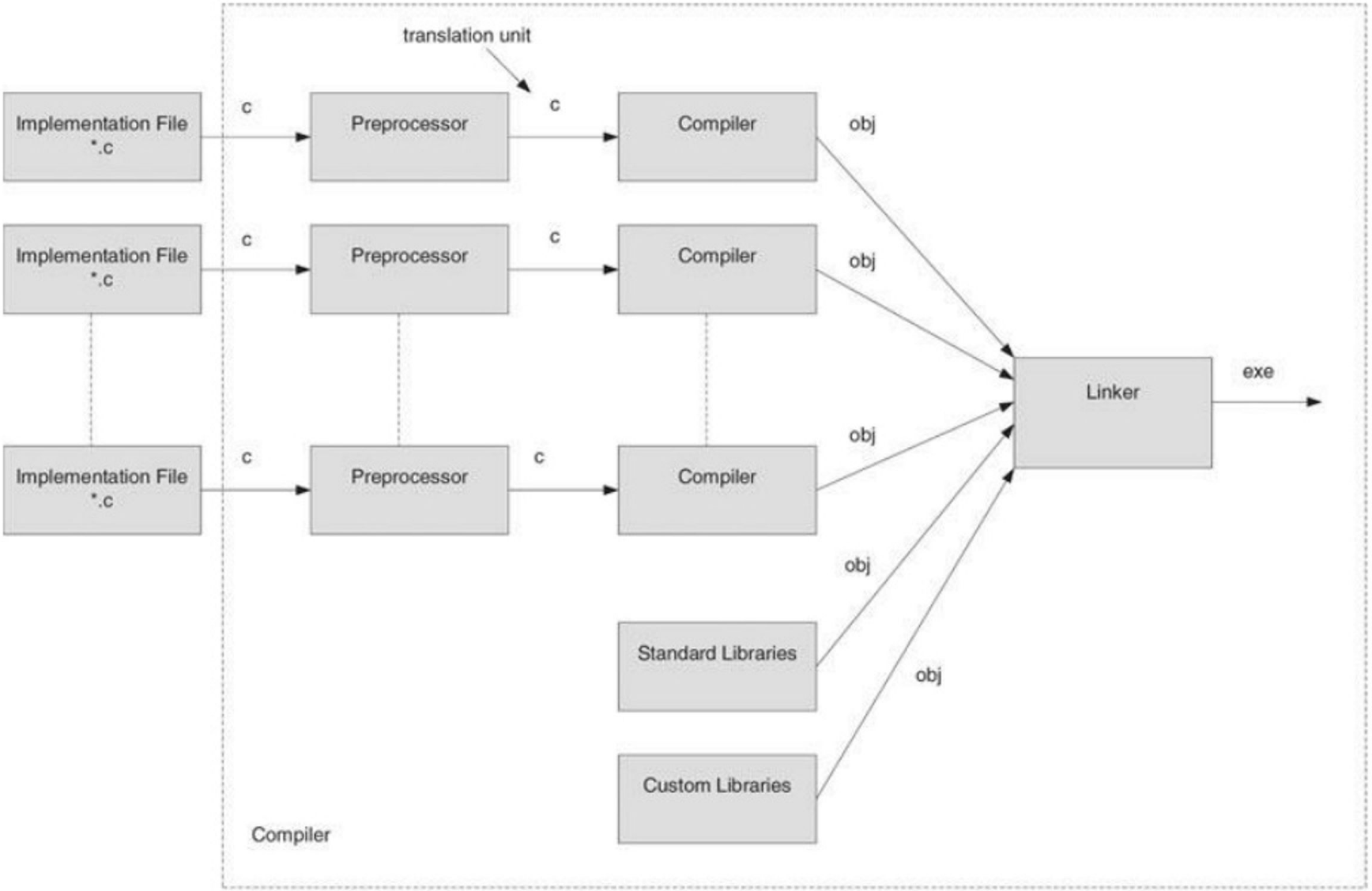


fig_06_22

Makefiles: example

```
A very simple makefile
    To make hw.exe (the .exe is implied) use hw.o and
        Use the gcc compiler with the identified input, flag, and output

    To make hw.o use hw.c and
        Use the gcc compiler with the identified input and flag
hw: hw.o
    gcc hw.o -o hw

hw.o: hw.c
    gcc -c hw.c

// Simple classic C program hw.c
#include <stdio.h>
int main(void)
{
    printf("hello world\n");
    return 0;
}
```

fig_06_23

Bitwise Operators

Pointers

Functions

Structs

Interrupts

(in C)

Bitwise operators: useful for dealing with signals of different widths;
NOTE: these are LOGICAL operators and variables are declared UNSIGNED—why?

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	

Additional useful reference:
http://www.cs.cf.ac.uk/Dave/C/node13.html

Examples of c operations at the byte level

```
// 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

Example of c program—"portShadow" mirrors what is on port;
Note use of parentheses to guarantee order of bit-level operations (even if we are using default order)

```
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);
}
```

Using shift operators (remember these are logical):

```
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
```

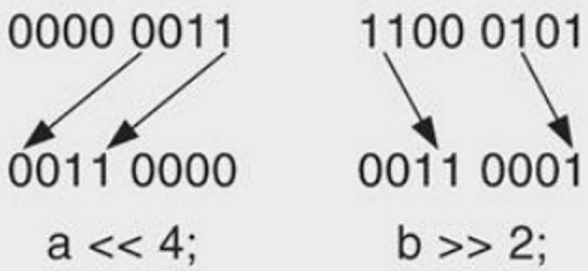


fig_07_02

Redoing the previous problem with shifts:

```
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);
}
```

Getting data from the port:

```
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

Using bit-level operations for arithmetic;
Are there any problems with this approach?

Multiply by x where x is 2, 4, 8, …

```
result = number << x;
```

Divide by y where y is 2, 4, 8, …

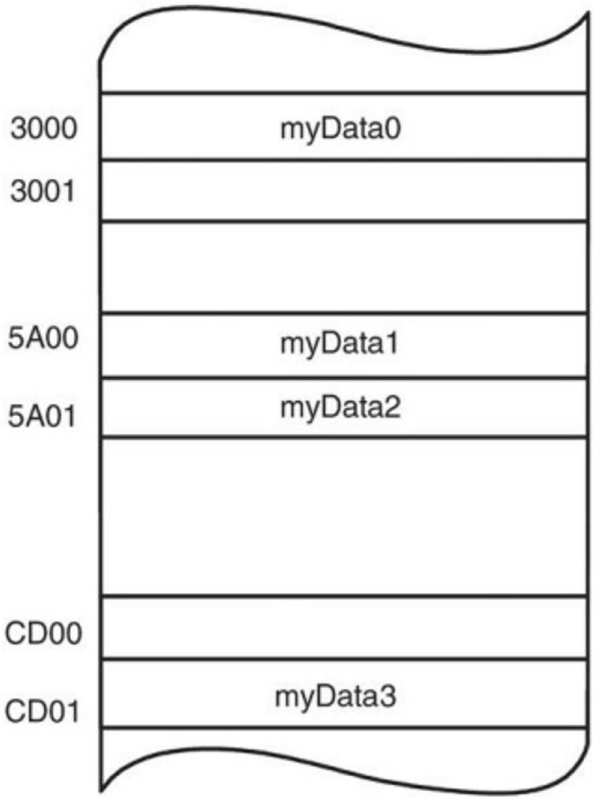
```
result = number >> y;
```

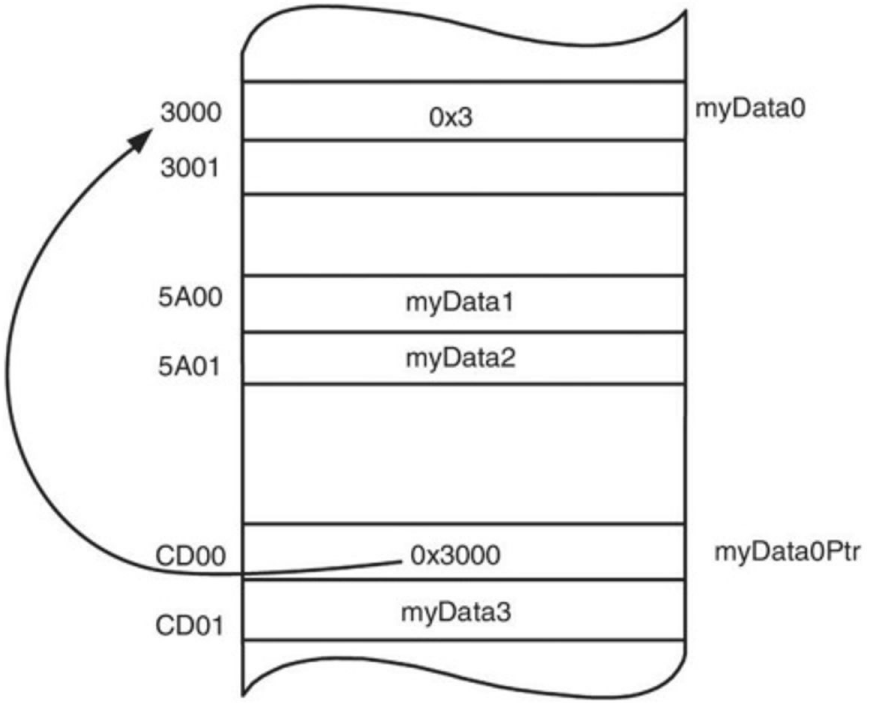
C: on signed data, left shift undefined if overflow occurs, right shift is implementation-dependent;
Java: all integers are signed

Can use shifts for slightly more complex multiplies and divides;
More complex operations (e.g., with 3 1's in multiplier or divider) are probably not efficient

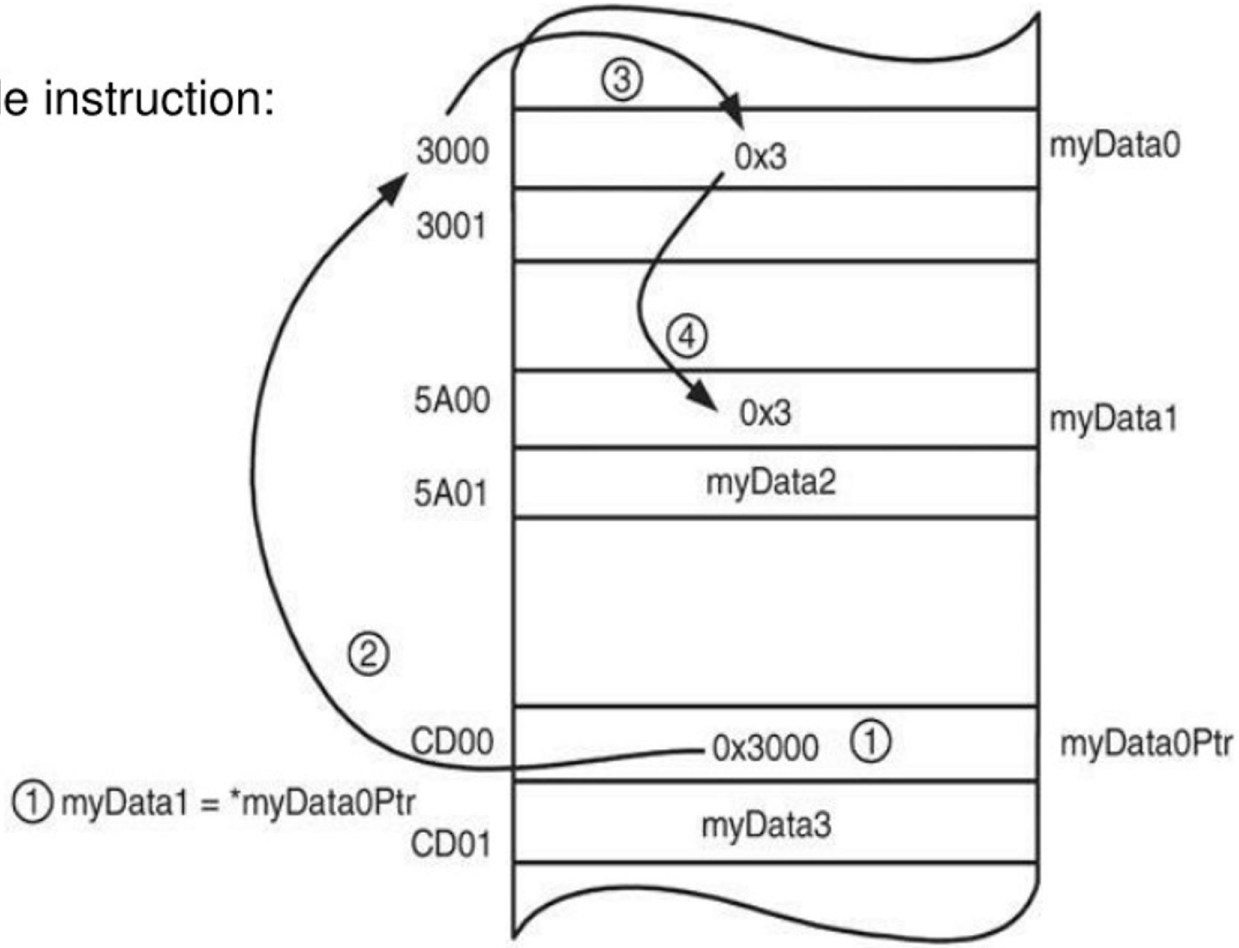
```
Multiply by x where x is a simple number such as 5, 6, 9, 10, 12,…
result = (number << 2) + number;          // multiply by 5, …0101
result = (number << 2) + (number << 1);   // multiply by 6, …0110
result = (number << 3) + number;          // multiply by 9, …1001
result = (number << 3) + (number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

Pointers: example data





Example instruction:



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

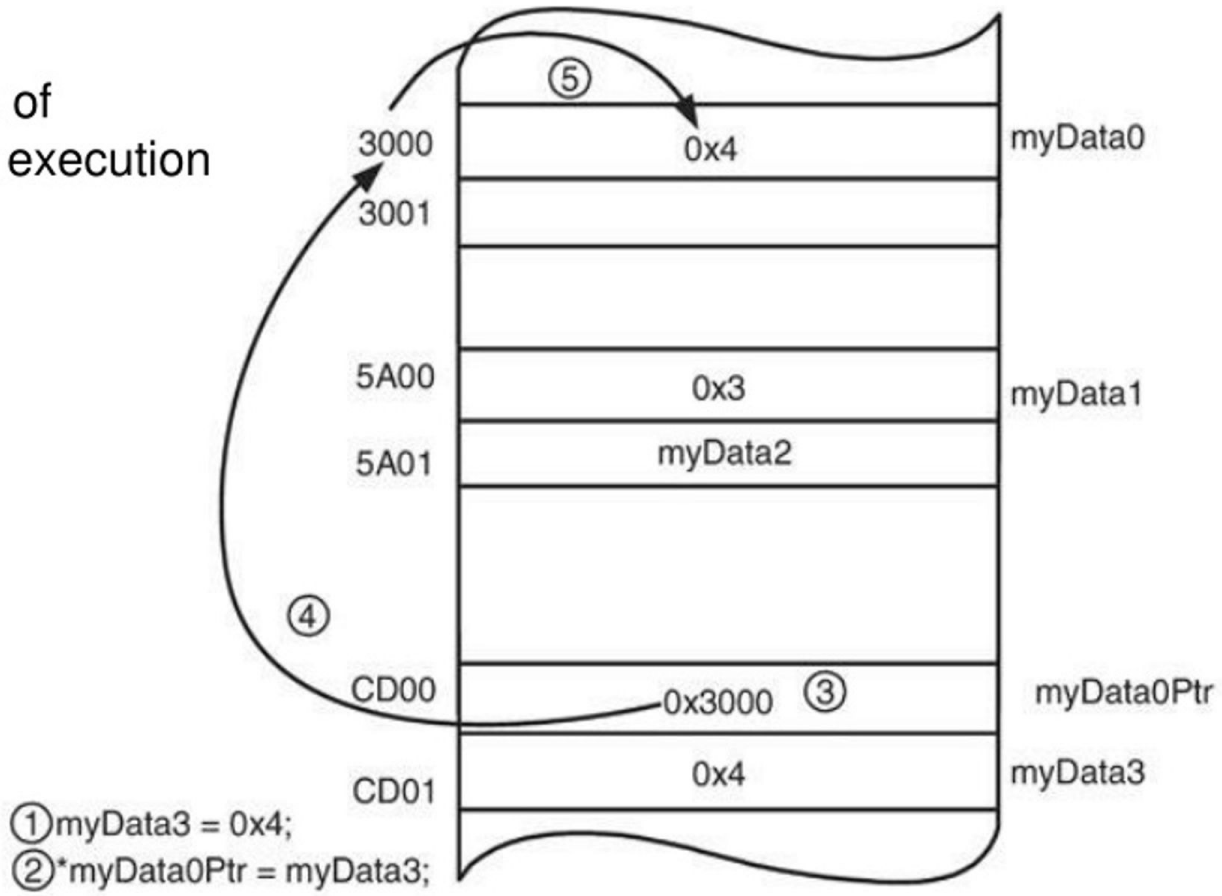
② Go to the address (0x3000) specified, pointed, or referred to by that value.

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

Example 2 of
Instruction execution



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

Example: what is output?

```
/*
 * 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

Pointer arithmetic—not for beginners!

```
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

Pointer arithmetic: additional examples

```
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.

fig_07_12b

Constant pointers:

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

Note: pointers can be “generic” or NULL:

Generic:
Type void: pointer can point to a variable of any type

example: 7.3, p. 265

NULL:
Pointer needs to be assigned a valid address before dereferencing, otherwise hard-to-find bugs can occur

Address 0 is not acceptable

Solution: use null pointer address, (void*) 0

Note: pointers can be “generic” or NULL:

Generic:
Type void: pointer can point to a variable of any type

example: 7.3, p. 265

NULL:
Pointer needs to be assigned a valid address before dereferencing, otherwise hard-to-find bugs can occur

Address 0 is not acceptable

Solution: use null pointer address, (void*) 0

C functions:

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;
}
```

Example:

```
#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

Function call: note order of parameters on stack

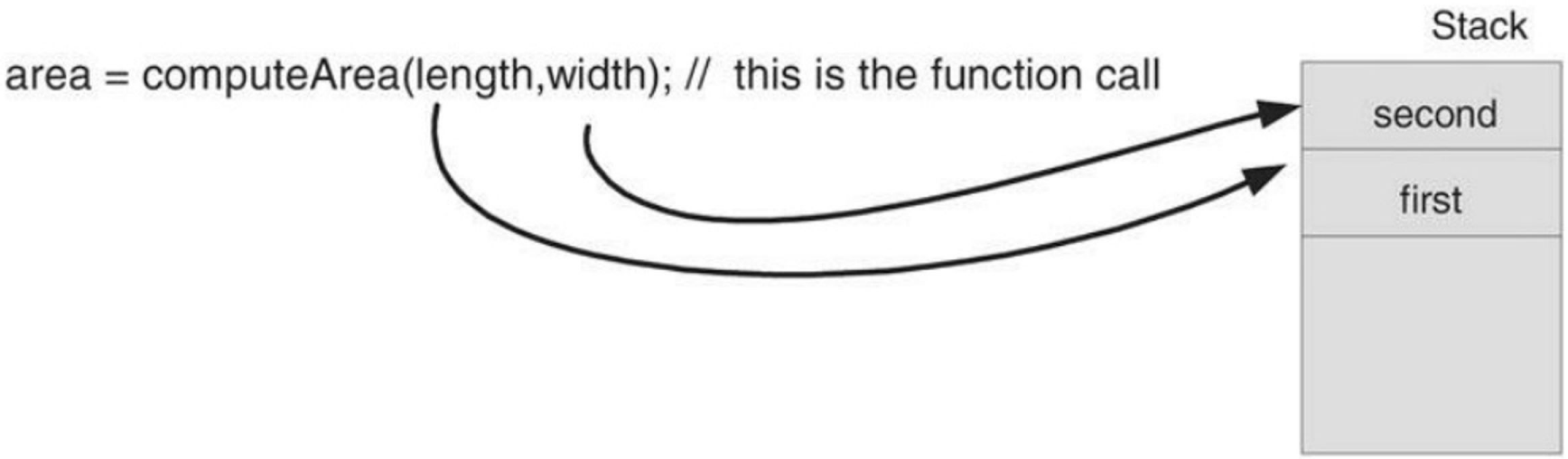


fig_07_17

Using stack to return result:

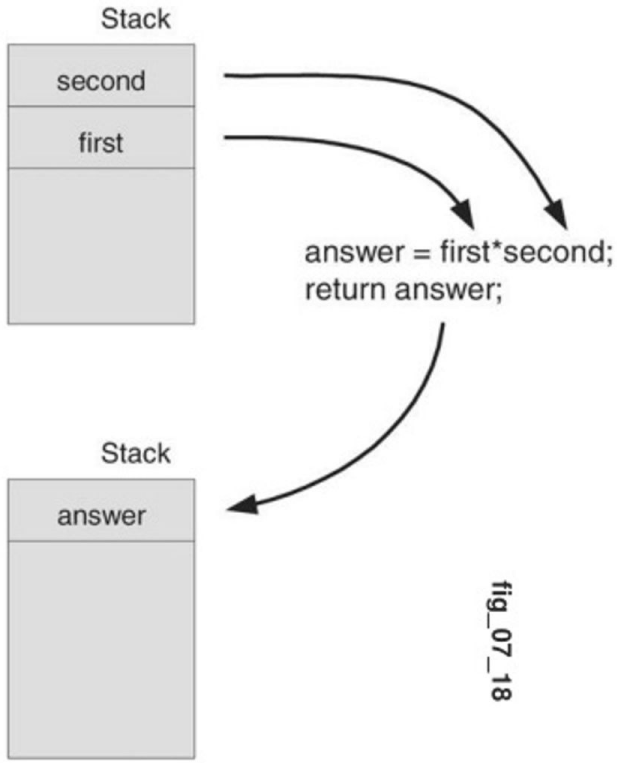
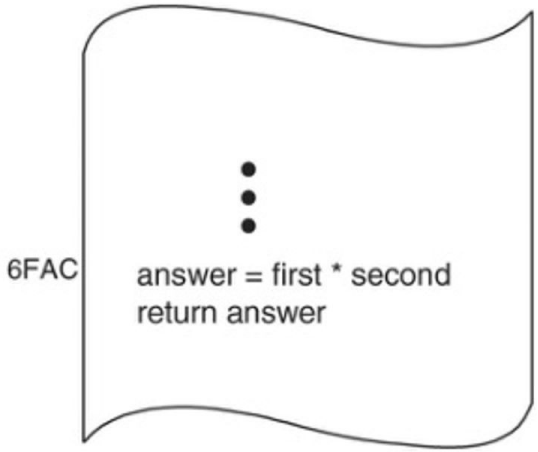


fig_07_18

Function body in memory:



Pass by value (default in c):

```
#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

Pass by reference:

```
#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

Information hiding: static qualifier prevents visibility outside this file (even to linker):

```
// 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;
}
```

```
// 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 documentation: template for header

```
/*
 * 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
 */
```

We can also define pointers to functions:

Syntax

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

arg list may be empty

Dereferencing methods:

syntax

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

or

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

arg list may be empty

Example:

```
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
```

Example 2:

```
//     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

Pointing to add:

```
// 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
```

Pointing to subtract: pointer does not know functionality, only address

```
// 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
```

User-defined data structure: struct
(NOT “object” or “class”; what is the difference?)

syntax

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

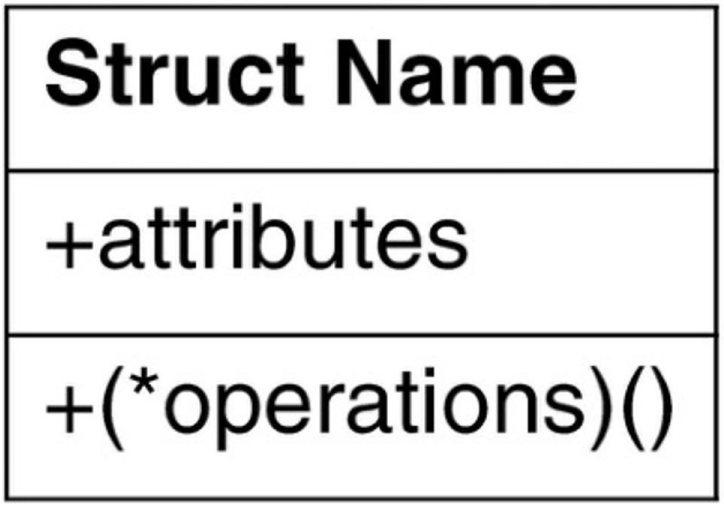


fig_07_31

User-defined data structure: struct

Example:

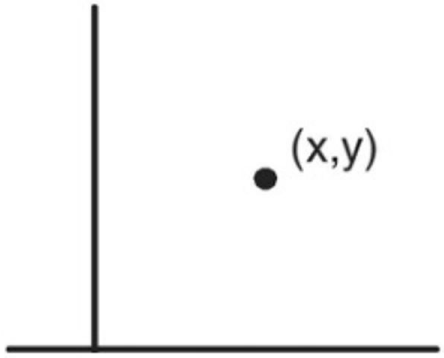


fig_07_30

Point
+x : int
+y : int

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

Can also define a type and use it repeatedly:

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

fig_07_34

Syntax for using struct:

syntax

```
struct StructTag
{
        struct body;
};

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

Example: define a rectangle

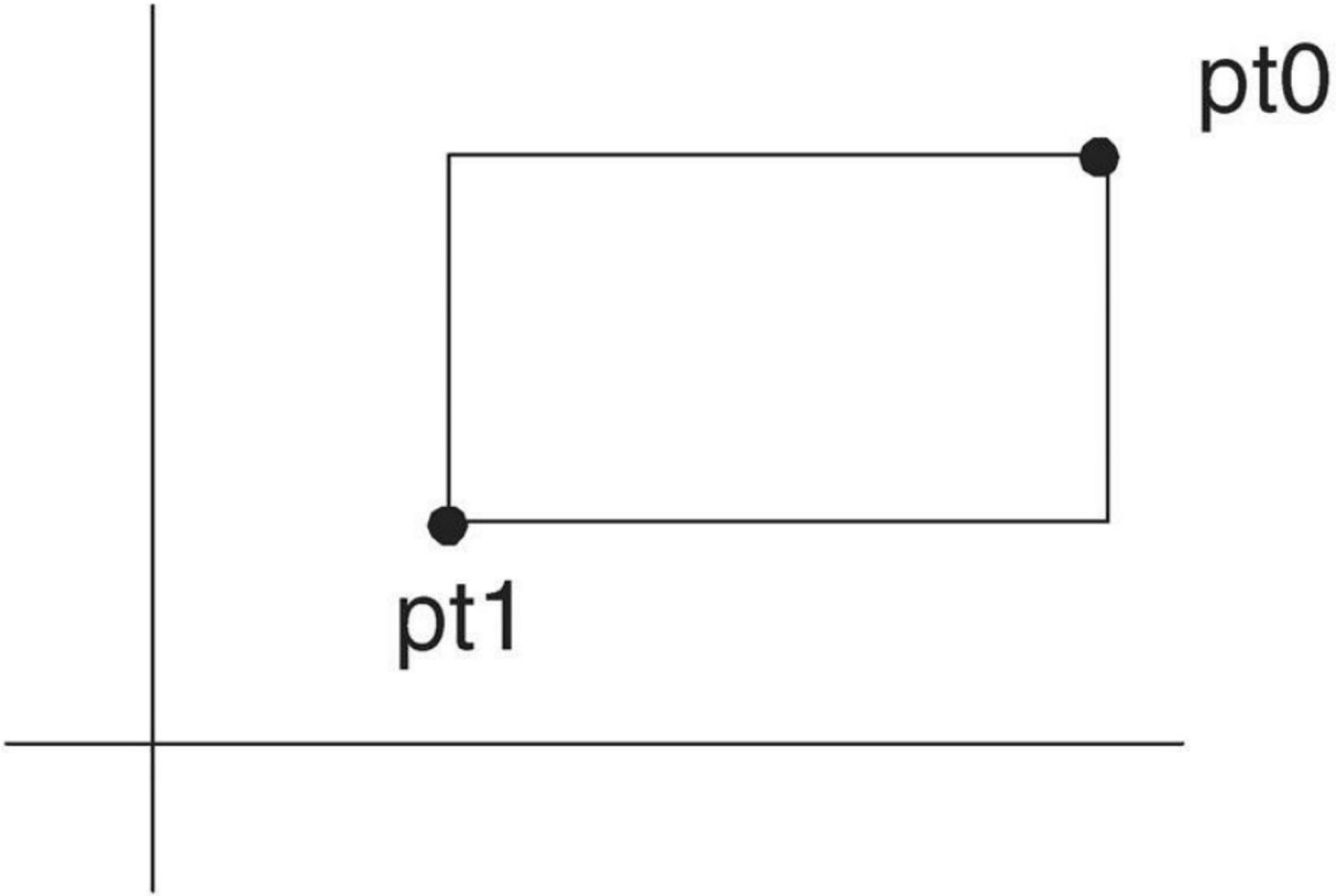


fig_07_36

One struct using another:

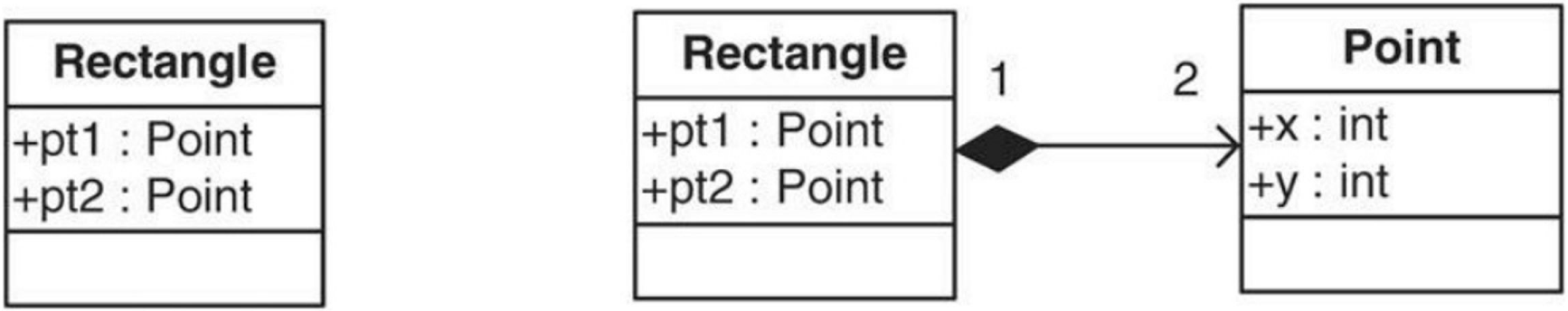


fig_07_37

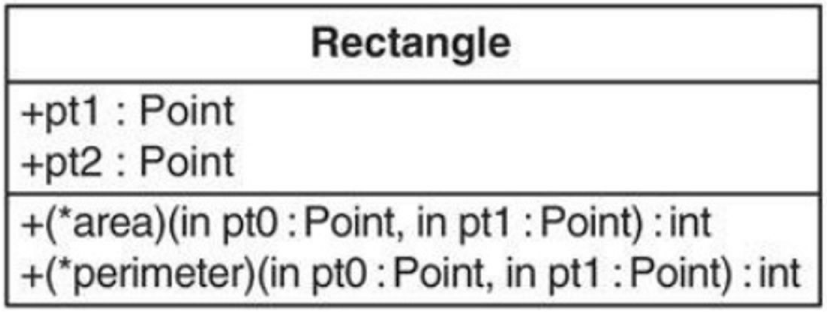
C code for this example:

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

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

fig_07_38

Defining the rectangle type:




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

Using point and rectangle definitions:

```
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

Rectangle functions:

```
#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;
}
```

fig_07_41

Example program:

```
#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;
}
```

Example: passing a struct to a function:

```
// 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;
}
```

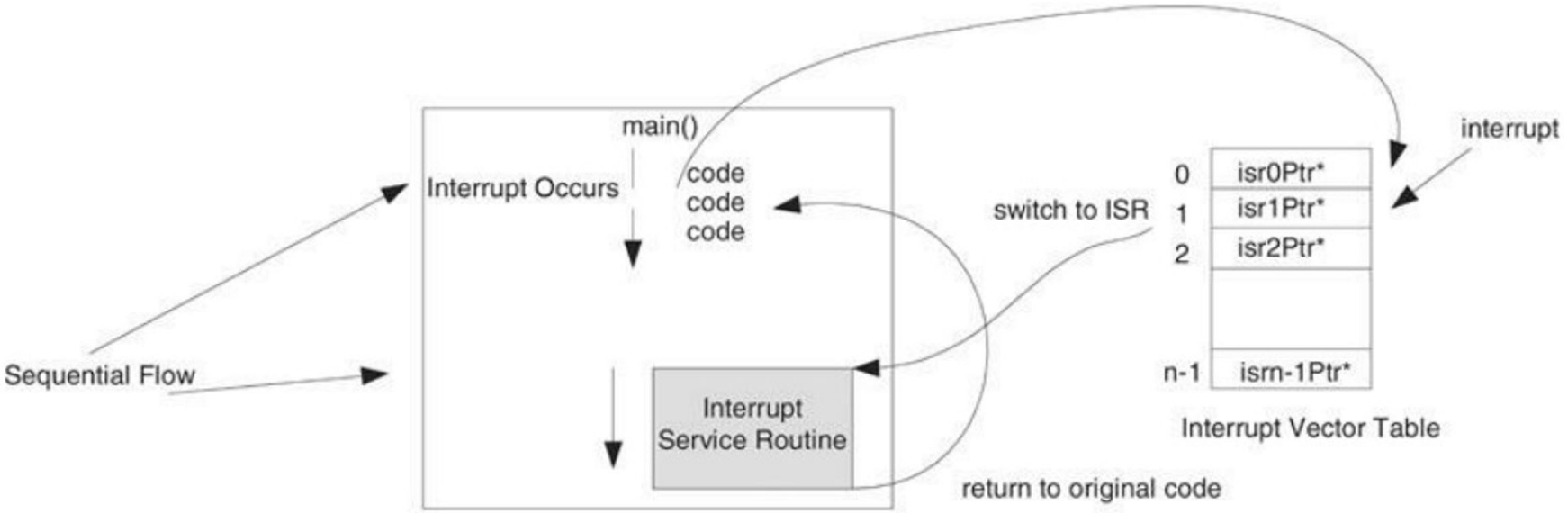
fig_07_43

Interrupt service routines: ISR—needs to be SHORT and SIMPLE

```
void ISRName(void)
{
        body
}
```

Interrupts

How an interrupt occurs:



Interrupt may be disabled under certain conditions

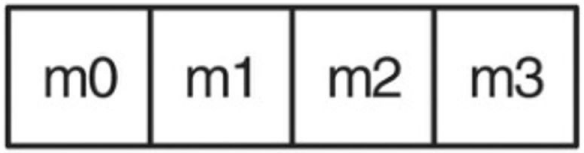
fig_07_45

Enable / disable control mechanisms:

Global

Masking

method 1: use priorities
method 2: use mask register
(bitwise masks)



Note: interrupts are transient, if we choose to ignore one we may not be able to service it later