Multiple Source Files

- Programs can be made up of multiple source files.
- One source file must contain the main() function as the entry point for execution.
- Advantages:
  - Grouping related files together helps to clarify the structure of a program
  - Each source file can be compiled separately (and can be checked out separately) – time saver.
  - Functions can be more easily re-used in other programs when grouped in separate source files.

Header Files (1)

- How can a function in one file:
  - call a function in another file or
  - use macros (#define directives) in another file or
  - or use external variables in another file?
- The #include directive tells the compiler to open another specified file and insert its contents into the current file.
- By convention header files have the extension .h.

Header Files (2)

- For C libraries:

  #include <stdio.h>

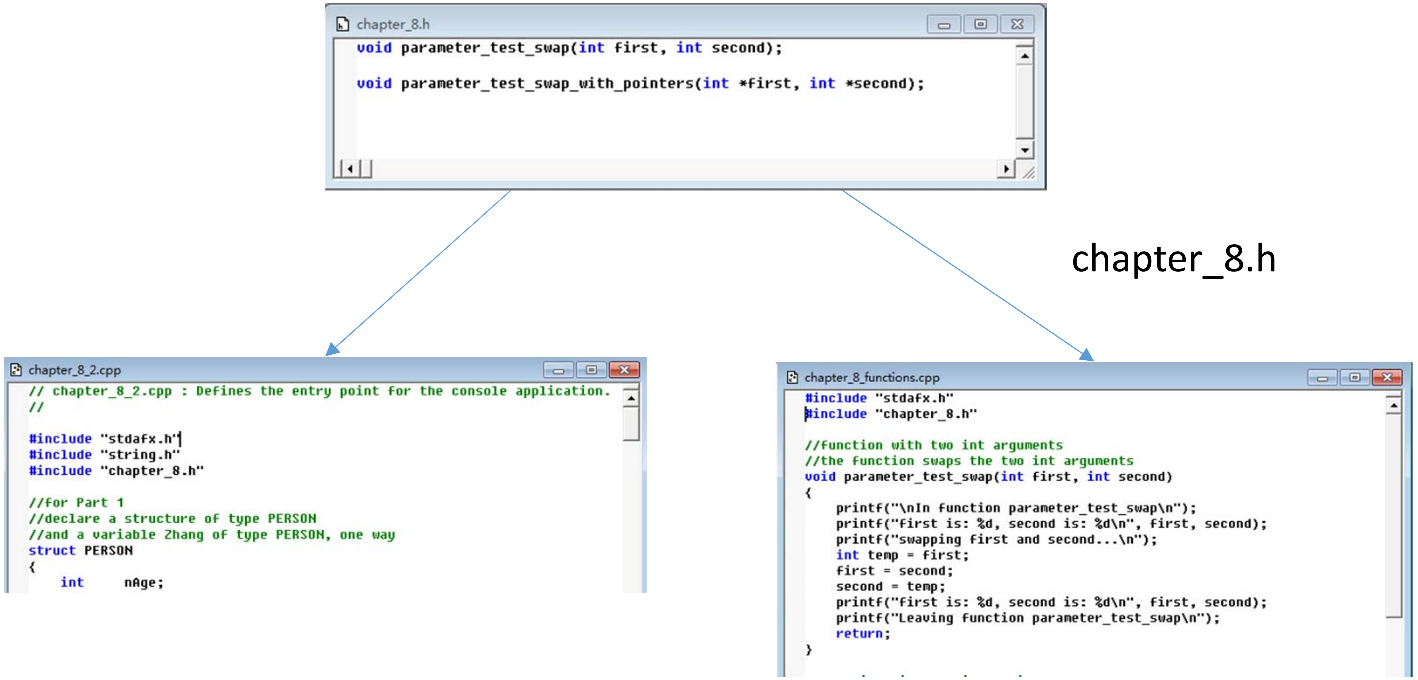
  Compiler searches in the directory for system header files.

- For other header files:

  #include “random.h”

  Compiler searches in the current directory, any specified directory and then the directory for system header files.

Example



chapter_8.h

chapter_8_2.cpp

chapter_8_2_functions.cpp

Sharing Function Prototypes

- The functions parameter_test_swap(int first, int second) and parameter_test_swap_with_pointers(int *first, int *second) have been moved to a separate file chapter8_2_functions.cpp.
- The program chapter8_2.cpp uses theses functions.
- Before calling a function the compiler needs to see the prototype of the function first.
- Therefore include the prototypes in a header file (here: chapter_8.h)
- Also include the header file in the source file in which the function is defined so that the compiler can check that the function's prototype matches the definition.

Bitwise Operators

- Basic binary encoding
- Bitwise operators
  - Left shift
  - Right shift
  - Bitwise complement
  - Bitwise AND
  - Bitwise exclusive OR
  - Bitwise inclusive OR
- Very Briefly: using bitwise operators to access bits
  - Setting a bit
  - Clearing a bit
  - Toggling a bit
  - Checking a bit
- Named bits

Motivation

- Low-level computing allows us to hold and manipulate several states in one variable.
- We **use individual bits** of an unsigned int variable to store data.
- In embedded systems it is useful to have fewer variables to save memory.
- In certain areas of Computing, e.g. image processing there is a lot of information to deal with, e.g. the colour of a pixel can be encoded in its RGB components. We could use one int to store the 3 colours rather than three ints.
- Advantages: faster processing, less memory needed, simpler (at least some very experienced programmers say that).

Binary Encoding

- The integer value 10 as unsigned short int:

15									6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

- To calculate the decimal value: $1 * 2^3 + 1 * 2^1$

Decimal to Binary Number Conversion

- Decimal value: 10
- Divide the number by the highest power of 2
- Then divide the remainder by the highest power of 2
- ...10/ 2^6, 10/ 2^5, 10/ 2^4 : result < 1 set bits to 0
- 10/ 2^3 = 1, 10%2 = 2 continue with remainder
- 2/ 2^2 : result < 1 set bit to 0
- 2/ 2^1 = 1
- remaining bit is 0 (number is fully converted)

Bitwise Operators

- <<   left shift
- >>   right shift
- ~   one's complement (unary)
- &   bitwise AND
- ^   bitwise exclusive OR
- | bitwise inclusive OR

Bitwise Shift Operators

- Transform the binary representation of an integer by shifting its bits to the left or to the right.
- Can be used on any integer type including char.
- Have lower priority than arithmetic operators.
- Examples:

0000000000001010 – decimal value ?

0000000000101000 – shift how many places?

0000000000000010 – shift how many places?

The Left Shift Operator

- Operator: <<
- Shifts the bits of an integer by the stated number of places.
- The non-negative variable j represents this here.
- For each bit that is shifted off the left, another 0 bit is added to the right
- Example:

  i << j;

  Shifts the bits in i by j places to the left.

The Right Shift Operator

- Operator: >>
- Shifts the bits of an integer by the stated number of places (j).

- Example:

  i >> j;

  Shift the bits in i by j places to the right.

Right Shift Portability Concerns

- For portability it is best to perform shift operations on unsigned numbers (positive numbers).
- If i is of an unsigned type or if the value of i is non-negative, zeros are added at the left as needed.
- We use unsigned short int.
- If i is a negative number, the result is implementation-defined,
  - some implementation add zeros at the left end
  - other preserve the signed bit by adding ones.

Examples

C code

```
unsigned short i = 5;
unsigned short j = i << 3;
unsigned short k = j >> 1;
i <<= 2;
i >>= 1;
unsigned short t = i << 2 + 1;
//the left shift operator has
// lower priority than the +
//operator
```

Binary representation

i: 0000000000000101

j: 0000000000101000

k: 0000000000010100

i: 0000000000010100

i: 0000000000001010

t: 0000000001010000

The Bitwise Complement Operation

- The operation produces the complement of its operand.
- Example:

unsigned int i = 10;

~ i

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1

The Bitwise Complement Operation

- Operator: ~ (tilde, twiddle)
- The operator produces the complement of its operand:
  - 0 is replaced by 1
  - 1 is replaced by 0
- The operator is unary (only one operand).

Example

- unsigned short i = 10;
- binary representation: 00000000 00001010
- unsigned short j = ~ i;
- binary representation: 1111111111110101

The Bitwise AND Operator (1)

- The bitwise AND operator:
  - Compares each bit of its first operand to the corresponding bit of the second operand.
  - If both bits are 1's,
    - the corresponding bit of the result is set to 1.
  - Otherwise,
    - it sets the corresponding result bit to 0.

The Bitwise AND Operator (2)

- Operator: & (ampersand)
- Both operands have to be of an integer type.
- The result has the same type as the converted operands.
- An expression can have more than one bitwise AND operator.

Example

C source code

```
unsigned short i = 92;
unsigned short j = 46;
unsigned short k = i & j;
//k is 12

////******////
short l = 1 & 4;
```

What is the binary value of l?

What is the decimal value of l?

Bit representation

0000000001011100

0000000000101110

0000000000001100

& vs. &&

- Do not mix up the logical AND operator (&&) with the bitwise AND operator (&)
- Example:

  1 && 4 evaluates to non-zero (true)

  1 & 4 ?

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1&4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The Bitwise Exclusive OR Operator (1)

- The bitwise exclusive OR operator:
  - Compares each bit of its first operand to the corresponding bit of the second operand.
  - If both bits are 1's or both bits are 0's,
    - the corresponding bit of the result is set to 0.
  - Otherwise,
    - it sets the corresponding result bit to 1.

The Bitwise Exclusive OR Operator (2)

- Operator: ^ (caret)
- Both operands must have an integer type.
- The result has the same type as the converted operands.
- The result is not an lvalue.

Example

C source code

- unsigned short i = 92;
- unsigned short j = 46;
- unsigned short k = i ^ j;

  //k is 114

Bit representation

0000000001011100

0000000000101110

0000000001110010

The Bitwise Inclusive OR Operator (1)

- The bitwise inclusive OR operator:
  - Compares the values (in binary format) of each operand.
  - If both bits are 0,
    - The result of that bit is 0;
  - Otherwise,
    - The result is 1.
- The bit pattern shows which bits in either of the operands has the value 1.

The Bitwise Inclusive OR Operator (2)

- Operator: | (pipe)
- Both operands must have an integral (or enumeration type).
- The result has the same type as the converted operands.
- The result is not an lvalue.

Example

C source code	Bit representation
• unsigned short i = 92;	0000000001011100
• unsigned short j = 46;	0000000000101110
• unsigned short k = i | j; //k is 126	0000000001111110

|| vs. |

- The bitwise OR operator (|) should not be confused with the logical OR operator (||).
- 1 | 4 evaluates to 5.

  0000000000000001

  0000000000000100

  0000000000000101
- 1 || 4 evaluates to non-zero (true).

Precedence

- Highest: << >> (explain the operation)
  ~ (explain the operation)
  & (explain the operation)
  ^ (explain the operation)
- Lowest: | (explain the operation)
- Always use brackets to force precedence.

Complex Expressions

C source code	**Binary representation**
unsigned short i = 10;	0000000000001010
unsigned short j = 8;	0000000000001000
unsigned short k = 11;	0000000000001011
unsigned short r = (i ^ j) & ~ k;	

0000000000000010 i ^ j

1111111111110100 ~k

0000000000000000 i ^ j & ~ k

Be Aware!

- The precedence of &, ^ and | is lower than the precedence of the relational and equality operators.
- Example:

  if (i & 7 != 0) {....}

i & 1

If i has the 1 bit set this is non-zero (true).

Using Bitwise Operators to Access Bits

- In low level programming certain information is often stored in single bits or a collection of bits.
- For example one (16 bit) unsigned short variable could be used to store 16 binary states.
- We number the bits from leftmost bit (# 15) to rightmost bit (# 0).

Setting a Bit (1)

- Assume we have a 16 bit unsigned short i
- We want to set the # 4 bit of i:

	15															0
i	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1

- Apply the bitwise inclusive OR operator with the

mask that sets the 4-bit 0000000000010000

i |= j 0000000010110001

- What happens if the bit is already set?
- What happens to other set bits?

Setting a Bit (2)

- If the position of the bit to be set is stored in the variable j (j = 4)
- We can use the shift operator to create a mask:

i |= 1 << j;

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1 << 4	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Clearing a Bit

- To clear the # 4 bit:
- Use a mask with a 0 bit in position 4 and 1 bits everywhere else.
- Apply the bitwise AND operator:

• int i = 255;	0000000011111111
• ~ (1 << j)	1111111111101111
• i & = ~ (1 << j)	0000000011101111

Testing a Bit (1)

- If the bit (here # 4) has been set (i.e. is 1) the expression should evaluate to non-zero
- If the bit (here # 4) has not been set (i.e. is 0) the expression should evaluate to zero
- Use a mask with a 1 in position 4 and 0 everywhere else.
- Then apply the bitwise AND operator:
  - Only the (# 4) bit is 1 the result will be non-zero.

Testing a Bit (2)

- if (i & 1 << j) {...}

i = 239;	0000000011101111
1	0000000000000001
1 << 4	0000000000010000
i & 1 << 4	0000000000000000

The result is 0 (i.e. not set).

To “Toggle” a Bit (1)

- To toggle means to “flip” the bit or to switch it on or off.
- We can do this by applying the exclusive OR operator ^ with a mask that has set the bits that we want to toggle.
- Use the left-shift operator to create a mask that has set the bits that we want to toggle.

“Toggle” a Bit (2)

C source code	Bitwise representation
• unsigned short int i:	0000000001100010
• mask, unsigned short int j:	0000000000011010
• i ^ j :	0000000001111000

- **If a relevant bit was set before the exclusive OR operation, it will be unset afterwards (1 ^ 1 = 0)**
- **If a relevant bit was unset before the exclusive OR operation, it will be set afterwards (0 ^ 1 = 1)**
- **The mask has only the relevant bits set, all non-relevant bits are 0, these will keep their state (0 ^ 0 = 0, 1 ^ 0 = 1)**

Named Bits

Bits with a significant meaning are often defined as macros.

Example:

The bits 0, 1 and 2 of a number represent the colours red, green and blue:

```
#define BLUE 1
#define GREEN 2
#define RED 4
```

No semicolons after #define

The BLUE Bit

- The int i holds the settings of RED, GREEN, BLUE.

- Setting BLUE:

```
i |= BLUE;
```

- Clearing BLUE:

```
i &= ~ BLUE;
```

- Testing BLUE:

```
if (i & BLUE) ...
```