Tomasz Müldner

C for Java Programmers

Copyright: Addison-Wesley Publishing Company, 2000

Chapter 1:

Introduction

Copyright: Addison-Wesley Publishing Company, 2000

Introduction: Preview

- History of C.
- Two programming paradigms; objected oriented and procedural.
- Comparison of Java and C.
- Software development process.
- Programming idiom.

Copyright: Addison-Wesley Publishing Company, 2000

Introduction: About C

- 1972: The C programming language developed by Dennis Ritchie
- 1973: C used to write the Unix operating system.
- 1978: Kernighan and Ritchie set the first standard of C
  K&R standard
- 1989: the American National Standards Institute adopted the
  ANSI C standard

Copyright: Addison-Wesley Publishing Company, 2000

Introduction: Programming paradigms

- **object-oriented programming paradigm**, based on the following design rule:

  decide which classes you need, provide a full set of operations for each class, and make commonality explicit by using inheritance.

- **procedural programming paradigm**, based on the following design rule:

  decide which procedures and data structures you want and use the best algorithms.

Copyright: Addison-Wesley Publishing Company, 2000

Introduction: Comparison of C and Java

- *primitive data types*: character, integer, and real
  In C, they are of different sizes,
  there is no Unicode 16-bit character set
- *structured data types*: arrays, structures and unions.
  In C, arrays are static
  there are no classes
- *Control structures* are similar
- Functions are similar

Copyright: Addison-Wesley Publishing Company, 2000

Introduction: Comparison of C and Java

- Java references are called pointers in C.
- Java constructs missing in C:
  - packages
  - threads
  - exception handling
  - garbage collection
  - standard Graphical User Interface (GUI)
  - built-in definition of a string
  - standard support for networking
  - support for program safety.

Introduction: Why is C still useful?

C provides:
- efficiency
- flexibility and power
- many high-level and low-level operations
- stability

C is used:
- data compression, graphics and computational geometry
- databases, operating systems
- there are zillions of lines of C legacy code

Introduction: Development Process

Four stages
- **preprocessing**
- editing
- compiling: translates source code -> object code
- linking: produces executable code

- **Portable** programs will run on any machine.

- Program **correctness** and **robustness** are most important than program *efficiency*

Introduction: Programming Style and Idioms

- **Programming style** includes recommendations for:
  - lexical conventions
  - a convention for writing comments

- In a natural language, an **idiom** is a phrase that has a specific meaning such as
  "don't pull my leg"

```
while(*p++ = *q++)
    ;
```

You'll learn how to *identify, create* and *use* idioms.

Chapter 2:

Example of a C Program

2: General Documentation

- Comments in C are similar to those in Java.
- There is no standard tool such as `javadoc` to produce documentation in HTML form.
- An "in-line" comment, starting with `//`, is not supported.
- The heading comment that starts the sample program follows a particular style convention used throughout this book. It always describes the intended meaning of the program, any known bugs, etc.

```
/* Author:      Tomasz Muldner
 * Date:        August, 1999
 * Version:     2.0
 * File:        Sample.c
 * Program that expects one or two filenames on the
 *  command line and produces a hexadecimal dump of the
 *  file whose name is passed as the first argument. If
 *  the second argument is present, the dump is stored in
 *  the file whose name is passed as this argument.
 * Otherwise, the dump is displayed on the screen.
 *  The format of the dump is as follows:
 *  Each line contains 16 hexadecimal ASCII codes of the
 *  corresponding 16 bytes read from the file; separated
 *  by a blank. This line is followed by a line containing
 *  the 16 corresponding characters, again separated by a
 *  blank, with each non-printable character displayed
 *  as a dot and other characters unchanged.
 */
```


2: Run-time Libraries and Function Declarations

- Include directives, starting with `#include`, are needed in C programs to include files, called **header** files, which typically have the "`.h`" extension.
- In my example, three files are included, respectively called `stdio.h`, `ctype.h` and `stdlib.h`.
- These header files contain declarations of various standard functions that are defined in the run-time libraries.

2: Global Definitions

Global definitions define entities (such as variables),

which are available to all the code that follows this definition.

In my example there is a single global definition:

```
FILE *outFile;
```

that defines a file handle, available to all the code that follows it, both the `hex()` and `main()` functions.


```
/* include files */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

/* global definitions */
FILE *outFile;      /* output file */
```


2: Function Declaration and Definition

- A **declaration** merely provides a function prototype:

```
void hex(unsigned char *p, int max);
```

- The declaration does not say anything about the implementation
- The **definition** of a function includes both the function prototype and the function body, where the *body* is the implementation of the function


```
/* Function declaration */
/*
 * Function: hex(p, max)
 * Purpose: writes to the global output file
 *   outFile the hexadecimal codes of max number of
 *   characters originating at the address given by the
 *   pointer p. This line is followed by the
 *   corresponding chars.
 *   Assumes that the file outFile has been
 *   opened for output.
 * Inputs:      p, max (parameters)
 *              outFile (global variable)
 * Returns:     nothing
 * Modifies:    outFile
 * Error checking:      none
 */
void hex(unsigned char *p, int max);
```

2: The main Function

Every C program must include a function called `main`:

```
int main(int argc, char *argv[])
```

`main()` is an integer function, and returns one of two standard return codes:

`EXIT_FAILURE`
`EXIT_SUCCESS`.


```
int main(int argc, char *argv[]) {
   FILE *inFile;               /* input file handle */
   int i, toFile;
   const int SIZE = 16;
   unsigned char line[SIZE];  /* local buffer */

   if(argc > 3 || argc < 2) {
      fprintf(stderr, "usage: %s filename [filename2]\n",
         argv[0]);
      return EXIT_FAILURE;
   }
   outFile = stdout;     /* set default output stream */
   toFile = (argc == 3); /* is there an output file */
   /* open I/O files */
   if((inFile = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "Cannot open file %s\n", argv[1]);
      return EXIT_FAILURE;
   }
```



```
  if(toFile && (outFile = fopen(argv[2], "w")) == NULL) {
      fprintf(stderr, "Cannot open file %s\n", argv[2]);
      fclose(inFile);
      return EXIT_FAILURE;
   }
   /* main loop; reads SIZE bytes at a time;
    * stores them in line, and calls hex()
    */
   while((i = fread(line, 1, SIZE, inFile)) > 0)
      hex(line, i);

   /* close I/O */
   if(fclose(inFile) == EOF) {
      fprintf(stderr, "Cannot close file %s\n", argv[1]);
      if(toFile)
         fclose(outFile);
      return EXIT_FAILURE;
   }
```



```
   if(toFile && fclose(outFile) == EOF) {
      fprintf(stderr, "Cannot close file %s\n", argv[2]);
      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```



```
/* Definition of hex() */
void hex(unsigned char *p, int max) {
   int i;
   unsigned char *paux;

   for(i = 0, paux = p; i < max; i++, paux++)
    fprintf(outFile, "%02x ", *paux);
    fputc('\n', outFile);

   for(i = 0, paux = p; i < max; i++, paux++)
    fprintf(outFile, "%c  ", isprint(*paux) ? *paux :
  '.');
   fputc('\n', outFile);
}
```


Chapter 3:

Lexical Structure, Primitive Data Types and Terminal I/0

3: Preview

- primitive data types: `int`, `char`, `float`, `double`, and their variants, such as `long double`.
- expressions, including the assignment expressions
- basic terminal I/O
- type conversions and ways of defining synonyms for existing data types.

3: Lexical Structure

- C is a free format language:

  any sequence of **whitespace** characters can be used in the place of a single whitespace character

- A program is **lexically correct** if it follows the lexical structure of the language:

  it contains only characters from the language's alphabet

  all its tokens are built according to the language's rules.

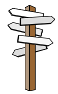
3: Comments

```
/* Hello */
```

Wrong:

```
/* outer /* inner */ */

//
/**  */
```


Programming Guidelines



Comments

```
if(isdigit)  /* error */

/*
 * Program to sort integer values
 */

k++; /* k is incremented by 1*/
```


Programming Guidelines

Comments and Identifiers

- Make every comment count.
- Don't over-comment.
- Make sure comments and code agree.

3: Line Structure, Identifiers

- Two lines

```
my\
 id
```

are *joined* during compiler translation into

```
myid
```

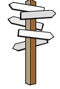
- **An identifier** is a sequence of letters, digits, and underscores that does not start with a digit
  (case-sensitive, but only the first 31 characters are *significant*)

Portability

Identifiers

- The documentation of programs, *which are available to clients*, should make it clear whether you are using identifiers whose length exceeds **6** characters.
- Never use identifiers that have more than **31** characters.

Programming Guidelines



Identifiers

- Use a *consistent* style throughout your code.
- Variables should have meaningful names *when appropriate*.
- You are encouraged to mix case to make your identifiers more readable:

```
longIdentifier
```


3: Primitive Data Types

C provides several primitive data types: `char`, `int`, `float` and `double`.

- No built-in Boolean type; instead use `int`:
  the value 0 stands for false,
  any non-zero value stands for true
- No guarantee that a specific amount of memory will be allocated to a particular data type.

  All implementations of C must follow certain rules.

3: Memory and Range of Values

- Computer memory consists of **words**;
  each word consists of a number of **bytes**,
  a byte consists of a number of **bits** (usually 8 bits).
- **Signed integers** use the leftmost bit (sign bit), to represent the sign.
  The largest unsigned 16-bit integer: $2^{16} - 1$
  The largest signed 16-bit integer: $2^{15} - 1$
- C provides a header file `limits.h`, it defines e.g.
  `CHAR_BIT` - the width of `char` type in bits (>= 8)
  `INT_MAX` is the maximum value of `int` (>= 32,767).

3: Integer Types

- plain, signed and unigned

```
short unsigned int
signed long
int
```

- size(`short`) <= size(`int`) <= size(`long`)
- Arithmetic operations on unsigned integers follow the rules of arithmetic modulo 2^n.
  Therefore, they can not produce an *overflow*

Portability

Ranges

To support portability, use only integers in the ranges specified in `limits.h`

For example, you can always use plain integers in the range from -32,767 to 32,767

Any other assumptions, such as that the size of an `int` is 4 bytes, must not be made.

Errors

Overflow

In order to check whether the sum of two integers i and *overflows*,
do not use

```
i + j > INT_MAX
```

Instead, use

```
i > INT_MAX - j
```


3: Character Types

- There are three character data types in C:
  (plain) `char`
  `unsigned char`
  `signed char`
- Character types are actually represented internally as integers (unsigned or signed).
- Two popular character sets:
  ASCII
  (American Standard Code for Information Interchange)
  EBCDIC (used by IBM computers)

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Portability

Characters

- In order to write portable code, you should explicitly specify `signed char` or `unsigned char`.
- You should *never* make any assumptions about the code values of characters, such as that the value of `'A'` is 65.

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

3: Floating-point Types

```
float
double
long double
```

Use `float.h` for sizes and ranges.

Only guarantee:

```
size(float) <= size(double) <= size(long double)
```

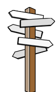
C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

3: Declarations of Variables and Constants

```
int i;    /* initial value undefined */
double d = 1.23;

const double PI = 3.1415926;
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines



Declarations

- A declaration of a different data type starts on a new line.
- The lists of identifiers are aligned to start in the same column.
- Each variable identifier is followed by one space.
- Each declaration that contains an initialization appear on a separate line:

```
int i = 1;
int j = 2;
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines

Declarations

- If comments are required, each declaration should be placed on a separate line:

```
int lower;     /* lower bound of the array */
int upper;     /* upper bound of the array */
```

- Names of constants will appear in upper case and `const` always appears in front of the declaration.

 Copyright: Addison-Wesley Publishing Company, 2000

3: Assignments

An **l-value:**

an expression that can be interpreted as an *address*.

Examples:
a variable is an l-value,
a constant is not.

```
x = y = z;
/* exactly one space before and after = */
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: The Main Program

- A program consists of one or more functions.
- There must be exactly one occurrence of `main():`

```
int main(int argc, char *argv[])

int main()
```

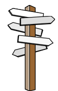
- `main()` is an integer function, which returns one of two standard codes: `EXIT_FAILURE` and `EXIT_SUCCESS`

  (defined in the standard library `stdlib.h`)

  Other values will not be *portable*.

 Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines



Main

The main function looks as follows:

`int main() {`	the left brace on the same line as the header
`body`	body indented to the right
`}`	the matching right brace aligned with `int`

 Copyright: Addison-Wesley Publishing Company, 2000

Idioms

Main Function

`main()` is an integer function, which returns one of two standard return codes:

```
EXIT_FAILURE
EXIT_SUCCESS
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Literal Constants

- integer: `123   47857587L`
- floating point: `12.66   23478.78899E-20`
- character: `'\n'  'd'`
- string: `"abc"`

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

String and Character Constants

String constants and character constants are very different:

```
"a"

'a'
```


3: Expressions

logical AND, as in	`e1 && e2`
logical OR, as in	`e1 || e2`
a conditional expression, as in	`e1 ? e2 : e3`
a comma expression, as in	`e1, e2`

Errors

Associativity

The expression

```
a < b < c
```

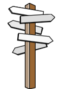
is interpreted as

```
a < b < c
```

and has a different meaning than

```
a < b && b < c
```


Programming Guidelines



Spacing

Don't use spaces around the following operators:

```
-> .   []   !   ~   ++   --   - (unary)
* (unary)   &
```

In *general*, use one space around the following operators:

```
=  +=  ?:  +  <  &&  + (binary) etc.
```

Examples

```
a->b        a[i]       *c
a = a + 2;
a= b+ 1;
a = a+b * 2;
```


3: Terminal I/O

```
#include <stdio.h>
```

- **Standard input** is usually a keyboard, unless it has been redirected
- **Standard output** is usually the screen, unless it has been redirected
- `EOF` is a constant defined in `stdio.h`; it stands for End-Of-File.
- I/O redirection:

```
F < test.in > test.out
```


3: Single Character I/O

`int getchar()`	to input a single character
`int putchar(int)`	to output a single character

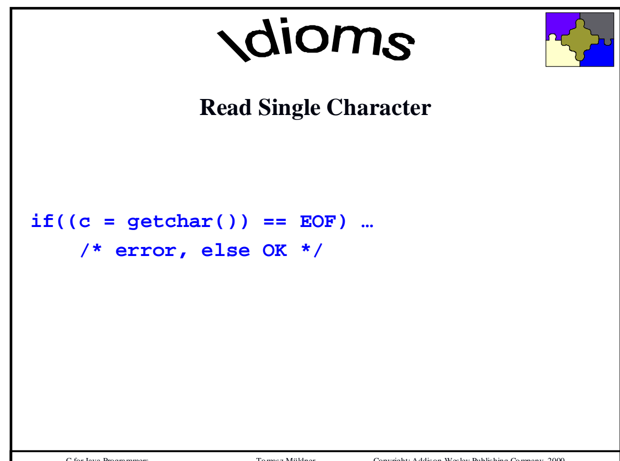
```c
/* File: ex1.c
 * Program that reads a single character and
 * outputs it, followed by end-of-line
 */
#include <stdio.h>
#include <stdlib.h>
int main() {
   int c;    /* chars must be read as ints */

   c = getchar();
   putchar(c);
   putchar('\n');

   return EXIT_SUCCESS;
}
```

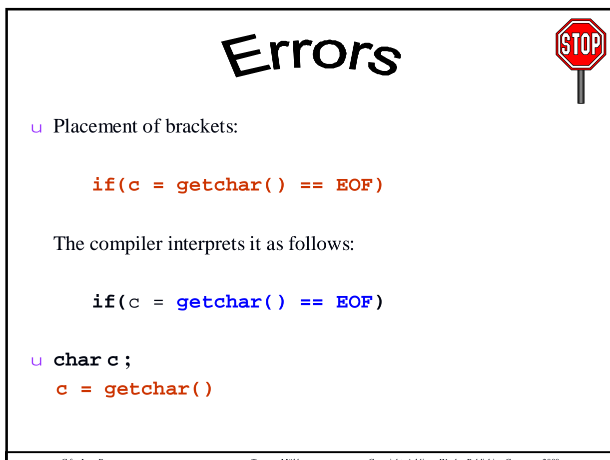
Include statements will not be shown in other examples


```c
/*
 * Program that reads two characters and
 * prints them in reverse order, separated by
 * a tab and ending with end of line.
 * Error checking: Program terminates if
 * either of the input operations fails.
 *  No error checking for the output
 */
int main() {
   int c, d;

   if((c = getchar()) == EOF)
      return EXIT_FAILURE; /* no output! */
```

Idiom?


```c
   if((d = getchar()) == EOF)
      return EXIT_FAILURE;

   putchar(d);
   putchar('\t');
   putchar(c);
   putchar('\n');

   return EXIT_SUCCESS;
}
```




```c
/* Read a single character, if it is lower case
 * 'a' then output 'A'; otherwise output the input
 * character. Finally, output the end-of-line
 */
int main() {
   int c;

   if((c = getchar()) == EOF)
      return EXIT_FAILURE;

   if(c == 'a')
      putchar('A');
    else putchar(c);
   putchar('\n');

   return EXIT_SUCCESS;
}
```

"Read Single Character" Idiom

3: Formatted I/O `printf`

`int printf("format", exp)` print formatted

```
printf("%d", 123);

printf("The value of i = %d.\n", i);

printf("Hello world\n");

(void)printf("Hello world\n"); /* document.*/

if(printf("Hello world\n") == EOF) ...
```

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

- `printf()` expects a string, so

```
printf("\n");
```

is correct, but

```
printf('\n');
```

is wrong.

 Copyright: Addison-Wesley Publishing Company, 2000

3: Formatted I/O `scanf`

`int scanf("format", &var)` read formatted

```
int i;
double d;

scanf("%d", &i);
scanf("%lf", &d);

scanf("%d%lf", &i, &d);
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
int main() {
   int i;

   printf("Enter an integer:");
   scanf("%d", &i);

   printf("%d\n", i);

   return EXIT_SUCCESS;
}
```

Idiom? (refers to the `printf`/`scanf` pair)

 Copyright: Addison-Wesley Publishing Company, 2000

3: Formatted I/O. Integer Conversions

To print integers, the following conversions are used:

`d`	signed decimal
`ld`	long decimal
`u`	unsigned decimal
`o`	unsigned octal
`x, X`	unsigned hexadecimal

```
printf("%d%o%x", 17, 18, 19);
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Formatted I/O Float Conversions

To print floating point values, the following conversions are used (the default precision is 6):

`f`	[-] ddd.ddd
`e`	[-] d.ddddde{sign}dd
`E`	[-] d.dddddE{sign}dd
`g`	shorter of f and e
`G`	shorter of f and E

```
printf("%5.3f\n", 123.3456789);
printf("%5.3e\n", 123.3456789);
123.346
1.233e+02
```

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

```
double fv;
long lv;

  printf("%lf", fv);
  scanf("%f", &fv);
  printf("%ld", lv);
  scanf("%d", &lv);

  printf("%d", 3.5);
  printf("%g", 4);
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Formatted I/O
String Conversions

To print characters and strings, the following conversions are used:

- `c` character
- `s` character string

```
   printf("%c", 'a');
   printf("%d", 'a');

   printf("This %s test", "is");
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Formatted I/O
More on `scanf()`

`scanf()` returns the number of items that have been successfully read, and EOF if no items have been read and end-file has been encountered.

For example `scanf("%d%d", &i, &j)` returns the following value:

- 2 if both input operations were successful
- 1 if only the first input operations was successful
- 0 if the input operation failed
- `EOF` if an end-of-file has been encountered.

 Copyright: Addison-Wesley Publishing Company, 2000

```
int main() {
   double i, j;

   printf("Enter two double values:");
   if(scanf("%lf%lf", &i, &j) != 2)
      return EXIT_FAILURE;

   printf("sum = %f\ndifference = %f\n",
               i + j, i - j);

   return EXIT_SUCCESS;
}
```

Idiom? (refers to the boxed `printf`/`if(scanf(...)`/`return EXIT_FAILURE;` lines)

 Copyright: Addison-Wesley Publishing Company, 2000

Idioms

Read Single Integer with prompt

```
printf("Enter integer: ");
if(scanf("%d", &i) != 1 ) …
  /* error, else OK */
```

Read Two Integers

```
if(scanf("%d%d", &i, &j) != 2 ) …
  /* error, else OK */
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: `sizeof`

`sizeof`(type name)

or

`sizeof` expression

returns the size of the data type or object represented by the expression.

```
   sizeof(int)
   sizeof i
```

 Copyright: Addison-Wesley Publishing Company, 2000

3:Type Conversions

Type T is **wider** than type S (and S is **narrower** than T), if

```
sizeof(T) >= sizeof(S)
```

The narrower type is *promoted* to the wider type,
the wider type is *demoted* to the narrower type

an `int` value can be safely promoted to `double`
a `double` value can *not* be safely demoted to `int`

 Copyright: Addison-Wesley Publishing Company, 2000

3: Arithmetic Conversions

If operands of an expression are of different types, then these operands will have their types changed, using *arithmetic conversions*. A lower precision type is *promoted* to a higher precision type according to the following hierarchy:

```
int
unsigned
long
unsigned long
float
double
long double
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Assignment and Type cast Conversions

- Assignment conversions occur when the expression on the right hand side of the assignment has to be converted to the type of the left-hand side.
- The **type cast** expression

```
(typ) exp
```

converts the expression `exp` to the type `typ`.

```
double f, c;
f = 10;      /* assignment conversion */
f = 100.2;
c = (5/9)*(f - 32);
c = ( (double)5/9 ) * (f - 32); /* cast */
```

 Copyright: Addison-Wesley Publishing Company, 2000

3: Type Synonyms: typedef

```
typedef existingType NewType;
```

For example, if you want to use a Boolean type, define

```
typedef int Boolean;

Boolean b = 1;
```

 Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines

Lexical recommendations

- Blank lines are used to separate logically related sections of code
- When specifying a new type using **typedef**, identifier names start with an upper case letter.

 Copyright: Addison-Wesley Publishing Company, 2000

Chapter 4:

Control Structures

4:
Control Statements

Java and C control structures differ in only one respect:
C does not support *labeled* `break` and `continue` statements, which are useful for controlling program flow through *nested* loops:

```
for(i = 0; i < length; i++)
    for(j = 0; j < length1; j++)
       if(f(i, j) == 0)
          goto done;

done:
```



```
/* Program that reads two integer values, and
 * outputs the maximum of these values.
 */
int main() {
   int i, j;

   printf("Enter two integers:");
   if(scanf("%d%d", &i, &j) != 2) {
     fprintf(stderr, "wrong input\n");
     return EXIT_FAILURE;
   }
   printf("Maximum of %d and %d is %d\n",
              i, j, i > j ? i : j);
   return EXIT_SUCCESS;
}
```

"Read Two Integers with prompt" Idiom

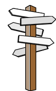
Programming Guidelines

Control Statements

- The body of the **if** statement is indented to the right, and all its instructions are aligned.
- No need for curly braces within a conditional statement, when *only one* statement is present:

```
if(condition) {
     single statement1
} else {
     single statement2
}
```


Programming Guidelines



While statement

```
while(expr) {
     stats
}
        while(expr)
        {
             stats;
        }
             while(expr)
                  {
                  stats;
                  }
```


Programming Guidelines

Control Statements

- A `while(1)` loop is equivalent to:

```
for(;;) {
     body
}
```

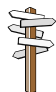
- The following

```
while(expr != 0)
     statement;
```

is equivalent to:

```
while(expr)
   statement;
```


Programming Guidelines



Loops with empty body

If the body of the loop is empty, then the corresponding semicolon is always placed on a separate line, indented to the right:

```
for(i = 1, sum = 0; i <= 10; i++, sum += i)
      ;


 for(i = 1, sum = 0; i <= 10; i++, sum += i);
```

```
/* Example 4.4
 * Read characters until "." or EOF and output
 * the ASCII value of the largest input
 * character.
 */
int main() {
   const char SENTINEL = '.';
   int aux;
   int maxi = 0;
```



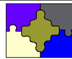
```
printf("Enter characters,. to terminate\n");

while(1) {
  if((aux = getchar())== EOF || aux == SENTINEL)
        break;

  if(aux > maxi)
    maxi = aux;
}
  printf("The largest value: %d\n", maxi);
  return EXIT_SUCCESS;
}
```

Idiom?

Idioms



Read Characters Until Sentinel

```
while(1) {
     if((aux = getchar()) == EOF || aux == SENTINEL)
        break;
      ...
}
```

or:

```
while(1) {
    if((aux = getchar()) == EOF)
       break;
    if(aux == SENTINEL)
       break;
    ...
```



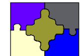
```
/*
 * File: ex4-5.c
 * Read integers until 0 and output the
 * largest integer
 * It also stops on an incorrect integer and
 * end-of-file
 */
int main() {
   const int SENTINEL = 0;
   int i;
   int maxi;
```



```
printf("Enter integers, 0 to stop\n");
if(scanf("%d", &maxi)!= 1 || maxi == SENTINEL){
   printf("No value read\n");
   return EXIT_SUCCESS;
}
while(1) {
  if(scanf("%d", &i) != 1 || i == SENTINEL)
     break;
  if(i > maxi)
     maxi = i;
};
printf("The largest value: %d\n", maxi);
return EXIT_SUCCESS;
}
```

Idiom?

Idioms



Read Integers Until Sentinel

```
while(1) {
     if(scanf("%d", &i) != 1 || i == SENTINEL)
        break;
      …
}
```

Idioms

Read Until Condition

```
while(1) {
  printf("enter integers a and b, a < b:");

  if(scanf("%d%d", &a, &b) == 2)
     return EXIT_FAILURE;

  if(a < b)
     break;
  …
}
```


"Read until Condition"
Idiom

```
/* Read a and b until a < b */
int main() {
 int a, b;
 while(1) {
   printf("enter two integers a and b, a < b:");
    if(scanf("%d%d", &a, &b) != 2) {
       fprintf(stderr, "wrong integer\n");
       return EXIT_FAILURE;
    }
    if(a < b)
       break;
    printf("a must be smaller than b\n");
 }
 … /* process */
```


4: Switch

```
switch(c) {
case ' ' : cblank++;
           break;
case '\t': ctabs++;
           break;
case '*' : cstars++;
           break;
default  : if(c >= 'a' && c <= 'z')
              clower++;
           break;
}
```


Errors

- `i = 8`   cmp.   `i == 8`
- **Watch for off-by-one errors**
- **Avoid the following errors:**

```
e1 & e2

e1 | e2

if(x = 1) …
```


Chapter 5:

Text Files

5: Preview

- I/O operations on *streams*
- No division into input streams and output streams
- Files are sequences of bytes
- Text files: *processing* is line-oriented
- Binary files: different processing.
- End-of-line; one of:
  - a single carriage return symbol
  - a single linefeed symbol
  - a carriage return followed by a linefeed symbol
- End-of-file for *interactive* input: ^Z, ^D

5: File Handles and Opening Files

```
FILE *fileHandle;

fileHandle = fopen(fileName, fileMode);
```

Examples

```
FILE *f;
FILE *g;

  f = fopen("test.dat", "r");
  g = fopen("test.out", "wb");
```


5: Opening Files

```
fileHandle = fopen(fileName, fileMode);
```

`"r"` open for input; (file must exist)
`"w"` open for output; (overwrite or create)
`"a"` open for output; (always append to this file
`"r+"` like `"r"` for I/O
`"w+"` like `"w"` for I/O
`"a+"` like `"a"` for I/O

The above modes may be used to specify a *binary* mode, by using the character `b`

5: Closing files and predefined handles

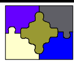
```
fclose(fileHandle);
```

File handles are *resources* that you have to manage:
close files as soon as you do not need them!

You can use three predefined file handles in your programs:

`stdin`	the standard input stream
`stdout`	the standard output stream
`stderr`	the standard error stream

Idioms



Opening a file

```
if((fileHandle = fopen(fname, fmode)) == NULL)
     /* failed */
```

Closing a file

```
if(fclose(fileHandle) == EOF)
     /* failed */
```


Errors

- To declare `FILE` variables, do not use

```
FILE *f, g;
```

- Do not use

```
open()
```

or

```
close()
```


5: Basic File I/O Operations

```
int getchar()          int fgetc(fileHandle)

int putchar(int)       int fputc(int, fileHandle)


int scanf(…)           int fscanf(fileHandle, …)

int printf(…)          int fprintf(fileHandle, …)
```

```
/*
 * Example 5-1
 * Program that reads three real values from the
 * file "t" and displays on the screen the
 * sum of these values
 */
int main() {
   FILE *f;
   double x, y, z;

   if((f = fopen("t", "r")) == NULL) {
      fprintf(stderr, " can't read %s\n", "t");
      return EXIT_FAILURE;
   }
```

"Opening a file"
Idiom


```
   if(fscanf(f, "%lf%lf%lf", &x, &y, &z) != 3) {
      fprintf(stderr, "File read failed\n");
      return EXIT_FAILURE;
   }

   printf("%f\n", x + y + z);

   if(fclose(f) == EOF) {
      fprintf(stderr, "File close failed\n");
      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

Idiom?

"Closing a file"
Idiom

Idioms

Read Single Character from a File

```
if((c = fgetc(fileHandle)) == EOF)
  /* error */
```

Read Single Integer from a File

```
if(fscanf(fileHandle, "%d", &i) != 1)
   /* error */
```


5: Testing for End-of-Line and End-of-File

```
while((c = getchar()) != '\n') /* bug */
     putchar(c);
  putchar(c);

while ((c = getchar()) != '\n')
    if(c == EOF)
       break;
     else putchar(c);

  if(c != EOF)
     putchar(c);
```


Idioms

Read a Line

```
while((c = getchar()) != '\n')
 ...
```

Read a Line from a File

```
while((c = fgetc(fileHandle)) != '\n')
 ...
```


Idioms

Read until end-of-file

```
while((c = getchar()) != EOF)
 ...
```

Read from a file until end-of-file

```
while((c = fgetc(fileHandle)) != EOF)
 ...
```

Clear until end-of-line

```
while(getchar() != '\n')
   ;
```

```
/* look for occurrences of 'a' in a file "t"*/
int main() {
  FILE *fileHandle;
  int i = 0; /* counter */
  int c;
  const int A = 'a';

  if((fileHandle = fopen("t", "r")) == NULL) {
     fprintf(stderr, "can't open %s\n", "t");
     return EXIT_FAILURE;
  }
```

"Opening a file"
Idiom


```
while((c = fgetc(fileHandle)) != EOF)
  if(c == A)
     i++;
```

"Read from a file until end-of-file"
Idiom

```
if(fclose(fileHandle) == EOF) {
    fprintf(stderr, "can't close %s\n", "t");
    return EXIT_FAILURE;
}

printf("There are %d occurrences of %c\n",
         i, A);
return EXIT_SUCCESS;
}
```

"Closing a file"
Idiom


```
/* Simple menu:
 * h   to say "Hello"
 * b   to say "Good buy"
 * q   to quit
 */
int main() {
 int c;

while(1) {
   printf("Enter your command (h/b/q)\n");
   c = getchar();

   while(getchar() != '\n')
      ;
```

"Clear until end of line"
Idiom


```
  switch(c) {
  case 'h':
  case 'H':     printf("Hello\n");
                break;
  case 'b':
  case 'B':     printf("Good buy\n");
                break;
  case 'q':
  case 'Q':     return EXIT_SUCCESS;

  default:      printf("unknown option\n");
  }
 } /* end of while(1) */
}
```



```
/* Modify an existing file to remove all
 * occurrences of ^M
 */
 int main() {
 int c;
   FILE *inOutFile;
   FILE *temp;

  if((inOutFile = fopen("test1", "r")) == NULL)
     return EXIT_FAILURE;

  if((temp = tmpfile()) == NULL)
     return EXIT_FAILURE;
```

"Opening a file"
Idiom


```
/* filter out all ^M */
while((c = fgetc(inOutFile)) != EOF)
   if(c != '\r')
      fputc(c, temp);
```

"Read from a file until end-of-file"
Idiom

```
 if(fclose(inOutFile) == EOF)
    return EXIT_FAILURE;
```

"Closing a file"
Idiom

```
  /* now, rewrite test1 and copy back */
  if((inOutFile = fopen("test1", "w")) == NULL)
     return EXIT_FAILURE;

  rewind(temp);

  while((c = fgetc(temp)) != EOF)
     fputc(c, inOutFile);

  if(fclose(inOutFile) == EOF)
     return EXIT_FAILURE;
}
```

"Opening a file" *Idiom*

"Read until end-of-file" *Idiom*

"Closing a file" *Idiom*

Chapter 6:

The C Preprocessor

6: Preview

- macros (with and without parameters)
- conditional compilation
- file inclusion
- predefined macros
- applications for debugging

macro → **macro replacement**

macro substitution

6: Parameterless Macros

```
#define  macroName  macroValue
```

During preprocessing, each occurrence of **macroName** in the source file will be replaced with the text specified in **macroValue**.

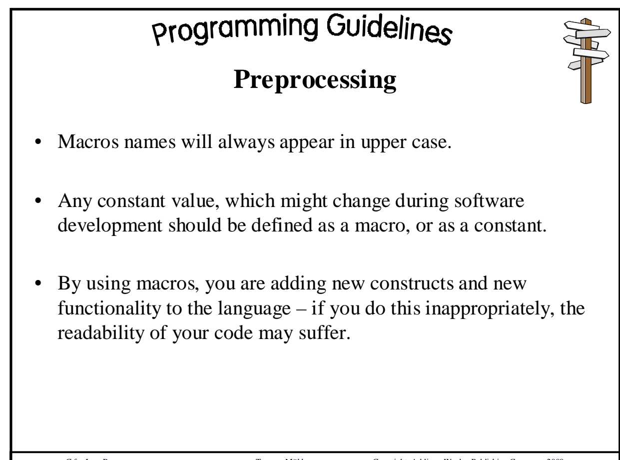
```
#define PI               3.14
#define SCREEN_W         80
#define SCREEN_H         25
#define PROMPT           Enter two \
                              integers
```


6: Parameterless Macros

```
#define PROMPT     printf("Enter real value: ")
#define SKIP       while(getchar() != '\n') \
                     ;

#define A          2 + 4
#define B          A * 3

#define A            (2 + 4)
#define B            (A * 3)
```

6: Predefined Macros

`__LINE__`	current line number of the source file
`__FILE__`	name of the current source file
`__TIME__`	time of translation
`__STDC__`	1 if the compiler conforms to ANSI C

```
printf("working on %s\n", __FILE__);
```


6: Macros with Parameters

```
#define macroName(parameters) macroValue
```

Examples

```
#define RANGE(i)  (1 <= (i) && (i) <= maxUsed)
#define R(x)      scanf("%d",&x);

#define READ(c, fileHandle)    \
               (c = fgetc(fileHandle))
```

Parenthesize aggressively!

Errors

```
#define PI = 3.14

#define PI 3.14;

#define F (x) (2*x)
```

- Enclose the entire macro, as well as each occurrence of a macro argument, in parentheses.
- Avoid side effects in macro arguments.

6: File Inclusion

`#include "filename"` the *current* directory and ...
`#include <filename>` special *system* directories

All relevant definitions may be grouped in a single **header** file

```
screen.h:
#define SCREEN_W    80
#define SCREEN_H    25

#include "screen.h"
int main() {
…
}
```


6: Standard Header Files

`stdio.h` - the basic declarations needed to perform I/O

`ctype.h` - for testing the state of characters

`math.h` - mathematical functions, such as abs() and sin()

6: Conditional Compilation (1)

```
#if constantExpression1
   part1
#elif constantExpression2
   part2
#else
   part3
#endif
```

6: Conditional Compilation (2)

```
#ifdef macroName
   partl
#else
   part2
#endif

#ifndef macroName
   partl
#else
   part2
#endif
```

 Copyright: Addison-Wesley Publishing Company, 2000

6: Debugging

```
#if 0
   part to be excluded
#endif


#define DEB    /* empty, but defined */
#ifdef DEB
   /* some debugging statement, for example */
   printf("value of i = %d", i);
#endif
/* production code */
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
int main() {
  int i, j;

  printf("Enter two integer values: ");
  if(scanf(%d%d, &i, &j) != 2)
     return EXIT_FAILURE;
#ifdef DEB
  printf("entered %d and %d\n", i, j);
#endif
  printf("sum = %d\n", i + j);

  return EXIT_FAILURE;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
  int i, j;
#ifdef DEB
  int res;
#endif
if(
#ifdef DEB
  (res =
#endif
  scanf(%d%d, &i, &j)
#ifdef DEB
  )
#endif
          ) != 2 )
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
#ifdef DEB
 {
  switch(res) {
  case 0: printf("both values were wrong\n");
          break;
  case 1: printf("OK first value %d\n", i);
          break;
  case EOF: printf("EOF\n");
          break;
  case 2: printf("both OK\n");
          break
  }
#endif
 ...
```

 Copyright: Addison-Wesley Publishing Company, 2000

6: Header files

To avoid multiple inclusion:

```
#ifndef SCREEN_H
#define SCREEN_H
...
/* contents of the header */
#endif
```

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

- Avoid side effects in macro arguments:

```
#define SQR(x)  (x*x)
SQR(i++);
```

- In a header file, use

```
#ifndef
```

rather than

```
#ifdef
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

6: Portability

```
#if IBMPC
#include <ibm.h>
#else
#include <generic.h>
/* use machine independent routines */
#endif

#ifdef IBMPC
typedef int MyInteger
#else
typedef long MyInteger
#endif
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Chapter 7:

Functions, Scope, and Introduction to Module-based Programming

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

7: Preview

- a review of functions
- modularization of programs: multiple files & separate compilation
- scope rules
- introduction to module based programming:
  - header files for representing interfaces
  - encapsulation of data in a file
  - kinds of modules
- module maintenance:
  - modifying existing modules
  - extending existing modules

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

7: Functions and Their Documentation

- A C program consists of one or more function definitions, including exactly one that must be called `main`
- The syntax for C functions is the same as the syntax for Java methods
- All functions are *stand-alone*, which means that they are not nested in any other construct, such as a class
- As in Java, parameters are passed *by value*

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

7: Function Declaration and Definition

A **declaration** merely provides a function prototype:
function header (includes the return type and the list of parameters)

```
void hex(unsigned char *p, int max);
```

The declaration does not say anything about the implementation.

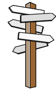
The **definition** of a function includes both the function prototype and the function body, that is its implementation.

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Function documentation

Function declaration or definition (or both) should be preceded by *documentation*:

Function: name
Purpose: a general description of the function
(typically, this is a description of what it is supposed to do)
Inputs: a list of parameters and global variables read in the function
Returns: value to be returned
Modifies: a list of parameters and global variables that are modified - describes any side-effects
Error checking: describes your assumptions about actual parameters - what happens if actual parameters are incorrect
Sample call:



Function documentation

- Documentation may also include a **Bugs** section, which documents cases that the implementation does not handle.
- Make sure comments and code agree
- In general, a function definition should not exceed one page. Code should be broken up; in particular, lines which are too long should be avoided.

7: Function Parameters

There are two types of function parameters:

formal parameters (appear in a declaration or a definition of a function)
actual parameters (appear in a call to the function).

`int f(int x);` here `x` is a formal parameter

`i = f(2*3);` here `2*3` is the actual parameter corresponding to the formal parameter.


```
/* Function:  maxi
 * Purpose:   find the maximum of its integer
 *   arguments
 * Inputs:    two parameters
 * Returns:   the maximum of parameters
 * Modifies: nothing
 * Error checking: none
 * Sample call: i = maxi(k, 3)
*/
int maxi(int, int);

int maxi(int i, int j) {
  return i > j ? i : j;
}
```



```
/* Function:  sqrtRev
 * Purpose:   compute square root of inverse
 * Inputs:    x (parameter)
 * Returns:   square root of 1/x
 * Modifies:  nothing
 * Error checking: none
 * Bugs: Fails if x is 0
 * Sample call: d = sqrtRev(2.4);
*/
double sqrtRev(double);
#include <math.h>  /* gcc -lm … */
double sqrtRev(double x) {
   return sqrt(1/x);
}
```



```
/* Function:  oneOverNseries
 * Purpose:   compute the sum of 1/N series
 * Inputs:    n (parameter)
 * Returns:   the sum of first n elements of
 *        1+ 1/2 + 1/3 + … 1/n
 * Modifies:  nothing
 * Error checking: returns 0 if n negative
 * Sample call: i = oneOverNseries(100);
*/
double oneOverNseries(int);
```

```
double oneOverNseries(int n) {
  double x;
  int i;

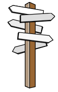
  if(n <= 0) return 0;

  for(x = 1, i = 1; i < n; i++)
     x += 1/((double)i);

  return x;
}

/* Check boundary conditions */
```


Programming Guidelines



Avoid

```
if(n/10 == 0)
   return 1;
else return 1 + digits(n/10);

if(n/10 == 0)
   return (1);
return (1 + digits(n/10));

if(n /= 10)
    return 1;
return 1 + digits(n);
```


7: `void` and Conversions

- Definition:
  `int f()` is equivalent to `int f(void)`
- Call:
  `f();` is equivalent to `(void)f();`

The value of each actual parameter is implicitly converted to the type of the corresponding formal parameter.

The same rules apply to return type conversion.

```
int f(int);
double x = f(1.2);
```


7: `exit` Function

To terminate the execution of an entire program:

```
  exit(int code);

double f(double x) {
 if(x < 0) {
   fprintf(stderr, "negative x in %s\n",
          __FILE__);
   exit(EXIT_FAILURE); /* no return … */
 }
 return sqrt(x);
}
```


Errors

- ```
  double v = f(2.5); /* call before decl. */
  double f() { … }
  ```
- ```
  double f() { return 2.5; }
  double f()
  ```
- ```
  double f(double v) {
    if(v == 0) return;
  }
  ```
- The code parameter of **`exit()`** should be one of the two values: **`EXIT_SUCCESS`** or **`EXIT_FAILURE`**.

7: Scope

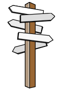
- The **lifetime** of a variable is the period of time during which memory is allocated to the variable
- Since storage is freed in the reverse order of allocation, a *stack* is a convenient data structure to represent it with
  (the **run time stack**)
- C's scope rules use *files* (Java uses classes).
  ```

7: Blocks and Global Variables

- A **block** is like a compound statement, enclosed in braces, and it may contain both definitions and statements.
- **Global variables** are defined outside the body of every function in the file (lifetime of the main program):

```
int flag = 0; /* global */
int f() {
…
}
int out = 1; /* global */
int main() {
...
}
```


Programming Guidelines
Global variables



- Global variables should be used with caution, and always carefully *documented*.
  Changing the value of a global variable as a result of calling a function should be avoided; these **side-effects** make testing, debugging, and in general maintaining the code very difficult.

- The *placement* of the definition of a global variable defines its scope, but also contributes to the readability of your program.
  For short files all global variables are defined at the top;
  for long files they are defined in the logically related place (before definitions of functions that may need these variables).

7: Storage Classes and Lifetime

- Static storage class for ***local*** variables (declared *inside* a block or function) - the lifetime of the entire program:

```
void login() {
   static int counter = 0;
   counter++;
   ..
}
```

- register variables:

```
   register int i;
```


7: Initialization of Variables

- at compile time:

```
const int a = 3 * 44;
```

- at run time:

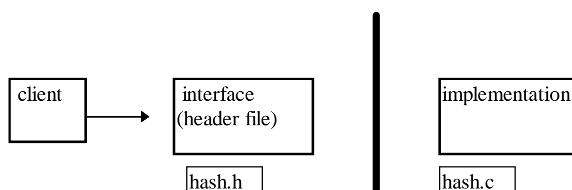
```
double x = sqrt(2.66);
```

- The value of a *local* variable that is declared, but not initialized, is undefined.

- Global variables are initialized to a "zero" value.

7: Modules; Interface and Implementation

Module consists of an interface and an implementation



7: Sharing Functions and Variables: `extern`

- **Separate compilation**: one or more source files may be compiled, creating object codes
- A function may be defined in one file and called in another file, as long as the call is preceded by the function declaration.

File: `a.c`

```
void foo();  /* extern void foo(); */
extern int myErrorNo;
```

File: `b.c`

```
int myErrorNo;
void foo(){ … }
```

Programming Guidelines

Programs and Files

A program typically consists of one or more files:

a) each file should not exceed 500 lines and its listing should begin on a new page.

b) in each source file, the first page should contain the name of the author, date, version number, etc.

c) avoid splitting a function header, a comment or a type/structure definition across a page break.

7: Linkage and the `static` Keyword (1)

- There are three *types of linkage*: internal, external and "no linkage".
- There are various default rules to specify the type of linkage, and two keywords that can be used to change the default rules: **`extern`** and **`static`**.
- The three default rules are:

- entities declared at the outermost level have *external linkage*
- entities declared inside a function have *no linkage*
- **`const`** identifiers and **`struct`**, **`union`** and **`enum`** types have *internal linkage.*

7: Linkage and the `static` Keyword (2)

- The **`static`** keyword applied to *global* entities changes the linkage of entities to internal.
- The **`extern`** keyword changes the linkage of entities to external.
- The linker uses various types of linkage as follows:
  - identifier with *external linkage*: may be shared by various files, and all occurrences of this identifier refer to the same entity
  - identifier with *no linkage*: refers to distinct entities
  - an identifier with *internal linkage*: all occurrences in a single file refer to the same entity. If a second file has an internally-defined identifier with the same name, all of those occurrences will be tied to a second entity defined for that identifier; there is no sharing of internally defined entities between modules.

7: Linkage and the `static` Keyword (3)

- use **`static`** *global* to specify private entities
- in rare cases when you need to *share a global variable*, use **`extern`**
- be careful to avoid conflicting definitions in multiple files, e.g.:

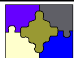
```
File a.c:
int f() { … }

File b.c:
double f() { … }
```


7: Header Files

- The header file corresponds to a Java interface.
- The client gets:
  - the header file
  - the object code of the implementation file.
- The header file is included in the application code, and this code is linked with the implementation file.
- The header file must contain any documentation that is necessary for the client to understand the semantics of all the functions that are declared in it. This documentation should be designed based on a "**need to know**" principle, and should not include any implementation details.

Idioms



Function Names

Use function names that are relevant to the module in which they appear:

`FunctionName_moduleName`

Header and Implementation

The implementation file always includes its corresponding header file.

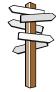
Static Identifiers

Any functions and variable definitions that are private to a file should be qualified as **`static`** (their names start with _, e.g. **`_counter`**)

Programming Guidelines
Interface and Implementation

- Header files should only include function declarations, macros, and definitions of constants.
- Avoid compiler dependent features, if you have to use any such features, use conditional compilation.
- A header file should provide all the documentation necessary to understand the semantics of this file.

Programming Guidelines
Interface and Implementation



- The documentation for the client is placed in the header file.
- The documentation for the implementor is placed in the implementation file.
- The documentation for the client and for the implementor may be different.

7: Types of Modules

- **Layer modules** add a customized interface to an existing module
- **Package modules** define new functions.
- **Type abstraction modules** represent provide a new data type, operations on this type and a set of values
- Modules may be **stateless**
- A module is referred to as **singleton** if the client can use only one instance of this module.

7: Stateless Package Modules
`Lines`

Header file lines.h:

```
#ifndef LINES_H
#define LINES_H
/* A header file for the module with two simple
 *   operations on lines. Here, a line is of
 *   the form y = a*x + b, a!=0, and is
 *   represented by a pair of double values.
 * Operations:
 *   check if two lines are parallel
 *   check if two lines are perpendicular
 */
```



```
/* Function: parallel_Lines
 * Purpose: check if two lines are parallel
 * Inputs: a1, b1, a2, b2 (parameters)
 * Returns: 1 if y=a1*x+b1 is parallel to
 *   y=a2*x+b2, 0 otherwise
 * Modifies: Nothing
 * Error checking: None
 * Sample call:
 *    i = parallel_Lines(4,2,4,7);
 * Since the lines y = 4x + 2 and y = 4x + 7 are
 *   parallel the value of i will be 1.
 */
int parallel_Lines(double a1, double b1, double
  a2, double b2);
```

"Function name" *Idiom*


```
/* Function: perpendicular_Lines
 * Purpose: check if two lines are perpendicular
 * Inputs: a1, b1, a2, b2 (parameters)
 * Returns: 1 if y=a1*x+b1 is perpendicular to
 *   y=a2*x+b2, 0 otherwise
 * Modifies: Nothing
 * Bugs: Returns 0 if a2=0
 * Sample call:
 *    i = perpendicular_Lines (4,2,4,7);
 */
int perpendicular_Lines(double a1, double b1,
  double a2, double b2);
#endif
```

"Function name" *Idiom*

Application file main.c:

"Header and Implementation" *Idiom*

```
#include "ex7-4-lines.h"
int main( ) {
  printf("Lines are %s parallel\n",
     parallel_Lines(1, 2, 3, 4)? "" : "not");
}
```


Implementation file lines.c:

```
#include "lines.h"
int parallel_Lines(double a1, double b1,
    double a2, double b2) {

     return a1 == a2;
}

int perpendicular_Lines(double a1, double b1,
    double a2, double b2) {

     return (a2 == 0) ? 0 : (a1 == -1/a2);
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

7: Layer Module with Error Handling: `IO`

`int getInt(FILE *handle)` reading an integer value

`void putInt(FILE *handle, int i)` printing an integer value

`isError_IO()` to test whether or not an error has occurred

`clearError_IO()` to clear any error indication

`printError_IO()` to print a description of the error on the standard error stream

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Implementation file io.c:

```
#include "io.h"
#define INPUT   1
#define OUTPUT  2
#define OK      0
static int errorFlag_ = OK;  /* private */
int getInt_IO(FILE *f) {
   int i;
   if(fscanf(f, "%d", &i) != 1) {
      errorFlag_ = INPUT;
      return 0; /* use: isError_IO() */
   }
   return i;
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

```
void printError_IO(void) {
switch(errorFlag_) {

case INPUT: fprintf(stderr, "input failed\n");
            return;

case OUTPUT:fprintf(stderr, "output failed\n");
            return;

case OK:   return;
/* defensive style: it can not happen! */
default:   fprintf(stderr, "unknown error\n");
           return;
}
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Idioms

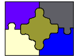
Module Extension

To extend a module M to a module M1:

- define the header file M1.h and the interface M1.c.
- M1.h includes M.h
- M1.c includes M1.h
- the client of M1 includes M1.h and links the application code with M.o and M1.o

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Idioms



Module Modification

To define module M1, which extends an existing module M, follow these steps:

1. Declare the new interface in M1.h (M1.h does not include M.h)
2. M1.c includes both M1.h and M.h
3. The client of M1 includes M1.h and links the application code with M.o and M1.o

7: Modules and Constructors

Layer module `IO2`, built on top of the module IO and designed to encapsulate file handles in the implementation file.

There are three constructors:

```
constructRead_IO2(char *inFilename);

constructWrite_IO2(char *outFilename);


constructReadWrite_IO2(char *inFilename,
                       char *outFilename);
```


7: Modules and Destructors

There are three destructors:

```
int destructRead_IO2();

int destructReadWrite_IO2();

int destructWrite_IO2();
```


IO2: Interface (cont.)

```
int getInt_IO2(void);
void putInt_IO2(int i);

void clearError_IO2(void);
int isError_IO2(void);
void printError_IO2(void);
```


IO2: Application

```
switch(constructReadWrite_IO2("in1", "out1")) {
   case OK_IO2:            break;    /* success */
   case FAIL_READ_IO2:    /* stdin instead */
               (void)constructRead_IO2(NULL);
                          break;
   case FAIL_WRITE_IO2:   /* stdout instead */
               (void)constructWrite_IO2(NULL);
                          break;
   case FAIL_IO2:
           (void)constructReadWrite_IO2(NULL, NULL);
                          break;
   }
```


IO2: Application (cont.)

```
clearError_IO2();
d = getInt_IO2();

if(isError_IO2())
   printError_IO2();
else putInt_IO2(d);

putInt_IO2(getInt_IO2());
```

IO2: Implementation

```
static FILE *fInput_;       /* file handles */
static FILE *fOutput_;
/* macros for testing initialization */
#define NOT_INITIALIZED   0
#define INITIALIZED       1
static int initializedInput_  = NOT_INITIALIZED;
static int initializedOutput_ = NOT_INITIALIZED;
/* macros for testing errors */
#define READ   1
#define WRITE  2
#define OK     0
```


IO2: Implementation (cont.)

```
int constructRead_IO2(char *fname) {
   if(initializedInput_ == INITIALIZED)
      return FAIL_IO2;
   if(fname == NULL) { /* stdin is always OK */
      initializedInput_ = INITIALIZED;
      fInput_ = stdin;
      return OK_IO2;
   }
   if((fInput_ = fopen(fname, "r")) == NULL)
      return FAIL_IO2;
   initializedInput_ = INITIALIZED;
   return OK_IO2;
}
```


IO2: Error handling

```
static int ioErrorFlag_ = OK;

int getInt_IO2(void) {
   if(initializedInput_ == NOT_INITIALIZED) {
      ioErrorFlag_ = READ;
      return 0;
   }
   return getInt_IO(fInput_);
}

int isError_IO2(void) {
   return ioErrorFlag_ != OK || isError_IO();
}
```


7: Modules and Caching Data
`fileOps`

`long words_fileOps();` the number of words in a file
`long chars_fileOps();` the number of characters in a file
`long lines_fileOps();` the number of lines in a file

```
construct_fileOps(filename);
destruct_fileOps();
```


`fileOps`: cache

```
#define CLEAR -1
static long words_ = CLEAR;
static long chars_ = CLEAR;
static long lines_ = CLEAR;
/* debugging version */
long words_fileOps() {
   int flag = 1;
   int c;
   if(initialized_ == NOT_INITIALIZED)
      return EOF;
   if(words_ != CLEAR)   /* use cache */
      return words_;
```



```
words_ = 0; /* reset cache */

rewind(f); /* read & process the input file */
while ((c = getc(f)) != EOF) {
   if(chars_ < 0)
     chars_--; /* cache characters? */
   if(lines_ < 0 && c == '\n')
      lines_--; /* cache lines? */
   if(isspace(c))  /* take care of words */
      flag = 1;
    else if(flag) {
          flag = 0;
          words_++;
```

```
#ifdef DEB
        printf("\nword #: %ld\n", words_);
        fputc(c, stderr);
#endif
      } /* flag is 0 */
#ifdef DEB
      else fputc(c, stderr);
#endif
   }
   rewind(f);
   chars_ = -chars_ - 1;
   lines_ = _lines - 1;
   return words_;  /* update cache */
}
```


7: Variable Number of Parameters

```
/* return a product of double arguments */
double product(int number, ...) {
   va_list list;
   double p;
   int i;
   va_start(list, number);
   for(i = 0, p = 1.0; i < number; i++)
      p *= va_arg(list, double);
   va_end(list);
   return p;
}
```


7: Overloading Functions

```
product(2, 2.0, 3.0) * product(1, 4.0, 5.0)

product(2, 3, 4)
```

There is no predefined type conversion between formal parameters and actual parameters.

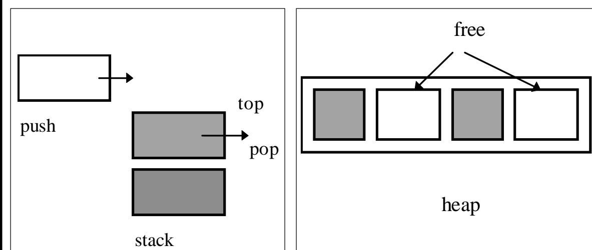
Chapter 8:

Pointers and their Applications

8: Preview

- Java references versus C pointers
- address and dereferencing operations
- dynamic memory allocation and deallocation
- pointer arithmetic
- passing function parameters by reference
- passing functions as parameters of other functions
- overriding a function
- continuation of the discussion of module-based programming
  - enumerations
  - modules for homogenous collections

8: Stack and Heap Based Memory



8: Stack and Heap Based Memory

Stack-based memory: *implicitly* managed by function calls and function returns.

Advantage: you do not have to be concerned with deallocation.

Disadvantage: may lead to programming errors, e.g.

dangling reference problem
a variable referencing a memory block
whose lifetime has expired

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Stack and Heap Based Memory

Heap-based memory: *explicitly* managed by the programmer.

May result in heap fragmentation

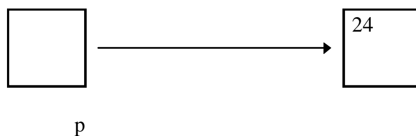
C programmers are *responsible* for memory management
Improper memory management may lead to **memory leakage**

Memory is a resource, and has to be managed like any other resource (e.g. file handles for open files)

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Basic Terminology

- A **pointer** is a variable whose value is a memory address representing the location of the chunk of memory on either the run-time stack or on the heap.
- Pointers have names, values and types.
- Value of p versus value pointed to by p



C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Declaring and Casting Pointers

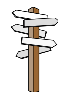
For any C data type T, you can define a variable of type "pointer to T":

`int *p;` pointer to **`int`**, or **`int`** pointer
`char *q;` pointer to **`char`**, or **`char`** pointer
`double **w;` pointer to pointer to **`double`**

Here:
`p` may point to a block of **`sizeof(int)`** bytes
`q` may point to a block of **`sizeof(char)`** bytes
`w` may point to a block of **`sizeof(double*)`** bytes

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines
Pointers



- The placement of the whitespace around the asterisk in a pointer declaration:

```
int* p;
int * p;
int *p;
```

I use the third convention

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Errors
Declaring pointers

To declare two pointers of the same type, use

```
int *p1, *p2;
```

rather than

```
int *p1, p2;
```

You can use

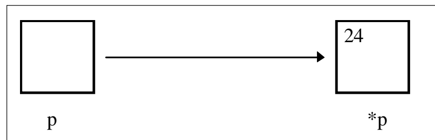
```
typedef int* Pint;
Pint p1, p2;
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Dereferencing Pointers and the Address Operator

```
int *p;
```

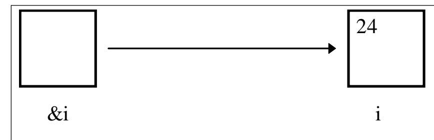
`p` is an `int` pointer,
`*p` is the contents of the memory object `p` points to;
(`*p` is exactly like an `int` variable)



8: Dereferencing Pointers and the Address Operator

```
int i;
```

`i` is an `int` variable
`&i` is like an `int` pointer, pointing to the variable `i`
(but you must not assign to it)

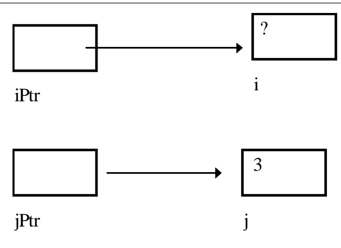


8: Pointer Assignments

```
int i;
int *iPtr = &i;
int j = 3;
int *jPtr = &j;
```

Consider:

```
*jPtr = *iPtr;
i = 4;
*jPtr = i;
iPtr = j;
```



Errors

Pointers and Errors

- Never use uninitialized pointers.
- To increment a dereferenced pointer, use

```
(*p)++
```

rather than

```
*p++    /* correct but means … */
```


8: Using Pointers

```
int i, j;
int *pi = &i;
int *pj = &j;

scanf("%d%d", pi, pj);
printf("%d\n", *pi > *pj ? *pi : *pj);
```


8: Qualified Pointers

A `const` qualifier:

`const int *p;` pointer to a constant integer, the value of `p` may change, but the value of `*p` can not

`int *const p;` constant pointer to integer; the value of `*p` can change, but the value of `p` can not

`const int *const p;` constant pointer to constant integer.

8: Generic Pointers and `NULL`

- Java: a reference to "any" kind of object
  use a variable of type `Object`
- C: a reference to "any" kind of object
  use a variable of type `void*`

```
void *p;
```

defines a *generic,* or *typeless* pointer `p`. Often casted to `(T*)p`

- `NULL` value can be assigned to any pointer, no matter what its type.

8: Generic Pointers and `NULL`

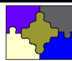
```
void *p = NULL;

int i = 2;
int *ip = &i;

p = ip;
printf("%d", *p);

printf("%d", *((int*)p ) );
```


Idioms



Generic Pointers

Data stored in a memory object can be recovered as a value of a specific data type. For example, for a generic pointer

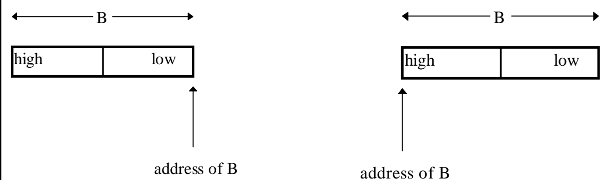
```
void *p
```

which *points* to an object containing a double value, you can retrieve this value using the following syntax:

```
*(double*)p
```


8: Architectures

There are two architectures that use different byte ordering. "little-endian" or *right-to-left byte* ordering architecture (e.g. Intel), "big-endian" or *left-to-right* byte ordering architecture (e.g. SPARC)
Below, block B of memory of type `int`, consists of two bytes:



Portability

Architecture

- Programs that assume a particular architecture (for example, a big endian) are not portable.
- Often, an issue in data compression and image processing.

8: Heap Memory Management

Two primary methods of allocating memory:

```
void *malloc(size_t requestedSize);
void *calloc(size_t requestedCount,
             size_t requestedSize);

T *p;
p = malloc(sizeof(T));     /* or: */
p = calloc(1, sizeof(T));
```

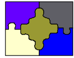
You should always remember to check if a call to a memory allocation function was *successful*.

8: Memory Allocation

```
int *p;
/* A block to store one int */
if((p = malloc(sizeof(int))) == NULL)
   exit(EXIT_FAILURE);
*p = 12;

int *q;
/* a block to store 3 ints */
if((q = malloc(3*sizeof(int))) == NULL)
   exit(EXIT_FAILURE);
*q = 5;
```


Idioms



Memory Allocation for n integers

```
   int* p;
   if((p = malloc(n*sizeof(int))) == NULL)
      error

#define MALLOC(p, type, n)    \
   ((p) = malloc((n)*sizeof(type))) == NULL)
```


Portability

`malloc`

- Always pass **`sizeof(type)`** as a parameter to a **`malloc()`** call, rather than the absolute value.
- For example, use
  `malloc(sizeof(int))`
  instead of
  `malloc(2)`

8: Memory Deallocation

Memory should be deallocated once the task it was allocated for has been completed.

```
int *p;

if(MALLOC(p, int, 1))
   exit(EXIT_FAILURE);
*p = 12;
...
free(p);
/* p not changed; don't use *p */
```


Idioms

Memory Deallocation

Always follow the call to

`free(p)`

with

`p = NULL`

Errors

Memory

- Avoid *mixing* statically and dynamically allocated memory

```
int i;
int *p;
&i = malloc(sizeof(int));
```

- Memory deallocation using **`free()`** should only be used if memory has been previously allocated with **`malloc()`**:

```
p = &i;
free(p);
```

Errors

Memory

- The value of a pointer `p` should never be dereferenced after the call to `free(p)`.
- Do not create "garbage" objects, such as

```
MALLOC(p, int, 1)

MALLOC(p, int, 1)
```


Errors

Memory

- Given two pointers `p` and `q`, the assignment `p = q` does not copy the block of memory pointed to by `q` into a block of memory pointed to by `p`
- Remember that after `p = q`; `p` and `q` share the value; if you call `free(p)` this would also deallocate `q`, now you must not call `free(q)`

8: Pointer Arithmetic

Valid operations on pointers include:

- the *sum* of a pointer and an integer
- the *difference* of a pointer and an integer
- pointer *comparison*
- the *difference* of two pointers.

8: The Sum of a Pointer and an Integer

To access other objects in the block of memory pointed to `p`, use

`p + n`

where `n` is an integer. This expression yields a pointer to the n-th *object* beyond the one that `p` currently points to.
The exact meaning of "object" is determined by the type of `p`.

This operation is not defined for generic pointers; and it's useful to define a macro:

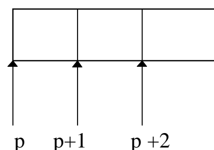
```
#define VOID(targ,size) \
   ((void *)(((char *)(targ)) + (size)))
```


8: The Sum of a Pointer and an Integer

```
#define SIZE 3
double *p;

if(MALLOC(p, double, SIZE))
   exit(EXIT_FAILURE);

*p = 1;
*(p + 1) = 2;
*(p + 2) = 3;
```



8: The Sum of a Pointer and an Integer: i-th object

To access *the i-th object* in a memory block:

`*(p+i)`

or

`p[i]`

`p[0]` and `*p` are equivalent.

8: The Sum of a Pointer and an Integer: i-th object

```
#define SIZE 3
double *p;

if(MALLOC(p, double, SIZE))
     exit(EXIT_FAILURE);

for(i = 0; i < SIZE; i++)
     if(scanf("%lf", p+i) == 0) /* no & */
        exit(EXIT_FAILURE);

*(p+SIZE) = 1.2;
```

 Copyright: Addison-Wesley Publishing Company, 2000

8: The Sum of a Pointer and an Integer: i-th object

`&p[i]` `&*(p+i)` `p+i`
are equivalent.

```
/* read in using [] */
for(i = 0; i < SIZE; i++)
      if(scanf("%lf", &p[i]) == 0)
         exit(EXIT_FAILURE);

/* find product */
 for(i = 0, product = 1; i < SIZE; i++)
      product *= p[i];
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: The Sum of a Pointer and an Integer: Traversals

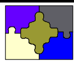
Pointers are often used for *traversals* in loops, for example

```
double *pi;

for(i = 0, pi = p; i < SIZE; i++, pi++)
   product *= *pi;
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Idioms



The i-th Object

`p[i]` is like a regular variable representing the *i-th object* in a block whose beginning is pointed to by `p`.

In particular, `*p` is the same as `p[0]`.

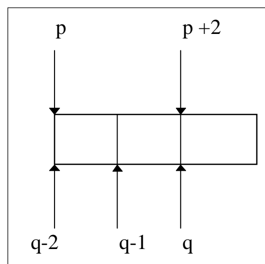
C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Difference of a Pointer and an Integer

Often, we need to access objects *preceding* the object pointed to by a pointer `q`.

`q - n`

yields a pointer to the n-th *object before* the one that `q` currently points to.



C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

8: Example

```
#define SIZE 3
 double *p, *q;
 if(MALLOC(p, double, SIZE))
      exit(EXIT_FAILURE);
 … /* initialize the block */
 /* output in reverse order */
 for(i = 0, q = p+SIZE-1; i < SIZE; i++)
    printf("%f\n", *(q-i));
```

or

```
 for(i = 0, q = p+SIZE-1; i < SIZE; i++)
    printf("%f\n", q[-i]);
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

Errors

Memory

Given a block of memory of `SIZE` objects, pointed to by `p`, you can set `q` to point to the last object in the block using:

```
p+SIZE-1
```

rather than

```
p+SIZE
```

('off by one' error).

8: Pointer Comparison

- Two pointers of the same type, `p` and `q`, may be compared as long as *both of them* point to objects within a *single* memory block
- Pointers may be compared using the <, >, <=, >=, == , !=
- When you are comparing two pointers, you are comparing the values of those pointers *rather than* the contents of memory locations pointed to by these pointers

8: Pointer Comparison

Assuming

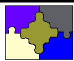
```
double *p, *pi;
```

and `p` pointing to the block of `SIZE` doubles:

```
for(pi = p, product = 1; pi < p+SIZE; pi++)
   product *= *pi;

/* print backwards */
 for(pi = p+SIZE-1; pi >= p; pi--)
      printf("%f\n", *pi);
```


Idioms



Block Traversal

```
for(pi = p; pi < p+SIZE; pi++)
   use pi here
```


8: Pointer Comparison

```
/* Find the largest element in the block p */
double *max;

for(max = p, pi = p+1; pi < p+SIZE; pi++)
   if(*max < *pi)
      max = pi;

printf("The largest value is %f\n", *max);
```

"Block Traversal"
Idiom

8: Pointer Comparison

Copy the contents of a block pointed to by `p` of size `SIZE` to another block pointed to by `q`:

```
double *pi, *qi;

if(MALLOC(q, double, SIZE))
     exit(EXIT_FAILURE);

for(qi = q, pi = p; qi < q+SIZE; qi++, pi++)
   *qi = *pi;
```

8: Pointer Subtraction

Given two pointers, `p` and `q`, such that:

- both pointers are of the same type,
- `p` > `q`
- both point to objects in a single memory block,

the expression

```
p - q
```

yields the number of *objects* between `p` and `q`, *including* the object pointed to by `q`.

The type of the result of pointer difference is `ptrdiff_t`, defined in `stddef.h`.

Copyright: Addison-Wesley Publishing Company, 2000

8: Pointer Subtraction

Find the *first* occurrence of the value `0` in a block of doubles (`EOF` if not found):

```
int position;
double *p, *pi;
/* p initialized */
...
for(pi = p; pi < p+SIZE; pi++)
   if(*pi == 0.0)
      break;

position = (pi == p+SIZE) ? EOF : pi-(p+1);
```


Copyright: Addison-Wesley Publishing Company, 2000

8: Working with Memory Blocks

Operations on memory blocks pointed to by *generic* pointers are declared in the standard header file `string.h`.

Generic pointers can not be dereferenced.

To copy a memory block of size `len` from a source `src` to a destination `dest` (these blocks shouldn't overlap):

```
void *memcpy(void *dest, const void *src,
             size_t len);
```

or, for overlapping blocks:

```
void *memmove(void *dest, const void *src,
              size_t len);
```


Copyright: Addison-Wesley Publishing Company, 2000

8: File I/O for Memory Blocks: Binary Files

- Binary files do not have a line-oriented structure
- They consist of *blocks of objects*, (for example, double objects) which allows them to store information in a concise way.
- Many applications can process binary data efficiently, but to display this data in a human-readable form, a specialized program is needed.
- Unfortunately, binary files are often not portable.

Copyright: Addison-Wesley Publishing Company, 2000

Portability

Binary Files

When you open binary files, you should always use `b` in the second argument for `fopen()`

For example, use:

```
fopen("test.bin", "wb")
```

rather than:

```
fopen("test.bin", "w")
```


Copyright: Addison-Wesley Publishing Company, 2000

8: Binary Files Random Access

Random access: you can directly operate on data stored at *any* position within the file.

Functions that can be used to operate on binary files (assuming that the file has been opened):

```
long ftell(FILE *f)
```

returns the *current position* in a file
upon error, returns `-1L` and sets `errno`

Copyright: Addison-Wesley Publishing Company, 2000

8: Binary Files
Random Access

```
int fseek(FILE *f, long offset, int mode)
```

sets the current position in a file by the specified offset
returns 0 if successful and a non-zero value otherwise.

There are three *predefined* macros:
- `SEEK_SET` the offset is from the *beginning* of the file
- `SEEK_CUR` the offset is from the *current position* in the file
- `SEEK_END` the offset is from the *end* of the file.

`rewind(FILE *f)` same as `fseek(f, 0L, SEEK_SET)`

8: Binary Files
Random Access

Examples, for a file of double values:

```
fseek(f, sizeof(double), SEEK_SET)
fseek(f, sizeof(double), SEEK_CUR)
fseek(f, -sizeof(double), SEEK_CUR)
fseek(f, -sizeof(double), SEEK_END)

   long currentPosition;
   …
   currentPosition = ftell(f);
   …
   fseek(f, currentPosition, SEEK_SET)
```



```
/* compute the size of a file  */
long fileSize(const char *filename) {
   FILE *f;
   long size;
   if((f = fopen(filename, "r")) == NULL)
      return -1L;
   if(fseek(f, 0L, SEEK_END) == 0) {
      size = ftell(f);
      if(fclose(f) == EOF)
         return -1L;
      return size;
   }
   fclose(f);
   return -1L;
}
```


8: Binary Files
Read and Write

```
size_t fread(void *buf, size_t elSize,
             size_t count, FILE *in);
```

reads from the file `in` up to `count` objects, each of size `elSize` and stores them in the block pointed to by `buf`
returns the number of objects that have been actually read; 0 on error (to tell the difference between the end-of- file and an error use `ferror()`)

```
size_t fwrite(void *buf, size_t elSize,
              size_t count, FILE *out);
```


8: Binary Files
Read and Write

To translates a text file containing double values into a binary file:

```
while(fscanf(in, "%lf", &d) == 1)
   if(fwrite(&d, sizeof(double), 1, out) != 1)
```

To translate a binary file of double data into a text file:

```
while(fread(&d, sizeof(double), 1, in) == 1) {
     fprintf(out, "%f\n", d);
     …
}
```


8: Pointers and Functions

Pointer Parameters: Pass by Value

formal parameter = actual parameter

```
void show(const char *p) {
   char *q;
   printf("[ ");
   for (q = p; *q != '\0'; q++)
      printf("%c ", *q);
   printf("]\n");
}
/* Call: show("abc"); */
```

8: Modifying Actual Parameters (Call by Reference)

In order to modify the actual parameter (pass it by reference) use:

formal parameter = &(actual parameter)

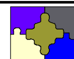
Trace the execution of:

```
void swap(int *x, int *y) {
   int temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
/* call:  int i = 2, j = 3; swap(&i, &j); */
```

 Copyright: Addison-Wesley Publishing Company, 2000

Idioms



Pass by Reference

1. Declare the formal parameter **FP** as a pointer, for example **int *FP**
2. In the body of the procedure, dereference **FP**, that is use ***FP**
3. In the call
   - if the actual parameter **AP**, is a *variable*, use the address of **AP**; for example **f(&AP)**
   - if actual parameter **AP** is a *pointer*, use **AP** without the address operator; for example **f(AP)**

 Copyright: Addison-Wesley Publishing Company, 2000

8: Modifying Actual Parameters (Call by Reference)

Examples of "Boolean" functions with value/reference parameters:

```
/* read up to n characters;
 * return the number of occurrences of c
 */
int readIt(int n, char c, int *occurrences);

/* return the sum and product of two values */
int compute(int n, int m,
            int *sum, int *product);
```

 Copyright: Addison-Wesley Publishing Company, 2000

8: Functions Returning Pointers

```
/* get a block of memory to store int values */
   int* getBlock(size_t size) {
      return malloc(size*sizeof(int));
   }
   ...
   int *p;
   if((p = getBlock(10)) == NULL)
      error
```

The client is responsible for using the "Memory Deallocation" idiom:

```
   free(p);
   p = NULL;
```

 Copyright: Addison-Wesley Publishing Company, 2000

Programming Guidelines

Pointers

Any memory allocation in a function must be documented clearly that the client knows who is responsible for freeing this memory.

 Copyright: Addison-Wesley Publishing Company, 2000

8: Passing Pointers by References

```
int getBlockRef(int **p, unsigned n) {
 if((*p = (int*)malloc(n*sizeof(int))) == NULL)
      return 0;
   return 1;
}
...
   int *q;
   if(getBlockRef( &q , 10) == 1)
      success
```

"Pass by Reference" *Idiom*

 Copyright: Addison-Wesley Publishing Company, 2000

8: Protecting Return Values and Parameter Values: `const`

```
double product(double *block, int size);
/* Does it modify memory pointed to by block?*/

double product(double *block, int size) {
   int i;

   for (i = 1; i < size; i++)
      block[0] *= block[i];

   return block[0];
}
```

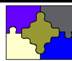

8: Protecting Return Values and Parameter Values: `const`

```
double product(const double *block, int size);

/* return a pointer to a constant */
const int* f(int p) {
   int *i;

   if((i = malloc(sizeof(int)) == NULL)
      return NULL;
   *i = p;
   return i;
}
```


Idioms



Efficient Pass by Reference

```
void f(const T* p);
```


8: The Dangling Reference Problem

Stack-based data are deallocated as soon as the function which defines the data terminates:

```
int *pi;
void f() {
   int i = 2;

   pi = &i;
   printf("the value pointed to by pi is %d\n",
      *pi);
}
```


8: Overriding Functions: Pointers to Functions

A pointer to a function determines the prototype of this function, but it does not specify its implementation:

```
int (*fp)(double); /* a pointer to a function */
int *fp(double); /* a function returning ... */
```

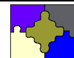
You can assign an existing function to the pointer *as long as* both have identical parameter lists and return types:

```
  int f(double); /* another function */
  fp = f;
```

You can call the function `f()` through the pointer `fp`:

```
   int i = fp(2.5);
```


Idioms



Override Function

To override a function, use a pointer to the function.

One application of this technique is to write ***generic*** sort functions that make the user provide the required function; such as the comparison function.

8: Functions as Parameters

```
void tabulate(double low, double high,
              double step, double (*f)(double)) {
   double x;

   for(x = low; x <= high; x += step)
      printf("%13.5f %20.10f\n", x, f(x));
}
double pol1(double x) {
  return x + 2;
}

tabulate(-1.0, 1.0, 0.01, pol1);
```


8: Reusability of Functions: Generic Pointers

Functions provide reusability:

- can be called over and over to perform the same task
- are required to specify both return *types* and a list of *typed* formal parameters to help catch various errors at compilation time.

Using generic pointers we can support typeless parameters:

- more general
- more dangerous


```
 /* Search a block of double values */
int search( const double *block , size_t size,
            double value) {
   double *p;

   if(block == NULL)
      return 0;

   for(p = block; p < block+size; p++)
      if(*p == value)
         return 1;

   return 0;
}
```

"Efficient Pass by Reference" *Idiom*

"Block Traversal" *Idiom*

8: Generic `search`

C does not support polymorphic programming, but it can be simulated using generic pointers (i.e. `void*`).

A function prototype may specify that a block of memory and the value it is looking for are *not* typed:

```
int searchGen(const void *block,
              size_t size, void *value);
  /* non-typed values
   * Need more parameters:
   */
int searchGen(const void *block,
  size_t size, void *value, size_t elSize
  int (*compare)(const void *, const void *));
```


8: Generic `search` and Modules

Header file contains:

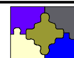
```
int searchGen(const void *block,
    size_t size, void *value, size_t elSize,
    int (*compare)(const void *, const void *));
```

`compare()` is called:

- a *virtual* function; its implementation is not known to `searchGen()` but will be provided when `searchGen()` is called.
- a **callback** function, because it *calls back* the function supplied by the client.

The implementation needs to know how to compare two elements. This type of information is provided by the callback function, which can be *called* by the implementation file, and *defined* by the client.

Idioms



Callback

The *implementation* file may get information from the client

using a **callback** function

passed as a parameter of another function in the *interface*.

8: Implementation of Generic `search` (incorrect)

```
int searchGen(const void *block,
  size_t size, void *value, size_t elSize,
  int (*compare)(const void *, const void *)) {
   void *p;

   if(block == NULL)
      return 0;
   for(p = block; p < block+size; p++)
      if(compare(p, value))
         return 1;
   return 0;
}
```


8: Application of Generic `search`

The client's responsibilities:

```
int comp(const double *x, const double *y) {
   return *x == *y;
}

int comp(const void *x, const void *y) {
   return *(double*)x == *(double*)y;
}
```

Note that this callback is sufficient for search, but not for sort.


```
/* Application of a generic search */
#define SIZE 10
double *b;
double v = 123.6;
int i;
if(MALLOC(b, double, SIZE))
   exit(EXIT_FAILURE);
for(i = 0; i < SIZE; i++) /* initialize */
   if(scanf("%lf", &b[i]) != 1) {
        free(b);
        exit(EXIT_FAILURE);
   }
printf("%f was %s one of the values\n",
 v, searchGen(b, SIZE, &v, sizeof(double), comp)
              == 1 ? "" : "not");
```


8: Correct Implementation of Generic `search`

```
int searchGen(const void *block,
   size_t size, void *value, size_t elSize,
   int (*compare)(const void *, const void *)) {
   void *p;
   if(block == NULL)
      return 0;
   for(p = (void*)block; p<VOID(block, size*elsize);
          p = VOID(p, elsize))
      if(compare(p, value))
         return 1;
   return 0;
}
```


Idioms

Traversing a Block of Generic Objects

```
for(p = (void*)block;
       p < VOID(block, size*elsize);
       p = VOID(p, elsize))
  ...
```

Accessing the i-th Object in a Block of Objects

To access the i-th object, use:

```
p = VOID(block, i*elSize)
```


Errors

```
int i;
void *p = &i;

*p = 2;
*(int*)p = 2;
```

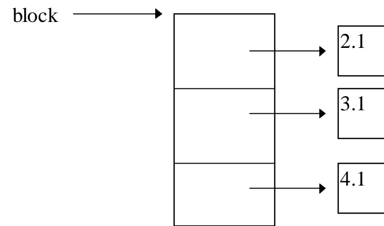
Errors

```
void* f() {
   int *ip;

   if((ip = (int*)malloc(sizeof(int))) == NULL)
     error;
   return ip;
}

   *f() = 2;
   *(int*)f() = 2;
```


8: Pointers to Blocks Containing Pointers



A block containing three pointers to double objects.
In order to access a single object, the code has to apply dereferencing twice

8: Pointers to Blocks Containing Pointers

"Memory Allocation *Idiom*

```
double **block;
#define SIZE 3
if((block=calloc(SIZE, sizeof(double*)))==NULL)
     error;

for(i = 0; i < SIZE; i++)
  if((block[i]=calloc(1, sizeof(double)))==NULL)
     error;

*(*block) = 2.1;
block[0][0] = 2.1;
```


8: Pointers to Blocks Containing Pointers

The complete code to initialize the block: ***Idiom ?***

```
   for(i = 0; i < SIZE; i++)
      block[i][0] = 2.1 + i;
```

To free memory :

```
   for(i = 0; i < SIZE; i++)
      free(block[i]);
   free(block);
   block = NULL;
```

"Memory Deallocation *Idiom*

Idioms

Block of Pointers

For a block `b` of pointers, use

`b[i][j]`

to refer to the `j`-th object in a block pointed to by `b[i]`.


```
#define SIZE 3  /* Triangular block of memory */
if((block=calloc(SIZE, sizeof(double*)))== NULL)
  error
for(i = 0; i < SIZE; i++)
 if((block[i]=calloc(i, sizeof(double)))== NULL)
    error
/* read in values */
for(i = 0; i < SIZE; i++)  /* for each row */
   for(j = 0; j <= i; j++)
     if(scanf("%lf", &block[i][j]) != 1)
        error
/* find the sum */
for(i = 0, sum = 0; i < SIZE; i++)
   for(j = 0; j <= i; j++)
      sum += block[i][j];
```

8: Module-based Programming: Part 2

A data structure is:

- **homogenous** if it consists of objects of the *same* type
- **heterogeneous** if it contains objects of different types

There are two types associated with any data structure:

- the type of each *object*
- the type of the *data structure* itself.

A data structure is:

- **concrete** if the type of each element is known;
- **generic** otherwise

The internal representation of the type of data structure should be hidden from the client; these types are called ***opaque***.

8: Homogenous Collection of Objects

Example. A homogenous generic module, called `Ops`, designed to operate on an *ordered* collection of objects:

- consists of objects of the same type
- this type is *not* known at the time the module is implemented
- the *client* of the module determines *types of objects*
- is a singleton module
- to *compare* two elements, uses a *callback* function

8: Interface of `Ops`

```
int construct_Ops(size_t nElements,
    size_t elSize,
    int (*compare)(const void *, const void *));
int destruct_Ops(void);

int read_Ops(const char *fname, size_t* size);
int search_Ops(void *value, int *result);
int sort_Ops();
int setSort_Ops(
   void (*sort)(void *base, size_t nElements,
   size_t elSize,
   int (*compare)(const void *, const void *)));
```


8: Application of `Ops`

```
#include "ex8-10-ops.h"
/* User-defined callback function */
int doubleComp(const void *x, const void *y) {
   double dx = *(double*)x;
   double dy = *(double*)y;

   if(dx < dy)
      return -1;
   if(dx > dy)
      return 1;
   return 0;
}
```



```
#define SIZE 10
int main() {
 size_t elements;
 double value;
 int result;

 /* start with constructor */
 if(construct_Ops(SIZE, sizeof(double),
    doubleComp) == 0)
      return EXIT_FAILURE;
 if(read_Ops("8-10.dat", &elements) == 0)
     return EXIT_FAILURE;

 printf("%d values read \n", elements);
```



```
 printf("Enter a value to search for: ");
 if(scanf("%lf", &value) != 1)
      return EXIT_FAILURE;

   if(search_Ops(&value, &result) == 0)
      fprintf(stderr, "search failed\n");
   else printf("%f was %s found\n", value,
                 result == 1 ? "" : "not");

   /* do not forget to destruct */
   if(destruct_Ops() == 0)
      return EXIT_FAILURE;
   return EXIT_SUCCESS;
}
```

8: Implementation of `Ops`

The implementation provides:

- by default, a quick sort `qsort()`
- an "intelligent" search function – this function keeps track of the state of the data (sorted or unsorted), and chooses either a binary or linear search implementation accordingly

For a binary search, use a built-in `bsort()`

8: Implementation of `Ops`

```
static int initialized_ = 0; /* initialized? */

static int isSorted_ = 0; /* sorted ? */

static void *base_ = NULL; /*stores collection */

static size_t nElements_; /* size of collection*/

static size_t elSize_; /*size of each element */

static size_t actualSize_; /* current size */
```


8: Implementation of `Ops`

```
static void (*sort_)(void *base,
  size_t nElements, size_t elSize,
  int (*compare)(const void *, const void *));
     /* current sorting routine */

static int (*compare_)(const void *,
                       const void *);
/* callback function */
```



```
int construct_Ops(size_t nElements, size_t elSize,
   int (*compare)(const void *, const void *) ) {

 if(initialized_)    /* singleton */
      return 0;
 if(nElements==0 || elSize==0 || compare== NULL)
      return 0;
 if((base_ = calloc(nElements, elSize)) == NULL)
      return 0;
 nElements_ = nElements;
 elSize_ = elSize;
 compare_ = compare;
 sort_ = qsort;    /* set default sort */
 initialized_ = 1;
 return 1;
}
```



```
int search_Ops(void *value, int *result) {
 if(!initialized_)
    return 0;
 if(isSorted_) /* binary search */
   *result = bsearch(value, base_, actualSize_,
                elSize_,compare_) != NULL;
 else {   /* linear search */
   void *p;
   *result = 0;
 for(p = (void*)block; p < VOID(block, size*elsize);
     p = VOID(p, elsize))
     if(compare_(value, p) == 0)
        *result = 1;
   }
   return 1;
}
```

"Traversing a Block of Objects" *Idiom*


```
int sort_Ops() {
   if(!initialized_)
      return 0;
   sort_(base_, actualSize_, elSize_, compare_);
   isSorted_ = 1;  /* caching */
   return 1;
}

int destruct_Ops(void) {
   if(!initialized_)
      return 0;
   free(base_);
   base_ = NULL;
   initialized_ = 0;
   return 1;
}
```

"Memory Deallocation" *Idiom*

8: Pure Modules: Enumeration

"Pure module" :
a module that consists of only a header file,
(has no accompanying implementation).

```
int hasMoreElements_Enum();   is there another element
void* nextElement_Enum();     get the next element
void reset_Enum();            reset the enumeration.
```

Why do we need `reset_Enum()`?

8: Using Enumerations

`Ops` with enumerations:

```
/* read values */
reset_Enum();
while(hasMoreElements_Enum())
  scanf("%lf", (double*)nextElement_Enum());

/*  show all values */
reset_Enum();
while(hasMoreElements_Enum())
 printf("%f\n", *(double*)nextElement_Enum());
```


8: Implementation of Enumeration

```
static int current_;
int hasMoreElements_Enum() {
   return current_ < actualSize_;
}
void* nextElement_Enum() {
   int i = current_;
   if(current_ >= actualSize_)
      return NULL;
   current_++;
   return VOID(base_, i*elSize_);
}
```

"Accessing the i-th Object" *Idiom*

8: Homogenous Concrete Modules

Homogenous module, with the element type *defined* in the header.
The client decides on the type of element and modifies the type:

```
typedef double Element_Ops1;
```

Less general, but easier to implement and use.
The `compare()` callback function is still required.

We do not want to change anything in the implementation code when changing `Element_Ops1`.

8: Interface of `Ops1`

```
typedef double Element_Ops1;
int construct_Ops1(size_t nElements,
  int (*compare)(const Element_Ops1 *,
                 const Element_Ops1 *));
int destruct_Ops1(void);
int read_Ops1(const char *fname, size_t *size);
int search_Ops1(const Element_Ops1 *value,
                int *result);
int sort_Ops1();
int setSort_Ops1(void (*sort)
        (Element_Ops1 *base, size_t nElements));
int setSize_Ops1(size_t size);
```


8: Application of `Ops1`

```
int doubleComp(const Element_Ops1 *x,
               const Element_Ops1 *y);
if(construct_Ops1(SIZE, doubleComp) == 0)
   return EXIT_FAILURE;
(void)setSize_Ops1(SIZE);
reset_Enum();
while(hasMoreElements_Enum())  {
   if(scanf("%lf", &d) != 1)
      return EXIT_FAILURE;
   if((dp = nextElement_Ops1) == NULL)
      return EXIT_FAILURE;
   *dp = d;
}
```

Chapter 9:

Strings

9: Preview

- Functions which process single characters
- Definitions of Strings and String Constants
- Formatted and Line Oriented String I/0
- C String operations: length, copy, comparisons, search
- Processing Tokens
- String-to-number Conversions
- Standard Error Handling
- Module for String Tokenizing
- Main Function's Arguments

9: Character Processing Functions

To classify:

`int isalnum(int c)`	is `c` an alphanumeric
`int isalpha(int c)`	is `c` an alphabetic letter
`int islower(int c)`	is `c` a lower case letter
`int isupper(int c)`	is `c` an upper case letter
`int isdigit(int c)`	is `c` a digit
`int isxdigit(int c)`	is `c` a hexadecimal digit
`int isodigit(int c)`	is `c` an octal digit

9: Character Processing Functions

To classify (continued):

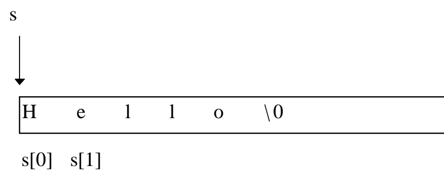
`int isprint(int c)`	is `c` printable (not a control character)
`int isgraph(int c)`	is `c` printable (not a space)
`int ispunct(int c)`	is `c` printable (not space or alphanumeric)
`int isspace(int c)`	is `c` whitespace

To convert:

```
int tolower(int c)
int toupper(int c)
```


9: Strings in C

C stores a string in a block of memory.
The string is terminated by the `\0` character:



9: Definitions of Strings

Strings are defined as pointers to characters:

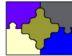
```
char *s;
```

To allocate memory for a string that can hold up to 10 characters:

```
#define SIZE 10
if((s = malloc((SIZE+1)*sizeof(char))) == NULL)
  …
s[0] = '\0';
```

"Memory allocation" *Idiom*

Idioms



Memory Allocation for a string of n characters

```
if((s = calloc(n+1, sizeof(char))) == NULL)
…
```

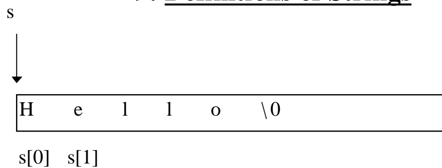
i-th character of a string

To refer to the `i`-th character in the string `s`, use `s[i]`, where `0` <= `i` < length of `s`.

Errors

1. When you allocate memory for a string that can have n characters:
   `calloc(n, sizeof(char))`
2. Do not use
   `calloc(sizeof(string), sizeof(char))`
3. Initialized pointers are not necessarily initialized strings.
   (If an initialized pointer points to a memory block that does not contain the null character, the string is not initialized).

9: Definitions of Strings



The string `s` above has the length 5; `"hello"`
`s+1` (a suffix of `s`) has length 4; `"ello"`
`s+2` (a suffix of `s`) has length 3; `"llo"`
`s+5` (a suffix of `s`) has a length 0; (it is a *null* string) `""`
However, `s+6` is not well defined.

Idioms

String Suffix

If `s` points to a string, then `s+n` points to the suffix of this string starting at the `n`-th position
(here, `n` has to be less than the length of `s`).

9: String Constants

```
char *name = "Kasia";
```

The block of memory for a string constant may be stored in "read-only" memory, and its contents should not be modified; therefore, do not reset any of the characters in the constant string:

```
name[0] = 'B';
```


Errors

Double and single quotes that enclose a single character signify different things:

`"W"` denotes a pointer to a memory block containing two characters:
`W \0`

`'W'` denotes the ordinal value of the character `W`.

9: Strings Parameters

C strings can be used as parameters as *any other pointers*

```
void modify(char *s) {
   s[0] = toupper(s[0]);
}
char *p;   /* modify(p); */
if((p = calloc(10, sizeof(char))) == NULL) error
p[0] = 'h'; p[1] = 'o'; /* p[2] == '\0' */
modify(p);
modify(p+1);
char *q = "hello";
modify(q);
```

"Memory allocation" *Idiom*

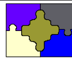
"String suffix" *Idiom*

9: Strings Parameters

```
/* Same as strlen() */
int length(const char *s) {
   char *p;

   for(p = s; *p; p++) /* *p != '\0'   */
      ;
   return p - s;
}
```


Idioms



Traversing a String

```
for(p = s; *p; p++)
   use *p
```



```
char *strdup(const char *s) { /* return copy of s */
  char *kopy;      /* copy of s */
  char *ps;        /* used for copying */
  char *pkopy;     /* for copying */

if((kopy =calloc((length(s)+1),sizeof(char)))==NULL)
      return NULL;

  /* memory allocated, now copy */
  for(ps = s, pkopy = kopy; *ps; ps++, pkopy++)
     *pkopy = *ps;
  *pkopy = *ps;

  return kopy;
}
```

"Memory allocation" *Idiom*

"String Traversal" *Idiom*


```
char *modify(const char *s) {
/* return a copy of s modified */
   char *news;

   if((news = strdup(s)) == NULL)
      return NULL;

   news[0] = toupper(news[0]);

   return news;
}
char *q = modify("c for java");
char *s = modify("c for java" + 6);
```

"i-th character" *Idiom*

"String suffix" *Idiom*

(the last one returns "Java")

9: Strings Parameters and Return Values

```
void modify1(const char *s, char **news) {
/* return through parameter a copy of s modified*/
   if(s == NULL)
      return;

   *news = strdup(s);
   (*news)[0] = toupper((*news)[0]);
}
char *p;
modify1("hello", &p);
```

9: Formatted String I/0

The formal control string `%s` is used for string I/O.

Leading whitespace characters are skipped in a search for the first non-whitespace character, and input stops when a *word* is read (a word is a sequence of characters not containing any whitespace).

Therefore, `scanf()` can read at most one word.

Errors

To input a string:

- use

  ```
  scanf("%s", s)
  ```

- rather then

  ```
  scanf("%s", &s)
  ```

- make sure that **s** is initialized; i.e. there is some memory allocated for **s** (for example, using `calloc()`)
- make sure that there is *enough* memory allocated for **s**, and consider using the *field width* to avoid overflow.

Idioms

Read a Single Word
(at most 10 characters):

```
if(scanf("%10s", s) != 1)
    error
```



```
int lower(char *s) { /* return number of l.c. letters */
   int i;
   char *q;
   for(i = 0, q = s, *q, q++)
      if(islower(*q))
        i++;
   return i;
}
int main() {
  const int M = 10;
  char *p;
  if((p = calloc(M + 1, sizeof(char)) == NULL)
     return EXIT_FAILURE;
  if(scanf("%10s", p) == EOF)
     return EXIT_FAILURE;
  printf("%d lower case letters in %s\n", lower(p), p);
  return EXIT_SUCCESS;
}
```


9: Formatted String I/0

There are two formatted string I/O operations:

```
int sscanf(s, "format", arguments)
int sprintf(s, "format", arguments)
```

```
#define N sizeof(int)+sizeof(double)+6
int i; double d; char *s;

if((s = calloc(N+1,sizeof(char))) == NULL)
    return EXIT_FAILURE;
sprintf(s, "%s %d %f", "test", 1, 1.5);
if(sscanf(s+4, "%d%f", &i, &d) != 2) ...
```


9: Line Oriented String I/0

```
char* fgets(char *buf, int n, FILE *in);
```

reads a line from the file `in`, and stores it in the block pointed to by `buf`. Stops reading when:

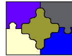
- `n-1` characters have been read
- end-of-line has been encountered; (`\n` is stored at the end of `buf`)
- end-of-file has been encountered.

In any case, `buf` is always properly terminated (`\0` is stored).
The function returns `buf` if successful and `NULL` if no characters have been read or there has been a reading error.

Often it is useful to rewrite the end-of-line character:

```
buf[strlen(buf)-1] = '\0';
```

Idioms



Read a Line (at most n-1 characters) from a File

```
if(fgets(buffer, n, f) == NULL)
    error
```



```
/* find the length of the longest line; at most max */
long longest(const char *fname, const int max) {
   char *line;
   FILE *f;
   long i = 0;
   if((f = fopen(fname, "r")) == NULL)
       return -1;
   if((line = calloc(max + 1, sizeof(char))) == NULL) {
      fclose(f); return -1;
   }
   while(fgets(line, max, f) != NULL)
      if(strlen(line) > i)
         i = strlen(line);
   free(line);
   if(fclose(f) == EOF)
      return -1;
   return i - 1;
}
```


9: Line Oriented String I/0

`int fputs(const char *s, FILE *out);`

writes the string `s`, excluding the null character, to the file `out`; returns `EOF` on error, and a nonnegative value otherwise.

`char* gets(char *buf);`

like `fgets()` but

if end-of-line has been encountered, it is *not* stored in `buf`

`int puts(const char *buf);`

like `fputs()` but

it writes to `stdout` and always appends `\n` to the string `buf`.

9: C String Operations: `strlen` and `strcpy`

To compute the length of a string, use:

`size_t strlen(const char *string);`

To copy `src` to `dest` and return `dest`:

`char *strcpy(char *dest, const char *src);`

To copy `n` characters of `src` to `dest` and return `dest`:

```
char *strncpy(char *dest, const char *src,
              size_t n);
```

If the length of `str` is less than `n`, then trailing characters are set to `\0` (`dest` may not be terminated by the null character).

9: C String Operations: `strcat`

Append (or, "catenate") `src` to `dest` and return `dest`:

`char *strcat(char *dest, const char *src);`

Append `n` characters of `src` to `dest` and return `dest`. If the length of `src` is less than `n`, then trailing characters are set to `\0` (always append the null character):

```
char *strncat(char *dest, const char *src,
              size_t n);
```



```
#define SIZE 5
char *dest;

if((dest =calloc(sizeof(char)*(SIZE+1)))== NULL)
   error
strcpy(dest, "Acadia");
strncpy(dest, "Acadia", SIZE);
strncpy(dest, "Acadia", strlen("Acadia"));
strcat(dest, "Hi");
dest[0] = '\0';
strcat(dest, "");
strcat(dest, "Hi");
strcat(dest, " how");
strncat(dest, " how", SIZE-strlen("Hi"));
```

```
char *strdup(const char *s) {
/* return a copy of s */
   char *kopy;     /* copy of s */

   if((kopy = calloc(strlen(s) + 1,
                sizeof(char))) == NULL)
      return NULL;
   strcpy(kopy, s);

   return kopy;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

- `strcpy(dest, src)` and `strcat(dest, src)` assume that there is enough memory allocated for the `dest` to perform the required operation
- `strncpy(dest, src)` does have to append the zero character
- `if(strlen(x) - strlen(y) >= 0)...`

 Copyright: Addison-Wesley Publishing Company, 2000

9: C String Operations: Comparisons

To lexicographically compare `s1` and `s2`:

```
int strcmp(const char *s1, const char *s2);
```

returns a negative value if `s1` is less than `s2`, 0 if the two are equal, a positive value of `s1` is greater than `s2`.

To lexicographically compare `n` characters `s1` and `s2`:

```
int strncmp(const char *s1, const char *s2,
            size_t n);
```

 Copyright: Addison-Wesley Publishing Company, 2000

Errors

- To check if `str1` is less than `str2`:

```
if(strcmp(str1, str2)) …
if(strcmp(str1, str2) == -1)
if(str1 < str2) ...
if(strcmp(str1, str2) < 0)...
```

- To copy a string `str2` to another string `str1`

```
str1 = str2
strcpy(str1, str2);
```

 Copyright: Addison-Wesley Publishing Company, 2000

9: C String Operations: Search

To search `str` for the *first* occurrence of `c` and return a pointer to this occurrence; (`NULL` if not found):

```
char *strchr(const char *str, int c);
```

To look for the *last* occurrence of `c`:

```
char *strrchr(const char *str, int c);
```

To search for a substring:

```
char *strstr(const char *str, const char *sstr);
```

Returns a pointer to the first occurrence of a substring `substr` in the string `str`, `NULL` if not found

 Copyright: Addison-Wesley Publishing Company, 2000

9: C String Operations: search

To search `str` for the *first* occurrence of any character that does *not* appear in *set*:

```
size_t strcspn(const char *str, const char *set);
```

return the length of the longest prefix of `str` that has been skipped (or *spanned*): `strcspn("Java after", "va")` returns 1.

To search `str` for the *first* occurrence of any character that does appear in *set*:

```
size_t strspn(const char *str, const char *set);
```

return the length of the longest prefix of `str` that has been skipped: `strspn("Java after", "Ja")` returns 2.

To return a *pointer* to the first character

```
char *strpbrk(const char *str, const char *set);
```

 Copyright: Addison-Wesley Publishing Company, 2000

```c
/* strip from s leading and traling characters from
 * set. For example:
 * char *p = strip(" ,hi, how are you,", " ,");
 */
char *strip(const char *s, const char *set) {
 int start = strspn(s, set); /* leading characters */
 int end;                    /* trailing characters */
 char *kopy;
 int length = strlen(s);

 if(length != start) { /* there are chars not in s */
  for(end = length; end > 1; end--) /* trailing */
    if(strchr(set, s[end]) == NULL)
      break;
  length = end - start + 1; /* left after strip */
  ...
```



```c
/*char *strip() continued */
  if((kopy = calloc(length + 1, sizeof(char)))==NULL)
     return NULL;
  memcpy(kopy, s + start, length);
  kopy[length] = '\0';
 } /* length != start */

 else { /* here, no characters in s */
  if((kopy = calloc(length + 1, sizeof(char)))==NULL)
     return NULL;
  strcpy(kopy, s);
 }

 return kopy;
}
```


9: Processing Tokens

```c
char *strtok(char *str, const char *sep);
```

separates `str` into tokens, using characters from `sep` as separators.

- The first parameter `str` may be `NULL` (but not in the first call).
- The *first* call takes the non-null first parameter, and returns a pointer to the first token (skipping over all separators)
- All *subsequent* calls take `NULL` as the first parameter and return a pointer to the next token.
- If the first call does not find any characters in `sep`, the function returns `NULL`.
- Modifies the string being tokenized (to preserve a string, you have to make a copy of it before you call `strtok()`).

9: String-to-number Conversions

```c
double strtod(const char *s, char **p);
long strtol(const char *s, char **p, int base);
unsigned long strtoul(const char *s, char **p,
                      int base);
```

Convert a string `s` *to* a number. If the conversion failed:

`*p` is set to the value of the original string `s`,

the global error variable `errno` is set to `ERANGE`.

Otherwise, `p` is set to point to the first character in the string `s` immediately following the converted part of this string.

A default `base`, signified by 0, is decimal, hexadecimal or octal, and it is derived from the string. (It also may be any number from 2 to 36).

9: Module for String Tokenizing

Files often store data records using a delimited format; e.g.

```
name|salary|id
```

Here, the first field is a string, the second is a double, and the third is a long integer. For example:

```
Mary Smith|2000|185594
John Kowalski|1000|2449488
```

We need to tokenize these strings.

9: Interface of module `token`

```c
int construct_Token(const char *str,
                    const char *delimiters);
int destruct_Token(void);

int hasMore_Token();
char *next_Token();
int count_Token();

int reset_Token();
```

9: Application of module `token`

```
char *nameS;
char *salaryS;
char *idS;
/* reads lines from a file */
while(fgets(line, SIZE, in) != NULL) {
  line[strlen(line)-1]= '\0';
  construct_Token(line, delim);
  if(hasMore_Token())
     nameS = next_Token();
  else
     error
  ...
```



```
 if(hasMore_Token())
  salaryS = next_Token();
 else error
 if(hasMore_Token())
     idS = next_Token();
 else error
 salary = strtod(salaryS, &err);
 if(err == salaryS)
  error
 id = strtol(idS, &err, 10);
 if(err == idS) error
 printf("Name: %s, salary %f, id: %ld\n", nameS,
        salary, id);
 destruct_Token();
```


9: Implementation of module `token`

The constructor tokenizes the entire string and stores it in a block of pointers to tokens.

The module maintains several private variables:

```
static char **block_;   /* pointers to tokens */
static int tokenNumber_; /* number of tokens */
static int current_;  /* current token number */
static int initialized_ = 0;
```



```
int construct_Token(const char *str,
                    const char *delimiters) {
  char *token;
  char *copyStr;
  int i;

  if(initialized_)
     return 0;
  if((copyStr = strdup_(str)) == NULL)
     return 0;
  /* traverse to set the value of tokenNumber_ */
  for(tokenNumber_ = 0,
             token = strtok(copyStr, delimiters);
        token != NULL;
        token = strtok(NULL, delimiters))
     tokenNumber_++;
```

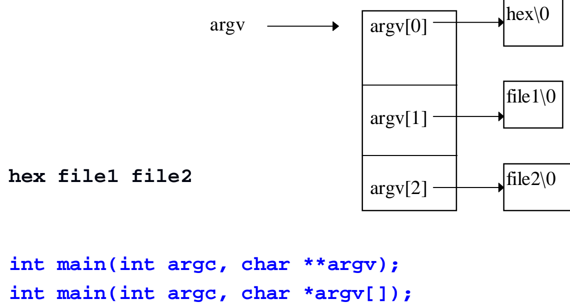


```
 if((block_ = calloc(sizeof(char*),tokenNumber_))
   == NULL){
     free(copyStr);
     return 0;
  }
  strcpy(copyStr, str);
/* traverse the string and store pointers to tokens*/
  for(i = 0, token = strtok(copyStr, delimiters);
          token != NULL;
          token = strtok(NULL, delimiters), i++)
     block_[i] = strdup(token);
  initialized_ = 1;
  current_ = 0;
  free(copyStr);
  return 1;
}
```



```
int destruct_Token(void) {
   int i;
   if(!initialized_)
      return 0;
   for(i = 0; i < tokenNumber_; i++)
      free(block_[i]);
   initialized_ = 0;
   free(block_);
   return 1;
}
char *next_Token() {
   if(!initialized_ || current_ == tokenNumber_)
      return 0;

   return block_[current_++];
}
```

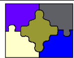
9: Main Function's Arguments




```
hex file1 file2
```

```
int main(int argc, char **argv);
int main(int argc, char *argv[]);
```


Idioms



Command Line

```
int main(int argc, char **argv) {
…
   switch(argc) {
   case …

   default: fprintf(stderr, "usage: %s … \n",
                       argv[0]);
            return EXIT_FAILURE;
}
```

This idiom only checks the number of required arguments.

9: Main Function's Arguments

To pass numerical values on the command line; for example:
in a program, which displays *up to* the first n lines from a file:

```
show -n fname
```

This program can be invoked without the first argument (`-n`),
to display *up to* the first 10 lines.
Assuming we have:

```
int display(const char *fname, int n,
            int Max);
```



```
#define DEFAULT 10
#define MAX     80
int main(int argc, char **argv) {
  int lines = DEFAULT;
  switch(argc) {
  case 3: /* retrieve the number of lines argument */
   if(argv[1][0] != '-' ||
       sscanf(argv[1] + 1, "%d", &lines)!=1 || lines <= 0)
     return EXIT_FAILURE;
   argv++;          /* no break: retrieve filename */
  case 2:  if(display(argv[1], lines, MAX) == 0)
                return EXIT_FAILURE;
           break;
  default:
      return EXIT_FAILURE;
 }
 return EXIT_SUCCESS;
}
```


Errors

Redirection is not a part of the command line of a program.

```
program one two < f1 > f2
```

has two command line arguments, not six.

Chapter 10:

Arrays

10: Single-Dimensional Arrays

C arrays:

- have a lower bound equal to zero
- are *static* - their size must be known at compile time.

To define an array:

```
type arrayName[size];
```

For example

```
int id[1000];
char *names[2*50+1];
#define SIZE 10
double scores[SIZE+1];
```


10: Single-Dimensional Arrays

Constants can not be used to define arrays

```
const int S = 10;
 int id3[S];

 int foo() {
      int id4[S]; /* Ok with gcc */
      …
 }
```


10: Single-Dimensional Arrays and Pointers

A single dimensional array is a *typed constant* pointer initialized to point to a block of memory that can hold a number of objects.

```
int id[1000];
int *pid;
```

`id` is an `int` pointer that points to a block of memory that can hold 1000 integer objects
`pid` is a pointer to `int`.

```
id = pid;
pid = id;
```


Errors

- ```
  int arrayName[SIZE];
  arrayName[SIZE] = 2;
  ```
- ```
  int n = 3;
  double s[n];
  ```
- Arrays are not l-values
- Side-effects make the result of assignments involving index expressions *implementation dependent*
  ```
  a[i] = i++;
  ```

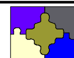

10: Comparing Arrays

```
#define SIZE 10
int x[SIZE];
int y[SIZE];
… initialization of x and y …
if(x == y) ...

int *px, *py;
for(px = x, py = y; px < x + SIZE; px++, py++)
   if(*px != *py)
        different
```

Block Traversal Idiom

Can be simpler...

Idioms



Comparing Arrays

```
for(i = 0; i < SIZE; i++)
   if(x[i] != y[i])
       different
```

Copying Arrays

```
for(i = 0; i < SIZE; i++)
  x[i] = y[i];
```

10: Arrays: sizeof

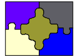
```
int id [1000];
int *pointerId;
```

`sizeof(id)` is `1000*sizeof(int)`
`sizeof(pointerId)` is the number of *bytes* used to store a pointer to an `int`.

To set `pointerId` to the last element of the array `id`:

```
pointerId = id + sizeof(id)/sizeof(id[0]) - 1;
```


Idioms



Array Type

```
typedef ElemType ArrayType[size];

typedef int ArrayType[20];
ArrayType x;
```


10: Arrays as Parameters

When arrays are used as function parameters, they are actually treated as pointers. The following two declarations are equivalent:

```
int maxiA(double arr[], int size);

int maxiP(double *arr, int size);
```

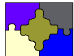
The second parameter is *necessary* to specify the size of the array.


```
int readArray(int x[], int size) {
   int i;

   for(i = 0; i < size; i++)
      if(scanf(%d", &x[i]) != 1)
         return 0;
   return 1;
}

void printArray(int x[], int size) {
   int i;

   for(i = 0; i < size; i++)
      printf(%d", x[i]);
}
```



```
/* Applications of readArray and printArray */
#define SIZE 20
double d[SIZE];

if(readArray(d, SIZE) == 0)
  error;
printArray(d, SIZE);

printArray(d, SIZE - 10);    /* prefix */

printArray(d + 2, SIZE - 2);  /* suffix */

printArray(d + 2, 5);          /* segment */
```


Idioms



Prefix and Suffix of an Array

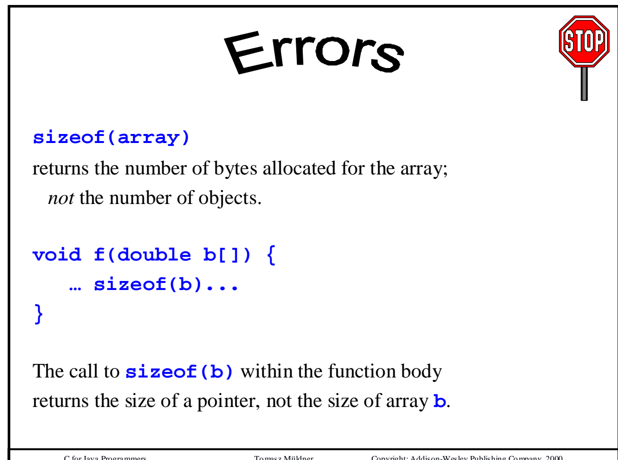
For a function `f(T* arr, int n,…)` operating on an array `arr` of size `n`, call

```
f(arr+start, segSize)
```

to operate on the segment of array `arr` of size `segSize`, starting from position `start`
(here, `segSize+start` must be less than or equal to `n`)

```
/* return the maximum value in the array through
 * function, minimum value through parameter
 */
double maxMin(double arr[], int size, double *min) {
  double max;
  double *p;
  for(max = *min = arr[0], p = arr + 1;
         p < arr + size; p++) {
     if(max < *p)
       max = *p;
     if(*min > *p)
       *min = *p;
   }
   return max;
}
double maxi = maxMin(d, SIZE, &min);
```

Block Traversal Idiom



C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

10: Array Initialization and Storage

- Local non-static arrays are allocated memory from the stack
- Global arrays and static local arrays are allocated memory in a special data segment, called BSS (Below Stack Segment), and their lifetime is the same as that of the main function.

Arrays can be *initialized*: `{ v1, v2, …, vn }`

```
int x[] = {1, 2, 3};
int x[3] = {1, 2, 3};
int x[3] = {1, 2};    /* x[2] is 0 */
int x[3] = {1, 2, 3, 4};
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

10: Arrays and Dangling Reference

```
char *setName(int i) {
   char name1[] = "Mary";
   char name2[] = "John";

   if(i == 0)
      return name1;

   return name2;
}

char *p = setName(1);
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

10: Multi-Dimensional Arrays

```
int x[2][3];
```

1	2	3	← row 0	x[0]
4	5	6	← row 1	x[1]

x

```
for(i = 0; i < 2; i++) {
  for(j = 0; j < 3; j++)
     printf("x[%d][%d] = %d\t", i, j, x[i][j]);
  putchar('\n');
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

10: Dynamic Array Module (with Preconditions)

A singleton module `Arr`:

- supports operations on a single dimensional array
- the element type is known to the implementation (`char` type)
- supports a dynamic array; i.e. its size is defined at run-time through a call to the constructor of the module
- a function `set(v, i)` will expand the array if the value of `i` is greater than the current array size.
- additional error checking ensures that the index expressions used on the array are within the current array bounds.

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

10: Testing Preconditions

A **precondition** is a *necessary* condition that must hold in order for an operation to be performed; e.g. for an array `x[size]` `0 <= i < size` is a precondition to using `x[i]`

The standard library `assert.h` provides `assert(int e)`:

```
assert(0 <= i && i < size)
```

The meaning of this macro depends on another macro `NDEBUG`. *By default* `assert()` is "enabled", and you have to explicitly undefine `NDEBUG` to disable it (this can be done on the compiler's command line).

Copyright: Addison-Wesley Publishing Company, 2000

10: Testing Preconditions

Calling `assert(i)`:

- `NDEBUG` is *not* defined **and** `i` evaluates to 0:
  the error message is displayed and the execution of the program is aborted calling `abort()`. The contents of the error message includes the text of the actual parameter, and two pre-defined macros: `__FILE__` and `__LINE__`
- `NDEBUG` is *not* defined **and** `i` evaluates to a value different from 0, or `NDEBUG` is defined:
  the result of calling `assert()` is void.

Copyright: Addison-Wesley Publishing Company, 2000

10: Interface of `Arr`

```
typedef char Element_Arr;
int construct_Arr(int initSize);
Element_Arr get_Arr(int i);
void set_Arr(Element_Arr value, int i);
int length_Arr();
void destruct_Arr(void);
```

Pre-conditions:

`0 <= i < length` for `get(i)` (`length` is the size of the array)
`0 <= i` for `set(v,i)`

Copyright: Addison-Wesley Publishing Company, 2000

10: Application of `Arr`

Read a line of an *unlimited* length.
The client of the module initializes the array to hold 80 elements:

```
construct_Arr(80);
```

Then, to read, and print this line:

```
for(i = 0; (c = fgetc(f)) != '\n'; i++)
    set_Arr(c, i);
printf("The line is: ");
for(i = 0; i < length_Arr(); i++)
    putchar(get_Arr(i));
```

Finally, the array is destructed:

```
destruct_Arr();
```


Copyright: Addison-Wesley Publishing Company, 2000

10: Implementation of `Arr`

```
static Element_Arr *block_;      /* to store data */
static int size_;               /* size of the block  */
static int init_ = 0;        /* initialization flag */
static const int increment_ = 10;   /* for expand */

int construct_Arr(int initSize) {
  if(init_) return 0;
  if((block_ =calloc(initSize, sizeof(Element_Arr)))
            == NULL)  return 0;
  size_ = initSize;
  init_ = 1;
  return 1;
}
```


Copyright: Addison-Wesley Publishing Company, 2000

```
/* private function to expand */
static int expand_(int size) {
 Element_Arr *new;
 int i;

 if((new = calloc(size, sizeof(Element_Arr)))
       == NULL)
    return 0;
 for(i = 0; i < size_; i++)
    new[i] = block_[i];
 size_ = size;
 free(block_);
 block_ = new;

 return 1;
}
```


Copyright: Addison-Wesley Publishing Company, 2000

```
/* set the i-th element, expand if needed */
void set_Arr(Element_Arr value, int i) {
  int res;
  /* precondition */
  assert(i >= 0 && init_);
  if(i < 0 || !init_)
     return;
  if(i >= size_) { /* expand */
     res = expand_(i + increment_);
     assert(res);
     if(res == 0)
        return;
  }
  block_[i] = value;
}
```


Chapter 11:

Structures and their Applications

11: Preview

- Comparison of structures and Java classes
- How to declare structures, combine structures and pointers as well as arrays, etc.
- Continuation of the discussion of module-based programming (structures used to design modules for which the user can create *multiple* instances)

11: Structures and Classes

- a structure has only *data* members (no functions)
- all its members are *public*.

Consider a Java class `Fractions` with a member method `Add(Fraction)`.

A call: `x.Add(y)` involves two objects:

- "`this`" object (here `x`)
- the object passed as a parameter (here `y`).

In C, functions that emulate methods need an extra parameter to represent "`this`" object.

11: Declaring Structures

Structures are user defined data types, which represent *heterogeneous* collections of data.

```
struct info {
     char firstName[20];
     char lastName[20];
     int age;
};
struct info i1, i2;
info j;
```


11: Declaring Structures

```
struct info {
     char firstName[20];
     char lastName[20];
     int age;
} i1, i2;

typedef struct info { /* can be omitted */
     char firstName[20];
     char lastName[20];
     int age;
} InfoT;
```

11: Using Structures

```
typedef struct InfoT {
     char firstName[20];
     char lastName[20];
     int age;
} InfoT;

InfoT p1;
```

In order to *access* members of a structure:

```
p1.age = 18;
printf("%s\n", p1.firstName);
```


Errors

```
struct example { ... };
example e;

struct example e;

struct example { ... }  /* no ; */
```


11: Nested Structures

```
typedef struct {
     char firstName[20];
     char lastName[20];
     int age;
} InfoT;
typedef struct {
     InfoT info;
     double salary;
} EmployeeT;
EmployeeT e1;
e1.info.age = 21;
```


11: Assigning and Comparing Structures

```
InfoT i1, i2;
i1 = i2;
```

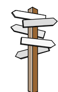
This a bitwise assignment, which only performs a *shallow* copy.

```
i1 == i2

strcmp(i1.firstName, i2.firstName) == 0 &&
    strcmp(i1.lastName, i2.lastName) == 0 &&
      i1.age == i2.age
```


11: Structures and Pointers

```
struct pair {
     double x;
     double y;
} w, *p;

typedef struct pair {
     double x;
     double y;
} PairT, *PairTP;
PartT x;
PairTP p;
```


Programming Guidelines
Structures and `typedef`

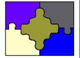


- Names of structures defined with `typedef` start with an upper case letter and end with the upper case `T`.
- Type names representing pointers to structures have names ending with `TP`

11: Structures and Pointers

```
 typedef struct pair {
      double x;
      double y;
 } PairT, *PairTP;
   PairT w;
   PairTP p = &w;
   PairTp q;
if((q = malloc(sizeof(PairT))) == NULL) …
if((q = malloc(sizeof(struct pair))) == NULL) …
   w.x = 2;
   p->x = 1;    (*p).x = 1;    *p.x = 1;
   q->y = 3.5;
```


Idioms



Member Access through Pointer

If `p` is a pointer to a structure that has a member `w`, then

`p->w`

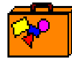
gives access to `w`.

Memory Allocation for a Structure

For a structure `s` and a pointer `p` to this structure, use:

```
if((p = malloc(sizeof(struct s)) == NULL) …
```


Portability



Size of structure

You can not assume that the size of a structure is the same as the sum of the sizes of all its members, because the compiler may use padding to satisfy memory alignment requirements.

11: Structures and Functions

```
typedef struct pair {
      double x;
      double y;
} PairT, *PairTP;
PairT constructorFunc(double x, double y) {
   PairT p;

   p.x = x;
   p.y = y;
   return p;
}
PairT w = constructorFunc(1, 2.2); /* COPY */
```



```
void constructorP(PairTP this,
                  double x, double y) {
   this->x = x;
   this->y = y;
}

PairT w;
PairTP p;

constructorP(&w, 1, 2); /* copy only doubles */

constructorP(p, 1, 2);
if((p = malloc(sizeof(PairT))) == NULL) error;
constructorP(p, 1, 2);
```



```
PairTP constructor(double x, double y) {
/* client responsible for deallocation */
   PairTP p;
   if((p = malloc(sizeof(PairT))) == NULL)
      return NULL;
   p->x = x;
   p->y = y;
   return p;
}

int compare(const PairTP p, const PairTP q) {
   return p->x == q->x && p->y == q->y;
}
```

11: Using a constructor

```
PairTP p1 = constructor(1, 2);
PairTP p2 = constructor(1, 3);

int i = compare(p1, p2);

free(p1);
free(p2);
```

Avoid leaving garbages:

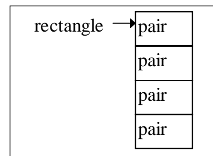
```
i = compare(p1, constructor(3.5, 7));
```


Errors

To allocate memory for a structure:

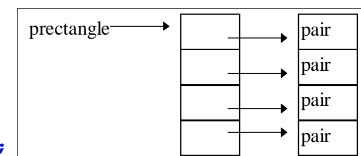
```
struct s { … };

malloc(sizeof(struct s *))

malloc(sizeof(struct s))
```


11: Blocks of Structures



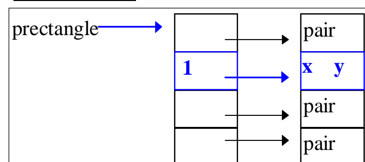

```
PairTP rectangle;
PairTP aux;
double x, y;
if((rectangle= malloc(4*sizeof(PairT)))==NULL)error;
 for(aux = rectangle; aux < rectangle + 4; aux++) {
   printf("Enter two double values:");
   if(scanf("%lf%lf", &x, &y) != 2) /* error */
     break;
   constructorP(aux, x, y);
 }
```


11: Blocks of Pointers to Structures




```
int i;
PairTP *prectangle;
for(i = 0; i < 4; i++) {
   printf("Enter two double values:");
   if(scanf("%lf%lf", &x, &y) != 2)
      error;
   if((prectangle[i] = constructor(x, y)) == NULL)
      error;
}
```

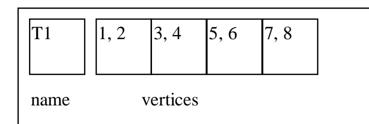

11: Blocks of Pointers to Structures




```
for(i = 0; i < 4; i++)
  printf("vertex %d = (%f %f)\n", i,
    prectangle[i][0].x, prectangle[i][0].y);
```

`prectangle[1]` is a pointer to the `1`-th pair,
`prectangle[1][0]` is the `1`-th pair,
`prectangle[1][0].x` is the `x` member.

11: Structures and Arrays



rectangle

```
#define MAX 20
typedef struct {
   char name[MAX+1];
   PairT vertices[4];
} RectangleT, *RectangleTP;
RectangleT rectangle;
```

T1	1, 2	3, 4	5, 6	7, 8
name	vertices			

rectangle

```
void Show(const RectangleTP s) {
   int i;
   printf("Rectangle %s\n", s->name);
   for(i = 0; i < 4; i++)
      printf("vertex %d = (%f %f)\n", i,
         s->vertices[i].x, s->vertices[i].y);
}
```

`s->name` the array of characters

`s->vertices` the array of pairs

`s->vertices[i]` the `i`-th pair

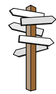
`s->vertices[i].x` the `x`-coordinate of the `i`-th pair

11: Initialization of Structures

```
typedef struct {
     double x;
     double y;
} PairT;
PairT q = {2.3, 4};
typedef struct {
     char name[MAX+1];
     PairT vertices[4];
} RectangleT;
RectangleT s = { "first",
           { {0, 1}, {2, 3}, {4, 5}, {1, 2} }
           };
```


Programming Guidelines

Arrays and Structures



For the sake of clarity, initializing arrays in structures always use nested braces.

11: Binary Files

There are up to 100 students, every student has name, up to 10 grades, the actual number of grades is stored, and the gpa :

```
#define STUDENTSN      100
#define MARKSN         10
#define NAMELENN       20
typedef struct {
   char name[NAMELENN+1];
   int marksNumber;
   double marks[MARKSN];
   double gpa;
} StudentT;
StudentT info[STUDENTSN];
```



```
/* Purpose: write number structures, contained in the
 *    array info, to the file fname.
 * Returns: 1 if successful; otherwise 0
 */
int saveStudent(const char *fname, StudentT info[],
                int number) {
FILE *out;
if((out = fopen(fname, "wb")) == NULL)
  return 0;
if(fwrite(info,sizeof(StudentT),number,out)!=number){
  fclose(out); return 0;
}
if(fclose(out) == EOF)
  return 0;
return 1;
}
```



```
/*
 * Purpose: initialize all of the gpa fields;
 *   reading data from a file and then
 *   writing them back to this file
 */
int updateGpa(const char *fname) {
  FILE *inOut;
  StudentT buffer;
  int i;
  if((inOut = fopen(fname, "r+b")) == NULL)
     return 0;
  /* read one structure at a time */
 while(fread(&buffer,sizeof(StudentT),1,inOut)!= 0) {
  for(i =0, buffer.gpa = 0.0; i < buffer.marksNumber;
         i++)          /* compute gpa */
    buffer.gpa += buffer.marks[i];
```

```
 if(buffer.marksNumber != 0)          /* set gpa */
   buffer.gpa /= buffer.marksNumber;
  /* move file pointer to the current structure */
 if(fseek(inOut, -sizeof(StudentT), SEEK_CUR)!= 0) {
   fclose(inOut); return 0;
 }
 if(fwrite(&buffer, sizeof(StudentT),1, inOut)==0) {
   fclose(inOut); return 0;
 }
} /* end of while */
if(!feof(inOut)) {
  fclose(inOut); return 0;
}
if(fclose(inOut) == EOF)
  return 0;
return 1;
}
```


11: Introduction to Linked Lists

A *list* is a collection of elements; each element contains data; here double values:

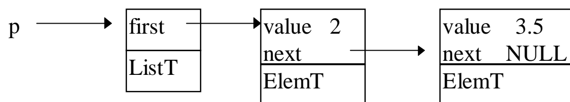
```
typedef double DataType;
typedef struct elem {
   DataType value;
   struct elem *next;
} ElemT, *ElemTP;
```

The value of `next` will be `NULL` if there is no next element, otherwise it will be a structure representing the next element.

11: Introduction to Linked Lists

```
typedef struct {
   ElemTP first;
} ListT, *ListTP;
```

The value of `first` is the first element of the list, or `NULL` if the list is empty:



11: Introduction to Linked Lists: create and destroy

```
ListTP construct(void) {
   ListTP p;
   if((p = malloc(sizeof(ListT))) == NULL)
      return NULL;
   p->first = NULL;
   return p;
}
void destruct(ListTP *this) {
   clear(*this);
   free(*this);
   *this = NULL;
}
```

Memory Allocation Idiom

Memory Deallocation Idiom

11: Introduction to Linked Lists

There are two kinds of list operations:

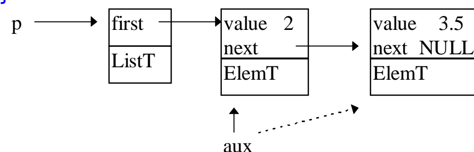
- *traversals*, for example print all elements in the list (you do not need to use memory management functions)
- *modifications*, for example insert or delete an element (you need to respectively use `malloc()` to insert a new element and `free()` to remove the existing element).

All list operations must preserve the following list *invariants*:

- for an *empty* list `p`, `p->first` is `NULL`
- for a non-empty list `p`, the value of `next` belonging to the *last* element is `NULL`.

11: Introduction to Linked Lists: Traversals

```
void printAll(const ListTP this) {
 ElemTP aux;
 for(aux=this->first; aux!=NULL; aux=aux->next)
    printf("%f\n", aux->value);
}
```



Traversal to the End of a List p

```
for(aux = p->first; aux!=NULL; aux=aux->next)...
```

Traversal to the Predecessor of the Last Object in a List

Assuming the list has more than one element

```
for(aux = p->first; aux->next != NULL;
    aux = aux->next)...
```


11: Introduction to Linked Lists: Modifications

```
int insertFront(ListTP this, DataType d) {
   ElemTP aux;

   if((aux = malloc(sizeof(ElemT))) == NULL)
      return 0;

   aux->next = this->first;   /* save state */
   aux->value = d;
   this->first = aux;
   return 1;
}
```


11: Introduction to Linked Lists: delete

```
int deleteLast(ListTP this, DataType *value) {
   ElemTP aux;

   if(this->first == NULL)  /* empty list */
      return 0;
   if(this->first->next == NULL) { /* single */
      *value = this->first->value;
      free(this->first);
      this->first = NULL;
      return 1;
   } ...
```

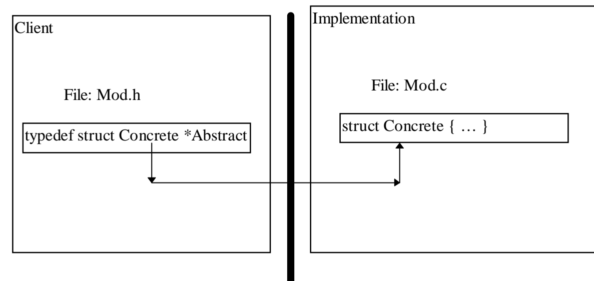

11: Introduction to Linked Lists: delete (cont.)

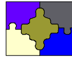
```
for(aux = this->first; aux->next->next != NULL;
   aux = aux->next)
      ;
   /* the predecessor of last element */

   *value = aux->next->value;
   free(aux->next);
   aux->next = NULL;

   return 1;
}
```


11: Introduction to Linked Lists: delete

```
int deleteFirst(ListTP this) {
   ElemTP aux = this->first;
   if(aux == NULL)   /* empty list */
      return 0;
   this->first = aux->next;
   free(aux);
   return 1;
}
void clear(ListTP this) {
   while(deleteFirst(this))
      ;
   this->first = NULL;
}
```


11: Implementation of Opaque Types





Opaque

In order to represent an opaque type `Abstract` use the definition of the form

```
typedef struct Concrete *Abstract
```

in the header file.

Define the structure `Concrete` in the implementation file.

11: Multiple Generic Enumerations

```
typedef struct Concrete_Enumeration
                *Abstract_Enumeration;
```

← Opaque Idiom

Interface:

```
Abstract_Enumeration create_Enumeration(void *);
int
  hasMoreElements_Enumeration(Abstract_Enumeration);
void* nextElement_Enumeration(Abstract_Enumeration);
/* use free() to destruct */
/* Auxiliary macro: */
#define nextElementTyped_Enumeration(type, p) \
   ((type)nextElement_Enumeration(p))
```


11: Generic Dynamic Array Module with Enumerations

A generic module `Arr` that can be used to create dynamic arrays of *any* data type.
The client can create multiple enumerations used to traverse the array.

```
typedef struct Concrete_Arr *Abstract_Arr;

Abstract_Arr construct_Arr(size_t elSize,
                           int initSize);
void *get_Arr(const Abstract_Arr this, int i);
void set_Arr(Abstract_Arr this, void *value, int i);
int length_Arr(const Abstract_Arr this);
void destruct_Arr(Abstract_Arr *this);
```


11: Application of `Arr`

Read a line of *unlimited* length from a file:

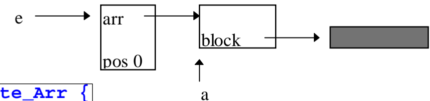
```
a1 = construct_Arr(sizeof(char), 80));

for(i = 0; (c = fgetc(f)) != '\n'; i++) {
   if(c == EOF) {
     fprintf(stderr, "could not read first line\n");
     destruct_Arr(&a1);
     return EXIT_FAILURE;
   }
   set_Arr(a1, &c, i);
}
```


11: Application of `Arr`

```
/* show using an enumeration */
e1 = construct_Enumeration(a1);
while(hasMoreElements_Enumeration(e1))
  putchar(nextElementTyped_Enumeration(char, e1));

/* destruct an array and an enumeration */
destruct_Arr(&a1);

free(e1);
```


11: Implementation of `Arr`

a = construct_Arr()
e = construct_Enum(a)




```
struct Concrete_Arr {
   void *block_;
   int size_;
   size_t elSize_;
   int init_;
};
```

```
struct Concrete_Enumeration {
   int pos;
   Abstract_Arr arr;
};
```

```
Abstract_Arr construct_Arr(size_t elSize,
                           int initSize) {
/* Array constructor */
 Abstract_Arr arr;

 if((arr=malloc(sizeof(struct Concrete_Arr)))==NULL)
   return NULL;
 arr->init_ = 1;
 arr->elSize_ = elSize;
 if((arr->block_ =calloc(initSize, elSize))==NULL) {
   free(arr);
   return NULL;
 }
 arr->size_ = initSize;
  return arr;
}
```



```
Abstract_Enumeration construct_Enumeration(void *p) {
/* Enumeration constructor */
   Abstract_Enumeration e;

   if((e = calloc(1,
       sizeof(struct Concrete_Enumeration))) == NULL)
    return NULL;

   e->arr = (Abstract_Arr)p;
   e->pos = 0;

   return e;
}
#define VOID(targ, size) ((void*)(((char*)(targ))\
                           +(size)))
```



```
static int expand_(Abstract_Arr this, int size) {
/* private function to expand */
   void *new;

   if((new = calloc(size, this->elSize_)) == NULL)
      return 0;

   memcpy(new, this->block_, this->size_);
   /* copy the old array */

   this->size_ = size;
   /* update size_ and block_ */
   free(this->block_);
   this->block_ = new;
   return 1;
}
```



```
void set_Arr(Abstract_Arr this, void *value, int i) {
/* set the i-th element */
   int res;

   assert(i >= 0 && this->init_);
   if(i < 0 || !this->init_)
      return;
   if(i >= this->size_) { /* expand */
      res = expand_(this, i + increment_);
      assert(this);
      if(res == 0)
         return;
   }
   memcpy(VOID(this->block_, i*this->elSize_),
      value, this->elSize_);
}
```



```
void* nextElement_Enumeration(Abstract_Enumeration e)
  {
/* get the next enumeration element */
   void *aux;

   assert(e->arr->init_);
   if(e->pos == e->arr->size_)
      return NULL;

   aux = VOID(e->arr->block, e->pos*e->arr->elSize_);
   e->pos++;

   return aux;
}
```


11: `List` Module

1) the type `DataType` of the data stored in list elements is *known to the implementation*
2) any number of lists can be created; all lists must have elements of the *same* type, `DataType`
3) the module support enumerations
4) the module does *not* support persistence
5) the representation of the list and list elements are not visible to the client.

This particular version of `List` will operate on a list of strings.

11: Shallow and Deep Copy

To insert a new element `d` into the list:

```
  aux->value = d;
```

(a *shallow* copy).
For a deep copy, use a callback function `copyData_List()`.
For example, for strings and doubles:

```
DataType copyData_List(const DataType v) {
   return strdup(v);
}
DataType copyData_List(const DataType v) {
   return v;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

11: Shallow and Deep Copy

We need another *callback* function, `freeData_List()`
For example, for string and doubles:

```
void freeData_List(DataType v) {
   free(v);
}

void freeData_List(DataType v) {
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

11: Interface of `List`

```
typedef char* DataType;
typedef struct Concrete_List *Abstract_List;

DataType copyData_List(const DataType v);
void freeData_List(DataType v);
Abstract_List construct_List(void);
int insert_List(Abstract_List this, int pos,
                DataType value);
int delete_List(Abstract_List this, int pos,
                DataType *value);
int length_List(const Abstract_List this);
void destruct_List(Abstract_List *this);
```

 Copyright: Addison-Wesley Publishing Company, 2000

11: Application of `List`

```
if((a1 = construct_List()) == NULL) {
   fprintf(stderr, "constructor failed\n");
   return EXIT_FAILURE;
}
/* read from a file words and store in the list */
while(fscanf(f, "%80s", buffer) == 1)
   if(insert_List(a1, 0, buffer) == 0) {
        fprintf(stderr, "insert list failed\n");
        destruct_List(&a1);
        return EXIT_FAILURE;
   }
```

 Copyright: Addison-Wesley Publishing Company, 2000

11: Application of `List`

```
if(fclose(f) == EOF) {
     fprintf(stderr, "file closing failed\n");
     destruct_List(&a1);
     return EXIT_FAILURE;
}
/* show what is in the list */
  e1 = construct_Enumeration(a1);
  printf("words backwards are:\n");
  while(hasMoreElements_Enumeration(e1)) {
     aux=nextElementTyped_Enumeration(char*, e1);
     printf("%s\n", aux);
     free(aux);
  }
```

 Copyright: Addison-Wesley Publishing Company, 2000

11: Implementation of `List`

```
typedef struct elem {
   DataType value;
   struct elem *next;
} ElemT, *ElemTP;
```

Interface:

```
typedef struct Concrete_List *Abstract_List;
```

Implementation:

```
struct Concrete_List {
  ElemTP first;
};
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
Abstract_List construct_List(void) {
/* list constructor */
   Abstract_List p;

   if((p = malloc(sizeof(struct Concrete_List)))
          == NULL)
      return NULL;

   p->first = NULL;

   return p;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
int insert_List(Abstract_List this, int pos,
                DataType value) {
/* insert at i-th position */
  int i;
  ElemTP auxp, auxm;
  int length = length_List(this);
  if(pos < 0 || pos > length)
     return 0;
  if(pos == 0) /* in front */ {
    if((auxm = malloc(sizeof(ElemT))) == NULL)
       return 0;
    auxm->value = copyData_List(value);
    auxm->next = this->first; /* connect with old */
    this->first = auxm;
    return 1;
  }
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
if(this->first == NULL) /* empty list */
     return 0;
  /* search */
  for(i = 1, auxp = this->first; i < pos;
         i++, auxp = auxp->next)
     ;
  /* insert after auxp */
  if((auxm = malloc(sizeof(ElemT))) == NULL)
     return 0;
  auxm->value = copyData_List(value);
  auxm->next = auxp->next;
  /* connect with old list*/
  auxp->next = auxm;

  return 1;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
void* nextElement_Enumeration(
         Abstract_Enumeration e) {
/* get the next element */
   DataType aux;

   if(e->pos == NULL)
      return NULL;

   aux = copyData_List(e->pos->value);
   e->pos = e->pos->next;

   return aux;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

Chapter 12:

Enumerated Types and Unions

 Copyright: Addison-Wesley Publishing Company, 2000

12: Enumerated Types

Enumerated types are ordered collections of named constants; e.g.

```
enum opcodes {
   lvalue, rvalue, push, plus
};
typedef enum opcodes {
   lvalue, rvalue, push, plus
} OpcodesT;

enum opcodes e;
OpcodesT f;
```

 Copyright: Addison-Wesley Publishing Company, 2000

12: Enumerated Types

The definition of

```
enum opcodes {
   lvalue, rvalue, push, plus
};
```

introduces four constants: `lvalue`, `rvalue`, `push`, and `plus`; all are of type convertible to `int`:

`lvalue`	represents the integer value	0
`rvalue`		1

and so on

 Copyright: Addison-Wesley Publishing Company, 2000

12: Enumerated Types

A declaration of an enumerated type may also *explicitly* define values:

```
enum opcodes {
   lvalue = 1, rvalue, push, plus
};
   enum opcodes e;
   e = lvalue;
   if(e == push) …

   int i = (int)rvalue; /* equal to 2 */
```

 Copyright: Addison-Wesley Publishing Company, 2000

12: Enumerated Types

To represent function return codes; e.g.

failure because a file can not be opened
failure because a file can not be closed
success

```
typedef enum {
   FOPEN, FCLOSE, FOK
} FoperT;
#define TOINT(f)   ((int)(f))
```

 Copyright: Addison-Wesley Publishing Company, 2000

12: Enumerated Types

Consider a function

```
 FoperT process();
```

To output the result of calling this function as a string

```
char *Messages[] = {
      "File can not be opened",
      "File can not be closed",
      "Successful operation",
      "This can not happen"
};

printf("result of calling process() is %s\n",
   Messages[TOINT(process())];
```

 Copyright: Addison-Wesley Publishing Company, 2000

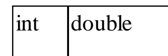
Programming Guidelines
Maintainability

- Store all program messages in one place; it will be easier to maintain them.
- When you consider various cases, do not forget to account for the "impossible" cases; they do happen.

 Copyright: Addison-Wesley Publishing Company, 2000

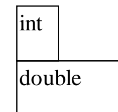
12: Unions

```
struct intAndDouble {
   int i;
   double d;
};
```



initAndDouble

```
union intOrDouble {
   int i;
   double d;
};
```



initOrDouble

 Copyright: Addison-Wesley Publishing Company, 2000

12: Unions

```
union intOrDouble {
   int i;
   double d;
} z;
```

You can access either member, using the familiar notation:

```
   z.d represents a double
   z.i represents an int
   z.d = 1.2;
   z.i = 4;
   printf("%d", z.i);
   printf("%f", z.d);
```

 Copyright: Addison-Wesley Publishing Company, 2000

12: Unions

Add a **tag field**:

```
typedef enum {
   integer, real
} TagTypeT;
typedef union {
   int i;
   double d;
} IntOrDoubleT;
typedef struct {
   TagTypeT tag;
   IntOrDoubleT value;
} TaggedValueT;
```

 Copyright: Addison-Wesley Publishing Company, 2000

12: Unions

```
typedef struct {
   TagTypeT tag;
   union {
      int i;
      double d;
   } value;
} TaggedValueT;
TaggedValueT v;
if(v.tag == integer)
   …v.value.i…;
else
   …v.value.d…;
```

 Copyright: Addison-Wesley Publishing Company, 2000

Chapter 13:

Bitwise Operations

 Copyright: Addison-Wesley Publishing Company, 2000

13: Preview

C provides the following bitwise operations:

`&`	bitwise and
`|`	bitwise or
`^`	bitwise xor, also called exclusive or
`<<`	left shift
`>>`	right shift
`~`	one's complement

Notation:

$(z)_i$ denotes the `i`-th bit in `z`

 Copyright: Addison-Wesley Publishing Company, 2000

13: Bitwise &

if $(x)_i == 1$ and $(y)_i == 1$ then $(x \& y)_i == 1$

otherwise $(x \& y)_i == 0$

Bitwise *and* is often used to *clear* bits or bytes. For example:

`x & 0xff` clears all bytes but the low-order byte:

				low-order byte
	x	…		B
&	0xff	0…0	0…0	11111111
	x & 0xff	0 … 0		B

 Copyright: Addison-Wesley Publishing Company, 2000

13: Bitwise |

if $(x)_i$ == 1 *or* $(y)_i$ == 1 then `(x | y)`$_i$ == 1
otherwise `(x | y)`$_i$ == 0

This operation is often used to *set* bits. For example to set the high order bit of the low order byte of `x`, use `x | 0x80`,

		low-order byte
x	C	$B_1B_2B_3$…
| 0x80	0…0	10000000
x | 0x80	C	$1B_2B_3$…

13: Bitwise ^

if $(x)_i$ == $(y)_i$ then `(x ^ y)`$_i$ == 0
otherwise `(x ^ y)`$_i$ == 1

Bitwise *xor* clears those bits that are the same in both arguments, and sets the other bits.
Can be used to test if two words are equal, for example
`x ^ y`
returns 0 if `x` and `y` are equal.

13: Bitwise ~

Bitwise complement: `~x`

`(~x)`$_i$ == 1 - $(x)_i$

clear any bits that are 1, and set those that are 0.
For example, assume that you want to clear the 3 low-order bits of `x`, and use a value that has 1's in all the other bits:
`x & ~7`

Similarly, to set a value (i.e. all bits equal to 1), use
`~0`

13: Left Shift

Left shift: `i << j`

The resulting word will have all bits shifted to the left by `j` positions;
for every bit that is shifted off the left end of the word, there is a zero bit added at the right end.

`x <<= 1` is equivalent to `x *= 2`

`x <<= 2` is equivalent to `x *= 4`.

13: Right Shift

Right shift: `i >> j`

The resulting word will have all bits shifted to the right by `j` positions; for every bit that is shifted off the right end of the word, there is a zero bit added at the left end.

`x >>= 1` is equivalent to `x /= 2`

`x >>= 2` is equivalent to `x /= 4`.

Portability

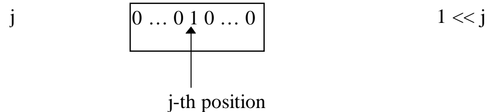
- Perform shifts only on unsigned integer numbers. Do not use shifts with negative values, such as

`x << -2`

13: Examples of bitwise operations

`getBit()` returns the `i`-th bit in `w`, using a bitwise *and* of its first argument and `MASK(j)`:

```
#define MASK(j)    (1 << j)
int getBit(int w, unsigned j) {
   return ((w & MASK(j)) == 0) ? 0 : 1;
}
```



 Copyright: Addison-Wesley Publishing Company, 2000

13: Bitfields

Bit fields provide a *non-portable* way to pack integer components into a memory block that is smaller than typically allocated:

```
struct {
   unsigned leading : 3;
   unsigned FLAG1: 1;
   unsigned FLAG2: 1;
   trailing: 11;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

Chapter 14:

Finale

 Copyright: Addison-Wesley Publishing Company, 2000

14: Preview

- continuation of the discussion of error handling
- characterization of the various kinds of modules
- shallow and deep interfaces.
- implementation of other data structures.

 Copyright: Addison-Wesley Publishing Company, 2000

14: Error Handling

There are several techniques to handle errors:

- Using the following interface:
  `isError_IO()` to test whether or not the error has occurred
  `clearError_IO()` to clear any error indication
  `printError_IO()` to print a description of the error
- Error handling provided by C: `errno`, and two functions: `strerror()` and `perror()`
- Testing of preconditions
- Java exception handling (absent in C)

 Copyright: Addison-Wesley Publishing Company, 2000

14: Error Handling Long Jumps

For a variable `env` of type `jmp_buf`, the call

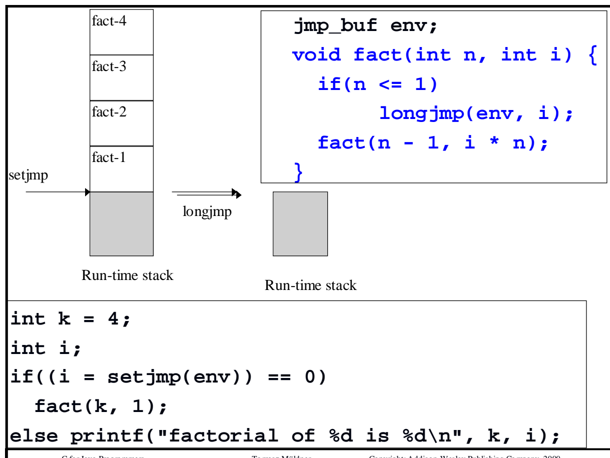
```
setjmp(env)
```

saves the current state of the run-time stack in `env`, and returns 0 to the caller. A subsequent call to

```
longjmp(env, val)
```

restores program execution at the point saved by `setjmp()`: control is returned to `setjmp(env)`, which returns the value `val`.

 Copyright: Addison-Wesley Publishing Company, 2000




```
jmp_buf env;
void fact(int n, int i) {
  if(n <= 1)
       longjmp(env, i);
  fact(n - 1, i * n);
}
```

```
int k = 4;
int i;
if((i = setjmp(env)) == 0)
  fact(k, 1);
else printf("factorial of %d is %d\n", k, i);
```


14: Simulating Exception Handling

```
typedef struct {
   char *reason;
   jmp_buf env;
} ExceptionT;

#define TRY(exc)        setjmp(exc.env)
#define SHOW(exc)       exc.reason
#define RAISE(exc, str)   {\
                   exc.reason = strdup(str); \
                   longjmp(exc.env, 1);      \
                      }
```



```
ExceptionT e;
/* global variablel specifies the exception */
int add(int x, int y) {
   if(x > 0 && y > 0 && x > INT_MAX - y)
      RAISE(e, "Overflow")
   if(x < 0 && y < 0 && x > INT_MIN + y)
      RAISE(e, "Underflow")
   return x + y;
}
char *catch(int x, int y, int *res) {
   if(TRY(e))
      return SHOW(e);
   *res = add(x, y);
   return NULL;
}
```


14: Characterization of Modules

- **Multiplicity**:
  Singleton (*at most one* of instance can be created)
  Ordinary (*any number* of instances can be created)
- **Generocity**:
  Generic (the element type is *not* known to the implementation)
  Concrete (the element type *is* known to the implementation)
- **Containment**:
  Shallow (using *standard assignment*)
  Deep (*cloning*)
- **Persistence**:
  Transient (exist only in the *internal* memory)
  Persistent (can be saved in the *external* memory)

14: Enumerations. Shallow and Deep Interfaces

The enumeration interface consists of:

```
int
  hasMoreElements_Enumeration(Abstract_Enumeration);
Abstract_Enumeration construct_Enumeration(void *);
```

(the same for a shallow and a deep interface); and:

```
void*
  nextElement_Enumeration(Abstract_Enumeration e)
```

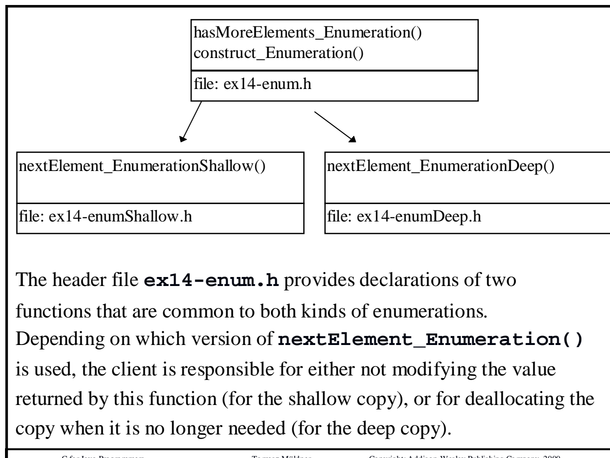
(different for a shallow and a deep interface).

14: Enumerations. Shallow and Deep Interfaces

Consider enumerations over a list of strings :

```
char *p = nextElement_Enumeration(e);
```

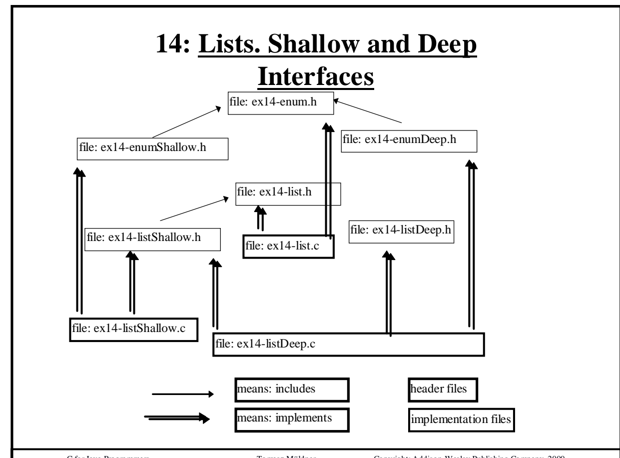
- Shallow copy
  The client can freely *access* the value of p, for example, to print it but must *avoid* modifying the value of p. To modify the data:
  `aux = strdup(p);`
- Deep copy
  The client is responsible for deallocating this copy:
  `free(p);`



The header file **`ex14-enum.h`** provides declarations of two functions that are common to both kinds of enumerations. Depending on which version of **`nextElement_Enumeration()`** is used, the client is responsible for either not modifying the value returned by this function (for the shallow copy), or for deallocating the copy when it is no longer needed (for the deep copy).

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: Lists. Shallow and Deep Interfaces



C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: Lists. Interface

```
typedef struct Concrete_List *Abstract_List;
Abstract_List construct_Lists(size_t elSize,
  void* (*copyData)(const void*),
  void (*freeData(void*));
int length_Lists(const Abstract_List this);
void destruct_Lists(Abstract_List *this);
int insert_ListsShallow(Abstract_List this,
                        int pos, void *value);
int delete_ListsShallow(Abstract_List this,
                        int pos, void *value);
int insert_ListsDeep(Abstract_List this,
                     int pos, void *value);
int delete_ListsDeep(Abstract_List this,
                     int pos, void *value);
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: Lists. Implementation

```
typedef struct elem {
   void *value;
   struct elem *next;
} ElemT, *ElemTP;
struct Concrete_List {
   ElemTP first;
   size_t elSize;
   void* (*copyData)(const void*);
   void (*freeData)(void*);
};
```

void *value struct elem *next
ElemT

size_t elSize ElemTP first
Concrete_List

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: List Constructor

```
Abstract_List construct_Lists(size_t elSize,
         void* (*copyData)(const void*),
         void (*freeData)(void*)){
   Abstract_List p;
   if((p = malloc(sizeof(struct Concrete_List)))
         == NULL)
      return NULL;
   p->first = NULL;
   p->elSize = elSize;
   p->freeData = freeData;
   p->copyData = copyData;
   return p;
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: Enumeration

```
struct Concrete_Enumeration {
   Abstract_List list;
   ElemTP pos;
};
Abstract_Enumeration construct_Enumeration(void *p) {
 Abstract_Enumeration e;
 if((e=calloc(1,sizeof(struct Concrete_Enumeration)))
      == NULL)
   return NULL;
 e->list = (Abstract_List)p;
 e->pos = e->list->first;
 return e;
}
```

C for Java Programmers Tomasz Müldner Copyright: Addison-Wesley Publishing Company, 2000

14: Deleting

```
int delete_ListsDeep(Abstract_List this,
                     int pos, void *value) {
   int length = length_Lists(this);
   ElemTP auxp, auxm;
   int i;

   if(this->first == NULL)
      return 0;

    if(pos < 1 || pos > length)
      return 0;
```

 Copyright: Addison-Wesley Publishing Company, 2000

```
if(pos == 1) { /* delete first */
  auxp = this->first;
  this->first = this->first->next;
  value = (this->copyData)(auxp->value);
  (this->freeData)(auxp->value);
  free(auxp);  return 1;
}
for(i = 1, auxp = this->first; i < pos-1;
  i++, auxp = auxp->next)
      ;
   auxm = auxp->next; /* will be deleted */
   auxp->next = auxp->next->next;
   value = (this->copyData)(auxm->value);
   (this->freeData)(auxm->value);
   free(auxm);  return 1;
}
```

 Copyright: Addison-Wesley Publishing Company, 2000

This is the end of my presentation.

If you have any comments, or corrections, please send email to:

tomasz.muldner@acadiau.ca

 Copyright: Addison-Wesley Publishing Company, 2000