Chapter 4:

Control Structures

Basic Control Structures

- -for (init;bool;inc) *statement*
- -while (bool) *statement*
- -if (bool) *statement* else *statement*
- switch (variable) {

  case value: *statement*

  case value: *statement*;

  default: *statement*

  }

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 xdone;

• xdone:
```

This (goto) should be used with discretion, or not at all for beginners.

Example

```
/* 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
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:
  - BUT, IF YOU ALWAYS PUT IN THE BRACES YOU'LL AVOID ONE OF THE MOST COMMON SOURCES OF ERROR

```
•         if(condition)
•              single statement
•              second statement  ←
```

- if (cond) stmt

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(e
•             st
```

- is equivalent to:

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

```
But I strongly suggest either
   while (expr) statement;
or
   while (expr) {
      statement;
      }
```

```
• /* Example 4.4    Example
•  * 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;

•
```

Example

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

 for (;;) {
  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 =
       break;
    if(aux == SENTINEL)
       break;
    ...
```

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

Example

```
/*
 * 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;
```

Example

```
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;
  …
}
```

Example

"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

Errors

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

```
e1 & e2

e1 | e2

if(x = 1) …     if ((x=1)!=0)
```

Chapter 5:

Text Files

5: Preview

- I/O operations on *streams*. *Loose connection to files.*
- Not such a clear division into input streams and output streams
- Files are sequences of bytes
- Text files: buffering (*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: ^D (control-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

- mail me < foo > bar

Idioms

File idioms

Opening a file

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

Closing a file

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

Errors

File errors

- To declare **`FILE`** variables, do not use

```
FILE *f, g;   (the "g" part is wrong)
```

- Do not use

```
open()
```

or

```
close()
```

5: Basic File I/O Operations

Familiar	More general cousin
`int getchar()`	`int fgetc(fileHandle)`
`int putchar(int)`	`int fputc(int, fileHandle)`
	Note nonstandard args
`int scanf(…)`	`int fscanf(fileHandle, …)`
`int printf(…)`	`int fprintf(fileHandle, …)`

Example

```
• /*
•  * 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

Example

```
• /*
•  * Example 5-1
•  * Program that reads three real values from the
•  * file "t" and displays on the screen the
•  * sum of these values
•  */
• #include <stdio.h>
• int main() {
•    FILE *f;
•    double x, y, z;
•    char *fname = "t";

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

"Opening a file"
Idiom

Example

```
   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

Single item 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')&&(c!=EOF))
•      putchar(c);

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

Idioms

Line idioms

Read a Line

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

Read a Line from a File

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

Idioms

End-of-line 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')
   ;
```

Examples

- Count occurrences of a specific letter in a file:
  - open file
  - read characters
    - if it's 'a' then increment count
  - print result
- Print a string triggered by a one-character code
  - read code character
  - switch on character
- Strip carriage returns (Windows text->UNIX text)
  - read characters from a file called "test1"
  - if not a carriage return, dump it out.

Assignment 1 solution outline

- Assignment 1 solution discussion will be deferred until next class.
  - Some people are too stressed out.
  - One person has a serious medical issue and wants to be here.
  - I want to cover material in preparation for your next assignment.
  - I want to examine the solutions that were submitted.

Assignment 2, C programming

- Assignment 2 will be distributed next class, barring any unforeseen events.
- Due date will be *about* 8 **school days** after it is given out.

Midterm

- The midterm is the Tuesday before February break.
- During class.
- Come to class as usual, unless otherwise instructed.
  - Check the class web page or the preceding class for any changes in plan re. the **venue**.

Example

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

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

"Opening a file"
Idiom

Example

```
while((c = fgetc(fileHandle)) != EOF){
  if(c == TARGET ) i++;
}
```

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

```
printf("There are %d occurrences of %c\n",
         i, TARGET );
if(fclose(fileHandle) == EOF) {
    fprintf(stderr, "can't close %s\n", "t");
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
```

"Closing a file"
Idiom

Example

```
/* 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

Example

```
 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;
 case EOF:     ….

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

Example

```
/* Modify an existing file to remove all
 * occurrences of ^M ('\r') from file test1
 * (that's the "carriage return" from the
 * windows world.)
 */
 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

Example

```
/* filter out all ^M ('\r') */
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

Example

```
  /* 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

Compilation

- C programs are transformed into executables in a sequence of 3 steps:
  - All these steps are performed apparently together by the command

    cc or gcc

  - **1) Transform C code into C code, but support certain simplifications.**
    - **THIS IS THE JOB OF THE C-PREPROCESSOR: cpp**
  - 2) Compile C code into object code (compiler)
  - 3) Combine object code modules into a single (more or less) monolithic executable (linker, called "ld")
    - [footnote: on many systems there is dynamic linking that occurs just before, or even during, execution and performs further linking.]

cpp

- macros (with and without parameters)
  - e.g. EOF is actually (-1)
- conditional compilation
- file inclusion
- predefined macros
- applications for debugging

- **macro** $\xrightarrow[\textit{macro substitution}]{}$ **macro replacement**

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 "
```

- #define foo 9

- int foo;
- foo = 8;

- int 9;
- 9=8;

6: Parameterless Macros

```
• #define PROMPT    printf("Enter real value: ")
• #define SKIP      while(getchar() != '\n');
• #define  ZX  "this\
• is\
• a\
• long"

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

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

•    foobar =   3 * A
```

- {
- int c;
- PROMPT;
- c=getchar();
- SKIP
- }

Programming Guidelines
Preprocessing

- Macros names will always appear in upper case.
  - Not a rule from the compiler or cpp, but one you should adhere to for the sake of readability.
- Any constant value, which might change during software development should be defined as a macro, or as a constant.
  - (Older versions of C didn't have constants.)

- By using macros, you are adding new constructs and new functionality to the language – if you do this inappropriately, the readability of your code may suffer.

fubar.c

```
#include <stdio.h>
#define QQ     1
#define TT       1
#define cc main(c,v) int c; char **v;{char tt[12],qq[7]; int q=0,o=1,l=1,m=1;struct {int c;} f;
#define incest qq[6]='\0';tt[11]='\0';if(QQ==atoi(v[1])+1){(void)fprintf(stderr,"%s factorial = %d\n",v[1],
      TT);exit(1);}o=c+f
#define x ;while(EOF!=(o=getchar())){if(l && q=='Q' &&
      o=='Q'){l=0;(void)getchar();(void)fread(qq,6,1,stdin);(void)printf("Q %6d",atoi(qq)+1);}else
if(m && q=='T' && o=='T'){m=0;(void)fread(tt,11,1,stdin);(void)printf("T %9d\n",atoi(tt)*QQ);}else
      {q=o;(void)putchar(o);}}exit(0);}
cc incest.c -o x
#define zxc ;{/*
cat incest.c | x $1 >! x1
if ($status != 0) then
exit
endif
mv x1 incest.c
chmod +x incest.c
exec incest.c $1
exit
*/
```

fubar.sh

```
•  :
•  # to run fubar in the 'proper' way

•  # parse args
•  if [ $# -ne 1 ]; then
•      echo "usage: $0 number" 1>&2
•      exit 1
•  fi

•  # run/compile it
•  rm -f ouroboros.c x1 x
•  ex - <<EOF
•  r fubar.c
•  7,8j
•  w ouroboros.c
•  EOF
•  chmod +x ouroboros.c
•  ouroboros.c $1
•  rm -f ouroboros.c x1 x
```

vanb.c

```
main(Q,O)char**O;{if(--
  Q){main(Q,O);O[Q][0]^=0X80;for(O[0][
  0]=0;O[++O[0][0]]!=0;)if(O[O[0][0]][
  0]>0)puts(O[O[0][0]]);puts("--------
  --");main(Q,O);}}
```

- David Van Brackle
- Department of Computer Science
- University of Central Florida
- Orlando, Florida
- 32816
- USA

This program computes all proper subsets of the set of arguments passed to it. Each subset is printed with one element on each line, followed by a line of ten dashes.

Try:

```
vanb the rug gary lent
vanb unix is better than os/2
```

Selected notes from the author:

The program has the following charming and possibly non-portable features:

* It has no local or global variables, only the command-line parameters.

* It calls main recursively.

* It alters the command-line parameters.

* It uses the fact that if the high bit is set in a character variable, the value is negative.

```
• 91% cc vanb.c
• 92% ./a.out the rug gary lent
rug
gary
lent
----------
gary
lent
----------
the
gary
lent
----------
the
lent
----------
lent
```

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 line %s\n", __LINE__);`

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

Errors

```
#define PI = 3.14

#define PI 3.14;

#define F (x) (2*x) < F is parameterless (space)
 On use:      y=F(2)    ->>  y=(x) (2*x)(2)
 #define F(x) (2*x)
             y=F(2)    ->> y=(2*2);
             y=F(j*3)   -->>  y=(2*j*3)
```

- To be safe, enclose the entire macro body, as well as each occurrence of a macro argument, in parentheses.
- Avoid side effects in macro arguments.

6: File Inclusion (cpp's job too)

`cpp looks in 2 "kinds" of places:`

- **`#include "filename"`** the *current* directory and ...
- **`#include <filename>`** special *system* (i.e. default) directories

  Aside: These defaults are controlled by the -I flag to cc/gcc
-
- 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()`
- **`string.h`** - string comparison function (see man string)

6: Conditional Compilation (1)

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

6: Conditional Compilation (2)

```
#ifdef macroName   <"if macro has been defined"
   part1
#else
   part2
#endif

#ifndef macroName   < "if macro has not been defined"
   part1
#else
   part2
#endif
```

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);`**

Example

```
• 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;
• }
```

Example

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

Example

```
#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
 ...
```

6: Header files

- To avoid multiple inclusion:

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

Errors

◆ Avoid side effects in macro arguments:

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

Errors

◆ probably want `SQR(i); i++;`

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
```

My favorite

- Debugging for when you are really stuck:
  - printf is your friend.
- In desperation, use LOTS of printfs, but then inserting them and removing them is a chore, and the output becomes cluttered

```
myfunc(int x, char c)
{
    printf("Starting myfunc\n");
    x = c;
    printf("Assigned to x\n");
        etc.
```

My favorite (solution)

Define some debug-specific macro code (ideally in a header)

```
#ifdef DEBUG
   #define debug(x)   printf(x)
#else
   #define debug(x)    /* x */
#endif

myfunc(int x, char c)
{
    debug("Starting myfunc\n");
    x = c;
    debug("Assigned to x\n");
        etc.
```

Define a macro at compile-time with the flag:

```
gcc -Dname
```

as in

```
gcc -DDEBUG  -DUSERID=3
```

Chapter 7:

Functions, Scope, and Introduction to Module-based Programming

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

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*

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.

Programming Guidelines
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**:

Programming Guidelines
Function documentation

- Documentation may also include a **Bugs** (or **features**) 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.
  - Keep in mind that code is something that people will have to read, often you, at some later date.

7: Review 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.

Example

```
•  /* 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;
•  }
```

Aside: MAX is often accomplished with a macro

- Why?
  - For a function the computer must
    - prepare arguments for transmission (put them on the stack, perhaps)
    - call the function
    - execute the function (of course)
    - return
  - A function has a fixed type
  - (However, a function avoids duplicated code)
- First, recall the expression (bool)?v1:v2
  - which means roughly: if (bool) v1; else v2;

MAX as a macro

- #define MAX(a,b) a>b?a:b
- WRONG!
  - imagine the code y = MAX(a,b)+2
  - we get y = a>b?a:b+2
    - which will assign either (a) or (b+2)
      - Note also the contrast with y= 2+MAX(a,b)
- #define MAX(a,b) ((a)>(b)?(a):(b)) <u>is just fine</u>

Example

```
/* 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 <= 0 (book's (x==0) is not quite right
                           here)
 * Sample call: d = sqrtRev(2.4);
*/
double sqrtRev(double);
#include <math.h>  /* gcc -lm … */
double sqrtRev(double x) {
   return sqrt(1/x);  /*SHOULD CHECK X>0*/
}
```

Example

```
/* 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 n);
```

Example

```
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();`
  `(discarding the return value)`

- 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

Errors

```
◆  double v = f(2.5); /* call before decl. */
   double f() { … }

◆  double f() { return 2.5; }
A  double f() /* too late */

◆  double f(double v) {
     if(v == 0) return; /* no ret'n value */
   }
```

- The code parameter of `exit()` should be one of the two values:
  `EXIT_SUCCESS` or `EXIT_FAILURE.`

7: `void` and Conversions

- Recall this slide: `int f()` is equivalent to `int f(void)`
- Call:
- `f();` is equivalent to `(void)f();`
  `(discarding the return value)`

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

What about using a function that has not been defined yet?
THIS IS ALLOWED IN C!

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 more 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++;
•        ..
•    }
```

- **`A static variable retains its value between excutions of the block it's in!`**
- register variables:

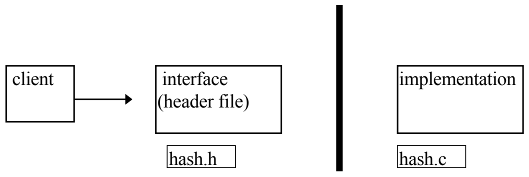
```
•         register int i;
                                  advisory only.
```

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

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`**

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.
  (Uh, well, maybe)

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.