

## Types, loops, subprograms (expanded review of CS I)

- Bitwise operators
- The preprocessor
- Numeric and character data
- Loops
- Subprograms

David Keil 1/03 1

## Subtopic: Bitwise operations

- Operations: AND (&), OR (|), XOR (^), complement (~), shift (<<, >>)
- Each operator performs a logical operation on each bit of operand(s)
- Applications:
  - Compact storage of status data
  - Storage of sets
  - Arithmetic at hardware level
  - Systems programming

David Keil 1/03 2

### Shift-left operator multiplies by a power of 2

```
void main()
{
    cout << "Enter an integer: ";
    unsigned int input;
    cin >> input;
    cout << input << " * 8 = "
        << (input << 3) << endl;
}
```

[mult8.cpp]

Sample I/O:  
Enter an integer:  
6 6 \* 8 = 48

Shift-left-3 multiplies  
by 8 because  $2^3 = 8$

David Keil 1/03 3

### Bitwise examples

C/C++

Operation operator	Example
Complement	~ ~ 10000000 <sub>2</sub> = 01111111 <sub>2</sub>
OR	1100 <sub>2</sub>   1001 <sub>2</sub> = 1101 <sub>2</sub>
AND	& 1100 <sub>2</sub> & 1001 <sub>2</sub> = 1000 <sub>2</sub>
Left shift	<< 1101 <sub>2</sub> << 1 = 11010 <sub>2</sub>
Right shift	>> 11000 <sub>2</sub> >> 2 = 110 <sub>2</sub>
XOR	^ 1001 <sub>2</sub> ^ 1010 <sub>2</sub> = 0011 <sub>2</sub>

David Keil 1/03 4

### Some bitwise problems

Evaluate:

- $8 \gg 1 =$
- $3 \ll 2 =$
- $7 \& 2 =$
- $6 | 3 =$
- $\sim 15 =$
- $9 \wedge 6 =$
- $a \ll b =$
- $c \gg d =$

David Keil 1/03 5

### Setting a bit to 1 with OR

- To set  $n$ th bit of a number, create a mask by shifting a 1 left  $(n - 1)$  times, and then bitwise-OR the number with the mask.
- E.g.,  $0001_2$  OR  $1010_2 = 1011_2$
- So to set 1st (rightmost) bit of 4-bit value ten ( $1010_2$ ), OR it with 0001:

```
cout << (10 | 1);
```

Output: 11

Must use parentheses here

David Keil 1/03 6

**To set the  $n$ th bit of a number**

```
int n,x;
cin >> n >> x;
int mask = 1 << (n - 1);
int result = x | mask;
```

The OR operation makes sure that the mask value's single 1 bit sets to 1 the corresponding bit in the result

David Keil 1/03 7

**Clearing a bit to 0 with AND**

- To clear  $n$ th bit of a number, create a mask by shifting a 1 left  $(n - 1)$  times and complement this; then bitwise-AND the number with the mask
- E.g., (1)  $\sim 0001_2 = 1110_2$   
(2)  $1110_2 \text{ AND } 1011_2 = 1010_2$   
(First bit in 1011 is cleared in 1010)
- To clear first (from right) bit of 4-bit value  $11_{10}$  ( $1011_2$ ), AND it with complement of 1 shifted left 0 places:  
**cout << (11 &  $\sim(1 << 0)$ );**

David Keil 1/03 8

**To clear the  $n$ th bit**

```
int n,x;
cin >> n >> x;
int mask = ~(1 << (n - 1));
int result = x & mask;
```

The AND operation makes sure that the mask value's single 0 bit clears to 0 the corresponding bit in the result

David Keil 1/03 9

**Subtopic: The preprocessor**

- `#include` lets source file access library constants, types, function declarations
- `#define` does text substitution in source:  
`#define TAX 0.05`
- `#define x y` : replaces string  $x$  with  $y$
- `#define LIB_H`: adds identifier  $LIB_H$  to symbol list
- `#ifdef LIB_H` causes source code up to `#endif` to be compiled only if  $LIB_H$  is *defined*
- `#ifndef LIB_H`: causes remaining source code up to `#endif` to be skipped if  $LIB_H$  defined
- `#endif`: completes *ifdef* or *ifndef*.

David Keil 1/03 10

**Preprocessor macros**

- Macros with arguments eliminate function-call overhead; but they are error prone
- Good example:*  
`#define MIN(A,B) ((A) < (B)) ? (A):(B)`
- Bad example:*  
`#define SQUARE(N) N * N`  
Expansion of `SQUARE(2+1)`:  
 $2 + 1 * 2 + 1 = 5$   
Expansion of `(8 / SQUARE(2))`:  
 $8 / 2 * 2 = 8$
- C++ supports inline functions; a safer option

David Keil 1/03 11

**Subtopic: Standard types**

- Integer
- Floating-point
- Character
- String

David Keil 1/03 12

### Small and large integer types

Type	Storage (bytes)	Min. value	Max. value
<i>char</i>	1	0	255
<i>int</i>	4	-2G	+2G
<i>unsigned int</i>	4	0	+4G
<i>short int</i>	2	-32K	+32K
<i>long int</i>	4	-2G	+2G

- Type qualifiers: *short*, *long*, *unsigned*, *const*
- Use the *sizeof* operator; e.g.:  

```
cout << sizeof(char);
```

[*sizeof.cpp*]

David Keil 1/03 13

### Floating-point data

**sign**   **mantissa (fraction)**   **exponent**  
**1 bit**   **23 bits**   **8 bits**

- Data types *float* and *double* represent numbers with possible fraction parts
- The numeral floats because the exponent part compensates for a shift to eliminate 0's on the left
- double* has twice the precision of *float*

David Keil 1/03 14

### Character data

- A character's data type is *char*
- A string is a sequence of characters
- A standard string class is evolving
- The C-language input/output library is *stdio.h*

David Keil 1/03 15

### C-style strings

- String literals use double quotes
- char s[20];* declares a *char* array (string)
- Null character terminates a string
- Strings must be manipulated with libraries such as *string.h*
- Input: *cin >> s;*
- Output: *cout << s;*
- Initialization: *char s[ ] = "Foo";*
- Conversions: *atoi*, *atof* (*stdlib.h*)

David Keil 1/03 16

### Some functions in *string.h*

The standard C-style-string library: *string.h*

*strlen(s)*: returns length of string *s*

*strcpy(s1,s2)*: copies contents of *s2* to *s1*

*strcat(s1,s2)*: concatenates *s2* to end of *s1*

*strcmp(s1,s2)*: returns 0 if *s1* and *s2* are the same; otherwise finds first character location *x* where *s1* and *s2* differ and returns (*s1[x]-s2[x]*)

*strtok(s,ch)*: returns address of first substring of *s* delimited by *ch*

David Keil 1/03 17

### Using *string.h* functions

```
#include <string.h>
void main()
{
    char s1[] = "abc",s2[] = "def",result[80];
    strcpy(result,s1);
    strcat(result,s2);
    cout << "result = " << result << endl
         << "strlen(" << result << ") = "
         << strlen(result) << endl
         << "strcmp(\"" << s1 << "\", \"" << s2 << "\") = "
         << strcmp(s1,s2) << endl;
}
```

**Output:**  
 result = abcdef    strlen(abcdef) = 6  
 strcmp("abc", "def") = -1

David Keil 1/03 18

### *strtok* tokenizes a string

```

void tokenize(char* s)
// Separates and displays components of <s>.
{
  cout << "You entered: " << endl;
  char* tkn = strtok(s, " ");
  while (tkn)
  {
    cout << tkn << endl;
    /* Subsequent calls to <strtok> using a NULL
       parameter 1 returns pointer to the character
       after the token */
    tkn = strtok(NULL, " ");
  }
}
void main()
{
  cout << "Enter words separated by spaces: " << endl;
  char input[80];
  gets(input);
  tokenize(input);
}
    
```

[tokdemo.cpp] David Keil 1/03 19

### Conditional expressions

`int average = (n > 0 ? sum/n : 0);`

- ...has the same meaning as:
 

```

if (n > 0)
  average = sum/n;
else
  average = 0;
            
```
- The conditional operator `?` is *ternary* (has 3 operands)
- The value of the conditional expression is operand 2 or 3, depending on the value of operand 1

David Keil 1/03 20

### Subtopic: Loops and debugging

- Kinds of loops: sentinel controlled, counted, general exit tested
- C/C++ loop statements: top tested (*while*), bottom tested (*do...while*), counted (*for*)
- Readability
- Correctness
- Debugging

David Keil 1/03 21

### Algorithm 1: What does it do, in plain English?

1.  $a \leftarrow -1$
2.  $c \leftarrow 0$
3.  $b \leftarrow 0$
4. Prompt for  $a$
5. If  $a = 0$ , go to step 9
6. If  $a = b$  then  $c \leftarrow 1$ ; go to step 9
7.  $b \leftarrow a$
8. Go to step 4
9. Display  $c$

David Keil 1/03 22

### Algorithm 2

```

input ← -1
match ← false
previous ← 0
Loop until match is true or input is 0:
  Prompt for input
  If input ≠ 0 and input = previous
    match ← true
  previous ← input
If match = true display "Yes";
otherwise display "No"
    
```

David Keil 1/03 23

### Conditions for software correctness

- Program eventually halts under all conditions
- Thus, values that control termination of a loop must *converge*
- For *all* possible inputs or parameter values, a program or function must have a correct result
- By reasoning about the code, we can judge correctness and find bugs

David Keil 1/03 24

## Two kinds of comments

- *Narrative*; tells what happens
- *Assertion*; tells what is true about variable values at an instant of time during execution
- Three useful assertions:
  - Precondition: Tells what is required for the code to be expected to work
  - Postcondition: Tells what is true after code runs (i.e., the spec)
  - Loop invariant: Tells what is true at the top of a loop

David Keil 1/03 25

## Loop-invariant example

```

void main() [max.cpp]
{
    double input, largest;
    cout << "Enter a number (< 0 to exit): ";
    cin >> largest;
    // Precondition: <largest> stores input
    while (input >= 0)
    {
        // Loop invariant: <largest> stores
        // maximum input so far
        cout << "Enter a number (< 0 to exit): ";
        cin >> input;
        if (input > largest)
            largest = input;
    };
    cout << "Largest = " << largest << endl;
    // Postcondition: program has displayed the
    // maximum value entered
}
    
```

David Keil 1/03 26

## Limits of testing

- This program can hang. What is the bug and how many tests would be expected to find it?

```

void main()
{
    unsigned input1, input2;
    srand(&time);
    cin >> input1 >> input2;
    if (rand() == input1 &&
        rand() == input2)
        while(1);
    cout << input1 << " + " << input2
         << " = " << input1 + input2 <<
    endl;
}
    
```

David Keil 1/03 27

## Problems (loops)

1. Suggest C++ loop statements for:
  - (a) input until user enters 0
  - (b) averaging 10 inputs
  - (c) finding logarithm of a number
2. Write a program to find the *second* largest value in a series of inputs terminated by a 0 input
3. Write a program that finds  $\sum_{k=1}^n \frac{1}{k}$  for input  $k$

David Keil 1/03 28

## Errors to watch for

- Uninitialized variables
- Unused variables or parameters
- Poorly indented code
- Variables, functions, or types with vague names (“value”, “process”, “data”...)
- Loops not *provably* terminating
- Loops that iterate once too many or too few times
- Array subscripts not *provably* in bounds

David Keil 1/03 29

## Using *assert* in debugging

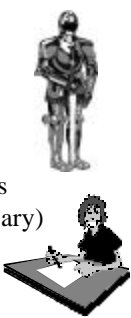
```

#include <assert.h> [div1.cpp]
float quotient(int a, int b);
void main()
{
    cout << "Enter two integers: ";
    int a, b;
    cin >> a >> b;
    cout << a << " / " << b << " = "
         << quotient(a, b);
}
float quotient(int a, int b)
// Returns (a/b). Terminates prog on div-by-0.
{
    assert(b != 0);
    return (float)a / b;
}
    
```

David Keil 1/03 30

## Software quality issues

- User interface must be clear and straightforward
- Robust (“bullet-proof”) software is ready to handle any error condition
- Program should give correct results (including error messages if necessary) on *any* input
- Ensuring quality begins at the specification stage



David Keil 1/03 31

## Subtopic: Subprograms

**Review**

- Defining and declaring functions
- Local variables and scope
- Value and reference parameters
- Return values
- Recursive functions

**New subtopics**

- C-style functions
- Static variables
- Default arguments
- Variable-size parameter lists
- Drivers and stubs
- Command-line arguments

David Keil 1/03 32

## 3 C/C++ language elements

- A *function prototype* (declaration) introduces a function name to the program
- A *function call* invokes the function
- A *function definition* spells out the function’s content
- A function definition has a *header* (type; function ID; parameters in parentheses) and a *body* (compound statement)

David Keil 1/03 33

## When a called function is undefined

- The compiler ensures that identifiers, e.g., function names, are declared before use
- The linker will detect function calls with no corresponding definition

```

#include <iostream.h>
int get_age();
void main()
{
    cout << get_age();
}
    
```

Linker error message:  
“undefined external”

David Keil 1/03 34

## A char function parameter

```

int number_on_dial(char letter)
// Returns number corresponding to <letter>.
{
    switch (tolower(letter))
    {
        case 'a': case 'b': case 'c': return 2;
        case 'd': case 'e': case 'f': return 3;
        case 'g': case 'h': case 'i': return 4;
        case 'j': case 'k': case 'l': return 5;
        case 'm': case 'n': case 'o': return 6;
        case 'p': case 'r': case 's': return 7;
        case 't': case 'u': case 'v': return 8;
        case 'w': case 'x': case 'y': return 9;
        default: return 0;
    }
}
    
```

[telephon.cpp] David Keil 1/03 35

## A double function parameter

```

int rounded(double n);
void main()
{
    double input;
    cout << "Enter a number: ";
    cin >> input;
    cout << input << " rounded off is "
         << rounded(input) << endl;
}
int rounded(double n)
// Returns <n>, rounded off.
{
    return (int)(n + 0.5);
}
    
```

Sample I/O:  
Enter a number: 4.2  
4.2 rounded off is 4

David Keil 1/03 36

## Bar-graph problem

- Write a function that draws a bar graph, taking two parameters, *occupancy* and *capacity*, and displaying ten characters (X's and underscores). The proportion of X's to 10 is the proportion of *capacity* filled by *occupancy*.
- E.g., for parameters (30,100), display:  
 XXX\_\_\_\_\_
- for (20,40), display:  
 XXXXX\_\_\_\_\_

David Keil 1/03 37

## Memory allocation for local data items

```

void main()
{
    int a = 2;
    write(5);
}

void write(int b)
{
    cout << b;
}
    
```

*When function terminates, activation frame is popped from stack*

*This function call pushes an activation frame onto stack*

[actframe.cpp]

David Keil 1/03 38

## Activation records in memory

- Each function call, at run time, causes an activation record to be pushed on top of the stack
- When the function terminates, the activation record is popped and its memory is released
- Activation records contain local variables and function parameters
- A *return* statement pushes the return value on the stack after activation record is popped

David Keil 1/03 39

## Reference parameters: example

```

void add(int, int, int&);

void main()
{
    int sum;
    add(3,4,sum);
    cout << "3 + 4 = " << sum;
}

void add(int a, int b, int& new_sum)
{
    new_sum = a + b;
}
    
```

*Function add changes the value of actual parameter sum*

*Formal reference parameter*

*Reference operator*

*Output:*  
3 + 4 = 7

David Keil 1/03 40

## Reference parameters

- Reference parameters can communicate data *to* and *from* a called function
- Unlike a value parameter, a formal reference parameter is another name for the same data location as the actual parameter
- Actual reference parameter must be a variable

David Keil 1/03 41

## Using a reference parameter

- Factorial of  $n$  is defined as the product of all natural numbers from 1 to  $n$  (e.g.,  $4! = 4 \times 3 \times 2 \times 1$ )

```

void factorial(int n,int& result)
// returns n * (n-1)! as result.
{
    // Precondition: <n> >= 0
    result = 1;
    for (int i=1; i <= n; ++i)
        // Loop invariant: <result> = i !
        result *= i;
    // Postcondition: <result> = n !
}
    
```

[factiter.cpp]

David Keil 1/03 42

## Displaying an address in C

```
int glo1 = 5, glo2 = 1; [intaddr.c]
void main(void)
{
    int local = 2;
    printf("&glo1 = %p &glo2 = %p ",
           &glo1, &glo2);
    printf("\n&local = %p\n", &local);
}
// In C++: cout << &glo1;
```

*Pointer type specifier* (points to `int`)  
*Address operator* (points to `&`)

*Memory addresses (in hexadecimal notation)*

```
Output:
&glo1 = 004165F4 &glo2 = 004165F8
&local = 0064FDF4
```

David Keil 1/03 43

## A string identifier is a pointer

```
char *name = "Bob";
```

*Equivalent code:*

```
(a) char name[] = "Bob";
(b) char name[4];
    strcpy(name, "Bob");
```

*Valid:* `char *p = name;` `cout << p;`

*Invalid:* `char *p;` `strcpy(p, name);`

*Copying to who knows where!*

David Keil 1/03 44

## In C, pass-by-reference requires pointer parameters

```
void factorial(int n, int * result)
// Passes back n * (n-1)! as 'result'
{
    *result = 1;
    for (int i=1; i <= n; ++i)
        *result *= i;
}
// Example of call:
int f;
factorial(5, &f);
printf("5! = %i", f);
```

David Keil 1/03 45

## A string variable is a pointer to char

```
char name[] = "Bob";
char * p = name;
name[0] = 'J';
cout << name;
```

`p[0] = 'J';`  
`cout << p;`

Job

David Keil 1/03 46

## String names are passed as pointers

```
void martha(char s[]);
void main(void)
{
    char name[80] = "George";
    martha(name);
    printf("name=%s\n", name);
}
void martha(char s[])
/* Copies "Martha" to <s>. */
{
    strcpy(s, "Martha");
}
```

*Output:* `name = Martha`

*[arrparam.c]*

David Keil 1/03 47

## Swap problems

- Write a function that swaps two integer parameters.  
*Example:* After `int a=2, b=5`, the statement `swap(a,b)` should leave `a` with the value 5 and `b` with the value 2 in the calling function.
- Is it possible to write two more functions of the same name as above, to swap two floats, two strings?

David Keil 1/03 48

### Kinds of parameters

	Actual	Formal
Value	5	<i>a</i>
Reference	<i>total</i>	<i>sum</i>

```

void main()
{
  int total;
  add(5, 6, total);
  cout << total;
}

void add(int a, int b, int&
sum)
{
  sum = a + b;
}
    
```

David Keil 1/03 49

### Pass streams as reference parameters

```

void save(ofstream& os, char* s)
{
  os << s << endl;
}
    
```

The stream *os* should be a reference because the function might change the state of the stream

David Keil 1/03 50

### A Boolean function

```

bool odd(int n);
void main()
{
  cout << "Enter an integer: ";
  int input;
  cin >> input;
  cout << "odd(" << input << ") = "
  << odd(input) << endl;
}

bool odd(int n)
// Returns true if <n> odd, else false
{
  return (n % 2 == 1);
}
    
```

Enter an integer: 3  
 odd(3) = 1

David Keil 1/03 51

### The recursive factorial function

```

int factorial(unsigned n)
{
  if (n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
    
```

- A recursive function (a) provides a direct solution for a simple base case, or (b) calls itself to reduce a problem to a simpler version of itself

David Keil 1/03 52

### Sum-1-to-*n* problems

- Write a function that returns the sum of all values from 1 to parameter *n*.

$$\sum_{k=1}^n k$$

Argument	Return value
1	1
2	3
3	6
4	10

- Now, write a *recursive* version.

David Keil 1/03 53

### Static variables

- A static variable is initialized once
- Example:

```

int new_ID().
// Returns a unique ID number
{
  static int ID = 0;
  ++ID;
  return ID;
}
    
```

- Keyword *static* is a storage-class qualifier
- Storage duration of *static* variable is execution of program; for automatic variable duration is execution of function

David Keil 1/03 54

### Some features of Kernighan and Ritchie C (1971-1989)

- Variables have to be declared at the start of a block in C
- const* was unavailable
- Parameters were declared between header and body:
 

```
average()
int a;
int b;
{
    return (a+b) / 2;
}
```

*Parameters*

ANSI C:  
**int average(int a, int b)**  
**{...}**
- Default function type: *int*

David Keil 1/03 55

### Default arguments (C++)

A function with a default argument may be called without passing any parameter in call.

```
int prompt_for_int(char prompt[] = "Enter an integer")
// Prompts for and returns int,
// displaying <prompt> or default prompt.
{
    cout << prompt << ": ";
    int input;
    cin >> input;
    return input;
}

void main()
{
    int salary = prompt_for_int("Enter salary"),
        n = prompt_for_int();
}
```

*Sample I/O:*  
**Enter salary: 50000**  
**Enter an integer: 2**

David Keil 1/03 56

### Variable-sized parameter lists

- Ordinarily parameters in call must agree in number and type with prototype
- Using type: *va\_list* defined in library: *stdarg.h*, function may be called with number of parameters determined by programmer
- Macros defined in *stdarg.h*:
  - void *va\_start(va\_list, int)*
  - <type> *va\_arg(va\_list, <type>)*
  - void *va\_end(va\_list)*

David Keil 1/03 57

### A function to average integers

```
double average(int numargs, ...)
{
    double total = 0;
    va_list arglist;
    va_start(arglist, numargs);
    for (int i = 1; i <= numargs; i++)
    {
        int next = va_arg(arglist, double);
        total += next;
    }
    va_end(arglist);
    return total / numargs;
}
```

*Special macros and type supplied by library *stdarg.h*\**

```
cout << average(2, 3, 4);    3
cout << average(6, 4, 3, 3); 4
```

David Keil 1/03 58

### A driver program tests a function

```
int log2(int n);
void main()
{
    cout << "Enter an integer: ";
    int input;
    cin >> input;
    cout << "base-2 logarithm of " << input
        << " is " << log2(input) << endl;
}

int log2(int n)
// Returns floor of base-2 logarithm of <n>.
{
    int count = 0;
    while (n > 1)
    {
        n /= 2;
        ++count;
    }
    return count;
}
```

*The purpose of driver, main, is to test function log2*

[log2demo.cpp]

David Keil 1/03 59

### A driver for factorial

```
void factorial(int n, int& result);
void main()
{
    cout << "Enter an integer: ";
    int input, result;
    cin >> input;
    factorial(input, result);
    cout << input << " != " << result
        << endl;
}
```

[from factiter.cpp]

*Sample I/O:*  
**Enter an integer: 7**  
**7 != 5040**

David Keil 1/03 60

```

void main()
{
  char option;
  do {
    cout << "1 Add" << endl
        << "2 Subtract" << endl
        << "3 Multiply" << endl
        << "4 Divide" << endl
        << "5 Quit" << endl;
    cout << "Choose operation: ";
    cin >> option;
    if (option >= '1' && option <= '4')
    {
      cout << "Enter two integers: ";
      int input1, input2;
      cin >> input1 >> input2;
      switch (option)
      {
        case '1': cout << input1 + input2 << endl; break;
        case '2': cout << input1 - input2 << endl; break;
        case '3': cout << input1 * input2 << endl; break;
        case '4': cout << input1 / input2 << endl; break;
      }
    }
  } while (option != '5');
}

```

**A menu  
presents  
user  
options**

*Menu*

[menu.cpp] David Keil 1/03 61

```

void add() { cout << "Calling <add>" << endl; }
void subtract() { cout << "Calling <subtract>" << endl; }
void multiply() { cout << "Calling <multiply>" << endl; }
void divide() { cout << "Calling <divide>" << endl; }

void main()
{
  char option;
  do {
    cout << "1 Add" << endl << "2 Subtract" << endl
        << "3 Multiply" << endl << "4 Divide" << endl
        << "5 Quit" << endl;
    cin >> option;
    if (option != '5')
    {
      switch (option)
      {
        case '1': add(); break;
        case '2': subtract(); break;
        case '3': multiply(); break;
        case '4': divide(); break;
      }
    }
  } while (option != '5');
}

```

**Stubs test a top-down design**

*Stub function definitions*

*Calls to stub functions*

[stub.cpp] David Keil 1/03 62

## Command-line arguments

- *argc*: how many command parameters
- *argv*: an array of strings
- *argv[1]*: the first command-line argument
- Applications: file names, switches

```

void main(int argc, char **argv)
{
  for (int i=0; i < argc; ++i)
    cout << argv[i] << " ";
}

```

*Output:* [The command line, verbatim]

David Keil 1/03 63