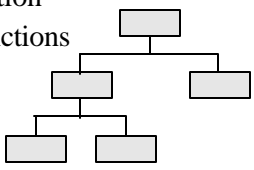


Topic: Subprograms for modularity

- Modular decomposition
- Defining C/C++ functions
- Local variables and scope
- Value parameters
- Reference parameters
- Return values
- Recursive functions



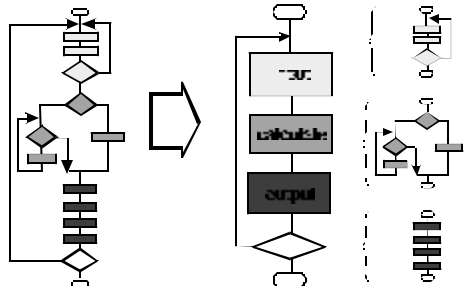
David Keil 7/03 1

Modular decomposition

- Some solutions are too complex to be easily understood as a unit
- A structured design can be decomposed into simpler modules
- This breakdown is called *procedural abstraction*
- We may continue the breakdown as needed by *stepwise refinement*

David Keil 7/03 2

Modular decomposition: case study

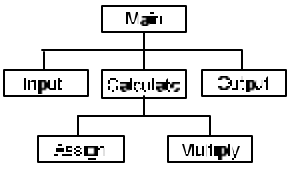


Separate modules are easier to understand.

David Keil 7/03 3

Module hierarchy diagrams

- *Example:*
Main invokes Input, Calculate, and Output;
Calculate calls Assign and Multiply



- A module hierarchy chart shows module dependencies, whereas a flowchart shows order of execution.

David Keil 7/03 4

Writing and calling C++ functions

Example: Drawing a rectangle

```

void horizontal();
void vertical();
} Function prototypes

void main()
{
    horizontal();
    vertical();
    vertical();
    horizontal();
}
} Function calls

void horizontal()
{ cout << "*****" << endl; }
void vertical()
{ cout << "*-----*" << endl; }
} Function definitions
    
```

David Keil 7/03 5

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 7/03 6

Local and global variables

```
#include <iostream.h> [addfunc.cpp]
int quantity = 2;
void add()
// Displays <quantity> doubled.
{
    int sum = 2 * quantity;
    cout << "sum = " << sum << endl;
}
void main()
{
    add();
}
```

*Global variable:
accessible in any function*

*Local variable:
accessible only
in one function*

*Output:
sum = 4*

David Keil 7/03 7

Local variables and scope of access

- A variable declared within a compound statement is visible only there.
- When a function is called, an *activation record* for the call, containing local variables, is placed on top of the stack.
- When the function terminates, the stack is popped and local variables are deallocated.
- Two activation records:

David Keil 7/03 8

Value parameters: example

```
void display_sum(int, int);
void main()
{
    cout << "2 + 5 = ";
    display_sum(2,5);
}
void display_sum(int a, int b)
{
    cout << a + b << endl;
}
```

Actual value parameters

Formal value parameters

*Output:
2 + 5 = 7*

David Keil 7/03 9

Value parameters

- A way for the calling function to pass data to the called function
- Value of *actual parameter* in function call is copied to the *formal parameter* declared in called function's definition
- Formal parameter is local, is deallocated when function terminates
- Prototype must specify parameter types; definition must specify parameter names and types
- Parameters may be of any types and quantity

David Keil 7/03 10

Actual value parameters are unaffected by a function call

```
void add(int, int, int);
void main()
{
    int apsum = 0;
    add(3,4,apsum);
    cout << "apsum = " << apsum;
}
void add(int a, int b,int fpsum)
{
    fpsum = a + b;
    cout << "fpsum = " << fpsum << endl;
}
```

actual parameter

formal parameter

*Output:
fpsum = 7
apsum = 0*

[valparm.cpp]

David Keil 7/03 11

Value parameters act like local variables

```
void display_sum(int a, int b)
// Displays (a + b)
{
    cout << a << "+" << b << "=";
    while (b-- > 0) a++;
    cout << a << endl;
}
void main()
{
    int a, b;
    cout << "Enter two numbers";
    cin >> a >> b;
    display_sum(a, b);
    display_sum(6, 3);
}
```

David Keil 7/03 12

Reference parameters: example

```
void add(int, int, int&);

void main()
{
    int result = 0;
    add(3,4,result);
    cout << "3 + 4 = " << result;
}

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

Function add changes the value of actual parameter result

Formal reference parameter

Reference operator

Output: 3 + 4 = 7

David Keil 7/03 13

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 7/03 14

Repetitive code and functions

```
float a,b,c;
cout << "Input 3 numbers: ";
cin >> a >> b >> c;
if (a > c) {
    float temp = a;
    a = c;
    c = temp;
}
if (a > b) {
    float temp = a;
    a = b;
    b = temp;
}
if (b > c) {
    float temp = b;
    b = c;
    c = temp;
}
cout << "You input " << a << ", " << b << ", " << c << endl;
```

A swap function can simplify the code at left to:

```
void swap(float& s, float& t)
{
    float temp = s;
    s = t;
    t = temp;
}
```

Sample I/O:
Input 3 numbers: 3 2 1
You input 1, 2, 3

[sort3.cpp]

Function return values: example

```
int input_age();
void main()
{
    int age = input_age();
    cout << "You are " << age << " years old" << endl;
}

int input_age()
{
    cout << "Your age? ";
    int input;
    cin >> input;
    return input;
}
```

The value of the function call is the value used in the return statement in the function defn.

Sample I/O:
Your age: 20
You are 20 years old.

7/03 16

A return value passes data back to the calling function

- The function call is an expression
- The *return* keyword precedes the returned value in the called function
- The return value's type must be type compatible with the function's type, declared in header
- The *return* statement terminates the function call
- The returned value goes on the stack for retrieval by the calling function

David Keil 7/03 17

sum with return value

```
int sum(int a, int b);

void main()
{
    int past_due, current;
    cout << "Enter past due and current: ";
    cin >> past_due >> current;
    cout << "You owe " << sum(past_due, current);
}

int sum(int a, int b)
{
    return a + b;
}
```

David Keil 7/03 18

Parameter and return-value types

Some examples:

```
float sum(float a, float b)
{
    return a + b;
}

bool is_even(int n)
{
    return (n % 2 == 0);
}

char nth_char(char s[], int n)
{
    return s[n];
}
```

David Keil 7/03 19

When a function is not defined

- 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>
intxget_age();
voidxmain()
{
    cout << get_age();
}
intxgetage()
{ int a; cin >> a; return a; }
```

Linker error message:
"undefined external"
because of misspelling

David Keil 7/03 20

Where a function is appropriate

```
void main()
{
    cout << "Enter an integer: ";
    int input;
    cin >> input;
    int result = 1;
    result = result * (input + 1);
    result = result * (input + 1);
    result = result * (input + 1);
    cout << "(" << input
         << " + 1) cubed = "
         << result << endl;
}
```

[cube.cpp]

Problem: modularize this program to avoid duplicate code.

David Keil 7/03 21

Stubs test a top-down design

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

Sub function definitions

Calls to stub functions

[stub.cpp]

David Keil 7/03 22

Recursion implements a loop

```
int input_age()
// Prompts for, returns age,
// repeats until gets valid input.
{
    cout << "Age? ";
    int age;
    cin >> age;
    if (age >= 0) return age;
    else return input_age();
}
```

Recursive function call

- A function that calls itself is recursive
- A base case (as, age >= 0) triggers a simple result; a recursive case triggers a recursive call
- No base case means infinite recursion

David Keil 7/03 23

A recursive function to add

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

```
int sum(int a, int b)
// Returns a + b. Recursive.
{
    if (a == 0)
        return b;
    else
        return sum(a-1, b+1);
}
```

David Keil 7/03 24

Recursion uses the stack

```

void backwards();
void main()
{
    backwards();
}

void backwards()
{
    char ch;
    cin.get(ch);
    if (ch != '\n')
        backwards();
    cout << ch;
}

```

[backwardcpp]

Sample I/O:
Hello
 olleH

Questions:
 How many *char* variables can store how much data? How?

David Keil 7/03 25

Guidelines for writing subprograms

- A module has a single purpose
- Its purpose is documented in a comment at the top
- Code longer than a page usually should be broken down
- Experienced programmers avoid side effects (e.g., output, modification of global variables)

David Keil 7/03 26

Discussion problems

1. Write a function that returns the absolute value of the difference between two integers.
2. Convert pseudocode in Topic-7 slide, "Invariants assure correctness", to a function that prompts for 5 numbers and displays them.
3. Convert pseudocode in Topic-7 slide, "Convergence assures termination," to a function that returns the floor of the base-2 logarithm of a number.

David Keil 7/03 27