6. C++ Subprograms

**Topic: Subprograms for modularity**
- Modular decomposition
- Defining C/C++ functions
- Local variables and scope
- Value parameters
- Reference parameters
- Return values
- Recursive functions

**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*

**Modular decomposition: case study**
- Separate modules are easier to understand.

**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.

**Writing and calling C++ functions**
*Example:* Drawing a rectangle

```cpp
void horizontal() ;
void vertical() ;
void main() {
    horizontal();
    vertical();
    horizontal();
}
void horizontal() { cout << "*******" << endl; }
void vertical() { cout << "*-----*" << endl; }
```

**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)
Local and global variables

```cpp
#include <iostream.h>

int quantity = 2;

void add()
// Displays <quantity> doubled.
{
    int sum = 2 * quantity;
    cout << "sum = " << sum << endl;
}

void main()
{
    add();
}
```

Output:
```
sum = 4
```

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:

Value parameters: example

```cpp
void display_sum(int a, int b)
{
    cout << a + b << endl;
}

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

Output:
```
2 + 5 = 7
```

Actual value parameters are unaffected by a function call

```cpp
void add(int a, int b, int fpsum)
{
    fpsum = a + b;
    cout << "fpsum = " << fpsum << endl;
}
```

Value parameters act like local variables

```cpp
void display_sum(int a, int b)
// Displays (a + b)
{
    cout << a << "+" << b << "=";
    while (b-- > 0) a++;
    cout << a << endl;
}
```

Output:
```
fsum = 7
```

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
Reference parameters: example

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

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

Output: 3 + 4 = 7

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

Repetitive code and functions

```c++
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;
```

Function return values: example

```c++
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;
}
```

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

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

```c++
int sum(int a, int b);
```
Parameter and return-value types

Some examples:

```cpp
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];
}
```

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

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

Where a function is appropriate

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

Problem: modularize this program to avoid duplicate code.

Recursion implements a loop

```cpp
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();
}
```

A recursive function to add

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

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

Sample I/O:
Hello
olleH

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

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)

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.