CSIS/COMP 1117B
Computer Programming

Functions

Functions

- Benefits of using functions
- Reference parameter
- Scope of a variable
- Recursion
- Storage management

Benefits of Using Functions

- Enhances readability of programs
  - provides a mechanism for structuring programs
  - each function performs a small set of well-defined operations
- Facilitates re-usability
  - the same function can be invoked from different parts of a program
  - can be used in other programs when packaged into a library

Another Prime Factors Printing Program

- For debugging purposes, it would more useful if our main program keeps processing input until told to stop:

```
#include <iostream>
using namespace std;
int  n = 2;
bool valid_input();            // declaration of external function, heading only
void print_factors(int m);  // compiler can check that they are called properly
int main() {
    while (1) {
        if ( valid_input() ) {          // ensure input is valid before continuing
            cout << n << " = ";       // prints out integer to be factored
            print_factors(n);          // prints out prime factors
            cout << endl;              // cleans up output
        } else if (n == 0) break;   // stop program on a 0 input
        else cout << "Input invalid for program, a number > 1 is required\n";
    }
}
```

Using External Functions

- In the previous example, the main program will reside in a file and the external functions in another file (and possibly each in its own file).
- Once the API of the functions is ***stable***, we can write the main program which will call the yet-to-be developed functions and compile the program to obtain an ***object module*** for main.
- When we do the functional decomposition of the program, we should also decide on the grouping of related functions into separate source files.
- Initially, except for the main program, all files contain skeleton function declarations and all files are compiled to generate object modules.
- Development proceeds by working on one source file at a time, which is then compiled to generate a new object module.
- An executable program is obtained by linking all the object modules together and can be tested on the specific functions that are currently being implemented.

Unfortunately . . .

- The new version of the prime factor printing program does not work!
- valid_input is supposed to input a value for a variable n defined outside of it ***but in the same file***. (It is possible to make a variable common to several source files but we have a better solution.)
- The problem can be solved if it is possible for valid_input to actually store a value into n that is to be passed as a parameter.

Reference Parameter

- Recall that in our prime factor printing program, we actually prefer the function valid_input to return the input value through a parameter rather than updating a global variable.
- That would make the program more reusable (because we can use it to input value to a variable that is passed as parameter instead of to a specific global variable).
- In C++, we can define a parameter to be a reference parameter if the function will update the **argument** (also called **actual parameter**).
- Such kind of parameters is called **reference parameters** and the actual parameter must be a variable.

Making valid_input to Work

- The valid_input function can be re-written to accept a reference parameter which will receive the input:

```
bool valid_input(int &n) {
    cin >> in;
    return n > 1;
}
```

- Note that the only change is the addition of a formal parameter with an & before the name of the formal. The additional & indicates that the parameter is a reference parameter. Without it, a parameter will be a value parameter (which is the default).
- Also, the updated valid_input will have to be called with a ***variable as actual***, i.e., valid_input(n);

The Example Again

```
#include <iostream>
using namespace std;
int  n = 2;
bool valid_input(int &n);   // declaration of external function, heading only
void print_factors(int m);  // compiler can check that they are called properly
int main() {
    while (1) {
        if ( valid_input(n) ) {          // ensure input is valid before continuing
            cout << n << " = ";       // prints out integer to be factored
            print_factors(n);          // prints out prime factors
            cout << endl;              // cleans up output
        } else if (n == 0) break;   // stop program on a 0 input
        else cout << "Input invalid for program, a number > 1 is required\n";
    }
}
```

Function Declaration Updated

- The updated syntax of a function declaration:

  <funct_decl> → <head> <body>

  <head> → [<type>] <name> ‘(‘ [<formal_list>] ‘)’

  <formal_list> → <formal> { ‘,’ <formal> }

  <formal> → <type> [‘&’] <identifier>

  <type> → <identifier>

  <name> → <identifier>

  <body> → <compound_stmt>

- Compound statement:

  <compound_stmt> → ‘{‘ { <statement> | <var_decl> } ‘}’

Things You Already Know

- Return type: void, int, char, bool, (*and more to follow*)*.*
- Formal parameter list: type and name of parameters, may be empty.
- The return statement for returning from a function together with a value if one is expected.
- Function invocation: actual parameters are supplied in the order defined in the function heading; variables are required for reference parameters and expressions for non-reference (value) parameters.
- Function invocation can be part of an expression or exist by itself as a statement.

Scope of A Variable

- The **scope** of a variable is the portion of a program where this variable is ***visible***.
- The scope starts from the declaration of a variable and ends at the end of the **block** (compound statement) containing the declaration.
  - note that a variable declaration can appear in the middle of a block!
- Re-declaration of a variable in the same block is ***not*** allowed
  - functions and global variables also need to have unique names
- Scope is a ***static*** concept, we use it to determine the declaration that give meaning to a certain identifier in a program.

Now The Confusing Part

- If another variable is declared in the inner block of the scope of a variable of the ***same*** name, the accessibility of the first variable is ***blocked*** in the scope of the second variable.
- For example, n can be used to name the (only) parameter used in print_factors; m is used to avoid any confusion that might be caused! If n is used to name the parameter in print_factors, the global variable n will not be accessible in the body of print_factor.
- Formal parameters are treated as variables local to a function and their scopes is the body of a function; consequently, names of local variables should be different from those of the formals.
- Non-nested blocks can have local variables or formal parameters having the same name and not causing any conflict. For example, both valid_input and print_factors can use n to name their sole parameter without causing any problem.

Recursion

- In mathematics, many definitions are recursive; for example, factorial is typically defined as:

  $n! = n \times (n - 1)!$
- In programming, a function is said to be **recursive** if it calls itself directly (i.e., there is at least one call to the function itself in the body of the function) or indirectly (if the function calls another function and this chain of calls eventually arrives at a call to the first function.
- In general, to solve a problem recursively, the solution typically is in the form of:
  - solve a small part of the original problem
  - the remaining problem is simpler than the original problem and is ***similar*** to the original problem and hence can be solved by the same solution!

Factorial Function

- The factorial function:

```
// factorial:  computes n!
int factorial (int n) {
    if (n > 1)  // calculation is needed only for n > 1
        return n * factorial (n – 1);
    else return 1;  // last 1 in the chain of multiplications
}
```

- Compare this with the iterative version:

```
int factorial (int n) {
    int  t;
    for (t = 1; n > 1; n--) t = t * n;
    return t;
}
```

GCD Again

- Our previous program for GCD is quite inefficient because it actually performs division using successive subtractions.
- The standard mathematical definition of GCD is

  for m, n > 0, GCD(m, n) = n, if n = m

  and GCD(m, n) = GCD(m mod n, n), if m > n

  and GCD(m, n) = GCD(n mod m, m), if n > m
- The recursive version:

```
int gcd (int m, int n) {
    // body same as the definition of GCD above!
    if (m == n) return m;
    if (m > n) return gcd(m % n, n);
    return gcd(m, n % m);
}
```

Applying Recursion to print_factors

- If the prime divides the given number, after outputting prime, we are left with the problem: print the prime factors of the quotient of the above division.
- If the prime does not divide the given number, we try a new prime on the given number.
- Recursion for print_factors is only applied to handling factors that are odd numbers and that part of operation is packaged into a recursive function print_prime

The New print_factors

```
void print_prime (int p, int n) {
    if ( n > 1 )  // check for termination, note method works only for n > 1
        if (n % p == 0) {  // found a factor
            cout << p;
            if ((n = n / p) > 1) cout << " X ";
            print_prime(p, n);
        }
        else if (p < sqrt(n))   // check for termination again
             print_prime(p + 2, n);
        else cout << n;         // output last factor
}
void print_factors (int n) {
      while (n % 2 == 0) {  // make sure that m is not divisible by 2
          cout << '2';
          if ((n = n / 2) <= 1) return;
          cout << " X ";
      }
      print_prime(3, n);
}
```

General Structure of Recursive Functions

- The body of a recursive function ***always*** starts with a test on termination and there may be further tests for termination in the body.
- Operations typically involve performing some simple operations which only partially solves the original problem, e.g., printing only 1 factor.
- The remaining work will be handled by recursively calling the function with ***simplified*** arguments; hence progressively less work will have to be done on the original problem on each subsequent call and eventually the entire problem is solved.
- In the case of print_prime, either p is increased or n reduced so that one of the termination conditions will become true eventually.

Storage Management

- When an executable program is loaded into memory for execution, three separate pieces of main memory will be allocated to that program:
  - a **code segment** for storing the program instructions
  - a **data segment** for the global variables
  - a **stack segment** for local variables and other information associated with function invocations
- *There is actually a fourth data segment for dynamic data (which will be explained later in the course).*

More on Stacks

- A **stack** maintains a number of storage cells in a last-in-first-out manner, i.e., the next data to be pulled from a stack will be the one that is deposited into it in the most recent past.
- Such a allocation/de-allocation behaviour suits the storage requirements of a chain of function calls/returns nicely – the function that is to be return will be the one that is called most recently.
- Technically, a stack is a storage mechanism such that:
  - only the **top** of a stack is accessible at any one time
  - adding a new element at the top of a stack is call **push**
  - removing an element from the top of a stack is called **pop**

Activation Record

- Each time when a function is called, some additional storage will need to be allocated to it even when it does not have any parameters nor local variables; we need to remember the address of the point of call (also called **return address**) so that we can resume program execution at this point when the function returns.
- Note that the return address is **not static** since a function can be called from different parts of a program at different time.
- Additional storage will have to be allocated for parameters and local variables if required.
- The storage that is allocated to a function for its parameters, local variables, and other information such as the return address, is called an **activation record**.

Function Call/Return

- When a function is called, its activation record is pushed onto the stack segment. Expressions representing the actual parameters are evaluated and theirs values are used to initialize the (formal) parameters (which resides in the activation record).
- Reference parameters will be initialized to the addresses of the variables that are passed as actual parameter. Hence, the function can access the actual parameters through these addresses.
- Upon return, the activation record will be popped from the stack segment and the return address used to resume the execution of the ***caller***.

Recursive Call

- When a function calls itself, a separate activation record will be allocated to this new call; hence, even though these two functions execute the same area of the code segment, they nonetheless have their own copies of their local variables and parameters.
- Their execution is basically *independent* of each other and is exactly the same as if the function calls a different function.
- Imagine that a recursive function being two separate functions which happen to use the same names for their local variables and the parameters but nonetheless are distinct (and are allowed by the scope rules).