Storage Classes, Scope, and Recursion

Lecture 6

3.10Storage Classes

- Variables attributes
  - Name, type, size, value
  - Storage class
    - How long variable exists in memory (lifetime)
  - Scope
    - Where variable can be referenced in program
      - Refers to code blocks
  - Linkage
    - For multiple-file program (see Ch. 6), which files can use it

3.10Storage Classes

- Static storage class
  - Variables exist for entire program
    - For functions, name exists for entire program
  - May not be accessible, scope rules still apply (more later)
- `static` keyword
  - Local variables in function
  - Keeps value between function calls
  - Only known in own function

3.11Scope Rules

- Scope
  - Portion of program where identifier can be used
- File scope
  - Defined outside a function, known in all functions
  - Global variables, function definitions and prototypes
- Function scope
  - Can only be referenced inside defining function
  - Only labels, e.g., identifiers with a colon (`case:`)

3.11Scope Rules

- Block scope
  - Begins at declaration, ends at right brace **}**
    - Can only be referenced in this range
  - Local variables, function parameters
  - `static` variables still have block scope
    - Storage class separate from scope
- Function-prototype scope
  - Parameter list of prototype
  - Names in prototype optional
    - Compiler ignores
  - In a single prototype, name can be used once

fig03_12.cpp (1 of 5)

```cpp
// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

using std::cout;
using std::endl;

void useLocal( void );       // function prototype
void useStaticLocal( void );  // function prototype
void useGlobal( void );      // function prototype

int x = 1;     // global variable

int main()
{
  int x = 5;   // local variable to main

  cout << "local x in main's outer scope is " << x << endl;

  { // start new scope

    int x = 7;

    cout << "local x in main's inner scope is " << x << endl;

  } // end new scope
```

fig03_12.cpp (2 of 5)

```

   cout << "local x in main's outer scope is " << x << endl;

   useLocal();       // useLocal has local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal();      // useGlobal uses global x
   useLocal();       // useLocal reinitializes its local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal();      // global x also retains its value

   cout << "\nlocal x in main is " << x << endl;

   return 0;   // indicates successful termination

} // end main

```

fig03_12.cpp (3 of 5)

```
// useLocal reinitializes local variable x during each call
void useLocal( void )
{
   int x = 25;  // initialized each time useLocal is called

   cout << endl << "local x is " << x
        << " on entering useLocal" << endl;
   ++x;
   cout << "local x is " << x
        << " on exiting useLocal" << endl;

} // end function useLocal

```

fig03_12.cpp (4 of 5)

```
// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal( void )
{
  // initialized only first time useStaticLocal is called
  static int x = 50;

  cout << endl << "local static x is " << x
     << " on entering useStaticLocal" << endl;
  ++x;
  cout << "local static x is " << x
     << " on exiting useStaticLocal" << endl;

} // end function useStaticLocal

```

(5 of 5)

```
// useGlobal modifies global variable x during each call
void useGlobal( void )
{
  cout << endl << "global x is " << x
     << " on entering useGlobal" << endl;
  x *= 10;
  cout << "global x is " << x
     << " on exiting useGlobal" << endl;

} // end function useGlobal
```

fig03_12.cpp output (1 of 2)

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
```

fig03_12.cpp output (2 of 2)

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Recursion

- Problem solving approach
  - Repeatedly break problem into smaller versions of the same problem
  - Goal: reduce problems to trivial size
  - Repeatedly combine solutions to subproblems
- Implemented by using functions that call itself on smaller problems

Definitions

- Recursive call
  - A function call in which the function being called is the same as the one making the call
- Direct recursion
  - When the function directly calls itself
  - This is the type of recursion we will study

Pros of Recursion

- Powerful programming technique to solve problems
  - Difficult to solve some problems without recursion
  - Problems can have simple, elegant recursive solutions

Classic Recursive Example

- Consider *n*! (*n* factorial)
  - The number of permutations of *n* elements
  - A mathematical description

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n*(n-1)*\ldots*1, & \text{if } n > 0 \end{cases}$$

- Consider 4!
  - 4! = 4 * 3 * 2 * 1 = 24
  - Notice 3! = 3 * 2 *1
    - part of 4!

Using Recursion

- Recursive definition of n!

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n*(n-1)! & \text{if } n > 0 \end{cases}$$

- Solve 4!
  - 4! = 4 * 3!
    - 3! = 3 * (3-1)! = 3 * 2!
      - 2! = 2 * (2-1)! = 2 * 1!
        - 1! = 1 * (1-1)! = 1 * 0!
        - o from definition 0! = 1

$$\begin{tikzcd}
4*6 = 24 \\
3*2 \text{ Report } 6 \arrow[u, bend right] \\
2*1 \text{ Report } 2 \arrow[u, bend right] \\
1*1 \text{ Report } 1 \arrow[u, bend right] \\
\text{Report } 1 \arrow[u, bend right]
\end{tikzcd}$$

Notice recursion **stops** when we reach a case for which we know the answer

Recursive Framework

- Base case
  - The case for which the solution can be stated nonrecursively
- General (recursive) case
  - The case for which the solution is expressed in terms of a smaller version of itself
- Recursive algorithm
  - A solution that is expressed in terms of
    1. Smaller instances of itself
    2. A base case

Coding Factorial Function

```
int factorial (int number) {
  if (number == 0)    // base case
    return 1;
  else      // General case
    return number * factorial(number -1);
}
```

- Notice the general case involves the recursive call

Verifying Recursive Functions

- Determine inductively whether a recursive algorithm works
- The Three-Question Method
  - Must answer yes to all questions
  1. The Base-Case Question
     - Is there a nonrecursive way out of the function and does the routine work correctly for this base case?
  2. The Smaller-Caller Question
     - Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
  3. The General-Case Question
     - Assuming that the recursive call(s) works correctly, does the entire function work correctly?

Writing Recursive Functions

1. Get an exact definition of the problem to be solved
2. Determine the *size* of the problem to be solved on this call to the function. On the initial call to the function, the size of the whole problem is expressed in the value(s) of the parameters(s)
3. Identify and solve the base case(s) in which the problem can be expressed nonrecursively.
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem – a recursive call.

Multiplying Rabbits (The Fibonacci Sequence)

- “Facts” about rabbits
  - Rabbits never die
  - A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  - At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair

How many pairs of rabbits are alive in month n?

- Write a function that finds the nth number in the Fibonacci Series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
    - `fibonacci(0) =`
    - `fibonacci(1) =`
- Function

```
int fibonacci( int n ) {




}
```

Writing Recursive Functions

1. Get an exact definition of the problem to be solved
2. Determine the *size* of the problem to be solved on this call to the function. On the initial call to the function, the size of the whole problem is expressed in the value(s) of the parameters(s)
3. Identify and solve the base case(s) in which the problem can be expressed nonrecursively.
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem – a recursive call.

Verify `fibonacci`

- The Three-Question Method
  1. The Base-Case Question
     - Is there a nonrecursive way out of the function and does the routine work correctly for this base case?
  2. The Smaller-Caller Question
     - Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
  3. The General-Case Question
     - Assuming that the recursive call(s) works correctly, does the entire function work correctly?

Evaluating code

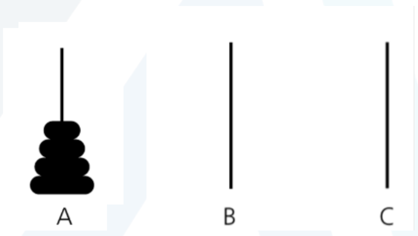
```
int main() {
 cout << myst(4, 2) << endl;
}
int myst(int i, int j)
 if (i == j)
    return 1;
 else if (j == 0)
    return 1;
 else
    return myst(i-1, j-1) + myst(i-1, j);
}
```

3.14Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

Tower of Hanoi

- Problem: Move disks of different sizes from one beg to another
  - Never have a larger disk on top of a smaller disk on top of a smaller disk
  - Have 3 pegs to work with



The Towers of Hanoi