CC-112 Programming Fundamentals

Recursion

Nazar Khan
Department of Computer Science
University of the Punjab

Recursion

- A *recursive function* is a function that calls itself either directly or indirectly.
- If a recursive function is called with a *base case*, the function simply returns a result.
- If it's called with a more complex problem, the function divides the problem into two conceptual pieces:
  1. a piece that the function knows how to do, and
  2. a slightly smaller version of the original problem.
- Because this new problem looks like the original problem, the function launches a recursive call to work on the smaller problem.

Recursion

- For recursion to terminate, each recursive call is a slightly simpler version of the original problem.
- The sequence of smaller and smaller problems must converge on the base case.
- When the function recognizes the base case, the result is returned to the previous function call, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result.
- Recall the `factorial` function from Assignment 1.

```
unsigned long long int factorial = 1;
for (int counter = number; counter >= 1; --counter)
  factorial *= counter;
```

Recursive version of factorial (!)

```
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(unsigned int number);

int main(void)
{
  // during each iteration, calculate
  // factorial(i) and display result
  for (unsigned int i = 0; i <= 22; ++i) {
    printf("%u! = %llu\n", i, factorial(i));
  }
}

// recursive definition of function factorial
unsigned long long int factorial(unsigned int number)
{
  // base case
  if (number <= 1) {
  return 1;
  }
  else { // recursive step
    return (number * factorial(number - 1));
  }
}
```

Weakness of C

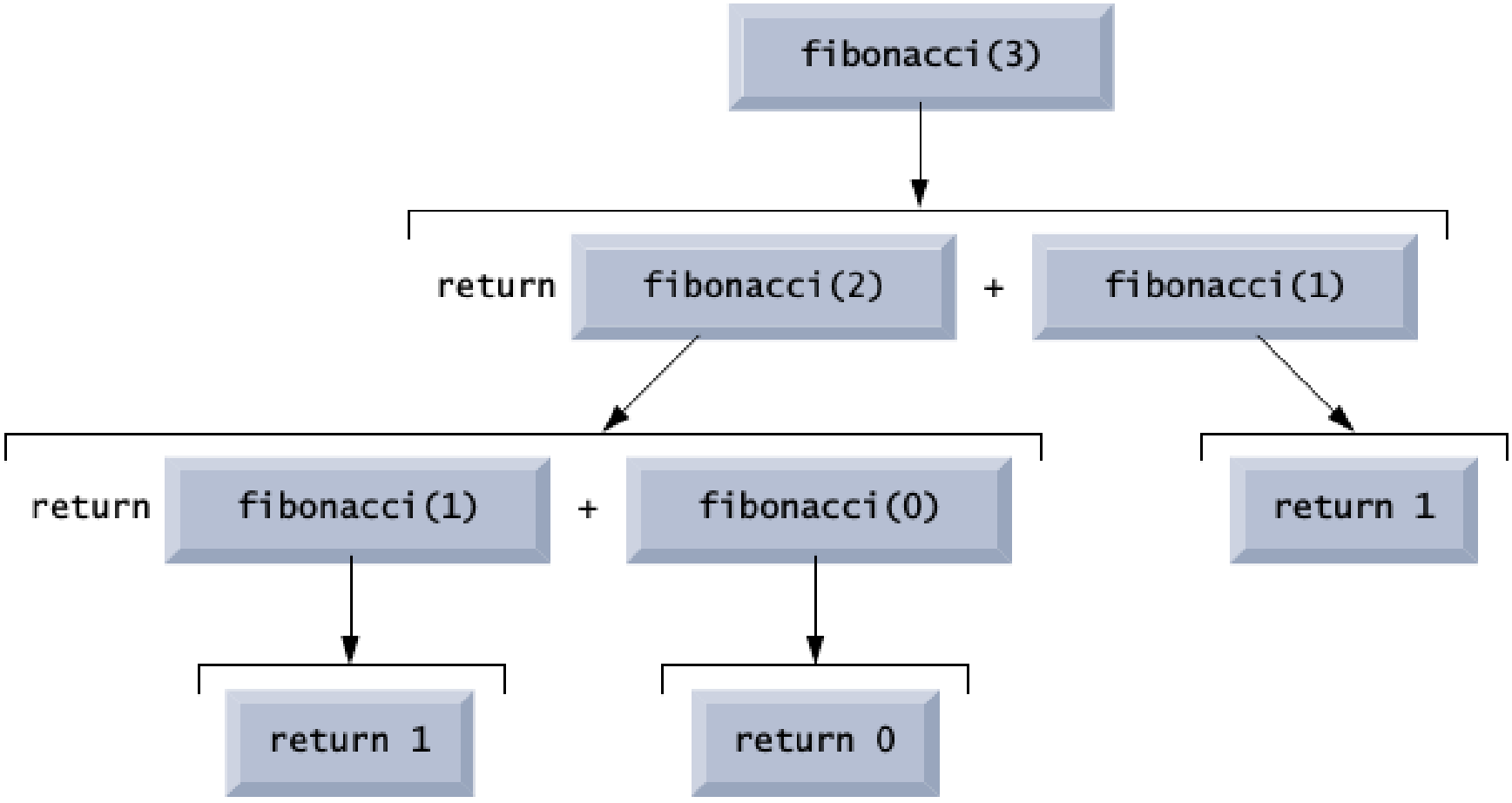
Even when we use `unsigned long long int`, we still can't calculate factorials beyond 21! This points to a weakness in C (and most other procedural programming languages) – namely that the language is *not easily extended* to handle the unique requirements of various applications.

C++ is an *extensible language* that, through "classes", allows us to create new data types, including ones that could hold arbitrarily large integers if we wish.

Example Using Recursion: Fibonacci Series

```c
// Recursive fibonacci function
#include <stdio.h>

unsigned long long int fibonacci(unsigned int n); // function prototype

int main(void)
{
  unsigned int number; // number input by user

  // obtain integer from user
  printf("%s", "Enter an integer: ");
  scanf("%u", &number);

  // calculate fibonacci value for number input by user
  unsigned long long int result = fibonacci(number);

  // display result
  printf("Fibonacci(%u) = %llu\n", number, result);
}

// Recursive definition of function fibonacci
unsigned long long int fibonacci(unsigned int n)
{
  // base case
  if (0 == n || 1 == n) {
    return n;
  }
  else { // recursive step
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Recursive Call Tree



Recursion vs. Iteration

- Both iteration and recursion are based on a control structure.
  - Iteration uses an iteration statement
  - Recursion uses a selection statement.
- Both iteration and recursion involve repetition.
  - Iteration uses an iteration statement.
  - Recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test.
  - Iteration terminates when the loop-continuation condition fails.
  - Recursion terminates when a base case is recognized.
- Iteration and recursion can occur infinitely.
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false.
  - Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.

Recursion vs. Iteration

- Recursion suffers from the overhead of repeated function calls.
- This can be expensive in both processor time and memory space.
- It should be implemented *intelligently*.