DASF004
Basic and Practice in Programming

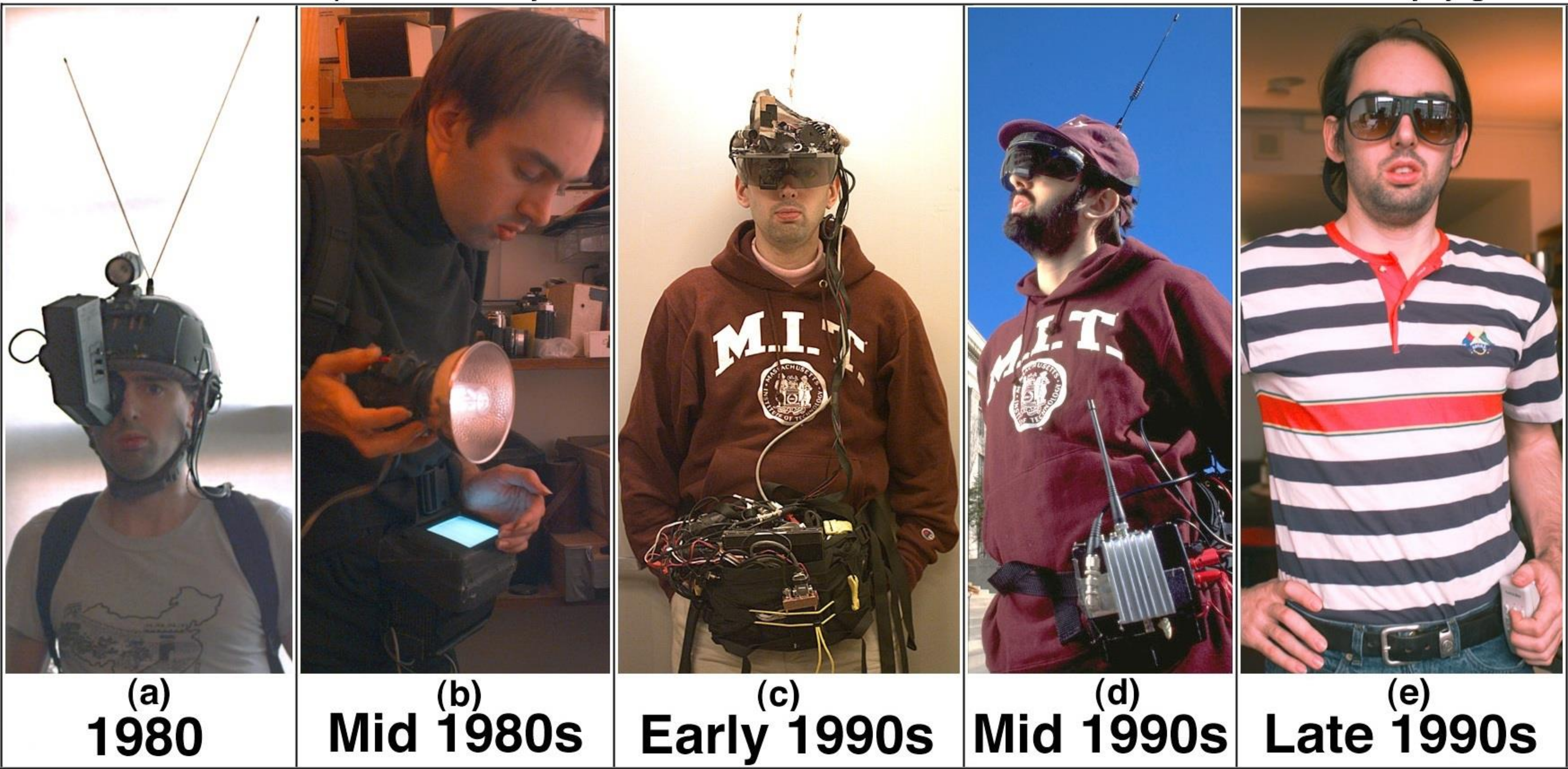
Lecture 5

Function 2

Food for your MIND!

- Augmented Cognition
  – Using technologies to enhance human capability
- Cyborg (<u>Cyb</u>ernetic <u>Org</u>anism) 사이보그
  – Integration of Artificial Element into Organism
- Iron man?

Wearable Computing


Steve Mann's "wearable computer" and "reality mediator" inventions of the 1970s have evolved into what looks like ordinary eyeglasses.

Food for your MIND

- Cyborg Research and Prosthesis Research
- Robotic and Control

What if you have three hands?

Another project at Georgia Tech University

https://www.youtube.com/watch?v=fKryPingtww

Agenda

- Function
- Random Variable
- Argument Coercion
- Recursive Function

Function (review)

- Code reuse
- Example:

```
#include <stdio.h>

int sum4(int w,int x,int y,int z);

int main (void)
{ int a=3;
  int b=6;
  int c=13;
  int d=492;
  int result = sum4(a,b,c,d)
  printf("Result: %d",result)
}

int sum4(int w, int x, int y, int z)
{ int result = w+x+y+z;
  return result;
}
```

Function (review)

- Code reuse
- Example:

```
#include <stdio.h>

int sum4(int w,int x,int y,int z);

int main (void)
{ int a=3;
  int b=6;
  int c=13;
  int d=492;
  int result = sum4(a,b,c,d)
  printf(“Result: %d”,result)
}

int sum4(int w, int x, int y, int z)
{ int result = w+x+y+z;
  return result;
}
```

Function (review)

- Code reuse
- Example:

```
#include <stdio.h>

int sum4(int w, int x, int y, int z)
{ int result = w+x+y+z;
  return result;
}

int main (void)
{ int a=3;
  int b=6;
  int c=13;
  int d=492;
  int result = sum4(a,b,c,d)
  printf("Result: %d",result)
}
```

scanf() Function

- Consider this code segment:
- When user enter "A", the loop

Executed two times.

Reason:

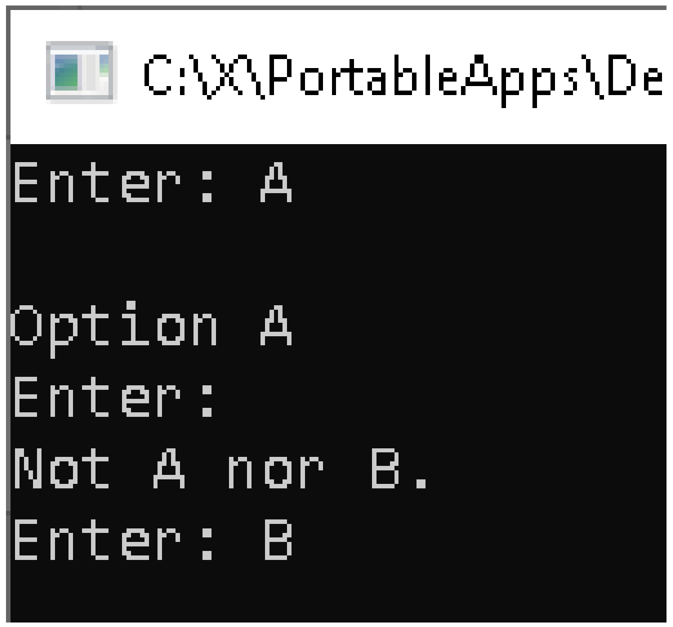
- When "A" is entered, a new line

  character is also entered.

```c
#include <stdio.h>

int main(void)
{    char option;

     for(;;)
     {    printf("Enter: ");
          scanf("%c",&option);
          if(option == 'A')
               printf("\nOption A\n");
          else if(option == 'B')
               printf("\nOption B\n");
          else
               printf("\nNot A nor B.\n");
     }
}
```



C:\X\PortableApps\De

```
Enter: A

Option A
Enter:
Not A nor B.
Enter: B
```

scanf() Function

- Consider this code segment:
- When user enter “A”, the loop

Executed two times.

Reason:

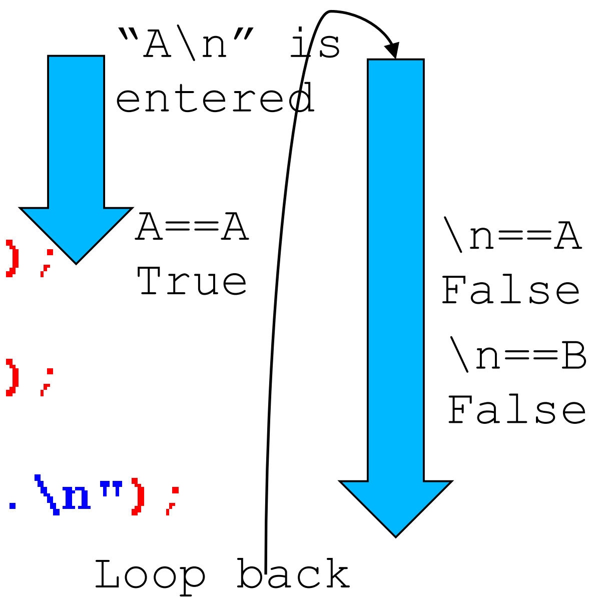
- When “A” is entered, a new lir

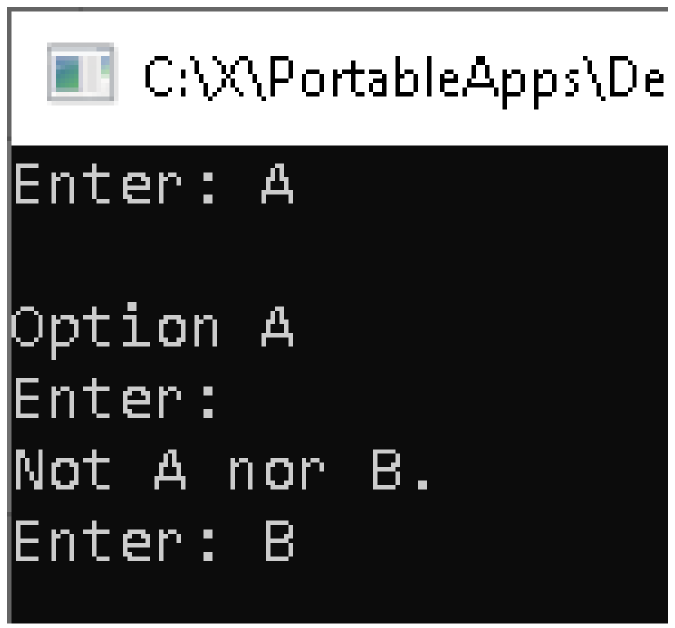
  character is also entered.

```
#include <stdio.h>

int  main(void)
{    char option;

     for(;;)
     {    printf("Enter: ");
          scanf("%c",&option);
          if(option == 'A')
               printf("\nOption A\n");
          else if(option == 'B')
               printf("\nOption B\n");
          else
               printf("\nNot A nor B.\n");
     }
}
```



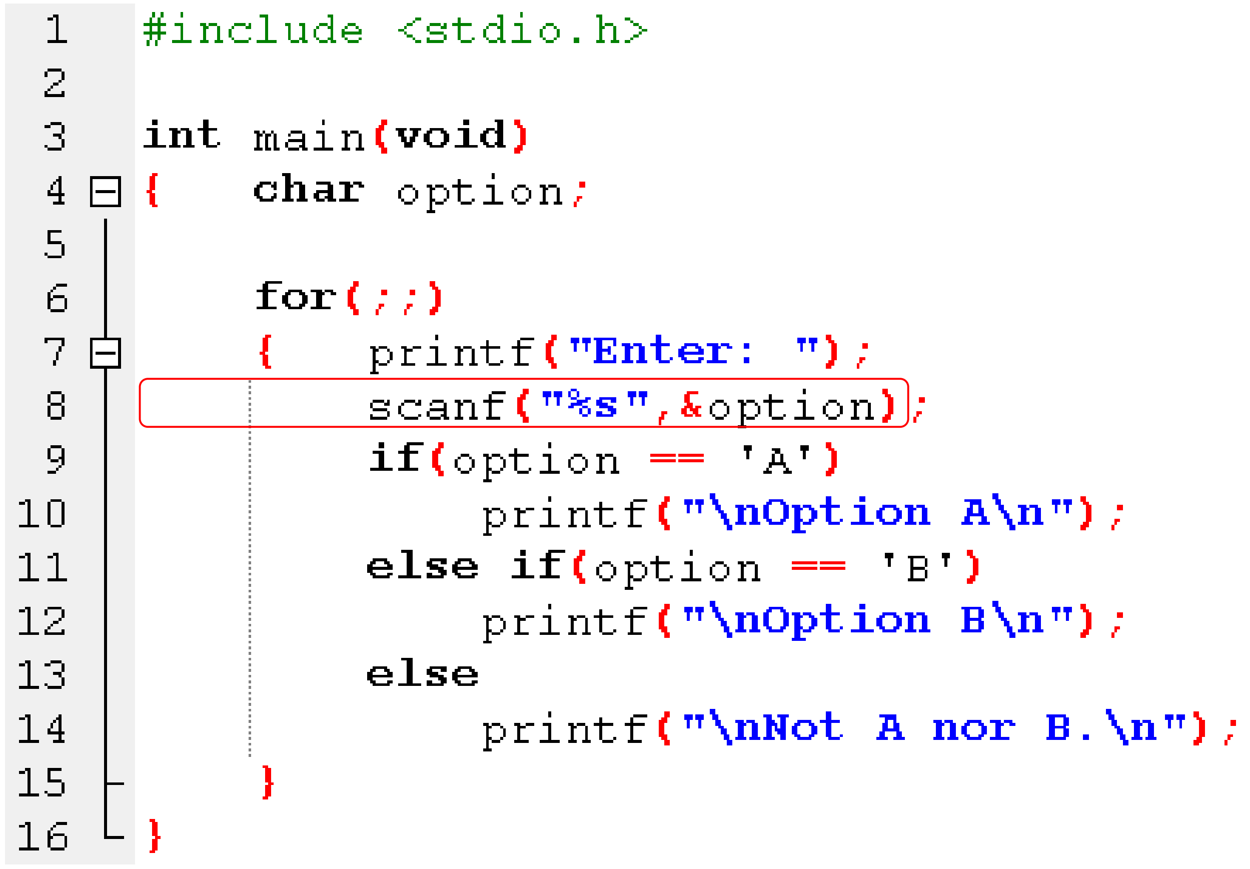


```
Enter: A

Option A
Enter:
Not A nor B.
Enter: B
```

scanf() Function

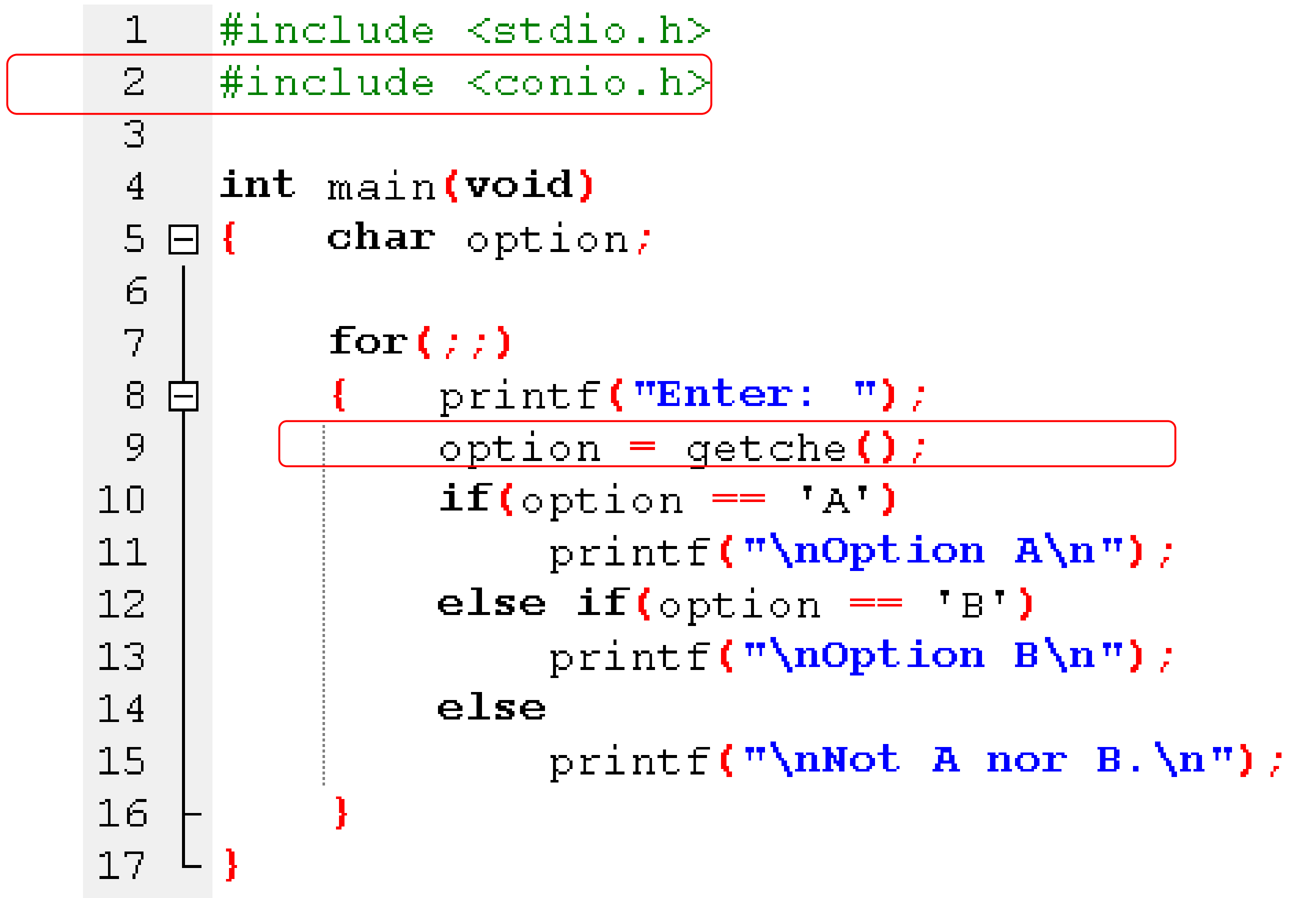
- How to fix this???
- You may take string as input
  - scanf(“%**s**”,&option);
- Or you may use the getche() function:
  - option = getche();



```
#include <stdio.h>

int main(void)
{   char option;

    for(;;)
    {   printf("Enter: ");
        scanf("%s",&option);
        if(option == 'A')
            printf("\nOption A\n");
        else if(option == 'B')
            printf("\nOption B\n");
        else
            printf("\nNot A nor B.\n");
    }
}
```



```
#include <stdio.h>
#include <conio.h>

int main(void)
{   char option;

    for(;;)
    {   printf("Enter: ");
        option = getche();
        if(option == 'A')
            printf("\nOption A\n");
        else if(option == 'B')
            printf("\nOption B\n");
        else
            printf("\nNot A nor B.\n");
    }
}
```

`rand()`: “pseudo” random function

- `rand()` function
  - `int rand(void);`
  - Returns an integer value randomly between 0 and 32,767 (max of `int` type)
  - Include the `<stdlib.h>` and `<time.h>` library
  - `#include <stdlib.h>`
  - `#include <time.h>`
  - Initialize random seed with `srand(time(NULL))` before calling `rand()`

- How do you use the `rand()` function to generate the random number you want?
  - e.g. the roll of a die [1,6]
  - e.g. pick a random number from 1-45 [1,45]

```
srand(time(NULL));    // initialize random seed
int die = rand() % 6 + 1;

rand() % 6;       // result is a random number between [0,5]
rand() % 6 + 1;  // result is a random number between [1,6]
```

Function for rolling a die

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int RollDie(void)
{  return rand() %6 + 1;
}

int main(void)
{ srand(time(NULL));   // initialize random seed

  printf("Roll a die: %d\n",RollDie());
  printf("Roll a die: %d\n",RollDie());
  printf("Roll a die: %d\n",RollDie());
  printf("Roll a die: %d\n",RollDie());
  printf("Roll a die: %d\n",RollDie());
  printf("Roll a die: %d\n",RollDie());
  return 0;
}
```

Function's Arguments

- Function prototype:
  - `int MyFunction(int x, int y)`
  - This function has two arguments, both are integers

- When you call this function, for example:

```
int x = 10;
int y = x + 2;
int result = MyFunction(x, y);
```

- What if you pass arguments with a wrong type, for example:

```
 double x = 10;
 double y = x + 2;
 int result = MyFunction(x, y);
```

Arguments Coercion

- Argument Coercion: forcing the arguments into the correct types
  - Consider the square root `sqrt()` function in the `<math.h>` library
    - The `sqrt()` function has the function prototype:
      ```
      double sqrt(double x);
      ```
    - You are expected to pass a double variable as the argument:
      ```
      double x = 4.0;
      double result = sqrt(x); //result = 2.0
      ```
    - If you pass an integer variable as argument, it will still work properly; it will be “coerced” into double
      ```
      int x = 4;
      double result = sqrt(x); //result = 2.0
      ```
      - Variable `x` is coerced into `double`, and then passed as argument

- The function prototype causes the compiler to convert a *copy* of the integer value 4 to the double value 4.0 before it is passed to sqrt.

Arguments Coercion: A Deeper Look

- You need to be very careful if you're passing arguments with a different type to a function:
  - These conversions can lead to incorrect results
  - For example, consider the following function:

```
int square(int x); // returns the square value
```

  - If you pass a double variable to this function

```
double x = 4.5;
int result = square(x);
```

  - Variable `result` will be 16 (4^2); not 20.25 (4.5^2)

- Converting large integer types to small integer types (e.g., long to short) may also result in changed values.
  - For example, consider the following function:

```
short square(short x); // returns the square value
```

  - If you pass a long variable to this function

```
long x = 60000;
long result = square(x);
```

  - Variable `result` will not be correct

Functions calling functions

```
#include <stdio.h>

void a(void);
void b(void);
void c(void);
void d(void);

int main(void) {
// your code goes here
a();
return 0;
}
```

```
void a() {
printf("a\n");
b();
return;
}

void b() {
printf("b\n");
c();
return;
}

void c() {
printf("c\n");
d();
return;
}

void d() {
printf("d\n");
return;
}
```

Functions calling functions

```
#include <stdio.h>

void a(void);
void b(void);
void c(void);
void d(void);

int main(void) {
// your code goes here
a();
return 0;
}
```

```
void a() {
printf("a\n");
b();
return;
}

void b() {
printf("b\n");
c();
return;
}

void c() {
printf("c\n");
d();
return;
}

void d() {
printf("d\n");
a();
return;
}
```

Functions calling functions

```
#include <stdio.h>

void a(void);
void b(void);
void c(void);
void d(void);

int main(void) {
// your code goes here
a();
return 0;
}
```

```
void a() {
printf("a\n");
a();
return;
}

void b() {
printf("b\n");
c();
return;
}

void c() {
printf("c\n");
d();
return;
}

void d() {
printf("d\n");
return;
}
```

Recursion

- Recursion is “*A programming technique whereby a function calls itself either directly or indirectly*.”
- Recursion is a powerful method that is a key component to functional programming

```
int MyFunction(int x)
{ return MyFunction(x-1);
}
```

```
int MyFunction1(int x)
{ return MyFunction2(x-1);
}

int MyFunction2(int x)
{ return MyFunction1(x-1);
}
```

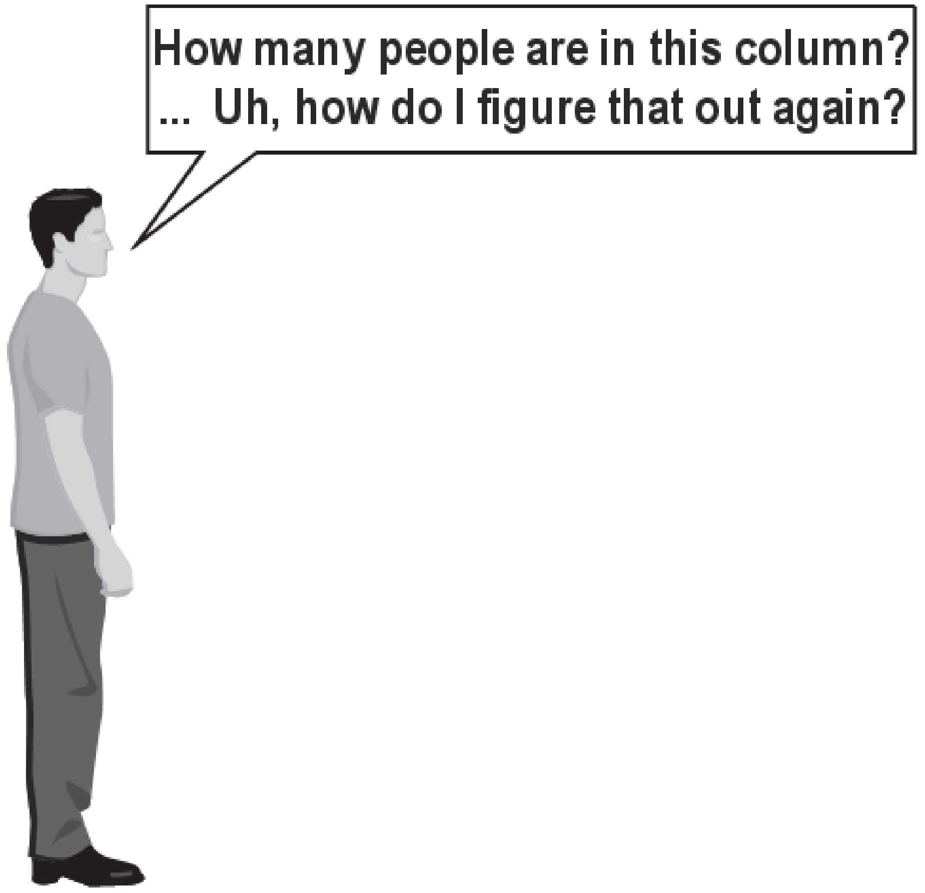
Problem...

(To a student in the front row)
<u>How many students total are directly behind you in your "column" of the classroom?</u>

You have poor vision, so you can
see only the people right next to you.
So you can't just look back and count.

But you are allowed to ask
questions of the person next to you.

How can we solve this problem?
(*recursively*)

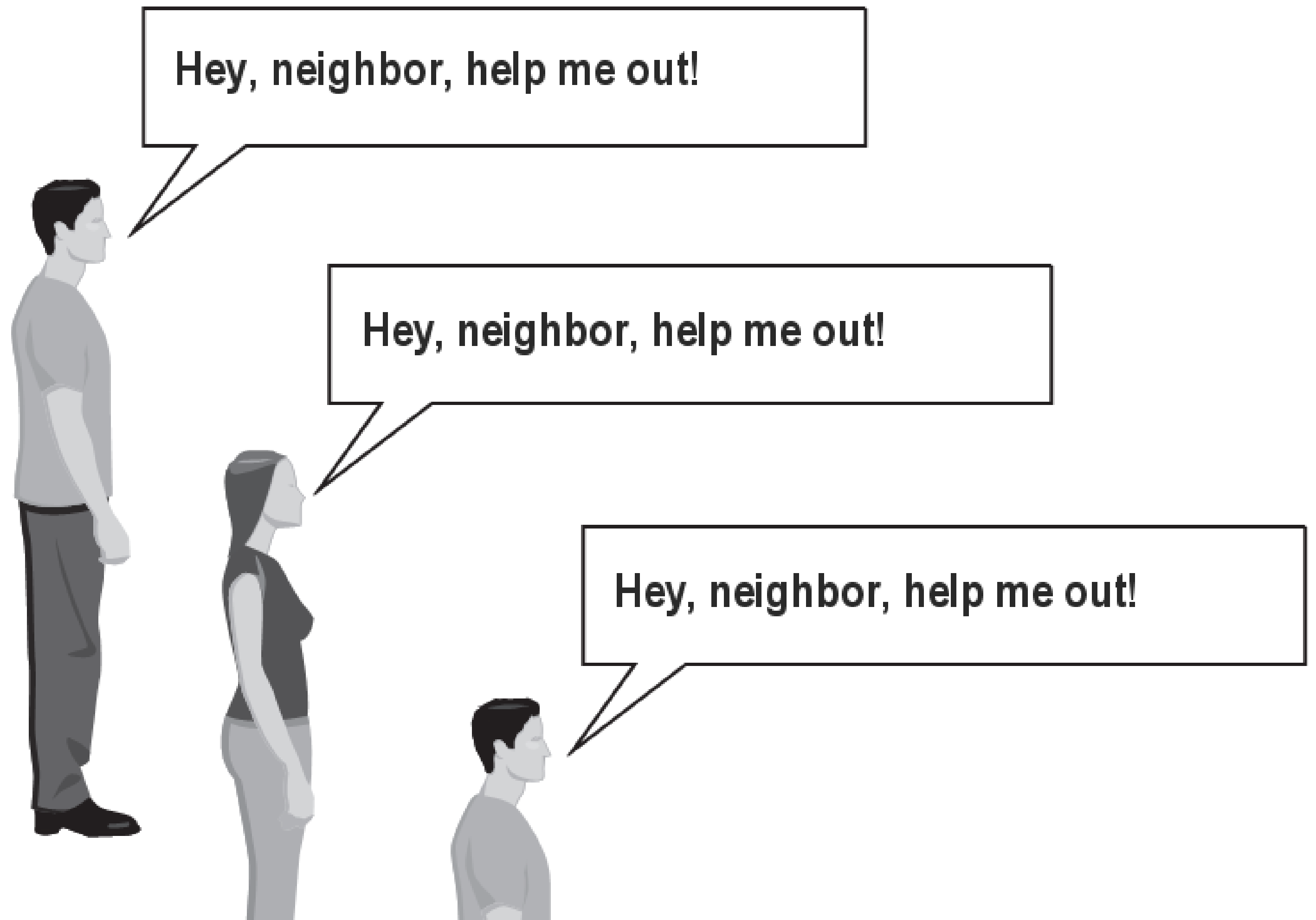

How many people are in this column?
... Uh, how do I figure that out again?

The idea

Recursion is all about breaking a big problem into smaller occurrences of that same problem.

Each person can solve a small part of the problem.

- What is a small version of the problem that would be easy to answer?
- What information from a neighbor might help me?


Hey, neighbor, help me out!
Hey, neighbor, help me out!
Hey, neighbor, help me out!

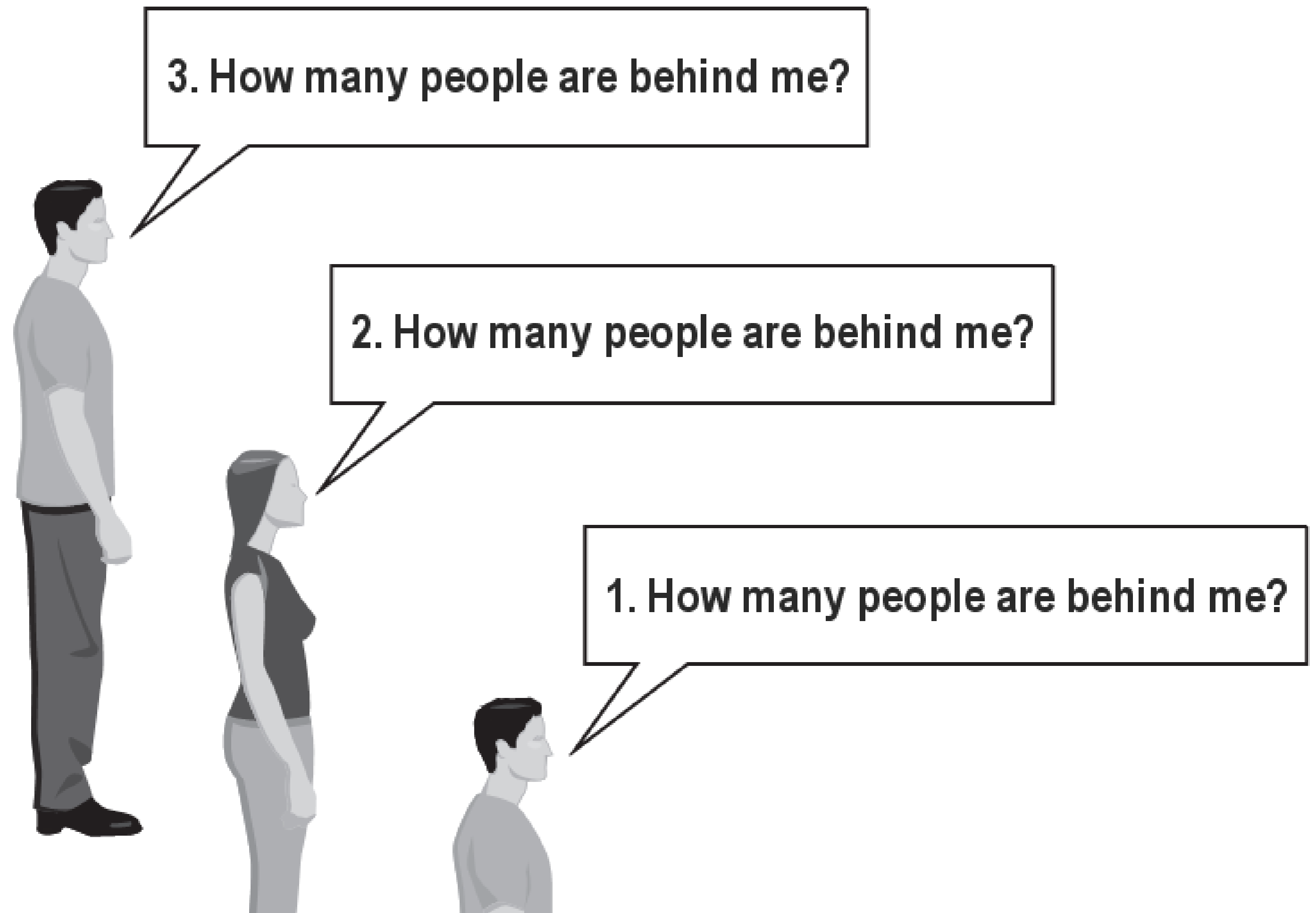
Recursive algorithm

Number of people behind me:

If there is someone behind me,
ask him/her how many people are behind him/her.

- When they respond with a value **N**, then I will answer **N + 1**.

If there is nobody behind me, I will answer **0**.


3. How many people are behind me?
2. How many people are behind me?
1. How many people are behind me?

Recursion and cases

- Every recursive algorithm involves at least 2 cases:

  base case: A simple occurrence that can be answered directly.

  recursive case: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

  Some recursive algorithms have more than one base or recursive case, but all have at least one of each.

  A crucial part of recursive programming is identifying these cases.

Recursion – Write a Function Calculating: n + (n-1) + (n-2) + … + 2 + 1

First Trail (a wrong answer): Recursion without a base case

```
int h(n):
{    return n + h(n – 1);
}
```

```
h(4)
4 + h(3)
4 + 3 + h(2)
4 + 3 + 2 + h(1)
4 + 3 + 2 + 1 + h(0)
4 + 3 + 2 + 1 + 0 + h(-1)
4 + 3 + 2 + 1 + 0 + -1 + h(-2)
...
```

Evaluating h leads to an infinite loop!

Recursion – Write a Function Calculating:
n + (n-1) + (n-2) + … + 2 + 1

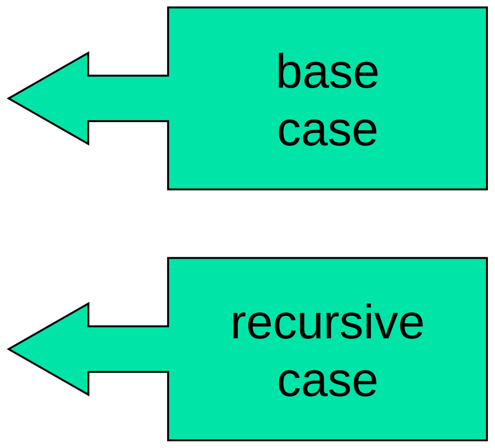
Second Trail (the correct answer)

```
int h(n):
{   if n == 1:
        return 1
    else:
        return n + h(n - 1)
}
```

h(4)
4 + h(3)
4 + 3 + h(2)
4 + 3 + 2 + h(1)
4 + 3 + 2 + 1

Terminology

```
int f(int n)
{   if (n == 1)
        return 1;
    else
        return n+f(n – 1);
}
```



"Useful" recursive functions have:

at least one *recursive case*

at least one *base case*
    so that the computation terminates

Factorial

n! = n x (n-1) x (n-2) x … x 3 x 2 x 1

4! = 4 × 3 × 2 × 1 = 24

Factorial

The value of 9! = 362,880

Does anyone know the value of 10! ???

How did you know?

Factorial

$9! = \quad\quad 9\times8\times7\times6\times5\times4\times3\times2\times1$

$10! = 10 \times 9\times8\times7\times6\times5\times4\times3\times2\times1$

$10! = 10 \times 9!$

$n! = n \times (n - 1)!$

That's a recursive definition!

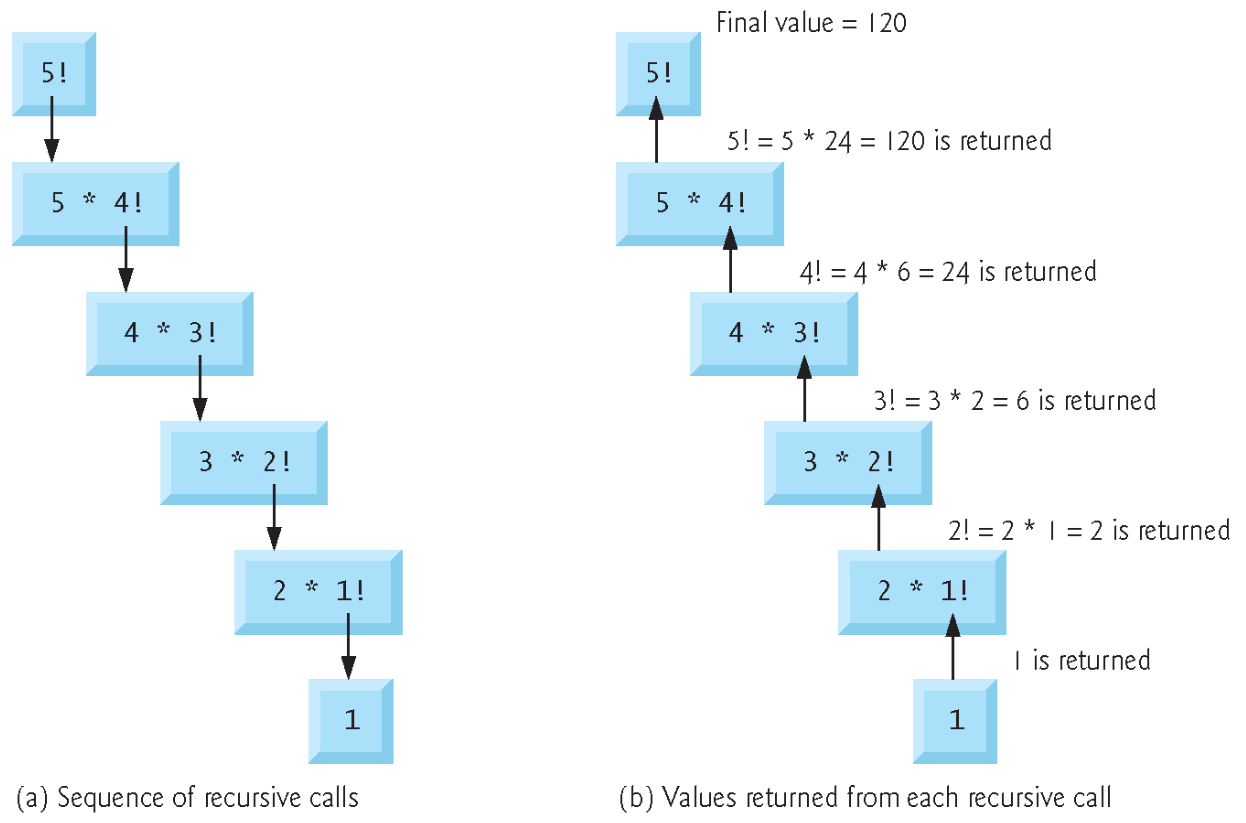
Recursive Definitions

In other words,

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

This definition says that 0! is 1, while the factorial of any other number is that number times the factorial of one less than that number.

Recursion



5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls
Final value = 120
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1 is returned
(b) Values returned from each recursive call

Fig. 5.17 | Recursive evaluation of 5!.

Recursion

```c
// Fig. 5.18: fig05_18.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial( unsigned int number );

// function main begins program execution
int main( void )
{
   unsigned int i; // counter

   // during each iteration, calculate
   // factorial( i ) and display result
   for ( i = 0; i <= 21; ++i ) {
      printf( "%u! = %llu\n", i, factorial( i ) );
   } // end for
} // end main

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

Fig. 5.18 | Recursive factorial function. (Part 2 of 3.)

Recursion

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Fig. 5.18 | Recursive factorial function. (Part 3 of 3.)

Recursion

- The recursive `factorial` function first tests whether a *terminating condition* is true, i.e., whether `number` is less than or equal to 1.
- The conversion specifier `%llu` is used to print `unsigned long long int` values.
- Unfortunately, the `factorial` function produces large values so quickly that even `unsigned long long int` does not help us print very many factorial values before the maximum value of an `unsigned long long int` variable is exceeded.
- Even when we use `unsigned long long int`, we still can’t calculate factorials beyond 21!

Recursion - example

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges to a constant value of 1.618….

Recursion - example

This number, too, repeatedly occurs in nature and has been called the golden ratio or the golden mean.

Humans tend to find the golden mean aesthetically pleasing.

Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean.

Postcards are often designed with a golden mean length/width ratio.

Recursion - example

The Fibonacci series may be defined recursively as follows:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(*n*) = *fibonacci*(*n* – 1) + *fibonacci*(*n* – 2)

Figure 5.19 calculates the *n*th Fibonacci number recursively using function `fibonacci`.

Notice that Fibonacci numbers tend to become large quickly.

Therefore, we've chosen the data type `unsigned int` for the parameter type and the data type `unsigned long long int` for the return type in function `fibonacci`.

In Fig. 5.19, each pair of output lines shows a separate run of the program.

Recursion - example

```c
// Fig. 5.19: fig05_19.c
// Recursive fibonacci function
#include <stdio.h>

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

// function main begins program execution
int main( void )
{
   unsigned long long int result; // fibonacci value
   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
   result = fibonacci( number );

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

// Recursive definition of function fibonacci
```

Fig. 5.19 | Recursive fibonacci function. (Part 1 of 3.)

Recursion - example

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

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

Fig. 5.19 | Recursive fibonacci function. (Part 2 of 3.)

Recursion - example

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 40
Fibonacci( 40 ) = 102334155
```

Fig. 5.19 | Recursive fibonacci function. (Part 3 of 3.)

Recursion - example

The call to `fibonacci` from `main` is not a recursive call (line 18), but all subsequent calls to `fibonacci` are recursive (line 32).

Each time `fibonacci` is invoked, it immediately tests for the base case—`n` is equal to 0 or 1.

If this is true, `n` is returned.

Interestingly, if `n` is greater than 1, the recursion step generates two recursive calls, each a slightly simpler problem than the original call to `fibonacci`.

Figure 5.20 shows how function `fibonacci` would evaluate `fibonacci(3)`.

Recursion - example

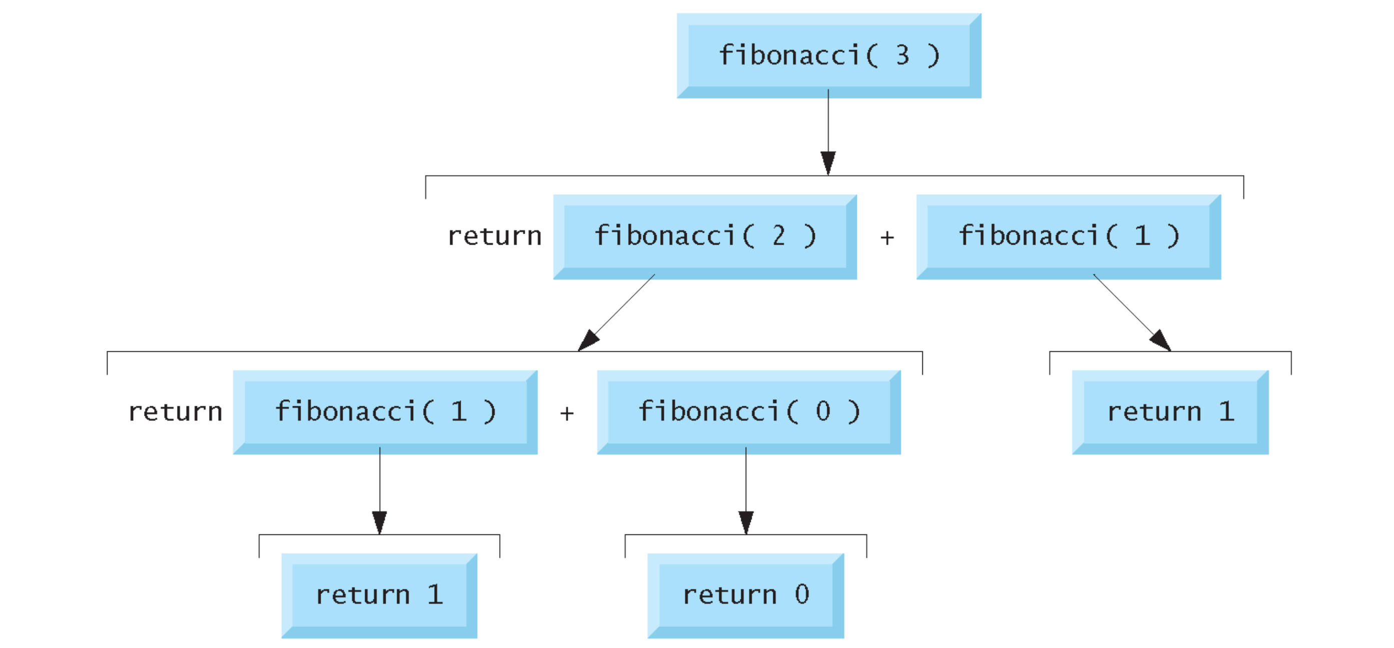


Fig. 5.20 | Set of recursive calls for `fibonacci( 3 )`.

Recursion - example

Order of Evaluation of Operands

This figure raises some interesting issues about the order in which C compilers will evaluate the operands of operators.

This is a different issue from the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence.

Fig. 5.20 shows that while evaluating `fibonacci(3)`, two recursive calls will be made, namely `fibonacci(2)` and `fibonacci(1)`.

But in what order will these calls be made? You might simply assume the operands will be evaluated left to right.

Recursion - example

For optimization reasons, C does not specify the order in which the operands of most operators (including `+`) are to be evaluated.

Therefore, you should make no assumption about the order in which these calls will execute.

The calls could in fact execute `fibonacci(2)` first and then `fibonacci(1)`, or the calls could execute in the reverse order, `fibonacci(1)` then `fibonacci(2)`.

In this program and in most other programs, the final result would be the same.

Recursion - example

Exponential Complexity

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers.

Each level of recursion in the `fibonacci` function has a doubling effect on the number of calls—the number of recursive calls that will be executed to calculate the n^{th} Fibonacci number is on the order of 2^n.

This rapidly gets out of hand.

Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on.

The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches.

Calculating a^n using recursive function

- Calculate a^n

```
int power(int a, int n);

int main( void )
{
int result=power(2,10);
printf("%d\n", result);
return 0;
}
```

```
int power(int a, int n) {

      if (n==0)
         return 1;
      return a*power(a,n-1);
}
```

Recursion vs. iteration

In the previous sections, we studied functions that can easily be implemented either recursively or iteratively.

In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through *repeated function calls*.

In fact, anything that can be done with a loop can be done with a simple recursive function!

Some problems that are simple to solve with recursion are quite difficult to solve with loops (e.g. tree traversal)

Recursion solution is usually shorter (less lines of code)

Performance: Sometimes recursion solution is faster than iteration solution (but sometimes it is slower)

Recursion is another tool in your problem-solving toolbox.

Recursion vs. iteration

Iteration and recursion each involve a *termination test*:

- Iteration terminates when the *loop-continuation condition fails*;
- Recursion when a *base case is recognized*.

Iteration with counter-controlled repetition and recursion each *gradually approach termination*:

- Iteration keeps modifying a counter until the counter assumes a value that makes the *loop-continuation condition fail*;
- Recursion keeps producing simpler versions of the original problem until the base case is reached.

Recursion vs. iteration

Both 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 each time in a manner that converges on the base case.

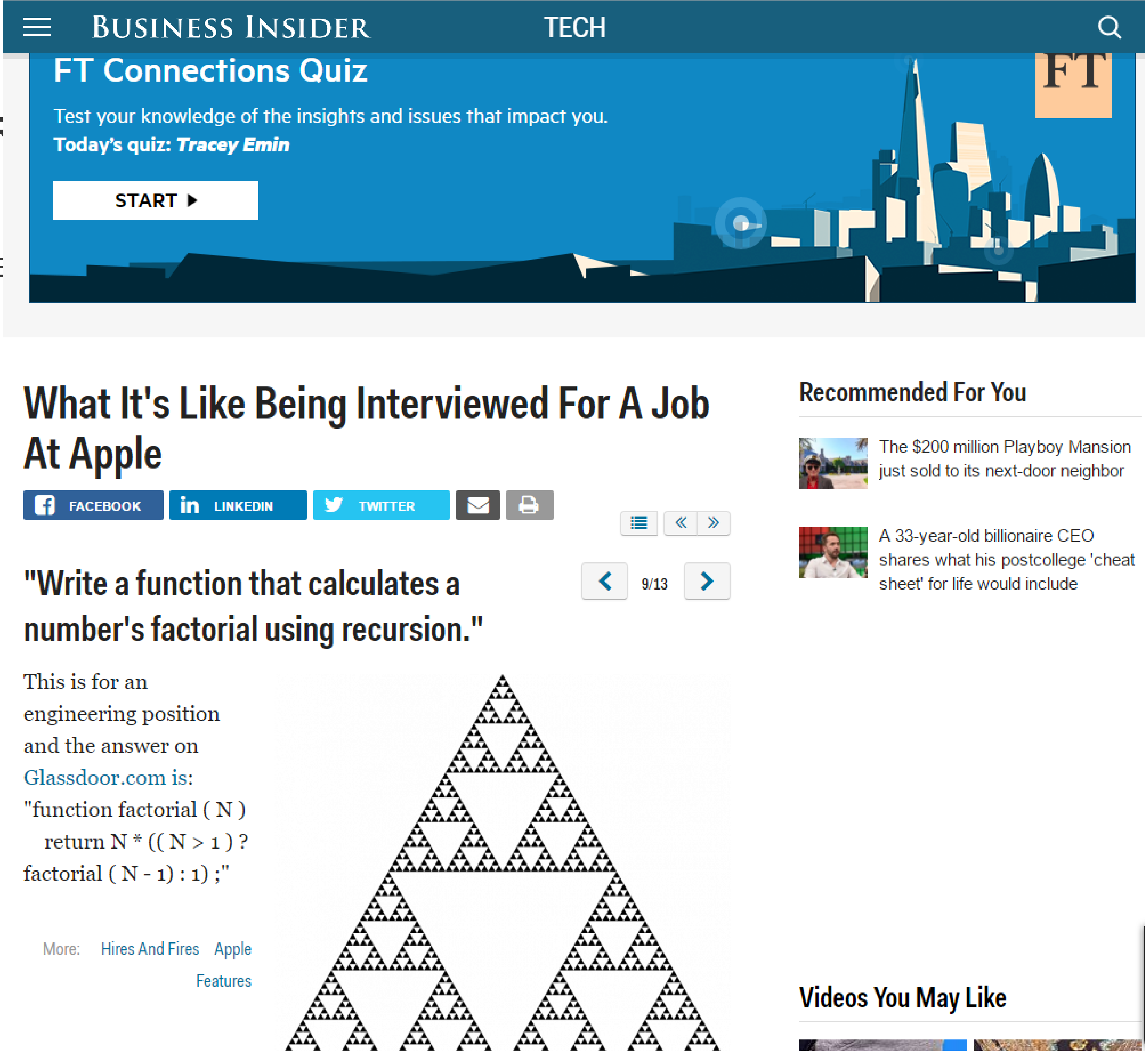
Recursion has many negatives.

It *repeatedly* invokes the mechanism, and consequently the *overhead, of function calls*.

This can be expensive in both processor time and memory space.

A F

- E



BUSINESS INSIDER TECH

FT Connections Quiz

Test your knowledge of the insights and issues that impact you.

Today's quiz: *Tracey Emin*

START ▶

What It's Like Being Interviewed For A Job At Apple

FACEBOOK LINKEDIN TWITTER

"Write a function that calculates a number's factorial using recursion."

9/13

This is for an engineering position and the answer on Glassdoor.com is:

```
"function factorial ( N )
   return N * (( N > 1 ) ? factorial ( N - 1) : 1) ;"
```

More: Hires And Fires Apple Features

Recommended For You

The $200 million Playboy Mansion just sold to its next-door neighbor

A 33-year-old billionaire CEO shares what his postcollege 'cheat sheet' for life would include

Videos You May Like

Q&A?