# Programming Methodology

Spring 2009

Control and Execution

교육과학기술부
국가지정연구실
National Research Lab

# Topics

**Expression evaluation**

**Statement execution**

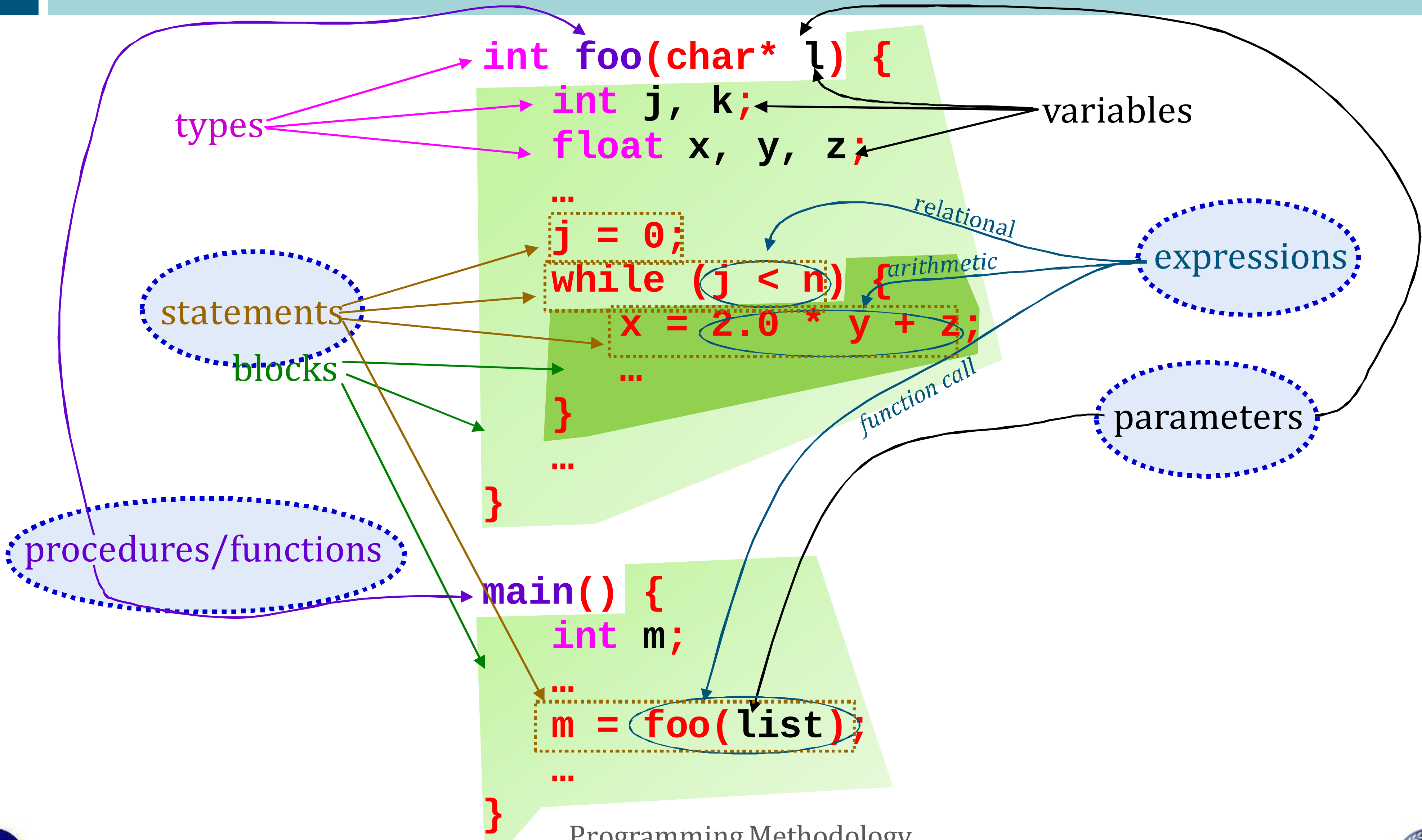
**Function/Procedure call** (with parameters)

**Iterations vs. Recursions**

**Exceptions**

# Programming language constructs





# Control structures



- Control structures control the order of execution of operations in a program.
- **expression-level** control structures
  - → precedence/associativity rules, parentheses, function calls
- **statement-level** control structures
  1. sequential structures: *$stmt_1$; $stmt_2$; …; $stmt_n$;*
     → a sequence of compound statements
  2. selective structures: *if-then-else*, *case*/*switch*
  3. iterative structures: *for*, *while*, *do*, *repeat*
  4. escape/exception/branch: *exit*, *break*, *goto*, *continue*
  5. recursive structures: by recursive function calls

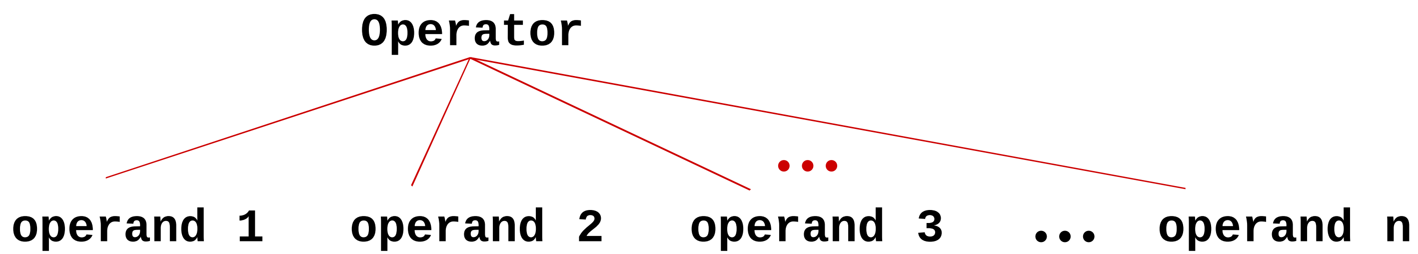
# Expressions

- An expression is ...
  - a means of specifying computations in a program.
  - composed of one or more operations.
- An operation = an operator + zero or more operands
  - operators: arithmetic, logical, relational, assignment, procedure call, reference/dereference, comma, id, constant, ...
  - operands: sub expressions

C++

```
p = &z;
x = y + *p / 0.4;
a[i] = (z > 0 ? foo(x, y, p) : -1);
```

- Syntax tree: abstract representation of expressions



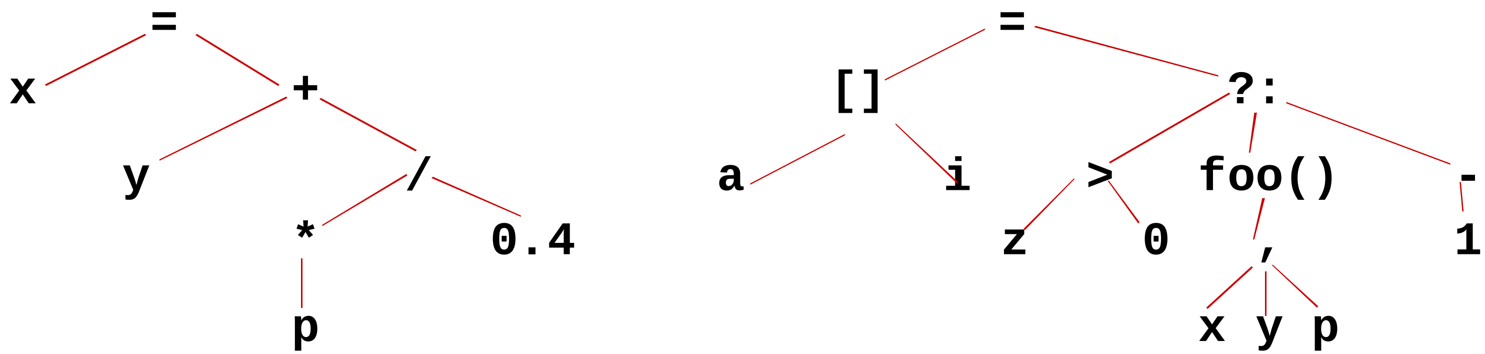

→ a node = an operator, children of a node = operands

# Evaluation of expressions



- Executing a program is actually a sequence of evaluation of expressions in the program.
- How does the compiler/machine determine the evaluation order of an expression?

→ *use a **syntax tree***




- The expression evaluation order in a language (in other words, the way to build a syntax tree) is defined by the language semantics.

# Rules specifying evaluation orders



- **Precedence rule**: the relative priority of operators when more than one kinds of operator are present
  - Ex: "* has higher precedence than +"
    - → thus, 3+4*5 is equal to 3+(4*5), not (3+4)*5.


- **Associativity rule**: the relative priority of operators when two adjacent operators with the same precedence occur in an expression
  - Ex: "- is left-associative" → thus, 3-4-5 is equal to (3-4)-5, not 3-(4-5)

++++q →

# Operator precedence/associativity in C

t->r.x *p++



| Precedence | Operators | Associativity |
|---|---|---|
| 15 | -> . [] () | Left |
| 14 | ++ -- ~ ! unary+ unary- * & | Right |
| 13 | * / % | Left |
| 12 | + - | Left |
| 11 | << >> | Left |
| 10 | < > <= => | Left |
| 9 | == != | Left |
| 8 | & | Left |
| 7 | ^ | Left |
| 6 | \| | Left |
| 5 | && | Left |
| 4 | \|\| | Left |
| 3 | ?: | Left |
| 2 | = | Right |
| 1 | , | Left |

p+++q →

p++++q →

p+++ +q →

p+++-q →

# Sequential structures



- When is the order of a sequence of compound statements important?

```
C  {
        statement1;
        statement2;
           …
        statementn;
   }
```

➔ It is when there is *data dependence* between the statements. That is, when the same location is modified by different statements.

- Find data dependences in the following statements.

```
a[2] = a[1] + 1
x = a[3] * x
a[4] = a[3] / y
```

```
x = z;
y = 2.3 + z;
w = z / 0.6;
print z;
```

```
x = 3.4
y = x – 2.1;
```

```
read y;
x = 5.0;
x = 7.1;
y = x * 1.1;
```

```
x = 8.8;
y = 3.9 * 1.2;
z = 4.1;
```

```
b[i] = j + 0.1;
b[i+1] = b[i-1];
i = i + 1;
m = b[i] + b[i-1];
```

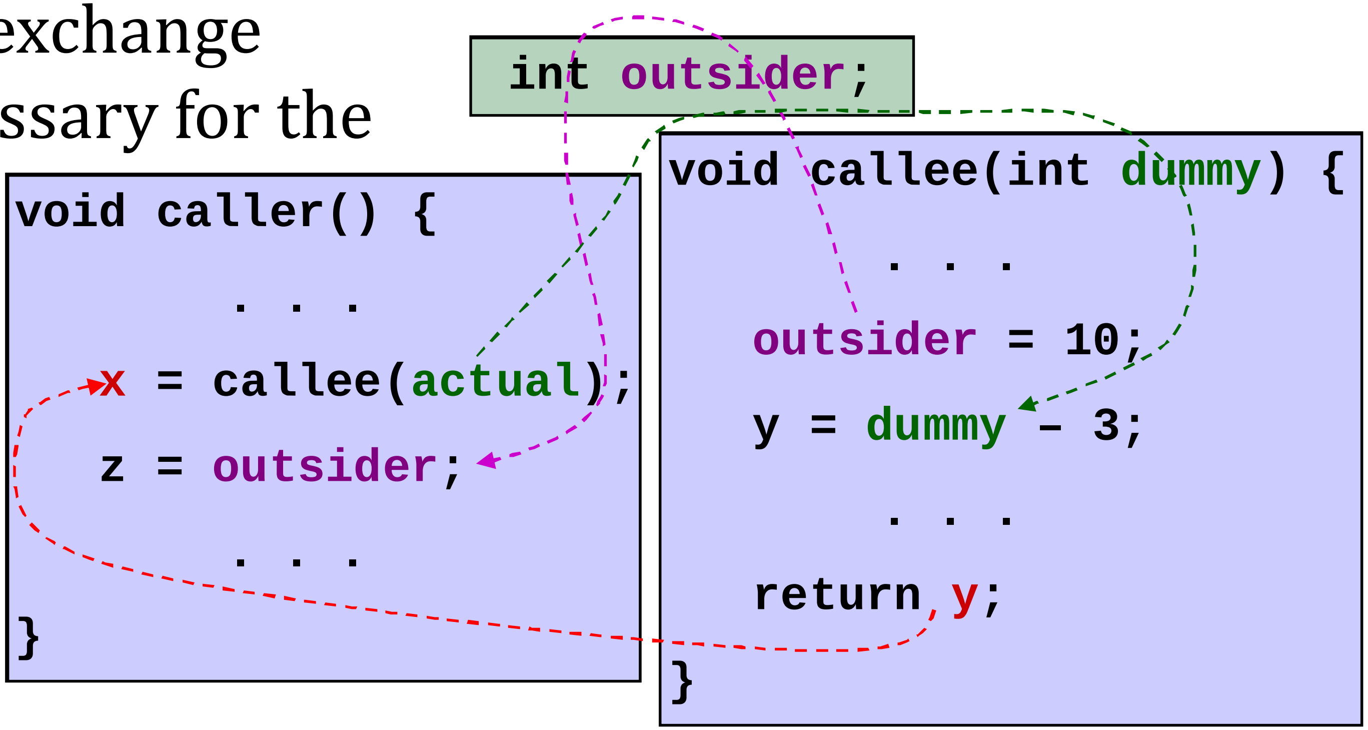
# Selective, iterative structures



- Selective structures:
  - choose control flow depending on conditional test
  - Most languages support → if/then/else, switch/case
- Iterative structures
  - looping construct
  - C → while, for, do-while
- *goto* statements
  - efficient, general purpose, easy to use and translate to machine codes
  - flattens hierarchical program structures into a linear collection of statements → difficult to read/understand
  - difficult to optimize or verify programs

# Function/Procedure calls

- Why is a **function call/invocation** a universal feature in programming languages?
- A function call involves a **caller** and a **callee**.
- The caller and the callee involved in the same call should communicate to exchange information necessary for the call. Then, how?
  - using global variables
  - using parameters and returning values



# Passing function parameters



- Given a function invocation `func(a1,a2,…,an)` ($a_1, a_2, \ldots, a_n$), and a function declaration *type* `func(`*type*$_1$ `d1`, *type*$_2$ `d2`, …, *type*$_n$ `dn)` ($d_1, d_2, \ldots, d_n$),
  - $a_i$ represents an expression for the `i`-th ***actual parameter/argument*** for the invocation provided by the caller
  - $d_i$ represents a variable for the `i`-th ***dummy/formal parameter*** for the callee `func`.
- parameter/argument passing

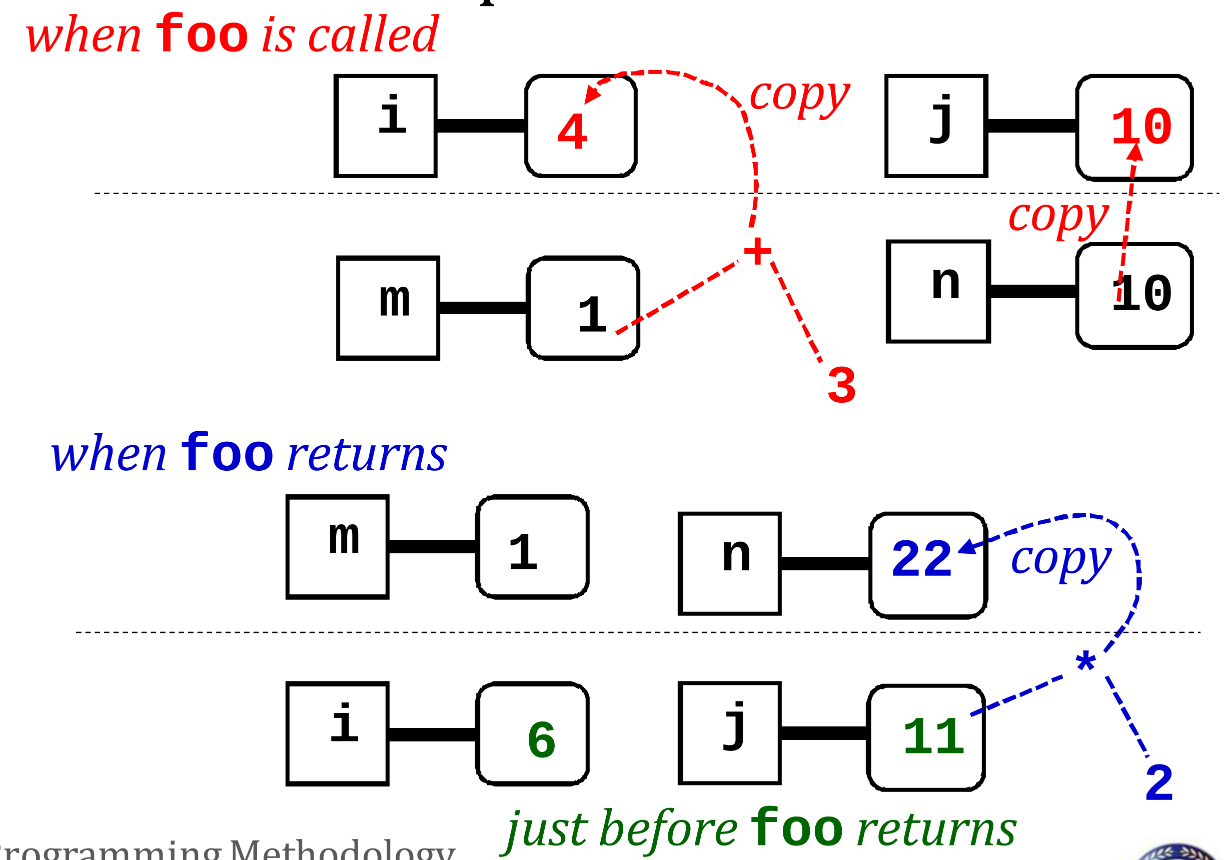
  ➔ Study of the different ways of communication between a caller and a callee with parameters and results
- parameter passing methods
  - Two most popular ones: **call-by-value**, **call-by-reference**
  - Others: call-by-result, call-by-value-result, call-by-sharing, call-by-name, call-by-need ➔ *Out of our scope!*

# Call-by-value



- When a procedure is called, the r-value of an actual parameter is assigned to the l-value of the matching formal parameter.
- secure because changes made on formal parameters do not affect the actual ones.

```
int foo(int i, int j) {
        …
        i = j++ - i;
        return j * 2;
}
void bar() {
        int m = 1;
        int n = 10;
        …
        n = foo(m+3,n);
}
```



# Call-by-value



- Typically, not appropriate if a callee wants to return multiple output results

```
int foo(int a, int b, …) {
   …
   return c;
}

int main() {
   …
   z = foo(x, y, …) …
   …
}
```

*single output*

*••• multi inputs*

  - But it is not impossible to return multi-results with call-by-value → use !

- Also, possibly expensive if large data needs to be passed.

```
struct S {
   int x, y;
   float a[1000][1000];
};

void foo(S dum) {
   …
   cout << dum.a[1][1];
   dum.a[1][1] = 99;
   …
}

int main() {
   S act;
   …
   act.a[1][1] = 55;
   foo(act);
   cout << act.a[1][1];
}
```

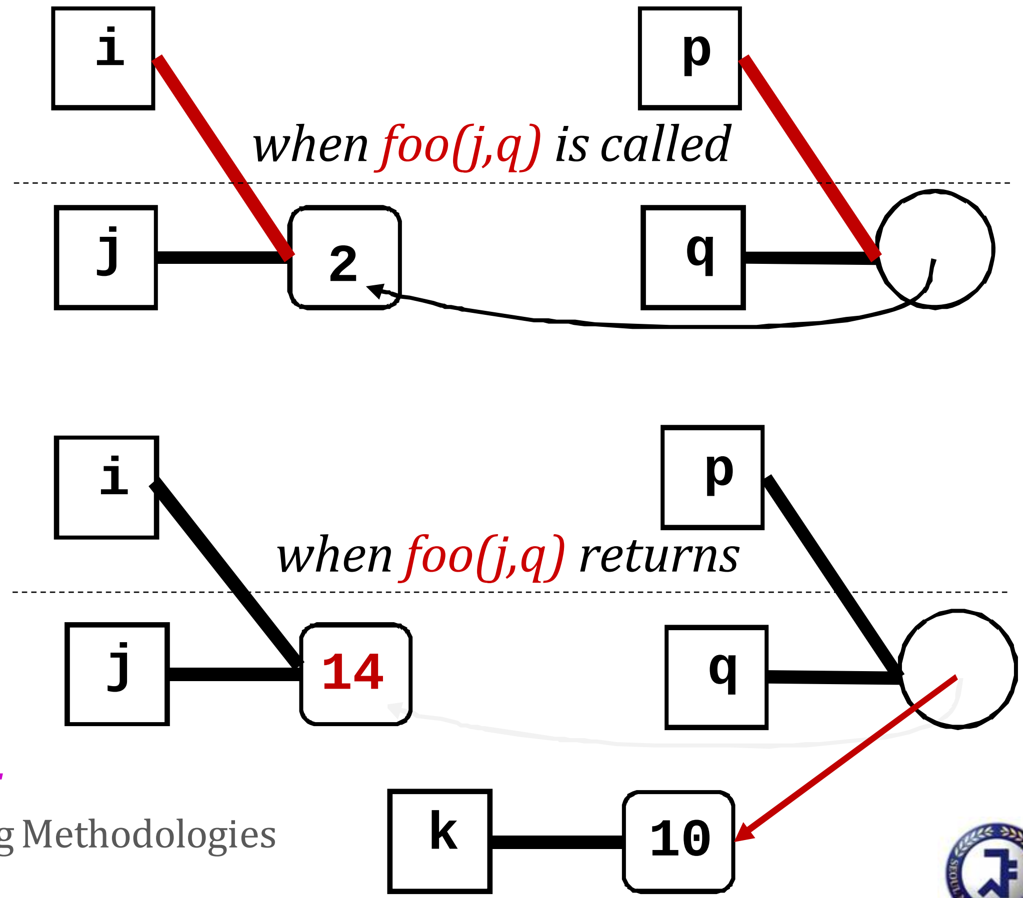
# Call-by-reference/location



- When a procedure is called, the l-value of an actual parameter is shared with the matching dummy parameter. → ***aliasing***
- For this, C++ uses a ***reference type*** for dummy parameters.

```
int k = 10;
void foo(int &i, int* &p) {
        i = 7;
        *p = i + *p;
        p = &k;
}
void bar() {
        int j = 2;
        int* q = &j;
        . . .
        foo(j, q);
        cout << j << *q;
        foo(j+3, q);
}
```

*error: j+3 has not l-value.*



# Call-by-reference/location



- can be used to return multiple output results

```
int foo(int a, int &b, …) {
    …
    b = …
    return c;
}

int main() {
    …
    z = foo(x, y, …) …
    …
}
```

*one output*

*••• multi inputs*

*one more output*

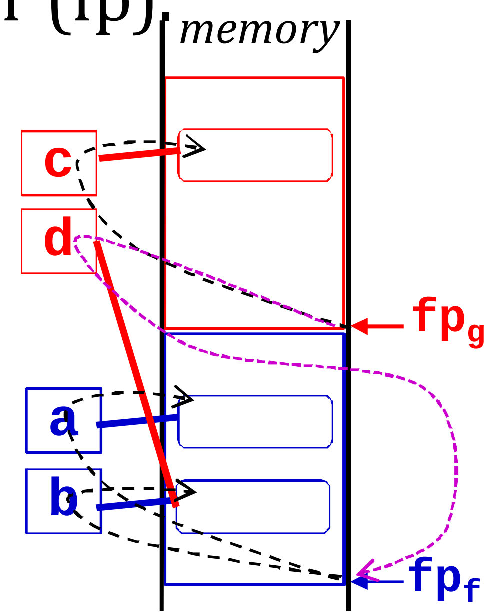
- can be efficient via aliasing when large data needs to be passed.

```
struct S {
    int x, y;
    float a[1000][1000];
};

void foo(S &dum) {
    …
    cout << dum.a[1][1];
    dum.a[1][1] = 99;
    …
}

int main() {
    S act;
    …
    act.a[1][1] = 55;
    foo(act);
    cout << act.a[1][1];
}
```

# Call-by-reference/location

- causes *aliasing*, which makes the code …
  - generally more efficient (ex: long arrays); but
  - error prone due to *side effects*, and
  - in some cases, even less efficient because call-by-reference is often implemented with an extra level of indirection thru a frame pointer (fp).

```
g(int c, int& d) {
   … = c + d
   …
}
f() {
  int a, b;
  … g(a, b);
  …
}
```




```
…
load r1, [fp_g+<c>]      ← c
load r2, [fp_g+<fp_f>]   ← fp_f
load r3, [r2+<b>]        ← d = b
add r4, r1, r3           ← c + d
…
```

```
…
load r14,[fp_f+<a>]      ← a
load r15,[fp_f+<b>]      ← b
call g
…
```

# Write protection thru constant dummy



- Using ***constant dummy parameters*** may prevent erroneous updates or side effects due to aliases created by call-by-reference.
  - Still call-by-reference, so avoid copying.
  - Yet, providing write-protection on dummy parameters

*The function* **gee** *guarantees that the actual parameter (not only* **y** *but also* **x***) is never modified inside* **gee***.*

```
void foo(int a) { // by value
  …
}
void bar(int& b) { // by ref
  …
}
void gee(const int& c) { // by const ref
  …
  c = …     // Compile error: write protection
}
int main() {
   int x = 1965;
   const int y = 2009;
   foo(x);    // OK?
   foo(y);    // OK?
   bar(x);    // OK?
   bar(y);    // OK?
   gee(x);    // OK?
   gee(y);    // OK?
}
```

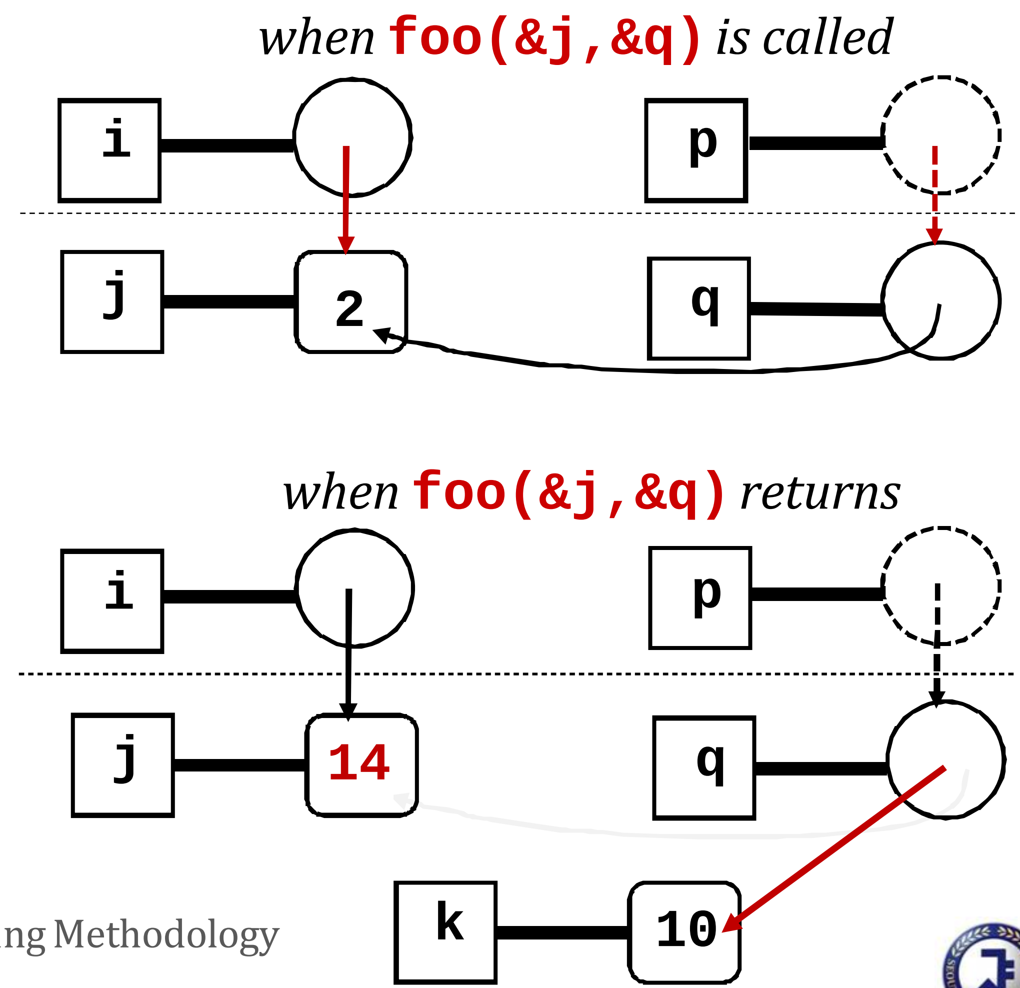
# Simulating call-by-reference thru pointers



- C cannot support true call-by-ref. (∵ *no reference type included*)
- But, it can simulate call-by-ref. by using pointers as call-by-value parameters.

```
int k = 10;
void foo(int *i, int* *p) {
        *i = 7;
        **p = *i + **p;
        *p = &k;
}
void bar() {
         int j = 2;
         int* q = &j;
         . . .
         foo(&j, &q);
         cout << j << *q;
}
```

*Same results as the original code with reference type dummies*



# Multidimensional arrays as parameters

*Note:* **`gee`** *and* **`foo/bar`** *may be compiled in separate files!*



- When a multidimensional array is passed, the callee should know the original dimension of the array declared in the caller.

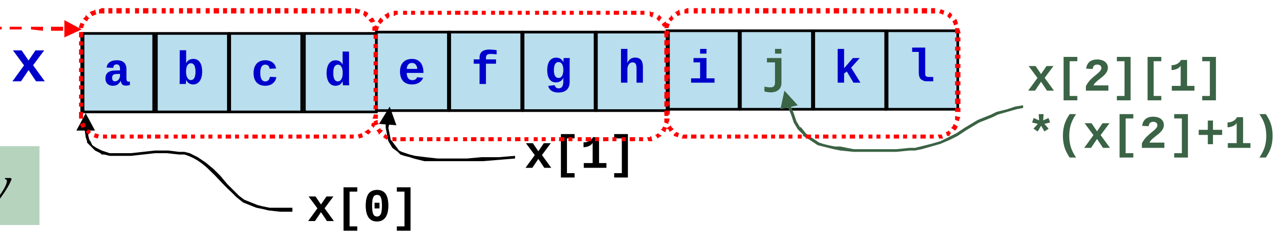
```
void foo () {
   int a[5][9];
   … gee(a) …
```

```
void bar () {
   int b[7][3];
   … gee(b) …
```

```
void gee (int c[][]) {
   …c[2][6]…   //error
}                        (or **c)
```

  - **`c[2][6]`** is invalid when **`bar`** calls **`gee`**. → *How can the compiler find this?*
  - It is valid when **`foo`** calls **`gee`**. → *But how to determine its exact address?*
- Recall: a multidimensional array in C/C++ is an *array of arrays*, and physically stored to 1-D memory in *row-major* order.

**x** consists of 3 row arrays.
Each row is an array with size = 4.




```
int x[3][4];  // 2-D array
```

  - To compute the exact address for **`c[i][j]`**, the compiler must evaluate

    *address of* **`c[i][j] = c+i·n+j`** → **n** = size of row array = # of columns

  - What does this imply? ➔ Compiler must know **n** when **gee** is called.

# Multidimensional arrays as parameters



- One solution for the above case

```
void foo () {
   int a[5][9];
   … geefoo(a) …
```

```
void bar () {
   int b[7][3];
   … geebar(b) …
```

```
void geefoo (int c[][9]) {
   …c[2][6]… // = *(c + 24)
}
```

```
void geebar (int c[][3]) {
   …c[2][6]… // Now the compiler
}          // knows this is error since 6 > 3
```

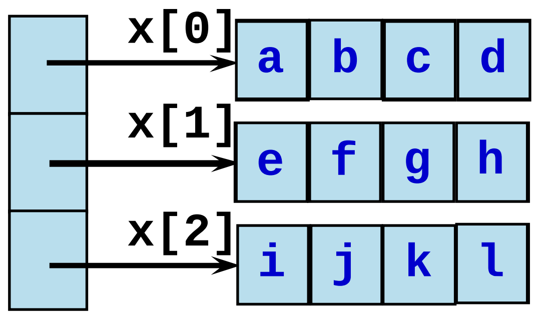
- The problem of this solution?
  - Poor reusability of code and increase of code size
  - Ex: a new function for **gee** must be written for every different row size.
- Alternative solution
  - Pass the array as a *pointer* along with its dimension information

```
#define gee_c(i,j) (*(c + ((i) * n) + (j)))
…
void gee1 (int *c, int m, int n) { // indicating c[m][n]
   … gee_c(2,6) …    // = *(c+24)  if called by foo
                     // = *(c+12)  if called by bar
```

  - The same function **gee1** can be used for **foo** and **bar**, regardless of the caller's array dimension.

# Multidimensional arrays as parameters

- The problem with **`gee1`**? ➔ Yes, indeed… because we need …
  - a special statement '#define' → awkward
  - to manage dimensions (**`m`**, **`n`**) separately → inconvenient, error prone
  - a different '#define' for other array parameters: **`gee_d`**, **`gee_e`**, …
- A better solution? ➔ Create a new **`class`** (i.e., ADT)!
  - In Java (also C#), arrays are objects of a system class, say *Array*.
  - Array objects are all 1-dimensional, but their elements can be also objects of Array. → For example, 2-D array **`x[3][4]`**
  - The Array class offers a named constant '**`length`**' that is set to the length of the array when the array object is created.




*Java Code*

```
int[][] x = new int[3][4];
… x.length …            // = 3
… x[i].length …         // = 4
```

# Multidimensional arrays as parameters



- Using Array objects for parameter passing
  - No need to separately pass dimensions for a multidimensional array since the compiler can extract them from the internal constant **`length`**.

*Java Code*

```
void gee2 (int[][] c) { // setting c.length and c[i].length
  … c[2][6] …     // valid if called by foo
                  // error if called by bar  since 6 > b[i].length = 3
```

```
void foo () {
  int[][] a = new int[5][9];
  … gee2(a) …
```

```
void bar () {
  int[][] b = new int[7][3];
  … gee2(b) …
```

  - Good reusability and code size reduction
- How about C++?
  - Unlike Java, C++ doesn't support such a system class as Array by default.
  - But, the programmers can create similar objects for multidimensional arrays by using the **`class`** construct.

# Functions as parameters



- The traditional view of a function $f: \mathcal{D} \rightarrow \mathcal{R}$

  *integers, characters, real numbers, strings …*

  - Ordinary data objects of primitive types have *first-class values*.
  - $f$ is a static piece of code for mapping input values of first-class into first-class output values.
  - Such a function is said to be **first-order**.
- In programming languages such as C/C++, Java, and Fortran, most functions are first-order.
  - Ex) **foo**: Z (*integer*) → **R** (*real*)

```
float foo (int x) {
   …
}
```

- In some languages, functions themselves can be considered as first-class values so that they can be passed as inputs to or as output from other functions.

```
int main () {
   … f(&g) …
}
```

```
… f ( … ) {
   …
}
```

```
int g (int n) {
   …
}
```

# Functions as parameters



- A function that takes functions as parameters or returns as outputs is called a **higher-order function** (HOF).
- C++ support a limited form of HOFs.
  - A C++ function may take another function as its parameters.
  - For this, C++ uses a *function pointer*.

```
int (*pf) (int);
pf = &square;      // pf = square
cout << pf(5);     // 25
pf = &double;      // Now pf = double
cout << pf(5);     // 10
```

```
int square(int x) {
  return x * x;
}
int double(int y) {
  return y + y;
}
```

  - C++ HOF foo:

```
void foo(int (*pf) (int)) { … }
…
int main() {          ≠ int *pf (int)
   …
   foo(&square);
   foo(&double);
   …
}
```

# HOFs



- HOFs are sometimes powerful and useful.
  - Treating functions as values increase the expressive power of a language. → functions handling functions : sums (Σ), derivatives ($d, \partial$)
  - They help abstract out common control patterns, leading to very concise programs. → repetitive applications of similar tasks
- Where HOFs are useful for concise programming
  - For example, the *summation* in mathematics (denoted by the notation Σ) is a HOF since it takes a function *f* as a parameter.

$$\sum_{j=1}^{m} f(j) = f(1) + f(2) + \ldots + f(m)$$

  - A notation Σ makes a mathematical expression concise and brief by capturing the common patterns among the expression.

$$\sum_{i=1}^{l} \sum_{j=1}^{m} \sum_{k=1}^{n} f(i, j, k) = f(1,1,1) + \ldots + f(1,1,n) + f(1,2,1) + \ldots + f(l, m, n)$$

# HOFs



- Where HOFs are useful for concise programming (cont'd)
  - Σ in math can be represented briefly by a HOF **sig** in C++.
  - Like Σ does for math, **sig** will make a program concise by taking any function *f* as a parameter of the common input/output types.

```
int sig(int (*f) (int), int m) {
  int result = 0;
  for (int j = 1; j < m; j++)
    result += f(j);      // Σj f(j)
  return result;
}
int main () {
  cout << sig(&square, 5);   // Σj j^2
  cout << sig(&double, 5);   // Σj 2j
}
```

```
int square(int x) {
  return x * x;
}

int double(int y) {
  return y + y;
}
```

*What if we cannot use HOFs for this example in C++?*

  - But, **sig** is not as flexible as Σ. ➡ `… sig(&sig(&f,m),n);` *//error :* $\Sigma_i \Sigma_j f(i,j)$

→ Such flexibility is possible in *functional languages* like scheme and ML.

# Another example of HOFs



```
! trapezoidal approximation for the definite integral
function integral (f,a,b,n) result(t)
  interface
   function f(x)
   real::f,x
  end interface
  real, intent(in)::a, b
  integer, intent(in):: n
  real::t
  real::h, sum
  integer:: i
  h = (b - a) / n
  sum = 0.5 * (f(a) + f(b))
  do i = 1, n-1
   sum = sum + f(a+i*h)
  enddo
  t = h * sum
end function integral
function bar(x) …
   …
end function bar
program main …
 …
write (*,*) integral(sin,l=0.0,u=3.14,n=100) ! calculate the approximation of ∫_0^π sin(x)dx
 …
write (*,*) integral(bar,l=1.0,u=2.0,n=15)   ! calculate the approximation of ∫_1^2 bar(x)dx
```

$$\int_a^b f(x)dx = h\left(\frac{f(a)+f(b)}{2} + f(a+h) + \ldots + f(b-h)\right)$$

- Integral $\int$ in math is another example where HOFs are useful for programming.
- Like C++, Fortran90 also supports a function that takes another function as its input parameter.
- The function **`integral`** can be implemented by using function parameters in C++.

# Evaluation order of function arguments



- Given a function invocation **func**($a_1, a_2, \ldots, a_n$), and a function declaration *type* **func**($type_1$ $d_1$, $type_2$ $d_2$, …, $type_n$ $d_n$),
  - $a_i$ represents an <u>expression</u> for the i-th *actual argument* for the invocation
  - $d_i$ represents a <u>variable</u> for the **i**-th *dummy argument* for **func**.
  - all the expressions for actual arguments are usually evaluated before **func** is called. (consider the syntax tree)
- The order of evaluation is imposed differently depending on specific languages or compilers.
  - no order imposed → Fortran
  - right-to-left → gnu C++, Visual C++
- The order is important due to _s__________t_ of expressions.

# Evaluation order of function arguments



- right-to-left → *gnu C++*

```
void foo(int m, int n) { cout << m * 10 + n; }

void bar(int &a, int &b) { cout << a * 100 + b; }

int main() {
    int k = 5;
    foo(k+1, k+1);  // call-by-value w/o side effect
    foo(++k, ++k);  // w/ side effect
    foo(k++, k++);  // w/ side effect
    bar(k+1, k+1);  // call-by-ref. w/o side effect → error:
    bar(++k, ++k);  // w/ side effect
    bar(k++, k++);  // w/ side effect → error:
}
```

output

- left-to-right
- Comparison of expressions **k++** and **++k**
  - ++k ≡ (**&b**=(k=k+1; **k**)
  - k++ ≡ (**&b**=(k; k=**k+1**)

So… →

```
++k = 10;   // ok
k++ = 10;   // error
```

# Evaluation order of function arguments



- Visual C++: *also right-to-left, but slightly different from gnu C++*

```
void foo(int m, int n) {
    cout << m * 10 + n;
}
void bar(int &a, int &b) {
    cout << a * 100 + b;
}
int main() {
    int k = 5;
    foo(k++, k++);  // output →     :     gnu C++
    foo(++k, ++k);  // output →     :     gnu C++
    foo(k++, ++k);  // output →     :     gnu C++
    foo(++k, k++);  // output →     :     gnu C++
    bar(++k, ++k);  // output →     :     gnu C++
}
```

**gnu C++**
foo(x++,++y) *// from r-2-l*
n = ++y; *// assign expr*
m = x++; *// assign expr*

**Visual C++** *// from r-2-l*
++y; *// for pre, compute first*
m = x++; *// for post, assign expr*
n = y; *// now assign var for pre*

- *What lesson do we take from the different results of compilers?*
  - Do not make any assumption on the evaluation order even with C/C++.
  - For better portability, compute all actual arguments that have potential side effects before the function invocation. →

```
x = k++;
y = ++k;
foo(x,y)
```

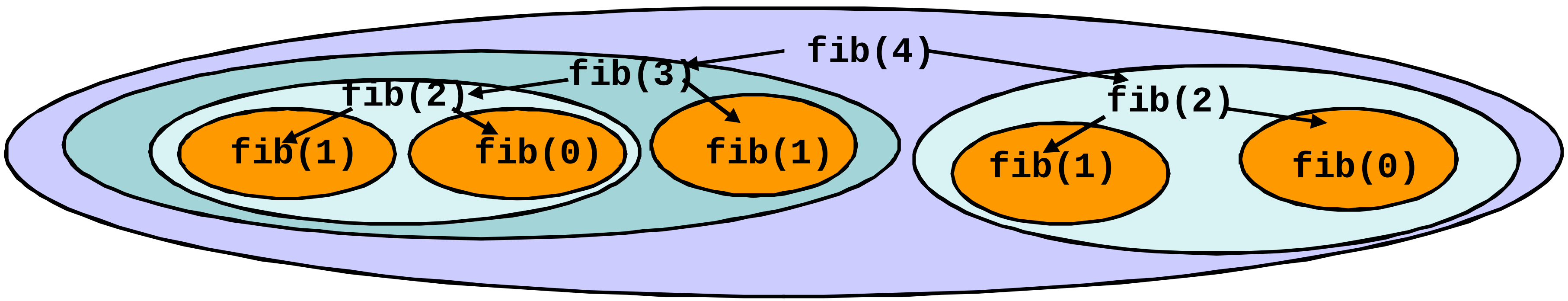
# Recursive structures



- A function ***f*** is *recursive* if it contains an application of ***f*** in its definition.

```
int fib(int n) {
  return ((n==0||n==1) ? 1 : fib(n-1)+fib(n-2)));
}
```

- Recursion simplifies programming by exploiting a divide-and-conquer method. → "*divide a large problem into smaller ones*"



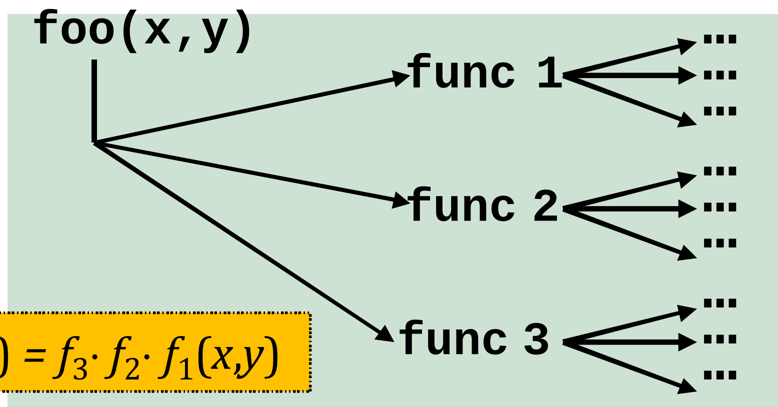

→ Rewrite the function **`fib`** without using recursion, and find how many more lines you need for your code without recursion.

# Recursive structures

Ex: $f_1 = F, f_2 = f_3 = \Sigma$ ➔ $foo(i, j) = \sum_j \sum_i F(i, j)$



- Recursion allows users to implement their algorithms in the applicative style rather than the imperative style.




$foo(x,y) = f_3 \cdot f_2 \cdot f_1(x,y)$

*Applicative/Functional Programming*

```
foo(x,y) {
        do this;
        do that;
        …
        do this;
        do that;
}
```

*Procedural/Imperative Programming*

- Recursion can be expensive if not carefully used.
  → *Compare these two functions that compute the factorial*

**compute the factorial with recursion**

```
int fac(int n) {
  return (n==0 ? 1 : n*fac(n-1));
}
```

**compute the factorial with iteration**

```
int faci(int n) {
  for (int p=1; n>0; n--) p=n*p;
    return p;
}
```

# Comparison of `fac` and `faci`



## Computation of `fac(4)`

```
fac(4)
4 * fac(3)
4 * (3 * fac(2))
4 * (3 * (2 * fac(1)))
4 * (3 * (2 * (1 * fac(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2))
4 * 6
24
```

*function call*

*return*

*5 function calls*

*upto 4 words to store the temporal data*

## Computation of `faci(4)`

```
faci(4)
p = 1  < -------  n = 4
p = 4  < -------  n = 3
p = 12 < -------  n = 2
p = 24 < -------  n = 1
                  n = 0
```

*return p*

*1 function call*

*1 word to store the temporal data*

- The main problem with the recursive version is that **`fac`** needs more memory space and function calls as the problem size **`n`** increases.
- In contrast, **`faci`** always needs only 1 function call and 1 word regardless of the value of **`n`**. → *Suppose* **`n = `** *1000!*

# Tail recursion



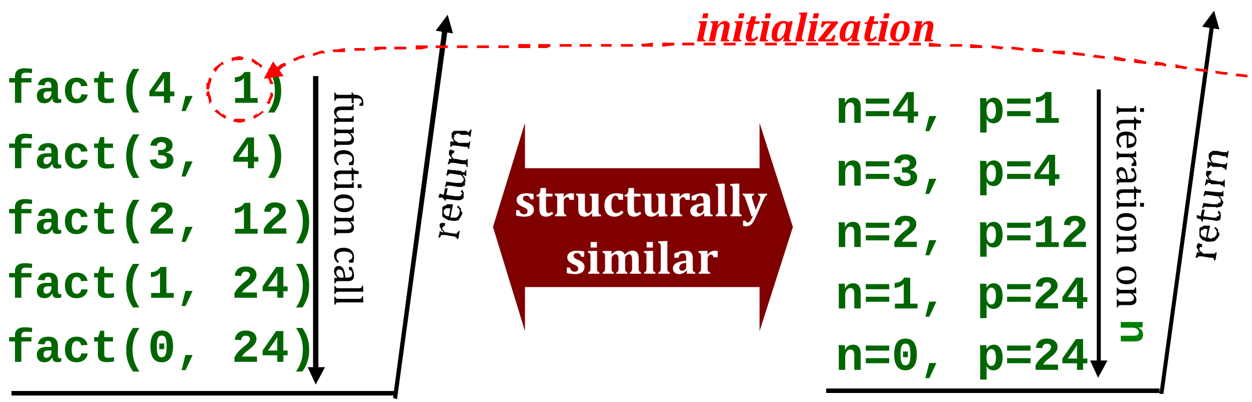
- A function *f* is **tail-recursive** if it is a recursive function that returns either *a value without needing recursion* or *the result of a recursive activation*.

**Ex:**
```
void fact(int n, int& p) { if (n > 0) { p*=n; fact(n-1,p);} }
```
*no* **int**!

→ cf: Neither **fib** nor **fac** is tail-recursive.

- What are tail-recursive functions so great about?

→ It is can always be translated to iterative structure.



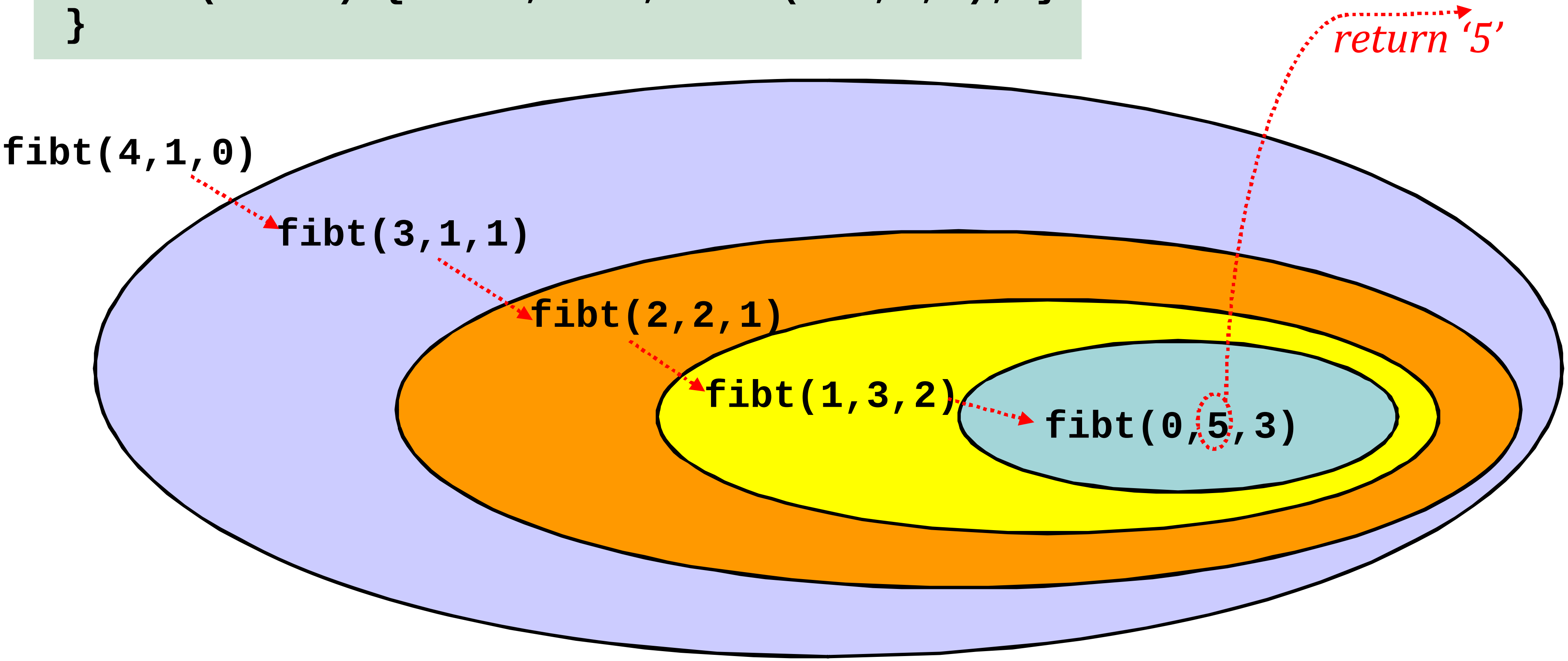

```
int faci(int n) {
  int p = 1;
   for (; n > 0; n--)
      p *= n;
   return p;
}
```

# Application of tail recursion



- Write a tail-recursive version of fib. → efficient & simple

```
void fibt(int n, int& l, int& r) {
   if (n > 0) { l+=r; r=l; fibt(n-1,l,r); }
}
```



# Inlining



- While enjoying the advantages of using functions, can we minimize the overhead for function calls?
- For this, some languages such as C++ support *explicit inlining*.
- Pros and cons of inlining
  - removes the overhead for function call.
  - Reckless use may increase the code size.
  - Inlined code is generally less readable and maintainable.
    - → *So, inlining is ideal for a small procedure invoked within frequently executed regions (e.g., loops).*
    - → *How about procedure with recursion?*

```
void foo() {
    int x, y, z;
    …
    y = bar(z,99);
    z = bar(88,y);
    …
}
inline int bar(int a, b) {
    int x, t;
    x = a * b;
    t = a - b;
    return x / t;
}
```

```
void foo() {
    int x,y,z,x1,t,x2,t1;
    …
    x1 = z * 99;
    t = z - 99;
    y = x1 / t;
    x2 = 88 * y;
    t1 = 88 - y;
    z = x2 / t1;
    …
}
```

# Exceptions



- Diverse types of error may occur in program execution
  - overflow, type error, segment faults, divide by zero, …
  - Example

```
int a = 9;
int b = 3;
…
… = 10 / (a – b * 3);
```

  - **Exceptions** are such errors detected at run time.
- What would happen if your program ignores exceptions?
  - Errors will eventually cause *low-level message* (from O/S or hardware) to be printed and to terminate the program execution.
  - Low-level message from Linux

```
$ a.out
Abort (core dump)
```

# Exceptions



- What is the problem with low-level messages?
  - They do not provide sufficient information about the error that caused your program to end.
  - They may even produce an unpredictable result or cause unexpected damage to your system. → *sudden crash of an aviation control system?*
- Alternative solution: *use test code defined by languages or users*

```
test_result = foo(a,b,c);
if (test_result is error) raise exception;
```

  - When an exception is raised, the normal program control is interrupted and the control is transferred to an **exception handler**, a special routine that handles the exception.
  - Errors are controlled by the user, so they can be led to safer, predictable and user-guided states.

# Exception handling models



```
foo (int i, char c) {
        float a[10];
        …
        if (error occurs)
                raise exception(error-type);
                …
}
```

```
exception_handler {
        switch (error-type) {
                case 1:  handler1 …
                case 2:  handler2 …
                …
                case n:  handlern …
        }
}
```

**resumption model** — continue

**termination model** — abort

error analysis
error report/print
error correction

- Exception handling makes programs robust & reliable.
- But, it may be tedious because it needs to test possible errors.
- This might be inefficient if errors occurs infrequently.

# Exception handling in C++



- C++ originally had no explicit support for exceptions.
- In 1990, the ANSI C++ accepted the exception handling.
  - It provides a programming construct with **three keywords** for exception handlers : **`try`**, **`catch`**, **`throw`**
  - Example

```
void foo(…) {
  …
  try {
    … // code that is expected to raise a exception
    throw expr // raise an exception with actual parameter
  }
  catch(type1 var1) { // a single formal parameter
    … // code for an exception handler1
  }
  …
  catch(typen varn) { // a single formal parameter
    … // code for an exception handlern
  }
  …
}
```

try block

handler block

handler match

type(*expr*) == *type$_i$*?

sequential search

jump to matched handler

resume execution from the first instruction following the try/handler block after exception is handled

# Example: exception handling in C++



a derived class of standard library class **exception**

```
#include <cstdlib>
#include <iostream>
#include <stdexcept>
using std;

class Divide_by_0:
      public runtime_error {
 public:
   Divide_by_0() :
   runtime_error("No, you cannot!") {}
};
```

```
Give two integers: 10  3
Can I divide 10 with 3?
Yes, you can divide 10 by 3.

Give two integers: 11 0
Can I divide 11 with 0?
Exception: No, you cannot!

Give two integers: …
```

functional call expression

```
void is_dividable(int a, int b) {
   if (!b) throw Divide_by_0();
   cout << "Yes, you can divide ";
}
int main() {
  int x, y;
  cout << "Give two integers:";
  while (cin >> x >> y) {
    cout << "Can I divide " <<
      x << " with " << y << "?\n";
    try {
      is_dividable(x, y);
      cout << x << " by " << y
           << ".\n";
    }
    catch(Divide_by_0& d) {
       cout << "Exception: "
         << d.what() << endl;
    }
    cout << "\nGive two integers:";
  }
  return 0;   // normal termination
}
```

Recall: a class = a type