

Subprograms

Programming Languages

CS 214



Categorizing Functions

Recall: The *function* set constructor: $f(D) \rightarrow R$
can be used to describe the operations in a language.

This approach categorizes functions _____
_____.

Example: C++ lets us use function notation to *cast*...

```
int( real )  $\rightarrow$  int
```

```
double( int )  $\rightarrow$  real
```

But if we write a *round()* function:

```
int round( double value ) { return int( value + 0.5 ); }
```

then *round()* is also a member of: _____

and *int()* and *round()* obviously *behave* very differently...



Functions: as *Mapping Rules*

The function constructor defines a function's _____
(i.e., it's domain- and range-sets), but not its _____
(i.e., indicate the domain-to-range element mappings).

Behavior can be defined via a *domain-to-range mapping rule*:

Example: In C++, we can *specify* that: $abs(int) \rightarrow int$
but to define the *behavior* of $abs()$, we need a rule:

$$abs(v) = \left\{ \begin{array}{l} \text{_____} \\ \text{_____} \end{array} \right.$$

A *mapping rule* must specify the range-value for each domain-value for which the function is defined.



Functions: as *Algorithms*

An alternative way to specify behavior is to specify:

- the function's _____
- the function's _____
- a _____ for computing the result, using the parameters.

```
-- Ada
function abs(val: in float)
    return float is
begin
    if val >= 0.0 then
        return val;
    else
        return -val;
    end if;
end abs;
```

```
"Lisp"
(defun abs (val)
  (if (>= val 0)
      val
      (- 0 val) ))
```

```
"Smalltalk (Number method)"
abs
    self >= 0
    ifTrue: ^self
    ifFalse: ^(0 - self).
```

Some like to view a HLL as a _____.



Functions and Operators

Most *functions* can be defined as *operators*, and vice versa.

Example: Ada provides an exponentiation operator _____
where C++ provides an exponentiation function _____.

So a 3rd-order polynomial can be expressed in C++ as

```
y = a * pow(x,3) + b * pow(x,2) + c * x + d;
```

or in Ada as:

```
y = a * x ** 3 + b * x ** 2 + c * x + d;
```

Superficially, functions and operators are *equivalent*:

- The _____ of a function \equiv the _____ of an operator.
- A *function* can be thought of as a _____.



Functions: as *Abstractions*

Others prefer to view functions as an *abstraction mechanism*:

- the ability to _____...

Example: If a library provides a *summation()* function, it might use any of these algorithms:

```
// iterative algorithm
int summation(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++)
        result += i;
    return result;
}
```

```
// recursive algorithm
int summation(int n) {
    if (n >= 2)
        return n + summation(n-1);
    else
        return 1;
}
```

```
// using Gauss' formula
int summation(int n) {
    return n * (n+1) / 2;
}
```

The name *summation()* is an *abstraction* that hides the details of the particular algorithm it uses.



Functions: as *Subprograms*

Imperative HLLs divide functions into two categories:

- _____: subprograms that map: $(P_1 \times P_2 \times \dots \times P_n) \rightarrow \emptyset$
- _____: subprograms that map: $(P_1 \times P_2 \times \dots \times P_n) \rightarrow R \neq \emptyset$

There are no standard names for these categories:

| HLL | $(D) \rightarrow \emptyset$ | $(D) \rightarrow R$ |
|----------|-----------------------------|---------------------|
| C/C++ | void function | function |
| Fortran | subroutine | function |
| Pascal | procedure | function |
| Modula-2 | proper procedure | function procedure |
| Ada | procedure | function |

We will describe subprograms mapping $(D) \rightarrow R$ as *functions*,
and describe subprograms mapping $(D) \rightarrow \emptyset$ as *procedures*.



Functions: as *Messages*

OO languages view functions as _____.

The *receiver* of a message executes its _____.

– The result is controlled by the _____, not the *sender*.

Different OO languages use different syntax for messages...

Example: To find the length of *anArray*, we send it a message:

```
// C++  
anArray->length()
```

```
// Java  
anArray.length
```

```
// Smalltalk  
anArray size
```

Example: To find the length of *aString*, we send it a message:

```
// C++  
aString->length()
```

```
// Java  
aString.length()
```

```
// Smalltalk  
aString size
```

Messages are something like _____...



Subprogram Mechanisms

To have a subprogram mechanism, a language must provide:

- A means of _____ the subprogram (specifying its *behavior*);
- A means of _____ the subprogram (or *activating* it).

In programming languages, to *define* a thing is to:

- _____; and
- _____.

Example: This is a C++
subprogram *definition*:
because it:

```
int summation(int n) {  
    return n * (n+1) / 2;  
}
```

- (i) reserves storage (for the function's code); and
- (ii) binds the name *summation* to the first address in that storage.



Definitions vs. Declarations

Where a *definition* binds a name to *storage*,
a _____ binds a name to a _____.

Example: This is a
C++ *declaration*:

```
int summation(int n);
```

because it tells the compiler this about *summation*:

allowing the compiler to type-check calls to the function.

For a *variable*, declaration and definition are _____...

```
int result;
```

This statement reserves a word of memory, and
binds the name *result* to the address of that word.

For *subprograms*, declaration and definition _____.



C/C++ Function Pointers

Implication of a function *definition*:

a C/C++ function's name is a _____.

Example: If *summation* and *factorial* are two functions:

```
int summation(int n) { return n * (n+1) / 2; }  
int factorial(int n) { ... definition of factorial ... }
```

then we can *declare a pointer type*:

```
typedef int * fptr(int);
```

use it to *define a pointer array*:

```
fptr fTable[2];
```

initialize our array:

```
fTable[0] = summation;  
fTable[1] = factorial;
```

and then *call either function*:

```
cout << fTable[i] (n);
```

Classes use a similar table for _____.



Subprogram Definitions

To allocate a subprogram's storage, 4 items are needed:

1. Its _____ (*data* storage for values sent by the caller);
2. Its _____ (*data* storage for the return value);
3. Its _____ (*data* storage for local variables); and
4. Its _____ or statements (*executable code* storage).

These are all provided by a subprogram's *definition*.

By contrast, a subprogram's *declaration* requires only:

1. Its _____ (i.e., its domain-set D); and
2. Its _____ (i.e., its range-set R)

This _____: $f(D) \rightarrow R$

lets the compiler check *calls* to the function for correctness.



Imperative Examples

Consider these imperative function definitions:

```
// C++  
void swap(int & a, int & b) {  
    int t = a; a = b; b = t;  
}
```

```
-- Ada  
procedure swap(a, b: in out integer) is  
    integer t;  
begin  
    t := a; a := b; b := t;  
end swap;
```

In each case, we have: _____

This allows the compiler to check that in calls: `swap(x, y);`
the arguments x and y are compatible with the parameters.



Subprograms: Lisp and Smalltalk

A Lisp subprogram definition uses the _____ function:

```
"Lisp"  
(defun factorial (n)  
  if (< n 2)  
    1  
    (* n (factorial (- n 1) )) )
```

When evaluated, *defun* parses the function that follows it and (assuming no errors) creates a symbol table entry for it.

A Smalltalk subprogram must be _____:

```
"Smalltalk Integer method"  
factorial  
  | result |  
  result := 1.  
  2 to: self  
    do: [:i | result:= result * i].  
  ^result
```

On an *accept event*, Smalltalk parses the method and (assuming no errors) creates a symbol table entry for it.



Calling Subprograms

In most languages, a subprogram is called by _____.

```
// C++  
swap(x, y);
```

```
-- Ada  
swap(x, y);
```

```
(* Modula-2 *)  
swap(x, y);
```

```
* Fortran  
CALL swap(x, y);
```

Fortran subroutines
must be called with
the *CALL* keyword.

Lisp functions must be called

(following an o-parenthesis):

```
"Lisp"  
(setq answer (factorial n) )
```

Smalltalk requires that a
message be sent to an object:

```
"Smalltalk"  
answer := 5 factorial
```



Issue: Parameterless Subprograms

Must parentheses be given at calls to parameterless functions?

- C/C++: _____

```
doSomething();
```

() is the *function-call operator*;
jumps to address preceding it

- Modula-2: _____

```
doSomething;
```

() delimits arguments

- Lisp: _____

```
(doSomething)
```

() delimits function calls

- Ada: _____

```
doSomething;
```

() delimits arguments (syntax)

- Fortran: _____

```
CALL doSomething
```

() delimits arguments

- Smalltalk: _____

```
obj doSomething
```

no method has 0 parameters...



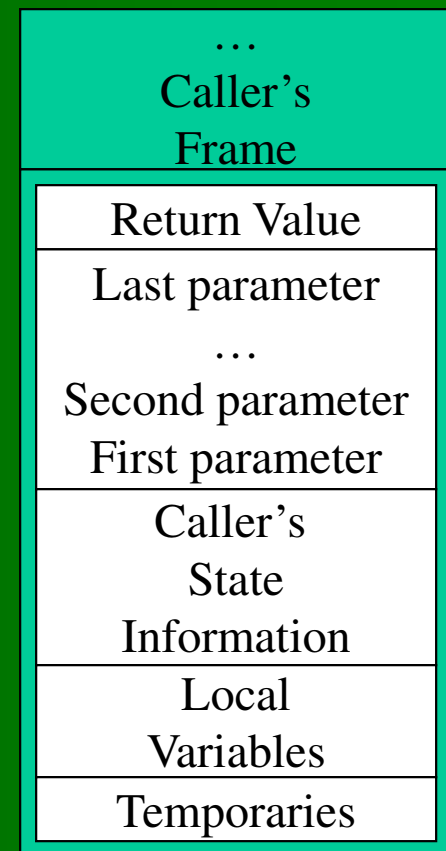
Activations

An *activation* is _____, and involves 3 steps:

- Space for the subprogram's data values is allocated on a special *run-time stack*;
- The caller's arguments are associated with the subprogram's parameters;
- Control is transferred from the caller to the starting address of the subprogram.

On Unix systems, the run-time stack grows “downward”

The space for one subprogram's data is called a *stack frame*, or _____.



run-time stack



Why a Stack?

Consider a *recursive* subprogram:

When called: $sum(3)$

$sum(3)$ calls: $sum(2)$

$sum(2)$ calls: $sum(1)$

$sum(1)$ returns 1 to: $sum(2)$

$sum(2)$ returns $2+1$ to: $sum(3)$

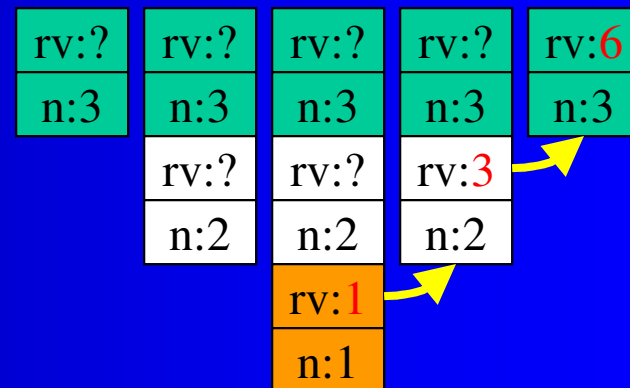
$sum(3)$ returns $3+3$ to its caller.

The call-sequence uses _____
behavior, so a *stack* is the appropriate data structure.

Each activation's parameters (n) and locals must be kept distinct.

A stack is necessary in _____

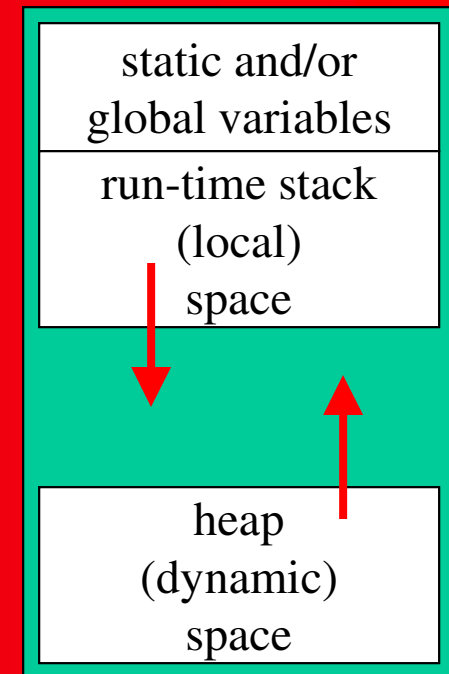
```
// C++
int sum(n) {
    if (n > 1)
        return n + sum(n-1);
    else
        return 1;
}
```



Memory Layout

On Unix systems, a program's data space is laid out something like this:

- Space for *static/global variables*
- The *run-time stack* for locals, parameters, etc.
- The *heap* for dynamically allocated variables.



This flexible design uses memory efficiently:

A typical program only runs out of memory if

- its *stack overruns its heap* (_____), or
- its *heap overruns its stack* (_____).



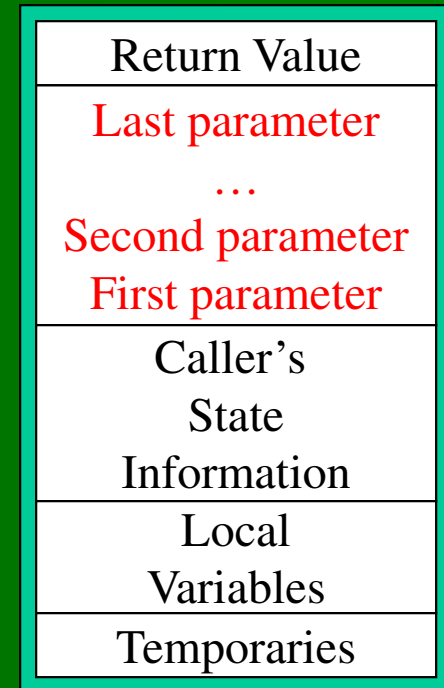
Parameter Passing

Parameters are allocated space _____
_____ on the run-time stack.

Before control is transferred to the subprogram, the call's arguments are “associated with” these parameters.

Exactly how arguments get associated with parameters depends on the *parameter passing mechanism* being used.

There are *four* general mechanisms: _____



Call-by-Value Parameters

- ... are _____ into which their arguments are _____.
- Changing a parameter doesn't affect its argument's value.
 - This is the *default* mechanism in most languages.
 - This is the *only* mechanism in C, Lisp, Java, Smalltalk, ...

// C++

```
int summ (int a, int b) {  
    return (a+b) * (b-a+1) / 2;  
}
```

-- Ada

```
function summ (a, b: in integer)  
    return integer is  
begin  
    return (a+b) * (b-a+1) / 2;  
end summ;
```

"Lisp"

```
(defun summ (a b)  
  (/ (* (+ a b) (+ (- b a) 1))  
     2) )
```

"Smalltalk Integer method"

```
summ: b  
    ^(self+b) * (b-self+1) / 2
```

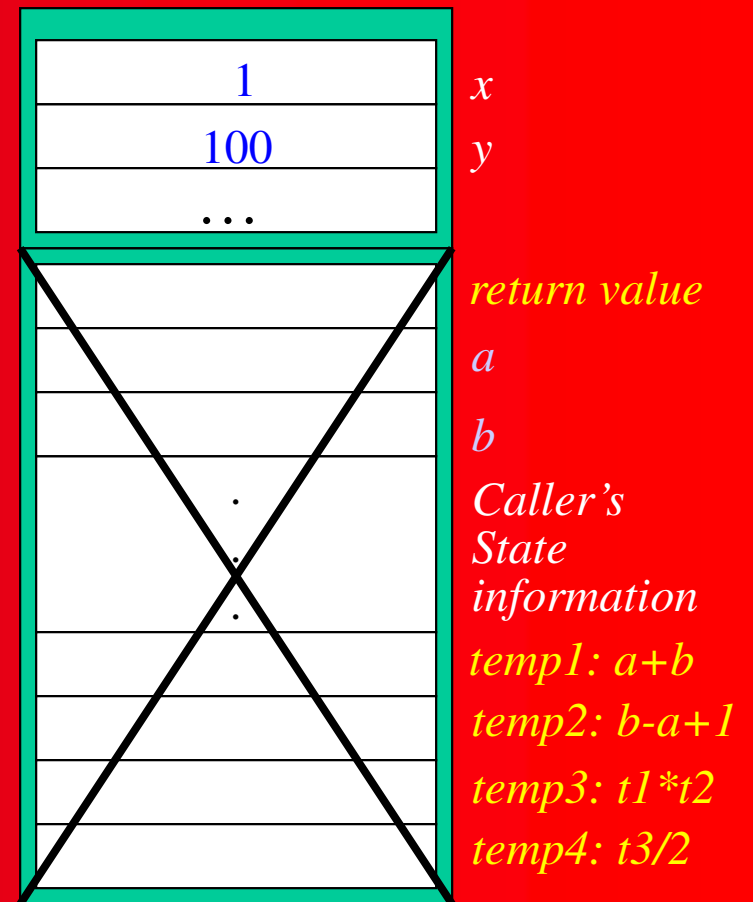
In Ada, *in* is optional, but is considered good programming style.



When function *summ()* is called

```
// C++  
total = summ(x, y);
```

- An activation record for *summ()* containing space for *a* and *b* is pushed onto the run-time stack.
- The arguments are evaluated and copied into their parameters.
- Control is transferred to *summ()* which executes and computes its return-value.
- *summ()*'s AR is popped, and control returns to the caller which retrieves the return-value from just “above” its stack-frame.



Call-by-Reference Parameters

... are _____ storing _____,
that are auto-dereferenced whenever they are accessed.

- The parameter is an *alias* for the argument.
- Changing the parameter's value changes the argument's value.

```
// C++  
void swap (int& a, int& b) {  
    int t = a; a = b; b = t;  
}
```

```
-- Ada  
procedure swap (a, b: in out integer)  
is t: integer;  
begin  
    t:= a; a:= b; b:= t;  
end swap;
```

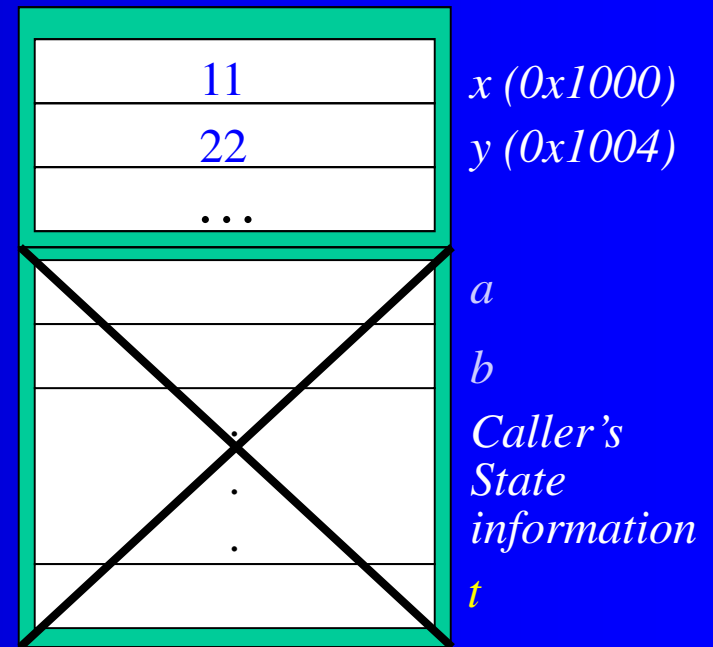
Smalltalk and Lisp
implicitly provide
call-by-reference,
because “variables”
are actually pointers
to dynamic objects.
Java is complicated...



When *swap()* is called

```
// C++  
swap(x, y);
```

- An activation record for *swap()* containing space for *a* and *b* is pushed onto the run-time stack.
- The *addresses* of the arguments are stored into their parameters.
- Control is transferred to *swap()* which executes, automatically dereferencing accesses to *a* and *b*.
- The RTS is popped, control returns to the caller, and the original values of *x* and *y* have been overwritten with new values.



Implementing Call-by-Reference?

Stroustrup's first C++ "compiler" just produced C code, so if C only provides the call-by-value mechanism, how can it handle the C++ call-by-reference mechanism?

```
// C++  
swap(x, y);
```

```
// C++  
void swap (int& a,  
           int& b);
```

```
// C++  
void swap (int& a,  
           int& b)  
{   int t = a;  
    a = b;  
    b = t;  
}
```

1. At the call, replace arguments with addresses:

```
/* C */  
swap(&x, &y);
```

2. In the declaration and definition, replace reference parameters with pointers:

```
/* C */  
void swap (int* a,  
           int* b);
```

3. Within the function definition, dereference each access to the parameter

```
/* C */  
void swap (int* a,  
           int* b)  
{   int t = *a;  
    *a = *b;  
    *b = t;  
}
```

Any compiler can implement call-by-reference this way.



Call-by-Copy-Restore Parameters

- ... store _____ both the value and address of their arguments.
- Within the subprogram, parameter accesses _____ use the local value
 - When the subprogram terminates, the local value is _____ copied back into the corresponding argument.
 - More time-efficient than call-by-reference for *heavily-used parameters* (avoids slow pointer-dereferencing).
 - Ada's _____ in-out parameters *may* use copy-restore...

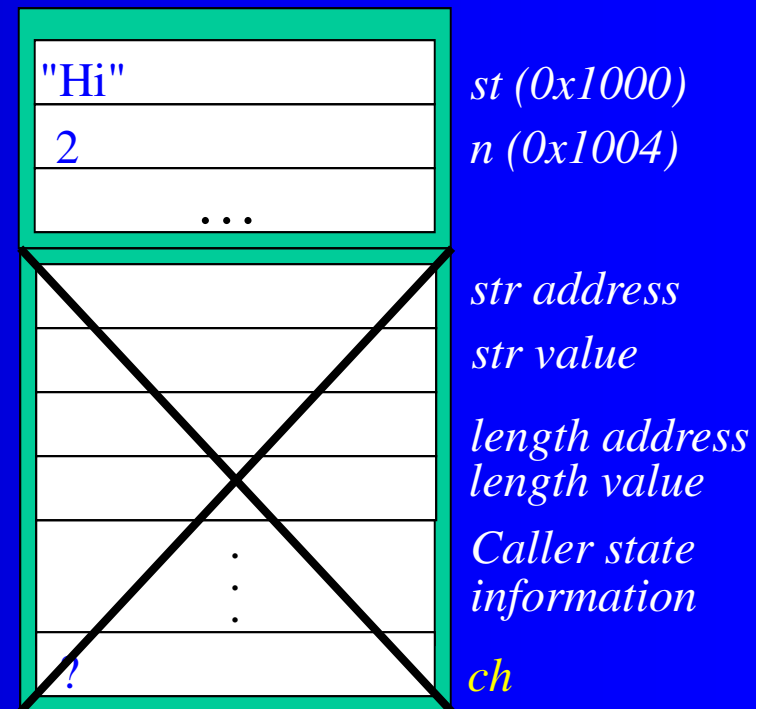
```
procedure get (str: in out ubString; length in out integer) is
  ch: character;
begin
  length:= 0; str:= ""; get(ch);
  while not End_Of_Line loop
    str:= str + ch;
    length:= length + 1;
    get(ch);
  end get;
```



When *get()* is called

```
-- Ada  
get (st, n);
```

- An activation record for *get()* containing space for the *data* and *address* of both *str* and *length* is pushed onto the run-time stack.
- Argument *values* and *addresses* are written to their parameters.
- Control is transferred to *get()* which executes, accessing only local values *str* and *length*.
- The original values of arguments *st* and *n* are overwritten with the values of parameters *str* and *length*, the RTS is popped, and control returns to the caller.



Aliasing

Copy-restore parameters behave the same as reference parameters, so long as the parameter is not an alias for a non-local that is accessed within the same subprogram.

Example:

Suppose we
have this
subprogram:

```
procedure aliasExample (param: in out integer) is
begin
  param:= 1;
  a:= 2;
end get;
```

and we execute:

```
a:= 0;
aliasExample(a);
put(a);
```

What is output, if *param* uses:

- call-by-reference?
- call-by-value-restore?

To avoid this, Ada forbids aliasing.



Call-by-Name Parameters

1. *Copy the body of the subprogram;*
2. *In the copy, substitute the arguments for the parameters;*
3. *Substitute the resulting copy for the call;*

The result is the *call-by-name mechanism* (aka macro-substitution).

```
/* C */  
#define SWAP (a, b) { int t = a; a = b; b = t; }
```

```
// C++  
inline void swap (int& a, int& b) { int t = a; a = b; b = t; }
```

- Call-by-name originated with *Algol-60*.
- By replacing the function-call with the altered body, call-by-name:
 - improves time-efficiency by eliminating the call and the RTS overhead; but
 - decreases space-efficiency by increasing the size of the program.



At each call to *swap()*

```
// C++ call to swap()  
swap(w, x);
```

```
// C++ call to swap()  
swap(y, z);
```

- The compiler makes a *copy* of the body of the function.

```
{ int t = a; a = b; b = t; }
```

```
{ int t = a; a = b; b = t; }
```

- In it, the compiler *substitutes arguments for parameters*.

```
{ int t = w; w = x; x = t; }
```

```
{ int t = y; y = z; z = t; }
```

- The compiler *substitutes the resulting body for the call*.

```
// C++ call to swap()  
{ int t = w; w = x; x = t; }
```

```
// C++ call to swap()  
{ int t = y; y = z; z = t; }
```

The resulting code is larger, but without the overhead of pushing a stack-frame, setting parameters, ... it runs faster.



Macro-Substitution Anomaly

Suppose we have defined this C macro:

```
#define SWAP (a, b) { int t = a; a = b; b = t; }
```

a and *i* are as follows:

and we call:

What we expect is:

but what we get is:

What happened? Our call:

is replaced by:

Tracing, we see:

t

| |
|---|
| 2 |
|---|

i

| |
|---|
| 2 |
|---|

a

| | | | | |
|----|----|----|----|----|
| 11 | 22 | 33 | 44 | 55 |
|----|----|----|----|----|

SWAP(*i*, *a*[*i*]);

i

| |
|--|
| |
|--|

a

| | | | | |
|----|----|--|----|----|
| 11 | 22 | | 44 | 55 |
|----|----|--|----|----|

bus error: core dumped

SWAP(*i*, *a*[*i*]);

{ *int t = i*; *i = a*[*i*]; *a*[*i*] = *t*; }

i

| |
|----|
| 33 |
|----|

a[*i*] → 33 → bus error

Because of such unexpected results, the use of macro-substitution (`#define`) for call-by-name is discouraged.



What About *inline*?

Suppose we have defined this C++ *inline* function:

```
inline void swap (int& a, int& b) { int t = a; a = b; b = t; }
```

a and *i* are as follows:
and we call:

| | | | | | | | |
|----------|---|----------|----|----|----|----|----|
| <i>i</i> | 2 | <i>a</i> | 11 | 22 | 33 | 44 | 55 |
|----------|---|----------|----|----|----|----|----|

`swap(i, a[i]);`

What we expect is:
and we get:

| | | | | | | | |
|----------|----|----------|----|----|---|----|----|
| <i>i</i> | 33 | <i>a</i> | 11 | 22 | 2 | 44 | 55 |
|----------|----|----------|----|----|---|----|----|

| | | | | | | | |
|----------|----|----------|----|----|---|----|----|
| <i>i</i> | 33 | <i>a</i> | 11 | 22 | 2 | 44 | 55 |
|----------|----|----------|----|----|---|----|----|

What happened? Our call:
is replaced by:

`swap(i, a[i]);`

```
{int* t1 = &i; int* t2 = &a[i];  
  int t = *t1; *t1 = *t2; *t2 = t;}
```

Since *a[i]* has a reference parameter, its address is computed
and stored (in *t2*), and _____ changes to *i* do not affect *t2*.

Call-by-name (via *inline*) is _____ safe in C++.



Summary

There are two broad categories of subprograms:

- procedures: that map: $(P_1 \times P_2 \times \dots \times P_n) \rightarrow$ null
- functions: that map: $(P_1 \times P_2 \times \dots \times P_n) \rightarrow$ R not equal to null

When a subprogram is *called*, an activation record containing space for its variables is pushed onto the runtime stack.

The four parameter-passing mechanisms are: Call-by-_____

- value stores a copy of the argument.
- reference stores the address (reference) of the argument and auto-dereferences all accesses to the parameter.
- copy-restore stores a copy and the address of the argument, and replaces the argument's value with the copy's value on termination.
- name makes a copy of the function, replaces the parameter in the copy with the argument, and then replaces the call with that copy.

