Programming Languages

Lecturer: William W.Y. Hsu
Chapter 8 – Subroutines and Control Abstraction
Abstraction

• Programming languages support the binding of names with potentially complex program fragments that can be used through an interface.
• Programmers only need to know about the purpose of the fragment rather than its implementation => Abstraction.
• A control abstraction performs a well-defined operation.
  ▫ Subroutines
• A data abstraction represents information
  ▫ Data structures.
  ▫ Most data structures include some number of control abstractions.
Subroutine Calls
Subroutines

- Execute an operation on behalf of a calling program unit.
- Subroutines can be parameterized.
  - The parameters in the definition of the function are known as formal parameters.
  - The parameters passed in the subroutine call are known as actual parameters or arguments.
  - At the time of the call, actual parameters are mapped to formal parameters.
- Functions are subroutines that return a value.
Subroutine Frames

- Each subroutine requires a **subroutine frame** (a.k.a. **activation record**) to keep track of:
  - Arguments and return values.
  - Local variables and temporaries.
  - Bookkeeping in formation.

- When a subroutine returns, its frame is removed.

```plaintext
1001: A(3)  Actual Parameters
...
2001: int A(int n) {
    int m = n * n;
    return m + A(n-1);
}
```

**Formal Parameters**
Subroutine Frames

- Activation record (subroutine frame) is used to store all related information for the execution of a subroutine.
- Before a subroutine is executed, the frame must be set up and some fields in the frame must be initialized:
  - Formal arguments must be replaced with actual arguments.
- This is done in the calling sequence, a sequence of instructions before and after a subroutine, to set-up the frame.
Subroutine Frames
Subroutine Frames

- Stack Pointer \( sp \)
  - Top of frame stack
- Frame Pointer \( fp \)
  - Access to arguments and locals via offset of \( fp \).
- The differ if temporary space is allocated in stack.
Subroutine Frames

- Stack Pointer  $sp$
  - Top of frame stack
- Frame Pointer  $fp$
  - Access to arguments and locals via offset of $fp$.
- The differ if temporary space is allocated in stack.
Review of Stack Layout

- Allocation strategies:
  - **Static**
    - Code.
    -Globals.
    - Own variables.
    - Explicit constants (including strings, sets, other aggregates).
    - Small scalars may be stored in the instructions themselves.
  - **Stack**
    - Parameters.
    - Local variables.
    - Temporaries.
    - Bookkeeping information.
  - **Heap**
    - Dynamic allocation.
Review of Stack Layout

- Contents of a stack frame:
  - Bookkeeping.
    - Return PC (dynamic link).
    - Saved registers.
    - Line number.
    - Saved display entries.
    - Static link.
  - Arguments and returns.
  - Local variables.
  - Temporaries.
Calling Sequences

- Common strategy is to divide registers into *caller-saves* and *callee-saves* sets.
  - Caller uses the "callee-saves" registers first.
  - "Caller-saves" registers if necessary.
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time.
  - Some storage layouts use a separate arguments pointer.
  - The VAX architecture encouraged this.
Saving and Restoring Registers: The Solution

- Solution 1: Save/restore in the calling sequence at caller.

```c
main()
{
    ...
    R1=10
    ...
    T1 = R1  // Calling
    call foo();  // sequence
    R1 = T1    // code
    ...
    R2 = R1
}
```

- Solution 2: Save/restore in prologue and epilogue at callee.

```c
foo()
{
    ...
    T1 = R1  // Prologue
    ...
    R1=20
    ...
    R1 = T1  // Epilogue
    return
}
```
Saving and Restoring Registers

- The compiler should generate code only to save and restore registers that matters.
  - If a subroutine does not use R3, R3 does not need to be saved in the calling sequence.
- Ideally, we should only save registers that is used in the caller and the callee.
  - Difficult due to separate compilation: no information of callee when compiling caller, and vice versa.
- Simple solution (with unnecessary save/restore):
  - Option 1: caller saves/restores all registers it uses.
  - Option 2: callee saves/restores all registers it uses.
- Compromised solution:
  - Partition registers into two sets, one for caller save one for callee save.
Calling Sequence

- On procedure call and return compilers generate code that execute to manage the runtime stack.
  - **Setup** at call to procedure $\texttt{foo}(a, b)$.
  - **Prologue** before foo code executes.
  - **Epilogue** at the end of foo code.
  - “**Teardown**” right after calling the code.
Setup $\text{foo}(a,b)$

1. **Move sp** to allocate a new stack frame.
2. **Copy args** $a,b$ into frame.
3. **Copy return** address into frame.
4. **Set fp** to point to new frame.
5. **Maintain static chain** or display.
6. **Move PC** to procedure address.
Prologue

1. Copy registers into local slots
2. Object initialization.
   - Objects that are used are initialized.
Epilogue

1. **Place return value** into slot in frame.
2. **Restore registers.**
   - Registers stored from “foo”’s subroutine are registered.
3. **Restore PC** to return address.
   - The program resumes from where it began.
“Teardown”

1. **Move sp & fp** (deallocate frame)
2. **Move return** values (if in registers)
   - If the return value was placed in a register, put it in the stack.
A Question

• Why local variables typically do not have a default value (while globals do)?

```cpp
int I;

main()
{
    int j;
    cout << I << j;
}
```
Hardware Support for Efficient Subroutine Execution

• Calling sequences are overheads to running subroutines.

• Register windows.
  ▫ Introduced in Berkeley RISC machines.
  ▫ Also used in Sun SPARC and Intel Itanium processors.
  ▫ Basic idea:
    • Maintain multiple sets (a window) of registers.
    • Using a new mapping (set) of registers when making subroutine calls.
      • Set and reset a mapping is cheaper than saving and restoring registers.
    • New and old mapping registers overlaps to allow parameter passing.
Language Support for Efficient Subroutine Execution

- In-line functions
  
  ```
  inline int max(int a, int b) {return a > b ? a : b;
  }
  ```

- How is it different from ordinary functions?
  - Such functions are not real functions, the routine body is expanded in-line at the point of call.
    - A copy of the routine body becomes a part of the caller.
    - No actual routine call occurs.

- Will inline function always improve performance?
  - Maybe, maybe not!
    - Many other factors: e.g. code size affecting cache/memory behavior.
Parameter Passing
Parameter Passing

- **Pass-by-value**: Input parameter.
- **Pass-by-result**: Output parameter.
- **Pass-by-value-result**: Input/output parameter.
- **Pass-by-reference**: Input/output parameter, no copy.
- **Pass-by-name**: Textual substitution.
- **Pass-by-sharing**: Values in the language are based on objects.

- Many languages (e.g., Pascal) provide value and reference directly.
Parameters

- First some definitions
- **Formal parameters**
  - Lisp: `(lambda (a b) (/ (+ a b)))`
  - C function:
    ```c
    float ave(float a, float b) { return (a+b)/2.0; }
    ```
- **Actual parameters**
  - Lisp function arguments: `(ave x 10.0)``
  - C function arguments: `ave(x, 10.0)`
- **Versus operands** (of operators and special forms)
  - Lisp special forms: `(if flag "yes" "no")`
  - C operators: `x > 0 && flag`
  - Operand handling often depends on the type of built-in operator.
Parameter Passing

- **C/C++**: functions
  - Parameters passed by value (C)
  - Parameters passed by reference can be simulated with pointers (C)
    ```c
    void proc(int* x, int y) { *x = *x + y } ...
    proc(&a, b);
    ```
  - Or directly passed by reference (C++)
    ```c
    void proc(int& x, int y) { x = x + y }
    proc(a, b);
    ```
Parameter Passing

- In a language with a reference model of variables (Lisp, Clu), pass by reference (sharing) is the obvious approach.
- It's also the only option in Fortran.
  - If you pass a constant, the compiler creates a temporary location to hold it.
  - If you modify the temporary, who cares?
- **Call-by name is an old Algol technique**
  - Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called THUNKS.
Parameter Passing

- Parameter passing mechanisms.
  - The mechanism to pass the actual parameter information to the subroutine.
    - Many potential ways, copying value, copying reference, etc
    - For different types of parameters, different parameter passing mechanisms can be used.
  - Some common parameter passing mechanisms:
    - Call by value (in)
    - Call by reference (in+out)
    - Call by result (out)
    - Call by value/result (in+out)
    - Call by name (in+out)
  - Different mechanisms used by C, Fortran, Pascal, C++, Java, Ada (and Algol 60)
Pass-by-value

```plaintext
int m=8, i=5;
foo(m);
print m;  # print 8
proc foo(byval b){
    b = i+b;
}
```
Pass-by-reference

```java
int m=8, i=5;
foo(m);
print m;  # print 13
proc foo(byval b) {
    b = i+b;
}
```
Pass-by-value-result

```c
int m=8, i=5;
foo(m);
print m;  // print 13
proc foo(byval b) {
    b = i+b;
}
```
Parameter Passing Modes in C

- **Call by value** parameter passing only: actual parameter is evaluated and its value is assigned to the formal parameter of the function being called.
- A formal parameter behaves like a local variable and can even be modified in the function without affecting the actual parameter.
- For example
  ```c
  int fac(int n)
  {
      if (n < 0) n = 1;
      return n ? n*fac(n-1) : 1;
  }
  ```
Parameter Passing Modes in C

- Objects can be modified in a function by passing pointers to the object to the function.
- For example
  
  ```c
  swap(int *a, int *b)
  { int t = *a; *a = *b; *b = t; }
  ```

  where `a` and `b` are integer pointer formal parameters, and `*a` and `*b` in the function body dereference the pointers to access the integers.

- A function call should explicitly pass pointers to integers, e.g. `swap(&x, &y)`, where `x` and `y` are integer variables and `&x` and `&y` are the addresses of their values.
Parameter Passing Modes in C

- Arrays and pointers are exchangeable in C: an array is automatically passed as a pointer to the array.

```c
int foo( int *a ){}

void main()
{
    int a[10];

    foo(a);
}
```
Parameter Passing Modes in Fortran

- Call by reference parameter passing only.
- If the actual parameter is an \textit{l-value}, e.g. a variable, its \textit{reference} is passed to the subroutine.
- If the actual parameter is an \textit{r-value}, e.g. an expression, it is evaluated and assigned to an \textit{invisible temporary variable} whose \textit{reference} is passed.
Parameter Passing Modes in Fortran

- For example
  
  ```fortran
  SUBROUTINE SHIFT(A, B, C)
  INTEGER A, B, C
  A = B
  B = C
  END
  ```

- When called with `SHIFT(X, Y, 0)` this results in `Y` being assigned to `X`, and `Y` is set to `0`.

- When called with `SHIFT(X, 2, 3)` this results in `2` being assigned to `X`. 

Emulating call-by-reference with call-by-value

• What if we want to call fortran functions from C programs or vice versa?
  ▫ We need to deal with the two different parameter passing mechanisms in the two languages.
  ▫ Must emulating call-by-reference by call-by-value

```fortran
SUBROUTINE SHIFT(A, B, C)
  INTEGER A, B, C
  A = B + C
END
```

```c
// g77 add ‘_’ to routine name automatically by default
void shift_(int *A, int *B, int *C)
{
  *A = *B + *C;
}
```

• Ask for example files. (Linux knowledge required)
Parameter Passing Modes in Pascal

- **Call by value** and **call by reference** parameter passing.
- Call by value is similar to C.
- Call by reference is indicated by `var` parameters.
- For example

  ```pascal
  procedure swap(var a:integer, var b:integer)
  var t;
  begin
    t := a; a := b; b := t
  end
  ```

  where the `var` formal parameters `a` and `b` are passed by reference (`var t` declares a local variable).
Parameter Passing Modes in Pascal

- Programs can suffer from unnecessary data duplication overhead.
- When a big array is passed by value it means that the entire array is copied.
- Therefore, passing arrays by reference instead of by value is not a bad idea to enhance efficiency.
- Do this unless the array is modified in the subroutine, because call by value will not affect actual parameter but call by reference does, which can lead to buggy code.
Comparison

• **Call-by-value**: the value of the actual parameter is copied to the formal parameter.
  ▫ For parameters with large data structures (e.g. a class with big arrays), call-by-value may result in high overheads.
    • The entire class including the arrays is copied!

• **Call-by-reference**: only the reference is copied for each parameter.
  ▫ No significant overhead for large data structures.
  ▫ But no clear distinction between in-mode or in/out-mode any longer.
    • In C++, this is fixed by the *const* qualifier.
Parameter Passing Modes in C++

- **Call by value** and **call by reference** parameter passing.
- Call by value is similar to C.
- Call by reference is indicated by using `&` for formal parameters.
- For example

```cpp
swap(int &a, int &b) {
    int t = a; a = b; b = t;
}
```
where the formal parameters `a` and `b` are passed by reference, e.g. `swap(x, y)` exchanges integer variables `x` and `y`. 
Parameter Passing Modes in C++

- Large objects should be passed by reference instead of by value to increase efficiency.
- Arrays are automatically passed by reference (like in C).
- To avoid objects to be inadvertently modified when passed by reference, `const` parameters can be used.
- For example
  ```cpp
  store_record_in_file(const huge_record &r)
  {
    ... 
  }
  ```
- Compiler will prohibit modifications of object in function.
- `const` parameters are also supported in ANSI C.
Parameter Passing Modes in Languages With Reference Model of Variables

- Smalltalk, Lisp, ML, Clu, and Java (partly) adopt reference model of variables.
- Every variable contains the memory address of the variable’s value.
- Parameter passing of variables is call by sharing in which the address of the value is passed to a subroutine.
- This is implemented similar to call by value: the content of the variable that is passed is an address to the variable’s value.
- For expressions, call by sharing passes the value of the expression.
Parameter Passing Modes in Java

- **Call by value** and **call by reference/sharing** parameter passing.
- Java adopts both value and reference models of variables.
- Variables of built-in types are passed by value.
  - Class instances are passed by sharing.
  - To pass a variable of built-in type by reference, the `&` can be used for formal parameters (like in C++).
Parameter Passing Modes in Ada

- Call by value, call by result, and call by value/result parameter passing.
- Indicated by Ada’s `in` (by value), `out` (by result), and `in out` (by value/result) modes for formal parameters.
- For example

```ada
procedure shift(a:out integer, b:in out integer, c:in integer) is
begin
  a := b; b := c;
end shift;

where a is passed out, b is passed in and out, and c is passed in.
```
Parameter Passing Modes in Ada

- **in** mode parameters can be read but not written in the subroutine.
  - Call by value, but writes to the parameter are prohibited in the subroutine.
- **out** mode parameters can be written but not read in the subroutine (Ada 95 allows read).
  - Call by result uses a local variable to which the writes are made and the resulting value is copied to the actual parameter when the subroutine returns.
Parameter Passing Modes in Ada

- **in out** mode parameters can be read and written in the subroutine.
  - Call by value/result uses a local variable that is initialized by assigning the actual parameter’s value to it and when the subroutine returns the variable’s value is copied back to the actual parameter.
Parameter Passing Modes in Ada

- The Ada compiler generates code for the example Ada `shift` procedure to implement call by value, call by result, and call by value/result with a structure that is somewhat similar to the following ANSI C function.

```c
void shift(int *a, int *b, const int c)
{
    // copy input values before start
    int tmpa, tmpb = *b, tmpc = c;
    tmpa = tmpb; tmpb = tmpc;
    // copy result value before return
    *a = tmpa; *b = tmpb;
}
```
Parameter Passing Modes in Ada

- That is, local variables are initialized with `in` mode parameters, operated on, and copied to `out` mode parameters.
- This is more efficient for simple types, because it avoids repeated pointer indirection necessary to access variables in memory that are passed by reference.
Parameter Passing Modes in Ada

- The Ada compiler may decide to use call by reference for passing non-scalars (e.g. records and arrays) to optimize the program.
- This works for in mode, because the parameter cannot be written (like using `const` in C++ with reference parameters).
- This works for out and in out modes, because the parameter is written anyway.
Parameter Passing: Ada

- Ada goes for semantics: who can do what
  - **in**: callee reads only
  - **out**: callee writes and can then read (formal not initialized); actual modified
  - **in out**: callee reads and writes; actual modified
- Ada in/out is always implemented as:
  - value/result for scalars, and either
  - value/result or reference for structured objects
- Pass-by-value is expensive for complex types, so it can be implemented by passing either values or references.
- However, programs can have different semantics with two solutions:
  - This is Illegal in Ada.
Ada Example

type t is record
    a, b : integer;
end record;

r : t;

procedure foo(s : in out t) is
begin
    r.a := r.a + 1;
    s.a := s.a + 1;
end foo;

... 
r.a := 3;
foo(r);
put(r.a); -- does this print 4 or 5?
Parameter Aliasing Problems

- An alias is a variable or formal parameter that refers to the same value location as another variable or formal parameter.
- Example variable aliases in C++:
  ```cpp
  // j refers to i (is an alias for i)
  int i, &j = i;
  ...
  i = 2; j = 3;
  cout << i;     // prints 3
  ```
Parameter Aliasing Problems

• Example parameter aliases in C++:
  
  ```
  shift(int &a, int &b, int &c)
  { a = b; b = c; }
  ```

  The result of `shift(x, y, x)` is that `x` is set to `y` but `y` is unchanged.

• Example variable and parameter aliases in C++:
  
  ```
  int sum = 0;
  score(int &total, int val)
  { sum += val; total += val; }
  ```

  The result of `score(sum, 7)` is that `sum` is incremented by 14.
Parameter Aliasing Problems

• Java adopts reference model of variables (call by sharing).
  ▫ Watch out for aliases as problems are hard to correct.
• Ada forbids parameter aliases.
  ▫ Allows compiler to use call by reference with the same effect as call by result.
  ▫ But not checked by compiler and resulting program behavior is undefined.
Call by Name Parameter Passing

- C/C++ macros (also called defines) adopt call by name.
- For example
  
  ```
  #define max(a,b) ((a)>(b) ? (a) : (b))
  ```

- A "call" to the macro replaces the macro by the body of the macro (called macro expansion), for example
  
  ```
  max(n+1, m)
  ```

  is replaced by
  
  ```
  ((n+1)>(m)?(n+1):(m))
  ```

  in the program text.

- Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameter expressions.
Call by Name Parameter Passing

• Formal parameters are often parenthesized to avoid syntax problems after expansion, for example \( \max(c?0:1,b) \) gives 
  \(((c?0:1)>(b)\?(c?0:1):(b))\) in which 
  \((c?0:1)>(b)\) would have had a different meaning without parenthesis.

• Call by name parameter passing reevaluates actual parameter expression each time the formal parameter is read.

• Watch out for reevaluation of function calls in actual parameters, for example \( \max(\text{somefunc}(),0) \) results in the evaluation of \( \text{somefunc}() \) twice if it returns a value \( >0 \).
Pass-by-name

- Arguments passed by name are re-evaluated in the caller’s referencing environment every time they are used.
- They are implemented using a hidden-subroutine, known as a thunk.
- This is a costly parameter passing mechanism.
- Think of it as an in-line substitution (subroutine code put in-line at point of call, with params substituted).
- Or, actual params substituted textually in the subroutine body for the formulas.
Pass-by-name

array A[1..100] of int;
int i=5;
foo(A[i], i);

#GOOD example
proc foo(name B, name k){
    k=6;
    B=7;
}

#text sub does this
proc foo{
    i=6;
    A[i]=7;
}

array A[1..100] of int;
int i=5;
foo(A[i]);

#BAD Example
proc foo(name B){
    int i=2;
    B=7;
}

#text sub does this
proc foo{
    int i=2;
    A[i]=7;
}
Pass-by-name

- Try evaluating: \[ y = \sum_{x=1}^{10} 3x^2 - 5x + 2 \]

- In pass-by-name:

\[
y := \text{sum}(3\cdot x\cdot x - 5\cdot x + 2, x, 1, 10)
\]

```plaintext
real proc sum(expr, i, low, high);
    value low, high;
    real expr;
    integer i, low, high;
begin
    real rtn;
    rtn := 0;
    for i := low step 1 until high do
        rtn := rtn + expr;
    sum := rtn
end sum;
```
Macro and Inline Function in C++

- Macro and inline function in C++ have similar functionality?
  - Are there any differences?

- Parameter passing mechanisms are not the same.
Parameter Passing Problems

- Call by name problem.
  - Cannot write a swap routine that always works!

```pascal
procedure swap(a, b)
integer a, b, t;
begin t := a; a := b; b := t
end swap
```

- Consider `swap(i, a[i])`, which executes

```pascal
t := i
i := a[i]  this changes i
a[i] := t  assigns t to wrong array element
```
Parameter Passing Problems

• Call by value/result problem.
  ▫ Behaves differently compared to call by reference in the presence of aliases.

```pascal
procedure shift(a:out integer, b:in out integer, c:in integer) is
begin
  a := b; b := c;
end shift;
```

• When `shift(x,x,0)` is called by reference the resulting value of `x` is 0.
• When `shift(x,x,0)` is called by value/result the resulting value of `x` is either unchanged or 0, because the order of copying out mode parameters is undefined.
Conformant Array Parameters

- Some languages support conformant array (or open array) parameters, e.g. Ada, Standard Pascal, Modula-2, and C.
- Pascal arrays have compile-time shape and size.
  - Problem: when required to sort arrays of different sizes because sort procedure accepts one type of array with one size only.
- C passes only pointers to arrays to functions and array size has to be determined using some other means (e.g. as a function parameter).
Conformant Array Parameters

- Array parameters in Standard Pascal are conformant and array size is not fixed at compile-time.
- For example:
  ```pascal
  function sum(A : array [low..high : integer] of real) : real 
  ... 
  Function sum accepts real typed arrays and low and high act like formal parameters that get the lower and upper bound index of the actual array parameter at run time.
Closures as Parameters

- Recall that a subroutine closure is a reference to a subroutine together with its referencing environment.
- Standard Pascal, Ada 95, Modula-2+3 fully support passing of subroutines as closures.
- Standard Pascal example:

```pascal
procedure apply_to_A(function f(n:integer):integer;
var A : array [low..high : integer] of integer);
var i:integer;
begin
  for i := low to high do A[i] := f(A[i])
end
```
Closures as Parameters

- C/C++ support pointers to subroutines.
- No need for reference environment, because nested subroutines are not allowed.
- Example:
  ```c
define apply_to_A(int (*f)(int), int A[], int A_size)
{
    int i;
    for (i=0; i<A_size; i++) A[i]=f(A[i]);
}
```
- The int (*f)(int) is a formal parameter that is a pointer to a function f from int to int.
Default Parameters

- Ada, C++, Common Lisp, and Fortran 90 support default parameters.
- A default parameter is a formal parameter with a default value.
- When the actual parameter value is omitted in a subroutine call, the user-specified default value is used.
- Example in C++:
  ```cpp
  void print_num(int n, int base = 10)
  ...
  ```
- A call to `print_num(35)` uses default value 10 for base as if `print_num(35, 10)` was called.
Default Parameters

- Example in Ada:

```ada
procedure put (item  : in integer;
               width : in field := default_width;
               base  : in number_base := 10)
is
  ...
```

- A call to `put(35)` uses default values for the width and base parameters.
Positional Versus Named Parameters

- Parameters are **positional** when the first actual parameter corresponds to the first formal parameter, the second actual to the second formal, etc.
  - All programming languages adopt this natural convention
- Ada, Modula-3, Common Lisp, and Fortran 90 also support **named parameters**.
- A named parameter (also called keyword parameter) names the formal parameter in a subroutine call.
Positional Versus Named Parameters

- For example in Ada:
  ```ada
  put(item => 35, base => 8);
  this "assigns" 35 to item and 8 to base, which is the same as:
  put(base => 8, item => 35);
  and we can mix positional and name parameters as well:
  put(35, base => 8);
  ```
Positional Versus Named Parameters

- Advantages:
  - Documentation of parameter purpose.
  - Allows default parameters anywhere in formal parameter list: with positional parameters, the use of default parameters is often restricted to the last parameters only, because the compiler cannot always tell which parameter is optional in a subroutine call.
Variable Number of Arguments

- C, C++, and Common Lisp are unusual in that they allow defining subroutines that take a variable number of arguments.
Variable Number of Arguments

#include <stdarg.h>

int plus(int num, ...) {
    int sum;
    va_list args; // declare list of arguments
    va_start(terms, num); // initialize list of arguments
    for (int i=0; i<=num; i++)
        sum += va_arg(args, int); // get next argument (must be int)
    va_end(args); // clean up list of arguments
    return sum;
}
Variable Number of Arguments

- Function `plus` adds a bunch of integers together, where the number of arguments is the first parameter to the function, e.g. `plus(4, 3, 2, 1, 4)` returns 10.
- Used in the `printf` and `scanf` text formatting functions in C.
- Using variable number of arguments in C and C++ is not type safe as parameter types are not checked.
- In Common Lisp, one can write `( + 3 2 1 4)` to add the integers.
Function Returns

- Most languages allow a function to return any type of data structure, except a subroutine.
- Modula-3 and Ada allow a function to return a subroutine implemented as a closure.
- C and C++ allow functions to return pointers to functions (no closures).
- Earlier languages use special variable to hold function return value.
Function Returns

- Example in Pascal:

```pascal
function max(a : integer; b : integer) : integer;
begin
  if a>b then max := a else max := b
end
```

- There is no return statement, instead the function returns with the value of `max` when it reaches end.
Function Returns

- Ada, C, C++, Java, and other more modern languages typically use an explicit return statement to return a value from a function.

- Example in C:
  ```c
  int max(int a, int b)
  { if (a>b) return a; else return b; }
  ```

- May require a temporary variable for incremental operations:
  ```c
  int fac(int n)
  { int rtn=1;
    for (i=2; i<=n; i++) rtn*=i;
    return rtn;
  }
  ```
  - Wastes time copying the temporary variable value into return slot of subroutine frame on the stack.
Function Returns

- `return` statement is more flexible, but wastes time copying the temporary variable value into return slot of subroutine frame on the stack.
Pass-by- and Call-by-

• They are the same THING!
Call-by-

- **Call-by-value**: Input parameter (Passes the actual value on ).
- **Call-by-result**: Output parameter (Passes the actual value upon conclusion of a program).
- **Call-by-value-result**: Input/output parameter (Passes actual value on start & on conclusion.
- **Call-by-reference**: Input/output parameter (Same value is maintained).
- **Call-by-name**: Textual substitution
- **Call-by-sharing**: Values in the language are based on objects
## Parameter Passing: Summary

<table>
<thead>
<tr>
<th>parameter mode</th>
<th>representative languages</th>
<th>implementation mechanism</th>
<th>permissible operations</th>
<th>change to actual?</th>
<th>alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>C/C++, Pascal, Java/C# (value types)</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>in, const</td>
<td>Ada, C/C++, Modula-3</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
<td>maybe</td>
</tr>
<tr>
<td>out</td>
<td>Ada</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>value/result</td>
<td>Algol W</td>
<td>value</td>
<td>read, write</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>var, ref</td>
<td>Fortran, Pascal, C++</td>
<td>reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing</td>
<td>Lisp/Scheme, ML, Java/C# (reference types)</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in out</td>
<td>Ada</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>name</td>
<td>Algol 60, Simula</td>
<td>closure (thunk)</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>need</td>
<td>Haskell, R</td>
<td>closure (thunk) with memoization</td>
<td>read, write*</td>
<td>yes*</td>
<td>yes*</td>
</tr>
</tbody>
</table>
gcc Calling Sequence for the x86 Architecture

- cdecl (C declaration) calling sequence.
  - Subroutine arguments are passed on the stack, in the reverse order.
  - Stack are aligned to a 16 byte boundary.
  - Calling function cleans the stack after the function call returns.
    - This allows for variable number of parameters.
  - Integer values and memory address are returned in EAX.
  - EAX, ECX, EDX are caller-saved, the rest are callee saved.
- See callingsequence.cpp and callingsequence.s
Exception Handling

• An exception is an unexpected or unusual condition that arises during program execution.
  ▫ A hardware-detected run-time error or unusual condition detected by software.
  ▫ Raised by the program or detected by the language implementation.

• Examples:
  ▫ Read a value after EOF reached.
  ▫ Arithmetic overflow.
  ▫ Wrong type for input data.
  ▫ User-defined conditions, not necessarily errors.

• Alternatives:
  ▫ Invent the value (e.g., -1)
  ▫ Always return the value and a status code (must be checked each time).
  ▫ Pass a closure (if available) to handle errors.
Exception Handling

- Exceptions move error-checking out of the normal flow of the program.
- No special values to be returned.
- No error checking after each call.
Exception Handlers

• What is an exception handler?
  ▫ Code executed when exception occurs.
  ▫ May need a different handler for each type of exception.

• Why design in exception handling facilities?
  ▫ Allow user to explicitly handle errors in a uniform manner.
  ▫ Allow user to handle errors without having to check these conditions.
  ▫ Explicitly in the program everywhere they might occur.
Exception Handlers Pioneered in PL/1

- Syntax: ON condition statement
- The nested statement is not executed when the ON statement is encountered, but when the condition occurs.
  - e.g., overflow condition
- The binding of handlers depends on the flow of control.
- After the statement is executed, the program:
  - Terminates if the condition is considered irrecoverable.
  - Continues at the statement that followed the one in which the exception occurred.
- Dynamic binding of handlers and automatic resumption can potentially make programs confusing and error-prone.
Exception Handlers

- Modern languages make exception handler lexically bound, so they replace the portion of the code yet-to-be-completed.
- In addition, exceptions that are not handled in the current block are propagated back up the dynamic chain.
  - The dynamic chain is the sequence of dynamic links.
  - Each activation record maintains a pointer to its caller, a.k.a., the dynamic link.
  - This is a restricted form of dynamic binding.
Exception Handlers

- Java uses lexically scoped exception handlers.

```java
try{
    int a[] = new int[2];
    a[4];
} catch (ArrayIndexOutOfBoundsException e){
    System.out.println("exception: "+
                     e.getMessage());
    e.printStackTrace();
}
```
Exception Handlers: Use of Exceptions

- Recover from an unexpected condition and continue.
  - e.g., request additional space to the OS after out-of-memory exception.
- Graceful termination after an unrecoverable exception.
  - Printing some helpful error message
  - e.g., dynamic Link and line number where the exception was raised in Java.
- Local handling and propagation of exception.
  - Some exception have to be resolved at multiple level in the dynamic chain.
  - e.g., Exceptions can be reraised in Java using the throw statement.
Returning Exceptions

• Propagation of exceptions effectively makes them return values.
• Consequently, programming languages include them in subroutine declarations.
  ▫ Modula-3 requires all exceptions that are not caught internally to be declared in the subroutine header.
  ▫ C++ makes the list of exception optional.
  ▫ Java divides them up into **checked** and **unchecked** exceptions.
Hierarchy of Exceptions

- In PL/1, exceptions do not have a type.
- In Ada, all exceptions are of type exception.
  - Exception handler can handle one specific exception or all of them.
- Since exceptions are classes in Java, exception handlers can capture an entire class of exceptions (parent classes and all its derived classes).
  - Hierarchy of exceptions.
Implementation

- Linked-list of dynamically-bound handlers maintained at runtime.
  - Each subroutine has a default handler that takes care of the epilogue before propagating an exception.
  - This is slow, since the list must be updated for each block of code.
- Compile-time table of blocks and handlers.
  - Two fields: starting address of the block and address of the corresponding handler.
  - Exception handling using a binary search indexed by the program counter.
  - Logarithmic cost of the number handlers.
Java

- Each subroutine has a separate exception handling table.
  - Thanks to independent compilation of code fragments.
- Each stack frame contains a pointer to the appropriate table.
C

- Exception can be simulated.
- `setjmp()` can store a representation of the current program state in a buffer.
  - Returns 0 if normal return, 1 if return from long jump.
- `longjmp()` can restore this state.

```c
if(!setjmp(buffer)){
    /* protected code */
} else {
    /* handler */
}
```
C

- The state is usually the set of registers.
- `longjmp()` restores this set of registers.
- Is this good enough?
- Changes to variables before the long jump are committed, but changes to registers are ignored.
- If the handler needs to see changes to a variable that may be modified in the protected code, the programmer must include the `volatile` keyword in the variable’s declaration.
Generics, Polymorphism

- Polymorphism is the property of code working for arguments/data of different types.
  - Sort (list) works for list of int, list of string.
- ML allows this but at cost... (dun dun dun) dynamic type checking.
- Generics, templates allow static type checking but some measure of polymorphism.
Generics, Polymorphism

generic compare (x, y: type T) returns bool{
    return x < y;
}
...
Creal = new compare(T=real);
Cint = new compare(T=int);
Cstr = new compare(T=string);
..
Generic inc(a: type T) returns T{
    return a + 1;
}
Cint = new inc (T=int);
Cstr = new inc (T=string);  #NO... Compiler reject
Coroutines

• Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.

• Coroutines can be used to implement
  ▫ iterators (Section 6.5.3)
  ▫ threads (to be discussed in Chapter 12)

• Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack.
Coroutines

Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)
First-, Second-, and Third-Class Subroutines

- **First-class object**: an object that can be passed as a parameter, returned from a subroutine, and assigned to a variable.
  - Primitive types such as integers in most programming languages.
- **Second-class object**: an object that can be passed as a parameter, but not returned from a subroutine or assigned to a variable.
  - Arrays in C/C++.
- **Third-class object**: an object that cannot be passed as a parameter, cannot be returned from a subroutine, and cannot be assigned to a variable.
  - Labels of goto-statements and subroutines in Ada 83.