Imperative Languages

Chapters 6 and 8

Key Concepts

- Values are read from memory, and used to compute new values that are when written back to memory (e.g., **x = y+z+w*v**)
- **Expressions** are used to produce values
  - Constants, variables, operators, function calls, etc.
  - Some expressions may have **side effects**: change the state of the memory (arguably, a bad idea)
- **Statements** do not produce values, and are used only because of their side effects
  - E.g., an assignment statement
  - Expressions are **evaluated**, statements are **executed**

Values of Expressions

- Normally, an expression E designates a **value**
  - This value is referred to as the **r-value** of E: if E appears on the right-hand side of an assignment statement, E stands for this value (e.g., **y+z+w*v**)
- But sometimes E designates a **location in memory**
  - Only if E can appear on the left-hand side of an assignment (e.g., **x**, **a[i]**, **p->s.f[j+k]** in C)
  - The **l-value** of E is that "chunk of memory"
- In C: **d = x; x = b+c;** uses the r-value of **x** in the first assignment, and the l-value of **x** in the second one
  - If the type of variable **x** is *int*, the r-value is the *int* number stored in memory (e.g., 192) and the l-value is the chunk of memory (typically, 4 bytes) where **x** resides

Pointers in C/C++

- Most values are the usual suspects: numbers, characters, structures, arrays, etc.
- Special category: **pointer values**
  - A pointer value is a **“handle”** to a chunk of memory
    - C implementations: the address of the first byte in memory
- Creating pointer values: address-of operator **&**
  - **&E**: find the l-value of E and create a handle to it
- Using pointer values: dereference operator *****
  - ***E**: use the r-value of E to get to the memory

```
x = 1; p = &x;  y = 2; q = &y;  a[7] = 3; r = &a[7];
*p = *q; *q = *q + *r;
```

References in Java

- Different syntax, essentially the same semantics

```
class Rectangle { public double height, width; }
main(…) {
  Rectangle x , y;
  x = new Rectangle(); // 1) Create a Rectangle object in memory
                       // 2) Produce a reference value which is
                       //    a handle to this object
                       // 3) Assign this reference value to x
  y = x;               // Copy the r-value of x
  y.width = 3.14;      // 1) Use the r-value of y to get to the object
}                      // 2) Assign based on the l-value of field width
```

Expressions

- Elements: names for “chunks of memory”; constants; function calls; operators
- **Operators** and their operands
  - Arity: unary, binary, ternary – e.g., **e1?e2:e3** in C
    - Unary: prefix or postfix – e.g., **++ e1** vs. **e1 ++** in C
    - Binary: prefix, infix, postfix: **+ e1 e2** vs. **e1 + e2** vs. **e1 e2 +**
  - Precedence and associativity: e.g., **y+z+w*v**
- **Functions**: built-in or programmer-defined
  - E.g., math library in C provides **double log(double x)**
  - Prefix notation: e.g., **pow (e1 , e2)** where e1 and e2 are function arguments (a.k.a. actual parameters)
  - Typically, functions should not have side effects

Side Effects of Expression Evaluation

- Desirable principle: we can replace an expression with the r-value of this expression

  x = 5; y = 1 + x++; if (y == x) printf("OK");

  x = 5; y = 1 + 5; if (y == x) printf("OK");

  – Known as **referential transparency**

  – Not possible when expressions have side effects
- Expressions in C
  – Operators **= ++ -- +=** etc. have side effects
    - E.g., **x=expr** evaluates to the value assigned to **x**
    - E.g., **a[v = x++] = y = z++ + w** is a valid expression

  – No **assignment statement**, but **expression statement**
    - **expr;** – evaluate the expression and throw away the value

Order of Evaluation

- Precedence and associativity are not enough
  - E.g., in **a – f(b) – c*d** will **f(b)** be evaluated before or after **a**? Will **a – f(b)** be evaluated before/after **c*d**?
    - What if **f(b)** has side effects – e.g., changes **a**, **c**, or **d**?
- Order for function arguments: e.g., **f(a, g(b), h(c))**
- The language semantics has to state this order
  - To clarify the behavior in the presence of **side effects**
  - To enable **compiler optimizations**: e.g., computing **c*d** before **f(b)** requires a register to remember the value during the call to **f** (may be bad for performance)
  - E.g., C does not specify order for operands/arguments (aim: performance) but Java does (aim: correctness)

Defined Order of Evaluation in C

- Boolean expressions: **e1 && e2** and **e1 || e2**
  - **e1** is evaluated before **e2**
  - **Short-circuit semantics**: **e2** may never be evaluated
    - **&&**: if **e1** evaluates to false; **||**: if **e1** evaluates to true
- Comma operator: **e1 , e2**
  - **e1** is evaluated before **e2**: e.g., **a=f(b) , c=g(d)**
- Conditional operator: **e1 ? e2 : e3**
  - **e1** is evaluated before **e2** and **e3**
- At the end of an expression statement: **e1; e2;**
  - **e1** is evaluated before **e2**

Statements

- **Assignment** statements (e.g., **x:=y+z** in Pascal)
- Control flow
  - **Selection** statements: e.g., **if-then-else**, **switch**
  - **Iteration** statements: e.g., **while**, **do-while**, **for**
  - **Jump** statements: e.g., **goto**, **return**, **break**, **continue**, **throw**
- Unstructured control flow: **goto** allows arbitrarily complex behavior, but leads to bad code
- Structured control flow: use standard "clean" abstractions such as if-then-else, while, etc.

Example of Unstructured Control Flow

```
void main() {
      int x = 1, y = 2, z = 3;
 L1: if (x >= 10) goto L3;
 L2: y = y+z;
      if (x == 10) goto L4;
      x = x+1;
      goto L1;
 L3: y = y + 1;
      goto L2;
 L4: printf("x = %d, y = %d", x , y);
}
```

Procedures

- Subroutines, procedures, functions, methods, …
  - **Subroutine**: the general term
    - **Procedure**: subroutine that does not return a value
    - **Function**: subroutine that returns a value
    - **Method**: subroutine in some object-oriented languages
  - Some people use "procedure" as the general term (instead of "subroutine")
    - **Procedural languages**: imperative languages in which procedures are a major abstraction mechanism (C, Fortran)
- Reusable **procedural abstraction**: a collection of statements is abstracted by **name**, list of **formal parameters**, and (optionally) **return value**

Basic Mechanism

- A caller (another procedure) makes a call
  - The caller provides **arguments** (a.k.a. **actual parameters**) – in general, expressions that are evaluated immediately before the call
- Parameter passing: the actual parameters are "mapped" to the **formal parameters**
  - Several parameter passing modes
- **Memory is allocated** for the formal parameters and the local variables of the called procedure
- The flow of control enters the procedure
  - Eventually returns to the caller (or throws an exception)

Scopes in Imperative Languages

- Which entities (variables, procedures, …) are **accessible** in which parts of a program? What is their **lifetime**?
- Example: Fortran has a set of subroutines (procedures)

  - **Procedure names** are visible everywhere
  - **Local variables** are visible only in the declaring proc
  - **Global variables** are visible everywhere

Static Scope Rule

- Algol, Pascal, Modula-2, C, C++, Java, …
- Entities accessible in a scope = entities declared in that scope + entities declared in surrounding scope (minus those with name conflicts) + entities declared in scopes surrounding that scope …
- Each scope is a box whose sides are one-way mirrors; you can look out of the box, but you can't look into a box

C++ Example

```
class Point {
  public: Point(double x, double y);
        virtual void print(); virtual void add(Point* q);
  private: double x,y;
};
Point::Point(double x, double y) { this->x = x; this->y = x; }
void Point::print() { cout<<x<<","<<y<<endl; }
void Point::add(Point* q) {
  q->print();
  {
    Point *q = new Point(100.0,100.0);
    this->x += q->x; this->y += q->y;
  }
  this->x += q->x; this->y += q->y;
}
int main(void) {
  Point* p1 = new Point(1.0,1.0); p1->print();
  Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
  return 0; }
```

Compile time vs. Run time

- At **compile time**, we consider the scopes and their nesting
  - Determines which entities (variables, etc.) are accessible in which parts of the code
    - Additional restrictions on accessibility may be imposed with “access modifiers” e.g., private, protected, etc.
- At **run time**, each scope has a **lifetime**
  - Anything declared in this scope has this lifetime – it becomes alive at the start of the scope, and “dies” at the end of the scope

C++ Example: Lifetimes

```
class Point {
  public: Point(double x, double y);
        virtual void print(); virtual void add(Point* q);
  private: double x,y;
};
Point::Point(double x, double y) { this->x = x; this->y = y; }
void Point::print() { cout<<x<<","<<y<<endl; }
void Point::add(Point* q) {
  q->print();
  {
   Point *q = new Point(100.0,100.0);
   this->x += q->x; this->y += q->y;
  }
  this->x += q->x; this->y += q->y;
}
int main(void) {
  Point* p1 = new Point(1.0,1.0); p1->print();
  Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
  return 0; }
```

Start of program

Global scope: **main**; class **Point** and all its methods (**Point**, **print**, **add**) and fields (**x**, **y**)

Local scope for main: **p1**, **p2** (locals)

Local scope for Point constructor: **this** (formal); **x**, **y** (formals)

Local scope for print: **this** (formal)

…

Local scope for add: **this** (formal); **q** (formal)

Local scope for block: **q**

…

End of program

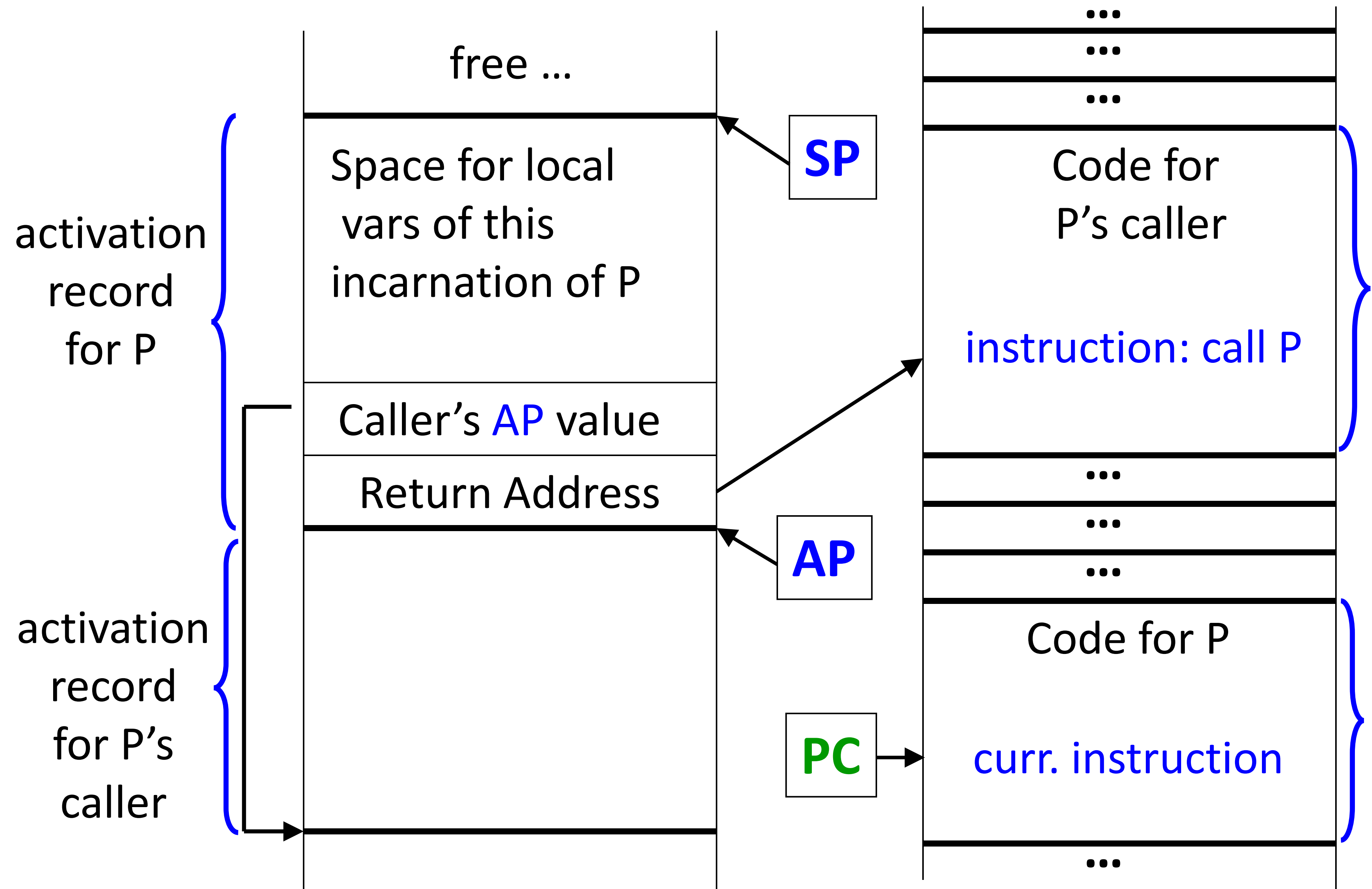
Implementation of Static Scoping

- Consider a language without nesting of procedures (e.g., C)
  - We have one global scope and then just separate local scopes for each procedure
    - All procedure names are in the global scope
    - Global variables in the global scope; local variables in each local scope
- Memory regions
  - code segment: code for all procedures
  - global (static) segment: the global variables
  - run-time call stack: the local variables

Run-time Call Stack

- When a procedure P begins execution:
  - An **activation record** for that incarnation of P is created on the stack (has space for local variables)
  - During this incarnation of P, the **activation record pointer** (**AP**) register will contain the (starting) address of this activation record
  - The **stack pointer** (**SP**) register will contain the address of the location immediately beyond this a.r.
- When this incarnation of P finishes, control returns to the caller, SP is set to the current AP, and AP set to the address of the activation record of the caller

Call Stack: Sample Implementation



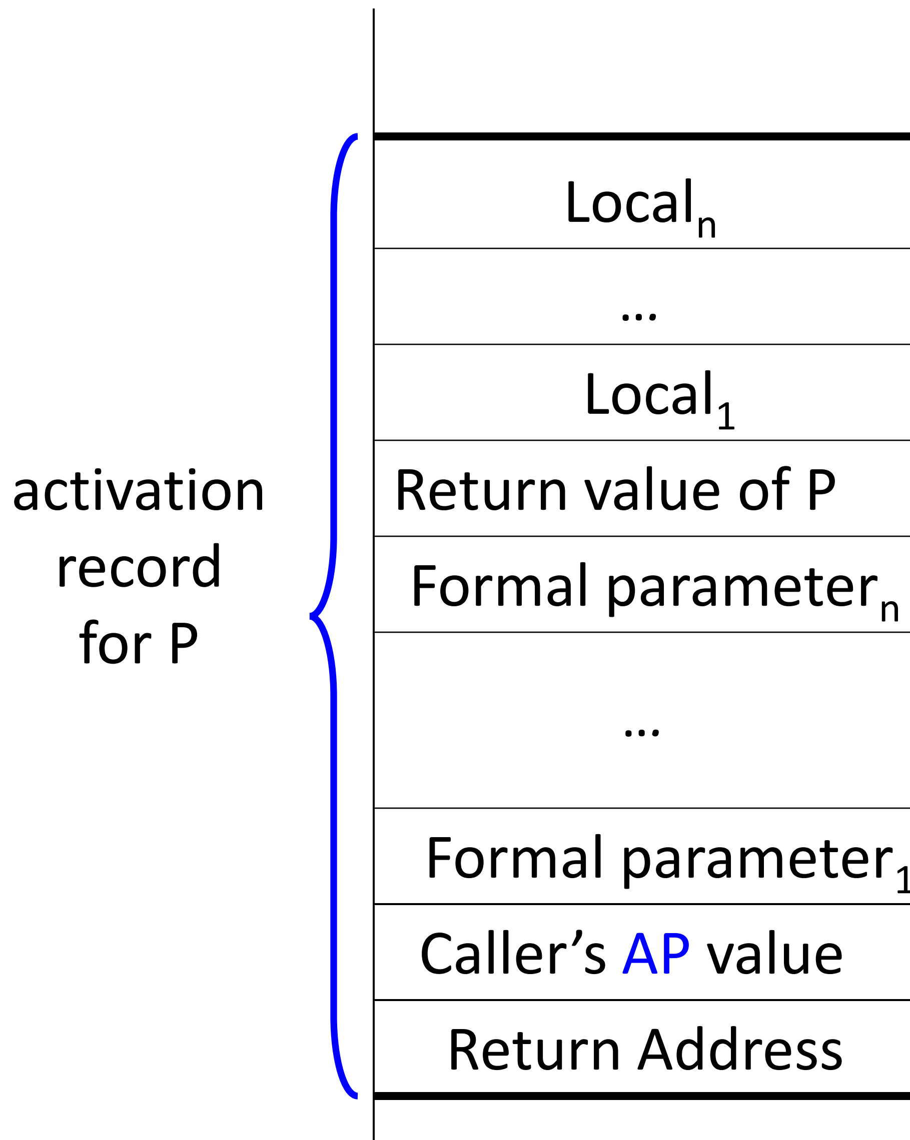
Compile-time Code Generation

- What code does the compiler produce to make this work?
  - **Mem** is the memory – think of it as an array of memory locations
  - **SP** is the stack pointer; points to the next free element of **Mem**
  - **AP** is activation record pointer; points to the first element of the current activation record.
    - Current activation record is from **Mem**[**AP**] through **Mem**[**SP**-1]
  - **PC** is the program counter

Code at Calls and Returns

- Code at "call P"
  - Save return address: **Mem**[**SP**]=PC+4, assuming 4 byte instructions
  - Save pointer to caller's activation record: **Mem**[**SP**+4]=AP
  - Allocate space for new activation record for P: **AP**=**SP** and **SP**=**SP**+**n** where **n** is the size of P's activation record; known at compile time
  - Jump to P: **PC**=address of first instruction in P; known at compile time
- Return: pop the activation record from the stack and go back to the caller: restore **AP**, **SP**, reset **PC**
  - **SP**=**AP**, **AP**=**Mem**[**AP**+4], **PC**=**Mem**[**SP**]

Call Stack: Parameters and Returns



- The formal parameters and the return values are at offsets (w.r.t. **AP**) that are known at compile time
- The caller of P can access them using its value of **SP** (the top of the stack), before and after the call

Parameter Passing Modes

- **Call-by-value:** C, Pascal, C++, Java, …
  - The formal parameter is essentially a local variable initialized with the corresponding argument

```
void Swap(int x, int y) // does not work
    { int z; z = x; x = y; y = z; }
```

- **Call-by-reference:** C++, Pascal, …
  - The parameter is not a new variable, but a new reference to the corresponding argument
  - The argument of the call must have an l-value; this l-value is being passed in the call
  - For large objects, could be more efficient than call-by-value (no need to copy large amounts of memory)

Example: Parameter Passing in C

- C does not have call-by-reference
  - Just call by value
- Using pointers, programmers usually "simulate" call-by-reference

```
int x = 1;
void main() {
        int y = 2;
        int* p;
        p = &x; increment(p);
        p = &y; increment(p);
}
void increment (int *f) { *f = *f + 1; }
```

Inside **increment**, ***f** and **x** may refer to the same memory: **aliases**

Example: Parameter Passing in C++

- C++ supports both call-by-value and call-by-reference

```
int x = 1;
void main() {
        int y = 2;
        int z = 3;
        increment(x, z);
        increment(y, z);
}
void increment (int& a, int b) { a = a + b; }
```

Inside **increment**, **a** and **x** may refer to the same memory: **aliases**

Variable Number of Parameters

- The number of parameters is not specified
  - But they all must be of the same type T
    - **T …** must appear at the end of the parameter list

```
void print_lines(int x, String … lines) { // Java code
    System.out.println("There are " + lines.length + " extra args");
    for (String str : lines) System.out.println(str);
}
void main() { print_lines(1, "arg2", "arg3", "arg4"); }
```

- Similar mechanisms exist in C, C++, C#

Lifetimes and Memory Management (1/2)

- More detailed discussion in Section 3.2
- **Static allocation**: address determined once and retained throughout the execution of the program
  - Global variables in C, Pascal, etc.
  - **static** fields in C++, Java, etc.
  - Local variables in languages without recursion
    - E.g., earlier versions of Fortran
  - **static** local variables in C
  - Large constants – e.g., string/array constants

Lifetimes and Memory Management (2/2)

- **Stack-based allocation**: address determined when the call happens; lifetime ends when the call ends
  - Push the activation record on the run-time call stack
    - Sometimes the activation record is called a **stack frame**
  - Local variables in languages with recursion
  - Relative address with the stack frame is determined at compile time
- **Heap-based allocation**: space allocated and deallocated **manually by the programmer**
  - C: **A* a = (A*)malloc(sizeof(A));** … **free (a);**
  - C++: **A* a = new A(); … delete a;**
  - Java: **A a = new A();** dealloc with garbage collection

Exceptions

- What do we with “exceptional situations”?
  - Try to open a file, but the file does not exist
  - Try to send a byte over a network socket, but the connection was dropped
  - Try to allocate new memory (e.g., **malloc** in C, **new** in C++/Java/C#), but we have run out of memory
  - Division by zero; use of null pointer/reference; etc.
- Ah hoc solutions (e.g., in C)
  - Use a special return value to signify failure
    - E.g., return value of 0 or -1 signifies an error
  - Set some global error flag – e.g., ***errno*** (integer variable)
    - A call **sqrt(-1)** will return “NaN” (“not a number”) and will set **errno** to EDOM (an integer error code for “argument not in the domain of the function”)

C Example

```
#include <stdio.h>
int main ()
{
  FILE* pFile;
  pFile=fopen("myfile.txt","r"); /* possible problem */
  if (pFile==NULL)
    perror ("Error opening"); /* perror prints a message based on  errno */
  else {
    fputc ('x',pFile);
    if (ferror (pFile))
      printf ("Error writing to myfile.txt\n");
    fclose (pFile);
  }
  return 0;
}
```

Java Example

```
import java.io.*;
class Main {
  public static void main(String[] args) {
        FileReader file = null;
        char c;
        try {
          file = new FileReader("myfile.txt"); // may throw FileNotFoundException
          c = (char) file.read(); // may throw IOException
          System.out.println("char: " + c);
        } catch (FileNotFoundException e) {
          System.err.println("Error opening");
        } catch (IOException e) {
          System.err.println("Error reading from myfile.txt");
        } finally {
          if (file != null) try { file.close(); } catch (IOException e) { }
        }
  }}
```

Basics of Java Exceptions

- **throw** e
- **try** { … } **catch** (SomeExceptionType e) { … } **catch** (AnotherExceptionType e) { … } … **finally** { … }
- Within a method

  try { … **throw** new ExceptionType(); … }
  catch (ExceptionType e) { … }

- Across methods (this is the common case)

  void **m1**() **throws** ExceptionType
  { … **throw** new ExceptionType(); … }

  void **m2**() **throws** ExceptionType { … **m1**(); … }

  void **m3**() { … **try** { … **m2**(); …} **catch** (ExceptionType e) { … }

  – What happens with the run-time call stack?