
Spring 2019

# CSE 216 : Programming Abstractions

## TOPIC 4 – SUBROUTINES

# Outline

- Introduction to subroutines
- Review of stack layout
- Inline expansion
- Parameter passing
- Generic subroutines and modules
- Exception handling
- Coroutines
- Events

# Subroutines

- Why use subroutines?
  - –Give a name to a task.
  - –We no longer care *how* the task is done.
- The *subroutine call* is an expression
  - –Subroutines take arguments (in the formal parameters)
  - –Values are placed into variables (actual parameters/arguments), and
  - –A value is (usually) returned
- Activation record or stack frames is a means to manage the space for local variables allocated to each subroutine call

# Review Of Memory Layout

- Memory allocation strategies
  - Static (compile time)
    - Code
    - Globals
    - *Own* variables
    - Explicit constants (including strings, sets, other aggregates)
  - Stack (run time)
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

# Stack based allocation of space for subroutines

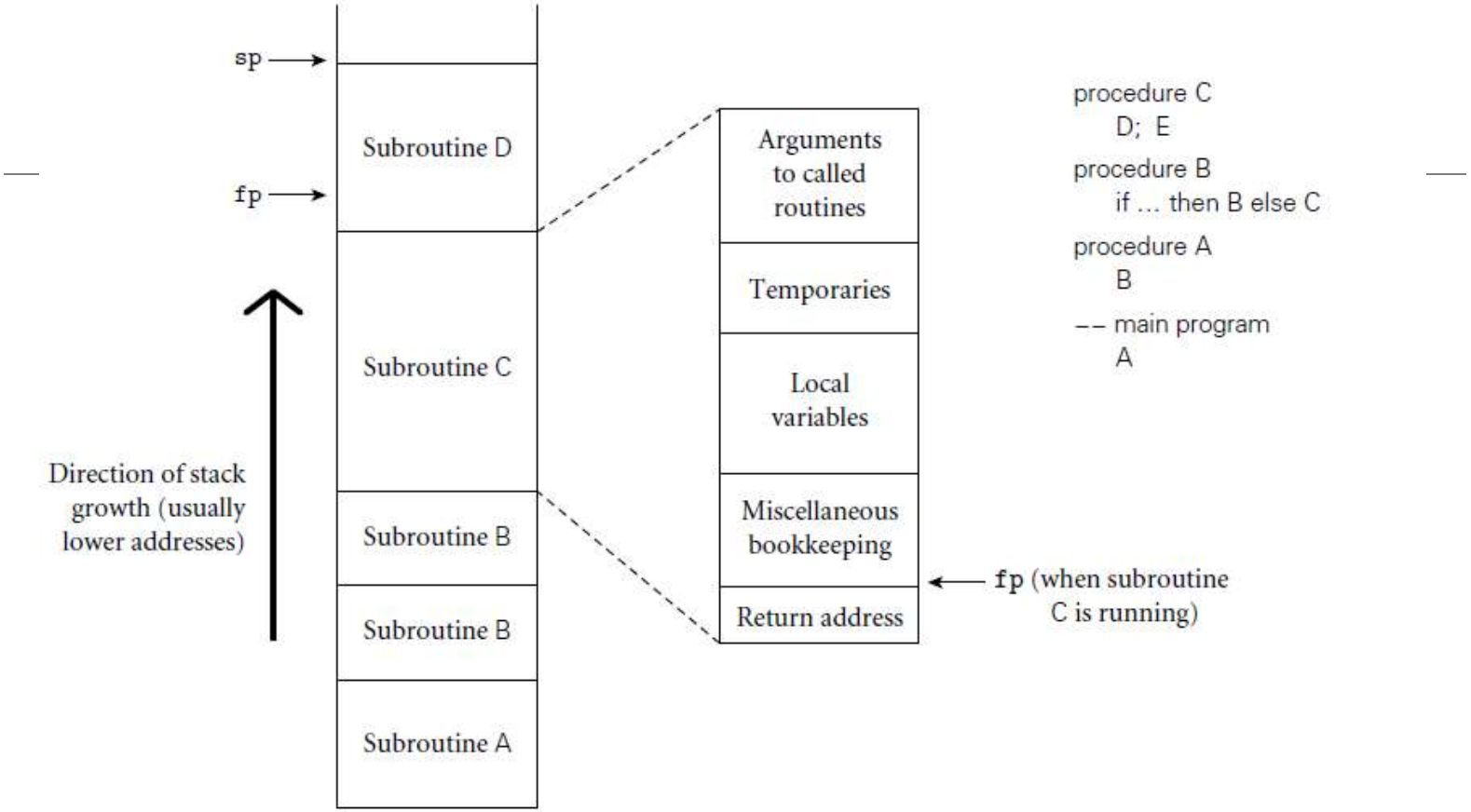



**Figure 3.1 Stack-based allocation of space for subroutines.** We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

# In-line expansion

- Alternative to stack-based calling conventions.
- During compile time, the compiler replaces a subroutine call with the code of the subroutine.
- Without inline functions, the compiler decides which functions to inline.
- The programmer has little or no control over which functions are inlined and which are not.

# In-line expansion by hand - C Example

```
int pred(int x) {
    if (x == 0)
       return 0;
    else
       return x - 1;
}
```

*Before inlining:*

```
int f(int y) {
    return pred(y) + pred(0) + pred(y+1);
}
```

*After inlining:*

```
int f(int y) {
    int temp;
    if (y   == 0) temp  = 0; else temp  = y       - 1; /* (1) */
    if (0   == 0) temp += 0; else temp += 0       - 1; /* (2) */
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1; /* (3) */
    return temp;
}
```

# In-line functions in C++

```
#include <iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    float div;
public:
    void get();
    void sum();
    void difference();
};
inline void operation :: get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: "
    << a+b << "\n";
}
```

```
inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: "
    << a+b << "\n";
}

inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: "
    << a-b << "\n";
}


int main()
{
    cout << "inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

# In-line expansion

- Advantages:
  - –Avoids overhead associated with subroutine calls; faster code.
  - –Encourages building abstractions in the form of many small subroutines.
  - –Related to but cleaner than macros.
  - –Inline functions are parsed by the compiler whereas, the macros in a program are expanded by preprocessor.
- Disadvantages:
  - –Code bloating.
  - –Cannot be used for recursive subroutines.
  - –Code profiling becomes more difficult.

# Parameter Passing

- Two distinct modes of passing parameters to functions.
- Pass by value:
  - –A copy of the argument's value is made and passed to the called function.
  - –Changes to the copy do not affect the original variable's value in the caller.
- Pass by reference:
  - –The caller passes the address of its data.
  - –The caller gives the called function to access the caller's data directly and to modify it if the called function chooses to do so.

# Parameter Passing

- C/C++ functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y } …
proc(&a,b);
```

  - or directly passed by reference (C++)

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

# Parameter Passing – by Value in C

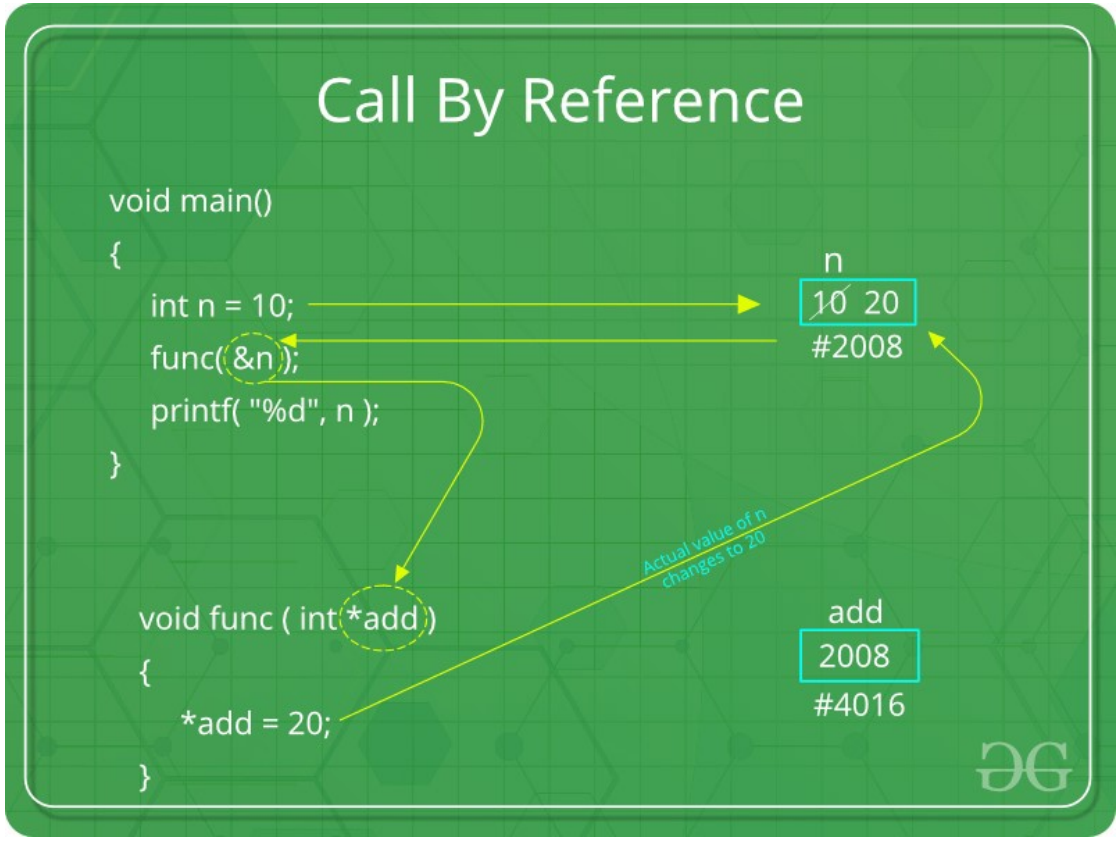
```c
#include <stdio.h>

void func(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n", a, b);
}
int main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    func(x, y);
    printf("In main, x = %d y = %d\n", x, y);
    return 0;
}
```

- https://www.geeksforgeeks.org/parameter-passing-techniques-in-c-cpp/

# Parameter Passing – by Reference in C




- https://www.geeksforgeeks.org/parameter-passing-techniques-in-c-cpp/

# Parameter Passing – by Reference in C

```c
#include <stdio.h>

void swapnum(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}

int main(void)
{
    int a = 10, b = 20;

    // passing parameters
    swapnum(&a, &b);

    printf("a is %d and b is %d\n", a, b);
    return 0;
}
```

- https://www.geeksforgeeks.org/parameter-passing-techniques-in-c-cpp/

# Parameter Passing

- Java
  - Java uses call-by-value for variables of built-in type (all of which are values)
  - Call-by-sharing for variables of user-defined class types (all of which are references)

https://dzone.com/articles/pass-by-value-vs-reference-in-java

# Parameter Passing
# Named Parameters

- The values are passed by ***associating*** each one with a ***parameter name***.

- E.g.,in Objective-C:

```
[window addNewControlWithTitle:@"Title"
        xPosition:20
        yPosition:50
            width:100
           height:50
        drawingNow:YES];
```

- E.g.,in Python:

```
window.addNewControl(title="Title",
                     xPosition=20,
                     yPosition=50,
                     width=100,
                     height=50,
                     drawingNow=true)
```

# Parameter Passing
# Default Parameters

- **Default parameters**: default values are provided to the function
  - –C++ example:

```
void PrintValues(int nValue1, int nValue2=10){
    using namespace std;
    cout << "1st value: " << nValue1 << endl;
    cout << "2nd value: " << nValue2 << endl;
}
int main(){
    PrintValues(1);
 // nValue2 will use default parameter of 10

 PrintValues(3, 4);
 // override default value for nValue2
}
```

# Parameter Passing
# Variadic functions

- functions of **indefinite** arity – one which accepts variable number of arguments
- Java

```
public class Program {
    private static void printArgs(String... strings) {
        for (int i = 0; i < strings.length; i++) {
            String string = strings[i];
            System.out.printf("Argument %d: %s", i, string);
        }
    }

    public static void main(String[] args) {
        printArgs("hello", "world");
    }
}
```

https://www.tutorialspoint.com/What-are-variadic-functions-in-Java

# Returning from a function

- Different ways of returning a value from a function.
  - –Return statement

    return *expression (causes immediate termination of function)*
  - –Avoid termination

    rtn := *expression*

    ...

    return rtn
- ML, its descendants, and several scripting languages allow a Multi-value returns
- In Python, for example, we might write

```
def foo():
  return 2, 3
  ...
  i, j = foo()
```

# Generic subroutines and modules

- Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
  - When defining a function, we don't need to give all the types
  - When we invoke the class or function we specify the type: *parametric polymorphism*
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right

```
public static <T extends Comparable<T>> void sort(T A[]) {
...
if (A[i].compareTo(A[j]) >= 0) {
...
}
...
}
Integer[] myArray = new Integer[50];
sort(myArray);
```

# Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors
- Raising exceptions:
  - Automatically by the run-time system as a result of an abnormal condition
  - (e.g., division by zero)
  - throw/raise statement to raise exceptions manually
  - Most languages allow exceptions to be handled locally and propagate unhandled exceptions up the dynamic chain.

# Exception Handling

- 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

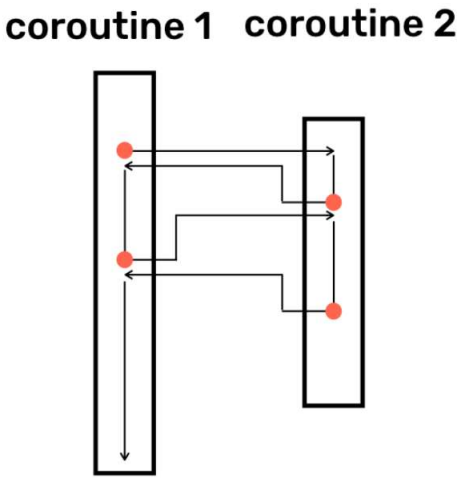
# Exception Handling

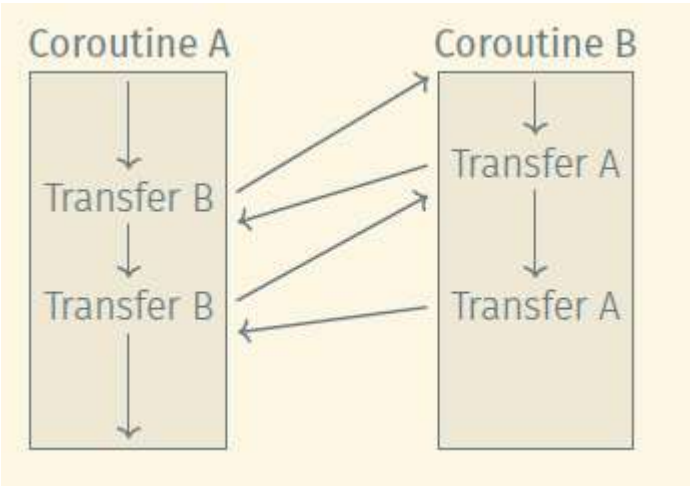
- Java:
  - –throw throws an exception.
  - –try encloses a protected block.
  - –catch defines an exception handler.
  - –finally defines block of clean-up code to execute no matter what.
  - –Only Throwable objects can be thrown.
  - –Must declare uncaught checked exceptions.
- C++:
  - –throw, try, and catch as in Java
  - –No finally block
  - –Any object can be thrown.
  - –Exception declarations on functions not required

```
try {
...
throw ...
...
}
catch (SomeException e1) {
...
}
catch (SomeException e2) {
...
}
finally {
...
}
```

# Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other voluntarily and explicitly, by name
- Coroutines can be used to implement
  - iterators
  - threads
  - Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack



# Coroutines

- As a simple application, consider a “screen-saver” program, which paints a mostly black picture on the screen of an inactive workstation, and which keeps the picture moving (to avoid phosphor or liquid-crystal “burn-in”), and also performs “sanity checks” on the file system in the background, looking for corrupted files
- We could write a loop which does screen update and check in one block, but this mixes tasks
- Better: use coroutines

```
coroutine check file system
  for all files …
coroutine update screen
  loop
       update screen
```

# Event types

- An event is something to which a running program (a process) needs to respond, but which occurs outside the program, at an unpredictable time.
  - The most common events are inputs to a graphical user interface (GUI) system: keystrokes, mouse motions, button clicks.
  - They may also be network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation
- A handler—a special subroutine— is invoked when a given event occurs.
- Thread-Based Handlers:
  - In modern programming languages and run-time systems, events are often handled by a separate thread of control, rather than by spontaneous subroutine calls

# Event types

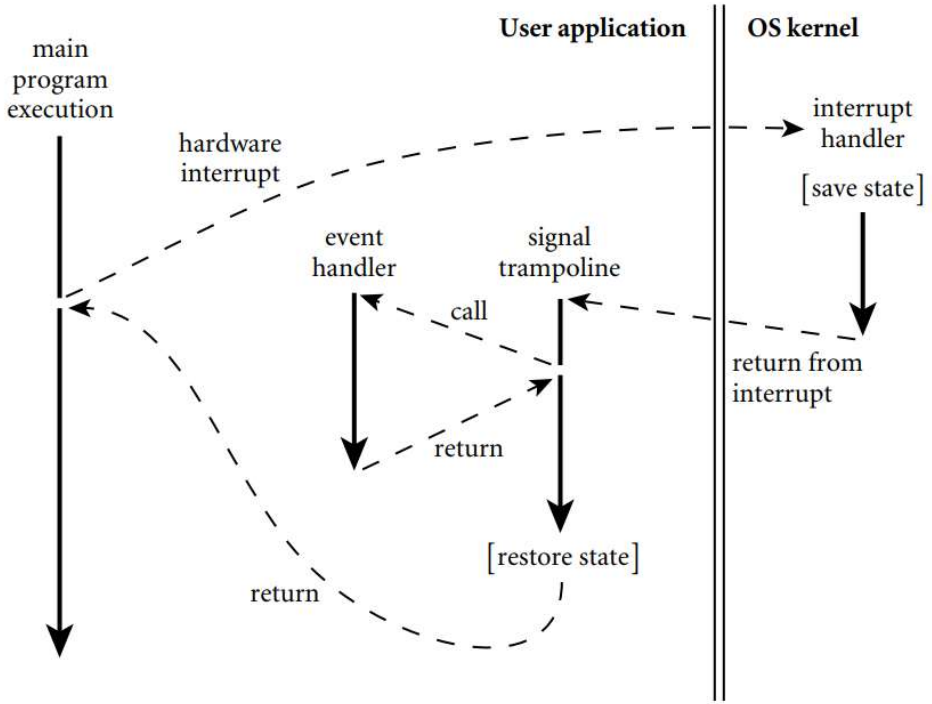



**Figure 8.7** **Signal delivery through a trampoline.** When an interrupt occurs (or when another process performs an operation that should appear as an event), the main program may be at an arbitrary place in its code. The kernel saves state and invokes a *trampoline* routine that in turn calls the event handler through the normal calling sequence. After the event handler returns, the trampoline restores the saved state and returns to where the main program left off.

# Questions?