Warm-up Activity

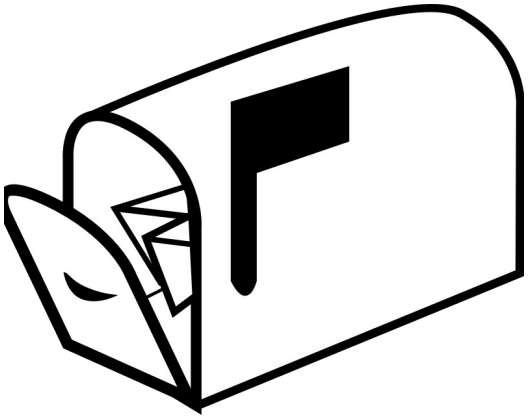
- What does the following C++ program print?

```
int foo(int x, int *y, int &z)
{
    x = 4; *y = 5; z = 6;
    printf("%d %d %d\n", x, *y, z);
    return x + *y + z;
}

int main()
{
    int a, b, c, d;
    a = 1; b = 2; c = 3;
    d = foo(a, &b, c);
    printf("%d %d %d %d\n", a, b, c, d);
}
```

CS 430
Spring 2022

Mike Lam, Professor



Parameters and Subprograms

Course overview

- General topics
  - **Syntax** (what a program looks like)
  - **Semantics** (what a program means)
  - **Implementation** (how a program executes)

9	Mar 21	13: Parameters (B)	9
		14: Subprogram Invocation (R)	
10	Mar 28	15: Prolog 1 (P)	16
11	Apr 4	16: Activations and Environments (B)	10
12	Apr 11	17: Prolog 2 (P)	
13	Apr 18	18: Abstraction and OOP (B)	11, 12
14	Apr 25	19: Concurrency and Error Handling (B)	13
15	May 2	20: History (R)	2
		Review	

Subprograms

- Subprograms are fundamental building blocks for programs
  - A form of process abstraction
  - Facilitates modularity and code re-use
- General subprogram characteristics
  - Single entry point (in most languages), may have multiple exits
  - Caller is suspended while subprogram (callee) is executing
  - Control returns to caller when subprogram completes
  - Most subprograms have names (but not all!)
- Procedure vs. function vs. method
  - Functions have return values, procedures do not
  - Methods are associated with classes & objects

Subprograms

- New-ish terms
  - Header: signaling syntax for defining a subprogram
  - Parameter profile: number, types, and order of parameters
  - Protocol: parameter types and return type (if a function)
  - Prototype: declaration without a full definition
  - Referencing environment (of a subprogram): variables visible inside the subprogram
  - Call site: location of a subprogram invocation (not in CPL)

Parameters

- Formal vs. actual parameters
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site
- Semantic models: *in*, *out*, *inout*
  - In parameters: **actual → formal**
  - Out parameters: **actual ← formal**
  - Inout parameters: **actual ↔ formal**

```
void foo(in int num, out int double, out int triple, inout int accumulator) {
  double = 2 * num;
  triple = 3 * num;
  accumulator = accumulator + num;
}
```

Semantic parameter passing keywords (GLSL)

Parameters

- Some languages have semantic keywords
  - Most use more nuanced *implementation* models
- Implementation models (when/what is copied):
  - **Pass-by-value (*in, value*)**
  - Pass-by-result (*out, value*)
  - Pass-by-copy or pass-by-value-result (*inout, value*)
  - **Pass-by-reference (*inout, reference*)**
  - **Pass-by-name (*inout, text*)**

Parameter Implementations

- Pass-by-value
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default
- Pass-by-reference
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
- Pass-by-name
  - Pro: powerful
  - Con: expensive to implement, very difficult to reason about
  - **Rarely used!** (although macros in C/Rust are a form of pass-by-name)

Caveat / nuance

- Few languages support “true” pass-by-reference
  - Most modern languages pass *reference types by value*
    - I.e., the reference is **copied** into the subprogram (but not back out)
    - Re-assigning the formal parameter doesn’t affect the actual parameter
  - C++ has “true” pass-by-reference
    - The “&” operator designates reference parameters

```
public void swap(Object a, Object b) {
  Object tmp = a;
  a = b;
  b = tmp;
}

Object x = new Integer(4);
Object y = new Integer(5);
swap(x, y);
System.out.printf("%d %d\n", x, y);
```

Output: “4 5”

Java

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

int x = 4;
int y = 5;
swap(x, y);
printf("%d %d\n", x, y);
```

Output: “5 4”

C++

Example

- Trace x, y, a, b, c, and d after each numbered line:

```
   foo(a, b, c, d):
1:   a = a + 1              # a is passed by value
2:   b = b + 1              # b is passed by copy
3:   c = c + 1              # c is passed by reference
4:   d = d + 1              # d is passed by name

     x = [1,2,3,4]
     y = 2
5:   foo(x[0], x[1], y, x[y])
```

Example

- Trace x, y, a, b, c, and d after each numbered line:

```
   foo(a, b, c, d):
1:   a = a + 1              # a is passed by value
2:   b = b + 1              # b is passed by copy
3:   c = c + 1              # c is passed by reference
4:   d = d + 1              # d is passed by name

     x = [1,2,3,4]
     y = 2
5:   foo(x[0], x[1], y, x[y])
```

```
      x = [1,2,3,4]  y=2   a=1  b=2  c=&y  d=x[y]
1:    x = [1,2,3,4]  y=2   a=2  b=2  c=&y  d=x[y]
2:    x = [1,2,3,4]  y=2   a=2  b=3  c=&y  d=x[y]
3:    x = [1,2,3,4]  y=3   a=2  b=3  c=&y  d=x[y]
4:    x = [1,2,3,5]  y=3   a=2  b=3  c=&y  d=x[y]
5:    x = [1,3,3,5]  y=3   a=2  b=3  c=&y  d=x[y]
```

Other Design Issues

- How are formal/actual parameters associated?
  - Positionally, by name (“keyword parameters”), or both?
- Are parameter default values allowed (i.e., can a parameter be optional)?
  - Any parameter or only the right-most one? (former generally requires name association)
- In what order are parameters computed/copied?
  - Generally left-to-right (most common) or right-to-left
- Are parameters type-checked?
  - Statically or dynamically?

```
def bar(a:, b:)
  puts “a is #{a}, b is #{b}”
end

bar(a:3, b:4)
bar(b:4, a:3)
```

Name association (Ruby)

```
def foo(a=0, b=1)
  puts “a is #{a}, b is #{b}”
end

foo(3, 4)
foo(3)
foo()
```

Default parameters (Ruby)

Other Design Issues

- Are local variables statically or dynamically allocated?
- Can a subprogram have a variable number of parameters?
  - Sometimes called variadic subprograms
- Can subprograms be nested?
- Can subprograms be polymorphic?
  - Ad-hoc/manual polymorphism via overloading
  - Subtype polymorphism
  - Parametric polymorphism (e.g., templates or generics)
- Are side effects allowed?
- Can a subprogram return multiple values?
  - If allowed, often “wrapped” in a tuple or array under the hood

Other Design Issues

- Can subprograms be passed as parameters?
  - How is this implemented?
    - Explicit via function pointers or implicit (e.g., lambdas)
  - Are subprograms first-class?
    - Can they also be returned or stored in variables?
  - If nested subprograms are also allowed, which referencing environment should be used?
    - Shallow/dynamic: call that invoked the subprogram
    - Deep/static: definition of subprogram
    - Ad-hoc: call that passed the subprogram (not used)

Closures

- A closure is a pairing of code (e.g., a subprogram or lambda) and a referencing environment or scope
  - Provides access to non-local variables inside the subprogram
  - Variables must remain accessible after the creating scope ends
  - Such variables are said to have unlimited extent and are often allocated on the heap instead of the stack
- Applications:
  - Parameterized callbacks
  - Avoiding re-computation
  - Function composition
  - Ad-hoc classes

Closure examples (not in CPL)

```
function buildCallback(amount) {
  return function donate() {
    chargeUser(amount);
    showThanks();
  }
}

button5.addEventListener('click', buildCallback(5));
button25.addEventListener('click', buildCallback(25));
button100.addEventListener('click', buildCallback(100));
```

Parameterized callbacks (Javascript)

```
aboveAverage :: [Double] -> [Double]
aboveAverage xs = filter (> mean) xs
  where mean = sum xs / length xs
```

Avoiding re-computation (Haskell)

```
removeSpaces = filter (/= ' ')
lowerize = map toLower
normalize = removeSpaces . lowerize
```

Function composition (Haskell)

Closure examples (not in CPL)

```
function constructPlayer(name, health) {
  const heal = () => {
    health += 10;
    console.log(`${name} has ${health} hit points.`);
  };
  const damage = () => {
    health -= 10;
    console.log(`${name} has ${health} hit points.`);
  }
  const rename = newName => name = newName;
  return {heal, damage, rename};
}

const player = constructPlayer('Mario', 99);
player.heal();
player.rename('Luigi');
player.damage();
```

Ad-hoc classes (Javascript)

Misc. Topics

- Macros
  - Call-by-name, “executed” at compile time
  - Often makes code more generic or parameterized

```
#include <stdio.h>

#define DEBUG
#define MAX_LEN 256
#define MAX(X,Y) ((X > Y) ? X : Y)

char name[MAX_LEN];

void debug_print(char* msg)
{
    static int longest_msg = 0;
    #ifdef DEBUG
    printf(“%s\n”, msg);
    #endif
    l_msg = MAX(l_msg, strlen(msg));
}
```

variadic function

```
int printf(char *, ...);

[other contents of stdio.h omitted]


char name[256];

void debug_print(char* msg)
{
    static int l_msg = 0;
    printf(“%s\n”, msg);
    l_msg = ((l_msg > strlen(msg)) ?
             l_msg : strlen(msg));
}
```

C example

Misc. Topics

- Coroutines
  - Co-operating subprograms can “resume” each other
  - Often used for iterators or generators

variadic function

```
fun roundRobin(vararg items: String): Sequence<String> {
  return sequence {
    var i = 0
    while (true) {
      yield(items[i])
      i = (i + 1) % items.size
    }
  }
}

val spinner = roundRobin("Green", "Yellow", "Red").iterator()
repeat(20) {
  println(spinner.next())
}
```

Kotlin example