7. Functions II

10. Managing Functions and Data

Global Variables (1)

```
#include <iostream>

double result = 0.0, arg1, arg2;
void get_input() {
   std::cin >> arg1 >> arg2;
}
void report() {
   std::cout << result << '\n';
}
void add()     {      result = arg1 + arg1;         }
void subtract(){      result = arg1 - arg2;         }

int main() {
   get_input();
   add();
   report();
}
```

Global Variables (2)

```
#include <iostream>
int x; // = 0?

int main() {
   int x = 10;
   std::cout << x << std::endl;
   std::cout << ::x << std::endl;       // scope resolution operator
}

// extern int i;
// declaration of a variable named i of type int,
// defined somewhere in the program.

// pure function
// The function return values are identical for identical arguments (no
//  variation with local static variables, non-local variables, mutable
//  reference arguments or input streams).
//  The function application has no side effects (no mutation of local
//  static variables, non-local variables, mutable reference arguments or
//  input/output streams).
```


Static Variables

```
#include <iostream>
#include <iomanip>
int count() {
   static int cnt = 0; // static int cnt;
   return ++cnt; // Increment and return current count
}
int main() {
   // Count to ten
   for (int i = 0; i < 10; i++)
      std::cout << count() << ' ';

   std::cout << '\n';
}


// A global static variable is one that can only be accessed in the
file where it is created. This variable is said to have file scope.
```

Overloaded Functions

```
// Function overloading is a C++ programming feature that allows us
// to have more than one function having same name but different
// parameter list

void f() { /* ... */ }
void f(int x) { /* ... */ }
void f(double x) { /* ... */ }
void f(int x, double y) { /* ... */ }
void f(double x, int y) { /* ... */ }
```


Default Arguments (1)

```
#include <iostream>
void countdown(int n=10) {
  while (n >= 0) // Count down from n to zero
  std::cout << n-- << '\n';
}

int main() {
  countdown(5);
  std::cout << "----------" << '\n';
  countdown();
}

-----------------------------------------------------------------
int sum_range(int n=0, int m=100);
int sum_range(int n, int m=100);
int sum_range(int n=0, int m);
```

Default Arguments (2)

```
#include <iostream>
void countdown(int n=10);
int main() {
  countdown(5);
  std::cout << "----------" << '\n';
  countdown();
}
void countdown(int n) {
  while (n >= 0) // Count down from n to zero
  std::cout << n-- << '\n';
}
```


Default Arguments (3)

```
// Mixing overloading and default arguments can produce ambiguities
// that the compiler will not allow.

void f() { /* .... */ }
void f(int n=0) { /* .... */ }

-----------------------------------------------------------------

void f(int m) { /* .... */ }
void f(int n=0) { /* .... */ }
```

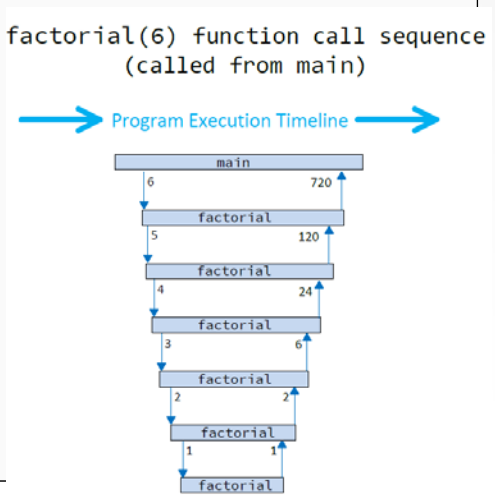
Recursion (1)

```
// Recursion: function calls itself directly or indirectly
#include <iostream>
void F(int n) {
   std::cout << n%10 << std::endl;
   if(n/10 > 0) F(n/10);
}
int main() {
   F(123);
}

-----------------------------------------------------------------
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}
```


Recursion (2)

```
factorial(6)   = 6 * factorial(5)
               = 6 * 5 * factorial(4)
               = 6 * 5 * 4 * factorial(3)
               = 6 * 5 * 4 * 3 * factorial(2)
               = 6 * 5 * 4 * 3 * 2 * factorial(1)
               = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
               = 6 * 5 * 4 * 3 * 2 * 1 * 1
               = 6 * 5 * 4 * 3 * 2 * 1
               = 6 * 5 * 4 * 3 * 2
               = 6 * 5 * 4 * 6
               = 6 * 5 * 24
               = 6 * 120
               = 720
```



Recursion (3)

```
#include <iostream>
int gcd(int m, int n) {
  if (n == 0)
    return m;
  else
    return gcd(n, m % n);
}

-----------------------------------------------------------------------

int fibonacci(int n) {
   if (n <= 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fibonacci(n - 2) + fibonacci(n - 1);
}
```


Making Functions Reusable

```
// prime.cpp
#include <cmath>
bool is_prime(int n) {
   bool result = true;
   double r = n, root = sqrt(r);
   for (int trial_factor = 2; result && trial_factor <= root;
      trial_factor++)
      result = (n % trial_factor != 0);
   return result;
}

// prime.h
bool is_prime(int);

// test.cpp
#include "prime.h"
int main() { /*    */ }
```

Pointers (1)

```
// &: address operator
#include <iostream>
int main() {
   int x = 10;

   std::cout << &x << std::endl;
}

// A pointer is a variable that holds a memory address
// where a value lives.
int *p1;
double *p2;
char *p3;
```

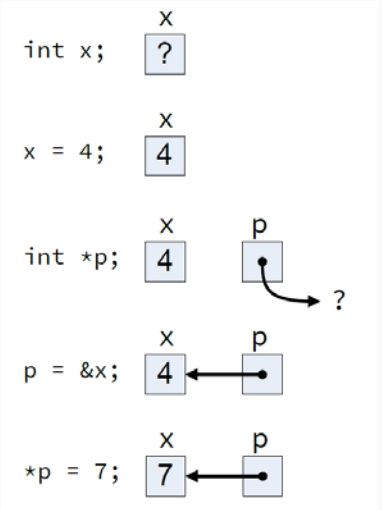

Pointers (2)

```
#include <iostream>
int main() {
   int x;
   x = 4;
   int *p;
   p = &x;

   std::cout << &x << std::endl;
   std::cout << p << std::endl;

   std::cout << x << std::endl;
   std::cout << *p << std::endl;

   *p = 7; // dereferencing/indirect operator
   std::cout << x << std::endl;
   std::cout << *p << std::endl;
}
```



Pointers (3)

```
// nullptr

// reinterpret_cast: conversion between unrelated types
//   int → ptr, ptr → int, ptr → ptr
//
// int *p = reinterpret_cast<int *>(5);
```


Reference Variable

```
int x;
int& r = x; // r is a reference variable. r aliases x, not address
-----------------------------------------------------
#include <iostream>
int main() {
   int x = 5;  int y = x;     int& r = x;
   std::cout << "x = " << x << '\n';
   std::cout << "y = " << y << '\n';
   std::cout << "r = " << r << '\n';
   x = 7;
   std::cout << "x = " << x << '\n';
   std::cout << "y = " << y << '\n';
   std::cout << "r = " << r << '\n';
   y = 8;
   std::cout << "x = " << x << '\n';
   std::cout << "y = " << y << '\n';
   std::cout << "r = " << r << '\n';
}
```

Pass by Reference (1)

```
#include <iostream>
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
   int var1 = 5, var2 = 19;
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
   swap(var1, var2);
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```


Pass by Reference (2)

```
#include <iostream>
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
   int var1 = 5, var2 = 19;
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
   swap(var1, var2);
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

Pass by Reference (3)

```
#include <iostream>
void swap(int *a, int *b) {   // Pass by reference via pointers
   int temp = *a;             // Pass by address
   *a = *b;
   *b = temp;
}

int main() {
   int var1 = 5, var2 = 19;
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
   swap(var1, var2);
   std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```


Higher-order Functions (1)

```
// parameter or return

#include <iostream>
int add(int x, int y) {
   return x + y;
}
int multiply(int x, int y) {
   return x * y;
}
int evaluate(int (*f)(int, int), int x, int y) { // function pointer
   return f(x, y);
}
int main() {
   std::cout << add(2, 3) << '\n';
   std::cout << evaluate(&add, 2, 3) << '\n';
   std::cout << evaluate(&multiply, 2, 3) << '\n';
}
```

Higher-order Functions (2)

```
#include <iostream>
int add(int x, int y) {
   return x + y;
}

int main() {
   int (*func)(int, int);
   func = add;
   std::cout << func(7, 2) << '\n';
}
```