Unit 4c

Argument Passing
Unit 4

• **Unit 1**: Scalar processing
  – aka IPO=Input-Process-Output Programs

• **Unit 2**: Linear (1D) Processing

• **Unit 3**: Multidimensional Processing

• **Unit 4**: Divide & Conquer
  (Functional Decomposition)
Argument Passing (Pass-by-Value)

• Passing an argument to a function makes a copy of the argument
• It is like e-mailing an attached document
  – You still have the original on your computer
  – The recipient has a copy which she can modify but it will not be reflected in your version
• Communication is essentially one-way
  – Caller communicates arguments to callee, but callee cannot communicate back because she is working on copies...
  – The only communication back to the caller is via a return value.
Pass by Value (1)

- **Fact**: Function arguments/parameters act like local variables to that function
  - They only as long as the function is executing and then get deallocated.

- When arguments are passed a **copy** of the actual argument value (e.g. 3) is given to the function's input argument
  - So the function is operating on a copy and that copy only lives as long as the function

```cpp
void dec(int);
int main()
{
    int y = 3;
    dec(y);
    cout << y << endl;
    return 0;
}
void dec(int y)
{
    y--;
}
```
Pass by Value (2)

- Wait! But they have the same name, 'y'
  - What's in a name...Each function is a separate entity and so two 'y' variables exist (one in main and one in decrement it)
  - The only way to communicate back to main is via return
  - Try to change the code appropriately

- **Main Point:** Each function is a completely separate "sandbox" (i.e. is isolated from other functions and their data) and copies of data are passed and returned between them

```cpp
void dec(int);
int main()
{
    int y = 3;
    dec(y);
    cout << y << endl;
    return 0;
}
void dec(int y)
{
    y--;
}
```

```cpp
__dec(int);
int main()
{
    int y = 3;
    __dec(y);
    cout << y << endl;
    return 0;
}
__dec(int y)
{
    y--;
    __________
}
```
Pass by Value Solution

• Wait! But they have the same name, 'y'
  – What's in a name...Each function is a separate entity and so two 'y' variables exist (one in main and one in decrement it)
  – The only way to communicate back to main is via return
  – Try to change the code appropriately

• **Main Point:** Each function is a completely separate "sandbox" (i.e. is isolated from other functions and their data) and copies of data are passed and returned between them

```c
void dec(int);
int main()
{
    int y = 3;
    dec(y);
    cout << y << endl;
    return 0;
}
void dec(int y)
{
    y--;
}
```
Reminders: Common Mistakes

- Problem 1: Don't list return type when you call a function. It will substitute the return value in place of call
- Problem 2: Need to save return value

```cpp
int dec(int val); // prototype
int main()
{
    int y = 3;
    int dec(y); // y = dec(y);
    cout << y << endl;
    return 0;
}
int dec(int val)
{
    val--; // y = dec(y);
    return val;
}
```

- Problem 1: Don't relist the type of the argument when you make the call
- Problem 2: Need to pass a variable that exists in the calling function

```cpp
int dec(int val); // prototype
int main()
{
    int y = 3;
    y = dec(int val); // y = dec(y);
    cout << y << endl;
    return 0;
}
int dec(int val)
{
    val--; // y = dec(y);
    return val;
}
```
Passing Arrays As Arguments

• Can we pass an array to another function?
  – YES!!

• Syntax:
  – **Step 1**: In the prototype/signature: Put empty square brackets after the parameter name if it is an array (e.g. `int data[]`)
  – **Step 2**: When you call the function, just provide the name of the array

```cpp
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
  int total = 0;
  for(int i=0; i < size; i++)
  {
    total += data[i];
  }
  return total;
}

int main()
{
  int vals[100];
  /* some code to initialize vals */
  int mysum = sum(vals, 100);
  cout << mysum << endl;
  // prints sum of all numbers
  return 0;
}
```
Pass-by-Value & Pass-by-Reference

• What are the pros and cons of emailing a document by:
  – Attaching it to the email
  – Sending a link (URL) to the document on some cloud service (etc. Google Docs)

• **Pass-by-value** is like emailing an attachment
  – A *copy* is made and sent

• **Pass-by-reference** means emailing a link to the original
  – *No copy is made* and *any modifications by the other party are seen by the originator*
Arrays And Pass-by-Reference

- **Single (scalar) variables** are passed-by-value in C/C++
  - Copies are passed
- **Arrays** are passed-by-reference
  - Links are passed
  - This means any change to the array by the function is visible upon return to the caller

```c
void dec(int y) {
    y--;  
}
```

```c
void init(int x[], int size) {
    for(int i=0; i < size; i++){
        x[i] = 0; // changing data[i]
    }
}
```
But Why?

- If we used pass-by-value then we'd have to make a copy of a potentially HUGE amount of data (what if the array had a million elements)
- To avoid copying vast amounts of data, we pass a link

// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
    int total = 0;
    for(int i=0; i < size; i++){
        total += data[i];
    }
    return total;
}

int main()
{
    int vals[100];
    /* some code to initialize vals */
    int mysum = sum(vals, 100);
    cout << mysum << endl;
    // prints sum of all numbers
    return 0;
}

© 2023 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
So What Is Actually Passed?

- The "link" that is passed is just the starting address (e.g. 520) of the array in memory
- The called function can now use 520 to access the original array (read it or write new values to it)

```c
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size)
{
    int total = 0;
    for(int i=0; i < size; i++){
        total += data[i];
    }
    return total;
}

int main()
{
    int vals[100];
    /* some code to initialize vals */
    int mysum = sum(vals, 100);
    cout << mysum << endl;
    // prints sum of all numbers
    return 0;
}
```
Analogy

• The first house on a certain block of Catalina Ave. has the address 3600.
• How many houses are on that block?
• There is no way to know!! We would have to count that separately.

• Suppose a large family reunion reserves a block of hotel rooms. The first room is number 428.
• How many rooms are in the reserved block?
• There is no way to know!! We would have to count that separately.
Arrays in C/C++ vs. Other Languages

- Notice that if `sum()` only has the start address it **would not know** how big the array is.
- **Unlike Java, Python** or other languages where you can call some function to find the size of an array, **C/C++ requires you to track the size yourself in a separate variable and pass it as a secondary argument.**

```c
// Function that takes an array
int sum(int data[], int size);

int sum(int data[], int size) {
    int total = 0;
    for(int i=0; i < size; i++){
        total += data[i];
    }
    return total;
}

int main() {
    int vals[100];
    /* some code to initialize vals */
    int mysum = sum(vals, 100);
    cout << mysum << endl;
    // prints sum of all numbers
    return 0;
}
```
Why Don't We Return Arrays from Functions

• In C++, we generally do **NOT** return arrays from a function...because we do **NOT** need to!

• **WHY?**
  – Because we modified the original array in the function

```cpp
int fill(int data[], int size);
void fill(int data[], int size);

int[100] fill(int data[], int size);
void fill(int data[], int size)
{
    for(int i=0; i < size; i++){
        data[i] = i;
    }
}

int main()
{
    int vals[100];
    /* some code to initialize vals */
    fill(vals, 100);
    cout << vals[0] << endl;
    // prints sum of all numbers
    return 0;
}
```
Array Passing Summary

• Syntax:
  – In the prototype/signature: Put empty square brackets after the parameter name if it is an array (e.g. void f1(int data[]))
  – When you call the function, just provide the name of the array (e.g. f1(data);)

• Functions only know what you pass them
  – You must pass the size of the array as an additional parameter in addition to the link to the array
  – Arrays are passed-by-reference meaning no copy is made and changes by a function are actually being made to the original

• The C++ std:: library provides some alternatives to "plain-old arrays" (like vectors), but you will learn about these in CS 103/104 and should not use them in CS 102
Functions as modular units

TIPS FOR CODING FUNCTIONS
Tips

• Look for common, repeated code and factor it out as a function
  – Find differences in data values or constants that are used and turn those into parameters
  – Any variables the code uses must be passed in as arguments
  – A value used after the code must be returned as the return value

• If you can carve out several lines of code that perform one logical task, consider pulling those lines out as a function
Finding Functions

```cpp
#include <iostream>
using namespace std;

int main()
{
    // Print flag of 3 rows
    for(int i=0; i < 3; i++){
        for(int k=0; k < 3-i; k++){
            cout << '/';
        }
        cout << endl;
    }

    // Print flag of 5 rows
    for(int i=0; i < 5; i++){
        for(int k=0; k < 5-i; k++){
            cout << '/';
        }
        cout << endl;
    }
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

void printFlag(int rows);

int main()
{
    printFlag(3);
    printFlag(5);
    return 0;
}

void printFlag(int rows)
{
    for(int i=0; i < rows; i++){
        for(int k=0; k < rows-i; k++){
            cout << '/';
        }
        cout << endl;
    }
}
```
Functions As Independent Units (1)

• Functions should be self contained units
  – Cannot access variables from other functions
  – Should implement a general "recipe" for how to do a task given generic inputs (parameter names)
  – Consider the arguments/parameters as generic names that represent specific inputs when the function is invoked

Remember:
• Functions can only access:
  • The input arguments
  • The local variables they declare
• Functions can only return one value
  • All other values (local variables & input arguments) "die" at the end of the function

```cpp
// Doesn't work
#include <iostream>
using namespace std;
void getInput(); // prototype
void getInput()
{
    cout << "Enter an int: ";
    cin >> x; // Can't access
    // x from main
}
int main()
{
    int x; // get from user
    getInput();
    cout << x << endl;
    return 0;
}
```

```cpp
// Does work
#include <iostream>
using namespace std;
int getInput(); // prototype
int getInput()
{
    int num;
    cout << "Enter an int: ";
    cin >> num;
    return num;
}
int main()
{
    int x; // get from user
    x = getInput();
    cout << x << endl;
    return 0;
}
```
Functions As Independent Units (2)

- Generally, don't do input/output in a function (unless the function specifically indicates it should)...just process values and return something
  - Example: Take the maximum of 2 numbers and print them out
  - Easy to extend to other tasks: take max of 3 numbers

```cpp
#include <iostream>
using namespace std;

int max(int a, int b);

int main()
{
    int x, y, mx;
    cin >> x >> y;
    cout << max(x, y) << endl;
    return 0;
}

int max(int a, int b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

Could you extend to take the max of 4?
Functions As Independent Units (3)

- Generally don't do input/output in a function (unless the function specifically indicates it will)...just process values and return something
  - Example: Take the maximum of 2 numbers and print them out
  - Easy to extend to other tasks: take max of 3 numbers

```cpp
// Bad function decomposition
// -- performs I/O.
#include <iostream>
using namespace std;

int max();

int main()
{
    int mx;
    mx = max();
    cout << mx << endl;
    return 0;
}

int max()
{
    int x, y;
    cin >> x >> y;
    if(x > y)
        return x;
    else
        return y;
}
```

This code works for the stated task (output max of 2 inputs) but cannot easily be reused for 3 or more numbers! [Can't do: max(max()); ]
- Would `cin` 4 numbers
- Doesn't take in an input

```cpp
// Bad function decomposition
// -- performs I/O.
#include <iostream>
using namespace std;

void max(int a, int b);

int main()
{
    cin >> x >> y;
    max(x, y);
    return 0;
}

void max(int a, int b)
{
    if(a > b)
        cout << a << endl;
    else
        cout << b << endl;
}
```

This code works for the stated task (output max of 2 inputs) but cannot easily be reused for 3 or more numbers! [Can't do: max(max(x,y),z); ]
- Would `cout` 2 values
EXERCISES
Tracing Exercise 1

- **Order It**: The numbers below represent lines of code in the program to the right. Order the lines of code from 1-8 [1=first to get executed / 8=last to be executed]. Note: Order them when they START execution not when they finish execution.
  - ____ 5
  - ____ 6
  - ____ 7
  - ____ 8
  - ____ 13
  - ____ 20
  - ____ 26
  - ____ 27

- **Scope It**: Notice that there are many 'char c' declarations and parameters. List the value of the character 'c' (or N/A if no possible value for 'c' exists) just BEFORE the following lines of code get executed:
  - ____ 6
  - ____ 8
  - ____ 13
  - ____ 14
  - ____ 20
  - ____ 21
  - ____ 27

```c
// Assume necessary prototypes have been declared for Q2 and Q3
int main()
{
  double f=1.0, g=2.0, h=3.0;
  int x=5, y=6, z =7;
  char c='U';
  x = myfunc();
  f = doit(x, y, c);
}

double doit(int cat, int dog, char c)
{
  double x = (double) cat;
  c = 'S';
  x = x / (double) dog;
  return x;
}

void yourfunc(char c)
{
  c = 'C';
  return;
}

int myfunc()
{
  yourfunc('!');
  return 2;
}
```
Exercises

• Exercises
  – extract-method1
  – draw-square
  – is-lower-vowel
  – is-vowel
More Exercises

BREAKING CODE INTO FUNCTIONS AND CALLING THEM
Factoring Code / Extracting Functions

• At this point, we should be able to go back to any program or exercise and "refactor" our code into functions (aka "extracting" functions).

• Approach
  – If a block of code performs one main task, you can try to extract that code as a function whose name matches that task description
  – Argument names in the function DON'T have to match the name used in the calling function (but they can)
Extracting Functions Example

- Here, we extract the code to find an element.
- Can you see another function that might be worth extracting?
Extracting Another Function

```cpp
#include <iostream>
using namespace std;

int fill_n(int nums[]);
int find(int nums[], int len, int target);

int main() {
    // setup array with data
    int n, val, data[100];
    cin >> n;
    for(int i=0; i < n; i++)
        { cin >> data[i]; }
    // now perform the given task
    cin >> val;
    int loc = find(data, n, val);
    cout << loc << endl;
    return 0;
}
int find(int nums[], int len, int target)
{
    for(int i=0; i < len; i++)
        { if(target == nums[i]){
            return i;
        }
    }
    return -1;
}
```

```cpp
#include <iostream>
using namespace std;

int fill_n(int nums[]);
int find(int nums[], int len, int target);

int main() {
    // setup array with data
    int n, val, data[100];
    n = fill_n(data);
    // now perform the given task
    cin >> val;
    int loc = find(data, n, val);
    cout << loc << endl;
    return 0;
}
int fill_n(int nums[])
{
    int n;
    cin >> n;
    for(int i=0; i < n; i++)
        { cin >> nums[i]; }
    return n;
}
int find(int nums[], int len, int target)
{  ...
}
Exercises To Refactor Using Functions

• Array Tasks
  – Unique
  – Unique-one
  – Insert-sorted
  – Remove-first
  – Remove-all
  – Max-To-Last (use to sort?)

• Nested Loops
  – etox_range

• HW
  – Grid
  – Priority

• Other examples
  – Prime Range
  – Count digits in number
SOLUTIONS
Tracing Exercise 1 (Solution)

• **Order It:** The numbers below represent lines of code in the program to the right. Order the lines of code from 1-8 [1=first to get executed / 8=last to be execute]. Note: Order them when they START execution not when they finish execution.
  - __1__ 5
  - __2__ 6
  - __6__ 7
  - __8__ 8
  - __7__ 13
  - __4__ 20
  - __3__ 26
  - __5__ 27

• **Scope It:** Notice that there are many 'char c' declarations and parameters. List the **value** of the character 'c' (or N/A if no possible value for 'c' exists) just BEFORE the following lines of code get executed:
  - __'U'__ 6
  - __'U'__ 8
  - __'U'__ 13
  - __'S'__ 14
  - __'I'__ 20
  - __'C'__ 21
  - __N/A__ 27

```c
// Assume necessary prototypes have // been declared for Q2 and Q3
1 int main()
2 {
3   double f=1.0, g=2.0, h=3.0;
4   int x=5, y=6, z=7;
5   char c='U';
6   x = myfunc();
7   f = doit(x, y, c);
8 }
9
double doit(int cat, int dog, char c)
10 {
11   double x = (double) cat;
12   c = 'S';
13   x = x / (double) dog;
14   return x;
15 }
16
void yourfunc(char c)
17 {
18   c = 'C';
19   return;
20 }
21
int myfunc()
22 {
23   yourfunc('!');
24   return 2;
25 }
26
```
#include <iostream>
using namespace std;

// prototypes
int fill_n(int nums[]);
int count(int data[], int len, int target);
bool unique_all(int data[], int len);

int fill_n(int nums[]) {
    int n;
    cin >> n;
    for(int i=0; i < n; i++) { cin >> nums[i]; }
    return n;
}

int count(int data[], int len, int target) {
    int cnt = 0;
    for(int k=0; k < len; k++) {
        if(data[k] == target){
            cnt++;
        }
    }
    return cnt;
}

int main() {
    // setup array with data
    int n, data[100];
    n = fill_n(data);

    // now perform the given task
    bool allUnique = unique_all(data, n);

    if(allUnique)
        { cout << "All unique" << endl; }
    else
        { cout << "Not all unique" << endl; }
    return 0;
}
# Solutions – Unique-One

```cpp
#include <iostream>
using namespace std;

// prototype
int fill_n(int nums[]);
int count(int data[], int len, int target);
int unique_one(int data[], int len);

int fill_n(int nums[]) {
    int n;
    cin >> n;
    for(int i=0; i < n; i++) {
        cin >> nums[i];
    }
    return n;
}

int count(int data[], int len, int target) {
    int cnt = 0;
    for(int k=0; k < len; k++) {
        if(data[k] == target){
            cnt++;
        }
    }
    return cnt;
}

int unique_one(int data[], int len) {
    for(int j=0; j < len; j++) {
        int cnt = count(data, len, data[j]);
        if(cnt == 1){
            return data[j];
        }
    }
    return -1;
}

int main() {
    // setup array with data
    int n, data[100];
    n = fill_n(data);
    // now perform the given task
    int unique = unique_one(data, n);
    cout << unique << endl;
    return 0;
}
```
#include <iostream>
using namespace std;

int findFirstLarger(
    int data[], int len, int val){
    for(int k=0; k < len; k++){
        if(data[k] > val){
            return k;
        }
    }
    return len;
}

void shiftOneRightFrom(
    int data[], int len, int fromIndex)
{
    for(int curr = len-1; curr >= fromIndex; curr--){
        data[curr+1] = data[curr];
    }
}

void printResults(int data[], int len){
    for(int i=0; i < len; i++){
        cout << data[i] << " ";
    }
    cout << endl;
}

int main()
{
    // setup array with data
    int n, data[100];
    cin >> n;
    for(int i=0; i < n; i++)
    {
        cin >> data[i];
    }
    // now perform the given task
    int val;
    cin >> val;
    if(n < 100){
        int loc = findFirstLarger(
            data, n, val);
        shiftOneRightFrom(data, n, loc);
        data[loc] = val;
        n++;
    }
    else {
        cout << "No room" << endl;
    }
    // Output the results
    printResults(data, n);
    return 0;
}
# Solutions – RemoveFirst

```cpp
#include <iostream>
using namespace std;

// prototypes
int find(int nums[], int len, int target);
void shiftOneLeftFrom(int data[], int len, int fromIndex);
void printResults(int data[], int len);

int main() {
    // setup array with data
    int n, data[100];
    cin >> n;
    for(int i=0; i < n; i++)
        { cin >> data[i]; }
    // now perform the given task
    int val, loc;
    cin >> val;
    // find first occurrence of val
    loc = find(data, n, val);
    if(loc != -1) {
        // shift items up from loc to n
        // invariant: data[loc] is always safe
to overwrite
        for(int i = fromIndex ; i < len-1; i++)
            data[i] = data[i+1];
    }
    // Output the results
    printResults(data, n);
    return 0;
}

int find(int nums[], int len, int target)
{
    for(int i=0; i < len; i++)
        if(target == nums[i])
            return i;
    return -1;
}

void shiftOneLeftFrom(int data[], int len, int fromIndex)
{
    // shift items up from loc to n
    // invariant: data[loc] is always safe
to overwrite
    for(int i = fromIndex ; i < len-1; i++)
        data[i] = data[i+1];
}

void printResults(int data[], int len){
    for(int i=0; i < len; i++){
        cout << data[i] << " ";
    }
    cout << endl;
}
```
#include <iostream>
using namespace std;

// prototypes
int find(int nums[], int len, int target);
void shiftOneLeftFrom(int data[], int len, int fromIndex);
void printResults(int data[], int len);

int find(int nums[], int len, int target) {
    for(int i=0; i < len; i++) {
        if(target == nums[i]){
            return i;
        }
    }
    return -1;
}

void shiftOneLeftFrom(int data[], int len, int fromIndex) {
    for(int i = fromIndex ; i < len-1; i++) {
        data[i] = data[i+1];
    }
}

int main() {
    // setup array with data
    int n, data[100];
    cin >> n;
    for(int i=0; i < n; i++)
        cin >> data[i];
    // now perform the given task
    int val, lead, trail;
    cin >> val;
    int loc = find(data, n, val);
    while ( loc != -1 ) {
        shiftOneLeftFrom(data, n, loc);
        n--;
        loc = find(data, n, val);
    }
    // Output the results
    printResults(data, n);
    return 0;
}