CS103 Unit 1c – Arrays and Functions

ARRAY BASICS

Motivating Example

- Suppose I need to store the grades for all students so I can then compute statistics, sort them, print them, etc.
- I would need to store them in variables that I could access and use
  - This is easy if I have 3 or 4 students
  - This is painful if I have many students
- Enter **arrays**
  - Collection of many variables referenced by one name
  - Individual elements can be accessed with an integer index

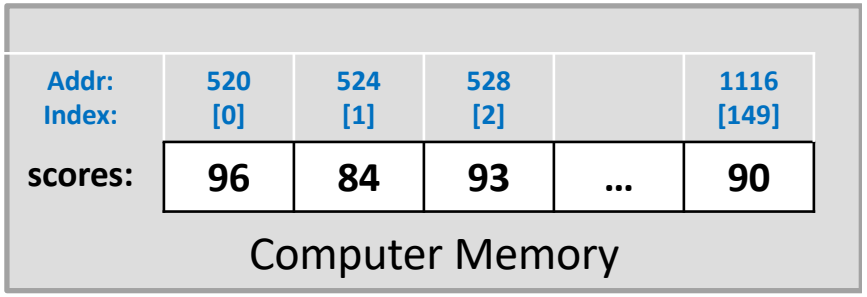
```
int main()
{
  int score1, score2, score3;
  cin >> score1 >> score2 >> score3;

  // output scores in sorted order
  if(score1 < score2 &&
     score1 < score3)
  {  /* score 1 is smallest */ }

  /* more */
}
```

```
int main()
{
  int score1,  score2,  score3,
      score4,  score5,  score6,
      score7,  score8,  score9,
      score10, score11, score12,
      score13, score14, score15,
      /* ... */
      score149, score150;
  cin >> score1 >> score2 >> score3
      >> score4 >> score5 >> score6
      /* ... */
```

Array Basics

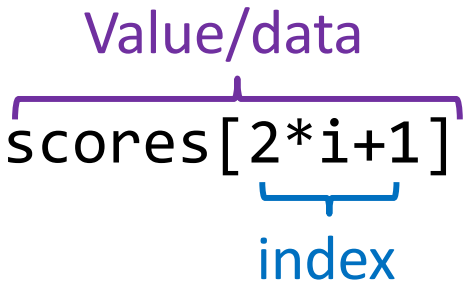
- An array is **a fixed size, named collection** of **ordered variables** of the **same type** that are accessed with an index and stored contiguously in memory
  - **Fixed size:** Cannot grow or shrink
  - **Named collection**: One *name* to refer to all variables in the array
  - **Ordered / Accessed with an index**: Individual element (variable) is accessed with its position/index (using [] brackets)
  - **Same Type**: Variables in one array must all be the same type (one array can't store doubles and ints)

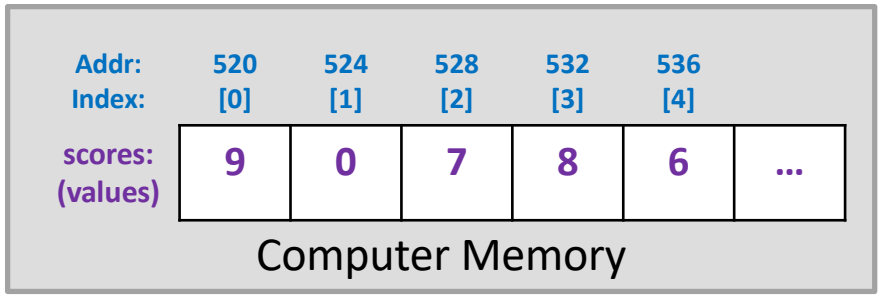
```
int main()
{
  int scores[150];
    // allocates 150 integers
    //  with garbage values

  // Initialize the array
  for(int i=0; i < 150; i++){
    cin >> scores[i];
    // or scores[i] = 0;
  }
}
```



Index vs. Value

- The expression in the square brackets is an index
- Using *array[index]* yields the data/value in the array at that index
- An index can be **ANY EXPRESSION**, even the value from an array or the return value from a function
- For an array declared to be size **n**, only indices **0 to n-1** are legal






```
int main()
{
  int scores[20];
  /* ... fill in the data ... */

  int i = 1;
  int x = scores[2*i + 1];      // x=8
  int y = scores[ scores[1] ]; // y=9
  int z = scores[max(4,2)];     // z=6
  return 0;
}
```

Array Intro (1)

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

int main() {
    // What are the initial values of the array?
    int data1[5];
    cout << "Data1: ";
    for(int i=0; i < 5; i++) { cout << data1[i] << " "; }
    cout << endl;

    // What size will be inferred for this array
    int data2[] = {103, 104, 170, 201, 270};
    cout << "Data2: ";
    for(int i=0; i < 5; i++) { cout << data2[i] << " "; }
    cout << endl;

    // What happens if you try to initialize elements in the array
    // but provide too few?
    int data3[5] = {103, 104, 105};
    cout << "Data3: ";
    for(int i=0; i < 5; i++) { cout << data3[i] << " "; }
    cout << endl;

    // ...
    return 0;
}
```

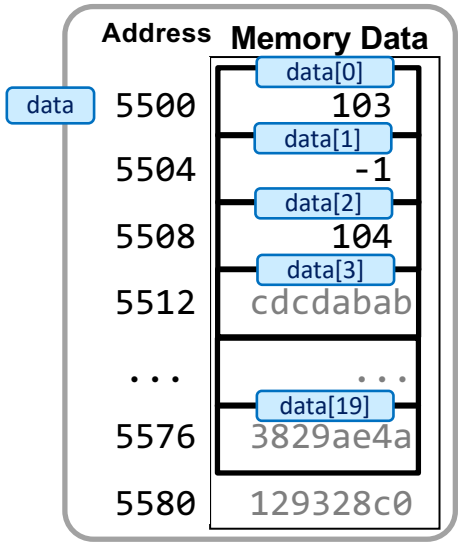
Array Intro (2)

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

int main() {
    // ...
    // Try to access out of bounds
    cout << "Out of bounds (5th element) " << data2[5] << endl;
    // Try to access out of bounds
    cout << "Out of bounds (-2nd element) " << data2[-2] << endl;
    // Try to access out of bounds
    // cout << "Out of bounds (millionth index) " << data2[1000000] << endl;

    // Try to uncomment and compile this.
    /*
    int n;
    cin >> n;
    int data4[n];

    for(int i=0; i < n; i++){
        cin >> data4[i];
    }

    cout << "Data4: ";
    for(int i=0; i < n; i++) { cout << data4[i] << " "; }
    cout << endl;
    */
    return 0;
}
```

Allocating and Accessing Arrays

- Arrays allow you to allocate a large number of variables in a single step…but you can still only access one element at a time
  - Recall: Computers can see and work with 1 data value at a time
- **Step 1**: Allocate the array for a SPECIFIC, FIXED size
  - Specifies the type, a name for the collection, and how many should be allocated
  - Values will be garbage, if not initialized
- **Step 2**: Use the individual array elements as if they were normal variables but remember to use the square bracket indexing syntax (e.g. `data[0]`)
  - **Loops** provide a nice way to process all items one at a time!

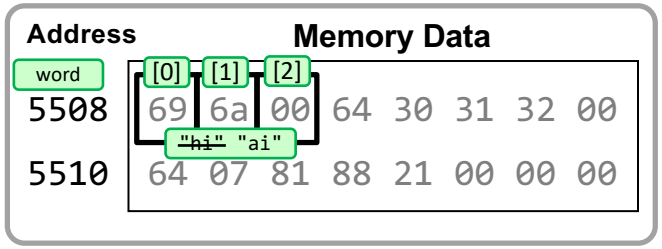
```
int data[20];
data[0] = 103;
data[1] = -1;
data[2] = data[0]+1;
```



Just as a dormitory is known by one name ("McCarthy") but has many dorm rooms, each with a number to identify them ("McCarthy 234")…

…arrays have one name for the whole collection of variables and uses integer indexes to specify a particular element.

```
char word[3] = "hi";
word[0] ='a';
```



Initializing Arrays With Constants

- Arrays can be initialized with **constants** when they are declared
- To do so, use an initialization list which is a comma separated list of constants in {…}
  - **Exception to the minimalist C/C++ rule:** If fewer values are provided than the size of the array, remaining elements will be filled with 0s
- If an initialization list is provided you need not specify the size in the square brackets (i.e. just use empty []) as the compiler can figure out what size the array must be by counting the initial values

```
int main()
{
  int data[5] = {9, 7, 8, 9, 5};

  double dec[4] = {0.25, 0.3};

  char str1[3] = {'C', 'S', '\0'};
  // For char arrays easier to use ""
  char str2[3] = "CS";
  // str2 initialization is same as str1
}
```

Index:	[0]	[1]	[2]	[3]	[4]
data:	9	7	9	9	5

dec:	0.25	0.3	0	0

str1:			

```
int main()
{
  int data[] = {9, 7, 8, 9, 5};
    // allocates array of size 5

  double dec[] = {0.25, 0.3, 0.18, 0.2};
    // allocates array of size 4

  char str2[] = "CS";
    // allocates array of size 3
}
```

Index:	[0]	[1]	[2]	[3]	[4]
data:	9	7	9	9	5

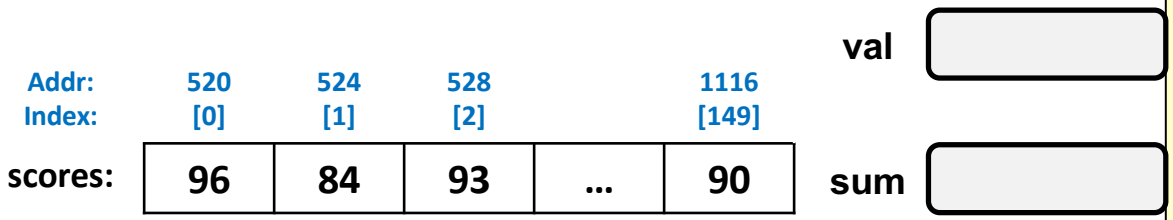
Specifying sizes is not necessary when using initial values list

When Do We Need Arrays?

- You may think an array is needed any time we need to process a sequence of many related data items of the same type
- But a better question is when do we need to **store** these related data items in an array?
- Answer: When we need to revisit the data more than once
  - If we just want to find the min/max or average we could just get the data from the user and update the sum or min/max as we go and not need to store each data item
  - Don't introduce arrays where they are not needed

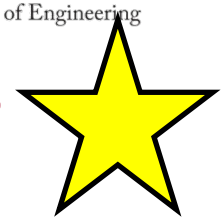
```
int main()
{
  int scores[100];
  // Get the data
  for(int i=0; i < 100; i++){
    cin >> scores[i];
  }
  // Average all values
  int sum = 0;
  for(int i=0; i < 100; i++){
    sum += scores[i];
  }
  cout << sum / 100.0 << endl;

  return 0;
}
```

```
int main()
{
  int val, sum = 0;
  // Get the data & average it
  // at the same time
  for(int i=0; i < 100; i++){
    cin >> val;
    sum += val;
  }
  cout << sum / 100.0 << endl;
  return 0;
}
```



C/C++ ARRAY SIMILARITIES AND DIFFERENCES WITH OTHER LANGUAGES

Coming From Other Languages



- **SIMILARITIES**: Like Python and Java, C/C++ arrays
  1. Use 0-based indexing (beginning element at index 0)
  2. Pair nicely with loops that can iterate over all the elements of an array
- **DIFFERENCES**: Unlike Python and Java, C/C++ arrays
  3. Are fixed size (size must be a constant) and then cannot grow easily after that
  4. Do not remember their size (no `len()` or `.length`) nor bounds-check an access (so accessing `scores[1048726]` will happily execute in C/C++ and likely cause a crash (aka the dreaded "***Segmentation Fault***")
  5. Are NOT objects (no `.append()` or `.length`) in C/C++, but degenerate to **pointers** (more to come soon)

```java
import java.util.Scanner;

class Scores {
  public static void main(String[] args)
  {
    Scanner in = new Scanner(System.in);

    int[] scores = new int[150];
    for(int i=0; i < scores.length; i++){
        scores[i] = in.nextInt();
    }
    // Do something with scores

  }
}
```

Java

```python
def main():
    scores = [0]*150
    for(i in range(len(scores)):
        scores[i] = int(input())
    # Do something with scores

# Execution starts here (weird)
if __name__ == "main":
    main()
```

Python

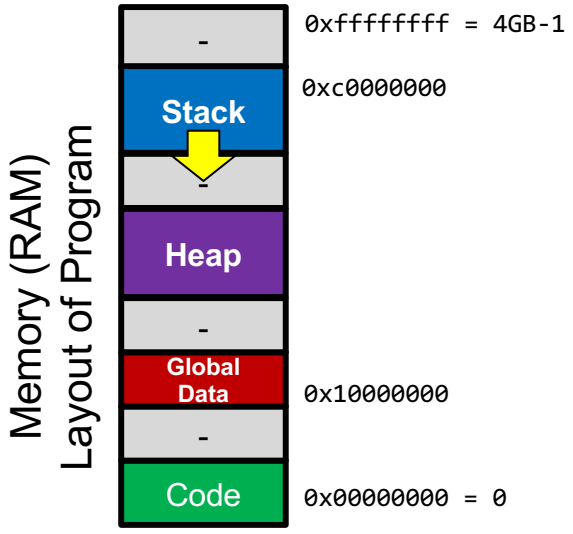
```cpp
int main()
{
  int scores[150];
    // allocates 150 integers
    //  with garbage values


  for(int i=0; i < 150; i++){
    cin >> scores[i];
  }
  // Do something with scores

}
```

C++

Fixed (Statically) Sized Arrays

- C/C++ needs to know the size of the array when the program is **compiled (statically)**, not when it is **run (dynamically)**.
- This implies the size of the array must be **ONE, FIXED (or constant) size** everytime the program is run
- What could go wrong if we did allow the variable-size array allocation?
  - Why don't hotels let you wait and specify how many rooms you want once you arrive?
  - Too large an allocation can cause "stack overflow" and corrupt the program

```
int main()
{
  // GOOD!!
  int data[42];   // 42 known at
                  // compile time

  // BAD!!
  int n;
  cin >> n;
  int data[n];    // n not known at
                  // compile time
}
```



Dealing With Variable Size Arrays

- C/C++ needs to know the size of the array when the program is **compiled**, not when it is **run**.
- Two approaches if we cannot know the necessary size at compile time:
  - Allocate a LARGE array of the maximum size potentially needed and then use only a portion of it as the program runs
  - (***Preferred***) Use dynamic memory (i.e. the new operator) to allocate a variable size array
    - **Major topic of discussion in a future unit (don't worry about it for now)**

```
int main()
{
  int data[100]; // max needed
  int n;
  cin >> n;

  for(int i=0; i < n; i++)
  {
    cin >> data[i]; // only use n
  }
}
```

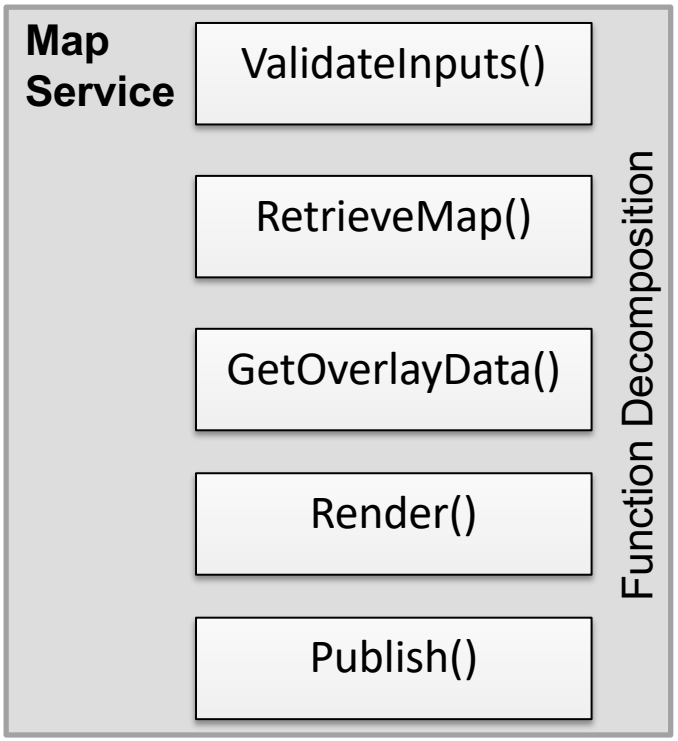
```
int main()
{
  int n;
  cin >> n;

  int* data = new int[n];
  for(int i=0; i < n; i++)
  {
    cin >> data[i]; // only use n
  }
  delete [] data; // needed cleanup
}
```

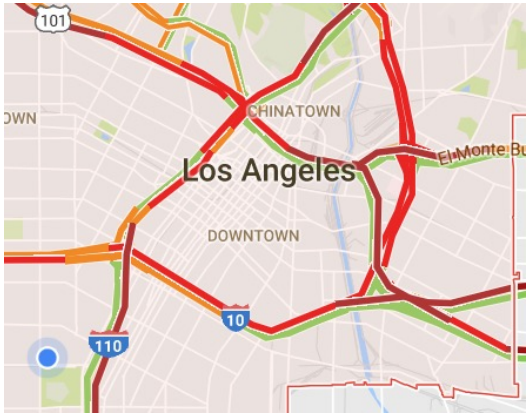
This approach will be discussed in detail in a future unit.

FUNCTIONS: A QUICK LOOK

Functions Overview

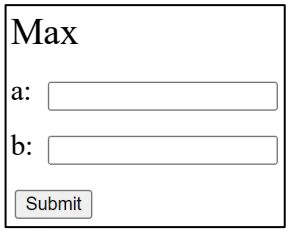
- **Functions** (aka procedures, subroutines, or methods) are the primary unit of code decomposition and abstraction in C/C++
  - Decomposition: Breaking programs into smaller units of code
  - Abstraction: Generalizing an action or concept without specifying how the details are implemented

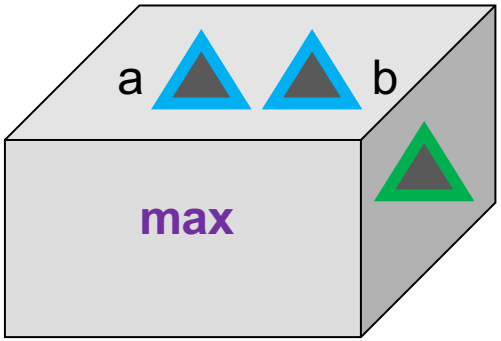




Function Signatures/Prototypes

- Also called *procedures* or *methods*
- We think of a function as a blackbox (don't know or care how it does the task internally) of code where **we can provide inputs and get back a value**
  - Or think of it as a web-app (or form) where you supply data to "named" inputs and get back a value
- In C/C++, a function has:
  - A **name**
  - Zero or more input parameters
  - 0 or 1 return (output) values
    - We only specify the type
    - 0 return values is indicated with `void` type
- The signature (or **prototype**) of a function specifies these aspects so others know how to "call" the function






```
int max(int a, int b);
```

Function Signature/Prototype

User Defined Functions

- We can define our own functions in 3 steps
- **Step 1**: "Declare" your function by placing the **prototype** (**signature**) at the top of your code
- **Step 2:** "Define" the function (actual code implementation) *anywhere* (above or below main()) by placing the code in { }
- **Step 3**: "Call" the function from main() or another function passing in desired inputs and using the return value (output)

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

int max(int a, int b); // prototype

int main()
{
  int x, y, mx;
  cin >> x >> y;

  /* Code for main */



}

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

Functions Intro

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

// Function prototypes
void printName(string name);
int factorial(int n);

// Function definitions
int main()
{
    printName("Tina");
    cout << factorial(4) << endl;
    return 0;
}

void printName(string name)
{
    if(name == ""){
        return;
    }
    cout << "Hello, " << name << endl;
}
```

```
int factorial(int n)
{
    if(n >= 0){
        if(n == 0) { return 1; }
        int f = 1;
        for(int i = 1; i <= n; i++) {
            f *= i;
        }
        return f;
    }
    // Return some value that will mean "error"
    return -1;
}

// Cannot have 2 return values
// double, double getCirclePerimAndArea(
//    double radius)
// {
//     return 2*M_PI*radius, M_PI*radius*radius;
// }
```

Calling a Function

- We "call" or "invoke" the function by:
  - Using its name and place variables or constants that the current function has declared in the order that we want them to map to the parameter/argument list
  - First variable listed (x) will map to the first parameter (a) in the function's argument list, the second variable (y) to the second parameter (b), etc.
- Don't
  - Relist the return type in the call
  - Relist the type of the arguments
  - Use variable names that don't exist in the current function
  - Forget to save the returned value

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

int max(int a, int b); // prototype

int main()
{
  int x, y, mx;
  cin >> x >> y;

  /* Call the function */
  mx = max(x, y);
  cout << mx << endl;

  /* Bad */
  mx = int max(x, y);
  mx = max(int x, int y);
  mx = max(a, b);
  max(x, y);
}

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

Calling a Function (2)

- Statements in a function are executed sequentially by default
- Defined once, called over and over
- Functions can call other functions can call other functions
- Example: Compute max of two integers

  Each call causes the program to pause the current function, go to the called function and execute its code with the given arguments then return to where the calling function left off,
- Return value is substituted in place of the function call

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

int max(int a, int b); // prototype

int main()
{
  int x, y;
  cin >> x >> y;  // User types: -5 7
                  7
  int mx = 1 + max(x, y); // call max
  cout << mx << endl;
                  0
  cout << max(0, x) << endl; // call max
}

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

Program Output (if user types -5 7):

```
8
0
```

Passing Arrays As Arguments

- Syntax:
  - **Step 1**: In the prototype and function definition:
    - Put empty square brackets **[ ]** after the **formal** parameter name if it is an array (e.g. `int data[]`) ..OR..
    - Put an ***** between the type and **formal** parameter name (e.g. `int* data`)
    - We'll prefer `int data[]` for now but `int* data` is **JUST AS VALID** and we'll learn more about it when we cover pointers)
  - **Step 2**: When you call the function, just provide the name of the array as the **actual** parameter

```
// Prototype
int init(int data[], int max_size);

int main()
{
  int vals[100];
  int len = init(vals, 100);
  // some code to process the input
  // in the vals array
  for(int i=0; i < len; i++) {
    cout << vals[i] << endl;
  }
  return 0;
}

int init(int data[], int max_size)
{
  int i=0, num;
  cin >> num;
  while( i < max_size && num != -1) {
    data[i] = num;
    i++;
    cin >> num;
  }
  return i;
}
```

MORE FUNCTION DETAILS

Function Prototypes

- The compiler (g++/clang++) needs to see a function's **prototype** or **definition** before it allows a **call** to that function
- The compiler will scan a file from top to bottom
- If it encounters a **call** to a function before the actual function **definition** it will complain…[Compile error]
- …Unless a **prototype** ("declaration") for the function is provided earlier
- A prototype only needs to include data types for the parameters but not their names (ends with a ';')
  - Prototype is used to check that you are calling it with the correct syntax (i.e. parameter data types & return type) (like a menu @ a restaurant)

X

```
int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}

double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler encounters a call to triangle_area() before it has seen its definition (Error!)

```
double triangle_area(double, double);

int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}

double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler sees a prototype and can check the syntax of any following call and expects the definition later.

The Need For Prototypes

- You might say:
  - "I don't like prototypes. I'll define each function before I call it"
- How would you order the functions in the program on the left if you did NOT want to use prototypes?
  - You can't!

```
int f1(int x)
{
   return f2(x-1);
}

int f2(int y)
{
  if(y <= 0) return 1;
  else return f1(y);
}

int main()
{
  cout << f1(5) << endl;
}
```

```
int f1(int x);
int f2(int y);

int main()
{
  cout << f1(5) << endl;
}

int f1(int x)
{
   return f2(x-1);
}

int f2(int y)
{
  if(y <= 0) return 1;
  else return f1(y);
}
```

Overloading: A Function's Signature

- What makes up a function signature **unique**:
  - **name**
  - **number** and **type** of arguments
- No two functions are allowed to have the same signature
- The following 6 functions are **unique** and allowed to have different implementations…
  - `int f1(int), int f1(double), int f1(int, double)`
  - `void f1(char), double f1(), int f1(int, char)`
- Return type does not make a function unique
  - `int f1()` and `double f1()` are not unique and thus not allowable
- Two functions with the same name are said to be "**overloaded**"
  - `int max(int, int); double max(double, double);`

Why Functions? Reuse (1)

Desired Program Output:

```
///
//
/
/////
////
///
//
/
```

- Functions are best used to perform code that would otherwise have to be duplicated
- By "factoring" common code into its own function and possibly parameterizing it we can make flexible, reusable blocks of code

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


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

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

  return 0;
}
```

Why Functions? Reuse (2)

Desired Program Output:

```
///
//
/
/////
////
///
//
/
```

- Here we have factored the common code into its own function parameterized based on how many rows are desired

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

void printTri(int rows);

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

void printTri(int rows)
{
  for(int i=0; i < rows; i++){
    for(int k=0; k < rows-i; k++){
      cout << '/';
    }
    cout << endl;
  }
}
```

Reuse

- We could create 1 or 2 functions to do this job
- How could defining printRow allow for reuse if chose to draw a different shape (like a square)?
- What else could we parameterize that might make this code more reusable?
  - The fill character (' / ')
- But don't go too crazy

Program Output:

```
///
//
/
/////
////
///
//
/
```

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

void printRow(int n);     // prototype
void printTri(int rows);  // prototype

int main()
{
  printTri(3);
  printTri(5);
  return 0;
}
void printTri(int rows)
{
  for(int i=0; i < rows; i++){
    printRow(rows-i);
  }
}
void printRow(int n);
{
  for(int i=0; i < n; i++){
    cout << '/';
  }
  cout << endl;
}
```

Review: Program Decomposition

- C is a **procedural** language
  - A **function** or **procedure** is the primary unit of code organization, problem decomposition, and abstraction
  - Function is a unit of code that
    - Can be called from other locations in the program
    - Can be passed variable inputs (a.k.a. arguments or parameters)
    - Can return a value to the code that called it
  - Can be reused
- C++ is considered an **object-oriented** language (adds objected-oriented constructs to C) though still supports a procedural approach
  - A **class** or **object** is the primary unit of code organization, problem decomposition, and abstraction
  - Can be reused

Exercise

- To decompose a program into **functions**, try listing the ***verbs*** or ***tasks*** that are performed to solve the problem
  - Model a card game as a series of tasks/procedures…
  - A database representing a social network

Nested Call Practice

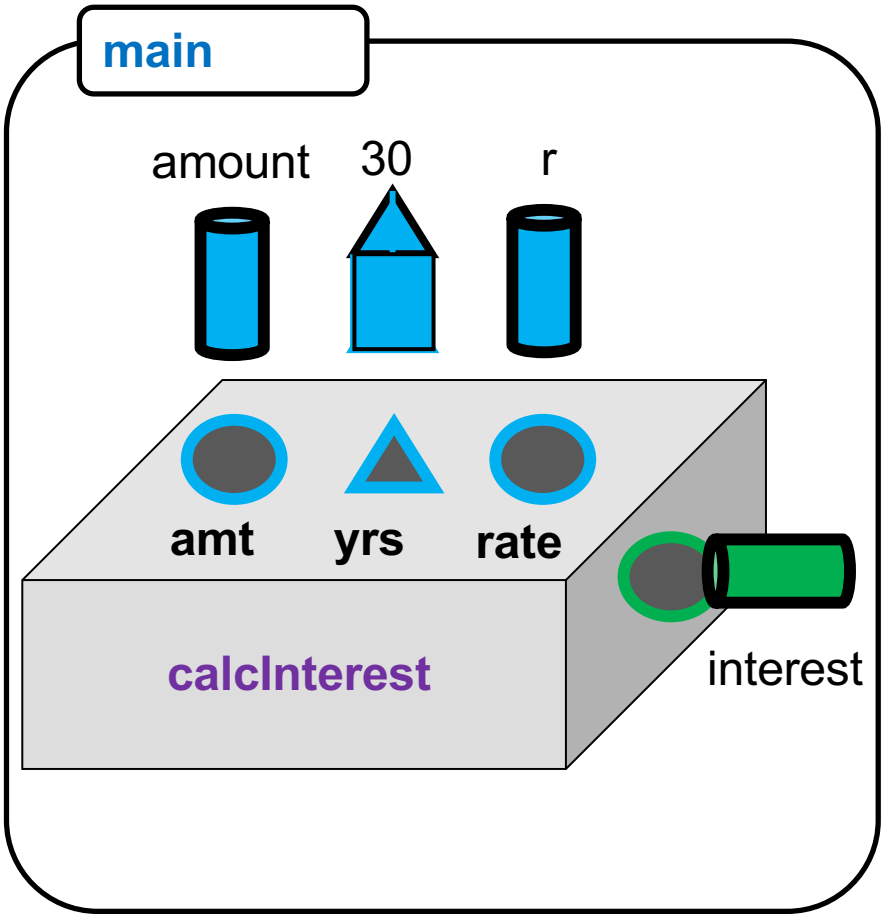
- Find characters in a string then use that function to find how many vowels are in a string
  - Exercise: draw-square
  - Exercise: vowels

MORE FUNCTION EXAMPLES

Example Functions 1

Function Signature/Prototype

```
double calcInterest(double amt, int yrs, double rate);
```

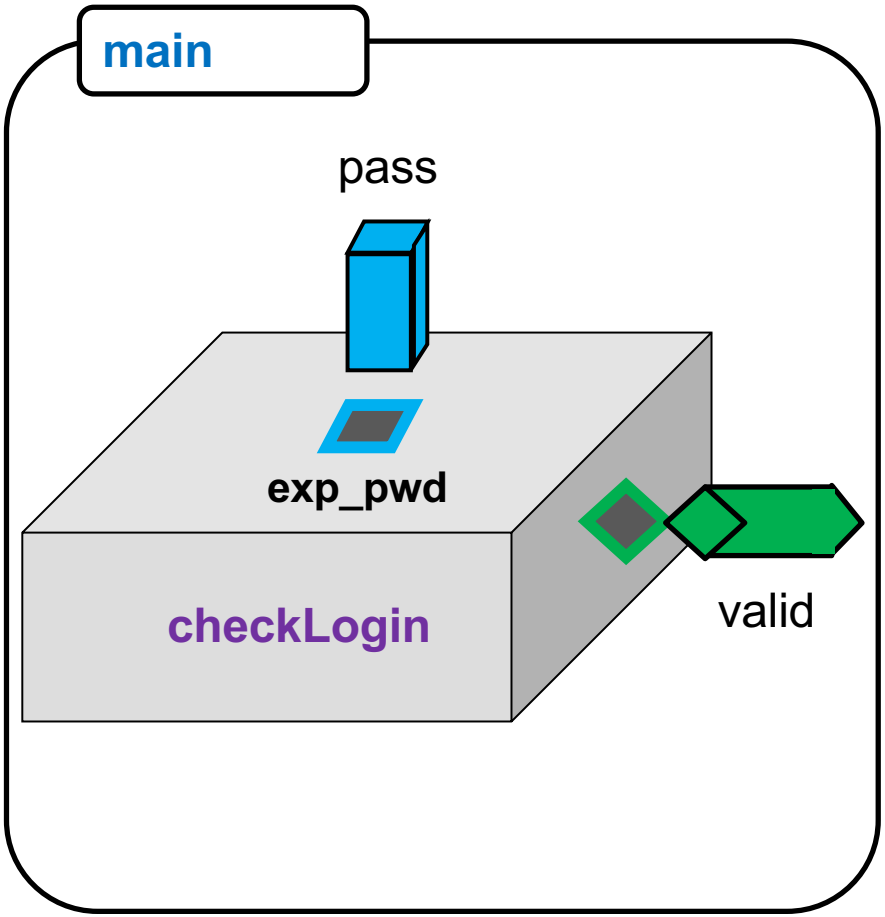



```
#include <iostream>
#include <cmath>
using namespace std;

// prototype
double calcInterest(double amt, int yrs, double rate);

int main()
{
  double amount, r;
  cin >> amount >> r;

  double interest = calcInterest(amount, 30, r);
  cout << "Interest: " << interest << endl;
  return 0;
}

double calcInterest(double amt, int yrs, double rate)
{
  return amt * pow(rate/12, 12*yrs);
}
```

Example Functions 2

Function Signature/Prototype

```
bool checkLogin(string exp_pwd);
```

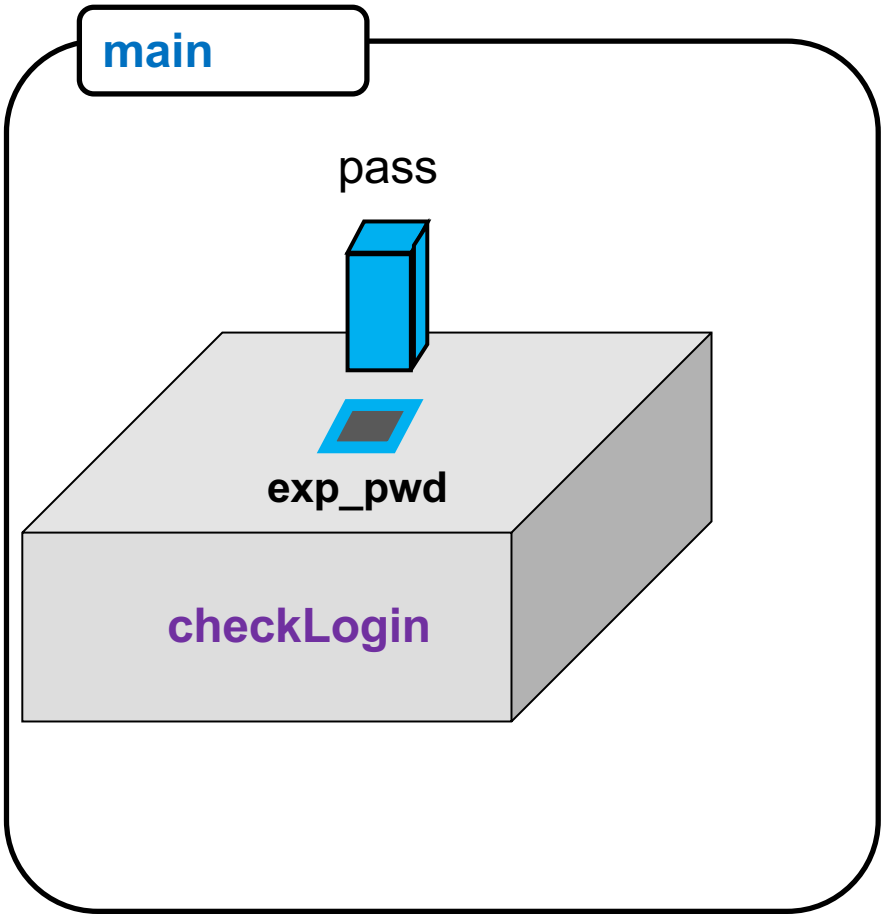



```
#include <iostream>
using namespace std;

// prototype
bool checkLogin(string exp_pwd);

int main()
{
  string pass = "Open123!"; // secret password
  bool valid;

  cout << "Enter your password: " << endl;
  valid = checkLogin(pass);
  if(valid == true) { cout << "Success!" << endl; }
  return 0;
}

bool checkLogin(string exp_pwd)
{
  string actual;
  cin >> actual;
  return actual == exp_pwd;
}
```

Example Functions 3

Function Signature/Prototype

```
void validateLogin(string exp_pwd);
```

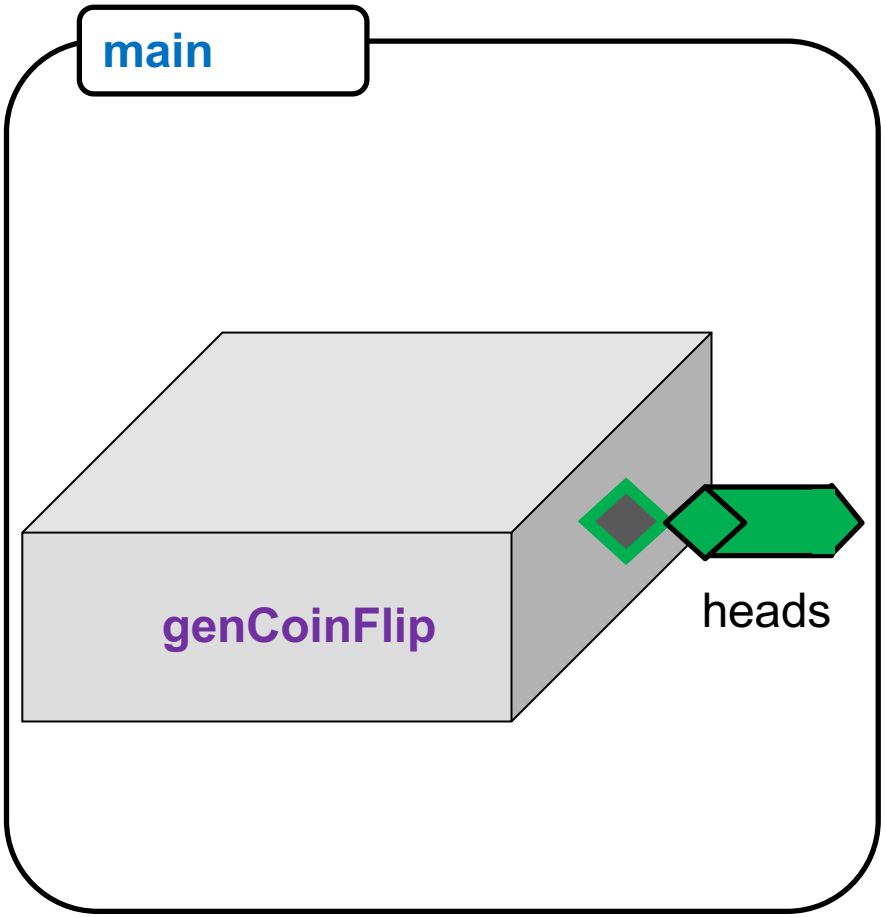



```
#include <iostream>
using namespace std;

// prototype
void validateLogin(string exp_pwd);

int main()
{
  string pass = "Open123!"; // secret password
  bool valid;

  cout << "Enter your password: " << endl;
  validateLogin(pass);
  return 0;
}

void validateLogin(string exp_pwd)
{
  string actual;
  cin >> actual;
  if(actual == exp_pwd){ cout << "Success!" << endl; }
  else { cout << "Incorrect!" << endl; }
}
```

Example Functions 4

Function Signature/Prototype

```
bool genCoinFlip();
```




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

// prototype
bool genCoinFlip();

int main()
{
  bool heads;

  heads = genCoinFlip();
  if(heads == true) { cout << "Heads!" << endl; }
  else { cout << "Tails!" << endl; }
  return 0;
}

bool genCoinFlip()
{
  int r = rand(); // Generate random integer
  return r%2;
}
```

SOLUTIONS

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

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

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