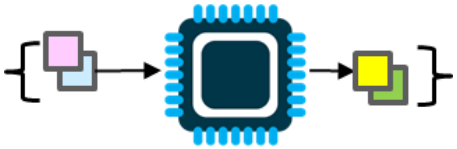
Unit 4b

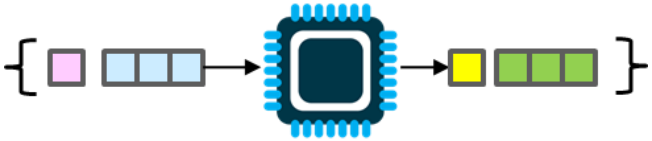
Writing Functions

Unit 4

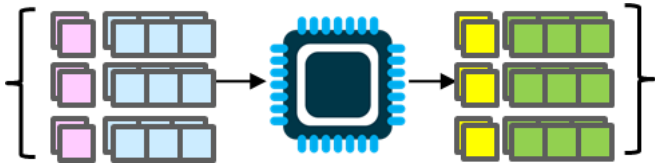
- **Unit 1**: Scalar processing
  - aka IPO=Input-Process-Output Programs



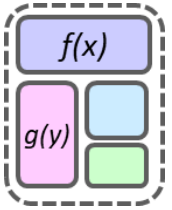
- **Unit 2**: Linear (1D) Processing



- **Unit 3**: Multidimensional Processing

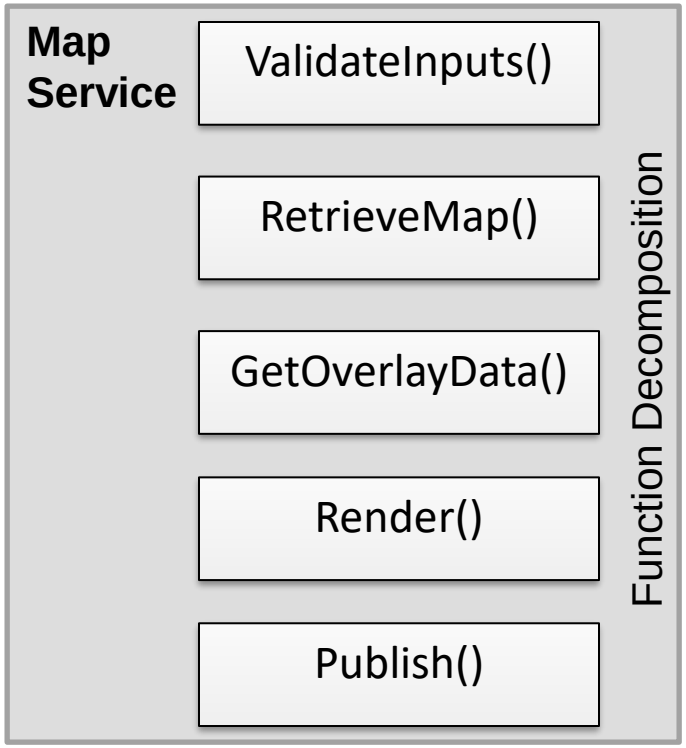


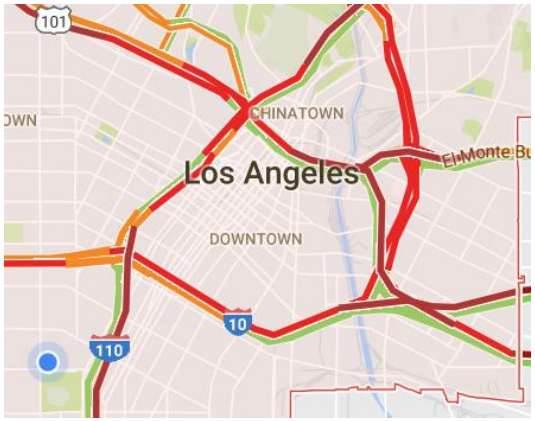
- **Unit 4**: Divide & Conquer (Functional Decomposition)



Functions Overview

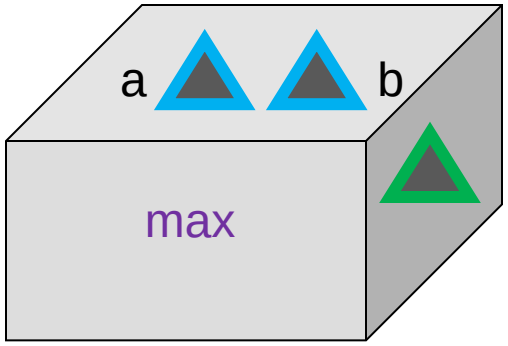
- Functions (aka procedures, subroutines, or methods) are the unit of code decomposition and abstraction
  - Decomposition: Breaking programs into smaller units of code
  - Abstraction: Generalizing an action or concept without specifying how the details are implemented





Function Signatures/Prototypes

- We think of a function as a blackbox (don't know how it does the task internally) where we can provide inputs and get back a value
- A function has:
  - A name
  - Zero or more input parameters
  - 0 or 1 return (output) values
    - We only specify the type
- The signature (or **prototype**) of a function specifies these aspects so others know how to "call" the function



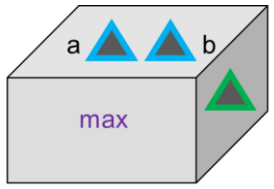

```
int max(int a, int b);
```

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
}
```




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

Calling a Function (1)

- 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);

  /* 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)

- The we can "call" (activate/invoke) our function from any other location in our code as many times as we like
- Semantics of a function call:
  - Pause the caller code (i.e. main)
  - Pass copies of the arguments to the function (i.e. pass a copy of x and y to a and b)
  - Let the function execute
  - When the function completes we return back to the caller (i.e. main) and resume execution
  - Any 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

  int mx = 1 + max(x, y); // call max
  cout << mx << endl;

  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
}
```

7

0

Program Output (if user types -5 7):

```
8
0
```

Execution Timeline

1. We always start at main() and execute sequentially until a function call or the end of main()
2. When we hit a function call, execution of main pauses and execution of max begins
3. Max will compare the arguments and return the bigger
4. Upon return we go back to where we left off in the previous function and replace the function call with the return value
5. We continue to evaluate the expression in which the function call was made (i.e. 1 + max)
6. We assign the result to mx
7. We continue sequentially until another function call or until the end of main()
8. Another function call again causes main to pause and max begins execution anew
9. Max compares arguments
10. It then returns to the caller (main) and substitutes its return value in the larger expression

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

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

int main()
{
1 int x, y, mx;
  cin >> x >> y;  // User types: -5 7
  6        5    7
  int mx = 1 + max(x, y); // call max
7 cout << mx << endl;
             0
2 cout << max(0, x) << endl; // call max
}   8

int max(int a, int b)          10
{                                  4
3 9 if(a > b)
      return a; // immediately stops max
  else
      return b; // immediately stops max
}
```

Program Output (if user types -5 7):

```
8
0
```

Why Functions? (1)

Desired Program Output:

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

- Functions are best use 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 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;
}
```

Why Functions? (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 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 Calling Functions

- We could create 1 or 2 functions to do this job (probably could all be done in printFlag but we want you to see how one function can call another)
- Anytime a function calls another, the caller pauses and the called function begins
- When a function ends it returns to the previous function (the one that called it)

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

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

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

Program Output:

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

Formal and Actual Parameters

TWO SETS OF ARGUMENT NAMES

Argument Names

- A script (play or movie) could be produced in many different places each with **different casts** (actual actors and actresses).
- Thus, the script is written in terms of ***character names*** but when the cast is chosen, ***actual people's names*** *are mapped/substituted.*
- **In the play**, everything is setup to use the ***character names*** (the audience wouldn't know what's going on if they use the actresses real name).
- Before or after the play, people's ***actual (real) names*** are used to refer to people
- The same thing happens with arguments passed to a function.

Julius Caesar

Cast

Character	Actor
Casca/Octavius' Solider	Alexis Lagon
Julius Caesar	Alec Mallmann
Portia/Octavius' Solider	Maddie Baylor
Marullus/First Citizen/Artemidorus/Octavius' Solider	Madelynn Collier
Cassius	Emily Kile
Decius Brutus	Echo Bartholomew
Lucius/Cinna the Poet/Brutus' Solider/Citizen	Nathaniel Huff
Claudius/Soothsayer/Servant to Caesar/Octavius' Servant	Sterling Barbett
Calpurnia/Varro/Fourth Citizen	Meghan Hinkson
Mark Antony	Seth Drenning
Lepidus/Caius Ligarius/Octavius' Solider	Emma Hastings
Trebonius/Citizen	Niko Salinas
Flavius/Third Citizen/Popilius Lena	Ethan Pearson
Cinna/Cobbler	David Stout
Metellus Cimber/Citizen	Shannon Hardy
Brutus	Sean Coady
Octavius/Second Citizen	Mason Clark

character names ***real names***

Formal and Actual Argument Names

- In the analogy, the play is a function
- **Formal argument name** (***character names***): The names used **INSIDE** the function code which act as a placeholder to know ***which input is which***)
  - When taking the power would pow(x,y) give the same result as pow(y,x)? No!
  - So we need to know which input is the base and which is the exponent…formal argument names help us know that.
- **Actual argument names** (**names of the actors**): The actual **values and variables in the calling function** that they want to pass or map to the functions formal arguments

Julius Caesar

Cast

Character	Actor
Casca/Octavius' Solider	Alexis Lagon
Julius Caesar	Alec Mallmann
Portia/Octavius' Solider	Maddie Baylor
Marullus/First Citizen/Artemidorus/Octavius' Solider	Madelynn Collier
Cassius	Emily Kile
Decius Brutus	Echo Bartholomew
Lucius/Cinna the Poet/Brutus' Solider/Citizen	Nathaniel Huff
Claudius/Soothsayer/Servant to	Sterling Barbett

Formal Arguments (character names) ***Actual Arguments (real names)***

```
// prototype
double mypow(double base, int exp);

int main() {
  double x=2,y=3;          // actual args
  double result1 = mypow(x,y);
  double result2 = mypow(y,2);
  ...
}
          // formal args
double mypow(double base, int exp)
{ . . . }
```

Mapping Formals to Actuals

- When we call a function, the ACTUAL arguments are assigned into the FORMAL arguments
  - Ex: `formal = actual;`
- The mapping is just based on the ORDER we list the formals in the function signature and the ORDER we list the actuals when we call the function
  - The first formal is assigned the first actual
  - The second formal is assigned the second actual
  - The third formal is assigned the third actual
  - …
- Each time we call the function we can use a different set of "actresses" (i.e. different actuals for each call)

```
#include <iostream>
using namespace std;
// prototype
double mypow(double base, int exp);

int main() {
  double x; int y;
  cin >> x >> y;
                  // actual args
  double result1 = mypow(x,y);
  double result2 = mypow(y,2);
  ...
}

// Playbill for 1st call
base = x;
exp = y;

// Playbill for 2nd call
base = y;
exp = 2;

          // formal args
double mypow(double base, int exp)
{
  int result = 1;
  for( ; exp != 0; exp--){
     result *= base;
  }
  return result;
}
```

Variable Scope

- "Scope" of a variable refers to the
  - **Visibility** (who can access it) and
  - **Lifetime** of a variable (how long is the memory reserved
- For now, there are 2 scopes we will learn
  - **Global:** Variables are declared ***outside*** of any function and are visible to ***all*** the code/functions in the program
    - For various reasons, it is "bad" practice to use global variables. You MAY NOT use them in CS 102.
  - **Local:** Variables are declared ***inside*** the { } in a function and are ***only*** visible in those { } and ***die*** when the the end brack } is reached

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

// Global Variable
int x=1;

int add_x()
{
  int n; // n is a "local" variable
  cin >> n;
  // y and z NOT visible (in scope) here
  // but x is since it is global
  return (n + x);
} // n dies here
int main()
{
  // y and z are "local" variables
  int y=0, z;

  z = add_x();
  y += z / x;  // n is NOT visible
  cout << x << " " << y << endl;
  return 0;
} // y and z die here
```

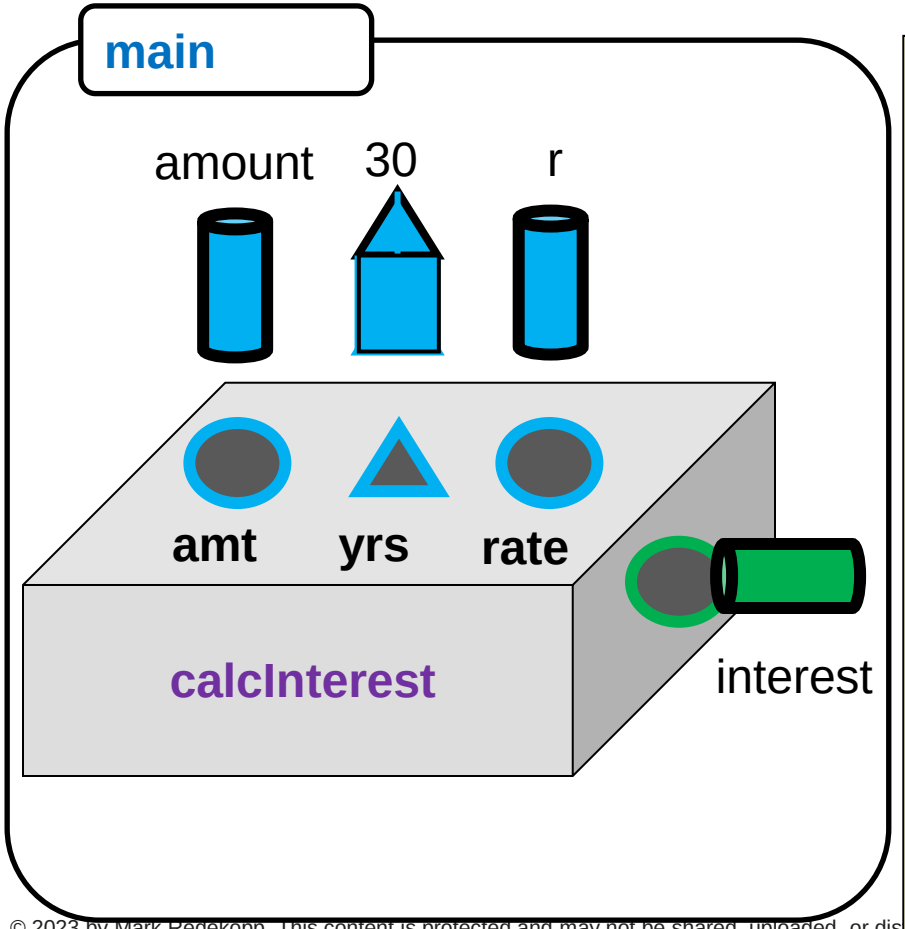
Exercises

- Exercises
  - hypotenuse
  - wakeup

Example Functions 1

Function Signature/Prototype

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

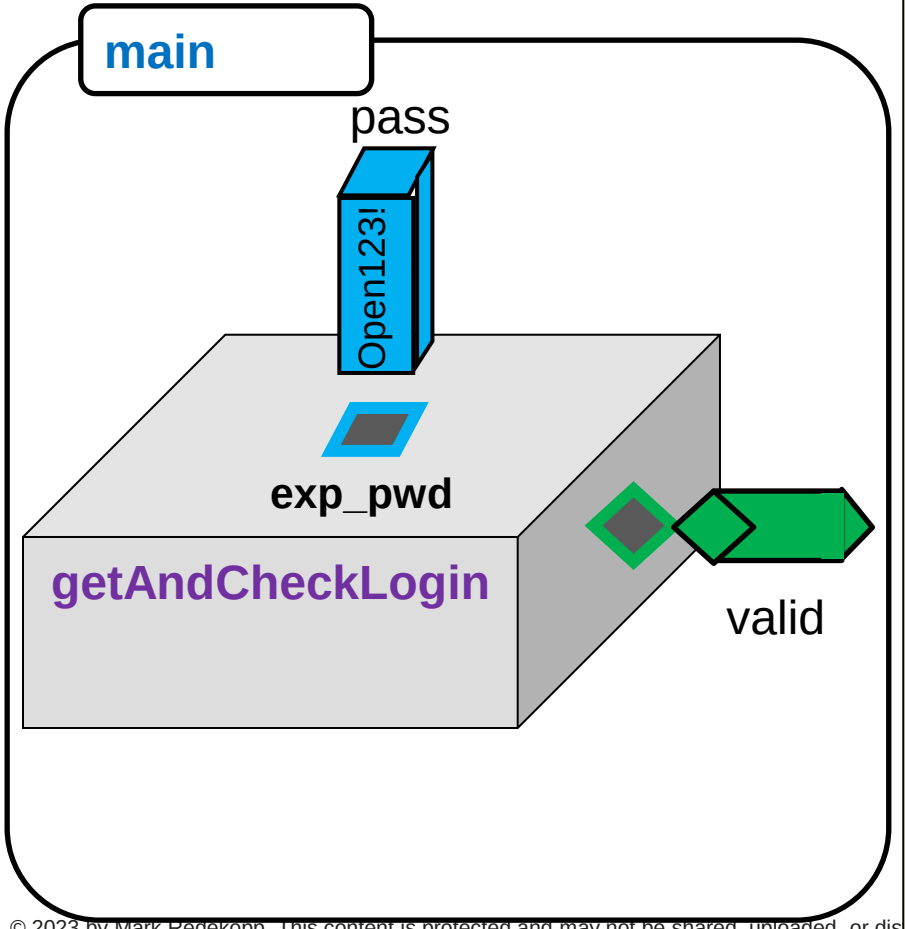



```
#include <iostream>
#include <cmath>
using namespace std;

// 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 getAndCheckLogin(string exp_pwd);
```

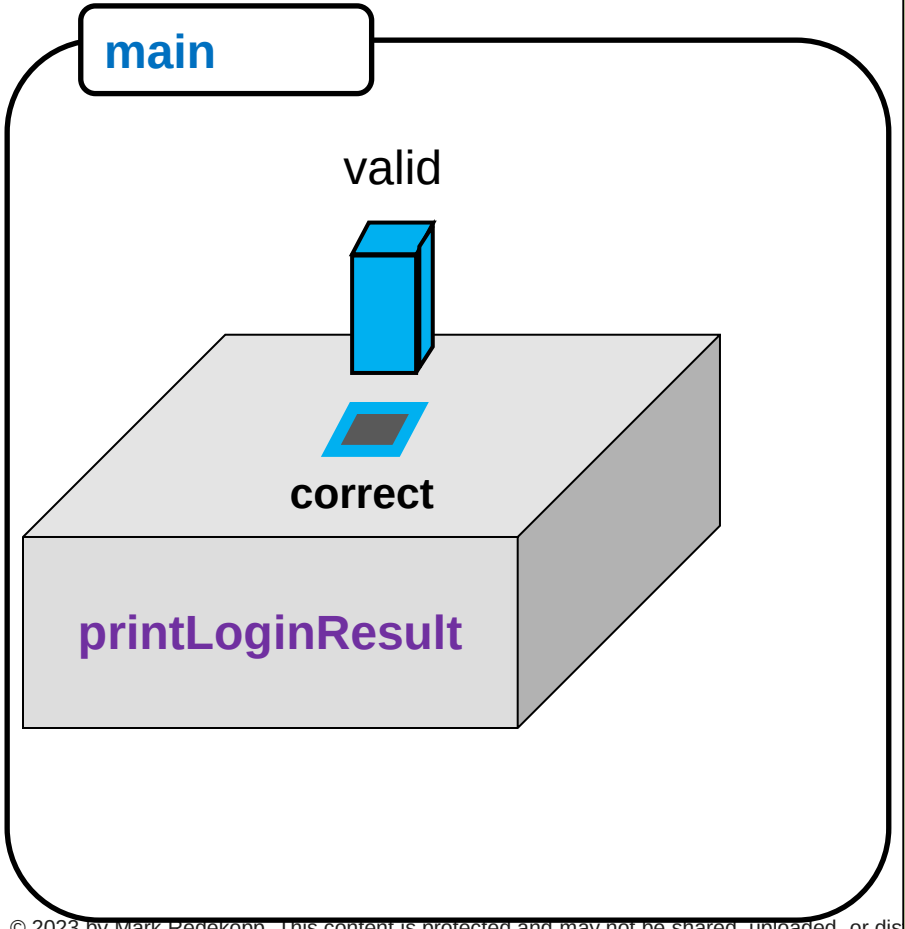



```
#include <iostream>
using namespace std;

// prototype
bool getAndCheckLogin(string exp_pwd);

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

  valid = getAndCheckLogin(pass);
  if(valid == true) { cout << "Success!" << endl; }
  else { cout << "Password mistmatch!" << endl; }
  return 0;
}

bool getAndCheckLogin(string exp_pwd)
{ string pwd;
  cout << "Enter your password: " << endl;
  cin >> pwd;
  return pwd == exp_pwd;
}
```

Example Functions 3

Function Signature/Prototype

```
void printLoginResult(bool correct);
```

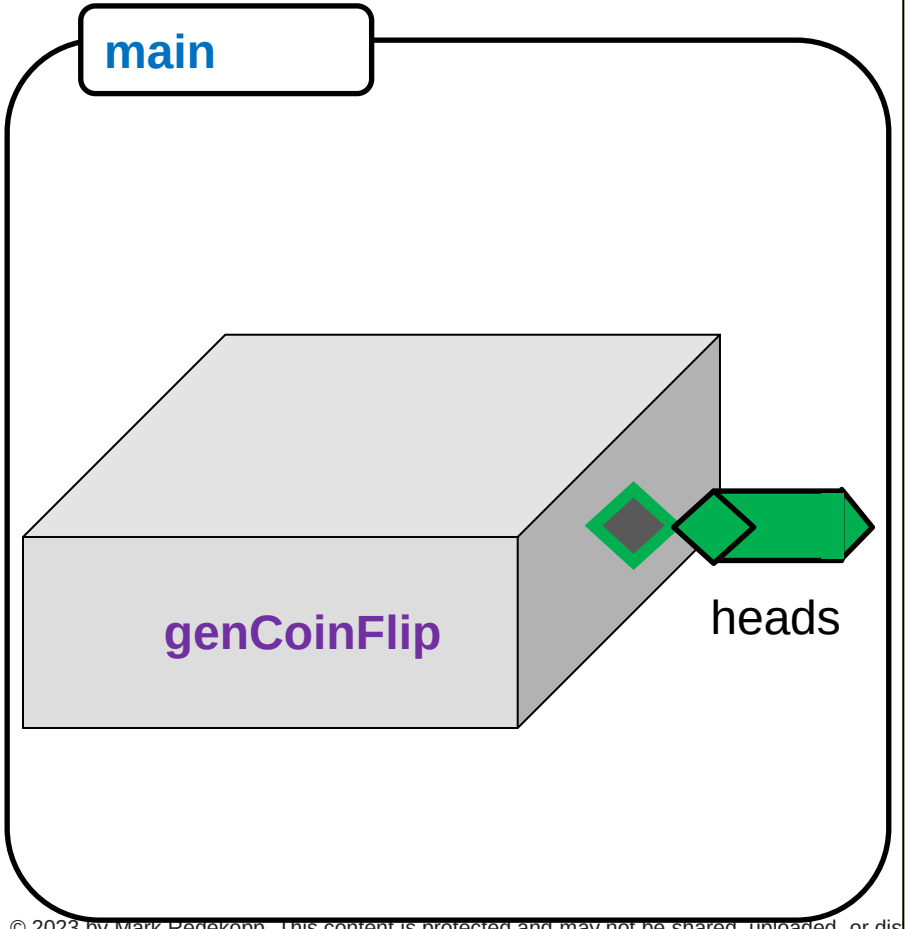



```
#include <iostream>
using namespace std;

// prototype
void printLoginResult(bool correct);

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

  valid = getAndCheckLogin(pass);
  printLoginResult(valid);
  return 0;
}

void printLoginResult(bool correct)
{
  if(correct == true) { cout << "Success!" << endl; }
  else { cout << "Password mistmatch!" << 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) == 1;
}
```

Predicate Functions

- We can write functions that return a bool (true or false) to assert or confirm something about the input
- These functions can then be called in the condition of an `if` statement or a loop

```
bool isNegative(double x)
{
  return x < 0;
}
bool isDigit(char c)
{
  return (c >= '0' && c <= '9');
}
bool __________(int s1, int s2, int s3, int s4)
{
  return (s1 == s2) && (s2 == s3) &&
         (s3 == s4);
}

int main()
{
  int a;  char b;  int f, g, y, z;
  cin >> a >> b >> f >> g >> y >> z;
  if( isNegative(a) ) {
    cout << "Error..neg. #" << endl;
  }
  if( isDigit(b) ) {
    cout << "digit character" << endl;
  }
  if( (f==g) && (g == y) && (y == z) )
  {
    cout << "Yes..._____________" << endl;
  }
  return 0;
}
```