ENGR 101 – Chapter 14

Functions

Recall: What is a Function?

- A function is an **abstraction** over a chunk of computation.
  - i.e. Data goes in, it gets processed, new data comes out.
  - We don't have to worry about how the computation works internally.
- Example: The `sqrt` function

```
x = 16;
y = sqrt(x);
```

- The **interface** for a function describes how we use it:
  - e.g. For `sqrt`: "Give it a number. It gives you back the square root."
  - e.g. For MATLAB's `size`: "Give it an array. It gives you back its dimensions."
- The **implementation** contains code to make the function work.

C++ Function Examples: square

- Let's write a simple function to square a number:

```
// Returns the square
// of the given number
int square(int n) {

  return n * n;

}
```

It returns an `int` value.

It takes an `int` parameter. A parameter is a variable you can use in the function code.

The implementation:
Multiply n by itself and return the result.

C++ Function Examples: abs

- Finding the absolute value of a number:

```
// Returns the absolute value of the given number
int abs(int n) {
  int a;
  if(n >= 0) {
    a = n;
  }
  else {
    a = -n;
  }
  return a;
}
```

A function's implementation can contain any kind of code.

Unlike MATLAB, C++ requires an explicit `return` statement.

Note: The code in this implementation can be improved. See later slides for details.

Calling Functions

Start with a driver program.
In C++ this is often the `main` function.

- Calling functions in C++ is similar to MATLAB.

```
int main() {
  int x = 3;
  int y = -8;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(x) << endl;
  cout << "abs value of x = " << abs(x) << endl;

  cout << "y = " << y << endl;
  cout << "abs(y) + y^2 = ";
  cout << square(y) + abs(y) << endl;
}
```

To call a function, specify its name and **arguments** in parentheses.

Function calls can be used in compound expressions.

Warning!

Be CAREFUL about using declarations. Only declare a variable to *create* it – not whenever you *use* it.

If you see a TYPE – that means it's a declaration.

- Calling functions in C++ is similar to MATLAB.

```
int main() {
  int x = 3;
  int y = -8;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(int n) << endl;
  cout << "abs value of x = " << abs(x) << endl;

  cout << "y = " << y << endl;
  cout << "abs(y) + y^2 = ";
  cout << square(y) + abs(y) << endl;
}
```

Error! Don't put the type here!

All right… so how does this all work together?

Lobster Demo

lobster.eecs.umich.edu – ENGR101_17_1

The Details

- When a function is called:

1. The values of the argument expressions are copied into the parameter variables.
2. The code for the function's implementation is run.
3. **As soon as a return statement is encountered, the function ends immediately.**
4. The returned value is transferred back to the calling code, where it is used wherever the function call had appeared.

Different from MATLAB

Type Checking return Statements

abs will always return an int.

```
int abs(int n) {
  if(n >= 0) {
    return n;
  }

  return "hello";
}
```

Error!
Can't convert from string to int.

```
int main() {

  string s = abs(-3);

  double y = abs(-3);

}
```

Error!
Can't convert from int to string.

Implicit conversion from int to double.

Improving the abs Function

- The intermediate variable a isn't necessary.

```
int abs(int n) {
  int a;
  if(n >= 0) {
    a = n;
  }
  else {
    a = -n;
  }
  return a;
}
```

```
int abs(int n) {
  if(n >= 0) {
    return n;
  }
  else {
    return -n;
  }
}
```

Multiple return statements are allowed.

Note: this is NOT the same as a "compound return" in MATLAB

Yet Another Version of the abs Function

- These two implementations work similarly…why?

```
int abs(int n) {
  if(n >= 0) {
    return n;
  }
  else {
    return -n;
  }
}
```

```
int abs(int n) {
  if(n >= 0) {
    return n;
  }

  return -n;
}
```

- In the version on the right, if we go into the `if` branch, we leave the function immediately. The 2nd `return` is only reached otherwise, which is the same effect as an `else` branch.

C++ and MATLAB Differences

- C++ does not allow returning more than one value.
  - MATLAB's compound return syntax allowed this.
- In C++, it is common to define several functions in the same file.
  - In MATLAB, functions are often written in their own file.
- In C++, a return statement determines the return value.
  - In MATLAB, whatever values were stored in the specified return variables at the end of the function were used as the return values.

Arguments and Parameters

- Functions may have many parameters, each with their own type.

```
// prints a message and returns the sum of x and y
int func(int x, string message, int y) {
  cout << message;
  return x + y;
}

int main() {
  int a = 3;
  int b = 4;
  int c = c(a, "hello", b);
}
```

This first line, including the name, parameters, and return type, is called the function **signature**.

The **ordering** of arguments you pass in is used to determine what goes to which parameter.

Motivation for Functions

- Functions make code easier to write/understand
- Functions are great for separating out a set of steps that get repeated (hi, iteration!)
- Even if it's just one or two lines of code in the function, the abstraction can make your overall program much clearer.
  - Remember: your code is a technical communication document in addition to an engineering tool!
- Let's look at an example…

Recall: Finding Prime Numbers

- An algorithm for finding the first N primes:

```
int main() {
  int N = 5;
  int x = 2;
  // Outer loop: iterate through candidate x values
  while(N > 0) {
    bool anyDivisible = false;
    // Inner loop: check y values to make sure none divide x
    for (int y = 2; y < x; ++y) {
      if( x % y == 0 ) { // Check divisibility
        anyDivisible = true;
      }
    }
    if( !anyDivisible ) { // were any divisible?
      cout << x << " ";
      --N;
    }
    ++x;
  }
  cout << "done!" << endl;
}
```

This code is quite complex. It takes a while to figure out what it does.

Recall: Finding Prime Numbers

- An algorithm for finding the first N primes:
  - Loop through numbers x, starting at 0, until we find N that are prime.
    - To determine if a number x is prime, ~~loop through all numbers y from 2 through x - 1 and check that x is not divisible by any of them.~~

Create an `isPrime` function as an abstraction for this part.

Finding Prime Numbers

- Functions allow us to manage complexity in our code.

```
int main() {
  int N = 5;
  int x = 2;
  // Iterate through candidate x values
  while(N > 0) {
    if( isPrime(x) ) { // Check primeness
      cout << x << " ";
      --N;
    }
    ++x;
  }
  cout << "done!" << endl;
}
```

This is much easier! We don't have to worry about the details of checking primeness.

Making an isPrime Function

- First, we decide on an interface for isPrime:

```
// Returns whether the given number is prime
bool isPrime(int n) {
}
```

Returns a bool

Takes an int

Comments are always helpful

- Given this interface, here's how we could use isPrime:

```
bool xIsPrime = isPrime(x);
if( xIsPrime ) {
  // do prime stuff
}
```

```
if( isPrime(x) ) {
  // do prime stuff
}
```

You can plug the function into the if directly.

Making an `isPrime` Function

- First, we decide on an interface for `isPrime`:

```
// Returns whether the given number is prime
bool isPrime(int n);
```

Returns a bool

Takes an `int`

Comments are always helpful

- Next, we write the implementation:

```
bool isPrime(int n) {
  for (int x = 2; x < n; ++x) {
    if( n % x == 0 ) { // Check divisibility
      return false;
    }
  }
  return true;
}
```

If any are divisible, we can immediately return false.

If we make it this far, none were divisible. Return `true`.

All together now…

Lobster Demo

lobster.eecs.umich.edu – ENGR101_17_2

void Functions

- Some functions don't return anything – they just do stuff.

void is a keyword that indicates no return value.

```
void print_row_of_X(int num) {
  for (int x = 0; x < num; ++x) {
    cout << "X";
  }
  cout << endl;
}
```

- Generally void functions will have some "side effect"…
  - e.g. Printing something
  - e.g. Changing pass-by-reference parameters (we'll talk about this later today)

5 min

Exercise: Printing Triangles

- Write a function to print out a triangle of Xs.
  - Use the `print_row_of_X` function in your code!

```
void print_row_of_X(int num);
```

```
void print_triangle_X3() {


  // YOUR CODE HERE



}
```

Example

X
XX
XXX
XX
X

Solution: Printing Triangles

- Write a function to print out a triangle of Xs.
  - Use the `print_row_of_X` function in your code!

```
void print_row_of_X(int num);
```

```
void print_triangle_X3() {
  for (int r = 1; r <= 3; ++r) {
    print_row_of_X(r);
  }

  for (int r = 2; r > 0; --r) {
    print_row_of_X(r);
  }
}
```

Example

X
XX
XXX
XX
X

Use Parameters to Make Functions Flexible

1

```
void print_triangle_X3() {
  for (int r = 1; r <= 3; ++r) {
    print_row_of_X(r);
  }
  for (int r = 2; r > 0; --r) {
    print_row_of_X(r);
  }
}
```

Can only print an X triangle of "size" 3.

2

```
void print_triangle_X(int size) {
  for (int r = 1; r <= size; ++r) {
    print_row_of_X(r);
  }
  for (int r = size - 1; r > 0; --r) {
    print_row_of_X(r);
  }
}
```

Can print an X triangle of **any "size"**.

Use Parameters to Make Functions Flexible

```
void print_row(int num, char c) {
  for (int x = 0; x < num; ++x) {
    cout << c;
  }
  cout << endl;
}
```

Can print a row of any "size" with **any character**!

3

```
void print_triangle(int size, char c) {
  for (int r = 1; r <= size; ++r) {
    print_row(r, c);
  }
  for (int r = size - 1; r > 0; --r) {
    print_row(r, c);
  }
}
```

Can print a triangle of **any "size" with any character**!

Example

3

```
void print_triangle(int size, char c) {
  for (int r = 1; r <= size; ++r) {
    print_row(r, c);
  }
  for (int r = size - 1; r > 0; --r) {
    print_row(r, c);
  }
}
```

Can print a triangle of **any "size" with any character**!

```
int main() {
  cout << "hello" << endl;
  print_triangle(2, 'X');

  cout << "goodbye" << endl;
  print_triangle(4, '>');
}
```

This has much less code duplication and is much easier to read than it would be without the functions.

Break Time

We'll start again in 5 minutes.

Recall: Scope

- A variable can only be used…
  - …after it's declaration
  - …within its **scope**.

- If you try to use a variable before its declaration or outside its scope, you'll get a compiler error!

Global Scope

- Variables declared outside of a function have **global scope.**
- They can be used anywhere in the program.
  - (After their declaration, of course.)

- In most cases, global variables are evil[1].
  - Because they can be used from anywhere…
    - It's hard to keep track of which parts of code use them.
    - They allow seemingly separate parts of code to interfere with each other.

1 That is, they are a poor design choice.

Global Constants

- One non-evil use of global variables is for **constants**.
- These are variables whose value will never change.
  - In C++, use the `const` keyword to enforce this.

```
const double PI = 3.14159;

double circleArea(double rad) {
  return PI * rad * rad;
}

double circleCircumference(double rad) {
  return 2 * PI * rad;
}
```

PI can be used in both functions because it has global scope.

Function Block Scope

- The body of a function constitutes a block.
  - All local variables in the function have this block scope.
  - All parameters also have this block scope!

```
int func(int n) {
  int x;
  ...
  ...
  ...
}
```

x and n both have the same scope.

Scope and Naming

- Different scopes can have variables with the same name.
  - The compiler considers these to be completely separate!

```
const double PI = 3.14159;

double circleArea(double rad) {
  return PI * rad * rad;
}

double circleCircumference(double rad) {
  return 2 * PI * rad;
}

int main() {
  double rad;
  cout << "Enter the radius: ";
  cin >> rad;
  cout << "Area: " << circleArea(rad) << endl;
  cout << "Circumference: " << circleCircumference(rad) << endl;
}
```

Think of each scope as having its very own set of variables.

There are three different variables named rad in this program.

Shadowing

Be CAREFUL about using declarations. Only declare a variable to *create* it – not whenever you *use* it.

If you see a TYPE – that means it's a declaration.

- Working with nested scopes can be tricky. Consider this:

```
int main() {
  string message = "unknown"; // initial message
  cout << "Enter a number: ";

  int x;
  cin >> x;

  if(x % 2 == 0) { // if x is even
    string message = "It's even!"; // set message to even
  }
  else {
    string message = "It's odd!"; // set message to odd
  }

  cout << message << endl;
}
```

This code always prints out "unknown", no matter what the user enters. What's wrong?

We accidentally declared a new `message` variable in the `if` branches, which "shadows" the original one!

Declaring Functions

- Just like variables, functions must be declared before use.
- This leads to compile errors if functions are defined in the wrong order:

```
int main() {
  int x = 3;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(x) << endl;
}

// Returns the square of the given number
int square(int n) {
  return n * n;
}
```

Error! square is not declared.

Declaring Functions: Order Matters!

- One solution: **declare and define** the functions before they're used (like we've been doing so far today)

```
// Returns the square of the given number
int square(int n) {
  return n * n;
}

int main() {
  int x = 3;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(x) << endl;
}
```

- But what if we don't want to do this? (e.g. for the sake of organization)

Function Prototypes

- A **function prototype** *declares* a function before it is actually *defined*.
  - It is written as the function signature followed by a `;`.

```
int square(int n);

int main() {
  int x = 3;

  cout << "x = " << x << endl;
  cout << "x squared = " << square(x) << endl;
}

// Returns the square of the given number
int square(int n) {
  return n * n;
}
```

"Hey compiler, just FYI there will be a function called `square` that works like this, so don't worry if you see it used somewhere."

Don't forget to actually define the function later on in your file, outside of `main()`

Swapping Variable Values

- A common task in programming is to swap the values of two variables with each other.
  - Why doesn't this work?

```
int main() {
  int x = 2;
  int y = 7;

  // Swap the values of x and y
  x = y;
  y = x;

  cout << "x is now: " << x << endl;
  cout << "y is now: " << y << endl;
}
```

Output:

```
x is now 7
y is now 7
```

Swapping Variable Values

- Use an auxiliary variable to prevent one variable from overwriting the other during the swap.

```
int main() {
  int x = 2;
  int y = 7;

  // Swap the values of x and y
  int oldX = x;
  x = y;
  y = oldX;

  cout << "x is now: " << x << endl;
  cout << "y is now: " << y << endl;
}
```

Output:

```
x is now 7
y is now 2
```

Writing a swap Function

- Let's write a function to swap variables…

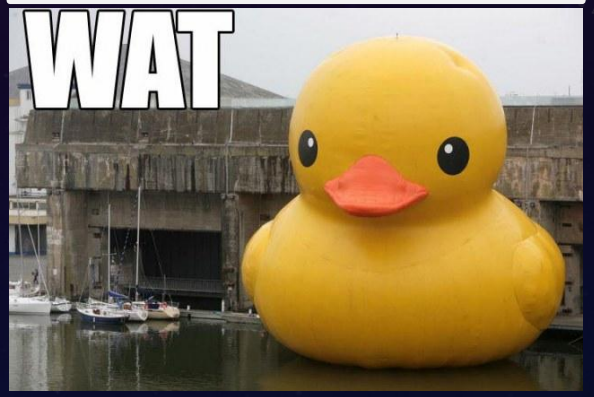
```
// Swap the values of a and b
void swap(int a, int b) {
  int oldA = a;
  a = b;
  b = oldA;
}

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);

  cout << "x is now: " << x << endl;
  cout << "y is now: " << y << endl;
}
```

It doesn't do anything.



Output:

```
x is now 2
y is now 7
```

Writing a swap Function

- Does it fix things to change the names to match x and y?

```
// Swap the values of x and Y
void swap(int x, int y) {
  int oldX = x;
  x = y;
  y = oldX;
}

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);

  cout << "x is now: " << x << endl;
  cout << "y is now: " << y << endl;
}
```

No. The compiler considers x and y in the swap function to be completely different variables, regardless of their name.

Output:

```
x is now 2
y is now 7
```

Parameter Passing

- There are two mechanisms for parameter passing in C++:

Pass-by-value. This is the default.

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

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are unchanged
}
```

a and b are given **copies** of the values of x and y

Pass-by-reference. Specify with &.

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

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are swapped
}
```

No copies!
a refers to x
b refers to y

Changes in the function are visible outside!

Draw a Picture

Pass-by-value. This is the default.

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

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are unchanged
}
```

a and b are given **copies** of the values of x and y

Pass-by-reference. Specify with &.

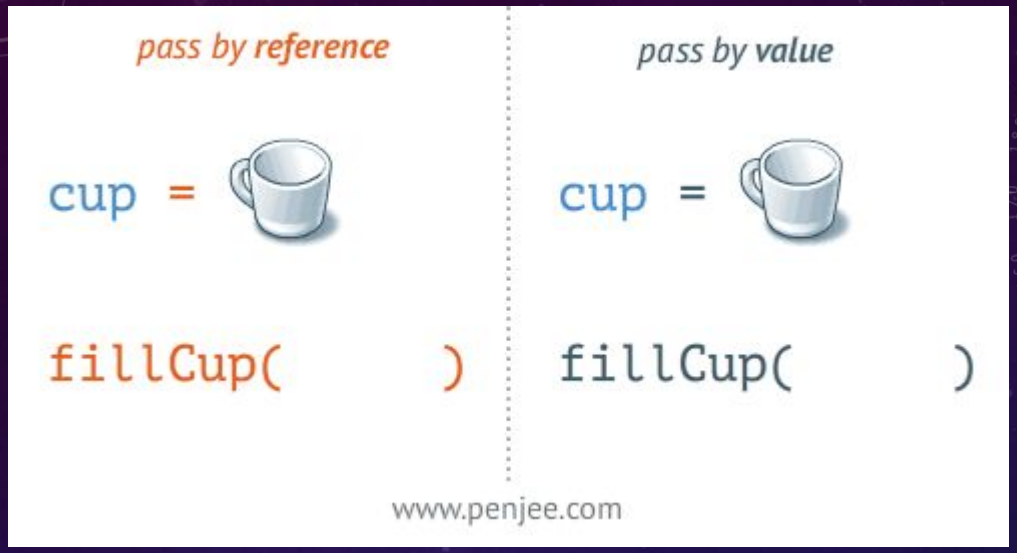
```
void swap(int &a, int &b) {
  int oldA = a;
  a = b;
  b = oldA;
}

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are swapped
}
```

No copies!
a refers to x
b refers to y

Changes in the function are visible outside!



Pass-by-reference. Specify with &.

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

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are swapped
}
```

No copies!
a refers to x
b refers to y

Changes in the function are visible outside!

Pass-by-value. This is the default.

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

int main() {
  int x = 2;
  int y = 7;

  swap(x, y);
  // x and y are unchanged
}
```

a and b are given **copies** of the values of x and y