function: a group of statements given a name that can be called from some point in the program. Functions allow us to write programs in pieces of code that perform individual tasks

```python
defining functions

`type  name(arg1, arg2, ...)`
```
```
{
  `body`
  return
}
```

- **type:** type of value (e.g. int, double, string, etc.) returned by the function
- **name:** identifier for calling the function
- **arguments:** `arg1, arg2, ... (as many as you want)`: each argument is a variable for the function to use. Arguments allow us to pass information to the function from where it is called
- **body:** a group of statements surrounded by `{ }` that tell what the function does
- **return:** a special statement that passes information back to where the function was called. The return value must have the same type as `type` (e.g. int, double, string, etc.). Often considered part of the body.
an example function

```c
int add(int x, int y) {
    int z = x + y;
    return z;
}
```

type: int
name: add
arguments: int x, int y (notice arguments are specified like variable declarations: type name)
body: int z = x + y
return: return z (specification: return name)
task: adds two numbers x and y and returns the result
another example function

```c
int main() {
    return type  function name  no arguments
    statements
    return 0;
}
```

**type:** int

**name:** main (main is a special c++ function. It’s the only one that is automatically called when running a program. All other functions must be called from the main function

**arguments:** none (for now)

**body:** bunch of statements

**return:** return 0 (if the main function is missing a return statement, the compiler assumes a “return 0;”)

**task:** whatever your heart desires (so long as you can program it)
using or calling functions: function call

execution spaces: place where a program runs. spaces don’t share data

#include <iostream>

using namespace std;

int add(int x, int y) {
    int z = x + y;
    return z;
}

int main() {
    int a = 1, b = 2;
    int c = add(a, b);
    cout << c << endl;
}
using or calling functions: function call

#include <iostream>

using namespace std;

int add(int x, int y) {
    int z = x + y;
    return z;
}

int main() {
    int a = 1, b = 2;
    int c = add(a, b);
    cout << c << endl;
}

1: before function call:
   A. program starts
   B. a = 1, b = 2

2: add function is called:
   A. space 0 (main) is paused
   B. space 1 is created to run add
   C. the values of a and b are copied into this new space, and renamed to the local variable names x and y. Thus, x = a, which is 1, and y = b, which is 2
   D. space 1 (add function) runs
using or calling functions: return

```cpp
#include <iostream>

using namespace std;

int add(int x, int y) {
    int z = x + y;
    return z;
}

int main() {
    int a = 1, b = 2;
    int c = 3;
    cout << c << endl;
}
```

1: before function call:
   A. program starts
   B. a = 1, b = 2

2: add function is called:
   A. space 0 (main) is paused
   B. space 1 is created to run add
   C. the values of a and b are copied into this new space, and renamed to the local variable names x and y. Thus, x = a, which is 1, and y = b, which is 2
   D. space 1 (add function) runs

4: return: end of the function is reached
   A. program returns to the location in space 0 where add was originally called
   B. replace the call with the return value (i.e. 3).
   space 1 disappears
using or calling functions: return

#include <iostream>

using namespace std;

int add(int x, int y) {
    int z = x + y;
    return z;
}

int main() {
    int a = 1, b = 2;
    int c = 3;
    cout << c << endl;
}

1: before function call:
   A. program starts
   B. a = 1, b = 2

2: add function is called:
   A. space 0 (main) is paused
   B. space 1 is created to run add
   C. the values of a and b are copied into this new space, and renamed to the local variable names x and y. Thus, x = a, which is 1, and y = b, which is 2
   D. space 1 (add function) runs

4: return: end of the function is reached
   A. program returns to the location in space 0 where add was originally called
   B. replace the call with the return value (i.e. 3).

space 1 disappears

5: after function call:
   A. space 0 (main) resumes with 3 replacing the function call to add.
   B. Then prints “3”
more examples of calling add

more ways to call add

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

int add(int x, int y); // declared first

int main() {
    // literals
    int a = add(1, 2);
    cout << a << endl;

    // user input
    int a, b, c;
    cin >> a >> b;
    int c = add(a, b);
    cout << c << endl;

    // w/o saving result
    cout << add(2, 4);  // 6
}

int add(int x, int y) { // define later
    return x + y
}
```

multiple calls

```cpp
int main() {
    int a = add(1, 2) + add(3, 4);
    cout << a << endl;  // 10

    int c = add(10, 5);
    int d = add(c, 3);
    int e = add(6, d);
    cout << e << endl;  // 24
}
```

nested calls

```cpp
int main() {
    int a = add(1, add(3, 4));
    cout << a << endl;  // 3

    int b = add(add(1,2), add(3,4));
    cout << b << endl;  // 10

    int c = add(10,5);
    cout << add(add(c, c), 1);  // 31
}
```
## function calling problems with add, sub

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sub(1, 2) + sub(2, 1)</code></td>
<td>0</td>
</tr>
<tr>
<td><code>add(sub(2, 1), sub(1, 2))</code></td>
<td>0</td>
</tr>
<tr>
<td><code>sub(10, add(2, 3))</code></td>
<td>5</td>
</tr>
<tr>
<td><code>add(1, sub(3, add(2, 3)))</code></td>
<td>-1</td>
</tr>
<tr>
<td><code>add(a, a)</code></td>
<td>2a</td>
</tr>
</tbody>
</table>

### what's the output?

### how to get this output?

Let $x = 2$, $y = 5$, $z = -3$

- `add(y, y)`
- `add(2, 3)`
- `sub(y, z)`
- `add(x, x)`
- `sub(x, z)`

- `sub(sub(add(y, z), x), x)`
- `sub(x, z)`
- `sub(add(x, x), y)`
- `sub(sub(y, z), add(add(x, x), add(x, x)))`
more function examples

**calculated hypotenuse length**

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

double square(double x) {
    return x * x;
}

int main() {
    double a, x, y, d;
    cin >> x >> y;
    a = square(x) + square(y);
    d = sqrt(a);
    cout << "Distance: " << d;
}
```

Notice this function has return type **void**, so it does not return anything, and does not require a return statement at the end.

**draws pyramids**

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

void draw_line(int size); 

int main() {
    int i, j, n;
    cin >> n;
    for (i = 0; i < 2 * n; i++) {
        if (i <= n) { // up
            draw_line(i);
        } else { // down
            draw_line(2 * n - i);
        }
    }
}

void draw_line(int size) { 
    for (int i = 0; i < size; i++) {
        cout << "0";
    }
    cout << endl;
}
```

*new library for std::sqrt*

*defined later*

*declared first*
nested functions

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

int square(int x) { return x * x; }

int sqrt(int x2) {
    int x = 0;
    while (square(x) < x2) {
        x++;
    }
    return x;
}

int hyp_length(int x, int y) {
    int z = square(x) + square(y);
    return sqrt(z);
}

int main() {
    double x, y;
    cin >> x >> y;
    cout << hyp_length(x, y) << endl;
}
```
pass-by-value vs. pass-by-reference

when the execution space is created:

**value:** argument values are copied into the function space. The original arguments are unchanged.

**reference:** argument variables are passed directly into the function space, and can change upon exit.

Why is this useful?

1. **efficiency:** copying argument values can be expensive if the arguments take up a large space in memory (e.g. really long strings like whole novels, or datatypes that take lots of memory)

2. **returning multiple arguments:** functions can have as many arguments as you want, but only one return value. By being able to change the input arguments, you can effectively return more data

### pass-by-value

```c++
int square(int x) {
    return x * x;
}
```
```
int x = 10;
int x = square(x);
cout << x << endl;
```

### pass-by-reference

```c++
void square(int& x) {
    x = x * x;
}
```
```
int x = 10;
square(x);
cout << x << endl;
```
pass-by-reference examples

### swap

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

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

int main() {
    int x = 10, y = 5;
    cout << x << " " << y;
    swap(x, y);
    cout << x << " " << y;
}
```

### string capitalizer

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

void capitalizer(string& s) {
    for (int i = 0; i < s.size(); i++) {
        s[i] = toupper(s[i]);
    }
}

int main() {
    string hi = "Hello!";
    capitalizer(hi);
    cout << hi << endl;
}
```

### sort (small to large)

```cpp
void sort(int& a, int& b) {
    if (a > b) {
        swap(a, b);
    }
}
```

### sort3 (small to large)

```cpp
void sort3(int& a, int& b, int& c) {
    if (a > b) { swap(a, b); }
    if (b > c) { swap(b, c); }
    if (a > b) { swap(a, b); }
}
```
project 3 tips

Incremental development: break down the program from larger steps into a bunch of smaller simpler steps, perhaps structured into various functions that perform specific tasks.

Comment well, so that both you and your TA can follow along what your program is doing.

Read all of the links on the website about processing strings and functions.