

User-defined functions

- A predefined function is just a function someone else wrote and compiled into a library
- A program can have multiple functions
 - `main` is required
 - Other functions can be defined the same way, then used just like predefined functions

Parts of a function definition

```
int main()                                int num;
{
    // your program here
}
```

Function heading	<code>int main()</code>
Name of the function	<code>main</code>
List of parameters, with types	<code>()</code>
Return type of the function	<code>int</code>
Function body	<code>{ // your program here }</code>

Writing cube(x)

```
double cube( double x )  
{  
    // your program here  
}
```

Function heading	<code>double cube(double x)</code>
Name of the function	<code>cube</code>
List of parameters, with types	<code>(double x)</code>
Return type of the function	<code>double</code>
Function body	<pre>{ // your program here }</pre>

Writing cube(x)

```
double cube( double x )  
{  
    double c;  
    c = x * x * x;  
  
    return c;  
}
```

return statement

```
return 0;
```

- When a return statement executes
 - Function immediately terminates
 - The specified value is returned
- When a `return` statement executes in the function `main`, the program terminates

Alternative cube (x)

```
double cube( double x )  
{  
    double c = x * x * x;  
    return c;  
}
```

```
double cube( double x )  
{  
    return x * x * x;  
}
```

Call and definition

- There are two distinct viewpoints on every function
 - The function call (outside)
 - Call by name
 - Provide (*pass in*) input parameters or *arguments*
 - Get back the return value and do something with it
 - The function definition (inside)
 - Receive the parameters
 - Do something with them (and also local variables)
 - Return (*pass out*) a value

Parameters

- *Formal parameters*
 - Used inside the function
 - Declared like variables (type and name) in the function heading
 - E.g. `x` in `double cube(double x)`
- *Actual parameters*
 - Passed from outside in the function call
 - Must match the number and types of the formal parameters
 - E.g. `5` in `cube(5);`
- Each actual parameter provides a value for a formal parameter
 - `x` gets the value `5`

A sum function

- Write a function definition to take the sum of three real numbers
 - Name: `sum_three`
 - Formal parameters: 3 real numbers (`x`, `y`, `z`)
 - Return value: 1 real number (the sum)
- To add 5, 6 and 7 and store in a variable `sum`:
`sum = sum_three(5, 6, 7);`

Formal Parameter in Definition	Actual Parameter in Call
<code>x</code>	5
<code>y</code>	6
<code>z</code>	7

A sum function

- The function definition (header + body):

```
double sum_three( double x, double y, double z )
{
    double sum;
    sum = x + y + z;
    return sum;
    // return x+y+z;
}
```

- The function call (to add 5, 6 and 7 and store in a variable `sum`) :

```
sum = sum_three( 5, 6, 7 );
```

Exercise: An average function

- Write a function definition to take the average of three numbers
 - Name: `average_three`
 - Parameters: `3 real numbers`
 - Return value: `1 real number (the average)`
1. Write the heading
 - Name, parameter list, return type
 2. Write the body
 - Declare any local variables necessary
 - Do something with the parameters
 - Return a value

Functions, variables and memory

- Each function has its own memory space
 - Including `main`
 - All variables and parameters declared in a function refer to memory *allocated* in that space
 - When a function ends, its variables are deallocated

```
double sum_three( double x, double y, double z )
{
    double sum;
    sum = x + y + z;
    return sum;
}

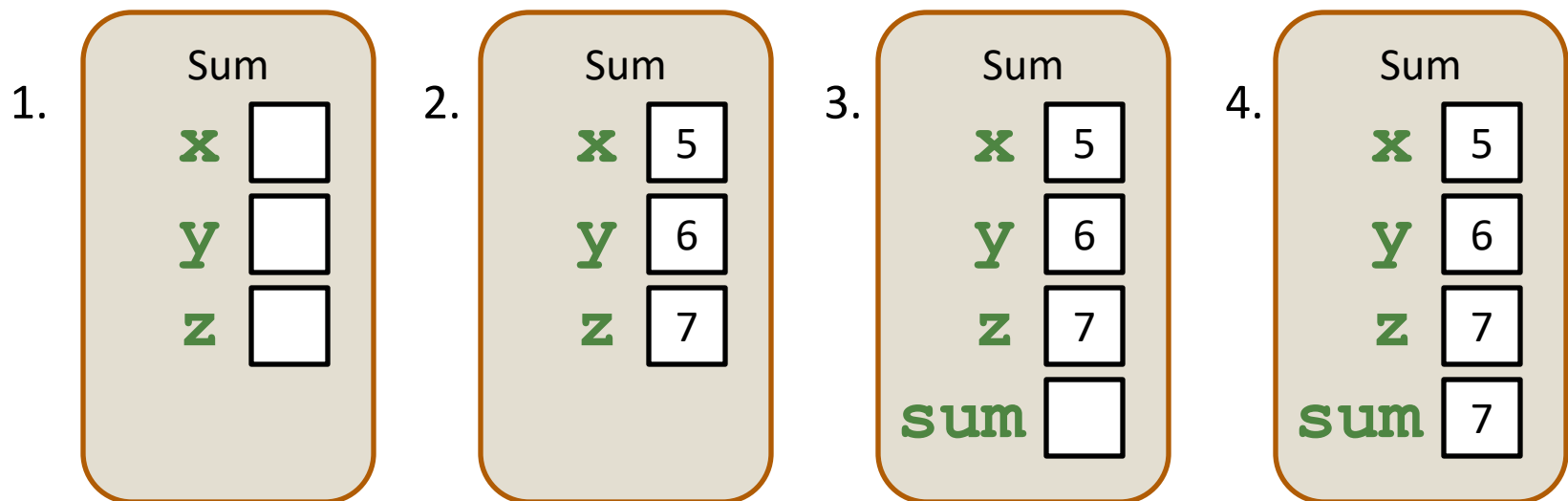
...
sum = sum_three( 5, 6, 7 );
```

Functions, variables and memory

```
Double sum_three(double x, double z, double y);
```

```
sum = sum_three( 5, 6, 7 );
```

1. Allocate memory for formal parameters
2. Assign actual parameter values
3. Allocate memory for declared variable sum
4. Calculate the sum
5. Return the sum (all memory de-allocated)



Functions, variables and memory

- Local variables and parameters inside a function are specific to that function!
 - They don't exist outside, which is why values must be passed in and returned
 - Functions cannot use variables declared in another function (even main)
 - We say that they are *out of scope*
- Variables with the same name in different functions do not refer to the same memory

The `void` return type

- A function does not have to return a value
 - The special type `void` indicates that a function does not return anything
 - A `void` function cannot be called as if it returned a value
- Given a function with the heading:

```
void thisFunction( int x )
```

 - This function call would cause an error:

```
y = thisFunction( x );
```
 - Putting a return statement in the function body would also cause an error

Using Functions

- Functions are like building blocks
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
 - Can be re-used (even in different programs)
 - A programmer can focus on just that part of the program and construct it, debug it, and perfect it
 - Different people can work on different functions simultaneously
 - Enhance program readability

Function Prototypes

- Like variables, functions must be declared before they are used
 - That's why we've been putting them at the top of the file, before `main`
 - This simultaneously *declares* and *defines* the function
 - In practice, we like to put `main` first, so we separate the function *declaration* from the function *definition*

Function Prototypes

- Functions are *declared* with a prototype
 - Looks just like the function heading as a statement (with ;)

```
double pow( double x, double y );
```

- As long as the *declaration* is before the function is used, you can put the *definition* anywhere
 - The definition is unchanged, still has heading and body
 - Convention is to put all function declarations together, followed by all function definitions (with `main` first)

```
// declare a function
```

```
int fun1();
```

```
int fun1(){  
    cout<<"hi";  
}
```

```
// allocate memory for a variable
```

```
int x;
```

```
x = 10;
```

Prototypes and Organization

```
// declare a function
```

```
double get_side();
```

```
// define a function
```

```
double get_side()
```

```
{  
    double input;  
    cout << "Please enter a side of the triangle: ";  
    cin >> input;  
    return input;  
}
```

```
int main()
```

```
{  
    double x;  
    // call  
    x = get_side();  
    return 0;  
}
```

Exercise 1

- Write the prototype for a function that takes in a string and a number n and returns the n th word in the string
 - Name: `nth_word`
 - Parameters: `1 string, 1 number`
 - Return value: `1 string`

Exercise 2

- Complete the code below:

```
string sentence = "They switched from the Swingline to the Boston  
    stapler, but I kept my Swingline stapler because it didn't bind  
    up as much, and I kept the staples for the Swingline stapler  
    and it's not okay because if they take my stapler then I'll set  
    the building on fire...";  
string word;  
  
// use the nth_word function to get the 14th word out of  
// sentence and store it in word
```

Using Multiple Files

- It can be convenient to organize code into more than one file
 - Particularly for reusing code (libraries)
- Standard libraries are included with:
`#include <name> (e.g. iostream, string, cmath, etc)`
- Files in a project are included with:
`#include "name" (e.g. main.h, functions.cpp, etc)`
- Including a file merely inserts that file in place of the include directive
- Useful for putting functions in their own file