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()
{
    // your program here
    return something_int_type;
}
```

Function heading	`int main()`
Name of the function	`main`
List of parameters, with types	`()`
Return type of the function	`int`

Function body	`{`
	`    // your program here`
	`}`

Define a user-defined function cube(x)

```
double cube( double x, int y, char z, string string_1)
{
    // your program here
    return something_in_double_type:
}
```

Function heading	`double cube( double x )`
Name of the function	`cube`
List of parameters, with types	`( double x )`
Return type of the function	`double`

Function body	`{`
	`  // your program here`
	`}`

Define a function called `cube(x)`

```
double cube( double x )
{
   double results;
   results = x * x * x;
   return results;
   cout << "this sentence is after the return" <<
   endl;
}

// x is the input parameter of the function
// input parameter of the function is different from
   the input using "cin" or reading in from file.
```

`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
x	5
y	6
z	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;
}
```

- 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

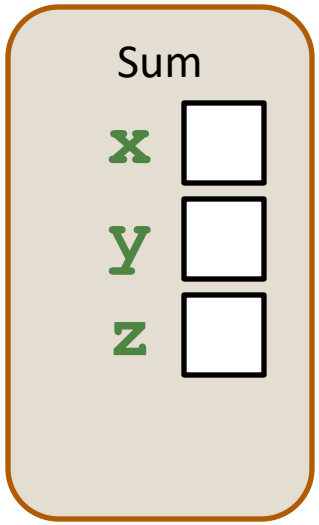
```
double sum_three( double x, double y, double z )
{
        double sum;
        sum = x + y + z;
        return sum;
}
…
sum = sum_three( 5, 6, 7 );
```

Functions, variables and memory

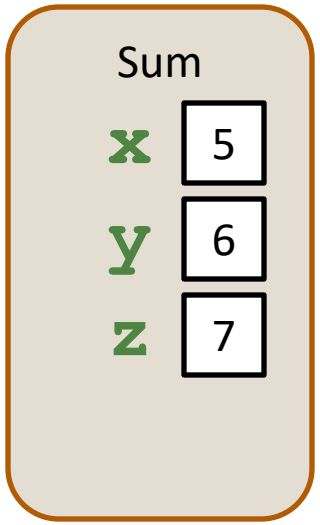
```
sum = sum_three( 5, 6, 7 );
```

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)

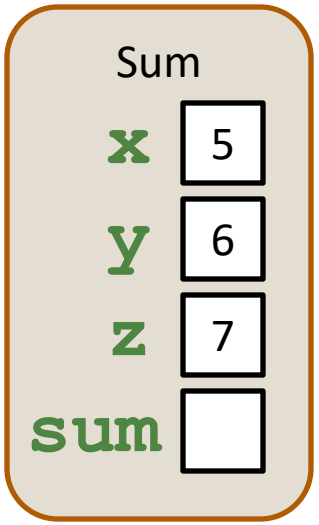
1.



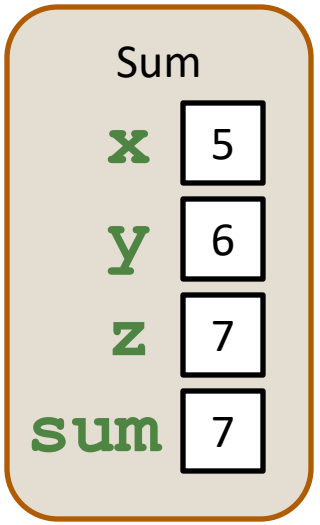
2.



3.



4.



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