Functions

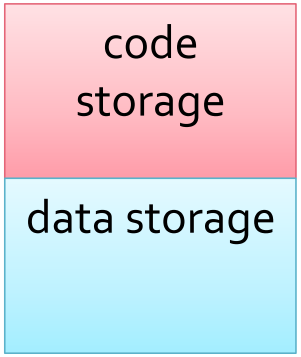
Functions

- Basic memory model
- Using functions
- Writing functions
  - Basics
  - Prototypes
  - Parameters
  - Return types
- Functions and memory
- Names and namespaces

Memory Model

Memory Management

- When a program runs it requires main memory (RAM) space for
  - Program instructions (the program)
  - Data required for the program
- There needs to be a system for efficiently allocating memory
  - We will only consider how memory is allocated to program data (variables)



Computer Memory

- We will often refer to main memory
  - By which we mean RAM or *random-access memory*
  - RAM is both readable and writable
- RAM can be thought of as a (very long) sequence of bits
  - In this simplified model we will number this sequence in bytes

RAM

- RAM is a long sequence of bytes
  - Starting with 0
  - Ending with the amount of main memory (-1)
- RAM is addressable and supports *random access*
  - That is, we can go to byte 2,335,712 without having to visit all the preceding bytes

RAM Illustrated

Consider a computer with 1GB* of RAM

*1 GB = 1,073,741,824 bytes

RAM can be considered as a sequence of bytes, addressed by their position

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	…
…	1073741816	1073741817	1073741818	1073741819	1073741820	1073741821	1073741822	1073741823

Note – this is a simplified and abstract model

Variable Declaration

- Declaring a variable reserves space for the variable in main memory
  - The amount of space is determined by the type
- The name and location of variables are recorded in the *symbol table*
  - The symbol table is also stored in RAM
  - The symbol table allows the program to find the address of variables
    - We will pretty much ignore it from now on!

Variable Declaration Example

For simplicity's sake assume that each address is in bytes and that memory allocated to the program starts at byte 2048

```
int x, y;
x = 223;
x = 17;
y = 3299;
```

`int x, y;` → Creates entries in the symbol table for *x* and *y*

variable	address
x	2048
y	2052

`x = 223;`, `x = 17;`, `y = 3299;` → These lines change the *values* stored in *x* and *y*, but do not change the location or amount of main memory that has been allocated

data	17				3299											
address	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	...

Variables

- There are two things that we might wish to find out about a variable
- Most of the time we just want to find what value is stored in a variable
  - By the end of the previous example the variable *x* had the value 17 stored in it
- Sometimes we* might want to know *where* a variable is located – its address
  - The address of *x* was 2,048
    - *OK, *we* probably don't want to know this, but a program might need this information

Simple Memory Model

- Variables are stored in main memory in sequence
  - We can find out the value of a variable
  - And its address
    - To retrieve the address write the variable name preceded by an ampersand (&)
- The *value* of a variable can be changed by assignment
  - But its storage location and the amount of memory allocated to a variable cannot change

Printing Variables

- We can use *cout* to print the value of a variable, or its address
  - `int x = 12;`
  - `cout << "x = " << x << endl;;`
  - `cout << "x's address = " << &x << endl;`
- Here is another example
  - `float y = 2.13;`
  - `cout << "y = " << x << endl;`
  - `cout << "y's address = " << &y << endl;`

> I wouldn't usually use *x* or *y* as the name of a variable since it doesn't convey any meaning, but in this example they are just arbitrary values

> This just shows that we *can* access the address of a variable, printing the address like this is not generally useful

Using Functions

Using C Functions

- Library functions are often used in C++
  - For example, the *cmath* library contains mathematical functions
    - Such as *sqrt* and *pow*
- When a function is *called* in a program the function code is executed
  - And the return value of the function (if any) replaces the function call (or *invocation*)

Circle Radius

```
// Prints the area of a circle
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

// Define Pi
const double = PI 3.14159265;

// Main program – next slide!
int main(){
    // ...
}
```

Circle Radius

```
int main(){
   float radius;
   float area;
   // Get keyboard input
   cout << "Please enter the radius: ";
   cin >> radius;
   // Calculate area
   area = PI * pow(radius, 2);
   // Print output
   cout << fixed << setprecision(2);
   cout << "The circle's area is " << area << endl;
   return 0;
}
```

Function call

Sets decimal places to 2

Using Functions

the function executes, and it is replaced by the result that it returns

```
area = PI * pow(radius, 2);
```

function name

arguments to the function

Function Precedence

- Function calls in statements have precedence over most other operations
  - Such as arithmetic or comparisons
  - A function is executed and its result returned before other operations are performed

Function Arguments

- Many functions need input
  - Data passed to a function are referred to as *arguments*
  - The number and type of arguments must match what is expected by the function
- Failing to give appropriate arguments results in a compilation error

Invalid Arguments

```
area = PI * pow("fred", 2);
```

no instance of overloaded function "pow" matches the argument list

the error message will vary by compiler

Function Execution

- When a function is called, program execution switches to the function
  - Each line of code in the function is executed in turn until
    - The last line is reached, or
    - A *return* statement is processed
- Execution then returns to the line of code where the function was called from
  - The function call is replaced by the return value

Writing Functions

Writing Functions

- As well as using library functions, we can write our own functions
  - This is very useful as it increases the modularity of programs
  - Functions can be written and tested in isolation
- It is good programming style to write many small functions

GCD

- Consider the calculation to find the greatest common divisor (gcd) of two integers
  - Which was used in a previous example to simplify fractions
  - This calculation might be useful in other places in a larger program
    - Or in other programs

Function Anatomy

function name

parameter list

return type

function header

```
int gcd(int a, int b)
{
        while(a != b)
        {
                if(a > b){
                        a = a-b;
                }else{
                        b = b-a;
                }
        }
        return a;
}
```

function body

return statement

Function Anatomy

- Functions have two parts
  - A header (or declaration)
    - Return type of the function
    - Function name
    - Function parameter list
  - A body (or definition)
    - The executable statements or implementation of the function
    - The value in the return statement (if any) must be the same type as the function's return type

Function Names

- Function's names are identifiers, so must follow identifier rules
  - Only contain a to z, A to Z, _, 0-9
  - Must start with a to z, A to Z
- By convention, function names should
  - Start with a lower case letter
    - They can be distinguished from variables by their parameter lists (brackets)
  - Have meaningful names

Functions That Return Nothing

- Not all functions need to return a value
  - For example a function to print instructions
- Functions that return nothing should be given a return type of *void*
  - A function that returns nothing (void) cannot be used in an assignment statement
- Void functions are often used for output

Parameter Lists

- Every function must have a parameter list
  - The parameter list consists of ()s which follow the function name
  - Parameter lists contain function input
  - A parameter list can be empty, but the brackets are still required when the function is defined
- Whenever a function is called (used) it also must be followed by brackets
  - Again, even if they are empty

A Function That Returns Nothing

```
#include <iostream>

void hi(){
  std::cout("hello world");
}

int main (){
  hi();
  return 0;
}
```

Noy very useful, but imagine it prints the instructions for a console application.

Using Functions

- A function that is written in a .cpp file can be used in that file
  - Just like a library function
- The function must be declared *before* it is used, either by
  - Defining the function, or
  - Writing a *forward declaration*, and defining it later
    - Also referred to as a *function prototype*

Identifier Not Found

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

const float PI = 3.14159;

int main()
{
   float r = 3;
   cout << "radius = " << r << ", volume = " <<  sphereVolume(r);

   return 0;
}

double sphereVolume(double radius)
{
   return PI * pow(radius, 3) * 4.0/3;
}
```

Error: C3861 'sphereVolume': identifier not found

The compiler processes the file one line at a time starting from the top, so when it reaches *sphereVolume* it does not recognize the name

Function Prototypes

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

const float PI = 3.14159;
// Function Prototypes
double sphereVolume(double radius);

int main()
{
   float r = 3;
   cout << "radius = " << r << ", volume = " <<  sphereVolume(r);

   return 0;
}

double sphereVolume(double radius)
{
   return PI * pow(radius, 3) * 4.0/3;
}
```

fix by inserting a *function prototype* before main (or by moving the function definition)

it is usually preferable to use function prototypes rather than defining all your functions above main

argument

parameter

Function Prototypes – Why?

- The compiler compiles a program one line at a time
  - Starting at the top of the file
  - If the compiler encounters an undefined identifier it will be unable to continue
- Functions can be declared before being used and defined later on in the file
  - A *function prototype* consists of the function header (followed by a semi-colon)

Function Input and Parameters

- Many functions require data to perform operations on (i.e. input)
- Such data can be given to functions in a number of ways
  - The function can obtain the data itself
    - By getting input from the user
  - Data can be provided via global variables
    - Which are visible throughout the file
  - Values can be passed to parameters

Global Variables

- A global variable is a variable that is declared outside any function
  - Including the main function
  - Variables declared inside a function are referred to as *local variables*
- Global variables can be used by any function
  - In the same file (and possibly other files)
  - **Avoid global variables**
    - With a few exceptions, such as constants

Why Avoid Global Variables?

- They make programs harder to understand
  - Functions that rely on global variables cannot be considered in isolation
- They make programs harder to modify
  - It is relatively straightforward to change a function that is self-contained
- They make programs less portable
  - If functions depend on global variables they cannot easily be used by other files

Parameter Lists

- Parameter lists are the preferred method of data input to a function
- Parameters are special variables that can be passed values from calling code
  - A function's parameters are given the value of variables passed to a function
    - Variables passed to functions are called arguments

Arguments

formal parameters

```
void printIntDivision(int dividend, int divisor)
{
	int quotient = dividend / divisor;
	int remainder = dividend % divisor;
	cout << dividend << "/"   << divisor << " = " <<
		quotient << " remainder" << remainder);
}
```

```
// ... main function
cin >> x;
// ...
cin >> y;
//...
printIntDivision(x, y); //prints result of x/y
```

arguments

Parameters and Arguments

- Parameters and arguments are different variables
  - They refer to different main memory locations
  - Parameters are given the values of arguments
  - They may even have the same name but still refer to *different* variables
- It is also possible to pass a function the *address* of a variable
  - Using pass by reference
    - Or pointers

Returning Values

- Many functions *return* values
  - The function performs a calculation
  - And sends the result back to calling code
    - e.g. *sqrt*, *pow*
  - Values are returned by a return statement
- The type of the value in a return statement must match the return type of the function

Return Statements

```
// Function to return a letter grade
char letterGrade(int grade){
    if (grade > 89)
        return 'A';
    if (grade > 74)
        return 'B';
    if (grade > 59)
        return 'C';
    if (grade > 49)
        return 'D';
    else //(grade < 50)
        return 'F';
}
```

As soon as any return statement is reached the function execution ends and the value is returned to the calling code

Functions

- Whenever we want to perform an action or a calculation we should write a function
  - Functions organize other programming constructs
  - Functions often need input (arguments) and often return values
- The return type of a function specifies the type of information that it returns
  - Note that a function with a return type of *void* returns nothing

Return Paths

- A function that specifies a return type must contain a return statement
  - Or an error results
- The compiler may not check that *all* paths within a function contain a return value
  - Such as nested if statements, or
  - If, else if, …, else statements
  - Some compilers may give warnings about this, but others may not

Return with No Value

- Is it possible to return from a function that does not return a value (i.e. is void)?
  - Yes!
  - Just use return with no value
- This allows us to return from any function before reaching the end of the function
  - For example, in a branch of an if statement

Function Design

- Functions should perform one task
  - In particular, functions that calculate values should not be responsible for output
    - Such values should be *returned* not printed
- Functions should be self contained
  - Input should be passed via parameters
  - Results should be returned using a return statement

Scope

Variable Scope

- Recall that two variables cannot have the same name
  - More accurately, two variables can have the same name as long as they have different *scope*
- The scope of a variable is the block in which it is declared
  - So variables with different scope may have different names

Examples of Scope

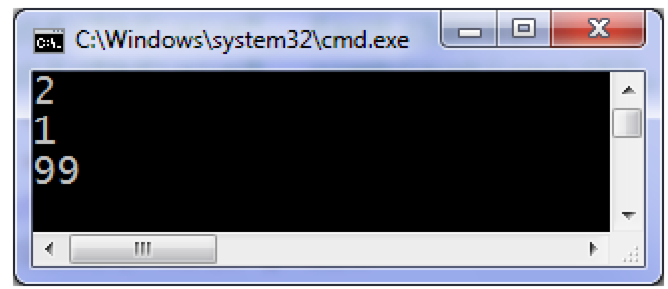
```
void f(){
    int x = 99;
    cout << x << endl;
}

int main()
{
    int x = 1;
    if (x == 1){
        int x = 2;
        cout << x << endl;
    }
    cout << x;
    f();
    cout << endl << endl;
    return 0;
}
```

f x scope

main x scope

if x scope



Scope – Functions

- The scope of any variable declared inside a function is local to that function
  - This includes a function's parameters
    - So a parameter of a function may not have the same name as a variable declared inside that function
    - Unless the variable has a more limited scope
- Functions cannot be defined inside functions
  - The scope of two different functions therefore does not overlap

Most – Least Scope

- The example shown previously had two variables called *x* in the main function
  - One variable declared in if (*if-x*) and one declared in main (*main-x*)
  - The scope of *main-x* encompassed the entire main function and overlapped with *if-x*
- In such cases only the variable with the *least* scope is visible

Memory Model Revisited

Memory Model

- Let's expand our memory model to cover functions
  - Which includes the main function
- It's not going to be very different from our original model
  - Main memory is a long sequence of binary digits
  - Variables are allocated in sequence
  - A system table allows us to find variables
    - But we are not going to go into much detail about it

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
```

In this simple memory model of the *call stack* memory is allocated to variables in sequence – in this example starting with byte 192

The call stack is the area of main memory used for function calls, also referred to as *automatic* memory

next free byte

name	num1				den1																								
data	48				140																								
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	…

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
```

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1				a				b																
data	48				140				48				140																
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	...

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
```

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1				a				b																
data	48				140				4				4																
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	...

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
```

return from gcd

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1																								
data	48				140																								
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	…

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
```

and assign memory to div

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1				div																				
data	48				140				4																				
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	…

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
num1 = num1/div;
den1 = den1/div;
```

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1				div																				
data	**12**				**35**				**4**																				
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	…

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
num1 = num1/div;
den1 = den1/div;
int num2 = 11;
int den2 = 288;
```

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1				den1				div				num2				den2												
data	12				35				4				11				288												
address	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	...

Memory Example

This program fragment simplifies two fractions

```
int num1 = 48;
int den1 = 140;
int div = gcd(num1, den1);
num1 = num1/div;
den1 = den1/div;
int num2 = 11;
int den2 = 288;
div = gcd(num2, den2);
```

```
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    return a;
}
```

next free byte

name	num1	den1	div	num2	den2	a	b	
data	12	35	4	11	288	11	288	
address	192 193 194 195	196 197 198 199	200 201 202 203	204 205 206 207	208 209 210 211	212 213 214 215	216 217 218 219	...

Why The Stack?

- The area of main memory used for function calls is called the *call stack*
- A stack is a simple data structure where items are inserted and removed at the same end
  - Usually referred to as the top of the stack
  - A stack is a LIFO (Last In First Out) structure
- The call stack behaves like a stack
  - Since the last function to be called
  - Is the first function to be removed (de-allocated)

Stack Memory

- The process for allocating main memory on the stack is simple
  - Always allocate memory in sequence
- A few things have to be recorded
  - The address of the next free byte
  - The starting address for each function's memory
  - The line number of each function call in the calling function

Stack Memory Process

- The process for allocating main memory on the stack is simple
  - Always allocate memory in sequence
- A few things have to be recorded
  - The address of the next free byte
  - The starting address for each function's memory
  - The line number of each function call in their calling functions
    - So the program can continue where it left off

Stack Memory Advantages

- Fast
  - The OS does not have to search for free space as variable is inserted at the next byte
  - Variables are found at the byte address given in the symbol table
- Releases memory automatically
  - When a function terminates its memory can be reused by resetting the next free byte
- Space efficient
  - There is very little administrative overhead

Stack Memory Disadvantages

- Variable size cannot change
  - The *value* of variables can change but more space cannot be allocated to an existing variable
    - Since it would overwrite the next variable on the stack
  - It might be useful to change the size of array variables
    - That store multiple values
- Memory is only released when a function ends
  - Variables declared at the start of a function call may not be needed for the lifetime of the function

Pass By Reference

Arguments: Pass By Value

- We have used *pass-by-value* to give information to functions
  - The data passed to a function is copied into the function's parameters
- This requires time to make a copy of the passed value
  - Trivial if we are passing an integer
  - Less trivial if we are passing a 100MB image

A Swap Function

- Let's say we want to write a function to swap the values in two variables
- Here is a first attempt

```
void swap(int x, int y)
{
	int temp = x;
	x = y;
	y = temp;
}
```

Does this work?

Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

```
void swap(int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

23	37	23	37
a	b	x	y

x and *y* are parameters of the swap function so have their own space in main memory

Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

```
void swap(int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

23	37	37	23	23
a	b	x	y	temp

Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

```
void swap(int x, int y)
{
	int temp = x;
	x = y;
	y = temp;
}
```

23	37	37	23	23
a	b	x	y	temp

note that neither *a* nor *b*'s values have changed, so the swap function achieved **nothing**!

swap has completed its execution so its memory is released

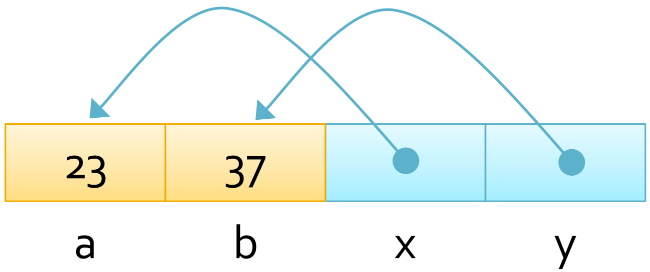
Pass By Reference

- There is an alternative mechanism for passing variables to a function
  - That allows the argument variables to be affected
  - The function's parameters refer directly to the argument variables
- The function parameters are preceded by an ampersand (&)
  - Immediately after the type name

Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(a, b);
```

```
void swap(int& x, int& y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

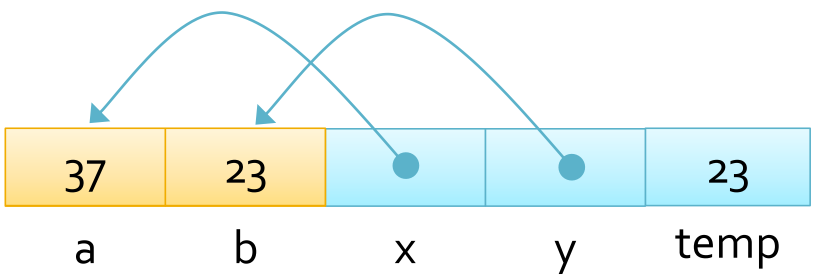


the parameters *x* and *y* refer to *a* and *b*, rather than storing their values

Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(a, b);
```

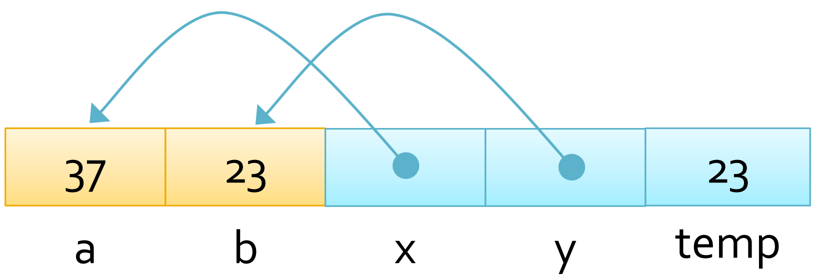
```
void swap(int& x, int& y)
{
        int temp = x;
        x = y;
        y = temp;
}
```



Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(a, b);
```

```
void swap(int& x, int& y)
{
        int temp = x;
        x = y;
        y = temp;
}
```



because *swap*'s parameters are passed by reference *a* and *b* have changed

swap has completed its execution so its memory is released

&

- The & is used to mean a number of different things, dependent on the context
  - Bitwise *and*
    - Which we probably won't talk about …
  - Pass by reference
  - Return by reference
    - Which we also probably won't talk about …
  - The address of operator
    - Which we will talk about later

Pass By Reference Notes

- A function may have both pass by reference and pass by value parameters
  - The pass by value parameters are not preceded by an ampersand
- Pass by reference parameters must be used when arguments are intended to be changed
  - They are also commonly used when a large object is passed to a function

Names

Function Overloading and Namespaces

Overloading Functions

- Like variables, function names must be unique
  - But we can imagine that the full name of a function includes the types of its parameters
    - So the swap function is called *swap-int-int*
- Two functions may be given the same name if they have differently typed parameter lists
  - Or different numbers of parameters
  - This is referred to as function *overloading*

Overloading Functions

- We can look up the *sqrt* function
  - On http://www.cplusplus.com/
- There are three different versions of *sqrt*
  - double sqrt (double x);
  - float sqrt (float x);
  - long double sqrt (long double x)
  - Allowing the square roots of these different types to be calculated without losing precision

Namespaces

- For convenience our programs often start with
  - `#include <iostream>`
  - `using namespace std;`
- The first line includes the *iostream* library so that we can use *cin* and *cout*
  - The second line allows us to use *cin* and *cout* without preceding them with *std::*
  - A namespace contains a list of function and constant names

Colliding Names

- Namespaces are a mechanism to handle multiple functions with the same name
- There are a number of standard libraries
  - That contain functions with names that are unique to that library
  - However is it quite possible for other libraries to contain functions with the same names
  - Namespaces are a means of coping with this

Functions and Namespaces

- The using keyword allows functions to be used without their namespace name
  - e.g. `using namespace std;`
  - Otherwise the function has to be used with its fully qualified name, that includes its namespace
  - e.g. `std::cout << "…";`
- `using` declarations can be made inside individual functions
  - So that they only apply within the body of that function