Capitolo 3 - Functions

<u>Outline</u>

◁ ▷

Capitolo 3 - Functions

Outline

3.1 Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program

3.2 Program Components in C++

- Programs written by
  - combining new functions with "prepackaged" functions in the C++ standard library.
  - The standard library provides a rich collection of functions.
- Functions are invoked by a function call
  - A function call specifies the function name and provides information (as arguments) that the called function needs
  - Boss to worker analogy:

    A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.

3.2 Program Components in C++

- Function definitions
  - Only written once
  - These statements are hidden from other functions.
  - Boss to worker analogy:

    The boss does not know how the worker gets the job done; he just wants it done

3.3 Math Library Functions

- Math library functions
  - Allow the programmer to perform common mathematical calculations
  - Are used by including the header file **`<cmath>`**
- Functions called by writing

  functionName (*argument*)

- Example

```
cout << sqrt( 900.0 );
```

  - Calls the **`sqrt`** (square root) function. The preceding statement would print **`30`**
  - The **`sqrt`** function takes an argument of type **`double`** and returns a result of type **`double`**, as do all functions in the math library

3.3 Math Library Functions

- Function arguments can be
  - Constants

```
sqrt( 4 );
```

  - Variables

```
sqrt( x );
```

  - Expressions

```
sqrt( sqrt( x ) ) ;
sqrt( 3 - 6x );
```

3.4 Functions

- Functions
  - Allow the programmer to modularize a program
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - Local variables passed when the function is called that provide the function with outside information

3.5 Function Definitions

- Create customized functions to
  - Take in data
  - Perform operations
  - Return the result
- Format for function definition:

```
return-value-type  function-name( parameter-list )
  {
        declarations and statements
  }
```

- Example:

```
int square( int y)
{
  return y * y;
}
```

Outline

```
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function
#include <iostream>

using std::cout;
using std::endl;

int square( int );    // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```

Notice how parameters and return value are declared.

1. Function prototype

2. Loop

3. Function definition

```
1  4  9  16  25  36  49  64  81  100
```

Program Output

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;
```

Outline

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

```

   return 0;
}

// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

3. Function definition

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 92 35 14
Maximum is: 92
```

```
Enter three integers: 45 19 98
Maximum is: 98
```

Program Output

3.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters
    - Information the function takes in
  - Return type
    - Type of information the function passes back to caller (default **`int`**)
    - **`void`** signifies the function returns nothing
  - Only needed if function definition comes after the function call in the program
- Example:

```
int maximum( int, int, int );
```

  - Takes in 3 **`int`**s
  - Returns an **`int`**

3.7 Header Files

- Header files
  - Contain function prototypes for library functions
  - **`<cstdlib>`** , **`<cmath>`**, etc.
  - Load with **`#include <filename>`**
    - Example:

```
#include <cmath>
```

- Custom header files
  - Defined by the programmer
  - Save as **`filename.h`**
  - Loaded into program using

```
#include "filename.h"
```

3.8 Random Number Generation

- **`rand`** function

  ```
  i = rand();
  ```

  - Load **`<cstdlib>`**
  - Generates a pseudorandom number between **`0`** and **`RAND_MAX`** (usually 32767)
    - A pseudorandom number is a preset sequence of "random" numbers
    - The same sequence is generated upon every program execution
- **`srand`** function
  - Jumps to a seeded location in a "random" sequence

    ```
    srand( seed );
    srand( time( 0 ) );   //must include <ctime>
    ```

  - **`time( 0 )`**
    - The time at which the program was compiled
  - Changes the seed every time the program is compiled, thereby allowing **`rand`** to generate random numbers

3.8 Random Number Generation

- Scaling
  - Reduces random number to a certain range
  - Modulus (`%`) operator
    - Reduces number between 0 and **`RAND_MAX`** to a number between 0 and the scaling factor
  - Example

```
i = rand() % 6 + 1;
```

    - Generates a number between **`1`** and **`6`**

Outline

1. Define loop

2. Output random number

```cpp
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>

int main()
{
   for ( int i = 1; i <= 20; i++ ) {
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

Notice **`rand() % 6`**. This returns a number between **0** and 5 (scaling). Add 1 to get a number between **1** and **6.**

Executing the program again gives the same "random" dice rolls.

```
         5         5         3         5         5
         2         4         2         5         5
         5         3         2         2         1
         5         1         4         6         4
```

Program Output

```
// Fig. 3.9: fig03_09.cpp
// Randomizing die-rolling program
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>

int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );

   for ( int i = 1; i <= 10; i++ ) {
      cout << setw( 10 ) << 1 + rand() % 6;

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

Outline

1. Initialize seed

2. Input value for seed

2.1 Use `srand` to change random sequence

2.2 Define Loop

3. Generate and output random numbers

Outline

Program Output

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

```
Enter seed: 432
         4         2         6         4         3
         2         5         1         4         4
```

```
Enter seed: 67
         1         6         5         1         4
         5         6         3         1         2
```

Notice how the die rolls change with the seed.

3.9 Example: A Game of Chance and Introducing `enum`

- Enumeration - set of integers with identifiers

  `enum` *typeName* {*constant1*, *constant2…*}**;**

  - Constants start at **`0`** (default), incremented by **`1`**
  - Unique constant names
  - Example:

```
enum Status {CONTINUE, WON, LOST};
```

- Create an enumeration variable of type *typeName*
  - Variable is constant, its value may not be reassigned

```
Status enumVar;  // create variable
enumVar = WON;   // set equal to WON
enumVar = 1;     // ERROR
```

Example: A Game of Chance and Introducing `enum`(II)

- Enumeration constants can have values pre-set

```
enum Months { JAN = 1, FEB, MAR, APR, MAY,
  JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

  - Starts at `1`, increments by `1`
- Craps simulator rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10
      - value becomes player's "point"
      - player must roll his point before rolling 7 to win

```
// Fig. 3.10: fig03_10.cpp
// Craps
#include <iostream>

using std::cout;
using std::endl;

#include <cstdlib>

#include <ctime>

using std::time;

int rollDice( void );  // function prototype

int main()
{
   enum Status { CONTINUE, WON, LOST };
   int sum, myPoint;
   Status gameStatus;

   srand( time( 0 ) );
   sum = rollDice();            // first roll of the dice

   switch ( sum ) {
      case 7:
      case 11:                  // win on first roll
         gameStatus = WON;
         break;
      case 2:
      case 3:
      case 12:                 // lose on first roll
         gameStatus = LOST;
         break;
```

Notice how the **enum** is defined

Outline

1. `rollDice` prototype

1.1 Initialize variables and `enum`

1.2 Seed `srand`

2. Define `switch` statement for win/loss/continue

```
      default:                 // remember point
         gameStatus = CONTINUE;
         myPoint = sum;
         cout << "Point is " << myPoint << endl;
         break;                // optional
   }

   while ( gameStatus == CONTINUE ) {    // keep rolling
      sum = rollDice();

      if ( sum == myPoint )       // win by making point
         gameStatus = WON;
      else
         if ( sum == 7 )          // lose by rolling 7
            gameStatus = LOST;
   }

   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

   return 0;
}
```

Outline

2.1 Define loop to continue playing

2.2 Print win/loss

3. Define `rollDice` function

```
int rollDice( void )
{
   int die1, die2, workSum;

   die1 = 1 + rand() % 6;
   die2 = 1 + rand() % 6;
   workSum = die1 + die2;
   cout << "Player rolled " << die1 << " + " << die2
        << " = " << workSum << endl;

   return workSum;
}
```

Program Output

```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

3.10 Storage Classes

- Storage class specifiers
  - Storage class
    - Where object exists in memory
  - Scope
    - Where object is referenced in program
  - Linkage
    - Where an identifier is known
- Automatic storage
  - Object created and destroyed within its block
  - **`auto`**
    - Default for local variables.
    - Example:

```
auto float x, y;
```

  - **`register`**
    - Tries to put variables into high-speed registers
  - Can only be used with local variables and parameters

◁ ▷

3.10 Storage Classes

- Static storage
  - Variables exist for entire program execution
  - `static`
    - Local variables defined in functions
    - Keep value after function ends
    - Only known in their own function
  - **`Extern`**
    - Default for global variables and functions.
    - Known in any function

3.11 Identifier Scope Rules

- File scope
  - Defined outside a function, known in all functions
  - Examples include, global variables, function definitions and functions prototypes
- Function scope
  - Can only be referenced inside a function body
  - Only labels (**`start:`**, **`case:`**, etc.)
- Block scope
  - Declared inside a block. Begins at declaration, ends at }
  - Variables, function parameters (local variables of function)
  - Outer blocks “hidden” from inner blocks if same variable name
- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere

Outline

1. **Function prototypes**

1.1 Initialize global variable

1.2 Initialize local variable

1.3 Initialize local variable in block

2. **Call functions**

3. **Output results**

```
// Fig. 3.12: fig03_12.cpp
// A scoping example
#include <iostream>

using std::cout;
using std::endl;

void a( void );   // function prototype
void b( void );   // function prototype
void c( void );   // function prototype

int x = 1;       // global var

int main()
{
   int x = 5;    // local variable to main

   cout << "local x in outer scope of main is " << x << endl;

   {             // start new scope
      int x = 7;

      cout << "local x in inner scope of main is " << x << endl;
   }             // end new scope

   cout << "local x in outer scope of main is " << x << endl;

   a();          // a has automatic local x
   b();          // b has static loca
   c();          // c uses global x
   a();          // a reinitializes a
   b();          // static local x retains its previous value
   c();          // global x also retains its value

```

x is different inside and outside the block.

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5
```

Outline

Define Functions

```
   cout << "local x in main is " << x << endl;

   return 0;
}

void a( void )
{
   int x = 25;  // initialized each time a is called

   cout << endl << "local x in a is " << x
        << " after entering a" << endl;
   ++x;
   cout << "local x in a is " << x
        << " before exiting a" << endl;
}

void b( void )
{
    static int x = 50;  // Static initialization onl
                        // first time b is called.
    cout << endl << "local static x is " << x
         << " on entering b" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting b" << endl;
}

void c( void )
{
   cout << endl << "global x is " << x
        << " on entering c" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting c" << endl;
}
```

Local automatic variables are created and destroyed each time **a** is called.

```
local x in a is 25 after entering a
local x in a is 26 before exiting a
```

Local static variables are not destroyed when the function ends.

```
local static x is 50 on entering b
local static x is 51 on exiting b
```

Global variables are always accessible. Function **c** references the global **x.**

```
global x is 1 on entering c
global x is 10 on exiting c
```

Outline

Program Output

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

3.12 Recursion

- Recursive functions
  - Are functions that calls themselves
  - Can only solve a base case
  - If not base case, the function breaks the problem into a slightly smaller, slightly simpler, problem that resembles the original problem and
    - Launches a new copy of itself to work on the smaller problem, slowly converging towards the base case
    - Makes a call to itself inside the **`return`** statement
  - Eventually the base case gets solved and then that value works its way back up to solve the whole problem

3.12 Recursion

- Example: factorial

  $n! = n * (n – 1) * (n – 2) * ... * 1$

  - Recursive relationship ($n! = n * (n – 1)!$)

    $5! = 5 * 4!$

    $4! = 4 * 3!...$

  - Base case ($1! = 0! = 1$)

3.13 Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of two previous ones
  - Example of a recursive formula:

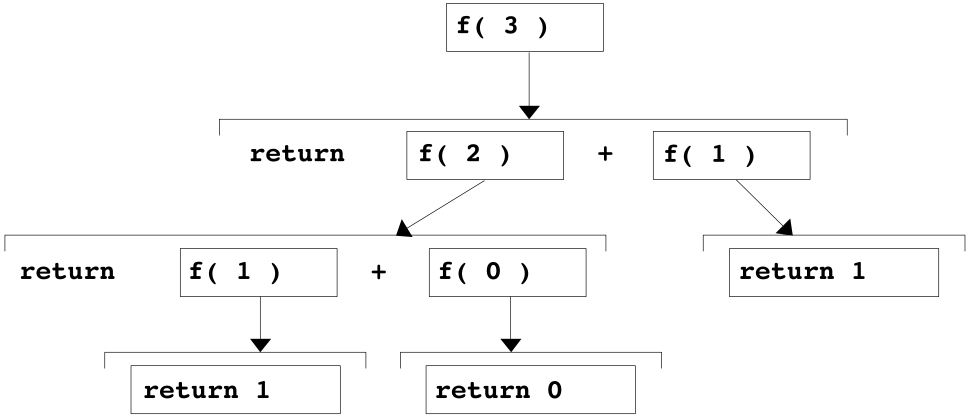
```
fib(n) = fib(n-1) + fib(n-2)
```

- C++ code for **`fibonacci`** function

```
long fibonacci( long n )
{
  if ( n == 0 || n == 1 )  // base case
  return n;
else return fibonacci( n - 1 ) +
  fibonacci( n - 2 );
}
```

3.13 Example Using Recursion: The Fibonacci Series

- Diagram of Fibonnaci function



Outline

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

```
// Fig. 3.15: fig03_15.cpp
// Recursive fibonacci function
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

unsigned long fibonacci( unsigned long );

int main()
{
    unsigned long result, number;

    cout << "Enter an integer: ";
    cin >> number;
    result = fibonacci( number );
    cout << "Fibonacci(" << number << ") = " << result << endl;
    return 0;
}

// Recursive definition of function fibonacci
unsigned long fibonacci( unsigned long n )
{
   if ( n == 0 || n == 1 )  // base case
      return n;
   else                     // recursive case
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

Only the base cases return values. All other cases call the **`fibonacci`** function again.

Program Output

```
Enter an integer: 0
Fibonacci(0) = 0
```

```
Enter an integer: 1
Fibonacci(1) = 1
```

```
Enter an integer: 2
Fibonacci(2) = 1
```

```
Enter an integer: 3
Fibonacci(3) = 2
```

```
Enter an integer: 4
Fibonacci(4) = 3
```

```
Enter an integer: 5
Fibonacci(5) = 5
```

```
Enter an integer: 10
Fibonacci(10) = 55
```

```
Enter an integer: 6
Fibonacci(6) = 8
```

```
Enter an integer: 20
Fibonacci(20) = 6765
```

```
Enter an integer: 30
Fibonacci(30) = 832040
```

```

Enter an integer: 35
Fibonacci(35) = 9227465
```

3.14 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

3.15 Functions with Empty Parameter Lists

- Empty parameter lists
  - Either writing **`void`** or leaving a parameter list empty indicates that the function takes no arguments

```
void print();
```

or

```
void print( void );
```

  - Function **`print`** takes no arguments and returns no value

```
// Fig. 3.18: fig03_18.cpp
// Functions that take no arguments
#include <iostream>

using std::cout;
using std::endl;

void function1();
void function2( void );

int main()
{
   function1();
   function2();

   return 0;
}

void function1()
{
   cout << "function1 takes no arguments" << endl;
}

void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
}
```

Notice the two ways of declaring no arguments.

```
function1 takes no arguments
function2 also takes no arguments
```

Outline

1. Function prototypes (take no arguments)

2. Call the functions

3. Function definitions

Program Output

3.16 Inline Functions

- **`inline`** functions
  - Reduce function-call overhead
  - Asks the compiler to copy code into program instead of using a function call
  - Compiler can ignore **`inline`**
  - Should be used with small, often-used functions
- Example:

```
inline double cube( const double s )
   { return s * s * s; }
```

3.17 References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Used to prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original
- Reference parameter alias for argument
  - **&** is used to signify a reference

```
void change( int &variable )
    { variable += 3; }
```

  - Adds 3 to the variable inputted

```
int y = &x.
```

  - A change to **y** will now affect **x** as well

Outline

```cpp
// Fig. 3.20: fig03_20.cpp
// Comparing call-by-value and call-by-reference
// with references.
#include <iostream>

using std::cout;
using std::endl;

int squareByValue( int );
void squareByReference( int & );

int main()
{
   int x = 2, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;
}

int squareByValue( int a )
{
   return a *= a;   // caller's argument not modified
}
```

Notice the use of the **&** operator

1. Function prototypes

1.1 Initialize variables

2. Print `x`

2.1 Call function and print `x`

2.2 Print `z`

2.3 Call function and print `z`

3. Function Definition of `squareByValue`

Outline

```

void squareByReference( int &cRef )
{
   cRef *= cRef;    // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

3.1 Function Definition of `squareByReference`

Program Output

3.18 Default Arguments

- If function parameter omitted, gets default value
  - Can be constants, global variables, or function calls
  - If not enough parameters specified, rightmost go to their defaults
- Set defaults in function prototype

```
int defaultFunction( int x = 1,
   int y = 2, int z = 3 );
```

```
// Fig. 3.23: fig03_23.cpp
// Using default arguments
#include <iostream>

using std::cout;
using std::endl;

int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
}
```

Outline

1. Function prototype

2. Print default volume

2.1 Print volume with one parameter

2.2 Print with 2 parameters

2.3 Print with all parameters.

3. Function definition

Program Output

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Notice how the rightmost values are defaulted.

3.19 Unary Scope Resolution Operator

- Unary scope resolution operator (`::`)
  - Access global variables if a local variable has same name
  - not needed if names are different
  - instead of **`variable`** use **`::variable`**

```cpp
// Fig. 3.24: fig03_24.cpp
// Using the unary scope resolution operator
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setprecision;

const double PI = 3.14159265358979;

int main()
{
   const float PI = static_cast< float >( ::PI );

   cout << setprecision( 20 )
        << "  Local float value of PI = " << PI
        << "\nGlobal double value of PI = " << ::PI << endl;

   return 0;
}
```

Notice the use of `::`

Outline

1. Define variables

2. Print variables

```
Local float value of PI = 3.141592741012573242
Global double value of PI = 3.141592653589790007
```

Program Output

3.20 Function Overloading

- Function overloading
  - Having functions with same name and different parameters
  - Should perform similar tasks (i.e., a function to square **`int`**s, and function to square **`float`**s).

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

  - Program chooses function by signature
    - signature determined by function name and parameter types
  - Can have the same return types

```
// Fig. 3.25: fig03_25.cpp
// Using overloaded functions
#include <iostream>

using std::cout;
using std::endl;

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

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

int main()
{
   cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

   return 0;
}
```

Functions have same name but different parameters

Outline

1. Define overloaded function

2. Call function

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Program Output

3.21 Function Templates

- Function templates
  - Compact way to make overloaded functions
  - Keyword **`template`**
  - Keyword **`class`** or **`typename`** before every formal type parameter (built in or user defined)

```
template < class T >
   // or template< typename T >
T square( T value1 )
{
   return value1 * value1;
}
```

  - **`T`** replaced by type parameter in function call.

```
int x;
int y = square(x);
```

    - If **`int`**, all **`T`**'s become **`int`**s
    - Can use **`float`**, **`double`**, **`long`**...