
Lecture 3: Functions

Dragan Mirkovic
Department of Computer Science
University of Houston

D. Mirkovic, C++ Programming, Spring 2005

Announcements

- Today:
 - Functions in C++
 - Ch 3. In Irvine.
 - The slides are based on the Kip Irvine lecture slides.
- Quiz #1
 - Thursday 2/10/2005
 - Ch. 1 and Ch. 2.

D. Mirkovic, C++ Programming, Spring 2005

Introduction

- Allow for a better structure in programs (modularity).
- A function is a block of instructions that is executed when it is called from some other point of the program.
- The instruction format is:

```
type name ( argument1, argument2, ... ) statement
```

– Where:

- **type** is the type of data returned by the function.
- **name** is the name by which it will be possible to call the function.
- **arguments** (as many as wanted can be specified).
 - They allow passing parameters to the function when it is called.
- **statement** is the function's body.
 - It can be a single instruction or a block of instructions.

D. Mirkovic, C++ Programming, Spring 2005

General Comments

- Default: `f(void) = int f(void)`
- Example:

```
int sum(int a, int b)
{
    return a+b;           //in module 1
}
```
- Declaration

```
int sum(int, int)       //in module 2
```
- Function signature = function name + parameter types
- External linkage = information about a function not included in a current compilation unit
 - Function prototype

D. Mirkovic, C++ Programming, Spring 2005

Function Name Encoding

- C++ compiler creates unique identifiers for functions having identical names but different parameters
 - Function name encoding or name mangling
- Example:
 - Function prototypes:

```
int sum(int x, int y);  
int sum(float x, float y);
```
 - Encoded as:
`sum_Fii` and `sum_Fff` respectively
- Provides *type-safe linkage*
- The actual encoding scheme depends on the compiler

D. Mirkovic, C++ Programming, Spring 2005

Parameters

- A *parameter value* (or *argument*) is the value passed to a function from the calling program.
- A *parameter variable* is the variable declared in the function parameter list that receives the value.
- Input parameters receive values that have been passed to the function.
- Output parameters return values from the function to the calling program.

D. Mirkovic, C++ Programming, Spring 2005

Future Value Function

- This function calculates the future value of a 1000 investment over 10 years, compounded monthly. Not very flexible, but at least it sticks to a single task.

```
double future_value( double p )  
{  
    double b = 1000 *  
        pow(1 + p / (12 * 100), 12 * 10);  
    return b;  
}
```

parameter variable

D. Mirkovic, C++ Programming, Spring 2005

Future Value Function

The following statements call future_value():

```
cout << "Please enter the interest rate "  
    " in percent:";  
double rate;  
cin >> rate;  
double balance = future_value(rate);  
cout << "After 10 years, the balance is "  
    << balance << "\n";
```

return value

parameter value

D. Mirkovic, C++ Programming, Spring 2005

Making it more Flexible

- This version of `future_value()` is more flexible because the initial balance and number of years are parameter variables:

```
double future_value(double initial_balance,
                    double p, int nyear)
{
    double b = initial_balance
              * pow(1 + p / (12 * 100), 12 * nyear);

    return b;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Using Comments

- This version of `future_value` is commented according to the standards required on your programming projects:

```
double future_value(double initial_balance,
                    double p, int nyear)
// Purpose: computes the value of an investment
//           with compound interest
// Receives:
//   initial_balance - the initial value of the
//                   investment.
//   p               - the interest rate in percent
//   nyear           - the number of years the investment
//                   is held
// Returns:  the balance after nyear years
// Remarks:  interest is compounded monthly
```

D. Mirkovic, C++ Programming, Spring 2005

Error Return

- Use the return statement to exit immediately. In this example, range errors are handled by returning a default value:

```
double future_value(double initial_balance,
                    double p, int nyear)
{
    if( nyear < 0 ) return 0;    // error
    if( p < 0 ) return 0;      // error

    double b = initial_balance
        * pow(1 + p / (12 * 100), 12 * nyear);
    return b;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Boolean functions (1)

- Boolean functions, or *predicates*, are well suited to data validation applications. They work best when given just a single task:

```
bool IsValidYear( int year )
{
    if( ( year > 1600 ) and ( year < 2100 ) )
        return true;
    else
        return false;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Boolean functions (2)

- Boolean functions can return their values into boolean expressions, leading to code that is easier to understand:

```
if( IsValidYear( birthYear )
    and IsLeapYear( birthYear ))
{
    cout << "You were born in a Leap Year\n";
}
```

D. Mirkovic, C++ Programming, Spring 2005

Parameter Variables as Constants

- It is poor coding style to modify parameter variables that were passed by value:

```
void PoorStyle( int x )
{
    cout << "Starting value of X: " << x << endl;
    x = x - 1;    // ???
    cout << "New value of X: " << x << endl;
}
```

(Of course, the effect is only temporary anyway.)

D. Mirkovic, C++ Programming, Spring 2005

Function Prototypes

- A function declaration (or *prototype*) is required when a function call has not been preceded by the full function definition. The prototype indicates essential calling information required by the compiler.

- Example:

```
double future_value(double initial_balance,  
                    double p, int nyear);  
// function prototype
```

D. Mirkovic, C++ Programming, Spring 2005

Side Effects

- A *side effect* occurs when a function produces some effect other than returning a value. For example, it might display information or modify a global variable.

```
double future_value(double initial_balance,  
                    double p, int nyear)  
{  
    if( nyear < 0 )  
    {  
        cout << "Error: invalid number of years\n";  
        return 0;  
    }  
    double b = initial_balance  
              * pow(1 + p / (12 * 100), 12 * nyear);  
    return b;  
}
```

D. Mirkovic, C++ Programming, Spring 2005

Procedures

- A procedure (or *void function*) always produces a side effect because it cannot not return a value.

```
void DisplayReport( double initial_balance,
                   double p, int nyear)
{
    cout << "An investment of " <<
    initial_balance
    << " will be worth "
    (etc.)
}
```

D. Mirkovic, C++ Programming, Spring 2005

Passing by Reference (1)

- Passing a parameter by reference means passing its address to a function. The receiving function may modify the variable via its address. This is often necessary, as in the case of the `swap()` function:

```
void swap( int & x, int & y )
// Exchange the values of two variables.
{
    int temp = x;
    x = y;
    y = temp;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Passing by Reference (2)

- A function that gets input from the user often must use reference parameters because a function cannot return two values.

```
void GetUserName( string & lastName,
                 string & firstName )
{
    cout << "Last name: ";
    cin >> lastName;
    cout << "First name: ";
    cin >> firstName;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Passing by Reference (3)

- In the calling program, the variables passed to the function are themselves updated:

```
string firstName;
string lastName;
GetUserName( lastName, firstName );
// now the values are changed...
```

- Object passed by reference must be a *modifiable lvalue* (lhs in assignment statement)
- The following is wrong, for example:

```
void locate(int &row, int &col); // prototype
...
locate(x+2, y-2);                // error
```

D. Mirkovic, C++ Programming, Spring 2005

Input and Output Parameters

- Input parameters contain data that have been passed to a function from a calling program.
- Output parameters contain data that are set inside the function and returned to the calling program.
- Input/Output parameters receive data from a calling program, modify the data, and return the data to the calling program.

D. Mirkovic, C++ Programming, Spring 2005

Input Parameters

- `ShowReport()` uses input parameters, `hours` and `payRate`:

```
double CalcPay( double hours, double payRate )
{
    return hours * payRate;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Output Parameters

- `GetUserName()`, seen earlier, uses two output parameters:

```
void GetUserName( string & lastName,
                 string & firstName )
{
    cout << "Last name: ";
    cin >> lastName;
    cout << "First name: ";
    cin >> firstName;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Input/Output Parameters

- The `swap()` function uses two input/output parameters:

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

D. Mirkovic, C++ Programming, Spring 2005

Passing by Constant Reference (1)

- When passing class objects by value, always pass by constant reference. The compiler will not let the called function modify constant parameters.

```
void ShowName( const string & firstName,
              const string & lastName )
{
    cout << firstName << " " << lastName;

    firstName = "George"; // error
}
```

D. Mirkovic, C++ Programming, Spring 2005

Passing by Constant Reference (2)

- Constant qualifier will allow passing an expression as a reference parameter
- Example:

```
void Display1(int &n);
void Display2(const int &n);
...
Display1(x + 3); // error
Display2(x + 3); // ok
```

D. Mirkovic, C++ Programming, Spring 2005

Don't Pass Objects by Value

- Passing an object by value wastes memory and processing time by creating a duplicate copy of the object on the stack.

```
void ShowName( string firstName, string
               lastName )
{
    cout << firstName << " " << lastName;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Functions that return references

- C++ functions can return references
- Example:

```
class Transcript {
public:
    void Erase();
    void Print() const;
}
class Student {
public:
    const Transcript &GetTranscript();
private:
    Transcript trs;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Functions that return references₍₁₎

- Returning a reference exposes the data to possible changes by the user
- Use `const` keyword to prevent changes
- Never return a non-constant reference to a private data member
- Never return a reference to an automatic object:

```
int & BadExample() { int n = 20; return n; }
```

D. Mirkovic, C++ Programming, Spring 2005

Self-Referencing

- Within a member function, the keyword `this` is the name of an implicit pointer to the current object.

- Example:

```
Point & Point::Set(int xVal, int yVal)
{
    x = xVal; y = yVal;
    return *this;
}
```

- The pointer should be dereferenced (the current object is the pointer and `this` is a pointer to a pointer)

D. Mirkovic, C++ Programming, Spring 2005

Variable Scope (1)

- A variable declared within a block is only visible from the point it is declared until the end of the block.

```
if( X > Y )
{
    // begin block
    int sum = X + Y;
}
// end block

cout << sum;    // error: not visible
```

D. Mirkovic, C++ Programming, Spring 2005

Variable Scope (2)

- Be careful not to *mask* a variable in an outer scope with another having the same name in an inner scope. If this function receives input of 10, what will it return?

```
int Evaluate( int n )
{
    int j = 99;
    if( n > 5 )
    {
        int j = 20;
    }
    return j;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Global Variables

- A *global* variable is declared outside of any function block. Avoid these whenever possible. Never use them as a way of avoiding passing parameters to functions.

```
double g_initial_balance, g_principal;
int g_nyear;

double future_value()
{
    return (g_initial_balance *
            pow(1 + g_principal / (12 * 100), 12 * g_nyear));
}
```

D. Mirkovic, C++ Programming, Spring 2005

Using Preconditions (1)

- The `assert()` macro can be called when you want to absolutely guarantee that a variable satisfies some requirement. Range checking is common:

```
double future_value(double initial_balance,
                   double p, int nyear)
{
    assert( nyear >= 0 );
    assert( p >= 0 );
    double b = initial_balance
                * pow(1 + p / (12 * 100), 12 * nyear);
    return b;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Using Preconditions (2)

- If the expression passed to the `assert()` macro is false, the program halts immediately with a diagnostic message:

```
Assertion failure in file mysub.cpp,  
line 201:  nyear >= 0
```

- Your program, of course, is given no chance to recover from the error. Also, `assert` only works when your program is compiled to a `DEBUG` target. (It can be disabled by `#define NDEBUG`)
- `Assert` aborts the program without closing the files

D. Mirkovic, C++ Programming, Spring 2005

Using Preconditions (3)

- A much better way to handle errors is to use the C++ *exception handling* mechanism. (Ch. 8)
- Your program is given a chance to recover and try again:

```
double future_value(double initial_balance,  
double p, int nyear)  
{  
    if( nyear >= 0 )  
        throw RangeException( "nyear", 0 );  
  
    // etc.
```

D. Mirkovic, C++ Programming, Spring 2005

Recursion (1)

- *Recursion* happens when a function either (a) calls itself, or (b) calls another function that eventually ends up calling the original function. Here, a function calls itself, creating what looks like *infinite recursion*:

```
void ChaseMyTail ()
{
    //...
    ChaseMyTail ();
}
```

D. Mirkovic, C++ Programming, Spring 2005

Recursion (2)

- *Indirect recursion* occurs when a series of function calls leads back to a function earlier in the series, creating a looping effect. This appears to work, but unwinding the execution stack can be a problem.

```
void One ()                void Three ()
{                          {
    Two ();                Four ();
}                          }

void Two ()               void Four ()
{                          {
    Three ();              One ();
}                          }
```

D. Mirkovic, C++ Programming, Spring 2005

Recursion (3)

- Recursion has to be controlled by providing an exit condition that allows the recursion to stop when a base case is reached.

```
long Factorial( int n )
{
    if( n == 0 ) ←————— Exit condition
        return 1;
    else
    {
        long result = n * factorial( n - 1 );
        return result;
    }
}
```

D. Mirkovic, C++ Programming, Spring 2005

Recursive Factorial Function

- Think of $n!$ as the product of n times $(n-1)!$. This is always true, as long as $n > 0$. The Factorial function works on this principle:

```
5! = 5 * 4!
and 4! = 4 * 3!
and 3! = 3 * 2!
and 2! = 2 * 1!
and 1! = 1 * 0!
and 0! = 1          (by definition)
```

D. Mirkovic, C++ Programming, Spring 2005

Recursive Factorial Function

- Recursion may not be the most effective way for computing factorials
- Wide variety of problems have combinatorial complexity and the best way to encode them is by using the recursion
- Example:
 - *Question:* how many ways can you rearrange the five letters ABCDE?

Answer: 5! ways.

D. Mirkovic, C++ Programming, Spring 2005

Friend functions

- Sometimes a function not belonging to a certain class requires access to the class's private and protected members
- The access can be granted by using a `friend` keyword
- A class cannot declare itself a friend of another class
- Example:
 - Global function accessing information from two different classes

```
void HireStudent (Student &S, Employee &E, float rate)
{S.employed = 1; E.payRate = rate;}
```

 - Where Student and Employee classes are:

D. Mirkovic, C++ Programming, Spring 2005

Friends functions - Example

```
class Student; //forward declarati
class Employee {
public:
    Employee( long idVal);
    friend void HireStudent
        (Student &S,
         Employee &E,
         float rate)
private:
    long id;
    float payRate;
}

class Student {
public:
    Student( long idVal);
    friend void HireStudent
        (Student &S,
         Employee &E,
         float rate)
private:
    long id;
    int employed;
}
```

D. Mirkovic, C++ Programming, Spring 2005

Friend classes

- Class in which all member functions have been granted full access to the private and protected members of another class
- Friend status is not transitive!
- Example:

```
class class1 { friend class2; //...};
class class2 { friend class3; //...};
class class3 is not automatically a friend of class1;
```
- Close coupling between classes
 - Not always desirable

D. Mirkovic, C++ Programming, Spring 2005

Function Overloading

- Multiple functions have the same name but different parameters
 - Should be used for functions that perform the same task on different types of objects

- Functions must have different signatures

- Examples:

```
void swap (double &, double &);  
void swap (int &, int &);  
void swap (Point &, Point &);
```

Or

```
istream & get (char &)  
istream & get (unsigned char &)
```

D. Mirkovic, C++ Programming, Spring 2005

Overloading Resolution

- Argument conversion
 - If a function is called with argument types different from its parameter types
 - As long as the types are assignment compatible, the arguments can be converted
- The overloaded functions are resolved on the best matching principle
 - For each argument the compiler finds the set of all functions that are assignment compatible
 - If the intersection contains a single function it is selected for the call (error otherwise).

D. Mirkovic, C++ Programming, Spring 2005

Summary

- Fundamentals of declaring and using functions (signatures, external linkage,...)
- Function name encoding
- Function parameters
- Preconditions
- Default function arguments
- Function overloading
- Recursions
- Friends

D. Mirkovic, C++ Programming, Spring 2005

Homework

- Exercises 3.10.2, page 102 and 3.10.3 on page 103 in text.

D. Mirkovic, C++ Programming, Spring 2005