SFU CMPT-212 2008-1 Topic: Functions

Ján Maňuch

E-mail: jmanuch@sfu.ca

Wednesday 6th February, 2008

Functions

Functions are an essential building block of structured and object-oriented programming.

Phases in the life cycle of a function:

- function *prototype* (function declaration): describes the shell of a function (name, parameters and return value)
  `return_type function_name(parameters);`
- function *definition*: describes what the function does

```
1 return_type function_name(parameters)
2 { statements } // function body
```

- function *call*: using the function within other code
  `function_name(arguments)`

Function call

Consider a function:

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

What happens when we call the function?

```
1   double a=1.2;
2   double b=cube(a);
```

- program stores the address of the next instruction
- *copies* the values of arguments to the stack ("passing the arguments")
- jumps to the address of the function — executes function
- if needed, places a return value in a specified location (a register or memory)
- jumps to the saved address

Function prototype

- describes the function *interface* to the compiler
- *purpose:* the compiler knows what kind of arguments to look for on the stack when it starts executing the function;
  and knows what return value to retrieve from the specified location after it finishes execution of the function

Note: does not need to provide the names of parameters (just types), and the names can be different from those specified in the function definition

Note: you have to specify type for each parameter

Example:

```
1 double length(double a, b);          // incorrect
2 double length(double a, double b); // correct
```

- *automatic conversions:* arguments and returned value are automatically converted to types specified in the prototype

void

Note. *The* `void` *type indicates "nothing". No variables can be declared as* `void` *(but you can declare a pointer* `void *`*, which can point to anything).*

Uses of `void`:

- as the only parameter of a function with no parameters (optional)
- as the return value of a *procedure*, a function that does not return any values (mandatory)

Note: in C, an empty parameter list means that function can take any number of arguments (e.g., `printf`);
in C++, it means no parameters!

Remark: to see how to declare a function with a variable number of parameters, look at
`http://www.mycplus.com/tutorial.asp?TID=177`

Sample questions

- What happens when a function is called?
- Consider the following function

```
1 float cube( double x)
2 { return x*x*x; }
```

and the following line of code

```
1 int a,b=2;
2 a=cube(b);
```

How many conversions happen when program executes line 2 of the code?

Passing the function arguments

3 possibilities:

- by *value*
- by *address* (through a pointer)
- by *reference* (C++ feature)

Note. *Unless explicitly specified, C++ passes function arguments* by value.

Example: swap1.cpp

Note (Passing arguments by address)**.** *Basically, it's again passing arguments by value, but the arguments are pointers to data.*

Example: swap2.cpp

Passing arrays

- syntax:

```
1 int sum(int array[], int size); // or
2 int sum(int *array, int size);
```

  the above declaration are equivalent (remember arrays are pointers).

- If we call

```
1   int a[10]; // declaration
2   sum(a);    // function call
```

  in function `sum`, `array[1]` will access directly `a[1]`.

  Example: arrfun2.cpp [Prata]

- For multidimensional arrays, you have to specify all sizes but the first in the type:

```
1   int data[5][5][5]; // 5x5x5 array
2   // ...
3   int sum(int a[][5][5], int xSize); // prototype of the function
```

const

Question: How to protect elements of an array passed to a function which is not supposed to modify the array?

Answer: Use `const`.

Example:

```
1  int sum(const int array[], int size) {
2    array[1]=10; // compiler error
3    ...
4  }
```

- `array` is an array to `const int`, each element of the array has a `const` value (within function `sum`) which cannot be modified

- Note the difference:

```
1    int a,b;
2    const int *pa=&a; // a pointer to const int
3    int *const pb=&a; // a const pointer to int
4    pa=&b;  // VALID
5    pb=&b;  // INVALID
6    *pa=10; // INVALID
7    *pb=10; // VALID
```

- why to use pointer/array to `const` data?
  - protects from programming errors (unwanted modification to data)
  - you can use `const` arguments in a function call

Example:

```
1  int sum(int *array, int size);
2  int sum_const(const int *array, int size);
3
4  int main() {
5    const int a[asize]={1,2,3,4,5};
6    sum(a,asize);       // compiler error
7    sum_const(a,asize); // OK
8  }
```

Passing structures

- recall structures can be assigned one to another (*memberwise assignment*)
- when passing structures by value, a copy of the original structure is created

Example: travel1.cpp [Prata]

- not efficient: values are copied uselessly
- pass structure addresses instead (or references, as we will see later):
  `travel_time *sum(travel_time *t1, travel_time *t2)`

Problem:

- `sum` has to allocate memory for new structure `travel_time` and the user of `sum` should remember to free the structure after using it!
  beware: do not return an address of a *local* variable!

  – **Local variable scope**: Variables declared inside a function *are no longer available* after the completion of the function call.

- there is a better solution:
  instead of returning the structure it's better if the user of `sum` provides the structure where to write the result

Example: travel2.cpp

Passing arguments by reference

Function arguments can be passed by reference using the "*reference*" operator &. This operator is *different* from the "*address*" operator.

Example: swap3.cpp

Reference variables

```
1  int count;
2  int &c=count; // c is an alias for count
```

Example: firstref.cpp [Prata]

Note: You cannot assign value to reference in other place then in declaration:

```
1  int count,count2;
2  int &c; // compiler error;
3          // reference variable has to be initialized
4  c=count2;
```

Reference properties

What happens if call `swap` from swap3.cpp as follows:

```
1  int x,y;
2  swap(x+2,y); // compiler error
```

- `x+2` creates a temporary variable containing the value of `x`$+2$ which is passed to `swap`
- the value of `y` would be lost after swapping (the temporary variable exists only temporarily)
- hence, the call above is *forbidden* by compiler

When is a temporary variable created?

- the argument is not an *Lvalue*, i.e., it cannot be assigned a value: either expression, or a constant
- wrong type argument, requiring conversion

Note: A temporary variable can be used as a argument to a reference parameter, but it has to be a `const` reference.

Example:

```
1  double cube(const double &side) {
2    // side += 1; // compiler error: side is const
3    return side*side*side;
4  }
5
6  int main() {
7    double a=cube(1.2); // ok
8  }
```

Question: If we don't want to modify the value, wouldn't it be easier to use *passing by value* for `side` instead?

Answer: "Yes". However, passing by a `const` reference make sense when passing big structures, which are not going to be modified. (Passing by value would make a copy of each data member of the structure.)

When to use reference arguments?

- to be able to modify the object,
- to avoid copying entire object,
- to return a reference from a function to an existing object and avoid copying the object.

In all cases we could use pointers too, but using references is easier.

Example: travel3.cpp

Returning from functions

- By default, functions return values *by value* (travel1.cpp).
- From the point of view of program flow, the effect of `return` on a function is similar to the effect of `break` on a loop.

Note. *Function calls* cannot appear *on the left sides of assignments unless*

- *a pointer is returned and dereferenced: Lside1.cpp; or*
- *a reference is returned: Lside2.cpp.*

Note: Never return a reference to a local variable!

Example:

```
1 travel_time & estimate()
2 {
3   travel_time t = {4, 32};
4   return t;  // dangerous!
5 }
```

Sample questions

- What are the possibilities of passing arguments to functions? Show three short examples for each possibility.
- What is the difference between the following declarations?

```
1 int a;
2 const int *p=&a;
3 int *const p=&a;
```

- In which situation it's necessary to use a `const` reference for a parameter to the function instead of a non-`const` reference?
- Which of the following lines will generate a compiler error. Explain why!

```
1 int i,j;
2 int &k=&j;
3 int *p=&i;
4 *p=&j;
```

- What is the advantage of passing an argument by constant reference over passing by value?
- What's wrong with the following code?

```
1 void print(double & a)
2 {
3   cout <<"double: "<<a;
4 }
5 // ..
6 double x=3.1;
7 print(x+0.1);
```

How would you fix it?

Pointers to Functions

Functions are code entry points in memory, with memory addresses.
Function addresses can be assigned to pointer variables.

- *declaration*:

```
1   return_type ( *pointer_name )( arguments );
```

Example: `double (*pf)(int,int);`
— `pf` is a pointer to function that takes two `int`s and returns `double`

Note: `double *pf(int,int);`
is a prototype of a function that takes two `int`s and returns a pointer to `double`

- *taking the address of a function*: just use the name of the function
Example:

```
1   double add(int,int); // a prototype of function
2   pf=add;              // pf points to the add() functio
```

- *using a pointer to call a function*:

```
1   (*pointer_name)( parameters ); // C style
2   pointer_name( parameters );    // C++ style
```

Example:

```
1   int i,j;
2   double a=(*pf)(1,2);
3   double b=pf(i,j);
```

Usage:

- generic functions;
  Example: generic.cpp
- hooks;
  Example: atexit.cpp

Inline functions

- *the compiler (probably) replaces the function call with the function code*
- as we have seen when calling a function, there is a certain machinery taking place, which takes time
- if we have a function with a short code which executes fast (compared to the machinery) and is called often, it makes sense to declare the function as an inline function
- inline functions are declared with the `inline` keyword:
```
inline double cube(double x);
```
- *advantage*: faster execution of the function
- *disadvantage*: the code of the function will appear in the executable file as many times as the number of places calling the function

Default arguments

- Function parameters, starting with the last, can be assigned default values.
  Example: `int func(int i, int j=3, int k=5)`
  This way, the number of parameters of a function can be variable up to a maximum number of parameters.
  Example:

```
1   func(5);     // calls func(5,3,5)
2   func(5,6);   // calls func(5,6,5)
3   func(5,6,7);
```

- Default parameters are allowed either in the function prototype (usually) or the function definition, but not both.

Example: default.cpp

Function Overloading (Polymorphism)

- the same name can be used for multiple functions with different argument lists (also called *function signatures*)
- the compiler will automatically recognize which version of the function to call by the number and types of arguments

Example: overloading.cpp

- reference to a type and the type have the same signature
- if the arguments doesn't match any signature, the compiler tries to use *automatic conversions* to match them;
- there can be several ways how to do it: if this happens we get a compiler error "ambiguous call" (exception: `const`, see overloading2.cpp)

Usage: functions performing the same task on different forms of data

Function templates

Function templates are high-level mechanisms designed for dealing with structurally similar processing for different types of data. Function templates are a code *abstraction* mechanism.

Example: swap4.cpp

- Function templates are not functions, they are *generic function descriptions*.

Note (Implicit instantiation)**.** *An instance of a function template is* created implicitly *by matching parameters of a call with the argument templates.*

Example:

```
1  int i,j;
2  Swap(i,j);
```

- the compiler generates the `int` version of function `Swap` and calls it

Note (Explicit instantiation)**.** *An instance of a function template can be declared* explicitly.

Example:

```
1   template void Swap<long>(long &,long &);
```

Note (Explicit specialization)**.** *An instance of a function template can* explicitly overload *the template.*

- if we want to have different behavior for a particular type

Example: say that for `char*` the `Swap` function is not supposed to exchange the addresses of strings but copy the strings from one location to another, and vice versa:

```
1   template <> void Swap<char*>(char *&s1,char *&s2)
2   {
3     ... code copying the strings around ...
4   }
```

Example: swap5.cpp

- function templates can be overloaded as well: twotemps.cpp [Prata]

Which function to call? (Overload resolution)

The compiler makes list of all candidates and ranks them. Among the functions with the highest rank it finds the best match.

If there is only one, fine: use the function. Otherwise, a compiler error: "ambiguous call".

Ranking:

1. exact match:
   - if there a non-template function, use it;
   - otherwise, if there is a template specialization, use it;
   - otherwise, use template;
2. conversion by promotion (as `short` to `int`)
3. automatic conversion (as `int` to `short`)
4. user-defined conversion

More details: see Chapter 8 of [Prata]

Sample questions

- What is the difference between following two lines:

```
1 int (*p)(int);
2 int *p(int);
```

- Declare a function `f` taking an `int` as parameter and returning a pointer to function taking `double` as a parameter and returning a pointer to `char`.
  Answer: pointer_function.cpp
- When is it useful to declare a function as an `inline` function?
- What are default arguments? How can you specify them? Give an example when default arguments are useful.
- What is function overloading? Give one example when the function overloading is very useful.

- Write a function template `Search` that can be used to search for a particular element in an array of arbitrary type `T`. The function should have as parameters an array (of type `T`), the size of the array (of type `long`) and the element (of type `T`) it is to search for. As result is given a pointer to the element searched for. If this element cannot be found, a null pointer should be returned.
- What is the difference between explicit instantiation and explicit specialization?
- Write a function template `doublesize` that can be used to double the size any dynamic array (allocated with the `new` operator) of arbitrary type `T`. The function should have as parameters: a reference to a pointer to the first element of an array of type `T` (that is a reference to a pointer to `T`) and a reference to the size of the array (of type `long`). It does not return anything. The function should allocate a new dynamic array of size twice the original size, copy the existing values, deallocate the old array and update the references passed to the function.

For example, the function template could be used in the following situation:

```
1 long size=10;
2 float *p=new float[size];
3 doublesize(p,size);
```

after which size is equal to `20`, and `p` points to a dynamic array of `float`s of size `20`.