Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger

LS 11 Algorithm Engineering

Lecture 6 • Winter 2011/12 • Nov 22

technische universität dortmund

department of computer science

Functions

- Functions group commonly used code into a unit which can be reused.
- Functions
  - are used to organize programs into smaller, independent units
    → makes program easier to understand
  - encapsulate algorithms that apply to a specific set of data
    → allows easy (and flexible) reuse of code
- We have already implemented and used functions!
  - We always implement the **`main()`** function in a program.
  - We used the **`std::sort()`** function for sorting a container
    → excellent example for a flexible algorithm
  - **`std::getline()`** is also a function

A Function

return type
name
parameters
body
return a value

```
// power(a,b) computes a to the power of b
double power(double base, unsigned int exponent)
{
    double p = 1.0;
    for(unsigned int i = 0; i < exponent; ++i)
        p *= base;

    return p;
}
```

- We must specify:
  - a return type: `double`
  - a name for the function: `power`
  - a list of parameters with their types:
    `double base, unsigned int exponent`
  - a block of code, the body of the function
- Inside the body, we have to return a value using `return`

Using our power function

```
double power(double base, unsigned int exponent)
{
    double p = 1.0;
    for(unsigned int i = 0; i < exponent; ++i)
        p *= base;

    return p;
}

int main()
{
    for(double i = 1.0; i < 8.0; ++i) {
        for(unsigned int j = 0; j < 5; ++j)
            cout << setw(8) << power(i,j);
        cout << endl;
    }
    return 0;
}
```

Output of the program

```
  1         1         1         1         1
  1         2         4         8        16
  1         3         9        27        81
  1         4        16        64       256
  1         5        25       125       625
  1         6        36       216      1296
  1         7        49       343      2401
```

Invoking (Calling) a Function

- We call a function as follows:

```
double number = power(2.0, 4);
```

- The following happens:
  - The arguments in the function call (here: 2 and 4) are evaluated (trivial in this case, but could also be arbitrary expressions)
  - The values of the function's parameters are set to the corresponding arguments:
    - power's **`base`** is set to 2.0
    - power's **`exponent`** is set to 4
  - The body of the function is executed
  - The function returns once a **`return`** statement is executed
  - The value returned by the function is the value of the expression after **`return`**

Example: Nested function calls

```
int inc(int a) {
    return ++a;
}

int add(int a, int b) {
    return a + b;
}

int triple(int a) {
    return 3 * a;
}

int main() {
    int a = 4, b = 2;
    cout << add( triple(a), inc(b) ) << endl;

    return 0;
}
```

has no effect on **a** or **b** in the **main** function!

The program returns:

```
15
```

It computes:

(3*4) + (2+1)

Flow of Control

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

void print_2_3_4(int value, int number)
{
    cout << "\n" << value <<
        " " << value;

    if(number <= 2)
        return;

    cout << " " << value;

    if(number <= 3)
        return;

    cout << " " << value;
}
```

```cpp
int main()
{
    int a = 2;

    print_2_3_4(0, a);
    print_2_3_4(2, ++a);
    ++a;

    // NEVER do something
    // like this!
    print_2_3_4(++a, a++);

    cout << endl;

    return 0;
}
```

Flow of Control Explained

- A function without return type can be declared as `void`
  - In this case we can use `return` without a value
  - If a function is declared as `void`, we can also omit the return statement
    → The function returns when we reach the end of the function body
- The execution of a function stops immediately when we hit a `return` statement
- There may be any number of `return` statements within a function body
- A function can also have an empty parameter list:

```
int doSomething() { … }
```

Flow of Control Explained

- When a function is called, its arguments are evaluated first, then the function is executed
- You can rely on the fact that all arguments will be evaluated before the execution of the function begins.
- You cannot rely on the order in which the arguments are evaluated!
- Do not write code like this:

```
// NEVER do something
// like this!
print_2_3_4(++a, a++);
```

- It is unspecified what happens!

Declaration of Functions

- Like variables, functions must be declared before they can be used:
  - Either by writing the code of the whole function;
  - or by just giving its prototype, e.g.

```
int power(double base, unsigned int exponent);
```

  - in the latter case, you must write the whole function somewhere, e.g. in a different source file

Call by Value

- Functions work on the values of their arguments (call by value)
- Possible disadvantages:
  - The values are copied to the parameter variables, this might be costly
  - Modifications on the parameter variables are lost once the function call returns
- The following example does not work as expected:

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int c = 4, d = 7;

    cout << c << " " << d << endl;
    swap(c,d);
    cout << c << " " << d << endl;

    return 0;
}
```

References

- To solve this problem, we can use references
- A reference is just a new name or alias for a variable
- By using references, we can have multiple "variable names" for the same memory location.
- References are declared as follows:

```
int  a = 7;
int &b = a;
```

  Here, **b** becomes a new name for the location of variable **a**.

- The following code sequence will print 8:

```
b = 8;
cout << a;
```

- References are in particular useful for function parameters!

Call by Reference

- Let's use reference parameters for **swap**:

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int c = 4, d = 7;

    cout << c << " " << d << endl;
    swap(c,d);
    cout << c << " " << d << endl;

    return 0;
}
```

- Now our program works as expected and exchanges the values of **c** and **d**.

Example: Passing a vector to a function

- Reference parameters are useful to avoid unnecessary copying of data
- Example: We want to print a vector

```
// call-by-value variant
void print_vector_cbv(vector<int> v)
{
    cout << "{";

    vector<int>::iterator it;
    for(it = v.begin(); it != v.end(); ++it)
       cout << " " << *it;

    cout << " }" << endl;
}
```

Call-by-Value

The whole vector must be copied!

Example: Passing a vector to a function

- Reference parameters are useful to avoid unnecessary copying of data
- Example: We want to print a vector

```
// call-by-reference variant
void print_vector_cbv(vector<int> &v)
{
    cout << "{";

    vector<int>::iterator it;
    for(it = v.begin(); it != v.end(); ++it)
        cout << " " << *it;

    cout << " }" << endl;
}
```

Call-by-Reference

No copy required

Const References

- Sometimes we want to explicitly express that a reference parameter is not changed (we just want to avoid copying)
- Use a const reference!

```
// call-by-const-reference variant
void print_vector_cbv(const vector<int> &v)
{
    cout << "{";

    vector<int>::const_iterator it;
    for(it = v.begin(); it != v.end(); ++it)
        cout << " " << *it;

    cout << " }" << endl;
}
```

Call-by-Const-Reference

No copy required

We have to use a **const_iterator**!

The Conditional Operator

- The conditional operator is a convenient notational alternative to simple `if`-`else` statements
- Example:
  - Instead of writing:

```
if (x > 0) a = b else a = c+1;
```

  - We can write:

```
a = (x > 0) ? b : c+1;
```

- The general form is

```
condition ? expr1 : expr2
```

  - If *condition* evaluates to true *expr1* is evaluated and returned
  - Otherwise *expr2* is evaluated and returned

The switch statement

```
char c; cin.get(c);

while(c != 'x')
{
    switch(c)
    {
    case 'a':
        ++count_a; break;
    case 'e':
        ++count_e; break;
    case 'i':
        ++count_i; break;
    default:
        ++count_other;
    }

    cin.get(c);
}
```

- `switch(`***`expression`***`)`
  - evaluates ***`expression`*** and jumps to the corresponding `case`
  - ***`expression`*** must be integral
- `case` ***`constant`***`:`
  - ***`constant`*** must be a constant
  - execution continues until a `break` statement occurs
  - no `break` statement: next case will also be executed, but not `default`
- `default:`
  - this (optional) case is executed if none of the above cases applies

Preparations for next week

- Overloading functions
- Comma operator