ADVANCED PROGRAMMING IN C++

Basics 2

Patrick Bader · SS 2023

Function Parameters

C: Always call by value

C++:

- Default: call by value
- Other kinds of calls are also possible:
  - Call by value
  - Call by reference
  - Call by constant reference
  - ...

Java:

- Elementary data types: call by value
- Reference types: call by value of the reference itself not the actual object

Call By Value

Parameter values are copied into the function arguments (even if it is huge)

Function can only modify its own copy

```
void f(int n) {
  n++;
}

int main() {
  int x = 2;
  f(x);
  cout << x << '\n';
  return 0;
}
```

Output:

```
2
```

Call By Reference

Function argument is of reference type: `&` in front of argument name

Refers to the original parameter value which is passed to the function

Function can modify the original parameter value

```
void f(int &n) {
  n++;
}

int main() {
  int x = 2;
  f(x);
  cout << x << '\n';
  return 0;
}
```

Output:

```
3
```

Call By Const Reference

Function argument is of const reference type `const &` in front of argument name

Refers to the original parameter value which is passed to the function

Function is not allowed to modify the original parameter value, as it is declared constant.

```
void f(const int &n) { // or: f(int const &n)
    n++; // <-- compile time error, constant value must not be modified
}

int main() {
    int x = 2;
    f(x);
    cout << x << '\n';
    return 0;
}
```

Arrays

Fixed number of consecutive elements of **same** type.

Elements can be of arbitrary type

Number of elements is defined with brackets:

```
int arr[10];
```

Size can be ommitted when directly initialized:

```
int arr2[] = { 1, 2, 3 };
```

Beware: Implicitly convertible to const pointer of element type

Should be used **rarely** and **only in low-level** code since there are better alternatives, e.g.:

```
std::array<int, 3> arr3 = {1, 2, 3};
```

Complex Data Types

Three types: `struct`, `class`, `union`

Allow data abstraction by aggregating data and related operations into single entities

Are **value types** like all other primitive types

Should be implemented to behave like primitive types

Needed for object-oriented programming

Complex Data Types - Structures

```
#include <iostream>
struct Point    // all members in a struct are public by default
{
  int x;
  int y;
};              // semicolon is needed (names of instances may be provided)

int main()
{
  Point v = { 1, 2 }; // creates an instance of Point with v.x == 1 and v.y ==2
  Point w;            // creates an instance of Point, members remain uninitialized
  w = v;              // copy values of v into w
  v.x = 10;
  w.y = 20;
  std::cout << "v: (" << v.x << ", " << v.y << ")\r\n";
  std::cout << "w: (" << w.x << ", " << w.y << ")\r\n";
}
```

Output:

```
v: (10, 2)
w: (1, 20)
```

Complex Data Types - Classes

```
#include <iostream>
class Point    // same as struct except all members in a class are private by default
{
public:         // make the following members public
  int x;
  int y;
};              // semicolon is needed (names of instances may be provided)

int main()
{
  Point v = { 1, 2 }; // creates an instance of Point with v.x == 1 and v.y ==2
  Point w;            // creates an instance of Point, members remain uninitialized
  w = v;      // copy values of v into w
  v.x = 10;
  w.y = 20;
  std::cout << "v: (" << v.x << ", " << v.y << ")\r\n";
  std::cout << "w: (" << w.x << ", " << w.y << ")\r\n";
}
```

Output:

```
v: (10, 2)
w: (1, 20)
```

Member Functions

Member functions (methods) belong to class instances

Are **declared** inside the class declaration

Declaration:

```
class MyClass
{
  int a;
public:
  void add(int b);
};
```

Definition (outside class declaration):

```
void MyClass::add(int b)  // Define member function add of class MyClass
{
  a += b;                 // or: this->a += b;
}
```

Have an implicit `this` parameter which `points to` the object the method is called on. Can be used to disambiguate variable names.

Code Organization

Header file

Usually contains only declarations

`MyClass.h`

```
#ifndef __MYCLASS_H__
#define __MYCLASS_H__

class MyClass
{
  int a;
public:
  void add(int b);
};
#endif
```

Source file

Contains definitions (actual implementations)

`MyClass.cpp`

```
#include "MyClass.h"

void MyClass::add(int b)
{
  this->a += b;
}
```

Code Organization - Usage

`main.cpp`

```
#include "MyClass.h"

void main()
{
  MyClass c;
  c.add(7);
}
```

- First include header file to make the class declaration visible
- The class can then be used as if it were declared locally

Build Process

Preprocessor

Allows inclusion of header files, macro expansions, and conditional compilation

Takes lines beginning with # as directives

Compiler

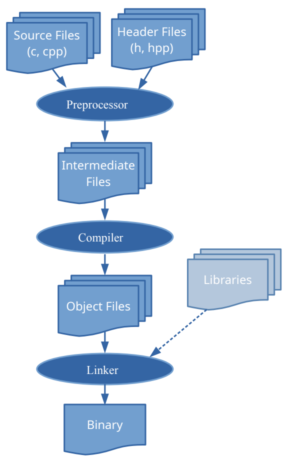
Translates human readable source code into computer executable machine code

C++ compilers are platform dependent

Linker

Takes one or more object files and combines them into a single binary file

Linking can be dynamic (dll, so) or static (lib, a)



Preprocessor In A Nutshell

The preprocessor **operates on text**

Used to make simple **text transformations** (replaces text with other text)

Used to **combine files** into a single file

Does not take C++ syntax into account!

Only use the preprocessor when absolutely necessary!

Preprocessor In A Nutshell

Insert file from default directories mostly used for headers of the standard library which have no file extension and for library headers:

```
#include <string>
```

Insert file relative to current directory mostly used for headers of own code:

```
#include "Class.h"
```

Preprocessor In A Nutshell

Define a simple macro:

```
#define FOO
```

Check whether macro is defined:

```
#ifdef FOO
// Conditionally use code
#endif
```

Check whether macro is not defined:

```
#ifndef FOO
// ...
#endif
```

Define an object-like macro:

```
#define VALUE 10
```

Preprocessor In A Nutshell

Remove macro definition:

```
#undef VALUE
```

Define macro a function-like macro:

```
#define BAD_MAX(A, B) A > B ? A : B

#define BETTER_MAX(A, B) (((A) > (B)) ? (A) : (B))

#define UNLESS(ARG) if(!ARG)
```

Multi-line macros:

```
#define THROW_ON_ERROR(CODE) if(CODE == 0x12){ \
  throw my_exception(CODE); \
}
```

Complex Data Types

Complex types, like `struct`s or `class`es should mimick the behaviour of built-in types

Primitive types support various operators, e.g. `+`, `-`, `>>`, `!`, ...

Complex types should also support these operators if needed

Operator Overloading

It is possible to overload operators for own data types:

```
Vector2D v1, v2, v3;
v1 = v2 + v3;
```

Almost all operators in C++ can be overloaded.

Two possibilities for implementation:

- Operators as **free functions**
- Operators as **member functions**

Operator Overloading - Member Function

Header file:

```cpp
class Vector2D {
public:
  Vector2D operator+(const Vector2D& rhs);
};
```

CPP file:

```cpp
Vector2D Vector2D::operator+(const Vector2D& rhs) {
  return Vector2D(x + rhs.x, y + rhs.y);
}
```

Usage:

```cpp
Vector2D v1, v2, v3;
v1 = v2 + v3;
```

Operator Overloading - Free Function

Header file:

```cpp
class Vector2D {
public:
  // Optional: allows the function to access private members
  //           does NOT declare a member operator!
  friend Vector2D operator+(const Vector2D& lhs,
                            const Vector2D& rhs);
};
```

CPP file:

```cpp
Vector2D operator+(const Vector2D& lhs, const Vector2D& rhs)
{
  return Vector2D(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

Usage:

```cpp
Vector2D v1, v2, v3;
v1 = v2 + v3;
```

Operator Overloading - Summary

See CPP-Reference for operators which are overloadable and advice on how to overload them: http://en.cppreference.com/w/cpp/language/operators

Free function overloads allow double dispatch:

```cpp
int i = 10;
Vector2D v = {1, 2};
Vector2D w = v + i; // first operand is Vector2D
Vector2D w = i + v; // first operand is int
```

Advice: Prefer overloading with free functions to overloading with member functions