Introduction to the C++ Programming Language

Day 2

Jonas Rylund Glesaaen
jonas@glesaaen.com

September 29th 2015

What will we learn?

- ~~Basic C++ syntax~~
- ~~Control structures~~
- Functions (today)
- Structs and classes (Wednesday and Thursday)
- Templates and STL (Thursday and Friday)
- Exceptions (Friday)

Today's topics

1. Scope
2. Functions
3. Libraries
4. Compiling and Linking
5. Debugging
6. Programming Practices
7. Recap

Scope

Variable Visibility

A scope is an area of visibility of a defined name

Scope #1

var a, b

Scope #2

var b, c

- a is not defined in scope #2
- c is not defined in scope #1
- the name b is not the same memory address in the two scopes

Variable Visibility

Scopes can also be nested

Scope #1

var a, b

Scope #2

var c, d

- a and b are both available in scope #2
- c and d are not in scope #1

Variable Visibility

One can also overshadow names in nested scopes

Scope #1

var a

Scope #2

var a

The a is not the same in the two scopes

a from Scope #1 is unavailable inside of Scope #2

Scope and memory management

A variable only takes up memory while it is ”in scope”

It is deleted when it goes ”out of scope”

Blocks ≈ Scope

Blocks define a new scope

```
{
  bool dancing = true;
  for (;;) {
    // ...

  }
}
```

Scope 1

Scope 2

```
{
  int baboons = 10;
  // ...
}
```

Scope 3

Functions

DRY, KISS, YAGNI, Occam, ...

Functions is a tool to divide one big problem into many small ones

DRY - Don't Repeat Yourself

KISS - Keep It Simple, ~~Stupid~~ Silly

If you copy-paste while programming, you are doing it WRONG

What is a function?

A function is a unit in your program that

- Takes input from the caller [optional]
- Does something
- Returns the result to the caller [optional]

The Function Signature

```
double integerPower(double x, int n) { ... }
```

The Function Signature

The name of the function
Similar to a variable name

The function body
(creates a scope)

```
double integerPower(double x, int n) { ... }
```

The return type
of the function

The input parameters of the function

Return types

Most types, including qualifiers, can be returned from a function but their meanings might not be that straight forward

To return something from a function, use the return keyword

Return types

Most types, including qualifiers, can be returned from a function but their meanings might not be that straight forward

To return something from a function, use the return keyword

```
int highFive()
{
  return 5;
}
```

Return types

reference / pointer

returning references and pointers can be very useful, just make sure the variables haven't gone out of scope

```
int & highFive()
{
  int five = 5;
  return five;
}
```

this will not work

Return types

void

void is the return type when you don't want to return anything

```
void sayHello()
{
  std::cout << "Hello!" << std::endl;
}
```

Return types

array

You can't return an array in the naive way, but you can return the underlying pointer, but there is the scope thing again

Return types

const

returning const prevents things like rvalue assignments

```
MyType function(int,double);

int main()
{
  // ...
  function(4,5.) = x;
}
```

Call-by-value vs Call-by-reference

Call-by-value is when you give your argument as a **non-reference** type

- Can only get feedback from the return value
- Function body can't manipulate input parameters
- Input parameters are copied in memory

Call-by-value vs Call-by-reference

```
double half(double input)
{
  return input/2;
}

int main()
{
  double var = 5;
  double half_of_var = half(var);
}
```

Call-by-value vs Call-by-reference

Call-by-reference is when you give your argument as a **reference** type

- The function can manipulate the arguments
- No unnecessary copying of variables
- A const reference is like a read-only argument
  {but remember that copying base types is basically free}

Call-by-value vs Call-by-reference

```
void half(double & input)
{
  input /= 2;
}

int main()
{
  double var = 5;
  half(var);
}
```

Arrays as input

```
double sum(const double arr[], int size) { ... }
```

Arrays are automatically called by reference

Can leave the array dimension unspecified

Multi-dimensional arrays

```
double print(const int array[][5][10]) { ... }
```

Can *only* leave the innermost size unspecified

Recursive functions

Functions can of course call other functions, including themselves

A function calling itself is called recursive

Recursive functions is like an advanced loop

Recursive functions - example

```
unsigned factorial(unsigned n)
{
  if (n < 2) {
    return 1;
  }

  return n * factorial(n-1);
}
```

Recursive functions - example

```
unsigned factorial(unsigned n)
{
  if (n < 2) {
    return 1;
  }

  return n * factorial(n-1);
}
```

Alternate return path

Static variables

Static variables aren't deleted when they go out of scope, but when the program exits

Their initialisation only happens the first time the block is run

```
int countCalls()
{
  static int times_called = 0;
  ++times_called;
  return times_called;
}
```

Definition vs Declaration

Every function must be declared before it is called

But that doesn't mean you have to stick all your functions at the top of your C++ file

Definition vs Declaration

Function declaration:

Argument name optional

```
void function(int, const double&);
```

Declare that the function exist, and define its signature, but not what it does

Function definition:

```
void function(int x, const double & d)
{
  // ...
}
```

Define what the function does when called

Definition vs Declaration - example

```
double square(double);        ← Declaration

int main()
{
  double three = 3.;
  auto nine = square(three);
}

double square(double d)       ← Definition
{
  return d*d;
}
```

Function overloading

Possible to create multiple functions with the same name but different argument list

This is called overloading

```
int sum(int, int);
int sum(int, int, int);
double sum(double,double);
double sum(double,double,double);
```

Helps create a consistent interface

Function overloading - example

```
double norm(double a, double b)
{
  return a*a + b*b;
}

double norm(double a, double b, double c)
{
  return a*a + b*b + c*c;
}
```

Function overloading - example

```
double norm(double a, double b)
{
  return a*a + b*b;
}

double norm(double a, double b, double c)
{
  return norm(a,b) + c*c;
}
```

Variadic functions

Can also make a function with an undefined number of arguments

These are called variadic functions

But variadic functions in C aren't very pretty, so we will wait until we talk about templates

Variadic functions - sneak peek

```
template <typename... Arguments>
double sum(double val1, Arguments... values)
{
  return val1 + sum(values...);
}

template<>
double sum(double d)
{
  return d;
}

sum(-5.2, 12.5);
sum(1, 4.5, 2.6, 9.4);
```

Variadic functions - sneak peek

```
template <typename... Arguments>
double average(Arguments... values)
{
  auto number_of_arguments = sizeof...(values);
  return sum(values...)/number_of_arguments;
}

average(5, 9, 0);
average(1, 5, 1, 8, 3, 2, 2, 9);
```


{C++11}

Default argument values

A variant of function overloading is giving the arguments default values

This should be done in the function declaration

```
double integerPower(double, int = 2);
```

Lambda functions

```
[...](double x, int n){ ... }
```


{C++11}

Lambda functions

The capture list
Gives the object more internal variables available in scope
Allows for closures in C++

```
[...](double x, int n){ ... }
```

The argument list

The function body

{C++11}

Lambda functions

The lambda function is a function literal, so it must be assigned to a variable

```
auto sum = [](int a, int b) { return a + b; };

int summed_value = sum(6,9);
```

The return type is inferred from the code

{C++11}

Lambda functions

The argument types can also be inferred from context

```
auto sum = [](auto a, auto b) { return a + b; };

int summed_value = sum(6, 9);
double summed_floats = sum(5.6, 9.2);
```

This is a pet example of templates at work

Live Example

Libraries

Using other people's work

There is no reason to reinvent the wheel
every time you write a program

You could of course write your own `cos` function,
but why would you do that?

{you would probably never get it as efficient and safe either}

The #include statement

The include command copies the content of the specified file into the current file

The #include statement

Before preprocessing

header.hpp

```
int sum(int, int);
int sum(int,int,int);
double sum(double,double);

int highFive();
void sayHello();
```

main.cpp

```
#include "header.hpp"

int main()
{
  auto total = sum(highFive(), 9);

  // ...

  for (auto i = 0; i < 10; ++i) {
    sayHello();
  }
}
```

After preprocessing

main.cpp

```
int sum(int, int);
int sum(int,int,int);
double sum(double,double);

int highFive();
void sayHello();

int main()
{
  auto total = sum(highFive(), 9);

  // ...

  for (auto i = 0; i < 10; ++i) {
    sayHello();
  }
}
```

Header files and source files

For larger projects one normally organises

declarations $\longrightarrow$ header files

definitions $\longrightarrow$ source files

Header guards - Motivation

Declaring a function more than once is illegal,
but with all the includes it is hard to keep track

Error: Nested includes

sum_functions.hpp

```cpp
int sum(int, int);
int sum(int,int,int);
double sum(double,double);
```

utilities.hpp

```cpp
#include "sum_functions.hpp"

int highFive();
void sayHello();
```

main.cpp

```cpp
#include "sum_functions.hpp"
#include "utilities.hpp"

int main()
{
  auto total = sum(highFive(), 9);

  // ...

  for (auto i = 0; i < 10; ++i) {
    sayHello();
  }
}
```

Error: Circular includes

sum_functions.hpp

```cpp
#include "utilities.hpp"

int sum(int, int);
int sum(int,int,int);
double sum(double,double);
```

utilities.hpp

```cpp
#include "sum_functions.hpp"

int highFive();
void sayHello();
```

Header guards

These issues can be resolved using header guards

Old Style

sum_functions.hpp

```
#ifndef SUM_FUNCTIONS_H
#define SUM_FUNCTIONS_H

int sum(int, int);
int sum(int,int,int);
double sum(double,double);

#endif /* SUM_FUNCTIONS_H */
```

Header guards

These issues can be resolved using header guards

New Style

sum_functions.hpp

```
#pragma once

int sum(int, int);
int sum(int,int,int);
double sum(double,double);
```

Header guards

The old style:

Cons: Need a unique name for all headers

The new style:

Pros: Easy to use, no extra names

Cons: Compiler support not guaranteed
{but the common compilers support it}

#include"" vs #include<>

#include""

Looks for the header files relative to the location of the source file

#include<>

Looks for the header files in the system include folders, e.g. /usr/include, /usr/local/include, ...

Can add more include folders with the -I compiler flags

Namespaces

You can further organise your code by putting all your functions and classes into namespaces

Namespaces scope the names you create when declaring functions and classes

We have already encountered the std namespace introduced by the standard libraries

Namespaces

```
namespace Summers {

  int sum(int,int);

  namespace Printers {

    void prettyPrintSum(int,int);
  }
}

int main()
{
  auto val = Summers::sum(5,1);

  Summers::Printers::prettyPrintSum(10,5);
}
```

Namespaces

Without namespaces you would have to check if any names in external libraries clashed with your every time you included a library

C libraries often have absurdly long method names

```
gsl_vector * vec = gsl_vector_alloc(5);
gsl_multiroot_fsolver * s = ↩
    → gsl_multiroot_fsolver_alloc();
```

Namespaces

You can include a name from a namespace with the using keyword

```
using std::cout, std::endl;
```

Or an entire namespace (but never do this!)

```
using namespace Summers;
```

inline functions

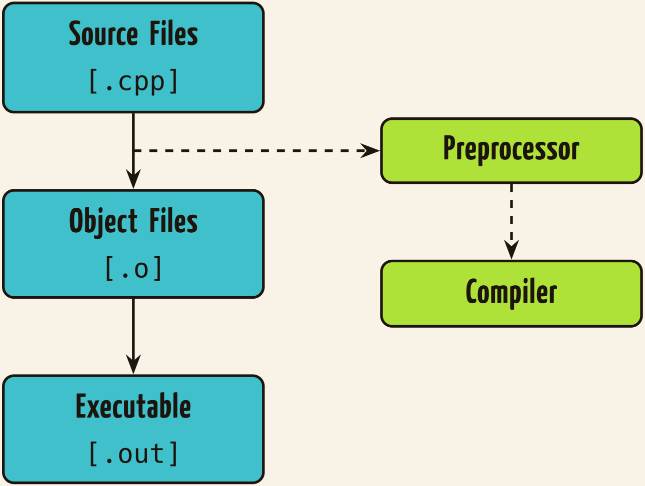
To define a function in the header you have to declare it inline

```
inline int sum(int a, int b)
{
  return a + b;
}
```

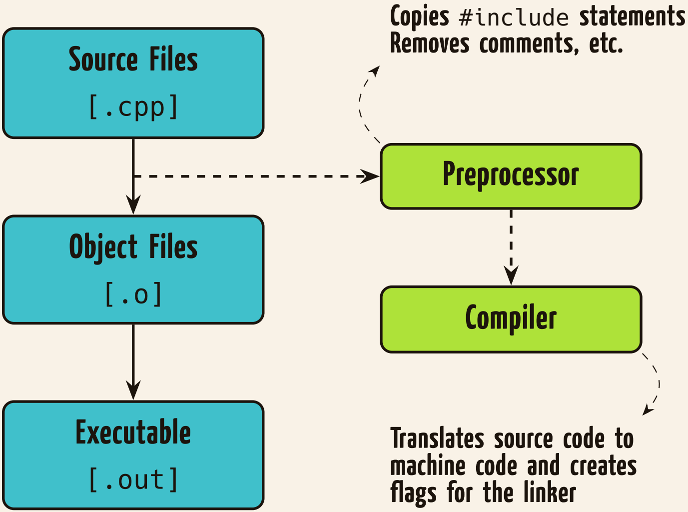
If not the linker will complain about duplicate definitions

Compiling and Linking

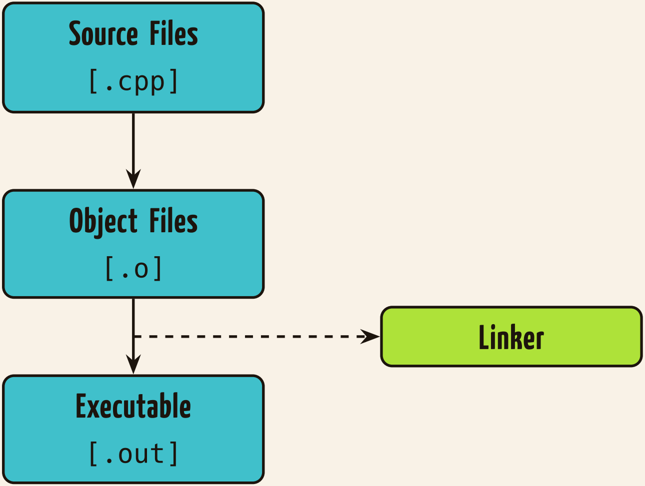
Steps of building a program



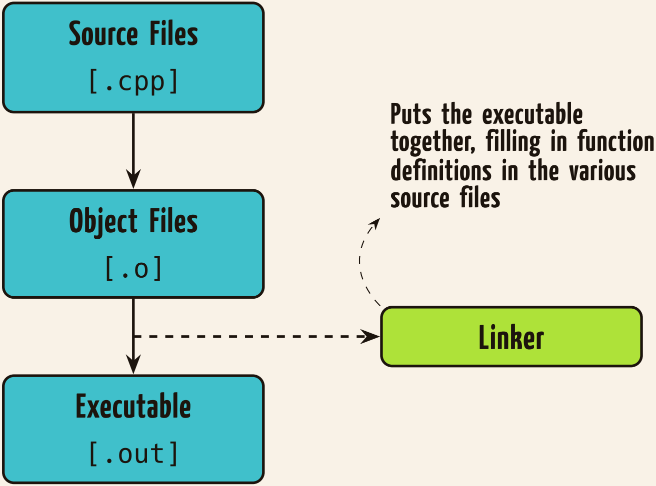
Steps of building a program



Steps of building a program



Steps of building a program



Steps of building a program

All of the above steps happen if you write

```
g++ -o program source1.cpp source2.cpp
```

Can use the `-c` flag to compile only

```
g++ -o source1.o -c source1.cpp      ← compiling
g++ -o source2.o -c source2.cpp
g++ -o program source1.o source2.o   ← linking
```

Quick introduction to makefiles

We use a build manager to automatise this process

Examples:
make, cmake, QMake, various IDE's, ...

Quick introduction to makefiles

A makefile is simply a list of ingredients and results for the various stages of compiling

```
result : ingredients
    method to obtain
```

Quick introduction to makefiles

```
program : source1.o source2.o
  g++ -o program source1.o source2.o

source1.o : source1.cpp
  g++ -o source1.o -c source1.cpp

source2.o : source2.cpp
  g++ -o source2.o -c source2.cpp
```

Quick introduction to makefiles

```
SRCS := $(wildcard *.cpp)
OBJS := $(SRCS:%.cpp=obj/%.o)

program : $(OBJS)
  g++ $(OBJS) -o $@

obj/%.o : %.cpp | obj
  g++ -c $< -o $@

obj :
  @mkdir -p obj
```

Compiling vs Linking

When compiling the code is converted to machine code, but function definitions might still be missing

These "holes" are filled when the program is linked and an executable is created

Various types of errors

There are in essence 3 types of errors

1. Compile errors
2. Linking errors
3. Runtime errors

Various types of errors

There are in essence 3 types of errors

1. Compile errors
   - Syntactic errors
   - Template lookup errors
   - Type conversion errors
2. Linking errors
3. Runtime errors

Normally easy to find with a good coding environment, comes with practice

Various types of errors

There are in essence 3 types of errors

1. Compile errors
2. Linking errors
   - Multiple definitions
   - Function definition not found
3. Runtime errors

A bit harder to find but in essence easy, also might require some practice

Various types of errors

There are in essence 3 types of errors

1. Compile errors
2. Linking errors
3. Runtime errors
   - Unexpected behaviour
   - Memory issues
   - Infinite loops

This is the real killer, need more advanced tools

Debugging

Runtime errors

```
int sumArray(int array[], unsigned size)
{
  unsigned index = 0;
  int result = array[index];

  do {
    ++index;
    result += array[index];
  } while (index < size);

  return result;
}
```

Runtime errors

```
double volumeOfCone(double r, double h)
{
  static const double pi = 4*std::atan(1);
  return static_cast<double>(1/3)*pi*r*r*h;
}
```

Runtime errors

```
void print(int ** array, unsigned size)
{
  for (int i = 0; i < size; ++i) {
    for (int j = 0; i < size; ++i)
      std::cout << array[i][j] << " ";

    std::cout << std::endl;
  }
}
```

Runtime errors

```
// Allocate memory
// Return: whether allocation was successful
bool allocate(int *, unsigned size);

void initialise(int * array, unsigned size, bool set_to_zero)
{
  //Only set to zero if allocation was successful
  if (set_to_zero && allocate(array,size)) {
    for (auto i = 0; i < size; ++i)
      array[i] = 0;
  }
}
```

Runtime errors

```
unsigned factorial(unsigned n)
{
  unsigned result = 1;
  while (n > 1) {
    result *= --n;
  }

  return result;
}
```

Runtime errors

```
unsigned f(unsigned n){return !n?1:--n*f(n);}
```

"printf debugging"

Print the current state of the variables

```
int sumArray(int array[], unsigned size)
{
  unsigned index = 0;
  int result = array[index];

  do {
    std::cout << index << std::endl;        ←——— Here?
    ++index;
    result += array[index];
  } while (index < size);

  return result;
}
```

”printf debugging”

Print the current state of the variables

```
int sumArray(int array[], unsigned size)
{
  unsigned index = 0;
  int result = array[index];

  do {
    ++index;
    std::cout << index << std::endl;        Here?
    result += array[index];
  } while (index < size);

  return result;
}
```

"printf debugging"

Advantage:

- Requires no additional knowledge

Disadvantage:

- A very static way of debugging
- Have to recompile every time
- If you realise something mid debugging, there is no way out
- Important information can be lost in output

Assert statements

An assert is a runtime test that terminate the program if it fails

Enabled by including the `<cassert>` library

Can be disabled using the `NDEBUG` preprocessor flag

`#define NDEBUG` or `g++ source.cpp -DNDEBUG`

Assert statements

```
//Uncomment to disable
//#define NDEBUG
#include<cassert>

int sumArray(int array[], unsigned size)
{
  unsigned index = 0;
  int result = array[index];

  do {
    ++index;
    assert(index < size);
    result += array[index];
  } while (index < size);

  return result;
}
```

Check before accessing

Assert statements

This is safer than printing as we can terminate the program, but it is still not very dynamic

To achieve a more dynamic debugging process, we will introduce dedicated debuggers

Live Example

Programming Practices

Good Programming Practices

- Don't Repeat Yourself (stay DRY)
- Write functions with a single responsibility
- Make use of namespaces to keep the global scope clean
- **Never** include the std namespace
- Use debuggers to find runtime errors

Useless refactoring

This is utterly useless...

```
int runProgram();

int main()
{
  return runProgram();
}

int runProgram()
{
  //Everything that was in main
}
```

Recap

Recap Day 2

- Variable visibility and lifetime is governed by its scope
- A function is a unit that does something
- Variables can be passed by value or reference
- static variables outlive their scope
- One can separate definition and declaration

Recap Day 2

- Function declarations can be #include 'd
- There are two steps to building a program: compiling and linking
- We have three types of errors: compiling, linking and runtime
- Dedicated debuggers can be used to find runtime errors