Function basics

Michael Nowak

Texas A&M University

Acknowledgement: Lecture slides based on those created by Bjarne Stroustrup for use with his textbook

Notes

Overview

Notes

Overview

Notes

Function refresher

- The simple example of a function from math class might look like

$$f(x) = x^2$$

  - This names f as a function that takes a number, which we give the name x, and it has the value of that number squared
  - When the function is written, the parameter, x, has no value
  - The value of x is specified when we use the function
  - $f(3)$ specifies 3 as the argument to the function f and x takes on the value of 3 during our calculations
  - We easily calculate $f(3) = 9$

Notes

Function refresher

- We can have functions with more than one variable, such as

$$f(x, y) = x^2 + y^2$$

  - We provide two values as arguments when we'd like to call this function; they are used for x and y
  - $f(2, 3) = 4 + 9 = 13$, the x-value is 2 and the y-value is 3
  - The order of the arguments matches the order of the parameters

Notes

Overview

Notes

Function declarations

- We can `declare` a function by writing a `declarator` of the form `f(args)`, where `f` is the `name` being introduced and `args` is a list of zero or more `parameters`, for example:

```
double mult(double, double);
```

  - The `base type` specifies the `return type` of the `function`
    - We can specify `void` as a "pseudo return type" for functions that do not return a value
  - The function's parameters are specified in a comma-separated list enclosed in parentheses
    - A function's parameter list can be left empty but can not be omitted (still need the parentheses)

Notes

Overview

Notes

Function definitions

- A `declaration` that fully specifies the entity being `declared` is called a `definition`
- We write a function `definition` by writing a function `declaration` with a statement block, i.e., a `function body`, that specifies a sequence of statements that the function will perform when called

```
double mult(double x, double y)
{
    return x*y;
}
```

Notes

Overview

Notes

Declarations and definitions

- In your C++ programs, you can't define something twice
  - A `definition` says what something is
- However, you can declare something multiple times
  - A `declaration` says how something can be used

Notes

Overview

Notes

Why both declarations and definitions?

- To refer to something, we need (only) its `declaration`
  - A declaration introduces a name to the compiler and how that name can be used
- Often we want the `definition` "elsewhere"
  - Later in a file
  - In another file
- `Declaration` are used to specify interfaces code
  - Declarations are frequently introduced into a program through "headers"
  - A "header" is a file containing declarations, giving you access to functions, types, etc. for use in your programs
- `Definition`s can be provided in different translation units
  - The object code containing those definitions will eventually be linked together with other object code by the `linker` in formulation of an `executable`

Notes

Overview

Notes

Write function declarations in a header file

- For your functions, write the declarations in some header file (`descriptive_name.h`)
- What happens if `descriptive_name.h` `#include`s `other_descriptive_name.h` which then `#include`s `descriptive_name.h` in the same translation unit?
  - We really want our header files to be included exactly once per translation unit
- We thus use `header guard`s to ensure that if a file has already been included, we do not include it again; `descriptive_name.h` would have the following structure:

```
#ifndef DESCRIPTIVE_NAME_H
#define DESCRIPTIVE_NAME_H
/* function declarations */
#endif
```

Notes

Overview

Notes

Write function definitions in a source file

- After writing the header file for `descriptive_name.h`, we would write a corresponding source file `descriptive_name.cpp` which would define the functions previously declared
- We conventionally `#include` the corresponding header `descriptive_name.h` in `descriptive_name.cpp` to ensure that each declaration has a definition with the same `function signature` and `return type`
  - If things didn't match up, we'd likely see a compiler or linker error
- `descriptive_name.cpp` will have the following structure:

```
#include "descriptive_name.h"

/* function definitions */
```

Notes

Overview

Notes

Using functions defined in other source files

- We can use our functions in say `main.cpp` by `#include "descriptive_name.h"`
- Just make sure that when you compile, you remember to compile both `main.cpp` and `descriptive_name.cpp`:

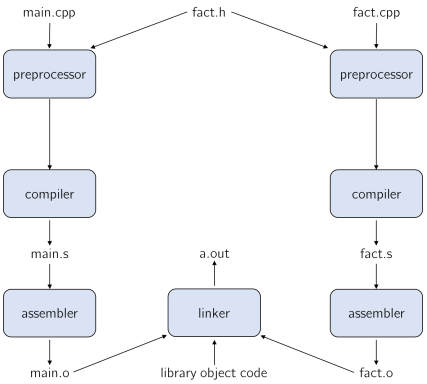
```
g++ -std=c++14 main.cpp descriptive_name.cpp
```

Notes

Overview

Notes

Separate compilation



Notes

Overview

Notes

Executing a function

- We execute a function through the `call operator`, which is a pair of parentheses
- The `call operator` takes an expression that is a function
- We provide a comma-separated list of `arguments` inside the parentheses
- The `arguments` are used to initialize the function's `parameters`
- The type of a call expression is the return type of the function

Notes

Overview

Notes

Writing a function

- We would like to write a function that calculates the `factorial` of a number
- The factorial function in mathematics, $n!$ means to multiply each integer together from 1 up to n, that is:

$$n! = \prod_{i=1}^{n} i$$

which can be rewritten informally as

$$n! = n \times n-1 \times \ldots \times 2 \times 1$$

  - For example, 3! is thus calculated as:
    $3! = 3 \times 2 \times 1$

Notes

Writing a function

- We can write an iterative algorithm in C++ to solve for the factorial of some value `int val` as:

```
int res = 1;
while(val > 1) {
    res *= val;
    val -= 1;
}
```

Notes

Writing a function

- We can `encapsulate` our iterative solution in a function
- We could define such a `function` as follows:

```
int fact(int val)
{
    int res = 1;
    while(val > 1) {
        res *= val;
        val -= 1;
    }
    return res;
}
```

  - This `function` is named `fact`
  - It takes an `int parameter` identified by `val`
  - The `function body` calculates the factorial of the argument used to initialize the parameter `val`
  - The `return statement` ends the execution of `fact`, returning `res` to the caller

Notes

Overview

Notes

Calling a function

```
int main()
{
    int j = fact(5);
    /* do something */
    return 0;
}
```

- As long as our function has been `declared` before its name is used, the source file containing the function call will compile to object code
- As long as our function has been `defined` in some source file being compiled, the linker will be able to join together the object code into an executable

Notes

Calling a function

```
int main()
{
    int j = fact(5);
    /* do something */
    return 0;
}
```

- The function call to `fact()` does two things
- It initializes the function's parameters from the corresponding arguments
  - An `int` variable named `val` is created
  - That variable is initialized by the argument in the function call, `5`
- It transfers control to that function
  - Execution of the calling function is suspended and execution of the called function begins
  - Execution of the function ends when the thread of execution passes over the `return statement`
  - At this point, a value (if applicable) is returned as control is transfered back to the calling function

Notes

Overview

Notes

Why functions?

- Chop a program into manageable pieces
  - "divide and conquer"
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug

Notes

Overview

Notes

References

- Lewis, M. C. (2015). *Introduction to the art of programming using Scala*. CRC Press.
- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.

Notes

Notes

Notes