First Program

Fundamentals

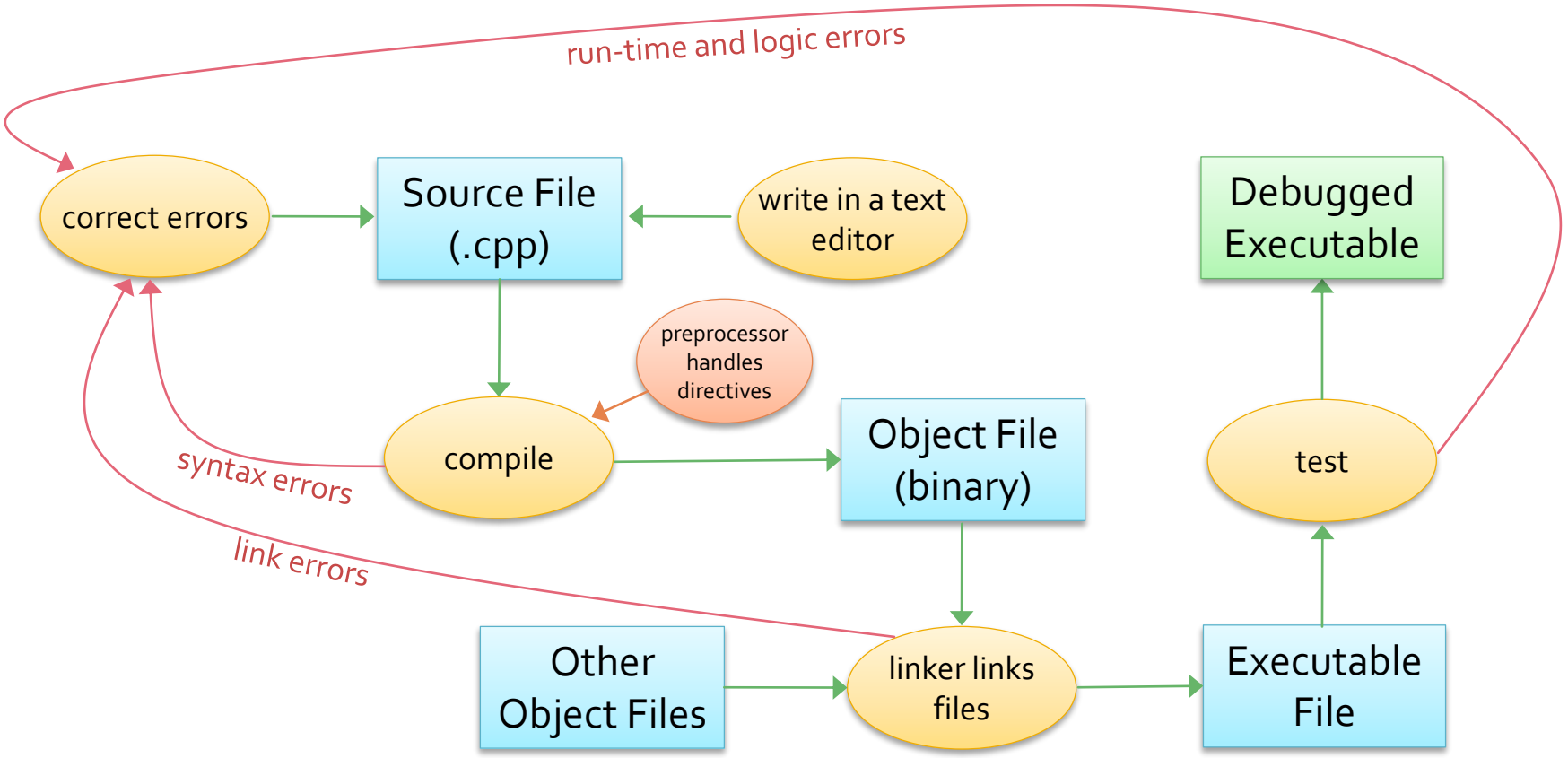
Fundamentals

- Compilation and Execution
- Simplifying Fractions
  - Loops
  - If Statements
- Variables
- Operations
- Using Functions
- Errors

Running a C++ Program

- C++ programs consist of a series of instructions using the C++ syntax
  - Usually broken down into a number of functions
    - And often in different files
- A C++ program needs to be translated into something that a computer can understand
  - This process is called *compiling*
  - A compiler translates high-level language code into machine code

Processing a C++ Project



Command Line Compilation

- Most OSs include C++ compilers and allow a program to be compiled and run
- To compile and run a program in Linux follow these steps
  - *Write* the program using a *text editor*
  - *Compile* the program using the *g++* compiler
  - *Run* the program created by compilation by typing its name

Linux Compilation Example

- Assume there is a C++ file named *test.cpp*
- Open a *Terminal* window in Linux
- At the command prompt type
  - `g++ –o test test.cpp`
    - The –o *switch* (or *flag*) tells the compiler to name the object file that is being created *test*
    - *test.cpp* is the name of the file to be compiled
- To run the newly created file type
  - `./test`

Errors

- In reality if we've just finished writing our first C++ file it will fail to compile
  - Because it contains compilation errors
  - That is, mistakes in C++ syntax
- The compiler will print a list of errors
  - That we can then fix
- More on this later …

IDEs

- An alternative to using the Terminal to compile programs is to use an IDE
  - Integrated Development Environment
- An IDE combines a text editor, a compiler, a debugger and (many) other tools
  - There are many IDEs
    - Visual Studio (Windows)
    - Code::Blocks (cross-platform)
    - Xcode (Mac OS)
    - Eclipse

Simplifying Fractions

Simplifying Fractions

- Assume we want to simplify fractions
  - 4/16 =
    - 1/4
  - 105/135 =
    - 7/9
- To simplify a fraction
  - Find the greatest common divisor (*gcd*) of the numerator and denominator
    - Using Euclid's algorithm
  - Divide numerator and denominator by the *gcd*

Requirements

- "*We want to simplify fractions*" is pretty vague
- Input
  - Prompt the uset to enter two integers
    - That represent a numerator and denominator
- Output
  - Print the simplified result in the form
    - numerator / denominator
- Process
  - Calculate the *gcd* of the numerator and denominator
  - Print numerator ÷ *gcd*, a "/", and denominator ÷ *gcd*

Solution

```
#include <iostream>

int main()
{
	int numerator = 0;
	int denominator = 1;

	// Get user input
	std::cout << "Enter the numerator: ";
	std::cin >> numerator;
	std::cout << "Enter the denominator: ";
	std::cin >> denominator;

	// Compute GCD of numerator and denominator using Euclid's algorithm
	int a = numerator;
	int b = denominator;
	while (a != b) {
		if (a > b) {
			a = a - b;
		}
		else {
			b = b - a;
		}
	}

	// Print result
	std::cout << std::endl << numerator/a << "/" << denominator/a << std::endl;
	return 0;
}
```

Discussion

- The program has a number of sections
- Introductory comments
  - That describe the program
    - Omitted from previous slide for space
- An include statement
  - That imports library functions
- A *main* function
  - That actually does the work
  - The main function includes an implementation of Euclid's algorithm (the *while* statement)

Comments

- Comments document programs
  - They explain what a program does
  - And how it works
  - They are ignored by the compiler
  - A comment starts with /* and ends with */
    - And may include more than one line
- Writing comments is an important part of writing a maintainable program
  - And you will lose marks if you don't write comments in assignments …

```
/*
 Program to simplify a fraction
 Author: John Edgar
 Date: September 2018
*/
```

Omitted from the sample code for space

More Comments

- There are two ways of writing comments
  - Starting with /* and ending with */
    - /* can also be used for comments that start on one line
    - and end on another */
  - Or single line comments that start with
    - // and continue to the end of the line
- Either method of commenting is acceptable
  - // style is particularly useful for short comments

#include iostream

- The include statement is a *pre-processor directive*
  - The pre-processor operates before the compiler
- #include tells the pre-processor to include the contents of *iostream* in our program
  - Opens the *iostream* file
  - Copies its contents
  - Pastes it, replacing the include directive

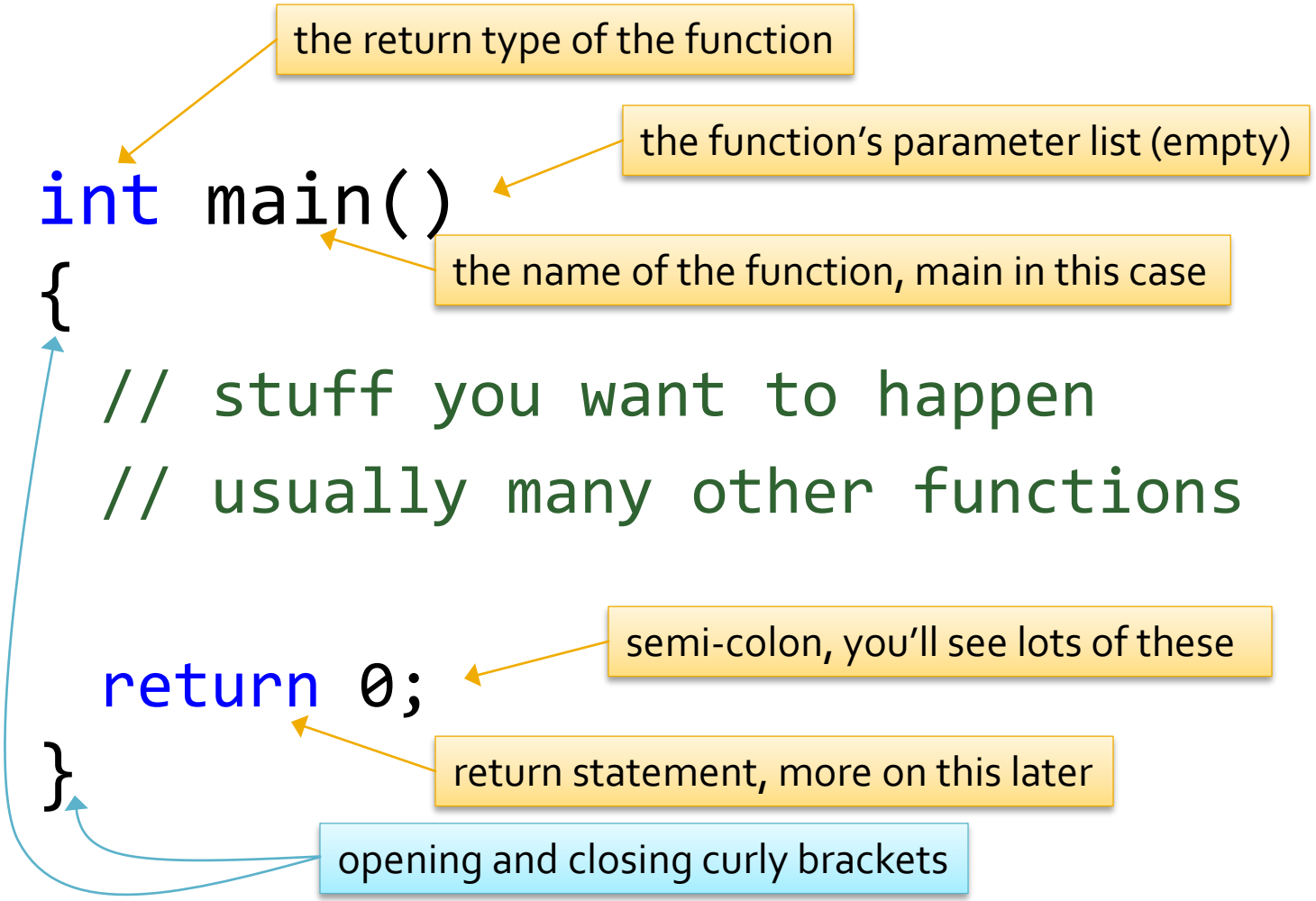
What's in iostream?

- The *iostream* file contains information that our program need
  - For this program it is access to *cin* and *cout*
    - Which are used for standard input and output
      - Usually input from the keyboard and output to the monitor
  - *iostream* contains input and output functions
    - Input and output streams
- Programs will often need to use functions and classes from library files

Main

- C++ programs have one main function that executes when the program is run
  - This function has to be called *main*
  - We will write (lots of) other functions that we will give different names to
- The main function always has the same general form

Skeleton of Main



Program Flow

- A program is a sequence of instructions executed in order
  - The order in which instructions are given is important
    - And cannot be arbitrarily changed with the expectation that the same results will occur
- When *main* executes it starts with the first line and ends when it reaches the *return* statement
  - Any instructions after the return statement will not be executed
- Various control statements may change program flow to allow loops or branching

Semi-Colons

- Semi-colons show the end of C++ instructions
  - Most C++ statements are written on one line
  - However, the newline character does not indicate the end of a statement
- Don't forget semi-colons
  - As the program won't compile without them
  - Generally, you need a semi-colon after each statement that does not start or end with { or }

Variables

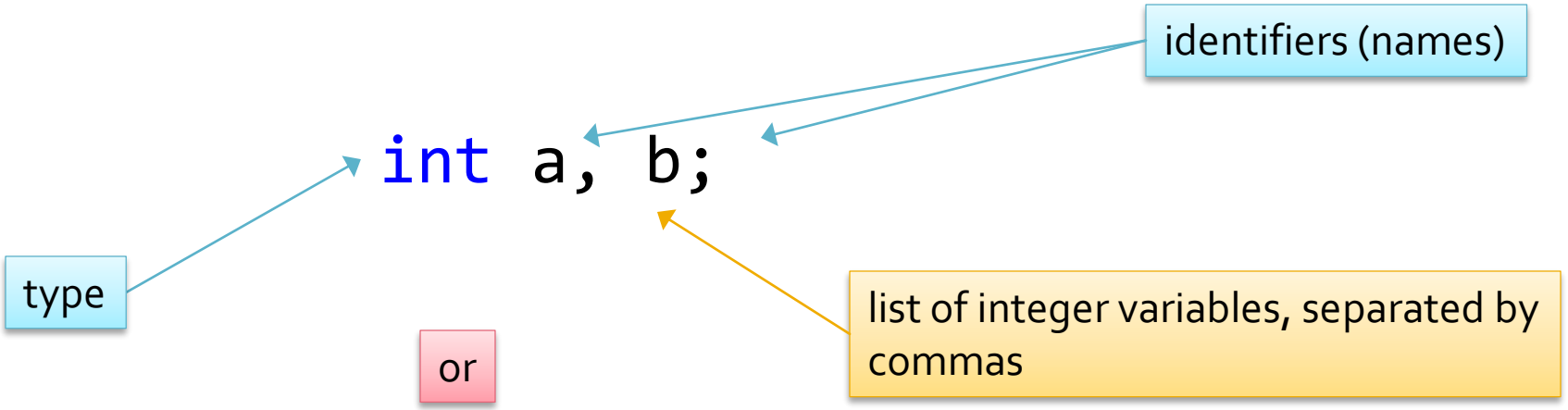
- The simplify program uses four variables
  - Variables are used to store data
  - Variables can be initialized or changed using the *assignment operator*, =

```
int numerator = 0;
int denominator = 1;
```

```
int a = numerator;
int b = denominator;
```

- The variables in simplify all have the same type
  - The type specifies what kind of data is stored
    - These variables are type *int* - used to store whole numbers
  - A variable's type is specified when only it is declared
    - Thereafter variables are referred to just by name without noting the type

Declaring Variables




```
int a = 213;
```

declaration and initialization

Note that it is important to give variables meaningful names; I used *a* and *b* because Euclid's algorithm finds the greatest common denominator of two arbitrary integers

Operations on Data

- Simplify includes operations on integer data
  - Subtraction and division
  - Assignment
- An operation requires an *operator* and at least one *operand*
  - a = a – b;

assignment is a *binary* operation since there are two operands, one on the left and one on the right

left operand | operator | right operand? | the right operand is the result of a-b

note that this one statement contains two binary operations with a distinct order

While Loops

- Loops are constructs that perform repetition
  - That is they repeat code inside the body of the loop until some condition is met
    - If the condition cannot be met the loop will continue running forever (an *infinite loop*)
- In the simplify program the loop continues to run as long as *a* and *b* have different values
  - The code that is repeated is contained within { }s that come after the condition

While Loop Anatomy

```
while(a != b)
{
    if(a > b){
        a = a-b;
    }else{
        b = b-a;
    }
}
```

keyword (while) and condition

the condition (a is not equal to b) evaluates to *true* or *false*

the body of the loop is repeatedly executed until the condition is false

the condition compares values stored in the integer variables

While General Form

```
while(condition)
{
    //loop body
}
```

the condition is a comparison that evaluates to true or false

conditions can be more complex and may not involve comparisons

the loop body consists of the statements that are to be repeated

the condition is tested before each iteration of the loop

If the condition is true the statements in the body of the loop are executed

if the condition is false the program jumps to the statement after the loop body's closing bracket

If Statements

- If statements are constructs that allow decisions to be made
  - One set of statements is performed if the condition is true
  - Another set of statements is performed if the condition is false
- Unlike loops, if statements do not perform repetition
  - One set of statements is performed and the program continues from the end of the if statement

Conditions

- Both while loops and if statements use conditions
  - A condition is a *Boolean expression* that evaluates to *true* or *false*
- Boolean expressions compare two values with a comparison operator
  - One of <, <=, >, >=, == (equal to), != (not equal to)
  - More complex conditions are possible and these will be discussed later in the course

If Statement Anatomy

```
if(a > b)
{
    a = a-b;
}else{
    b = b-a;
}
```

keyword (if) and condition

the condition (a > b) is true if a is greater than b and false otherwise

executed if a is greater than b

keyword else, i.e. otherwise …

executed if a is not greater than b

if statements are not required to have an else clause

Printing Data

- The last line of the program prints the result using *cout*

```
std::cout << std::endl << numerator/a << "/" << denominator/a << std::endl;
```

  - The << operator is used to output data of each type
    - << is the insertion operator
  - Text is written as a double-quoted *string*, e.g. "/"
  - The *std::endl* constant prints a newline character
- A *string* is a *sequence* of characters
  - Strings can be of any length
  - And must be enclosed in " "s

Format – Colour

- A text editor designed for use with C++ colours words to aid readers
  - Different text editors use different colours
  - The Linux text editor uses different colours from those presented in this presentation
- It is common to colour the following
  - Keywords (blue in presentation)
  - Strings, i.e. words enclosed by ""s (red)
  - Comments (green)

Format – Indentation

- Indentation is used to indicate flow of control
  - Everything in the main function's brackets is indented underneath the word main
  - The body of the while loop is farther indented under the while statement
- Indentation helps the reader, not the compiler
  - Even though indentation is ignored by the compiler it is still very important
  - Our goal is not just to produce programs that work but to produce programs that are easy to maintain
    - i.e. that are easy for people to understand

Colours and Indentation

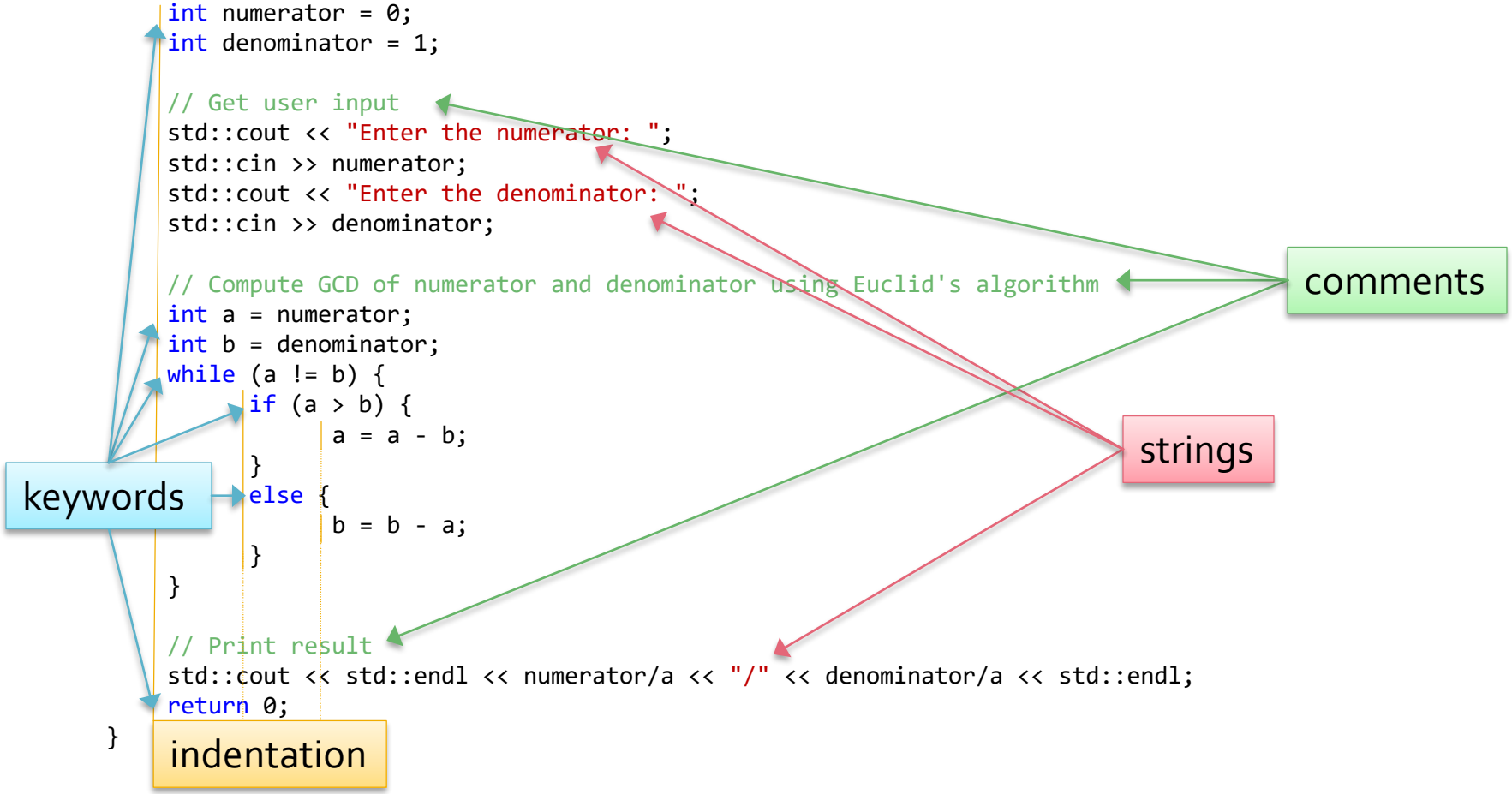
```
#include <iostream>

int main()
{
    int numerator = 0;
    int denominator = 1;

    // Get user input
    std::cout << "Enter the numerator: ";
    std::cin >> numerator;
    std::cout << "Enter the denominator: ";
    std::cin >> denominator;

    // Compute GCD of numerator and denominator using Euclid's algorithm
    int a = numerator;
    int b = denominator;
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }

    // Print result
    std::cout << std::endl << numerator/a << "/" << denominator/a << std::endl;
    return 0;
}
```



Simplify – the Bad Version

And here is the same program with no comments, colours or sensible formatting

```
#include <iostream>
int main(){int numerator = 0;int denominator = 1;std::cout << "Enter the numerator: ";std::cin >>
numerator;std::cout << "Enter the denominator: ";std::cin >> denominator; int a = numerator;int b
= denominator; while (a != b) {if (a > b) { a = a - b; }else { b = b - a; }}std::cout << std::endl
<< numerator / a << "/" << denominator / a << std::endl;return 0; }
```

This program does compile and run correctly but is very difficult to read

Important note – if the above program was submitted as a solution to an assignment it would lose a lot of marks, even though it works

There is a lot more to writing a *good* program than producing something that works

Why isn't this program very good?

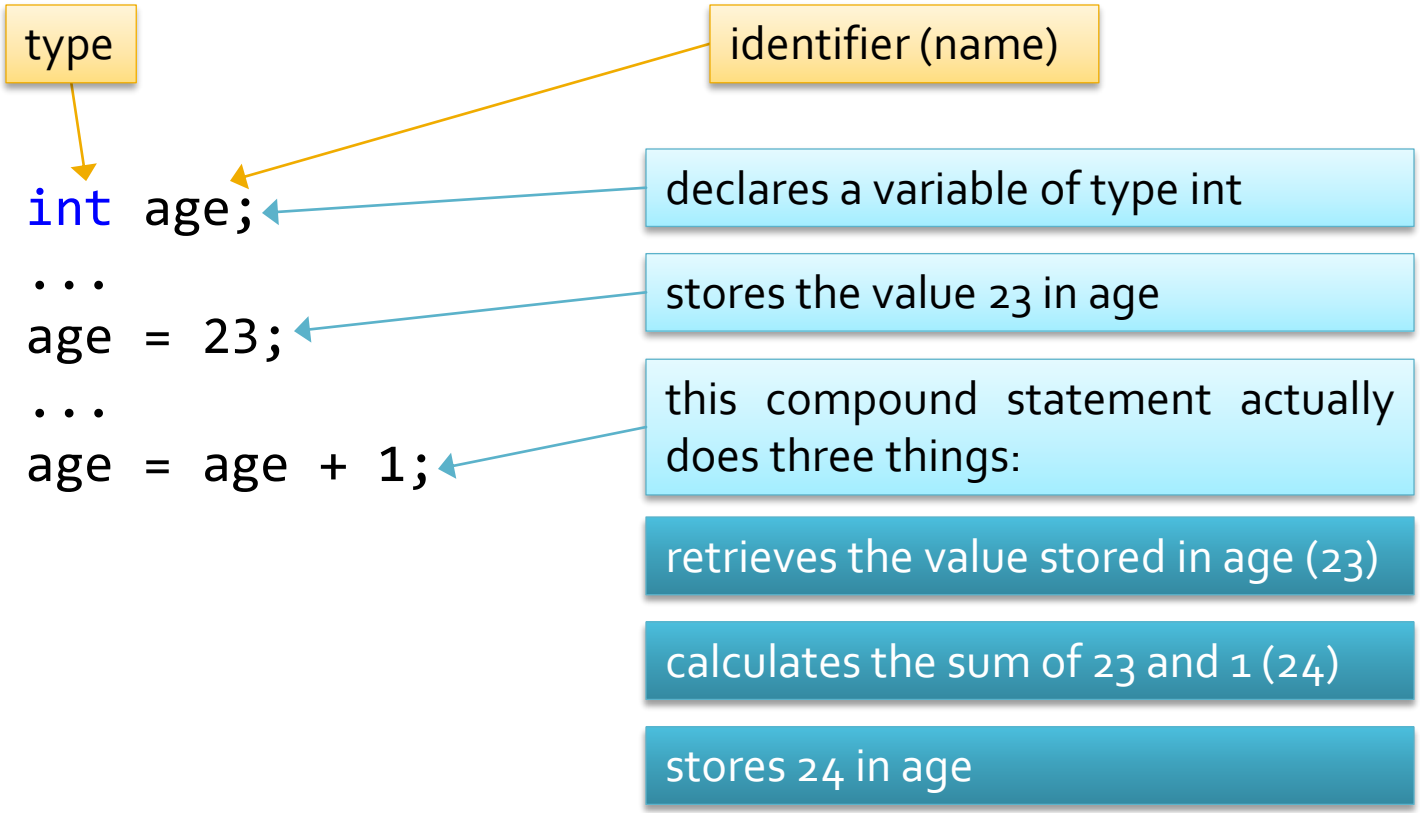
- Let's critique the simplify program
  - ...
- Major criticisms
  - Very limited application
    - But beware bloat
  - No GUI
  - Bare bones input and output
  - Lack of error handling
  - GCD calculation should be a function

Variables

What is a Variable?

- A variable stores data required for a program
  - It is a sequence of bytes
  - The bit pattern stored in those bytes represents the data to be stored
    - Variables cannot be empty
- The data stored in a variable is referred to by the name of the variable
  - And can be operated on
    - Modified or
    - Replaced

Declaring Variables



Rules for Declaring Variables

- A variable is declared only once
  - Variables must be declared before they are used
- Variables may be declared in a list
  - Each variable is separated by a comma
- Variables may be assigned a value when declared
  - Known as *initialization*

Variable Lists and Initialization

```
int days = 213;

int hours, minutes, seconds;
```

declaration and initialization

list of integer variables, separated by commas

Duplicating Names

- Declaring a variable reserves space for the variable's data in main memory
  - A variable can only be declared once
  - Repeating a declaration is an attempt to declare a new variable with the same name
  - Which is illegal
- Variables with the same name are allowed as long as their *scope* is different

Initializing Variables

- Variables should be *initialized* before first being used
  - That is, given sensible starting values
  - This can be done at the same time as they are *declared*
- Forgetting to initialize variables can result in unexpected and unwanted results
  - Note that a memory location can never be empty
    - Since it is a sequence of bits

Reserved Words

- Variables in simplify were declared as type *int*, and *int* is a reserved word or keyword
  - Part of the C++ language
- There are other keywords, many of which we will encounter in this course
  - A text editor designed for writing C++ programs will usually indicate keywords by colour
- Keywords cannot be used as variable names
  - Fortunately …

Tokens

- A *token* is the smallest unit of a program
- Tokens include
  - Reserved words: parts of the C++ language
  - Identifiers: names defined by the user
    - Variables
    - Functions
    - Constants
    - Structures
    - Classes

C++ Identifiers

- User created names must only contain
  - Letters, both upper and lower case
  - Digits (0 to 9)
  - The underscore character (_)
- Identifiers must *begin* with a letter or _
  - By convention variables usually start with a lower case letter
- C++ is case sensitive
  - age and aGe are different identifiers

Naming Things

- Identifiers should be descriptive to make programs easier to understand
  - Names should be a reasonable length, and
  - Give information about the identifier's purpose
- Say we want to store a rectangle's height
  - *height*, or *rect_height*, or *rectHeight* are fine
  - *ht*, or, even worse, *a*, are not descriptive
  - *variableForTheHeightOfARectangle* is too long

Legal Identifiers?

- cycle
  - ✓
- A!star
  - ✗ (!)
- int
  - ✗ (reserved word)
- Score
  - ✓ (though upper case not recommended for a variable)
- xc-y
  - ✗ (-)
- race#1
  - ✗ (#)
- my_variable
  - ✓ (but not descriptive)
- Int
  - ✓ (but horrible – why?)
- cmpt130variable
  - ✓ (but not descriptive)
- 3rdVariable
  - ✗ (starting 3)

Variables Names Summary

- A variable declaration should begin with the type of the variable
- A variable name should
  - Be legal
    - Only contain a-z, A-Z, 0-9 or _
    - Not be a reserved word
  - Start with a lower case letter
  - Convey meaning

What is a Variable?

- A variable is a value stored in main memory
  - Main memory = RAM = Random Access Memory
- Values are encoded in binary
- A variable is therefore a sequence of **b**inary dig**its** (bits) of a particular length
- Variables have:
  - A type – the kind of data the variable stores
  - A value – the data stored in the variable
  - An address – the location of the variable

Operations

Operations

- In the example program we saw two types of operations
  - Numeric operations
  - Assignments
- Sometimes the meaning of an operator may be dependent on the type of the operands
  - e.g. division

Numeric Operations

- C++ uses normal arithmetic operators
  - These can be used wherever it is legal to use a value of the type produced by an expression
  - The result of the operators varies somewhat based on the type of the operands
- The +, -, and * operations perform addition, subtraction, and multiplication
  - The result of division is determined by type

Integer Division

- When both operands are integers the / operator performs integer division
  - The number of times the RHS goes into the LHS
  - 11 / 3 is equal to 3
- The % (modulus) operator returns the remainder in integer division
  - 11 % 3 is equal to 2

Increment

- It is common to add one to (some) variables
  - C++ provides the increment operator, `++`
    - `count++; // adds one to count`
  - This is typically used with *int*s but can be used with other numeric types
  - There is also a decrement operator, `--`
- There are two ways to use increment
  - prefix and postfix
  - `x++` may have a different result from `++x`

++ Prefix and Postfix ++

- The increment operator can be used before or after a variable
  - Before is the prefix version: ++x
  - After is the postfix version: x++
- The prefix ++ increments the variable *before* other operations in the statement
- The postfix ++ increments the variable *after* other operations in the statement

Assignment

- The *assignment operator*, =, is used to assign values to variables
  - The value on the *right* side is stored in the variable on the *left* side
    - The bit representation of the left operand is copied to the memory address of the right operand
- The type of the right operand must be compatible with the left operand
  - More on this later

Assignment And …

- C++ has shorthand operators that combine assignment and arithmetic
  - `count += 2; // count = count+2`
  - `cost -= rebate; /* cost = cost-rebate */`
  - `bonus *= 2; // bonus = bonus*2`
  - `time /= speed; /* time = time/speed */`
- These operators are just typing shorthand
  - They still retrieve the value of the variable, perform the operation and then store the result in it

Precedence

- In a complex expression operations are performed *one at a time* in *order*
  - The order is determined by *precedence rules*
    - Expressions in ()s are evaluated first
    - Then function calls
    - Then normal arithmetic precedence (*,/) then (+,-)
    - Assignment is usually last
  - More complete precedence rules will be provided later in the course

Constants

- A constant is a special kind of variable
  - Used to represent a value that does not change
    - such as the value of *pi*
- Use constants to name values in a program that won't change
  - To avoid *magic numbers* in a program
    - Numbers whose origin is unclear
  - This makes a program easier to understand and easier to modify

Declaring Constants

- Constants are declared by preceding the variable declaration with the keyword `const`
  - The constant must be given a value, and cannot be changed in the program
  - By convention constants are written in capitals
- Assume that there are 10 branches in a program to maintain bank information
  - `const int BRANCH_COUNT = 10;`

Using Functions

Using Functions

- To use a function write the name of the function followed by its arguments
  - The arguments must be in parentheses
  - If there are no arguments the function name must be followed by empty parentheses
  - The arguments must be of the appropriate type
- Some functions return values
  - And are usually assigned to a variable
  - Or used in a calculation

Function Call Examples

```
printf("hello world");

answer = sqrt(7.9);

area = PI * pow(radius, 2);
```

printf has no return value

the result of calling *sqrt* is assigned to *answer*

pow executes, and is replaced by its return value

function names

function arguments

Function Precedence

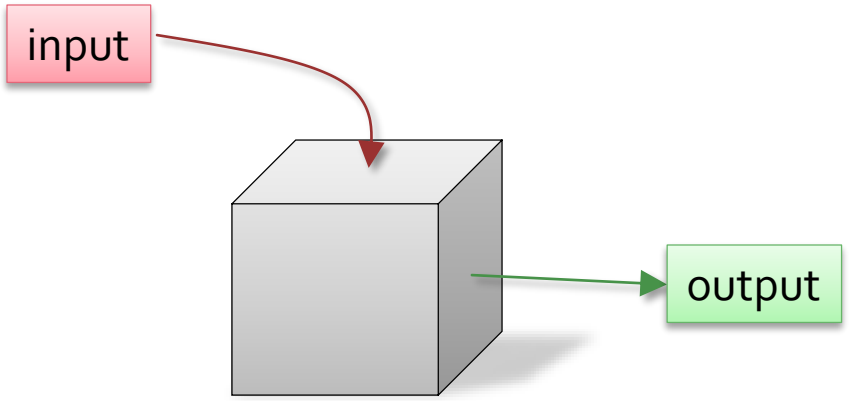
- Function calls in statements have precedence over most other operations
  - Such as arithmetic or comparisons
  - A function is executed and its result returned before other operations are performed
- The function call is replaced by its return value
  - Which may then be used in a calculation

Function Execution

- When a function is called program execution switches to the function
  - Each statement in the function is executed in turn until
    - The last line is reached, or
    - A *return* statement is processed
- Execution then returns to the line of code where the function was called from
  - The function call is replaced by the return value

What is a Function?

- It may be helpful to think of a function as a black box
  - That has inputs and produces output through some process



Function Calls

- Consider this line of code:
  - `x = sqrt(y);`
- What is necessary for this to work?
  - *x* and *y* must be existing variables
  - *sqrt* must be a valid function name
- But that's not all! The *sqrt* function must
  - Return a value, that is compatible with *x*'s type
  - Require input of a type compatible with *y*'s type

A Function Call Example

- Here is another example
  - `x = sqrt(y) + 4;`
- There are a number of steps here
  - The program switches execution to *sqrt* function
  - Which computes the return value
  - The return value replaces the function call
  - 4 gets added to the result of calling *sqrt*
  - And the result of the sum is assigned to *x*

Using Functions

- We will use many functions in this course
  - Including ones we have written ourselves
- When using a function, it must be given the appropriate arguments
  - Arguments are information that the function needs to perform its task
  - They are given to (passed to) the function in the brackets following the function's name

Errors

Errors

- Most programs contain errors
  - Most large *working* programs contain errors
- A substantial amount of time is spent finding and fixing errors (known as debugging)
- We can categorize errors into three very broad types
  - Compilation and Linker errors
  - Run-time errors
  - Logic errors

Compilation Errors

- These errors prevent a program from being compiled
- Compilers are very picky and complain at the slightest mistake, some common errors are
  - Misspelled words, C++ is case sensitive, so *int* is different from *Int*
  - Missing semi-colons, or brackets, or quotes
  - Incorrect arguments for functions
- You will spend a lot of time looking for errors

Other Errors

- Once a program compiles and runs there is still testing (and work) to do
- *Run-time errors* cause programs to crash
  - As a result, for example, of input type errors
    - These need to be predicted and handled in some way
- *Logic errors* cause programs to return incorrect results
  - And need to be caught by rigorous testing