

CS 161: Introduction to Computers and Programming – Chapter 1

The purpose of chapter 1 is to give you a general overview of how computers and programs work. The notes and Chapter 1 introduce the basics of programs, some fundamental terminology and a structure or sequence of steps for creating a program.

Many of the topics discussed in Chapter 1 are covered in much more depth in subsequent classes in your degree program. The notes highlight where these future classes will give you a much more complete understanding of the topic.

Contents

What is a program and why do we create them?	4
Computer hardware.....	4
The CPU.....	4
The fetch/decode/execute cycle	4
Main memory.....	5
RAM details.....	5
Secondary storage	5
Input devices.....	5
Output devices.....	5
Programs and programming languages.....	6
Algorithms.....	6
Turning algorithms into programs	6
Machine code, assembly language, and compilers	6
High level languages	6
Source code, object code, and executable code	7
Source code.....	7
Converting source code to executable code	7
What is a program made of?	7
Language elements	7
Key words.....	7
Programmer-defined identifiers.....	7
Operators	7
Punctuation.....	7
Syntax.....	8
Lines and statements	8
Variables.....	8
Input and output.....	8
Example of console output	9
How do I go about writing a program?.....	9
Step 1. Define what the program should do	9
Step 2. Visualize the program running on the computer	10

Step 3. Use design tools to create a model of the program	10
Hierarchy charts	10
Flow charts	10
Pseudocode	10
Step 4. Check the model for logical errors	11
Step 5. Write the program source code	11
Step 6. Compile the source code	11
Step 7. Correct any errors found during compilation	11
Step 8. Link the program to create an executable program	11
Step 9. Run the program using test data for input	11
Step 10. Correct any errors found while running the program	11
Step 11. Validate the results of the program	12
Some suggestions on the programming steps	12
Final thoughts	12

Introduction to Computers and Programming

What is a program and why do we create them?

The book *C++ Early Objects*, defines a program as “a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.”

Computers and software go hand in hand. Computers are powerful because they can do many things and programs are the way that we get computers to do these many things.

Computer hardware

A typical computer consists of:

1. The central processing unit (CPU)
2. Main memory (random-access memory, or RAM)
3. Secondary storage devices
4. Input devices
5. Output devices

The CPU

The CPU is the heart of the computer. A program is a sequence of instructions stored in main memory. When a program is run, the CPU fetches the instructions and executes or follows the instructions.

The fetch/decode/execute cycle

The process of program execution is a repeating cycle called fetch/decode/execute.

fetch – get the next instruction from main memory.

decode – determine what instruction to perform

execute – perform the instruction

Every program ends up as a sequence of basic instructions that consist of arithmetic and logic operations and control flow operations.

Arithmetic and logic operations include add, subtract, multiply, divide and comparison of values (equality, less than, greater than).

Control flow operations are used to determine what instruction to execute next. For example, based on the instruction, the program may skip or branch to another part of the instructions list.

You will learn the details of how CPUs process instructions in CS 271, Computer architecture and assembly language.

Main memory

Main memory or RAM is used to store the program while it is executing and to store the data that the program is working with.

RAM details

- The CPU is able to quickly access any location in RAM
- RAM is called volatile storage. Unlike persistent storage, when a computer is turned off or when a program finishes executing, the values stored in RAM are erased.
- RAM is divided into storage units called bytes. A byte is a sequence of eight bits.
- A bit is the smallest element of the RAM and it stores a binary digit, either a 0 or a 1. Every program and every data value in your computer is stored as sequences of 0s and 1s.

Secondary storage

Secondary storage provides long lasting and persistent storage. Unlike RAM, data stored within secondary storage does not disappear when a computer is turned off or restarted. The most common form of secondary storage for large computers is a disk drive but computers can use other forms of secondary storage such as solid state drives which use memory chips that maintain data values without power.

Like main memory, secondary storage also stores information as sequences of 0s and 1s as bits and bytes.

Input devices

We typically think of keyboards and mice but input devices can include cameras, microphones, and many other types of various sensors when you start thinking of computers embedded in cars, electronics, and almost any electrical device.

Output devices

The information a computer sends to the outside world is called output. If a person is involved, output is typically sent to an output device such as the computer screen or a printer. Not all programs will output data to an output device. Instead, the output may be sent out over a computer network or stored in a database.

Programs and programming languages

As stated above, programming involves creating a set of instructions that a computer will follow to solve a problem or accomplish a task. Let's refine our terminology in this section.

Algorithms

An algorithm specifies a finite sequence of clearly defined operations to solve a specific problem or class of problems. You can describe the steps of an algorithm in many ways including words (also known as natural language), flow charts, pseudo-code (described below), and programming language code.

As the complexity of the problems increase, it is important to design algorithms that are efficient (i.e. quick) and correct in that they produce the specified output for any valid input. In CS 325, Analysis of Algorithms, you will learn techniques for analyzing complexity and for proving correctness.

Turning algorithms into programs

An algorithm can be implemented in many different computer languages and a single program may use or implement many different algorithms. For example, you may use a sorting algorithm to order messages and a decryption algorithm to understand the messages.

Machine code, assembly language, and compilers

A computer's CPU executes your program's instructions. However, while you write a program in a language like C++, a computer CPU can only follow instructions coded as sequences of 0s and 1s. A software compiler is a special program that converts statements written in the computer language to a binary form (0s and 1s) called machine code. Since it is hard for us to recognize 0 and 1 sequences, there is a low-level (close to hardware) programming language called assembly language which uses short abbreviations and patterns to describe what the CPU must do. For example, the assembly statement "MOV AL, 61h;" means copy the following value (61h, hexadecimal representation of 97) into memory location "AL".

You will learn much more about machine code and assembly language in CS 271, Computer architecture and Assembly Language.

High level languages

In this class, you will learn C++ which is a high level language. High level languages are those computer languages that hide many of the low level details of the computer system and tend to use more natural words and symbols versus words such as "MOV" in assembly language which is a low level language.

C++ is one of many high level languages. To see the current popularity of all computer languages, go to the [TIOBE Index](#).

Source code, object code, and executable code

Source code

When you start creating programs in this class, you will be creating source code. Source code is saved in a simple text file called a source file.

Converting source code to executable code

Your computer does not understand source code. You must use a compiler to convert source code to executable code which you can start and run on your computer.

During the process of converting your source code to an executable file, the C++ compiler will create object code.

Source code is converted to what is called object code by the compiler. The object code for a C++ program is saved in files with an .o or .obj suffix. In a final step called linking, the object files are combined with any library routines (routines provided by the language for use by you) to produce the final executable file with an .exe extension.

Depending on how you compile your program, you may or may not actually see the various steps of converting your source files into an executable. For example in many IDEs (integrated development environments such as Visual Studio, Code::Blocks, or XCode), the intermediate steps are taken care of automatically so you can click on a “build” button and the executable is created.

What is a program made of?

Language elements

Most programming languages include the following elements.

Key words

Key words are words that have a special meaning in the language. They can only be used for their intended purpose. Also known as reserved words.

Programmer-defined identifiers

Programmer-defined identifiers are words you choose as the programmer to define variables or programming routines.

Operators

Operators perform operations on one or more operands. An operand is a piece of data. The various arithmetic symbols such as +, *, and / are examples of operators.

Punctuation

Punctuation characters mark the beginning or ending of a statement or separate items in a list.

Syntax

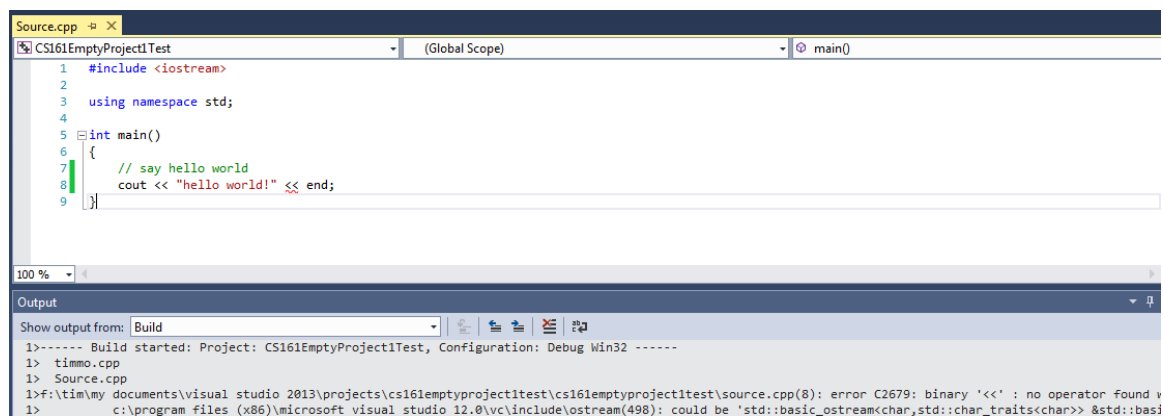
Rules that must be followed when constructing a program. These rules define how you can combine key words, programmer-defined identifiers, operators, and punctuation.

C++ Specifics

You will start learning the C++-specific language elements in Chapter 2.

Lines and statements

We often think of programs as made of up lines and statements. A line is just a single line in the program. You can display line numbers in most IDE source code editors. In Visual Studio 2013, you have to turn them on as they are off by default. You will often see references to line numbers when you compile your program and you have an error.



```
Source.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     // say hello world
8     cout << "hello world!" << end;
9 }
```

Output

Show output from: Build

1>----- Build started: Project: CS161EmptyProject1Test, Configuration: Debug Win32 -----
1> timmo.cpp
1> Source.cpp
1>f:\tim\my documents\visual studio 2013\projects\cs161emptyproject1test\cs161emptyproject1test\source.cpp(8): error C2679: binary '<<' : no operator found v
1> c:\program files (x86)\microsoft visual studio 12.0\vc\include\ostream(498): could be 'std::basic_ostream<char,std::char_traits<char>> &std::basi

The screen shot shows a program with an error. When the program was compiled, the output (gray shaded windows above) included “source.cpp(8)” indicating that the problem was on line 8 of the file called source.cpp.

A statement is a complete instruction that causes the computer to perform some action. A statement may span more than one line. The meaning of a statement will make more sense once you start programming in Chapter 2.

Variables

A variable is an item you create in a program to store a piece of data. You can name a variable anything you want as long as it is not a keyword and you follow the language’s syntax rules. As the name implies, the value of the data associated with a variable can change during the program execution.

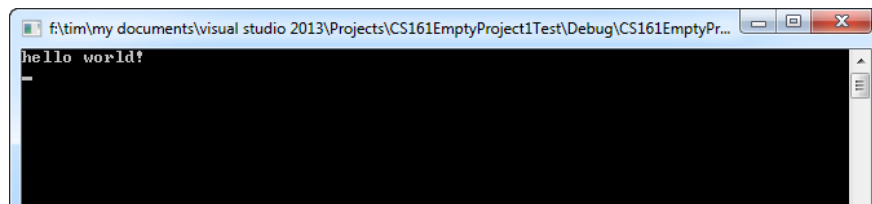
Input and output

Two of the most important considerations when programming are the input and output. Many of the programs you will write for the class assignments will use keyboard inputs. The program will prompt for input and you will type in a response. As you progress through the degree

program, you will gain experience using files, databases, web resources, and other sources for input.

In many if not all of your CS 161 assignments, you will direct the program output to the console. You don't often see console output if you are running applications on Windows or OS X because those types of application programs use graphical user interfaces (GUI). However, GUI programming adds a lot more work to creating a program and our job in CS 161 is to teach you the fundamentals of programming so we will stick with console output.

Example of console output



How do I go about writing a program?

The goal of Chapter 1 is to provide you with a general background about computers and programs. In Chapter 2, you will start creating programs in C++. To prepare you for writing your first program, the book presents the following step by step process.

Step 1. Define what the program should do

In any level of software creation ranging from your first home work to a team creating a new iPhone app, it is essential to have a clear understanding of what the program will do and not do.

You should consider the following:

Purpose – overall description of what the program will do.

Input – what types of input are needed and where will they come from.

Processing – what kind of work needs to be done.

Output – what will you do with the outcome of the program and how will you display, save, or transmit the program outcomes.

Programmers are often presented with a set of requirements which define the inputs, the functionality, and the output.

The software engineering classes, CS 361 and CS 362, will include instruction on requirements gathering and assessment.

One of things that make programming so interesting is that there are unlimited ways to write a program to meet a set of requirements or serve a specific purpose. This allows extreme amounts of creativity.

It is pretty exciting when someone asks you “can you create a program to do this super cool thing?” and you can say yes!

Step 2. Visualize the program running on the computer

Once you know what the program needs to do, start conceptualizing what the program will look like. You can do this in your mind but as programs get more complex, it can be helpful to create mockups or diagrams of input and output screens.

As you start programming, I think you will notice that you start paying more attention to the software you use and what you like and don't like about an application.

CS 352, Introduction to usability engineering, will teach you ways to assess and improve the usability of software.

Step 3. Use design tools to create a model of the program

Before you start actually writing code, it is very helpful to create a model of the program. Some simple modeling tools are hierarchy charts, flow charts, and pseudocode.

Hierarchy charts

A hierarchy chart is a graphical representation of the structure of a program. Boxes are used to represent each step in the program and boxes are connected to show relationships between steps. The most important step or task is listed at the top of the chart and subtasks are then listed in subsequent levels of the chart. See page 20 in the book for an example of a hierarchy chart. Hierarchy charts typically end up with a pyramid shape.

Flow charts

A flow chart like a hierarchy chart lists tasks in boxes but in a flow chart there is a defined flow or logical sequence of steps from a starting point to an ending point.

Pseudocode

Pseudocode is cross between human language and computer language. Basically, you describe what the program needs to do in simple statements or lines that you write without worrying about the exact syntax of the language.

For example, if a program included a task to search a directory for all text files, the pseudocode might look like the following:

```
create empty list for saving text files
get a list of all files in the directory
for each file in the directory list
```

if file is a text file
 save file name in list of text files

Step 4. Check the model for logical errors

Logic errors are mistakes that cause a program to produce erroneous results. Trace through your flow chart or pseudocode and make the each step produces the desired results. Make sure your model considers all possible cases. In the pseudocode example above for searching a directory for text files, my model should properly handle cases where the directory does not exist, the directory is empty and the directory includes no text files. Bugs or crashes in computer programs often result from unexpected or unusual situations because the programmer builds the program based on typical or expected inputs and results.

Step 5. Write the program source code

After understanding the purpose of the program and developing a model of the program, start writing the program using the tool of your choice. Since source files are simple text files, you can write programs in simple text editors but many programmers prefer to use IDEs which includes features to color text by purpose such as comments, keywords, and variables, to highlight syntax errors as soon as you create them and to use auto completion to reduce the amount of typing required.

Step 6. Compile the source code

Build the program using the compiler. The compiler checks for syntax errors and creates a machine code version of your program.

Step 7. Correct any errors found during compilation

If the compiler reports errors, fix them and recompile the program.

Step 8. Link the program to create an executable program

As noted earlier, many IDEs link the program and create the executable automatically. However, you may find classes or tools where you will need to link the object files to create the final executable program.

Step 9. Run the program using test data for input

When you successfully build or compile a program, you have created a program without syntax errors. However, it is important to check for logic errors. Remember, successful compilation does not mean your program will run correctly. Test your program with both expected and unexpected inputs.

Step 10. Correct any errors found while running the program

If you find logic errors in your program, you have to correct them. However, unlike syntax errors where the compiler will tell you that you have an error on a specific line, you must find logic errors by yourself. We will cover debugging techniques in a separate set of notes but techniques include running a program line by line and by watching the values of variables as the computers executes the steps of your program.

Step 11. Validate the results of the program

This is a final test to validate that your program solves the problem specified in step 1. This testing should run through all steps of the program from beginning to end.

Some suggestions on the programming steps

The book presents a very orderly progression of creating a program which is a good starting point. However, I would suggest breaking down steps 5 through 9 into smaller components. I would never write an entire program before compiling and testing. Consider the following:

1. Break up your program into components such as input, a set of processing steps, and output.
2. Write the code for a component, then compile it and test it.
3. Once a component is complete (i.e. fully tested), proceed to the next component.
4. As you complete components, you test the integration of the components. By this, I mean do the components work together as expected.

The reason for breaking a program up into components is that it is often easier to debug from a known point of where things work. For example, I build and test the first component and everything works. After I add the second component, the program logic is broken. Since the first component had worked, I would focus my debugging on the second “new” task. If you write the entire program before testing, you have to consider that the error could be happening anywhere in the program.

Final thoughts

You are going to learn how to program using C++ in this class. As you take additional classes and go out into the professional world you are going to learn other languages. As you do that, you will see that most programs and software languages do the same thing. You can get input, do math, create variables, assign variables, sort values, work with text, output data etc. The approach to solving the problems are the same; it is just that the syntax is different. As you learn different languages, each language will be easier to learn because you will understand the underlying approach and techniques for solving problems so you just have to learn the syntax differences.