

1 Week 1

1.1 Introduction to programming

Every one of you has used a computer before. You've used a computer to help enroll in classes, order some clothing, or watch Netflix. All of these were once done without the use of computers, but we use a computer today to make things quicker. Imagine the steps involved in shopping. Items are selected. Then when the customer goes to purchase items, the prices are tallied, and tax & shipping calculated. The customer will supply the payment and shipping address information. In a few days they'll get their items in the mail. All without ever having to interact with a human. The computer facilitates the job of a sales clerk via a sequences of predictable instructions. A program has been written to handle a reasonable inputs (different clothing items, valid credit card information) and then executes according to the customer and business owner's personalizations.

As programmers we write code to automate processes. Our code provides instruction for the computer so it must have syntax that's understood by the computer. The computer is very simple and literal. Moreover, we must anticipate human user inputs and how they'll be stored and processed by the computer. Different inputs can produce different results. In the next few weeks we'll think about processing code and learning how the computer interprets things in practice.

1.2 Why C++?

Today might be the day you begin programming. The language that's been chosen as your first programming language is C++. This has been done for a few reasons. There is lots to learn about C++ so it's been organized into a 3-part series (PIC 10A, 10B, 10C). The other main reason is once you've complete this introductory course, you'll be aware of most of the things you could encounter in other programming languages such as Python, Java, and JavaScript. For example, consider when we declare a variable we'll use for counting like an integer `i`. In C++ we indicate it's an integer at the beginning `int i = 0`. In Python (which can be written using C), we'd write `i = 0`; the type `int` is not explicitly written but the variable `i` will still be of type `int`. C++ will encourage you to question types more readily and you'll immediately avoid most 'type errors' that new Python coders experience. C++ forces you to think about concepts which will be used for all languages. Also, you have less control over memory in Python but knowing practically how memory works will lead you to program more strategically.

Another component of C++ that we benefit from is the reason it's different from C language: its ability to create objects. In the second half of the class you will learn object-oriented program. This allows you to use code to create instructions for more complex computer things such as creating new new data types. You might use object-oriented programming to create a machine learning estimator in Python or a graphical user interface in C++ or Python. Perhaps you might want to create a button on a webpage that a user can click to send a message to their friend. Once you get an understanding of object-oriented programming through C++, you will know what to look for in another languages you could learn.

1.3 Writing a first program

Earlier we talked about how a program is a sequence of instructions interpreted by the computer. There's a few more steps involved going from the text we type to the computer actually doing some action. This involves a translation from the human-readable C++ we will write to a assembly code

the computer understands how to execute. You don't need to worry about this machine level code but you should know there is some processing going on behind the scenes. One step in that process is checking that your written C++ code obeys syntax rules. This is to avoid code being written and leaving too much up for the computer to interpret. This can be disastrous since remember: computers are simple and literal. Compilers will warn the user before the code executes if there is a syntax error. Certain compilers can be stricter about some things than others, but there really not "strictest" compiler. Ideally you want to test multiple compilers before publicly sharing a code because you don't always know who will be using it and you want everything to run smoothly as possible to avoid getting complaints.

1.3.1 Compilers and how your homework code is tested

For this course, you will be testing your code on a compiler that everyone will have access to. This is Visual Studio Community 2023 which is the one available on all the PIC lab computers (MS 2000). If you have Windows OS, you can download a copy on your own computer. If you have Mac OS, you can download XCode but you should test your final code on the PIC lab computer before submitting it. There's been places where XCode's compiler is more lenient than VS Community and vice versa. Note the PIC lab is only open weekdays 9-5.

When you are given assignment, you will be given test cases. For example, you might write a program that allows the user to input two integers and the program should output the sum. So if you gave the computer 1 and 2, you should expect to see 3. Your homework will include some test cases and the results from running the solution code. You are responsible for checking those test cases as well as additional made-up test cases. Think of what another user might type. Have a friend or classmate come up with some test cases. Double check the assignment to see what could be valid test cases. Post on Piazza your input test cases and the output results from your code to check with your classmates and instructors. It's fine to share test cases without sharing your code.

1.3.2 What is a .cpp file?

Let's start programming. To begin programming you really only need a simple text editor. Something even simpler than a rich text editor like Microsoft Word that does fancy things like images, fonts, and formatting. You might use text editor like [Sublime Text](#), that only takes characters but presents it in nice color. To write a C++ program you create a simple text file and give it the extension .cpp. Then you're ready to go. Let's call our first program `main.cpp`.

Ok. There's a bit more to it. I mentioned before, you need a way for the computer to interpret the text you write in `main.cpp`. So to make this easier we use a Integrated Development Environment (IDE) like [XCode](#) or [VS Community](#). So you should open up a project in there and name a file `main.cpp`. `main.cpp` is still just a text file but by pressing the "Play" button in the IDE, our (hopefully) error-less code will be compiled to something the computer can use.

1.3.3 Template for your first program

Here's a template for your first program:

```
/* These are block comments,  
   which are great for taking notes.  
   This will be ignored by the computer. */
```

```

int main() {
    // These are line comments
    // insert code within: int main() {...}
    // cout means "console out"
    cout << "Hello, World!\n";
    return 0;
}

```

Put this in a script `main.cpp` and press the “Play” button.

- If you’re using Xcode, the console (where you will interact with the program) is at the bottom and might be collapsed (look for a square with a solid rectangle inside and click to toggle).
- When you use VS Community, the console appears in the foreground as a separate window.

You should see the result of executing the code: `Hello, World!` and another message saying the function `main` returned 0, which for now means the computer was able to successfully run the code you wrote.

But maybe something went wrong. You might have encounter your first bug. See if you can find the error message. There appears to be an error with the word `cout`. The simple, literal computer has no idea what that is. Let’s type at the very top, above the block comment the lines

```

#include <iostream>
using namespace std;

```

These lines include a file that has the definition of `cout`. `iostream` is some C++ code that handles stuff like printing to the console. The command `cout` also lives inside the `std` namespace. This is where many of our commands for this class will live so we include `using namespace std;` to type less (`cout` instead of `std::cout`). Now your code should run successfully.

In the next few exercises we will go over the errors you might encounter, how to read them, and approaches to fixing them.

1.4 Encountering, reading, and fixing basic errors.

When you get an error it can feel pretty alarming at first. It might tell you your Build Failed, which feels pretty harsh. On top of that, the simple, literal computer will give you a seemingly cryptic message because it will be in mostly the computer’s language. It might even suggest a ‘Fix’ based on what it thinks could be the correction. Though, I would not apply a fix if you don’t have an idea what the error is first. Mostly, it will be correct but it could do something really weird. At least once PIC instructor broke their Xcode installation mid-lecture by “experimenting” with ‘Fix’.

The very first thing you should consider, rather than try to interpret or Google the phrases it returned, is where the error occurred in your programming. Look for which line the IDE points out as the troubled line in its error message. Then find that line in your code. Your IDE might even highlight the line. Examine that line, character by character to see if you written what you intended.

Aside: If a piece of code was copied from a pdf (best to avoid this), it might not be the same symbol your IDE expects. For example, quotes come up specially formatted and strange characters can be inserted. If your computer autocorrects your typing (maybe you have a setting / feature

installed to autoformat or assist in typing in a language other than English), it can cause issues in your syntax or when you type into the console. See if you can turn that setting off for your IDE or text editor so it doesn't cause you issues.

There are going to be three types errors you'll encounter and handle by fixing your code. Other types of errors are rarer and might just mean you need to save your files and restart your IDE. You'll learn and practice spotting errors in discussion and will demonstrate your understanding of them on exams.

The three main types of errors are:

Build Errors

- Breaking the rules of the language.
- Syntax errors. The compiler finds these before the code has been executed.
- Examples: Missing semicolon, missing headers, namespace error, using `>` for `>>` and `<` for `<<`, missing `{}`

Runtime Errors

- Runtime errors occur while the computer is executing code.
- Examples: Indexing a container out of range using `.at`, allocating heap memory that is too big.

Logic Errors

- When your code runs without a runtime error but is not doing the correct thing, for example, saying `1 + 1` is 5. These errors won't be flagged by the computer; you will have to spot them by testing your code.
- You might be able to spot your error when reviewing your code. The best approach to finding a logic error is to comment out code within `main` and go line-by-line, printing out (`cout`) the variables to see what's happening.

Inexperienced or careless coders can encounter 'undefined behavior'. This is when the C++ Bible (The Standard) make no guarantees about how the code will execute. It is especially bad because one compiler might run code without error but another compiler might hit an error. Therefore, it's best to ensure that the behavior of your code is well-defined across compilers by avoiding undefined behavior.

1.5 Exercises

1.5.1 Error spotting exercises

Error spotting will be on the exam. Look at the follow code below and state:

- What errors occurred (build, run time, or logic) or if there are none.
- In the case of a logic error, what does the code currently accomplish?
- How the code can be fixed (use the line numbers to help with the description).

A description of how the code should perform will be put in a block comment. After you make your initial guess, type the code in your IDE and run it. Copying very carefully may help you find some of the errors. Play around with the code to fix it.

```
1. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code should print: 0 */
    5  int main() {
    6
    7      int i = 0
    8      cout << i << endl;
    9
   10      return 0;
   11  }
```

```
2. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code should print: Hello, World! */
    5  int main() {
    6      cout << "Hello, World!\n";
    7      return 0;
    8  }
```

```
3. 1  #include <iostream>
    2  using namespace std;
    3
    4  #error Oops
    5
    6  /* This code should print: Hello, World! */
    7  int main() {
    8      cout << "Hello, World!\n";
    9      return 0;
   10  }
```

```
4. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code should print: I feel pretty. */
    5  int main() {
```

```

6
7     cout << "I feel pretty." < endl;
8
9     return 0;
10 }

```

5.

```

1  #include <iostream>
2  using namespace std;
3
4  /* This code should print: I feel pretty. */
5  int main() {
6
7      cout << " feel pretty." << endl;
8
9      return 0;
10 }

```

This a small error and a moreover a difficult one to spot. The computer how no idea it wasn't printing the correct thing and doesn't show an error. However, it's important you have the program do what it intends. In this case, you need to print things formatted exactly as you meant. This could be a reason you'll lose a couple of points on homework. If you were working on a project with other programmers, the format of your code output could matter. Having data formatted consistently will mean your colleague can know what to expect as the input into their program.

6.

```

1  #include <iostream>
2  using namespace std;
3
4  /* This code should print: Hello, world! */
5  int main() {
6
7      return "Hello, world!";
8  }

```

7.

```

1  #include <iostream>
2  using namespace std;
3
4  /* This code should divide x by 0 and store it as y */
5  int main() {
6
7      int x = 10;
8      int y = x / 0;
9

```

```

10     return 0;
11 }

```

For this one, you should try it on multiple computers. Some may encounter a runtime error; some may not. VS will give an error; whereas XCode silently lets users divide by 0. Welcome to C++ and the perils of undefined behavior: not all compilers will produce machine code that executes the same. One should avoid undefined behavior at all costs!!

```

8.  1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code should store x + y as z which is 10 + 5 or 15. */
    5  int main() {
    6      int x = 10;
    7      int y = 5;
    8      int z;
    9
   10      x + y = z;
   11      return 0;
   12  }

```

```

9.  1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code is meant to
    5     assign the value 0 to i,
    6     assign the value 1 to j,
    7     then assign the value of j to i.
    8
    9     Another way to say this, this code is meant to
   10     assign i the value 0,
   11     assign j the value 1,
   12     and then assign j the value i. */
   13  int main() {
   14      int i = 0;
   15      int j = 1;
   16      i == j;
   17
   18      return 0;
   19  }

```

Notice: A warning not an error, but listening to warnings is useful.

```

10. 1  #include <iostream>
    2  #include <string>

```

```

3  using namespace std;
4
5  /* This code is meant to store s as 4 and x as 4. */
6  int main() {
7      string s = 4;
8      int x = "4";
9      return 0;
10 }

```

You could mean to do something like `int a = 4; string s = to_string(a);`. We will get more into casting in the next two weeks.

```

11. 1  #include <iostream>
2     using namespace std;
3
4     /* This code declares an array (a list) of numbers with a very big size */
5     int main() {
6         int* array = new int[10000000000000000];
7         return 0;
8     }

```

On many compilers, division by zero or square root of negative number could give a runtime error but officially it is undefined behavior. The previous code declares an array (a list) of numbers with a very big size. This results in out-of-memory error indicated by bad alloc in C++. This code is just to demonstrate a runtime error and might be discussed in later lectures.

1.5.2 Practice coding exercises

Review: variables, arithmetic

Write code to compute various things.

12. Calculate the area of a circle from its radius.
13. Convert Fahrenheit to Celsius:

$$C = (F - 32) / 1.8.$$
14. Determine if a number is even (Hint: `m % n` gives the remainder of `m / n`).
15. Compute the last digit of an int.

1.5.3 Predict the output exercises

Preview: variables, casting, arithmetic

In the following section, you will read code to predict the output. You will be tested to show you can understand code execution. You can assume all code in these problems runs error-free and

the proper libraries are included as well as the `std` namespace. Answer these by writing the output in a box to the right of the problem. (Don't include the Program ended with `exit` code: 0 line).

16. `#include <iostream>`
`using namespace std;`

```
int main() {  
    cout << "2 + 3 = " << 2 + 3 << endl;  
    return 0;  
}
```

17. `#include <iostream>`
`using namespace std;`

```
int main() {  
    double i = 1.7;  
    double j = 2;  
  
    cout << i + i << endl;  
    cout << j + j << endl;  
    cout << i + j << endl;  
  
    cout << i << endl;  
    cout << j << endl;  
  
    return 0;  
}
```

18. `#include <iostream>`
`using namespace std;`

```
int main() {  
    double velocity_init = 2;  
    double position_init = 8;  
    double gravity = -10;  
  
    double time_current = 1;  
  
    double velocity_current = velocity_init + gravity * time_current;  
    double position_current = position_init + velocity_init * time_current  
        + .5 * gravity * time_current * time_current;  
  
    cout << velocity_current << endl;
```

```

        cout << position_current << endl;

        return 0;
    }

```

19. `#include <iostream>`
`using namespace std;`

```

int main() {
    double i = 1.7;
    int j = 2;

    cout << i + static_cast<double>(j) << endl;
    cout << i + j << endl;
    cout << j + static_cast<int>(i) << endl;

    cout << i << endl;
    cout << j << endl;

    return 0;
}

```

20. `#include <iostream>`
`using namespace std;`

```

int main() {
    /* Refer to http://www.asciitable.com/
       The first 3 lines print
           65
           97
          112
       */
    cout << static_cast<int>('A') << '\n'
         << static_cast<int>('a') << '\n'
         << static_cast<int>('p') << endl;
    cout << static_cast<int>('A') - static_cast<int>('a') << endl;
    cout << 'A' - 'a' << endl;
    return 0;
}

```

21. `#include <iostream>`
`using namespace std;`

```

int main() {
    /* Refer to http://www.asciitable.com/ */

    cout << 'p' + 'A' - 'a' << endl;
    cout << static_cast<char>('p' + 'A' - 'a') << endl;
    return 0;
}

```

Could you do this without the ASCII table?

22. `#include <iostream>`
`using namespace std;`

```

int main() {
    int i = 0;
    int ipp = i++;
    cout << i << ' ' << ipp << endl;

    int j = 0;
    int ppj = ++j;
    cout << j << ' ' << ppj << endl;

    return 0;
}

```

23. `#include <iostream>`
`using namespace std;`

```

int main() {
    int i = 0;
    cout << i++ << endl;
    cout << i << endl;

    int j = 0;
    cout << ++j << endl;
    cout << j << endl;

    return 0;
}

```

Note ++i is better when not using assignment or cout.