# PY410 / 505
# Computational Physics 1

**Salvatore Rappoccio**

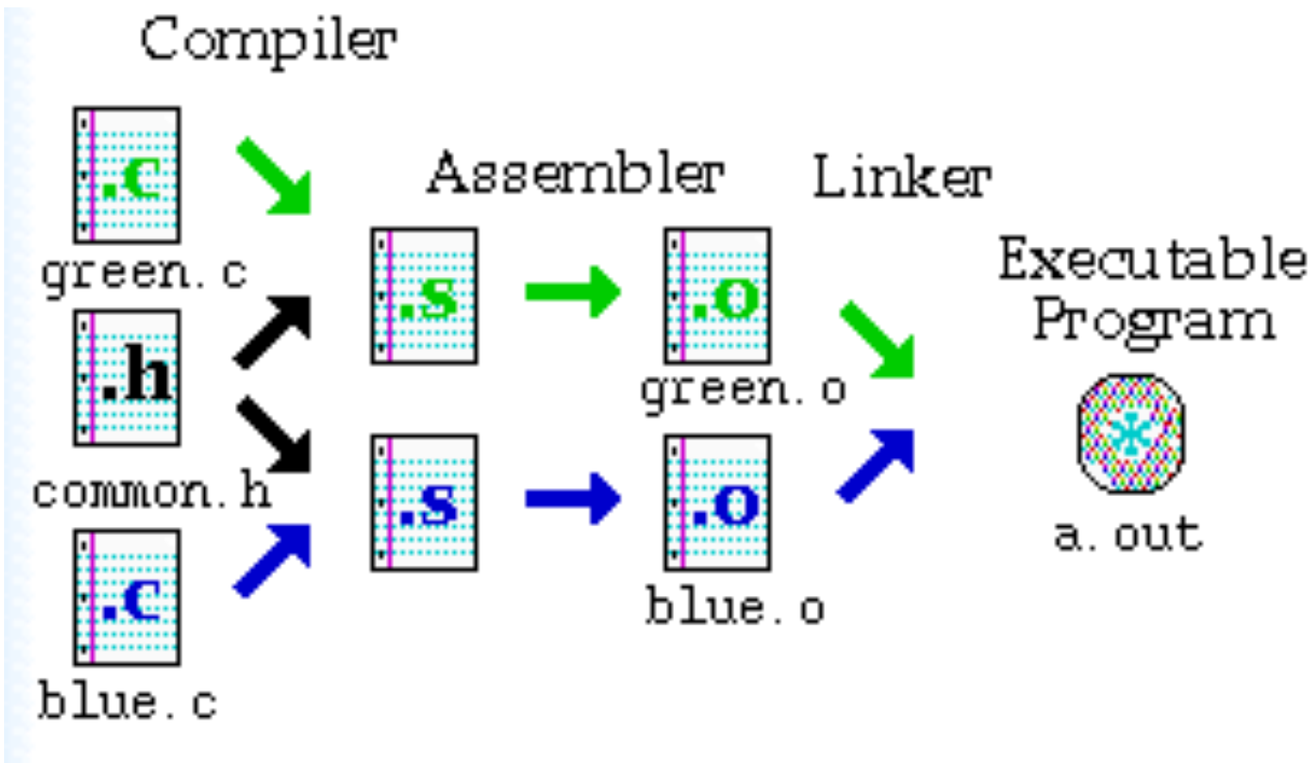
# Code

- Code is in CompPhys/ReviewCpp/BasicExamples

# C++

- Now that we know how data are represented, we need to learn how to do things with it.

- The first thing we need to learn is how to translate C++ to machine code :



# C++

- The “Hello World” program is simple:
- Go to “CompPhys/ReviewCpp/hello.cc“:

```
#include <iostream>

int main(void){
  std::cout << "Hello, world" << std::endl;
  return 0;
}
```

# C++

- The “Hello World” program is simple:
- Go to “CompPhys/ReviewCpp/hello.cc“:

```
#include <iostream>

int main(void){
  std::cout << "Hello, world" << std::endl;
  return 0;
}
```

Includes input/output modules

Main function : inputs nothing (void), returns an int

Prints “Hello, world” to the screen, with an end line

Returns 0 (success)

# C++

- In general, the program will always look something like this

*include directives*

```
int main() {
```

*program statements*

```
}
```

# C++

- In C++, you can add “comments” to your code too
- These are notes to yourself, and are ignored by the compiler

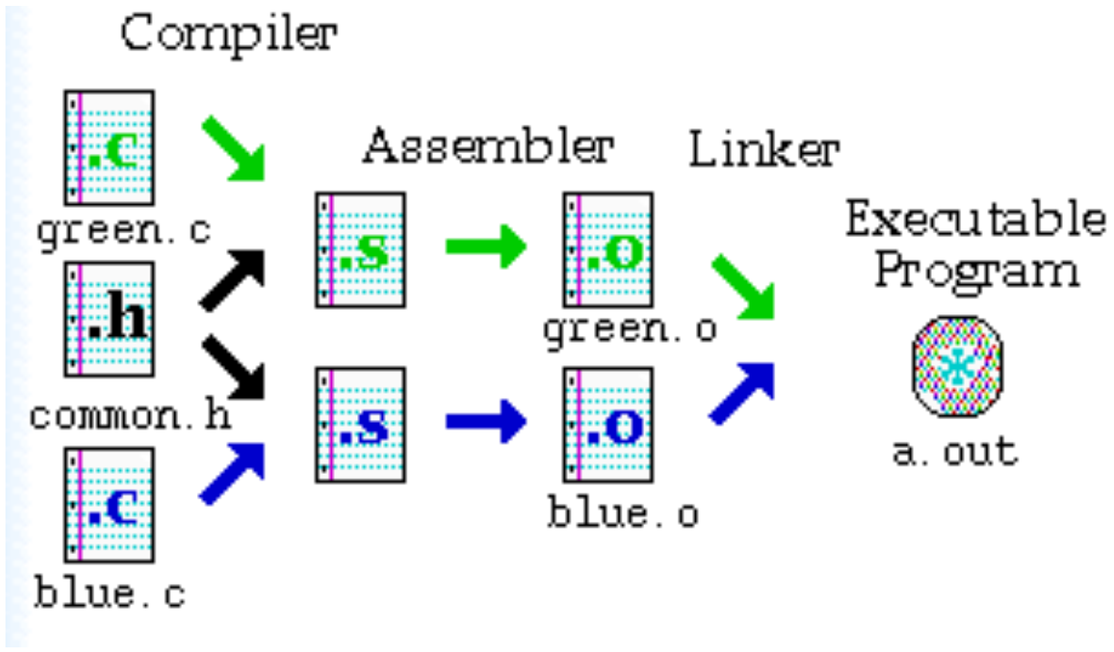
- The syntax is
  - “// “ at the beginning of a line : comments the line
    - // This is all one comment.
  - “/* */“ anything you type in between the *’s is a comment
    - /* This is a comment.
      So is this.
      And this.

      And this.
      */

# C++

- Now time to compile and execute :

```
$ ls
hello.cc
$ g++ hello.cc -o a.out
$ ls
a.out  hello.cc
$ ./a.out
Hello, world
```



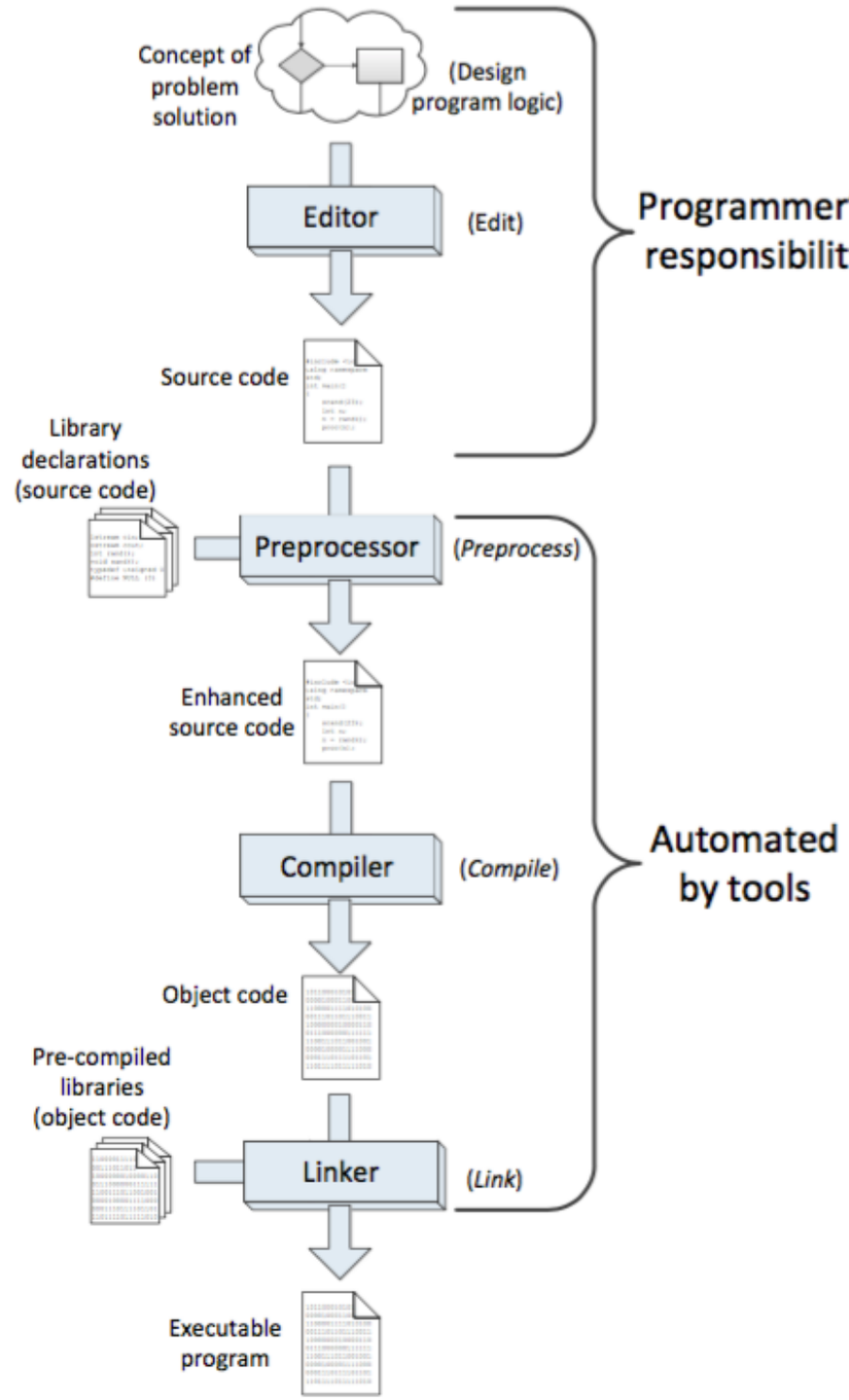

- First you see “hello.cc” in your directory
- Then we execute the compilation command with “g++”
  - The “-o” is the label we want the output to be named
- Then we see the executable “a.out” already
- What about the Assembler and Linker?
  - More on that later.
- We execute it with “./“ (this means “execute from this directory”)

# C++

- Figure 1.1 of the textbook gives a more complete overview

- You are going to be editing and then calling a compiler to do the rest

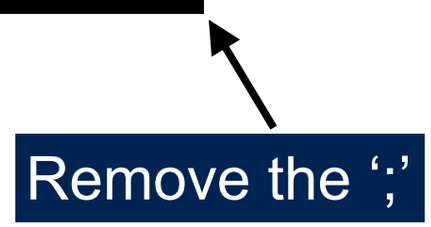
**Figure 1.1** Source code to target code sequence



# C++

- Notice : The syntax here has to be precise.
- What happens if we make a mistake?
- Compiler errors!
- Example: remove the semicolon (“;”) after “std::endl”:

```
std::cout << "Hello, world" << std::endl
```




- Now try to compile:

```
$ g++ hello.cc -o a.out
hello.cc: In function ‘int main()’:
hello.cc:5: error: expected ‘;’ before ‘return’
```

- Tells you EXACTLY what is wrong
  - Line number 5, expects “;” before the statement “return”
- Fix, and try again, and it works again.

# C++

- Moral of the story :
  - Computers are dumb.
  - They do EXACTLY what you tell them to do, so it's almost always (>99%) your fault when something goes wrong
  - This is called "pilot error" (you're the pilot)

- They also tell you EXACTLY what went wrong.
  - It may not be anything you are thinking about at that time, but it will be extremely precise
    - Note : **Clarity is NOT the same thing as being precise.**
      - Example : You get overheated in your jacket. Someone asks you what is wrong, and you reply "The internal core temperature of my body has not been maintained accurately, the insulation of the air within my coat is trapping excess heat inside it, causing my temperature to rise. My brain has sent an electrical signal to my pituitary gland to activate glands in my body to release perspiration to attempt to utilize a phase transition of the water to steam to cool my skin".
    - That's precise, but not clear.
    - "I'm sweating" is clear, but not precise.

# C++

- What does THAT mean?

- How about a different mistake : take the “std::” off the “cout”:

```
cout << "Hello, world" << std::endl;
```

Take off the “std::” here

- Now you get :

```
$ g++ hello.cc -o a.out
hello.cc: In function ‘int main()’:
hello.cc:4: error: ‘cout’ was not declared in this scope
```

- At this point, you have no idea what that means, but it tells you precisely what is wrong. It does not tell you “You missed the std::” like the last time, but it tells you what it knows to be the problem.
- We will get to “declarations” and “scope” later

# C++

- We will be using the “g++” compiler
  - This is the “GNU” compiler
  - What’s “GNU”? “GNU’s not Unix”.
  - Free, and up to date with ANSI standards

# C++

- These are examples of “compiler errors”
  - That means you didn’t set the code up correctly within a file

- We will also get to “linker errors”
  - That means you didn’t set the code up correctly ACROSS different files

- And of course, there are “runtime errors”
  - That means the code is technically correct syntactically, but you didn’t think about the output of the code.

# C++

- There are many tools to find problems:
  - Debugger : trace a program's execution
  - Profiler : examines the resource usage (memory and CPU)

- We will rely on very few of these tools in class, but if we have time later we will cover them

# C++ : Values and Variables

- Now that we have the structure, how do we do stuff?

- First concept : “Values”
  - ints like 1, 2, 4, 3359
  - floats like 3.14
  - strings like “my string”

- For numbers, these are represented just by the number in the code.
- Example: Instead of printing the string “Hello, World”, print the value “4”:

```
std::cout << 4 << std::endl;
```

- Now, if we compile and execute:

```
$ ./a.out
4
```

# C++ : Values and Variables

- But we already know that some kind of representation is necessary to store this, in two's complement
- What happens if you exceed the range of your chosen variable?
  - Example :

```
std::cout << -3000000000000000000000000000000 << std::endl;
```

  - Then the compiler error you get is

```
$ g++ hello.cc -o a.out
hello.cc:4:17: warning: integer constant is too large for its type
hello.cc:4: warning: integer constant is too large for ‘long’ type
hello.cc: In function ‘int main()’:
hello.cc:4: error: ambiguous overload for ‘operator<<’ in ‘std::cout << 23459524086239068l6’
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/ostream:457: note:
std::basic_ostream<char, _Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, char) [with _Traits =
std::char_traits<char>]
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/ostream:451: note:
std::basic_ostream<_CharT, _Traits>& std::operator<<(std::basic_ostream<_CharT, _Traits>&, char) [with _CharT
= char, _Traits = std::char_traits<char>]
```

- Huh?
  - It is first giving you a warning ("integer constant is too large for its type") and some details ("integer constant is too large for 'long' type)
  - Then the actual errors you get arise because the compiler is using the wrong type if the constant is too large!

# C++: Values and Variables

- So this is actually the best case scenario : the compiler caught the error.

- Technically this is NOT a syntax error : it is completely, 100% correct syntax
- This is the first example of a “runtime error” : the error would occur at runtime
- However, some clever compilers can catch SOME runtime errors, but it is **NOT REQUIRED BY THE STANDARD**
- So, if you take that code and use it on a different compiler and machine, it may run happily and give you a garbage representation of your too-big-value in two’s complement
  - That’s the worst case scenario: The code happily runs and gives you complete garbage, which is exactly what you told it to do.

# C++ : Values and Variables

- Now we move to variables
- A variable is similar to a mathematical variable:

```
int x;
```

- However, there is a difference : this has a specific register that it occupies. There is some piece of hardware somewhere in memory that is allocated to “x”. This memory will have an “**address**” and a “**value**”
- What’s in it? Well, print and see:

```
int x;
std::cout << x << std::endl;
```

- We compile and run:

```
$ g++ hello.cc -o a.out
$ ./a.out
0
```

Variable x has address “0x1010” →

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x9ab38 |
| 0x1010 | 0x0000 |
| 0x1011 | 0x1003 |

# C++ : Values and Variables

- Great! C++ sets this to something sensible, like “zero”!

- Great, right?

# C++ : Values and Variables

- WRONG! This is ALSO compiler dependent.

- The C++ standard does NOT specify what the initial value is for any given variable, so it is technically allowed to be filled with random garbage.

- So this is technically the second runtime error!
  - Even though the program is executing, there is no guarantee it will execute the same on a different platform
  - It may even be right ACCIDENTALLY
  - There is not even a guarantee that the program will execute the same on a different DAY

Variable x could be at “0x1011” →

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x9ab38 |
| 0x1010 | 0x0000 |
| 0x1011 | 0x1003 |

# C++ : Values and Variables

- So, moral of the story : you need to INITIALIZE EVERYTHING in C++. Some compilers will not even let you use variables without initialization.

- Here is the better code :

```
int x(0);
std::cout << x << std::endl;
```

- Note the initialization of x as “x(0)”
- The parentheses indicate that this is an ARGUMENT to a FUNCTION (more on that later)
- Can also use an equal-sign, the **“assignment operator”**:

```
int x=0;
std::cout << x << std::endl;
```

- Either works.

# C++ : Values and Variables

- So now when we compile and run, we get the same output as before, but this time, it is GUARANTEED

```
$ g++ hello.cc -o a.out
$ ./a.out
0
```

- Accept no substitutes! Your code must be guaranteed!
- So you will have something like this:

Before assignment

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x9ab38 |
| 0x1010 | 0x0000 |
| 0x1011 | 0x1003 |

"x" →

After assignment

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x0000 |
| 0x1010 | 0x0000 |
| 0x1011 | 0x1003 |

"x" →

# C++ : Values and Variables

- So let's come back to the assignment operator
- The value "x" in memory is a specific register with the value "0" stored on it.
- If you change the value of x to be "x = 10", the MEMORY address remains the same, but the VALUE within it changes to "10"

Before

| | Memory Address | Value |
|---|---|---|
| | 0x1000 | 0x04a45 |
| "x" → | 0x1001 | 0x0000 |

After

| | Memory Address | Value |
|---|---|---|
| | 0x1000 | 0x04a45 |
| "x" → | 0x1001 | 0x 000a |

- Specifically, the sequence is a "COPY" :
  - Take the value "10" (1010b)
  - Copy it into the register that "x" is allocated (address 0x1001)
  - Ensure that the value "10" is also preserved

# C++ : Values and Variables

- Why am I taking care to be pedantic about this?

- Because in C++, “assignment” can be very generic, so you could have:

  FacebookData x = facebook.copy();

- This is terrible, terrible, terrible, because it actually copies ALL of the data in facebook three times.
  - Copies FacebookData to a local register in “facebook.copy()”.
  - Copies that register to a global scope register
  - Copes that copied register to “x”

- Ugh, that could take awhile.

# C++ : Values and Variables

- We can assign values to variables over and over:

```
int x=0;
std::cout << x << std::endl;
x = 10;
std::cout << x << std::endl;
x = 20;
std::cout << x << std::endl;
```

- Two important things :
  - You don't need the "int" in front of "x" except for the first time
  - If you DO put an "int" in front of "x", it changes the place in memory, and removes the first one from consideration:

int x = 0;  // once

"x" →

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x0000 |

int x = 0;  // again

"x" →

| Memory Address | Value |
|---|---|
| 0x1000 | 0x04a45 |
| 0x1001 | 0x0000 |

# C++ : Values and Variables

- What if we try to copy the other way around:

```
int x=0;
std::cout << x << std::endl;
1 = x;
```

- We get a compilation error:

```
$ g++ hello.cc -o a.out
hello.cc: In function ‘int main()’:
hello.cc:6: error: lvalue required as left operand of
assignment
```

- More on that later

# C++ : Values and Variables

- Assignment can also work with variables :

```
int x=0, y=1;
std::cout << x << std::endl;
std::cout << y << std::endl;

x = y;
std::cout << x << std::endl;
std::cout << y << std::endl;
```

- Now what does this do?

```
$ ./a.out
0
1
1
1
```

# C++ : Values and Variables

- What is happening?

```
int x=0, y=1;
```

| | Memory Address | Value |
|---|---|---|
| x | 0x1000 | 0x0000 |
| y | 0x1001 | 0x0001 |

Put the value “0” into “x”
Put the value “1” into “y”

```
x = y;
```

| | Memory Address | Value |
|---|---|---|
| x | 0x1000 | 0x0001 |
| y | 0x1001 | 0x0001 |

Copy the value from “y”
place it into the value for “x”

# C++ : Identifiers

- We have now seen that there is a special word in C++ : “int”.
- What about others?
- They are called “identifiers”
- It is a special term in the language that cannot be used as a variable name

- We will go through these as we go along

| | | | |
|---|---|---|---|
| alignas | decltype | namespace | struct |
| alignof | default | new | switch |
| and | delete | noexcept | template |
| and_eq | double | not | this |
| asm | do | not_eq | thread_local |
| auto | dynamic_cast | nullptr | throw |
| bitand | else | operator | true |
| bitor | enum | or | try |
| bool | explicit | or_eq | typedef |
| break | export | private | typeid |
| case | extern | protected | typename |
| catch | false | public | union |
| char | float | register | unsigned |
| char16_t | for | reinterpret_cast | using |
| char32_t | friend | return | virtual |
| class | goto | short | void |
| compl | if | signed | volatile |
| const | inline | sizeof | wchar_t |
| constexpr | int | static | while |
| const_cast | long | static_assert | xor |
| continue | mutable | static_cast | xor_eq |

# C++: Doubles and Floats

- We've seen the double and float representations
- To assign double and float values, the syntax of "3.14" is by default a double, unless you append "f" at the end ("3.14f")
- Typically use "float" unless you need higher precision

# C++: Bools

- Logical true and false values are stored in a 1-bit type called a “bool” (as in, “boolean”, like, “boolean logic”)

- Can be either true or false:

```
bool b = true;
b = false;
```

- This can be used for flow of control (more later)

# C++: Constants

- So if we have a variable, what if we want it to remain… umm… constant?

- Syntax in C++ is “const” :

```
const double PI = 3.14159;
```

- Can be used like a variable, except it cannot be reassigned. Try to change the value in your code:

```
PI = 2.0;
```

- You get a compilation error:

```
$ g++ hello.cc -o a.out
hello.cc: In function ‘int main()’:
hello.cc:5: error: assignment of read-only variable ‘PI’
```

# C++: Constants

- Good programming practice : whenever possible, make something “const”
- This avoids bugs by accidental mis-assignment

- Brings up the concept of **“Principle of least privilege”**
  - Your code should only be ABLE to adjust the information it needs

- In the context of “const”, this means to intentionally take away privilege by declaring things “const” when possible.

# C++: Characters

- Moving on from ints, floats, and doubles, the next data type is a “character” (“char”)

- The “char” is meant to hold single characters (letters, numbers, punctuation, etc)
  - Only needs to be 1 byte long to store every possible character on earth
  - Stored as a number
- Universal code to convert to a character (ASCII)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | '\0' | 16 | — | 32 | ' ' | 48 | '0' | 64 | @@ | 80 | 'P' | 96 | ' ' | 112 | 'p' |
| 1 | — | 17 | — | 33 | '!' | 49 | '1' | 65 | 'A' | 81 | 'Q' | 97 | 'a' | 113 | 'q' |
| 2 | — | 18 | — | 34 | '"' | 50 | '2' | 66 | 'B' | 82 | 'R' | 98 | 'b' | 114 | 'r' |
| 3 | — | 19 | — | 35 | '#' | 51 | '3' | 67 | 'C' | 83 | 'S' | 99 | 'c' | 115 | 's' |
| 4 | — | 20 | — | 36 | '$' | 52 | '4' | 68 | 'D' | 84 | 'T' | 100 | 'd' | 116 | 't' |
| 5 | — | 21 | — | 37 | '%' | 53 | '5' | 69 | 'E' | 85 | 'U' | 101 | 'e' | 117 | 'u' |
| 6 | — | 22 | — | 38 | & | 54 | '6' | 70 | 'F' | 86 | 'V' | 102 | 'f' | 118 | 'v' |
| 7 | '\a' | 23 | — | 39 | '\'' | 55 | '7' | 71 | 'G' | 87 | 'W' | 103 | 'g' | 119 | 'w' |
| 8 | '\b' | 24 | — | 40 | '(' | 56 | '8' | 72 | 'H' | 88 | 'X' | 104 | 'h' | 120 | 'x' |
| 9 | '\t' | 25 | — | 41 | ')' | 57 | '9' | 73 | 'I' | 89 | 'Y' | 105 | 'i' | 121 | 'y' |
| 10 | '\n' | 26 | — | 42 | '*' | 58 | ':' | 74 | 'J' | 90 | 'Z' | 106 | 'j' | 122 | 'z' |
| 11 | — | 27 | — | 43 | '+' | 59 | ';' | 75 | 'K' | 91 | '[' | 107 | 'k' | 123 | '{' |
| 12 | '\f' | 28 | — | 44 | ',' | 60 | '<' | 76 | 'L' | 92 | '\\' | 108 | 'l' | 124 | '\|' |
| 13 | '\r' | 29 | — | 45 | '-' | 61 | '=' | 77 | 'M' | 93 | ']' | 109 | 'm' | 125 | '}' |
| 14 | — | 30 | — | 46 | '.' | 62 | '>' | 78 | 'N' | 94 | '^' | 110 | 'n' | 126 | '~' |
| 15 | — | 31 | — | 47 | '/' | 63 | '?' | 79 | 'O' | 95 | '_' | 111 | 'o' | 127 | — |

# C++: Characters

- Syntax to declare a single character is within SINGLE quotes :

```
char ch = 'c';
```

- Can then output :

```
std::cout << ch << std::endl;
```

- There are a few special characters:

  - `'\n'`—the newline character
  - `'\r'`—the carriage return character
  - `'\b'`—the backspace character
  - `'\a'`—the "alert" character (causes a "beep" sound or other tone on some systems)
  - `'\t'`—the tab character
  - `'\f'`—the formfeed character
  - `'\0'`—the *null* character (used in C strings, see Section 11.2.6)

# C++: Enumerations

- The next type is an enumeration (“enum”)

- These are basically aliases for integers, but can increase clarity:

```
enum Weight{
    Light, Medium, Heavy
};
```

- Well, USUALLY, anyway

```
// Conway enum:
enum {
  True=true,
  AlternativeTrue=false
};
```