# From python to C++

COMP120

# Learning Outcomes

By the end of this session, you should be able to:

- Outline the key differences between programming in Python and C++
- Explain important concepts associated with C++ programming
- Apply knowledge of coding in Python to coding in C++

# Lecture Outline

- Basic language structure
- Differences to python
- POD data types

# What is C++?

- Bjarne Stroustrup describes C++ the language he invented as
- “C++ is a general purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

# Python vs C++

- A Python script is executed through an interpreter
- C++ must be compiled into an executable to run.
- Python is dynamically typed, C++ is statically typed
- Python has built in garbage collection, C++ does not.
- In Python, variables are in scope even outside the loops in which they are first instantiated. C++ uses { } to define scope
- White space is semantically important in Python (not so much in C++)
- C++ uses an abstraction called a pointer to handle memory.

# The inevitable helloWorld.py

```
#!/usr/bin/python
import sys

def main(argv=None) :
  print "Hello␣World!"

if __name__ == "__main__":
    sys.exit(main())
```

- We can either run invoking the python interpreter
- or chmod +x and ./helloWorld.py

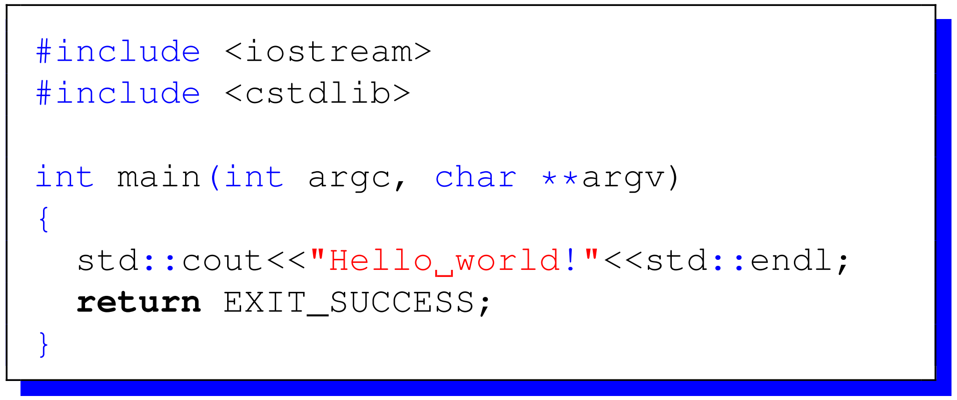
# The inevitable helloWorld.cpp

```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv)
{
  std::cout<<"Hello␣world!"<<std::endl;
  return EXIT_SUCCESS;
}
```

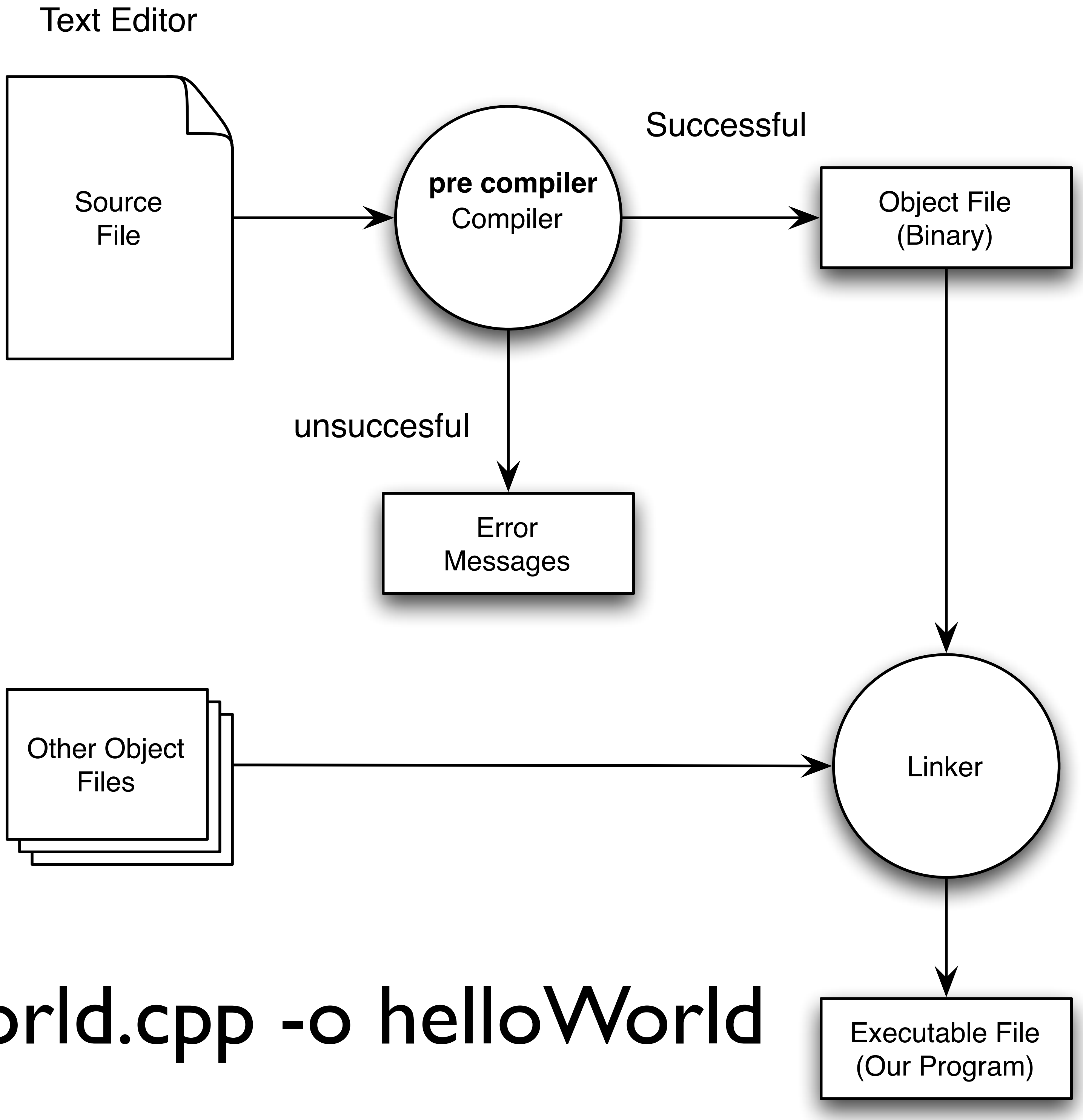
clang++ -Wall -g helloWorld.cpp -o helloWorld

g++ -Wall -g helloWorld.cpp -o helloWorld

# The Compilation Process



```
#include <iostream>
#include <cstdlib>

int main(int argc, char **argv)
{
  std::cout<<"Hello␣world!"<<std::endl;
  return EXIT_SUCCESS;
}
```



helloWorld.cpp

clang++-Wall -g helloWorld.cpp -o helloWorld

# clang++

- clang is (just one) c language compiler from the llvm (Low Level Virtual Machine) project
- The clang command is invoked on the command line and passed a series of command line options to determine how the compiler works
- If we have a single source file we can combine the compilation and linking stages in one go

| flag | usage |
|---|---|
| -Wall | turn on all warnings |
| -g | enable debugging output |
| -o | output to file name (else a.out is used) |

# clang++ vs g++

- clang++ is a modern C++ compiler based on the llvm architecture.
- It has the best error reporting and diagnostics of the two compilers
- both support (to different extents) c++ 11 however versions must be checked
- some ABI elements are compatible however mixing both compilers is usually problematic
- However this is also true of different version of the same compiler.

# variable declarations in C/C++

- In C and C++ we can declare variables using the syntax

```
<variable type> <variable identifier>;
<variable type> <var1>,<var2> ... <var n>;
```

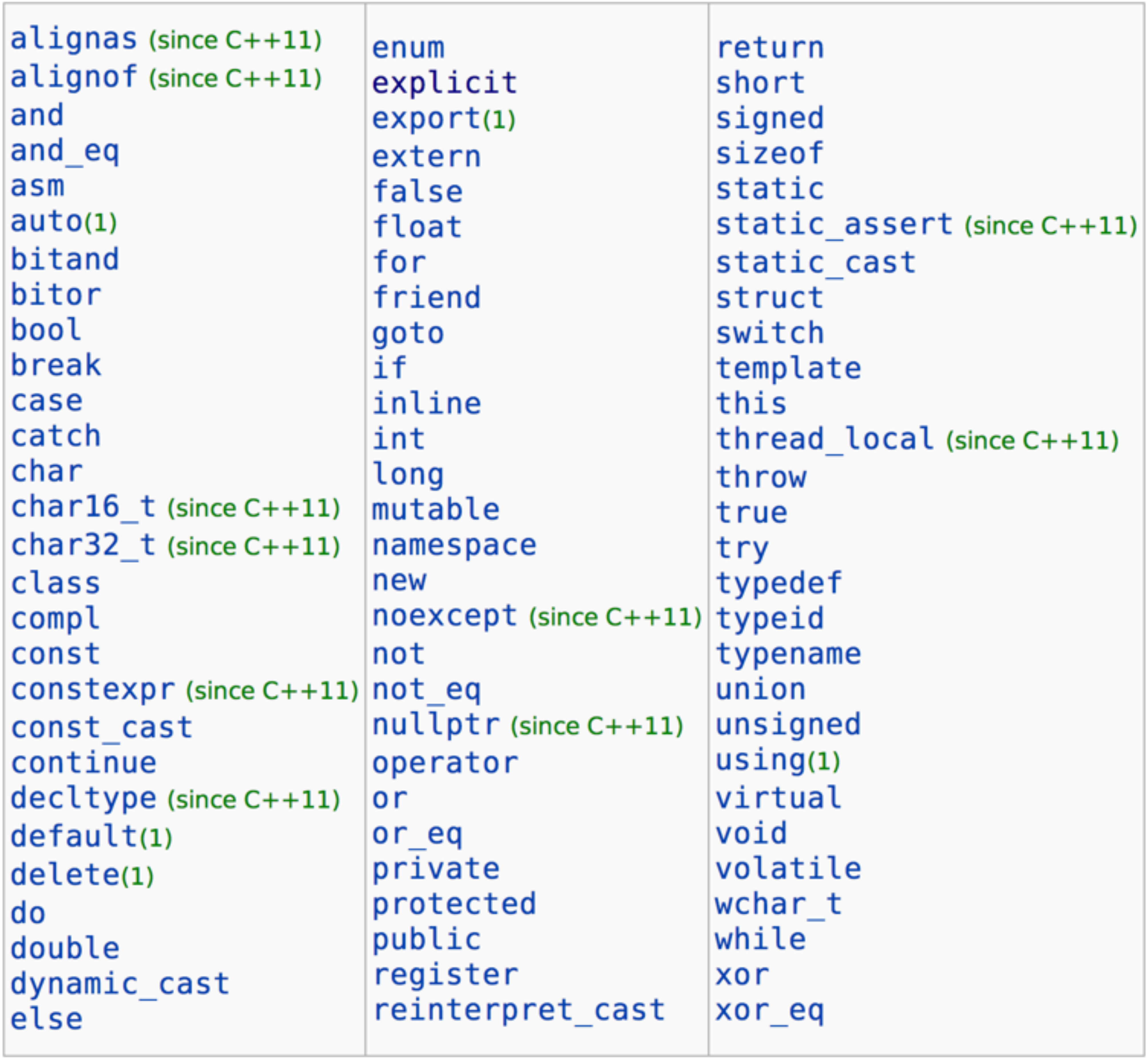
- Where variable type indicates one of the C data types
- identifier is a valid name for a variable

# valid variable names

- The following rules must be applied to C/C++ variable names
  - must not begin with a number
  - spaces are not allowed in names
  - Only letters digits and _ are valid characters
  - C++ keywords are not allowed

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

letter =
"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|
"w"|"x"|"y"|"z"|

"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|
"W"|"X"|"Y"|"Z";


start char = "letter" | "_";

variable name = start char , {digit} | {letter};
```

# C++ reserved words



alignas (since C++11)
alignof (since C++11)
and
and_eq
asm
auto(1)
bitand
bitor
bool
break
case
catch
char
char16_t (since C++11)
char32_t (since C++11)
class
compl
const
constexpr (since C++11)
const_cast
continue
decltype (since C++11)
default(1)
delete(1)
do
double
dynamic_cast
else
enum
explicit
export(1)
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept (since C++11)
not
not_eq
nullptr (since C++11)
operator
or
or_eq
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_assert (since C++11)
static_cast
struct
switch
template
this
thread_local (since C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using(1)
virtual
void
volatile
wchar_t
while
xor
xor_eq

# C++ reserved words

- The reserved words are the core part of the language.

- Since they are used by the language, these keywords are not available for re-definition or overloading.

- Attempting this will cause a compiler error.

- This is the same for most programming languages.

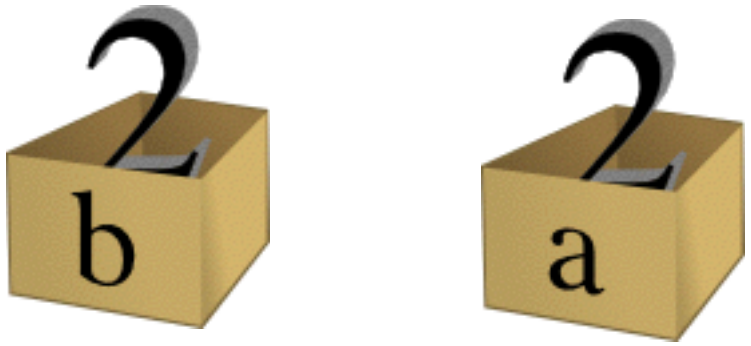
# Python and C++ Variables

- Most languages have variables for example int a=1;
- This effectively puts the value in a box
- Assigning another value to the same variable replaces the contents of the box: a=2;

# Python and C++ Variables

- Assigning one variable to another makes a copy of the value and puts it in the new box:

- int b = a;

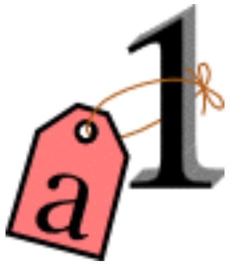


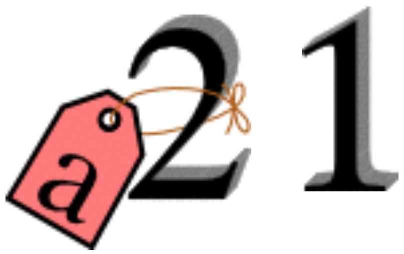
- "b" is a second box, with a copy of integer 2.

- Box "a" has a separate copy.

# Python and C++ Variables

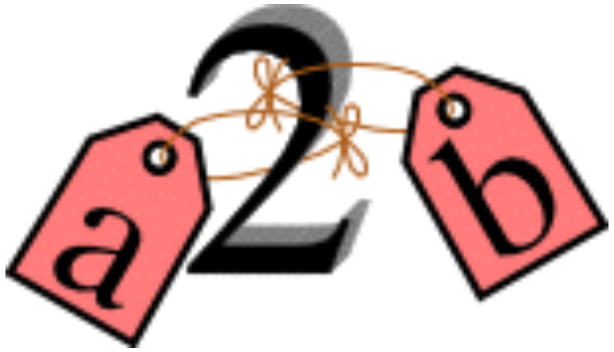
- In Python, a "name" or "identifier" is like a parcel tag (or name tag) attached to an object.





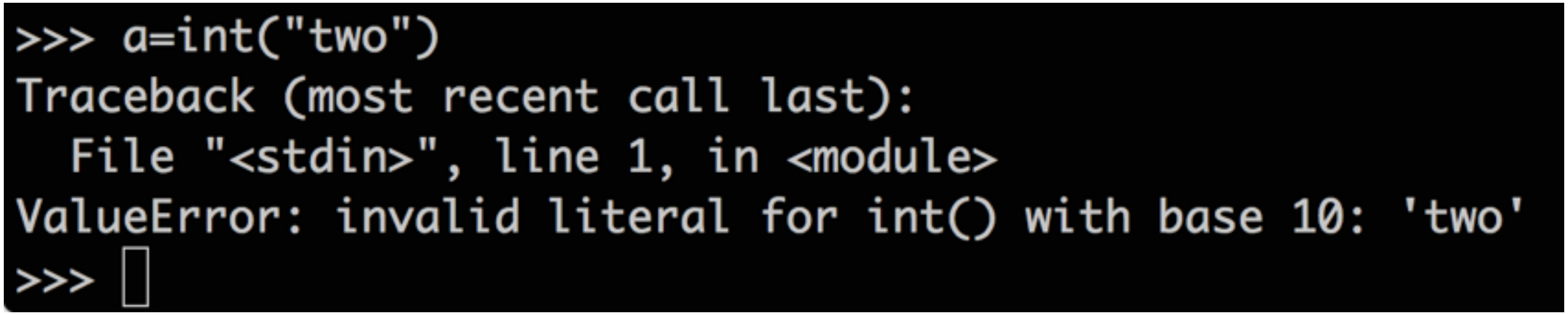
# Python and C++ Variables

- If we assign one name to another, we're just attaching another name tag to an existing object:
- b=a



# Dynamic typing

- A program is dynamically typed when the majority of type checking is done at run time rather than compile time.
- This is done by python
- This is flexible as it allowing programs to generate types and functionality based on run-time data
- This may result in runtime errors



```
>>> a=int("two")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'two'
>>> 
```

# static typing

- A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

- The most widely used statically typed languages are not formally type safe.

- We can circumvent this at runtime (coercion / type casting)

- C/C++ is a statically typed language

# POD Types

- Plain old data (or PODS) are the basic built in data types
- They can also be collections of other types including structures and classes
- The rules to determine a POD and non-POD type are quite complex in places
- I will come back to some of these later when we introduce classes
- For now all the following basic types are POD

# integer data type

- In C / C++ we can specify an integer using the int keyword.
- The range of an integer is dependant upon the machine architecture but is usually a whole 16, 32 or 64-bit (2, 4 or 8 bytes, respectively) addressable word.
- By default the int data type is signed (can be positive or negative)
- Typical range is –2147483648 to +2147483647

# example

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int a=10;
  int b=20;
  printf("a+b␣=␣%d␣\n",a+b);

  int aBigNumber = 4294967295;
  int one= 1;

  printf("aBigNumber+one␣=␣%d␣\n",aBigNumber+one);


  return EXIT_SUCCESS;
}
```

# ordinal data types

- The ordinal data types in C can be either signed or unsigned.
- C gives the programmer the following ordinal data types
- char, short int, long int
- each can be pre-fixed with the keyword unsigned

# ordinal data types

| Data type | Description |
|---|---|
| char | Small data type only needs 1 byte / 8 bits of memory to store. |
| short int | Integer data type half the size of an integer |
| int | Integer data type, size dependent upon platform using it |
| long int | Integer data type twice the size of the int data type |

# sizeof()

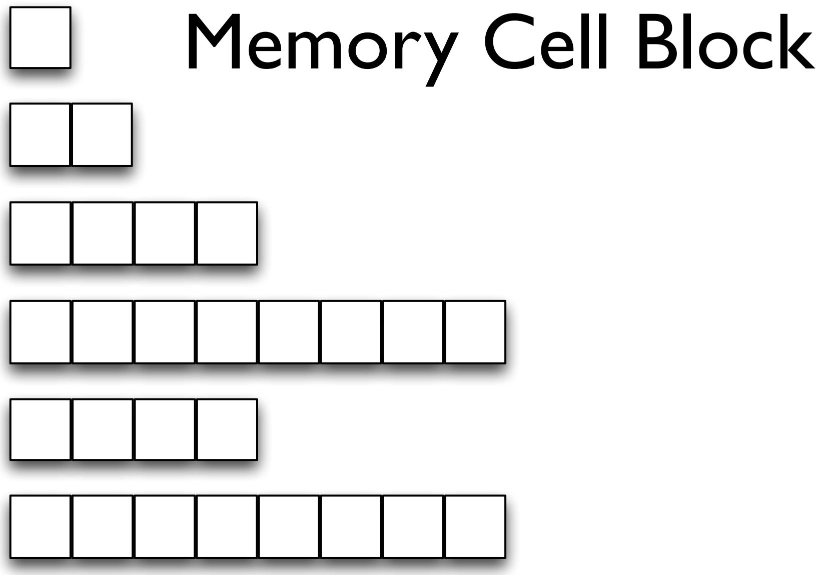
- In C/C++ sizeof is a unary operator that must be implemented by the developer of the compiler
- it appears as a C/C++ function when we use it but will return the size in bytes of the data type passed to it.
- The following program demonstrates sizeof

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("sizeof(char)=␣%ld␣\n",sizeof(char));
  printf("sizeof(short␣int)=␣%ld␣\n",sizeof(short int));
  printf("sizeof(int)=␣%ld␣\n",sizeof(int));
  printf("sizeof(long␣int)=␣%ld␣\n",sizeof(long int));

  printf("unsigned␣versions\n");

  printf("sizeof(unsigned␣char)=␣%ld␣\n",sizeof(unsigned char));
  printf("sizeof(unsigned␣short␣int)=␣%ld␣\n",sizeof(unsigned short int));
  printf("sizeof(unsigned␣int)=␣%ld␣\n",sizeof(unsigned int));
  printf("sizeof(unsigned␣long␣int)=␣%ld␣\n",sizeof(unsigned long int));

  return EXIT_SUCCESS;
}
```

```
sizeof(char)= 1
sizeof(short int)= 2
sizeof(int)= 4
sizeof(long int)= 8
unsigned versions
sizeof(unsigned char)= 1
sizeof(unsigned short int)= 2
sizeof(unsigned int)= 4
sizeof(unsigned long int)= 8
```

```
#include <cstdio>
#include <iostream>

int main()
{
  std::cout<<"sizeof(char)=␣␣"<<sizeof(char)<<std::endl;
  std::cout<<"sizeof(short␣int)=␣␣"<<sizeof(short int)<<std::endl;
  std::cout<<"sizeof(int)=␣"<<sizeof(int)<<std::endl;
  std::cout<<"sizeof(long␣int)=␣"<<sizeof(long int)<<std::endl;
  std::cout<<"sizeof(float)=␣"<<sizeof(float)<<std::endl;
  std::cout<<"sizeof(double)=␣"<<sizeof(double)<<std::endl;

  std::cout<<"unsigned␣versions\n"<<std::endl;

  std::cout<<"sizeof(unsigned␣char)=␣"<<sizeof(unsigned char)<<std::endl;
  std::cout<<"sizeof(unsigned␣short␣int)=␣"<<sizeof(unsigned short int)<<std::endl;
  std::cout<<"sizeof(unsigned␣int)=␣"<<sizeof(unsigned int)<<std::endl;
  std::cout<<"sizeof(unsigned␣long␣int)=␣"<<sizeof(unsigned long int)<<std::endl;

  return EXIT_SUCCESS;
}
```

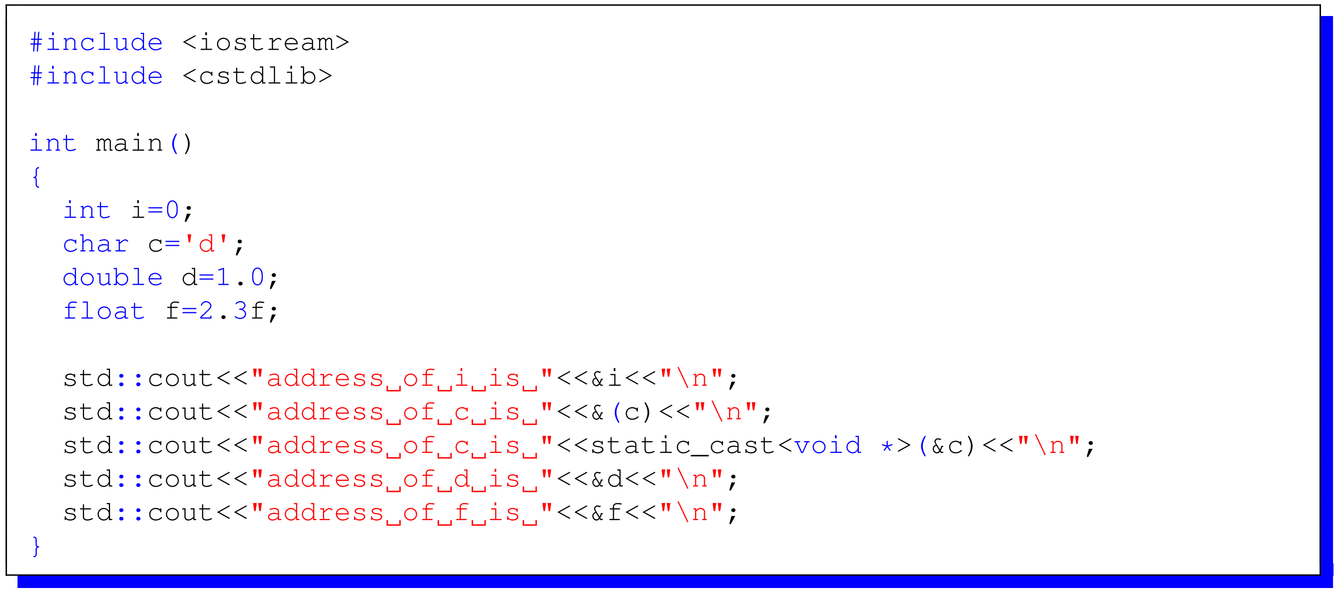
sizeof(char)=1
sizeof(short int)=2
sizeof(int)=4
sizeof(long int)=8
sizeof(float)=4
sizeof(double)=8



Memory Cell Block

# & the address operator

- Sometimes know as a reference operator will give us the memory address of the cell containing the value
- we can usually use & to print it out.



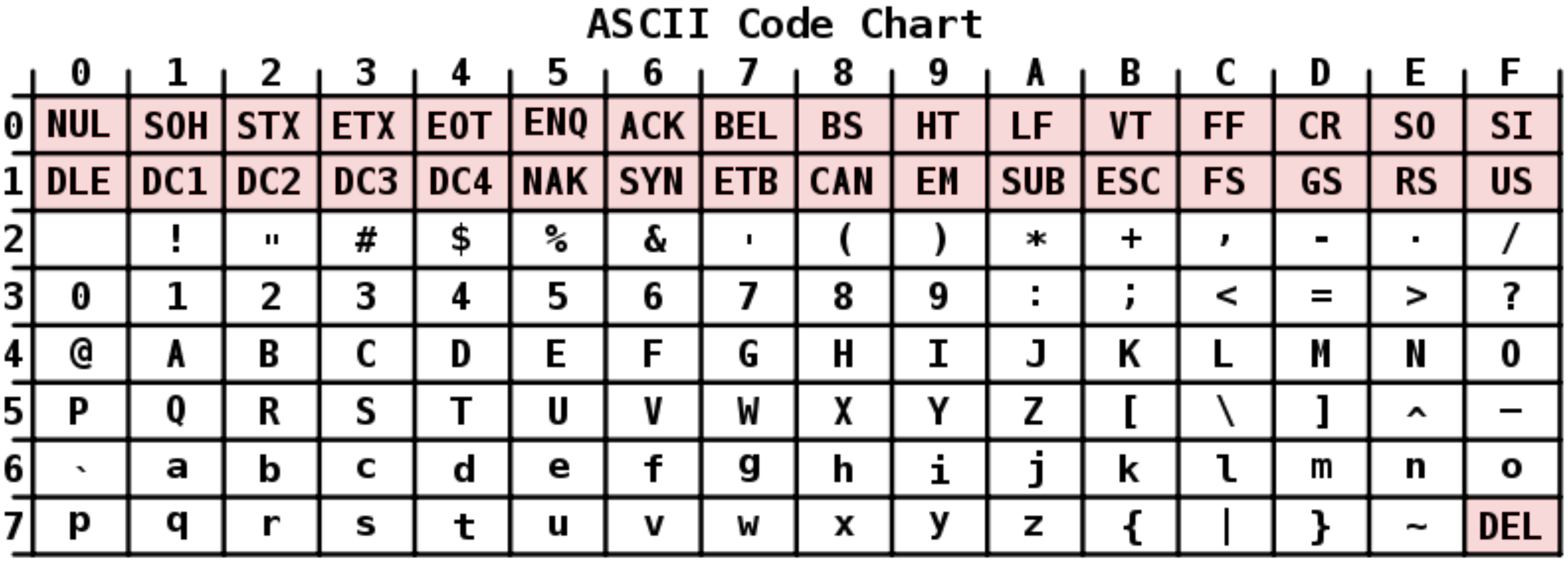
```
#include <iostream>
#include <cstdlib>

int main()
{
  int i=0;
  char c='d';
  double d=1.0;
  float f=2.3f;

  std::cout<<"address␣of␣i␣is␣"<<&i<<"\n";
  std::cout<<"address␣of␣c␣is␣"<<&(c)<<"\n";
  std::cout<<"address␣of␣c␣is␣"<<static_cast<void *>(&c)<<"\n";
  std::cout<<"address␣of␣d␣is␣"<<&d<<"\n";
  std::cout<<"address␣of␣f␣is␣"<<&f<<"\n";
}
```

```
address of i is 0x7fff52ebc72c
address of c is d
address of c is 0x7fff52ebc72b
address of d is 0x7fff52ebc720
address of f is 0x7fff52ebc71c
```

# char

- The char data type is useful for representing ASCII characters
- It usually takes up 1 byte and can represent either 0 to +255 (unsigned) or -128 to +128 (signed)
- Whilst this is used to store numeric values we can use the convenience single quote method to assign a char from a character as shown in the next program

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char a='z';
   printf("%c␣\n",a);
   a=42;
   printf("%c␣\n",a);

   return EXIT_SUCCESS;
}
```



ASCII Code Chart

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

http://upload.wikimedia.org/wikipedia/commons/4/4f/ASCII_Code_Chart.svg

# real numbers

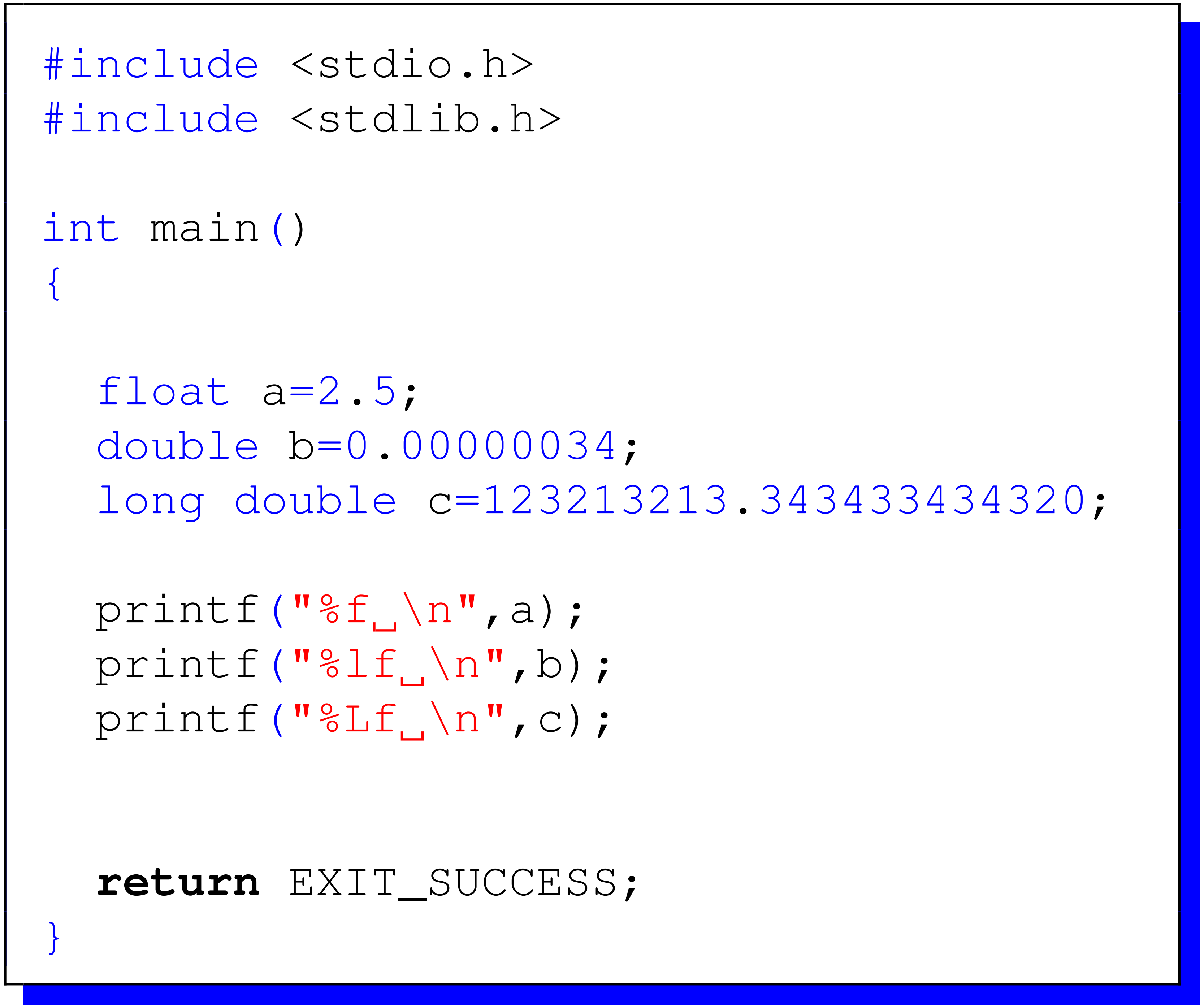
- In computing we use floating point data types to represent real numbers (numbers with a fractional part)
- These numbers are always approximations as we have to move the decimal.
- Numbers are, in general, represented approximately to a fixed number of significant digits and scaled using an exponent.

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

# real numbers in C

- C has two real data types float and double
- the long prefix may be used with double to increase the precision

| Type Specifiers | Precision (decimal digits) | | Exponent range | |
|---|---|---|---|---|
| | Minimum | IEEE 754 | Minimum | IEEE 754 |
| float | 6 | 7.2(24 bits) | ±37 | ±38 (8 bits) |
| double | 10 | 15.9(53 bits) | ±37 | ±307(11 bits) |
| long double | 10 | 34.0(113 bits) | ±37 | ±4931 (15 bits) |



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

  float a=2.5;
  double b=0.00000034;
  long double c=123213213.343433434320;

  printf("%f␣\n",a);
  printf("%lf␣\n",b);
  printf("%Lf␣\n",c);


  return EXIT_SUCCESS;
}
```

```
2.500000
0.000000
123213213.343433
```

Note truncated printf output

need to use %n.nf and specify decimal places to print e.g. %.8lf

# Arithmetic expressions

- Most programs are algorithmic in nature which means we have to do some maths
- The table below shows the available arithmetic operators

| Operator | Meaning | Examples |
|---|---|---|
| + | addition | 5 + 2 is 7<br>5.0 + 2.0 is 7.0 |
| - | subtraction | 5 - 2 is 3<br>5.0-2.0 is 3.0 |
| * | multiplication | 5*2 is 10<br>5.0*2.0=10.0 |
| / | division | 5/2 is 2<br>5.0/2.0 is 2.5 |
| % | remainder (modulus) | 5%2 is 1 |

# The / Operator

- When applied to two positive integers the division operator computes the integral part of the result dividing its first operand by its second

- For example

```
7.0 / 2.0 is 3.5
7 / 2 is 3
299.0 / 100.0 is 2.99 (double value)
299 / 100 is 2  (integer value)
```

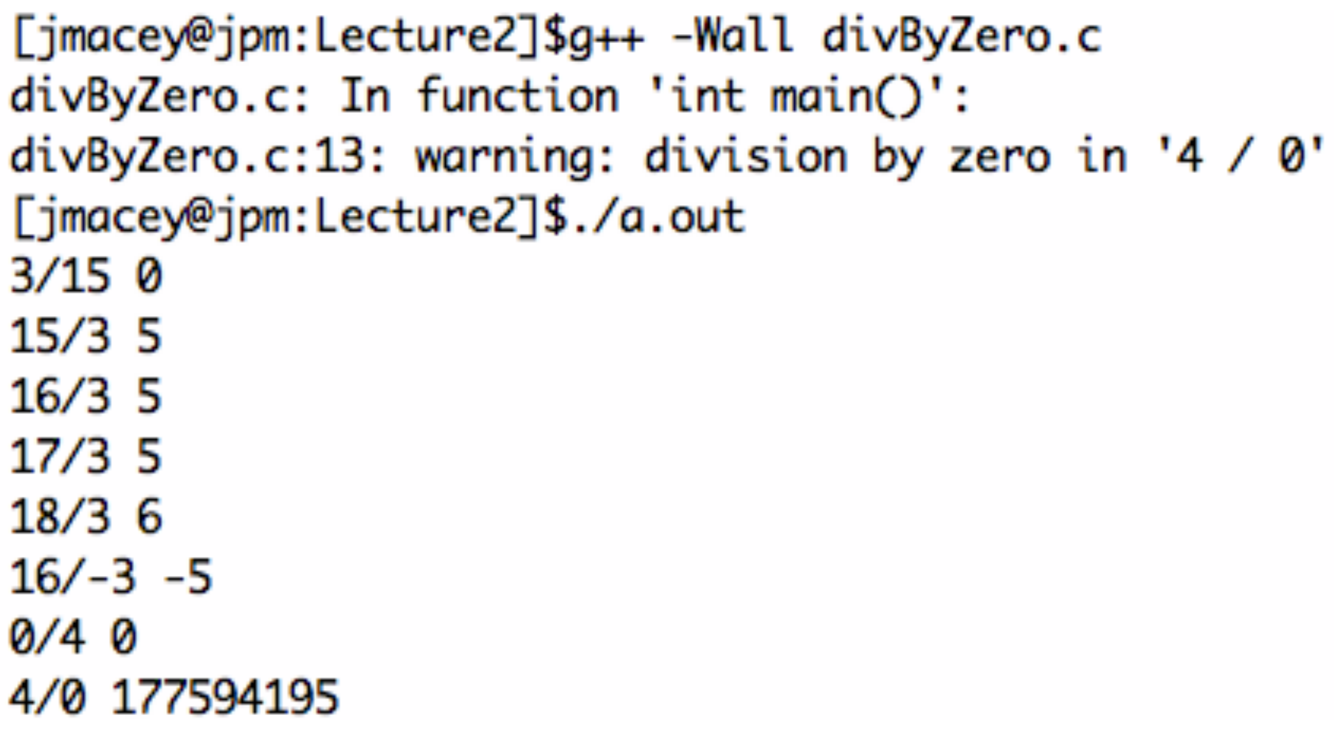
- If the / Operator is used with a negative and positive integer, the results vary from one C implementation to another

- For this reason you should avoid division by -ve integers

# More on /

- It is also important not to do division by 0 as the program may crash, some modern compilers will try to warn of this as seen with the program opposite

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("3/15␣%d\n",3/15);
  printf("15/3␣%d\n",15/3);
  printf("16/3␣%d\n",16/3);
  printf("17/3␣%d\n",17/3);
  printf("18/3␣%d\n",18/3);
  printf("16/-3␣%d\n",16/-3);
  printf("0/4␣%d\n",0/4);
  printf("4/0␣%d\n",4/0);

  return EXIT_SUCCESS;
}
```



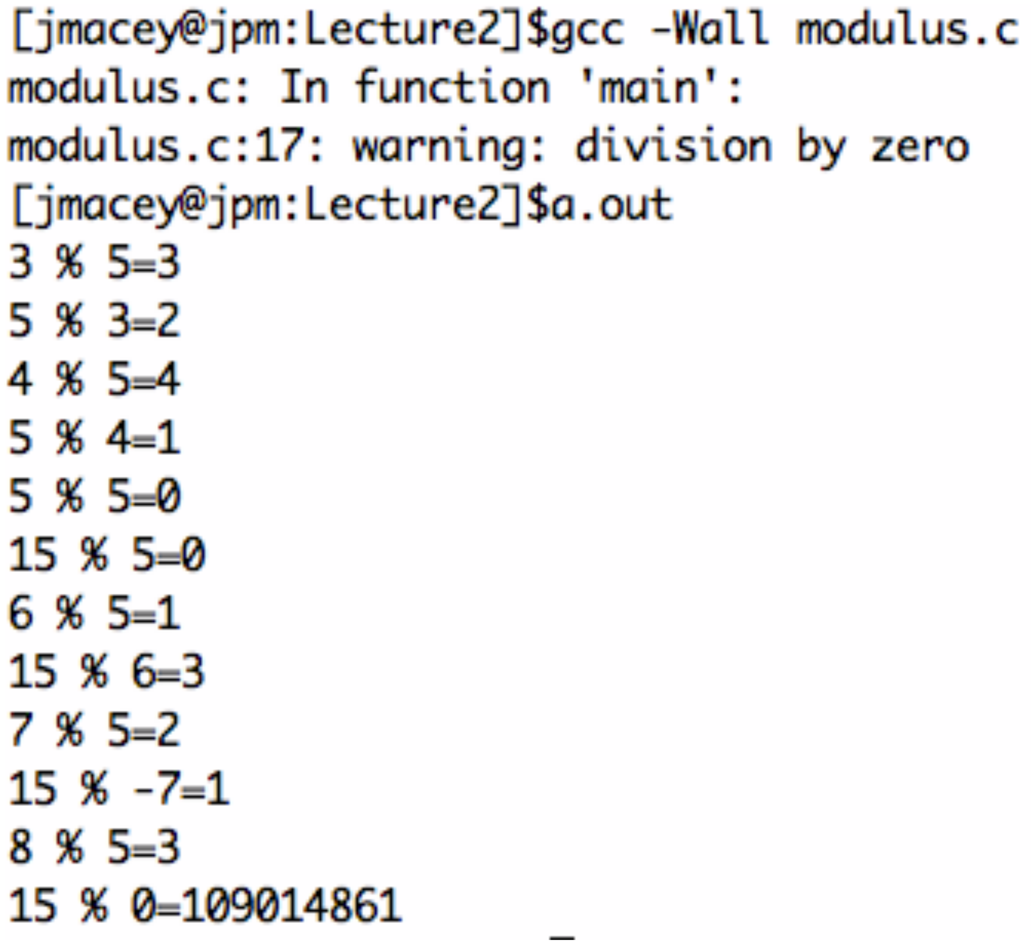
```
[jmacey@jpm:Lecture2]$g++ -Wall divByZero.c
divByZero.c: In function 'int main()':
divByZero.c:13: warning: division by zero in '4 / 0'
[jmacey@jpm:Lecture2]$./a.out
3/15 0
15/3 5
16/3 5
17/3 5
18/3 6
16/-3 -5
0/4 0
4/0 177594195
```

# The % (modulus) Operator

- The remainder operator (%) returns the integer remainder of the result of dividing the first operand with the second
- For example the value of 7 % 2 is 1
- The magnitude of m % n must always be lest than the division n

$$
\begin{array}{c}
7/2 \\
\downarrow \\
7 \div 2 = 3 \\
3 * 2 = 6 \\
\frac{6}{7-6} \qquad 7 \,\%\, 2 = 1
\end{array}
\qquad\qquad
\begin{array}{c}
299/100 \\
\downarrow \\
299 \div 100 = 2 \\
2 * 100 = 200 \\
\frac{200}{299-200} = 299 \,\%\, 100 = 99
\end{array}
$$

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("3␣%%␣5=%d\n",3%5);
  printf("5␣%%␣3=%d\n",5%3);
  printf("4␣%%␣5=%d\n",4%5);
  printf("5␣%%␣4=%d\n",5%4);
  printf("5␣%%␣5=%d\n",5%5);
  printf("15␣%%␣5=%d\n",15%5);
  printf("6␣%%␣5=%d\n",6%5);
  printf("15␣%%␣6=%d\n",15%6);
  printf("7␣%%␣5=%d\n",7%5);
  printf("15␣%%␣-7=%d\n",15%-7);
  printf("8␣%%␣5=%d\n",8%5);
  printf("15␣%%␣0=%d\n",15%0);

  return EXIT_SUCCESS;
}
```



```
[jmacey@jpm:Lecture2]$gcc -Wall modulus.c
modulus.c: In function 'main':
modulus.c:17: warning: division by zero
[jmacey@jpm:Lecture2]$a.out
3 % 5=3
5 % 3=2
4 % 5=4
5 % 4=1
5 % 5=0
15 % 5=0
6 % 5=1
15 % 6=3
7 % 5=2
15 % -7=1
8 % 5=3
15 % 0=109014861
```

# Data type of an expression

- There are certain rules to define the results of mixing data types
- For example

```
int a=10;
int b=23;
int c;

c=a+b; // will result in a integer value
```

- However if we mix the types we will get different results depending upon the receiving variables data type

```
double x;
int n;

x = 9 * 0.5; // will result in x = 4.5
n = 9 * 0.5; // will result in n = 4
```

# Expressions with Multiple Operators

- There are rules as to how expressions are evaluated
  - Parentheses Rule : All expressions in parentheses must be evaluated separately. Nested parenthesised expressions must be evaluated from the inside out, with the innermost expression evaluated first.
  - Operator precedence rule : Operators in the same expression are evaluated in the following order.

| | |
|---|---|
| `unary +, -` | `first` |
| `*, /, %` | `next` |
| `binary +,-` | `last` |

# Expressions with Multiple Operators

- Associativity Rule : Unary operators in the same sub-expression and at the same precedence levels (such as + and -) are evaluated right to left.
- Binary operators in the same sub-expression and the same precedence level (such as + and -) are evaluated left to right.
- To help avoid problems with the order of evaluation it is best to use parenthesis

```
x * y * z + a / b -c * d;

can be written

(x * y * z) + (a / b) - (c * d);
```

# Mathematical Formulas as C/C++ expressions

| Mathematical Formula | C Expression |
|---|---|
| $b^2 - 4ac$ | b * b - 4 * a * c |
| $a + b - c$ | a + b - c |
| $\frac{a+b}{c+d}$ | (a + b) / (c + d) |
| $\frac{1}{1+x^2}$ | 1 / (1 + x * x ) |
| a$\times - (b + c)$ | a * -(b + c) |

Notice that C has no equivalent to $x^2$so we have to evaluate it as x * x

If any other power is required the *pow(double x, double y)* function must be used to evaluate $x^y$

# References

- Sebesta R. W. 2002, Concepts of Programming Languages, 5th edition, Addison Wesley, International Ed

- Daniel D. McCracken and Edwin D. Reilly. 2003. Backus-Naur form (BNF). In Encyclopedia of Computer Science (4th ed.), Anthony Ralston, Edwin D. Reilly, and David Hemmendinger (Eds.). John Wiley and Sons Ltd., Chichester, UK 129-131.

- Hanly. J. R Koffman E. B. 1999, Problem Solving & Program Design in C, 3rd Edition, Addison Wesley, International Ed

# references

- http://www.stroustrup.com/C++.html
- http://en.cppreference.com/w/cpp/keyword
- http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
- Based on a talk by Jon Macey: http://nccastaff.bournemouth.ac.uk/jmacey/cppintro
- https://pdfs.semanticscholar.org/9ad1/030685050e949d1a3d6d92bababcbe075e07.pdf

# glossary

- **ABI** Application binary interface, the low level interface between modules such as OS and libraries. Usually machine code level.
- **API** Application programming interface usually a library / set of modules at source code level to programming different libraries