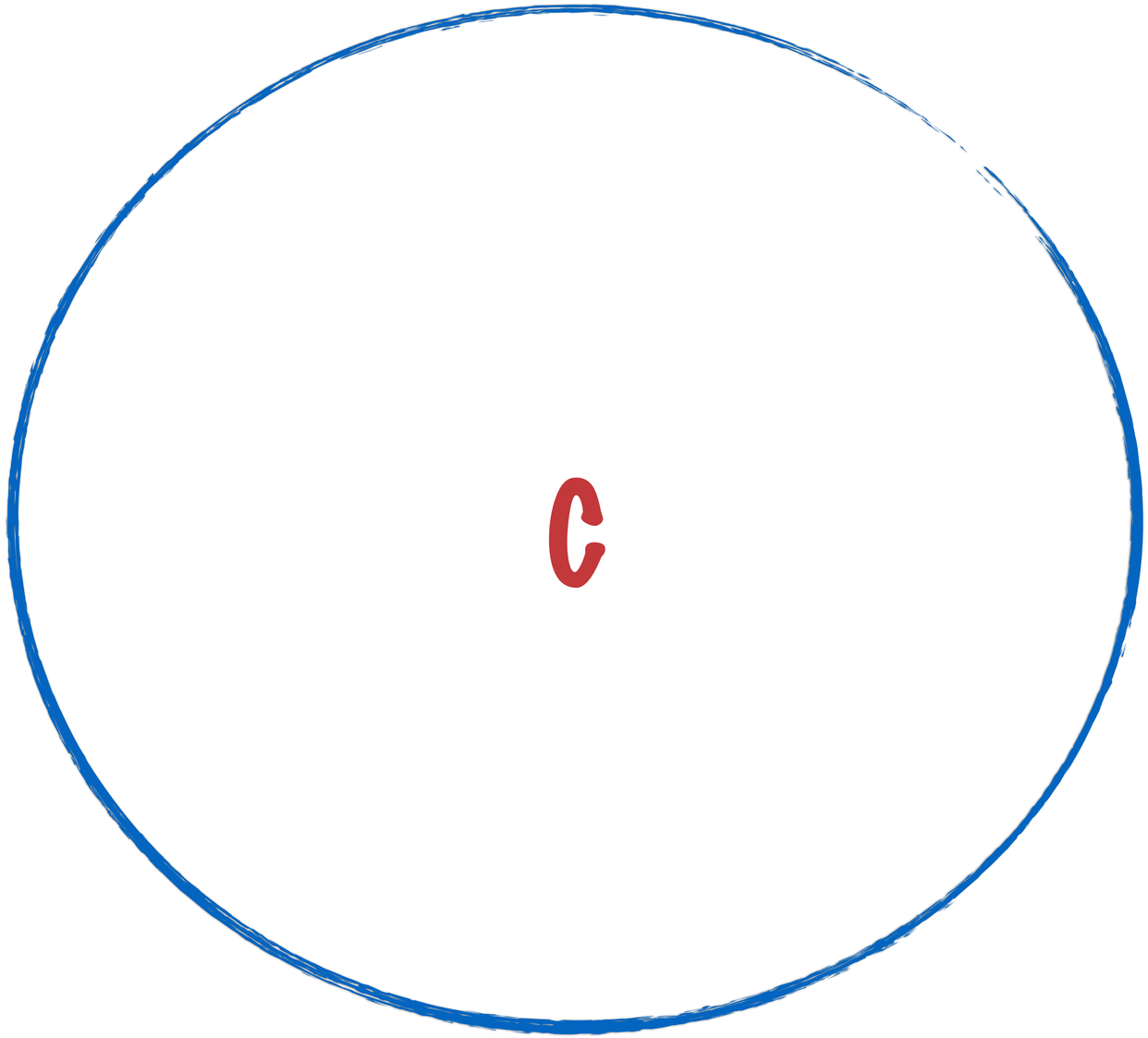
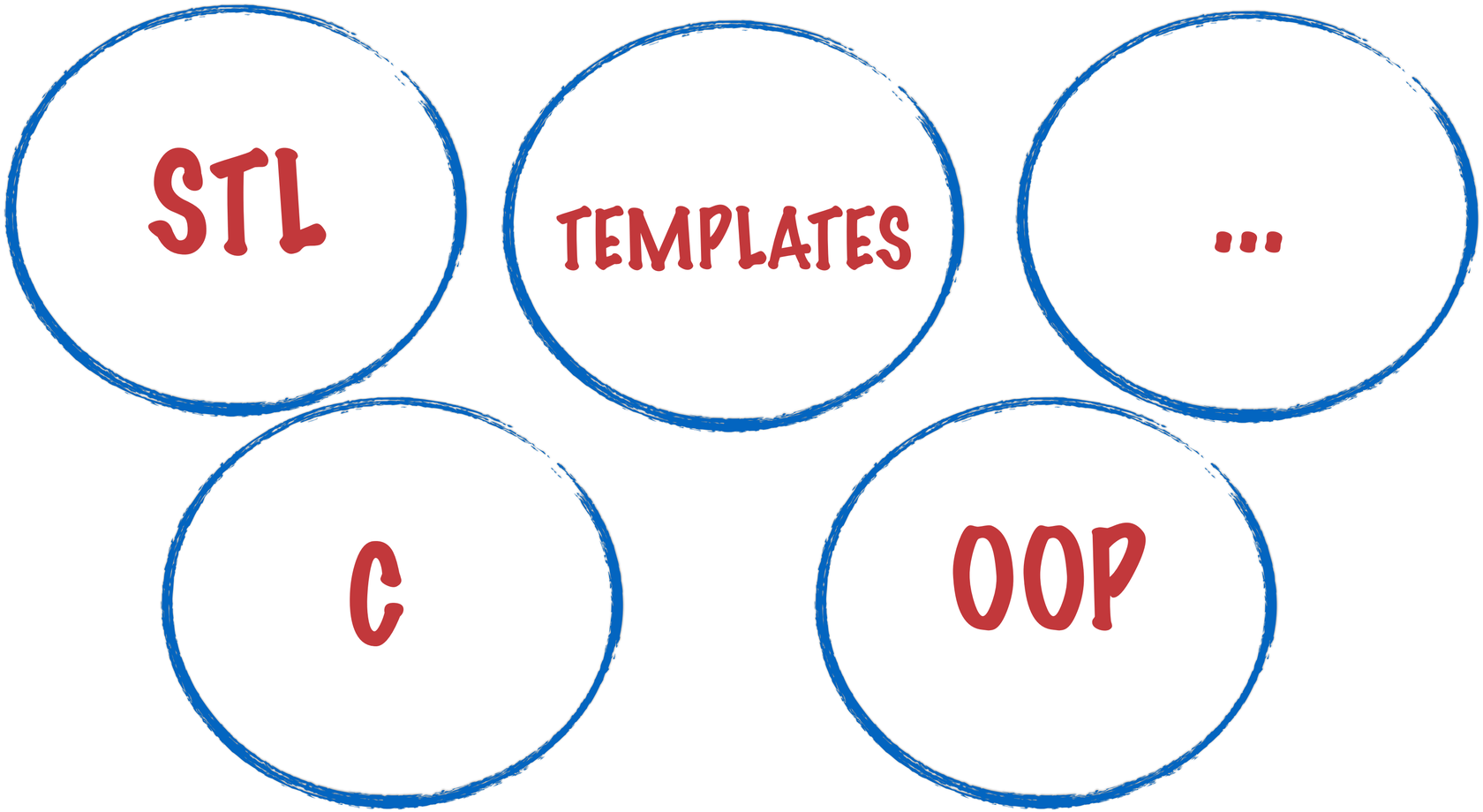
C++ AND C

C++ AND C

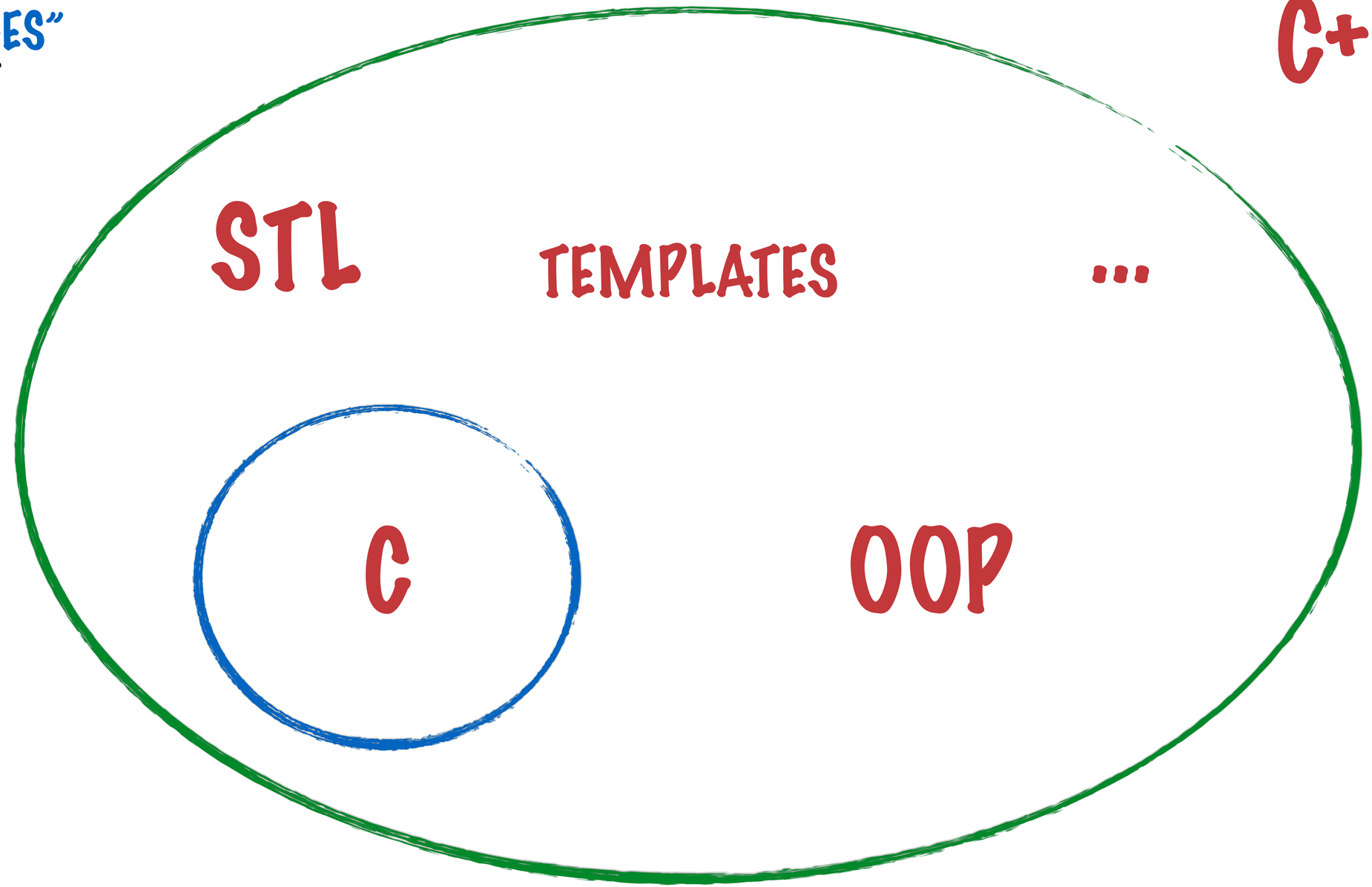


C++ AND C



C++ AND C

"VIEW C++ AS A FEDERATION OF LANGUAGES"
— SCOTT MYERS



"VIEW C++ AS A FEDERATION OF LANGUAGES"
— SCOTT MYERS

C++ AND C

C++ IS A SUPERSET OF C

BUT SOME PARTS OF C ARE BASICALLY RENDERED REDUNDANT BY NEW C++ FEATURES

C++ IS A SUPERSET OF C

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{ // C Translation: printf("Hi there! This is a C++ world.\n");
   cout << "Hi there! This is a C++ world" << endl;
   int someNumber;  //hold user input in this variable
   cout << "Please enter some number: ";
   cin >> someNumber;
   cout << "The number you entered was:" << someNumber << endl;
   return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    // C Translation: Standard output stream
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

// C Translation: send-to

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;   // C Translation:"\n"
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

// C Translation: /*hold user input in this variable*/

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber; // C Translation: scanf("%d", &someNumber);
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;  // C Translation: standard input stream
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;  // C Translation: get-from
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()  // C Translation (note the extra.h!): #include <stdio.h>
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

LET'S WRITE A REALLY SIMPLE C++ PROGRAM

- SAY HELLO TO THE USER
- PROMPT HER FOR A NUMBER
- THEN PRINT THAT NUMBER TO SCREEN.

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

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int someNumber;  //hold user input in this variable
    cout << "Please enter some number: ";
    cin >> someNumber;
    cout << "The number you entered was:" << someNumber << endl;
    return 0;
}
```

// C Translation: none!

EVEN THIS VERY SIMPLE PROGRAM
WOULD HAVE GIVEN YOU A FLAVOUR OF
SOME DIFFERENCES BETWEEN C AND C++

SCREEN IO

COMMENTS

NAMESPACES

COMMENTS

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

int main()
{
    cout << “Hi there! This is a C++ world” << endl;
    int someNumber;
    cout << “Please enter some number: “;
    cin >> someNumber;
    cout << “The number you entered was:“ << someNumber << endl;
    return 0;
}
```

// C Translation: /*hold user input in this variable*/

//hold user input in this variable

COMMENTS

// C-style comments :

```
/*hold user input in this variable*/
```

C comments are block comments, with start and end delimiters

// C++-style comments :

```
//hold user input in this variable
```

C++ comments are single line comments, end-delimiter is simply the end of line (carriage return)

You can also use C-style comments in C++

EVEN THIS VERY SIMPLE PROGRAM
WOULD HAVE GIVEN YOU A FLAVOUR OF
SOME DIFFERENCES BETWEEN C AND C++

SCREEN IO

COMMENTS

NAMESPACES

NAMESPACES

C HAS TWO TYPES OF DATA (VARIABLES)

GLOBAL

Defined outside all functions, and accessible to all functions

LOCAL

Defined inside a function (or a scope) and only visible inside that function (or scope)

NAMESPACES

C HAS TWO TYPES OF DATA (VARIABLES)

GLOBAL

Defined outside all functions, and accessible to all functions

LOCAL

Defined inside a function or scope, and only visible inside that function (or scope)

As the size and complexity of a program grows, global variables lead to naming conflicts

As the size and complexity of programs grows, global variables lead to naming conflicts

C++ MITIGATES THIS BY DIVIDING THE WORLD INTO: NAMESPACES

C++ MITIGATES THIS BY DIVIDING THE WORLD INTO: NAMESPACES

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

int main()
{
    cout << “Hi there! This is a C++ world” << endl;
    int someNumber;  //hold user input in this variable
    cout << “Please enter some number: “;
    cin >> someNumber;
    cout << “The number you entered was:“ << someNumber << endl;
    return 0;
}
```

This line indicates that we want to use functions and variables from the namespace std

By default, code is written inside a namespace called ‘std’, but programmers can and should define and stay within their own namespaces

C++ MITIGATES THIS BY DIVIDING THE WORLD INTO: NAMESPACES

```
namespace client_code {
  int someVariable = 100;

  int doSomething() {
  }
}
```

```
namespace utilities {
  char someVariable = 'a';

  int doSomethingElse() {
  }
}
```

C++ MITIGATES THIS BY DIVIDING THE WORLD INTO: NAMESPACES

By default, code is written inside a namespace called 'std', but programmers can and should define and stay within their own namespaces

how do you refer to a variable in another, different, namespace?

THE SCOPE RESOLUTION OPERATOR

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
  int a = -10; // This is the version of a defined inside namespace 'private_space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

the same variable, with three different scopes..

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

If no scope resolution operator (::) is used, the local version will be used only if it exists

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

If no scope resolution operator (::) is used, the local version will be used only if it exists

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;

   cout << "Please enter some number: ";
   cout << "Variable a (no local so global) = " << a << endl; //prints 10
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

If there is no local version then the global one in the **std** namespace (**::**) is used

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int a = 20;  //this is a local (inside main) variable, within namespace 'std'
    cout << "Please enter some number: ";
    cout << "Variable a (local) = " << a << endl; //prints 20
    cout << "Variable a (global) = " << ::a << endl; // prints 10
    cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
    return 0;
}
```

If the scope resolution operator (`::`) is used with a blank namespace, **std** is the default namespace, it's the global namespace

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

If the scope resolution operator (`::`) is used with a blank namespace, **std** is the default namespace, it's the global namespace

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

C++ searches first in the global namespace, then in any other namespaces which have been included - more on the inclusion later

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
    cout << "Hi there! This is a C++ world" << endl;
    int a = 20;  //this is a local (inside main) variable, within namespace 'std'
    cout << "Please enter some number: ";
    cout << "Variable a (local) = " << a << endl; //prints 20
    cout << "Variable a (global) = " << ::a << endl; // prints 10
    cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
    return 0;
}
```

If the scope resolution operator (`::`) is used with an explicit namespace (here `private_space`) that version is used

THE SCOPE RESOLUTION OPERATOR

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

int a = 10; // This is a 'global' (within namespace std) variable

namespace private_space {
int a = -10; // This is the version of a defined inside namespace 'private space'
}

int main()
{
   cout << "Hi there! This is a C++ world" << endl;
   int a = 20;  //this is a local (inside main) variable, within namespace 'std'
   cout << "Please enter some number: ";
   cout << "Variable a (local) = " << a << endl; //prints 20
   cout << "Variable a (global) = " << ::a << endl; // prints 10
   cout << "Variable a (private_space) = " << private_space::a << endl; // prints -10
   return 0;
}
```

If the scope resolution operator (`::`) is used with an explicit namespace (here `private_space`) that version is used

EVEN THIS VERY SIMPLE PROGRAM
WOULD HAVE GIVEN YOU A FLAVOUR OF
SOME DIFFERENCES BETWEEN C AND C++

SCREEN IO

COMMENTS

NAMESPACES

SCREEN IO

`cout` `clog` `cerr` `cin`

STANDARD OUTPUT STREAM - WRITE TO THIS STREAM TO WRITE TO SCREEN

use the `<<` operator

STANDARD INPUT STREAM - READ FROM HERE TO READ FROM KEYBOARD

use the `>>` operator

ERRM..ARE N'T THESE THE BITWISE SHIFT OPERATORS?

use the `<<` operator

use the `>>` operator

ERRM..ARE N'T THESE THE BITWISE SHIFT OPERATORS?

EXACTLY! BUT C++ ALLOWS SOMETHING MIND BENDING

OPERATOR OVERLOADING

OPERATOR OVERLOADING

THE SAME OPERATOR CAN DO DIFFERENT THINGS, DEPENDING ON THE CONTEXT

AND A PROGRAMMER CAN REDEFINE OPERATORS AT WILL

THE SAME OPERATOR CAN DO DIFFERENT THINGS, DEPENDING ON THE CONTEXT

OPERATOR OVERLOADING

AND A PROGRAMMER CAN REDEFINE OPERATORS AT WILL

```
struct ComplexNumber
{
   int realPart;
   int imaginaryPart;
};
struct ComplexNumber p1 = {3,4};
struct ComplexNumber p2 = {4,0};
struct ComplexNumber p2 = p1 + p2;
```

THE SAME OPERATOR CAN DO DIFFERENT THINGS, DEPENDING ON THE CONTEXT

OPERATOR OVERLOADING

AND A PROGRAMMER CAN REDEFINE OPERATORS AT WILL

```
struct ComplexNumber
{
   int realPart;
   int imaginaryPart;
};
struct ComplexNumber p1 = {3,4};
struct ComplexNumber p2 = {4,0};
struct ComplexNumber p2 = p1 + p2;
```

OPERATOR OVERLOADING

```
struct ComplexNumber
{
   int realPart;
   int imaginaryPart;
};
struct ComplexNumber p1 = {3,4};
struct ComplexNumber p2 = {4,0};
struct ComplexNumber p2 = p1 + p2;
```

THE SAME OPERATOR CAN DO DIFFERENT THINGS, DEPENDING ON THE CONTEXT

AND A PROGRAMMER CAN REDEFINE OPERATORS AT WILL

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF SOME DIFFERENCES BETWEEN C AND C++

SCREEN IO

OPERATOR OVERLOADING

COMMENTS

FUNCTION OVERLOADING ETC

NAMESPACES

FUNCTION OVERLOADING

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

```
double getArea(double radius)
{
   return 3.14 * radius * radius;
}
double getArea(double length, double breadth)
{
   return length * breadth;
}
```

SAME NAME, OVERLOADED FUNCTION

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

```
double getArea(double radius)
{
   return 3.14 * radius * radius;
}
double getArea(double length, double breadth)
{
   return length * breadth;
}
```

NUMBER, TYPE AND ORDER OF ARGUMENTS DIFFERENT!

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

```
double getArea(double radius)
{
  return 3.14 * radius * radius;
}
double getArea(double length, double breadth)
{
  return length * breadth;
}
double radius r = 5;
double squareSide = 3;

cout <<“Area of a circle of radius “<< radius << ” = “ << getArea(radius) << endl;
cout <<“Area of a square of length “<< squareSide <<“ =“
     <<getArea(squareSide,squareSide)<<endl;
```

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

FUNCTION #1

```
double getArea(double radius)
{
  return 3.14 * radius * radius;
}
```

FUNCTION #2

```
double getArea(double length, double breadth)
{
  return length * breadth;
}
```

```
double radius r = 5;
double squareSide = 3;

cout <<"Area of a circle of radius "<< radius << " = " << getArea(radius) << endl;
cout <<"Area of a square of length "<< squareSide <<" ="
     <<getArea(squareSide,squareSide)<<endl;
```

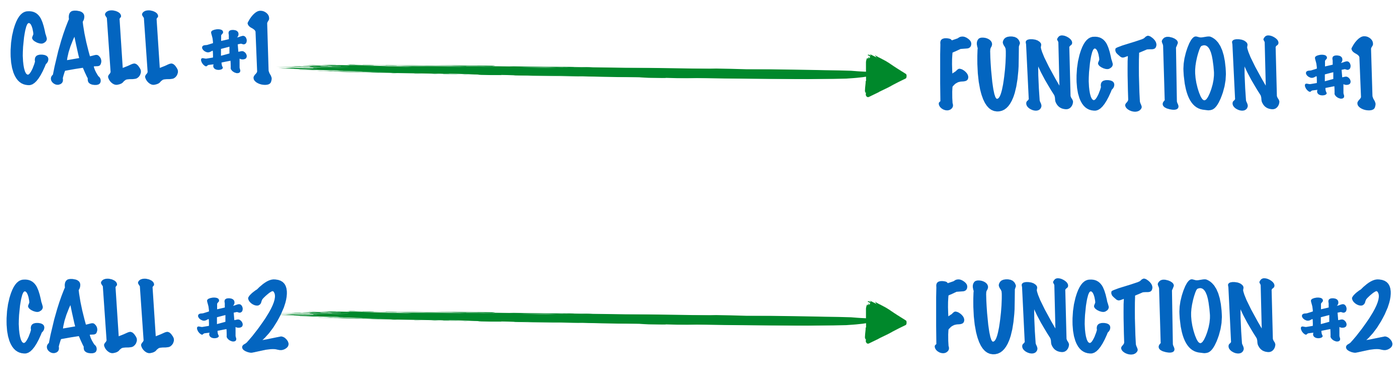
CALL #1

CALL #2

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS



THE C++ COMPILER FIGURES OUT WHICH VERSION TO CALL FROM THE PARAMETERS PASSED IN

PROGRAMMERS CAN DEFINE MULTIPLE FUNCTIONS WITH THE SAME NAME

FUNCTION OVERLOADING

EACH OVERLOADED VERSION HAS A DIFFERENT SIGNATURE: NUMBER, ORDER AND TYPE OF FUNCTION PARAMETERS

IN C++, FUNCTIONS CAN'T BE OVERLOADED ON THE RETURN TYPE!

FUNCTIONS
(OVERLOADING ETC)

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<“A point in 3-D space: x=“<<x<<“,y=“<<y<<“,z=“<<z<<endl;
}

print_3D_Coordinates(); // prints “x=0,y=0,z=0”
print_3D_Coordinates(10); // prints “x=10,y=0,z=0”
print_3D_Coordinates(10,20); // prints “x=10,y=20,z=0”
print_3D_Coordinates(10,20,30); // prints “x=10,y=20,z=30”
```

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

NO PARAMETERS SPECIFIED AT ALL - ALL DEFAULTS USED!

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

X = 0, Y = 0 , Z = 0

NO PARAMETERS SPECIFIED AT ALL - ALL DEFAULTS USED!

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}

print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

THIS GETS INTERESTING - ONE VALUE SPECIFIED..

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

THIS GETS INTERESTING - ONE VALUE SPECIFIED..

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

X = 10

THIS GETS INTERESTING - ONE VALUE SPECIFIED..

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

PROGRAMMERS CAN ALSO SPECIFY DEFAULT VALUES FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

X = 10 Y = 0, Z = 0

THIS GETS INTERESTING - ONE VALUE SPECIFIED..

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

DEFAULTS USED FOR MISSING VALUES

PROGRAMMERS CAN ALSO SPECIFY **DEFAULT VALUES** FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}

print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

SAME DEAL - TWO VALUES SPECIFIED..

PROGRAMMERS CAN ALSO SPECIFY DEFAULT VALUES FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

SAME DEAL - TWO VALUES SPECIFIED..

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

PROGRAMMERS CAN ALSO SPECIFY DEFAULT VALUES FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

X = 10, Y = 20

SAME DEAL - TWO VALUES SPECIFIED..

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

PROGRAMMERS CAN ALSO SPECIFY DEFAULT VALUES FOR FUNCTION PARAMETERS

IF THE CALLING CODE OMITS THOSE PARAMETERS, THE DEFAULT VALUE WILL BE USED

```
double print_3D_Coordinates(double x=0,double y=0,double z = 0)
{
  cout<<"A point in 3-D space: x="<<x<<",y="<<y<<",z="<<z<<endl;
}
```

X = 10, Y = 20 Z = 0

SAME DEAL - TWO VALUES SPECIFIED..

```
print_3D_Coordinates(); // prints "x=0,y=0,z=0"
print_3D_Coordinates(10); // prints "x=10,y=0,z=0"
print_3D_Coordinates(10,20); // prints "x=10,y=20,z=0"
print_3D_Coordinates(10,20,30); // prints "x=10,y=20,z=30"
```

ARGUMENT VALUES ARE FILLED IN FROM LEFT TO RIGHT..

DEFAULTS USED FOR MISSING VALUES

FUNCTION OVERLOADING AND DEFAULT VALUES CAN GET COMPLICATED

THE C++ COMPILER WILL FLAG ERRORS IF ANY AMBIGUITY CREEPS IN

FUNCTION OVERLOADING AND DEFAULT VALUES CAN GET COMPLICATED

THE C++ COMPILER WILL FLAG ERRORS IF ANY AMBIGUITY CREEPS IN

```
double getPerimeter(double radius,double PI = 3.1415)
{
  return 2 * PI * radius;
}
double getPerimeter(double side1, double side2, double side3)
{
  return side1 + side2 + side3;
}
```

FUNCTION OVERLOADING AND DEFAULT VALUES CAN GET COMPLICATED

THE C++ COMPILER WILL FLAG ERRORS IF ANY AMBIGUITY CREEPS IN

```
double getPerimeter(double radius,double PI = 3.1415)
{
  return 2 * PI * radius;
}
double getPerimeter(double side1, double side2, double side3)
{
  return side1 + side2 + side3;
}
```

THIS IS ALLOWED, WE CAN SPECIFY DEFAULT VALUES IN OVERLOADED METHODS

FUNCTION OVERLOADING AND DEFAULT VALUES CAN GET COMPLICATED

THE C++ COMPILER WILL FLAG ERRORS IF ANY AMBIGUITY CREEPS IN

THIS IS ALLOWED, WE CAN SPECIFY DEFAULT VALUES IN OVERLOADED METHODS

```
double getPerimeter(double radius,double PI = 3.1415)
{
  return 2 * PI * radius;
}
double getPerimeter(double side1, double side2, double side3)
{
  return side1 + side2 + side3;
}

double radius r = 5;
double triangleSide = 3;

cout <<"Perimeter of a circle of radius "<< radius << " = " << getPerimeter(radius) << endl;
cout <<"Perimeter of a triangle of sides"<< triangleSide <<" , "
     <<getPerimeter(triangleSide, triangleSide, triangleSide)<<endl;
```

THIS WORKS, NO AMBIGUITY ABOUT WHICH METHOD WILL BE CALLED

FUNCTION OVERLOADING AND DEFAULT VALUES CAN GET COMPLICATED

THE C++ COMPILER WILL FLAG ERRORS IF ANY AMBIGUITY CREEPS IN

```
double getPerimeter(double radius,double PI = 3.1415)
{
  return 2 * PI * radius;
}
double getPerimeter(double side1, double side2, doubl
{
  return side1 + side2 + side3;
}
double getPerimeter(double length, double breadth)
{
  return 2 * (length + breadth);
}

double squareSide = 3;

cout <<"Perimeter of a square of sides"<< squareSide <<" ="
     <<getPerimeter(squareSide, squareSide)<<endl;
```

WHICH METHOD DOES THIS REFER TO?

AMBIGUOUS! COMPILE ERROR!

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF

SOME DIFFERENCES BETWEEN C AND C++

REFERENCES

OPERATOR OVERLOADING

SCREEN IO

COMMENTS

FUNCTION OVERLOADING AND DEFAULT VALUES

NAMESPACES

REFERENCES

HOW WOULD WE WRITE, IN C, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

REFERENCES

HOW WOULD WE WRITE, IN C, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

REFERENCES

HOW WOULD WE WRITE, IN C, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

THE FUNCTION BODY MUST TAKE IN AND DEREFERENCE POINTERS

REFERENCES

HOW WOULD WE WRITE, IN C, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

THE FUNCTION BODY MUST TAKE IN AND DEREFERENCE POINTERS

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

THE CALLING CODE MUST USE THIS DIFFICULT '&' TO PASS IN THE ADDRESS OF THE UNDERLYING MEMORY LOCATION OF THE VARIABLE

REFERENCES

HOW WOULD WE WRITE, IN C, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

TO MAKE SUCH COMMON USE-CASES SIMPLE, C++ INTRODUCED THE IDEA OF

TO MAKE SUCH COMMON USE-CASES SIMPLE, C++ INTRODUCED THE IDEA OF

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

```
// writing the function
double swap(int& a,int& b)
{
  int temp = a;
  a = b;
  b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

```
a = 5, b = 10
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

```
a = 10, b = 5
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

REFERENCES

HOW WOULD WE WRITE, IN C++, A FUNCTION THAT SWAPS THE VALUES OF 2 VARIABLES?

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

REFERENCES ARE IN FACT, POINTERS - C++ JUST MAKES THEIR SYNTAX FAR EASIER TO USE

REFERENCES

REFERENCES ARE IN FACT, POINTERS - C++ JUST MAKES THEIR SYNTAX FAR EASIER TO USE

NO MEMORY ALLOCATION, MEMORY LEAKS, OR EVEN NULLS (A REFERENCE CAN NEVER BE NULL!)

REFERENCES

RULE #1: A REFERENCE MUST ALWAYS BE INITIALISED

```
int& x = 3;
```

X

```
int& x;
x = 3;
```

REFERENCES

RULE #2: A REFERENCE CAN NEVER BE RE-ASSIGNED

X

~~int& x = 4;~~

~~x = 3;~~

REFERENCES

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

REFERENCES

RULE #4: REFERENCES CAN NEVER BE NULL

REFERENCES

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

REFERENCES

RULE #6:

C++ Standard 8.3.2/4:

> There shall be no references to references, **no arrays of references**, and no pointers to references.

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF SOME DIFFERENCES BETWEEN C AND C++

REFERENCES

OPERATOR OVERLOADING

SCREEN IO

CONST

COMMENTS

FUNCTION OVERLOADING AND DEFAULT VALUES

NAMESPACES

CONST

WITH REFERENCES, IT IS VERY EASY TO INADVERTENTLY CHANGE THE VALUE OF A VARIABLE WITHOUT MEANING TO

CONST

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

WITH REFERENCES, IT IS VERY EASY TO INADVERTENTLY CHANGE THE VALUE OF A VARIABLE WITHOUT MEANING TO

IN OUR EXAMPLE OF SWAPPING 2 VALUES, THERE IS NOTHING TO EXPLICITLY WARN THE CALLING CODE THAT THE VALUES OF A AND B ARE CHANGED BY THE FUNCTION SWAP

IN ORDER TO PREVENT UNWANTED OR INADVERTENT MODIFICATIONS, C++ INTRODUCED A NEW KEYWORD

WITH REFERENCES, IT IS VERY EASY TO INADVERTENTLY CHANGE THE VALUE OF A VARIABLE WITHOUT MEANING TO

IN ORDER TO PREVENT UNWANTED OR INADVERTENT MODIFICATIONS, C++ INTRODUCED A NEW KEYWORD

CONST

ANY VARIABLE CAN BE MARKED AS CONST, AND ANY MODIFICATION TO ITS VALUE WILL YIELD AN ERROR

```
const double PI = 3.1415;
```

CONST

IN C++, YOU SHOULD ENTIRELY AVOID USING `#DEFINE` TO DEFINE CONSTANTS - IT LACKS THE SCOPE GRANULARITY OF `const`

```
const double PI = 3.1415;
```

X ~~`#DEFINE PI 3.1415;`~~

CONST

BACK TO OUR SWAP EXAMPLE - WOULD THIS WORK?

```
// writing the function
double swap(const int& a,const int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

ANY VARIABLE CAN BE MARKED AS CONST, AND ANY MODIFICATION TO ITS VALUE WILL YIELD AN ERROR

CONST

BACK TO OUR SWAP EXAMPLE - WOULD THIS WORK?

```
// writing the function
double swap(const int& a,const int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

X

ANY VARIABLE CAN BE MARKED AS CONST, AND ANY MODIFICATION TO ITS VALUE WILL YIELD AN ERROR

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF SOME DIFFERENCES BETWEEN C AND C++

REFERENCES

OPERATOR OVERLOADING

SCREEN IO

CONST

bool

COMMENTS

FUNCTION OVERLOADING AND DEFAULT VALUES

NAMESPACES

bool

IN C, THERE WAS NO EXPLICIT BOOLEAN TYPE

0 EVALUATED TO **FALSE**, AND ANY NON-ZERO VALUE EVALUATED TO **TRUE**

C++ HAS AN EXPLICIT `bool` TYPE, WITH VALUES `true` AND `false`

bool

IN C, THERE WAS NO EXPLICIT BOOLEAN TYPE

0 EVALUATED TO `FALSE`, AND ANY NON-ZERO VALUE EVALUATED TO `TRUE`

C++ HAS AN EXPLICIT `bool` TYPE, WITH VALUES `true` AND `false`

DON'T MIX `bool` VALUES `true` AND `false` WITH THE C `TRUE` AND `FALSE`. ONLY USE THE C++ `bool`

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF SOME DIFFERENCES BETWEEN C AND C++

REFERENCES

OPERATOR OVERLOADING

SCREEN IO

CONST

COMMENTS

bool

FUNCTION OVERLOADING AND DEFAULT VALUES

NAMESPACES

EVEN THIS VERY SIMPLE PROGRAM WOULD HAVE GIVEN YOU A FLAVOUR OF SOME DIFFERENCES BETWEEN C AND C++

THIS LIST DOES NOT EVEN START WITH THE MOST IMPORTANT DIFFERENCE OF THEM ALL -

OBJECT-ORIENTED PROGRAMMING

OBJECT-ORIENTED PROGRAMMING

C++ IS A FUNDAMENTALLY OBJECT-ORIENTED LANGUAGE, C IS NOT.

OBJECTS VERSUS STRUCTS

NEW/DELETE VERSUS MALLOC/FREE

STRING VERSUS CHAR*

OBJECT-ORIENTED PROGRAMMING

C++ IS A FUNDAMENTALLY OBJECT-ORIENTED LANGUAGE, C IS NOT.

BUT WE WILL GET TO THAT IN GOOD TIME :-)