C++ for C Programmers
(In 20 Steps)

Adapted from
Ian Parberry

Contents of This Lecture

1. #include

- There is a new way to **#include** libraries (the old method still works although the compiler may complain).
- The **.h** extension is not used any more, and the names of standard C libraries are written beginning with a **c**.
- In order for the program to use these libraries correctly **using namespace** std; has to be added like this:

Include Example

```
using namespace std;
#include <cmath> //This is math.h
int main(){
  double a;
  a = 1.2;
  a = sin(a);
  printf("%f\n", a);
  return 0;
}
```


2. Stream Input and Output

```
using namespace std;
#include <iostream>
int main(){
  int a;
  char s[100];
  cout << "Sample program." << endl;
  cout << endl;
  cout << "Type your age: ";
  cin >> a;
  cout << "Type your name: ";
  cin >> s;
```


Streams Example

```
  cout << "Type your name: ";
  cin >> s;
  cout << endl;
  cout << "Hello " << s << ", you're ";
  cout << a << " old." << endl;
  cout << endl << "Bye!" << endl;
  return 0;
}
```

Output

```
This is a sample program.
Type your age : 42
Type your name: Arthur
Hello Arthur, you're 42 old.
Bye!
```

3. Variable Declaration

Variables can be declared any place in the code. Scope lasts until the end of the block.

```
int main(){
  double a = 0.1415;
  a = a + 3.0;
  double c;
  c = a/4;
  cout << "c contains : " << c << endl;
  return 0;
}
```

Output

```
c contains 0.785375
```

7

More on Local Scope

- Try to use this feature to make your source code more readable and not to mess it up.
- Like in C, variables can be encapsulated between { } blocks.
- Then they are local in scope to the zone encapsulated between the { and }.
- Whatever happens with such variables inside the encapsulated zone will have no effect outside the zone.

8

4. Variable Initialization

A variable can be initialized by a calculation involving other variables:

```
double a = 12 * 3.25;
double b = a + 1.112;
```

9

5. Loop Control Variables

C++ allows you to declare the control variable to be local to a loop:

```
for(int i = 0; i < 4; i++){
  //stuff
}
```

You may be tempted to use `i` after the loop. Some early C++ compilers allow this. Modern ones don't.

10

6. Global Variables

A global variable can be accessed even if there is local variable with the same name.

```
using namespace std;
#include <iostream>
double a = 128;
int main(){
  double a = 256;
  cout << "Local a is " << a << endl;
  cout << "Global a is " << ::a << endl;
  return 0;
}
```

Output

```
Local a is 256
Global a is 128
```

11

7. Inline Functions

- If a function contains just straight-line code, it can be declared **inline**.
- This means its code will be inserted everywhere the function is used. That's somewhat like a macro.
- The main advantage is the program will be faster.
- A small drawback is it will be bigger, because the full code of the function was inserted everywhere it is used.

```
inline double hypotenuse(double a, double b){
  return sqrt(a*a + b*b);
}
```

12

8. Default Parameters

```
using namespace std;
#include <iostream>
double test (double a, double b = 7){
  return a - b;
}
int main (){
  cout << test (14, 5) << endl;
  cout << test (14) << endl;
  return 0;
};
```

Output

```
9
7
```

13

9. Function Overload

Different functions can have the same name provided something allows the compiler to distinguish between them: number or type of parameters.

```
double test(double a, double b){
  return a + b;
}
int test(int a, int b){
  return a - b;
}
```

14

```
int main(){
  double m = 7, n = 4;
  int k = 5, p = 3;
  cout << test(m, n) << ", " << test(k, p) << endl;
  return 0;
}
```

```
int main(){
  double m = 7, n = 4;
  int k = 5, p = 3;
  cout << test(m, n) << ", " << test(k, p) << endl;
  return 0;
}
```

Output

```
11, 2
```

15

10. Memory Allocation

- The keywords **`new`** and **`delete`** can be used to allocate and deallocate memory.
- They are cleaner than the functions **`malloc`** and **`free`** from standard C.
- **`new`**`[]` and **`delete`**`[]` are used for arrays.

16

```
double *d;
d = new double;
*d = 45.3;
cout << "Type a number: ";
cin >> *d;
*d = *d + 5;
cout << "Result: " << *d << endl;
delete d;
```

Output

```
Type a number: 6
Result: 11
```

17

```
int n = 30;
d = new double[n];
for(int i=0; i<n; i++)
  d[i] = i;
delete [] d;

char *s;
s = new char[100];
strcpy (s, "Hello!");
delete [] s;
```

18

11. Functions in Structs

```
struct vector{
  double x;
  double y;
  double surface(){
    double s;
    s = x*y;
    if(s<0) s = -s;
    return s;
  }
};
```

19

```
int main(){
  vector a;
  a.x = 3; a.y = 4;
  cout << "Surface is " << a.surface() << endl;
  return 0;
}
```

Output

```
Surface is 12
```

20

12. The Class

- A `class` is a `struct` that can keep data `private`.
- Only the functions of the class can access `private` data.
- Data that is not `private` can be made `public`.
- Here are two examples of a `class` definition.
  1. The first behaves exactly the same way as the `struct` example above because the class data `x` and `y` are defined as public.
  2. The second keeps `x` and `y` private.

21

```
class vector{
  public:
    double x;
    double y;
    double surface(){
      double s;
      s = x*y;
      if(s < 0) s = -s;
      return s;
    }
};
```

22

```
class vector1{
  private:
    double x;
    double y;
  public:
    double surface(){
      double s;
      s = x*y;
      if(s < 0) s = -s;
      return s;
    }
};
```

23

Declaring and Instantiating a Class

```
vector myvector;
myvector.x = 3.1415;
double s = myvector.surface();
```

```
vector* myvector;
myvector = new vector;
myvector -> x = 3.1415;
double s = myvector -> surface();
delete myvector;
```

24

Declaring and Instantiating a Class

```
vector1 myvector;
myvector.x = 3.1415; //is not legal
double s = myvector.surface();
```

```
vector1* myvector;
myvector = new vector1;
myvector -> x = 3.1415; //is not legal
double s = myvector -> surface();
delete myvector;
```

25

13. Constructors and Destructors

- The constructor is automatically called whenever an instance of a class is created by declaration or by `new`.
- The destructor is automatically called whenever an instance of a class is destroyed by end of scope or by `delete`.
- The constructor will initialize the variables of the instance, do some calculations, allocate some memory for the instance,… whatever is needed.
- The destructor cleans up afterwards, most importantly, it must free allocated memory!

26

```
class vector2{
  private:
    double x;
    double y;
    char* name;
  public:
    vector2(); //constructor
};
```

```
vector2::vector2(){
  x = y = 0;
  name = NULL;
}
```

27

```
class vector3{
  private:
    double x;
    double y;
    char* name;
  public:
    vector3(char* s); //constructor
    ~vector3(); //destructor
};
```

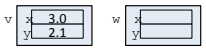
```
vector3::vector3(char* s){ //constructor
  x = y = 0;
  name = new char[strlen(s)+1];
  strcpy(name, s);
}
vector3::~vector3(){ //destructor
  delete [] name;
}
```

28

The Copy Constructor

No Problems.

```
vector v, w;
v.x = 3; v.y = 2.1;
```

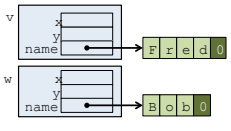



```
w = v;
```

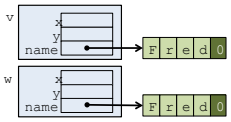
29

What We Want

```
vector3 v("Fred");
vector3 w("Bob");
```



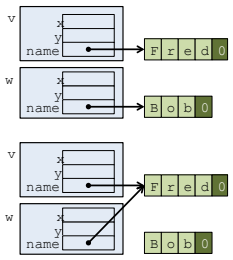

```
w = v;
```



30

What We Get

```
vector3 v("Fred");
vector3 w("Bob");
```




```
w = v;
```


31

More Later

- We need to override the assignment operation and write a copy constructor.
- We'll cover this later when we get to more advanced C++ topics.
- For now, be aware of it as a "gotcha".

32

14. Separating Code and Header

```
//Instead of doing this:
using namespace std;
#include <iostream>
class vector{
  public:
    double myfn(){
      //some code goes here
    }
}
int main(){
  vector k;
  cout << "Myfn returns: " << k.myfn() << endl;
  return 0;
}
```

33

Use a Function Prototype

```
using namespace std;
#include <iostream>
class vector{
  public:
    double myfn(); //This is a function prototype
};
double vector::myfn(){
  //some code goes here
}
int main(){
  vector k;
  cout << "Myfn returns: " << k.myfn() << endl;
  return 0;
}
```

34

Main.cpp

```
using namespace std;
#include <iostream>
#include "vector.h"
int main (){
  vector k;
  cout << "Myfn returns: " << k.myfn() << endl;
  return 0;
}
```

35

Vector.h

```
class vector{
  public:
    double myfn();
};
```

Vector.cpp

```
double vector::myfn(){
  //some code goes here
}
```

36

Using Multiple Code & Header Files

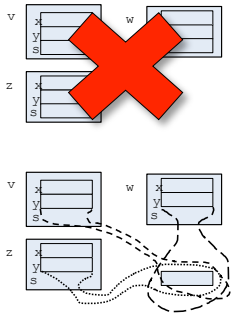
- To compile: `g++ Main.cpp Vector.cpp`
- Gotcha: The compiler will get upset if you include a header file in multiple source code files. Fix:

```
#ifndef VECTORHEADERFILE
#define VECTORHEADERFILE
class vector{
  public:
    double x, y;
    double myfn();
};
#endif
```

37

15. Static Variables

```
class vector{
  public:
    double x,y;
    static int s;
}
vector v, w, z;
```



38

```
void myfn{
  int s = 0;
  cout << s <<endl;
  s++;
}
void main(){
  myfn();
  myfn();
  myfn();
}
```

Output

```
0
0
0
```

```
void myfn{
  static int s = 0;
  cout << s <<endl;
  s++;
}
void main(){
  myfn();
  myfn();
  myfn();
}
```

Output

```
0
1
2
```

39

16. Constants

```
#define PI 3.1415928 //C
```

```
const double PI = 3.1415928; //C++
```

40

17. Derived Classes

- A class can be *derived* from another class.
- The new class *inherits* the member variables and member functions of the *base class*.
- Additional variables and/or functions can be added.

41

Base Class Example

```
class vector{
  public:
    double x, y;
    vector(double a=0, double b=0){
      x = a; y = b;
    }
    double module(){
      return sqrt(x*x + y*y);
    }
};
```

42

Derived Class Example

```
class trivector: public vector{
  public:
    double z;
    trivector(
      double m=0, double n=0, double p=0): vector(m, n){
        z = p;
    }
    trivector(vector a){
      x = a.x; y = a.y; z = 0;
    }
    double module(){
      return sqrt(x*x + y*y + z*z);
    }
};
```

43

18. Virtual Functions

```
vector* vptr;
trivector t(1,1,1);
vptr = &t; //this is legal
cout << vptr->module();
```

```
virtual double module(){
  return sqrt(x*x + y*y);
}
```

Output

```
1.41
```

Output

```
1.73
```

44

19. Public, Private, and Protected

Public: accessible from everywhere.

Private: accessible only from member functions of the class but not derived classes.

Protected: accessible only from member functions of the class and derived classes.

45

20. File Input and Output

```
using namespace std;
#include <iostream>
#include <fstream>
int main(){
  fstream f;
  f.open("test.txt", ios::out);
  f << "Text output to a file." << endl;
  double a = 345;
  f << "A number: " << a << endl;
  f.close();
  return 0;
}
```

Output File Contents

```
> cat test.txt
Text output to a file.
A number: 345
```

46

```
using namespace std;
#include <iostream>
#include <fstream>
int main(){
  fstream f;
  char c;
  cout << "What's inside test.txt?" << endl;
  f.open("test.txt", ios::in);
  while(!f.eof()){
    f.get(c); // Or c = f.get()
    cout << c;
  }
  f.close();
  return 0;
}
```

Output

```
What's inside test.txt?"
Text output to a file.
A number: 345
```

47