BABAR C++ Course

Paul F. Kunz

Stanford Linear Accelerator Center

No prior knowledge of C assumed

I’m not an expert in C++

Will try to do the dull stuff quickly, then move into OOP and OO design

You need to practice to really learn C++

First two sessions is about the same for C, C++, Objective-C, Java, and C#

Preliminaries

Recommended text book:

- John J. Barton and Lee R. Nackman
  Scientific and Engineering **C**++
  Addison-Wesley
  IBSN: 0-201-53393-6

Compiling examples

Create `a.out` executable with

- for gcc: `g++ file.C`
- for other compilers: ?

Type `a.out` to run.

Comments

Two forms of comments allowed

- Tradition C style

```
/* This is a comment */

/*
 * This is a multiline
 * comment
 */

a = /* ugly comment */ b + c;
```

- New C++ style

```
// This is a comment

//
// This is a multiline
// comment
//

a = b + c; // comment after an expression
```

Main program

All programs must have a main

Most trivial is

```
int main() {
    return 0;
}
```

- under UNIX, suffix is .C or .cc or .cpp or .cxx
- under Windows do not use .C
- `main()` is a function called by the OS
- this `main()` takes no arguments
- braces (“`{`” and “`}`”) denote body of function
- `main` returns 0 to the OS (success!)
- a statement ends with semi-colon (“`;`”), otherwise completely free form
- same rules as C (except `.c` suffix is used)

C++ Input and Output

Introduce I/O **early, so we can run programs from shell and see something happen :-)**

Example

```
#include <iostream> // preprocessor command
using namespace std;

int main() {
  // Read and print three floating point numbers
  float a, b, c;
  cin >> a >> b >> c;  // input
  // output
  cout << a << ", " << b << ", " << c << endl;

  return 0;
}
```

- `iostream` is header file containing declarations needed to use C++ I/O system
- `a`, `b`, and `c` are floating point variables (like `REAL*4`)
- `cin >>` reads from `stdin`, *i.e.* the keyboard
- `cout <<` prints to `stdout`, *i.e.* the screen
- `endl` is special variable: the end-of-line (‘`\n`’ in C) Unlike Fortran, you control the end-of-line.

More on I/O

Controlling end-of-line has its advantages

Example

```
// Print the equation coefficients of a*x + b*y + c = 0
cout << "Coefficients: " << a << ", " << b << ", " << c << endl;

// Compute and print the x-intercept.
cout << "x-intercept: ";
if (a != 0) {
    cout << -c / a << ", "; // a not equal to 0
}
else {
    cout << "none, "; // a is equal to 0
}
```

- an expression can be input to `cout <<`
- we print the result of the expression, or “none” on same line las label.

math.h

Unlike Fortran, there are no intrinsic functions

But there are standard libraries

One must include header file to make library functions available at compile time

Example

```
#include <iostream>
#include <math.h>
using namespace std;

int main() {

  float angle;     // Angle, in degrees
  cin >> angle;
  cout << cos(angle * M_PI / 180.0 ) << endl;
                   // M_PI is from <cmath>
  return 0;
}
```

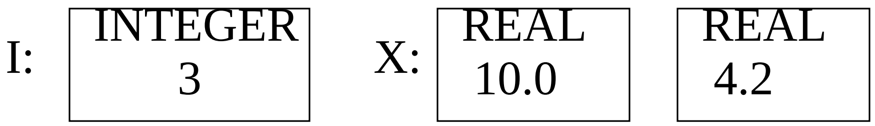
- functions can be input to `cout <<`
- see `/usr/include/math.h` to get list of functions
- useful constants are defined as well
- C and C++ share same library

Variables, Objects, and Types

Consider

```
      INTEGER I
      REAL X
      DATA I/3/, X/10.0/
      CALL S(X, 4.2)
```

- we have three objects with initial value



I: INTEGER 3

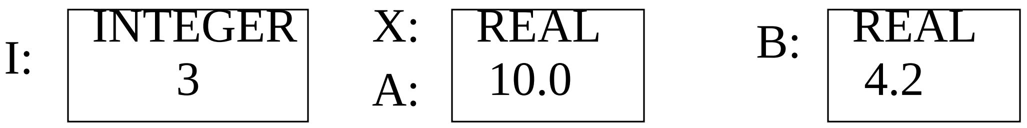
X: REAL 10.0

REAL 4.2

Consider (`simple.f`) `S()`

```
      SUBROUTINE S(A, B)
      REAL A, B
      A = B
      END
```

- we have still only three objects, but,



I: INTEGER 3

X: A: REAL 10.0

B: REAL 4.2

- thus `X` gets changed by `S()` in calling routine
- we say: Fortran passes by reference

Declaring types and initializing

Consider

```
int i = 3;
float x = 10.0;
```

- variable names must start with a letter or “_”, and are case sensitive
- initialization can occur on same line
- multiple declarations are allowed
- type declaration is *mandatory*
  (like having `IMPLICIT NONE` in every file)
- for all of the above, same rules in C
- type declaration must be before first use, but does not have to be before first executable statement

```
int i = 3;
float x = 10.0;
i = i + 1;
int j = i;
```

- general practice is to make type declaration just before first use

Types

Both Fortran and C/C**++ have** ***types***

Fortran	C++ or C
`LOGICAL`	`bool (C++ only)`
`CHARACTER*1`	`char`
`INTEGER*2`	`short`
`INTEGER*4`	`int long`
`REAL*4`	`float`
`REAL*8`	`double`
`COMPLEX`	

- defines the meaning of bits in memory
- defines which machine instructions to generate on certain operations
- `limits.h` gives you the valid range of integer types
- `float.h` gives you the valid range, precision, *etc.* of floating point types
- as with Fortran, watch out on 64 bit machines

Arithmetic Operators

Both Fortran and C/C++ have *operators*

Fortran	Purpose	C or C++
`X + Y`	`add`	`x + y`
`X - Y`	`subtract`	`x - y`
`X*Y`	`multiply`	`x*y`
`X/Y`	`divide`	`x/y`
`MOD(X,Y)`	`modulus`	`x%y`
`X**Y`	`exponentiations`	`pow(x,y)`
`+X`	`unary plus`	`+x`
`-Y`	`unary minus`	`-y`
	`postincrement`	`x++`
	`preincrement`	`++x`
	`postdecrement`	`x--`
	`predecrement`	`--x`

- `x++` is equivalent to `x = x + 1`
- `x++` means current value, then increment it
- `++x` means increment it, then use it.
- sorry, can’t do `x**2`; use `x*x` instead (for sub-expressions like `(x+y)**2`, we’ll see some tricks later)

Exercise

What is the output of

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

int main() {

  int i = 1;
  cout << i << ", ";
  cout << (++i) << ", ";
  cout << i << ", ";
  cout << (i++) << ", ";
  cout << i << endl;

  return 0;
}
```

Should be

```
 1, 2, 2, 2, 3
```

Try changing ++ to `--`

Relational Operators

Both Fortran and C/C++ define relational operators

Fortran	Purpose	C or C++
`X .LT. Y`	less than	`x < y`
`X .LE. Y`	less than or equal	`x <= y`
`X .GT. Y`	greater than	`x > y`
`X .GE. Y`	greater than or equal	`x >= y`
`X .EQ. Y`	equal	`x == y`
`X .NE. Y`	not equal	`x != y`

- zero is false and non-zero is true

Logical operators and Values

Both Fortran and C/C++ have logical operations and values

Fortran	Purpose	C or C++
.FALSE.	false value	0 or false
.TRUE.	true value	non-zero or true
.NOT. X	logical negation	!x
X .AND. Y	logical and	x && y
X .OR. Y	logical inclusive or	x || y

- `&&` and `||` evaluate from left to right and right hand expression not evaluated if it doesn't need to be
- the following never divides by zero

```
if ( d && (x/d < 10.0) ) {
    // do some stuff
}
```

- Only C++ has `true` and `false` as values.

Characters

C/C**++ only has one byte characters**

Constants of type `char` **use single quotes**

```
char a = ’a’;
char aa = ’A’;
```

Use *escape sequence* for unprintable characters and special cases

- `’\n’` for new line
- `’\’’` for single quote
- `’\”’` for double quotes
- `’\?’` for question mark
- `’\ddd’` for octal number
- `’\xdd’` for hexadecimal

Bitwise Operators

Both Fortran and C/C++ have bitwise operators

Fortran	Purpose	C/C++
`NOT(I)`	`complement`	`~i`
`IAND(I,J)`	`and`	`i&j`
`IEOR(I,J)`	`exclusive or`	`i^j`
`IOR(I,J)`	`inclusive or`	`i|j`
`ISHFT(I,N)`	`shift left`	`i<<n`
`ISHFT(I,-N)`	`shift right`	`i>>n`

- can be used on any integer type (`char`, `short`, `int`, *etc.*)
- right shift might not do sign extension
- most often used for on-line DAQ and trigger
- also used for unpacking compressed data

Assignment operators

C/C**++ has many assignment operators**

Fortran	Purpose	C or C++
X = Y	assignment	x = y
X = X + Y	add assignment	x += y
X = X - Y	subtract assignment	x -= y
X = X*Y	multiply assignment	x *= y
X = X/Y	divide assignment	x /= y
X = MOD(X,Y)	modulus assignment	x %= y
X = ISHFT(X,-N)	right shift assignment	x >>= n
X = ISHFT(X,N)	left shift assignment	x <<= n
X = IAND(X,Y)	and assignment	x &= y
X = IOR(X,Y)	or assignment	x |= y
X = IEOR(X,Y)	xor assignment	x ^= y

- takes some time to get use to
- makes code more compact

Operator Precedence

Both Fortran and C/C++ use precedence rules to determine order to evaluate expressions

- `z = a*x + b*y + c;` evaluates as you would expect
- also left to right or right to left precedence defined
- can over ride default by use of parentheses
- when in doubt, use parentheses
- make code easy to understand
- don’t make clever use of precedence

`if` Statements

C/C++ **`if` statement is analogous to Fortran**

```
if (current_temp > maximum_safe_temp) {
    cerr << "EMERGENCY: Too hot--flushing" << endl;
    flushWithWater();
}
```

Any expression that evaluates to numeric value is allowed.

```
if ( !(channel = openChannel(“temperature”)) ) {
    cerr << "Could not open channel" << endl;
    exit(1);
}
```

`if` gotchas

Braces are optional when single expression is in the block

```
if ( x < 0 )
    x = -x;  // abs(x)
    y = -y;  // always executed
```

- leaves potential for future error
- suggest single expressions remain on same line

```
if ( x < 0 ) x = -x;  // abs(x)
```

Any expression, including assignment

```
int  i, j;
// some code setting i and j
if ( i = j ) {
    // some stuff
}
```

- a common mistake; this sets `i = j` and then does some stuff if `j` is non-zero

`if else` Statements

Analogous to Fortran

```
if ( x < 0 ) {
    y = -x;
} else {
    y = x;
}
```

C/C**++ also has condition operator**

```
y = (x < 0) ? -x : x; // y = abs(x)
```

- use only for simple expressions
- else code can become unreadable

Also have

```
if ( x < 0 ) {
    y = -x;
} else if (x > 0) {
    y = x;
} else {
    y = 0;
}
```

Coding Styles

C/C**++ is free form**

Common styles for `if` block are

```
if ( x < 0 ) {
    y = -x;
} else {
    y = x;
}
// or
if ( x < 0 )
{
    y = -x;
}
else
{
    y = x;
}
```

- the first is more common

`while` loop

C/C++ **`while` is when block should be executed zero or more times**

General form

```
while (expression) {
    statement
    ...
}
```

- any expression that returns numeric value
- same rules as `if` block for braces
- Fortran equivalent requires `GOTO`

```
10 IF (.NOT. expression ) GOTO 20
   statement
    ...
   GOTO 10
20 CONTINUE
```

`while` Example

Example

```
#include <iostream>
#include <math.h>
using namespace std;

int main() {
  float x;
  while (cin >> x) {
    cout << x << sqrt(x) << endl;
  }
  return 0;
}
```

- reads terminal until end-of-file
- <ctrl>-d is end-of-file for UNIX
- I can not explain how this works until later

`do-while` loop

C/C++ **`do-while` is when block should be executed one or more times**

General form

```
do {
    statement
    ...
} while(expression);
```

- any expression that returns numeric value
- same rules as `if` block for braces
- Fortran equivalent requires `GOTO`

```
10 CONTINUE
    statement
    ...
   IF(expression)GOTO 10
```

do-while Example

Snippet from use of Newton’s method

```
x = initial_guess;
do {
    dx = f(x) / fprime(x);
    x  -= dx;
} while (fabs(dx) > desired_accuracy);
```

`for` loop

C/C++ `for` loop much more general than Fortran `DO` loop

```
for(init-statement; test-expr; increment-expr) {
    statement
    ...
}
```

- the test expression can be any that returns numeric value like `if` block
- function calls and I/O are also allowed

In Fortran

```
   DO 10 I = 1, J, K
    statements
    ...
10 CONTINUE
```

In C or C++

```
for( i = 1; i <= j; i += k ) {
    statements
    ...
}
```

More Examples

Typically, one sees

```
for(int i = 0; i < count; i++) {
    // statements in loop body
}
```

- where `i` is declared and typed in init-statement

Nested loops might iterate over all pairs with

```
for(i = 0; i < count - 1; i++) {
    for(j = i+1; j < count; j++) {
       // statements in loop body
    }
}
```

Use of two running indices might be

```
for(i = 0, j = count-1; i < count-1; i++, j--) {
   // statements in loop body
}
```

- separate expressions with commas

`break` and `continue` Statements

Consider following Fortran

```
     DO 100 I = 1, 100
         IF ( I .EQ. J ) GO TO 100
         IF ( I .GT. J ) GO TO 200
             ! do some work
100 CONTINUE
200 CONTINUE
```

- common need to break out of loop or continue to next iteration.

Equivalent C++ **code is**

```
for (i = 0; i < 100; i++ ) {
    if ( i == j ) continue;
    if ( i > j ) break;
    // do some work
}
```

- `continue` goes to next iteration of current loop
- `break` step out of current loop
- `goto` exists in C/C++ but rarely used
- we’ll make less use of these constructs in C++, then in either C or Fortran

Arrays

A collection of elements of same type

```
float x[100]; // like REAL*4 X(100) in F77
```

- access first element of array with `x[0]`
- access last element of array with `x[99]`

Initializing array elements

```
float x[3] = {1.1, 2.2, 3.3};
float y[] = {1.1, 2.2, 3.3, 4.4};
```

- can let the compiler calculate the dimension

Multi-dimensions arrays

```
float m[4][4]; // like REAL*4 M(4,4) in F77
int m [2][3] = { {1,2,3}
                 {4,5,6} };
```

- elements appear row-wise
- Fortran elements appear column-wise
- Thus `m[0][1]` in C/C++ is `M(2,1)` in Fortran
- royal pain to interface C/C++ with Fortran

Example Code and a Test

Multiplying matrices

```
float m[3][3], m1[3][3], m2[3][3];
// Code that initializes m1 and m2 ...

// m = m1 * m2
double sum;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum = 0.0;
        for (int k = 0; k < 3; k++) {
            sum += m1[i][k] * m2[k][j];
        }
        m[i][j] = sum;
    }
}
```

- If you understand this code, then you know enough C/C++ to code the algorithmic part of your code
- At the beginning of this session, the above code would probably have been gibberish
- If you can not understand this code, then I’m going too fast :-(

A Pause for Reflection

What have we learned so far?

- we've seen how to do in C/C++ everything you can do in Fortran 77 except functions, `COMMON` blocks, and character arrays.
- some aspects of C/C++ are more convenient than Fortran; some are not
- but we've seen nothing fundamentally new, things are just different

Next session, we start with some new stuff and we're not even finished with chapter 2!

In particular, the replacement for `COMMON` blocks is going to be quite different