Lectures on C++

Glen Cowan
Physics Department
Royal Holloway, University of London
Egham, Surrey TW20 0EX

01784 443452
g.cowan@rhul.ac.uk
www.pp.rhul.ac.uk/~cowan/stat_course.html

C++ Outline

1 Introduction to C++ and UNIX environment
2 Variables, types, expressions, loops
3 Type casting, functions
4 Files and streams
5 Arrays, strings, pointers
6 Classes, intro to Object Oriented Programming
7 Memory allocation, operator overloading, templates
8 Inheritance, STL, **`gmake, ddd`**

Some resources on C++

There are many web based resources, e.g.,

`www.doc.ic.ac.uk/~wjk/C++Intro` (Rob Miller, IC course)

`www.cplusplus.com` (online reference)

`www.icce.rug.nl/documents/cplusplus` (F. Brokken)

See links on course site or google for "C++ tutorial", etc.

There are thousands of books – see e.g.

W. Savitch, *Problem Solving with C++*, 4^{th} edition (lots of detail – very thick).

B. Stroustrup, *The C++ Programming Language* (the classic – even thicker).

Lippman, Lajoie (& Moo), *C++ Primer*, A-W, 1998.

Introduction to UNIX/Linux

We will learn C++ using the Linux operating system

Open source, quasi-free version of UNIX

UNIX and C developed ~1970 at Bell Labs

Short, cryptic commands: `cd`, `ls`, `grep`, …

Other operating systems in 1970s, 80s ‘better’, (e.g. VMS) but, fast ‘RISC processors’ in early 1990s needed a cheap solution → we got UNIX

In 1991, Linus Torvalds writes a free, open source version of UNIX called Linux.

We currently use the distribution from CERN

Basic UNIX

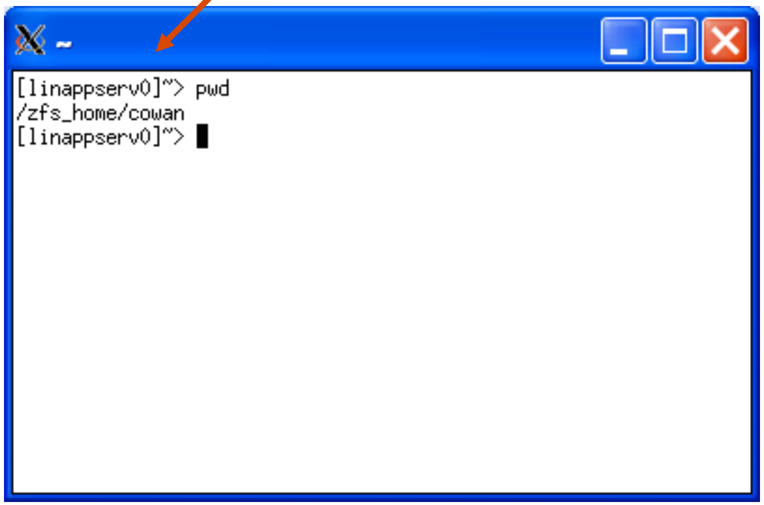
UNIX tasks divide neatly into:

interaction between operating system and computer (the kernel),
interaction between operating system and user (the shell).

Several shells (i.e. command sets) available: sh, csh, tcsh, bash, …

Shell commands typed at a prompt, here **`[linappserv0]~>`**
often set to indicate name of computer:

Command **`pwd`** to "print working directory", i.e., show the directory (folder) you're sitting in.

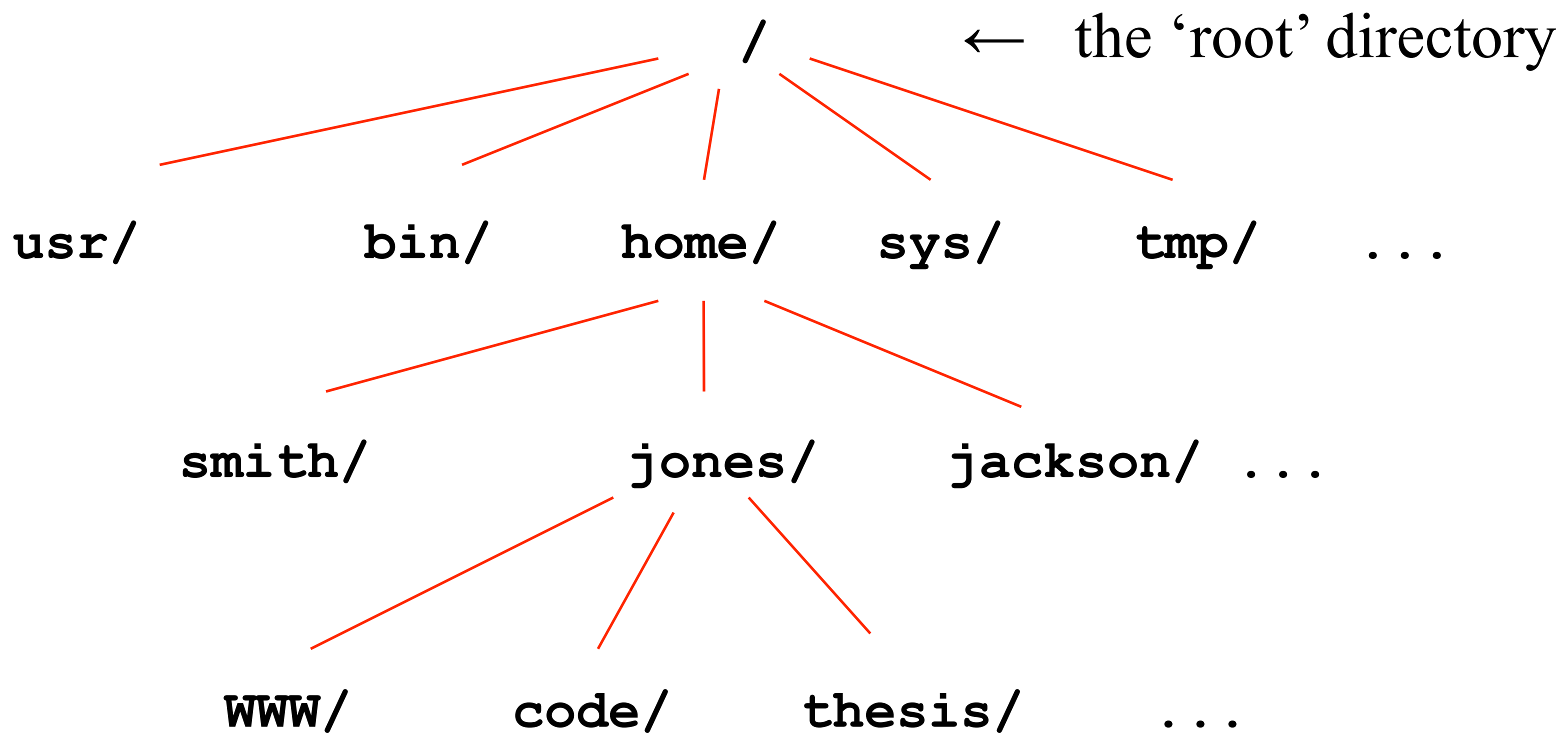


Commands are case sensitive.

`PWD` will not work .

UNIX file structure

Tree-like structure for files and directories (like folders):



File/directory names are case sensitive: **`thesis`** $\neq$ **`Thesis`**

Simple UNIX file tricks

A complete file name specifies the entire 'path'

```
/home/jones/thesis/chapter1.tex
```

A tilde points to the home directory:

`~/thesis/chapter1.tex` ← the logged in user (e.g. jones)

`~smith/analysis/result.dat` ← a different user

Single dot points to current directory, two dots for the one above:

`/home/jones/thesis` ← current directory

`../code` ← same as /home/jones/code

A few UNIX commands (case sensitive!)

`pwd`	Show present working directory
`ls`	List files in present working directory
`ls -la`	List files of present working directory with details
`man ls`	Show manual page for `ls`. Works for all commands.
`man -k` ***keyword***	Searches man pages for info on "keyword".
`cd`	Change present working directory to home directory.
`mkdir` ***foo***	Create subdirectory foo
`cd` ***foo***	Change to subdirectory foo (go down in tree)
`cd ..`	Go up one directory in tree
`rmdir` ***foo***	Remove subdirectory foo (must be empty)
`emacs` ***foo*** `&`	Edit file foo with emacs (& to run in background)
`more` ***foo***	Display file foo (space for next page)
`less` ***foo***	Similar to `more` ***foo***, but able to back up (`q` to quit)
`rm` ***foo***	Delete file foo

A few more UNIX commands

cp *foo bar*	Copy file foo to file bar, e.g., **cp ~smith/*foo* ./** copies Smith's file foo to my current directory
mv *foo bar*	Rename file foo to bar
lpr *foo*	Print file foo. Use **-P** to specify print queue, e.g., **lpr -Plj1 *foo*** (site dependent).
ps	Show existing processes
kill *345*	Kill process 345 (**kill -9** as last resort)
./*foo*	Run executable program foo in current directory
***ctrl*-c**	Terminate currently executing program
chmod ug+x *foo*	Change access mode so user and group have privilege to execute foo (Check with **ls -la**)

Better to read a book or online tutorial and use **man** pages

UNIX file access

If you type **ls -la**, you will see that each file and directory is characterized by a set of file access rights:

```
[linappserv1]~/stat/atlas/higgs> ls -la
total 498
drwxr-xr-x   4 cowan hep2    512 Oct  5 11:31 ./
drwxr-xr-x  15 cowan hep2   1536 Dec  2  2008 ../
drwxr-xr-x   2 cowan hep2    512 Dec 12  2007 hgg/
-rw-r--r--   1 cowan hep2  59200 Oct  5 11:30 higgs_bayes.dvi
-rw-r--r--   1 cowan hep2 376373 Oct  5 11:30 higgs_bayes.ps
-rw-r--r--   1 cowan hep2  59493 Oct  5 11:30 higgs_bayes.tex
drwxr-xr-x   2 cowan hep2    512 Dec  6  2007 hww_toy_combination_inputs/
[linappserv1]~/stat/atlas/higgs>
```

Three groups of letters refer to: user (u), group (g) and other (o). The possible permissions are read (r), write (w), execute (x).

Default may be everyone in your group has read access to all of your files. To change this, use **chmod**, e.g.

```
chmod go-rwx hgg
```

prevents group and other from seeing the directory **hgg**.

Introduction to C++

Language C developed (from B) ~ 1970 at Bell Labs

Used to create parts of UNIX

C++ derived from C in early 1980s by Bjarne Stroustrup

“C with classes”, i.e., user-defined data types that allow “Object Oriented Programming”.

Java syntax based largely on C++ (head start if you know java)

C++ is case sensitive (**`a`** not same as **`A`**).

Currently most widely used programming language in High Energy Physics and many other science/engineering fields.

Recent switch after four decades of FORTRAN.

Compiling and running a simple C++ program

Using,e.g., emacs, create a file **HelloWorld.cc** containing:

```
// My first C++ program
#include <iostream>
using namespace std;
int main(){
  cout << "Hello World!" << endl;
  return 0;
}
```

We now need to compile the file (creates machine-readable code):

```
g++ -o HelloWorld HelloWorld.cc
```

Invokes compiler (gcc) name of output file source code

Run the program:

```
./HelloWorld
Hello World!
```

← you type this
← computer shows this

Notes on compiling/linking

```
g++ -o HelloWorld HelloWorld.cc
```

is an abbreviated way of saying first

```
g++ -c HelloWorld.cc
```

Compiler (`-c`) produces `HelloWorld.o.` ('object files')
Then 'link' the object file(s) with

```
g++ -o HelloWorld HelloWorld.o
```

If the program contains more than one source file, list with spaces; use \ to continue to a new line:

```
g++ -o HelloWorld HelloWorld.cc Bonjour.cc \
GruessGott.cc YoDude.cc
```

Writing programs in the Real World

Usually create a new directory for each new program.

For trivial programs, type compile commands by hand.

For less trivial but still small projects, create a file (a ‘script’) to contain the commands needed to build the program:

```
#!/bin/sh
# File build.sh to build HelloWorld
g++ -o HelloWorld HelloWorld.cc Bonjour.cc \
GruessGott.cc YoDude.cc
```

To use, must first have ‘execute access’ for the file:

```
chmod ug+x build.sh        ← do this only once
./build.sh                 ← executes the script
```

A closer look at HelloWorld.cc

```
// My first C++ program
```

is a comment (preferred style)

The older ‘C style’ comments are also allowed (cannot be nested):

```
/*
   These lines
   here are comments
*/

/* and so are these */
```

You should include enough comments in your code to make it understandable by someone else (or by yourself, later).

Each file should start with comments indicating author’s name, main purpose of the code, required input, etc.

More HelloWorld.cc – include statements

`#include <iostream>` is a compiler directive.

Compiler directives start with `#`. These statements are not executed at run time but rather provide information to the compiler.

`#include <iostream>` tells the compiler that the code will use library routines whose definitions can be found in a file called `iostream`, usually located somewhere under `/usr/include`

Old style was `#include <iostream.h>`

`iostream` contains functions that perform i/o operations to communicate with keyboard and monitor.

In this case, we are using the iostream object `cout` to send text to the monitor. We will include it in almost all programs.

More HelloWorld.cc

`using namespace std;` More later. For now, just do it.

A C++ program is made up of functions. Every program contains exactly one function called main:

```
int main(){
  // body of program goes here

  return 0;
}
```

Functions "return" a value of a given type; `main` returns `int` (integer).

The `()` are for arguments. Here `main` takes no arguments.

The body of a function is enclosed in curly braces: `{ }`

`return 0;` means `main` returns a value of 0 .

Finishing up HelloWorld.cc

The 'meat' of HelloWorld is contained in the line

```
cout << "Hello World!" << endl;
```

Like all statements, it ends with a semi-colon.

`cout` is an "output stream object".

You send strings (sequences of characters) to `cout` with `<<`

We will see it also works for numerical quantities (automatic conversion to strings), e.g., `cout << "x = " << x << endl;`

Sending `endl` to `cout` indicates a new line. (Try omitting this.)

Old style was `"Hello World!\n"`

C++ building blocks

All of the words in a C++ program are either:

Reserved words: cannot be changed, e.g.,

`if, else, int, double, for, while, class`, ...

Library identifiers: default meanings usually not changed, e.g., **`cout, sqrt`** (square root), ...

Programmer-supplied identifiers:

e.g. variables created by the programmer,

`x, y, probeTemperature, photonEnergy`, ...

Valid identifier must begin with a letter or underscore (“_”) , and can consist of letters, digits, and underscores.

Try to use meaningful variable names; suggest lowerCamelCase.

Data types

Data values can be stored in variables of several types.

Think of the variable as a small blackboard, and we have different types of blackboards for integers, reals, etc. The variable name is a label for the blackboard.

Basic integer type: **`int`** (also **`short, unsigned, long int`**, ...)

Number of bits used depends on compiler; typically 32 bits.

Basic floating point types (i.e., for real numbers):

`float` usually 32 bits

`double` usually 64 bits ← best for our purposes

Boolean: **`bool`** (equal to **`true`** or **`false`**)

Character: **`char`** (single ASCII character only, can be blank), no native 'string' type; more on C++ strings later.

Declaring variables

All variables must be declared before use.

Usually declare just before 1st use.

Examples

```
int main(){
  int numPhotons;          // Use int to count things
  double photonEnergy;     // Use double for reals
  bool goodEvent;          // Use bool for true or false
  int minNum, maxNum;      // More than one on line
  int n = 17;              // Can initialize value
  double x = 37.2;         // when variable declared.
  char yesOrNo = ‘y’;      // Value of char in ‘ ’
      ...
}
```

Assignment of values to variables

Declaring a variable establishes its name; value is undefined (unless done together with declaration).

Value is assigned using = (the assignment operator):

```
int main(){
  bool aOK = true;  // true, false predefined constants
  double x, y, z;
  x = 3.7;
  y = 5.2;
  z = x + y;
  cout << "z = " << z << endl;
  z = z + 2.8;        // N.B. not like usual equation
  cout << "now z = " << z << endl;
      ...
}
```

Constants

Sometimes we want to ensure the value of a variable doesn't change.

Useful to keep parameters of a problem in an easy to find place, where they are easy to modify.

Use keyword **`const`** in declaration:

```
const int numChannels = 12;
const double PI = 3.14159265;

// Attempted redefinition by Indiana State Legislature
PI = 3.2;               // ERROR will not compile
```

Old C style retained for compatibility (avoid this):

```
#define PI 3.14159265
```

Enumerations

Sometimes we want to assign numerical values to words, e.g.,

January = 1, February = 2, etc.

Use an 'enumeration' with keyword **`enum`**

```
enum { RED, GREEN, BLUE };
```

is shorthand for

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

Enumeration starts by default with zero; can override:

```
enum { RED = 1, GREEN = 3, BLUE = 7 }
```

(If not assigned explicitly, value is one greater than previous.)

Expressions

C++ has obvious(?) notation for mathematical expressions:

operation	symbol
addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
modulus	`%`

Note division of `int` values is truncated:

```
int n, m;  n = 5;  m = 3;
int ratio = n/m;        // ratio has value of 1
```

Modulus gives remainder of integer division:

```
int nModM = n%m;           // nModM has value 2
```

Operator precedence

`*` and `/` have precedence over `+` and `-`, i.e.,

`x*y +  u/v` means `(x*y) + (u/v)`

`*` and `/` have same precedence, carry out left to right:

`x/y/u*v` means `((x/y) / u) * v`

Similar for `+` and `-`

`x - y + z` means `(x - y) + z`

Many more rules (google for C++ operator precedence).

Easy to forget the details, so use parentheses unless it's obvious.

Boolean expressions and operators

Boolean expressions are either true or false, e.g.,

```
int n, m; n = 5; m = 3;
bool b = n < m;                // value of b is false
```

C++ notation for boolean expressions:

```
greater than                 >
greater than or equals       >=
less than                    <
less than or equals          <=
equals                       ==
not equals                   !=
```

not =

Can be combined with `&&` ("and"), `||` ("or") and `!` ("not"), e.g.,

```
(n < m) && (n != 0)                  (false)
(n%m >= 5) || !(n == m)              (true)
```

Precedence of operations not obvious; if in doubt use parentheses.

Shorthand assignment statements

full statement	shorthand equivalent
`n = n + m`	`n += m`
`n = n - m`	`n -= m`
`n = n * m`	`n *= m`
`n = n / m`	`n /= m`
`n = n % m`	`n %= m`

Special case of increment or decrement by one:

full statement	shorthand equivalent
`n = n + 1`	`n++` (or `++n`)
`n = n - 1`	`n--` (or `--n`)

`++` or `--` before variable means first increment (or decrement), then carry out other operations in the statement (more later).

Getting input from the keyboard

Sometimes we want to type in a value from the keyboard and assign this value to a variable. For this use the iostream object **`cin`**:

```
int age;
cout << "Enter your age" << endl;
cin >> age;
cout << "Your age is " << age << endl;
```

When you run the program you see

```
Enter your age
23                    ← you type this, then "Enter"
Your age is 23
```

(Why is there no "**`jin`**" in java? What were they thinking???)

if and else

Simple flow control is done with `if` and `else`:

```
if ( boolean test expression ){
   Statements executed if test expression true
}
```

or

```
if (expression1 ){
   Statements executed if expression1 true
}
else if ( expression2 ) {
   Statements executed if expression1 false
   and expression2 true
}
else {
   Statements executed if both expression1 and
   expression2 false
}
```

more on if and else

Note indentation and placement of curly braces:

```
if ( x > y ){
   x = 0.5*x;
}
```

Some people prefer

```
if ( x > y )
{
   x = 0.5*x;
}
```

If only a single statement is to be executed, you can omit the curly braces -- this is usually a bad idea:

```
if  ( x > y )  x = 0.5*x;
```

Putting it together -- `checkArea.cc`

```
#include <iostream>
using namespace std;
int main() {
  const double maxArea = 20.0;
  double width, height;
  cout << "Enter width" << endl;
  cin >> width;
  cout << "Enter height" << endl;
  cin >> height;
  double area = width*height;
  if ( area > maxArea ){
    cout << "Area too large" << endl;
  }
  else {
    cout << "Dimensions are OK" << endl;
  }
  return 0;
}
```

“while” loops

A `while` loop allows a set of statements to be repeated as long as a particular condition is true:

```
while( boolean expression ){
  // statements to be executed as long as
  // boolean expression is true

}
```

For this to be useful, the boolean expression must be updated upon each pass through the loop:

```
while (x < xMax){
  x += y;
  ...
}
```

Possible that statements never executed, or that loop is infinite.

"do-while" loops

A `do-while` loop is similar to a `while` loop, but always executes at least once, then continues as long as the specified condition is true.

```
do {
  // statements to be executed first time
  // through loop and then as long as
  // boolean expression is true

} while ( boolean expression )
```

Can be useful if first pass needed to initialize the boolean expression.

“for” loops

A `for` loop allows a set of statements to be repeated a fixed number of times. The general form is:

```
for ( initialization action ;
      boolean expression ; update action ){
   // statements to be executed

}
```

Often this will take on the form:

```
for (int i=0; i<n; i++){
  // statements to be executed n times

}
```

Note that here `i` is defined only inside the `{ }`.

Examples of loops

A for loop:

```
int sum = 0;
for (int i = 1; i<=n; i++){
  sum += i;
}
cout << "sum of integers from 1 to " << n <<
     " is " << sum << endl;
```

A do-while loop:

```
int n;
bool gotValidInput = false;
do {
  cout << "Enter a positive integer" << endl;
  cin >> n;
  gotValidInput = n > 0;
} while ( !gotValidInput );
```

Nested loops

Loops (as well as if-else structures, etc.) can be nested, i.e., you can put one inside another:

```
// loop over pixels in an image

for (int row=1; row<=nRows; row++){
  for (int column=1; column<=nColumns; column++){
    int b = imageBrightness(row, column);
    ...

  }     // loop over columns ends here
}       // loop over rows ends here
```

We can put any kind of loop into any other kind, e.g., **`while`** loops inside **`for`** loops, vice versa, etc.

More control of loops

`continue` causes a single iteration of loop to be skipped (jumps back to start of loop).

`break` causes exit from entire loop (only innermost one if inside nested loops).

```
while ( processEvent ) {

  if ( eventSize > maxSize ) { continue; }

  if ( numEventsDone > maxEventsDone ) {
    break;
  }

//  rest of statements in loop ...

}
```

Usually best to avoid `continue` or `break` by use of `if` statements.

Type casting

Often we need to interpret the value of a variable of one type as being of a different type, e.g., we may want to carry out floating-point division using variables of type **`int`**.

Suppose we have: **`int n, m; n = 5; m = 3;`** and we want to know the real-valued ratio of **`n/m`** (i.e. not truncated). We need to "type cast" **`n`** and **`m`** from **`int`** to **`double`** (or **`float`**):

```
double x = static_cast<double>(n) /
           static_cast<double>(m);
```

will give **`x = 1.666666...`**

Will also work here with **`static_cast<double>(n)/m;`**
but **`static_cast<double>(n/m);`** gives **`1.0`**.

Similarly we can use **`static_cast<int>(x)`** to turn a **`float`** or **`double`** into an **`int`**, etc.

Digression #1: `bool` vs. `int`

C and earlier versions of C++ did not have the type `bool`. Instead, an `int` value of zero was interpreted as false, and any other value as true. This still works in C++:

```
int num = 1;
if ( num ) {
   ...               // condition true if num != 0
```

It is best to avoid this. If you want true or false, use `bool`. If you want to check whether a number is zero, then use the corresponding boolean expression:

```
if ( num != 0 ) {
   ...               // condition true if num != 0
```

Digression #2: value of an assignment and `==` vs. `=`

Recall `=` is the assignment operator, e.g., `x = 3;`

`==` is used in boolean expressions, e.g., `if ( x == 3 ) { ...`

In C++, an assignment statement has an associated value, equal to the value assigned to the left-hand side. We may see:

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

This says first assign `0` to `y`, then assign its value (`0`) to `x`. This can lead to very confusing code. Or worse:

```
if ( x = 0 ) { ... // condition always false!
```

Here what the author probably meant was

```
if ( x == 0 ) { ...
```

Standard mathematical functions

Simple mathematical functions are available through the standard C library **`cmath`** (previously **`math.h`**), including:

```
abs    acos   asin   atan   atan2  cos    cosh   exp
fabs   fmod   log    log10  pow    sin    sinh   sqrt
tan    tanh
```

Most of these can be used with **`float`** or **`double`** arguments; return value is then of same type.

Raising to a power, $z = x^y$, with **`z = pow(x,y)`** involves log and exponentiation operations; not very efficient for **`z = 2, 3`**, etc. Some advocate e.g. **`double xSquared = x*x;`**

To use these functions we need: **`#include <cmath>`**

Google for C++ cmath or see **`www.cplusplus.com`** for more info.

A simple example

Create file **testMath.cc** containing:

```
// Simple program to illustrate cmath library
#include <iostream>
#include <cmath>
using namespace std;
int main() {

  for (int i=1; i<=10; i++){
    double x = static_cast<double>(i);
    double y = sqrt(x);
    double z = pow(x, 1./3.);    // note decimal pts
    cout << x << "   " << y << "   " << z << endl;
  }

}
```

Note indentation and use of blank lines for clarity.

Running testMath

Compile and link: `g++ -o testMath testMath.cc`

Run the program: `./testMath`

```
1  1  1
2  1.41421  1.25992
3  1.73205  1.44225
4  2  1.5874
...
```

The numbers don't line up in neat columns -- more later.

Often it is useful to save output directly to a file. Unix allows us to redirect the output:

```
./testMath > outputFile.txt
```

Similarly, use `>>` to append file, `>!` to insist on overwriting.

These tricks work with any Unix commands, e.g., `ls`, `grep`, ...

Improved i/o: formatting tricks

Often it's convenient to control the formatting of numbers.

```
cout.setf(ios::fixed);
cout.precision(4);
```

will result in 4 places always to the right of the decimal point.

```
cout.setf(ios::scientific);
```

will give scientific notation, e.g., `3.4516e+05`. To undo this, use `cout.unsetf(ios::scientific);`

`cout.width(15)` will cause next item sent to `cout` to occupy 15 spaces, e.g.,

```
cout.width(5); cout << x;
cout.width(10); cout << y;
cout.width(10); cout << z << endl;
```

To use `cout.width` need `#include <iomanip>` .

More formatting: **`printf`** and **`scanf`**

Much of this can be done more easily with the C function **`printf`**:

```
printf ("formatting info" [, arguments]);
```

For example, for **`float`** or **`double x`** and **`int i`**:

```
printf("%f %d \n", x, i);
```

will give a decimal notation for **`x`** and integer for **`i`**.
`\n` does (almost) same as **`endl`**;

Suppose we want 8 spaces for **`x`**, 3 to the right of the decimal point, and 10 spaces for **`i`**:

```
printf("%8.3f %10d \n", x, i);
```

For more info google for printf examples, etc.

Also **`scanf`**, analogue of **`cin`**.

To use **`printf`** need **`#include <cstdlib>`** .

Scope basics

The scope of a variable is that region of the program in which it can be used.

If a block of code is enclosed in braces `{ }`, then this delimits the scope for variables declared inside the braces. This includes braces used for loops and if structures:

```
int x = 5;
for (int i=0; i<n; i++){
  int y = i + 3;
  x = x + y;
}
cout << "x = " << x << endl;  // OK
cout << "y = " << y << endl;  // BUG -- y out of scope
cout << "i = " << i << endl;  // BUG -- i out of scope
```

Variables declared outside any function, including **`main`**, have ‘global scope’. They can be used anywhere in the program.

More scope

The meaning of a variable can be redefined in a limited ‘local scope’:

```
int x = 5;
{
  double x = 3.7;
  cout << "x = " << x << endl;  // will print x = 3.7
}
cout << "x = " << x << endl;    // will print x = 5
```

(This is bad style; example is only to illustrate local scope.)

In general try to keep the scope of variables as local as possible. This minimizes the chance of clashes with other variables to which you might try to assign the same name.

Namespaces

A namespace defines a set of names (identifiers of variables, functions, objects) and a context in which they are used.

E.g., variables declared outside of any function are in the global namespace (they have global scope); and can be used anywhere.

A namespace can be defined with the **`namespace`** keyword:

```
namespace aNameSpace {
  double x = 1.0;
}
```

To refer to this **`x`** in some other part of the program (outside of its local namespace), we can use

```
aNameSpace::x
```

`::` is the scope resolution operator.

The `std` namespace

C++ provides automatically a namespace called **`std`**.

It contains all identifiers used in the standard C++ library (lots!), including, e.g., **`cin`**, **`cout`**, **`endl`**, ...

To use, e.g., **`cout`**, **`endl`**, we can say:

```
using std::cout;
using std::endl;
int main(){
  cout << "Hello" << endl;
  ...
```

or we can omit **`using`** and say

```
int main(){
  std::cout << "Hello" << std::endl;
  ...
```

using namespace std;

Or we can simply say

```
using namespace std;
int main(){
  cout << "Hello" << endl;
  ...
```

Although I do this in the lecture notes to keep them compact, it is not a good idea in real code. The namespace **std** contains thousands of identifiers and you run the risk of a name clash.

This construction can also be used with user-defined namespaces:

```
using namespace aNameSpace;
int main(){
  cout << x << endl;          // uses aNameSpace::x
  ...
```

Functions

Up to now we have seen the function **`main`**, as well as mathematical functions such as **`sqrt`** and **`cos`**. We can also define other functions, e.g.,

```
const double PI = 3.14159265;     // global constant
double ellipseArea(double, double);  // prototype
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}

double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The usefulness of functions

Now we can 'call' **`ellipseArea`** whenever we need the area of an ellipse; this is modular programming.

The user doesn't need to know about the internal workings of the function, only that it returns the right result.

'Procedural abstraction' means that the implementation details of a function are hidden in its definition, and needn't concern the user of the function.

A well written function can be re-used in other parts of the program and in other programs.

Functions allow large programs to be developed by teams (as is true for classes, which we will see soon).

Declaring functions

Before we can use a function, we need to declare it at the top of the file (before **`int main()`**).

```
double ellipseArea(double, double);
```

This is called the 'prototype' of the function. It begins with the function's 'return type'. The function can be used in an expression like a variable of this type.

The prototype must also specify the types of the arguments, in the correct sequence. Variable names are optional in the prototype.

The specification of the types and order of the arguments is called the function's signature.

Defining functions

The function must then be defined, i.e., we must say what it does with its arguments and what it returns.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The first word defines the type of value returned, here **`double`**.

Then comes a list of parameters, each preceded by its type.

Note the scope of **`a`** and **`b`** is local to the function **`ellipseArea`**. We could have given them names different from the **`a`** and **`b`** in the main program (and we often do).

Then the body of the function does the necessary computation and finally we have the **`return`** statement followed by the corresponding value of the function.

Return type of a function

The prototype must also indicate the return type of the function, e.g., **`int, float, double, char, bool`**.

```
double ellipseArea(double, double);
```

The function's return statement must return a value of this type.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

When calling the function, it must be used in the same manner as an expression of the corresponding return type, e.g.,

```
double volume = ellipseArea(a, b) * height;
```

Return type **void**

The return type may be ‘void’, in which case there is no return statement in the function (like a FORTRAN subroutine):

```
void showProduct(double a, double b){
  cout << "a*b = " << a*b << endl;
}
```

To call a function with return type void, we simply write its name with any arguments followed by a semicolon:

```
showProduct(3, 7);
```

Putting functions in separate files

Often we put functions in a separate files. The declaration of a function goes in a 'header file' called, e.g., **ellipseArea.h**, which contains the prototype:

```
#ifndef ELLIPSE_AREA_H
#define ELLIPSE_AREA_H

// function to compute area of an ellipse

double ellipseArea(double, double);

#endif
```

The directives **#ifndef** (if not defined), etc., serve to ensure that the prototype is not included multiple times. If **ELLIPSE_AREA_H** is already defined, the declaration is skipped.

Putting functions in separate files, continued

Then the header file is included (note use of `" "` rather than `< >`) in all files where the function is called:

```
#include <iostream>
#include "ellipseArea.h"
using namespace std;
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}
```

(`ellipseArea.h` does not have to be included in the file `ellipseArea.cc` where the function is defined.)

Passing arguments by value

Consider a function that tries to change the value of an argument:

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

It won't work:

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;  // x still = 1
```

This is because the argument is passed 'by value'. Only a copy of the value of **x** is passed to the function.

In general this is a Good Thing. We don't want arguments of functions to have their values changed unexpectedly.

Sometimes, however, we want to return modified values of the arguments. But a function can only return a single value.

Passing arguments by reference

We can change the argument's value passing it 'by reference'.
To do this we include an `&` after the argument type in the function's prototype and in its definition (but no `&` in the function call):

```
void tryToChangeArg(int&);          // prototype

void tryToChangeArg(int& x){        // definition
  x = 2*x;
}

int main(){
  int x = 1;
  tryToChangeArg(x);
  cout << "now x = " << x << endl;  // now x = 2
}
```

Argument passed by reference must be a variable, e.g.,
`tryToChangeArg(7);` will not compile.

Default arguments

Sometimes it is convenient to specify default arguments for functions in their declaration:

```
double line(double x, double slope=1, double offset=0);
```

The function is then defined as usual:

```
double line(double x, double slope, double offset){
  return x*slope  +  offset;
}
```

We can then call the function with or without the defaults:

```
y = line (x, 3.7, 5.2);   // here slope=3.7, offset=5.2
y = line (x, 3.7);        // uses offset=0;
y = line (x);             // uses slope=1, offset=0
```

Function overloading

We can define versions of a function with different numbers or types of arguments (signatures). This is called function overloading:

```
double cube(double);
double cube (double x){
  return x*x*x;
}

double cube(float);
double cube (float x){
  double xd = static_cast<double>(x);
  return xd*xd*xd;
}
```

Return type can be same or different; argument list must differ in number of arguments or in their types.

Function overloading, cont.

When we call the function, the compiler looks at the signature of the arguments passed and figures out which version to use:

```
float x;
double y;
double z = cube(x);  // calls cube(float) version
double z = cube(y);  // calls cube(double) version
```

This is done e.g. in the standard math library **`cmath`**. There is a version of **`sqrt`** that takes a **`float`** (and returns **`float`**), and another that takes a **`double`** (and returns **`double`**).

Note it is not sufficient if functions differ only by return type -- they must differ in their argument list to be overloaded.

Operators (**`+`**, **`-`**, etc.) can also be overloaded. More later.

Writing to and reading from files

Here is a simple program that opens an existing file in order to read data from it:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(){
  // create an ifstream object (name arbitrary)...
  ifstream myInput;
  // Now open an existing file...
  myInput.open("myDataFile.txt");
  // check that operation worked...
  if ( myInput.fail() ) {
    cout << "Sorry, couldn't open file" << endl;
    exit(1);      // from cstdlib
  }
  ...
```

Reading from an input stream

The input file stream object is analogous to **`cin`**, but instead of getting data from the keyboard it reads from a file. Note use of "dot" to call the ifstream's "member functions", **`open`**, **`fail`**, etc.

Suppose the file contains columns of numbers like

```
1.0   7.38  0.43
2.0   8.59  0.52
3.0   9.01  0.55
...
```

We can read in these numbers from the file:

```
double x, y, z;
for(int i=1; i<=numLines; i++){
  myInput >> x >> y >> z;
  cout << "Read " << x << " " << y << " " << z << endl;
}
```

This loop requires that we know the number of lines in the file.

Reading to the end of the file

Often we don't know the number of lines in a file ahead of time. We can use the "end of file" (`eof`) function:

```
double x, y, z;
int line = 0;
while ( !myInput.eof() ){
  myInput >> x >> y >> z;
  if ( !myInput.eof() ) {
    line++;
    cout << x << " " << y << " " << z << endl;
  }
}
cout << lines << " lines read from file" << endl;
...
myInput.close();      // close when finished
```

Note some gymnastics needed to avoid getting last line twice.

Writing data to a file

We can write to a file with an **ofstream** object:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(){
  // create an ofstream object (name arbitrary)...
  ofstream myOutput;
  // Now open a new file...
  myOutput.open("myDataFile.txt");
  // check that operation worked...
  if ( myOutput.fail() ) {
    cout << "Sorry, couldn't open file" << endl;
    exit(1);       // from cstdlib
  }
  ...
```

Writing data to a file, cont.

Now the **`ofstream`** object behaves like **`cout`**:

```
for (int i=1; i<=n; i++){
  myOutput << i << "\t" << i*i << endl;
}
```

Note use of tab character **`\t`** for formatting (could also use e.g. **`"    "`** or)

Alternatively use the functions **`setf`**, **`precision`**, **`width`**, etc. These work the same way with an **`ofstream`** object as they do with **`cout`**, e.g.,

```
myOutput.setf(ios::fixed);
myOutput.precision(4);
...
```

File access modes

The previous program would overwrite an existing file.
To append an existing file, we can specify:

```
myOutput.open("myDataFile.txt", ios::app);
```

This is an example of a file access mode. Another useful one is:

```
myOutput.open("myDataFile.txt", ios::bin);
```

The data is then written as binary, not formatted. This is much more compact, but we can't check the values with an editor.

For more than one option, separate with vertical bar:

```
myOutput.open("myDataFile.txt", ios::bin | ios::app);
```

Many options, also for **`ifstream`**. Google for details.

Putting it together

Now let's put together some of what we've just seen. The program reads from a file a series of exam scores, computes the average and writes it to another file. In file **`examAve.cc`** we have

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "aveScore.h"
using namespace std;
int main(){
  // open input file
  ifstream inFile;
  inFile.open("studentScores.txt");
  if ( inFile.fail() ) {
    cerr << "Couldn't open input file" << endl;
    exit(1);
  }
              ...
```

examAve, continued

```
// open the output file
ofstream outFile;
outFile.open("averageScores.txt");
if ( outFile.fail() ) {
  cerr << "Couldn't open output file" << endl;
  exit(1);
}

while ( !inFile.eof() ){
  int studentNum;
  double test1, test2, test3;
  inFile >> studentNum >> test1 >> test2 >> test3;
  if( !inFile.eof() ){
    double ave = aveScore (test1, test2, test3);
    outFile << studentNum << "\t" << ave << endl;
  }
}
```

More `examAve`

```
  // close up
  inFile.close();
  outFile.close();
  return 0;
}
```

Now the file `aveScore.cc` contains

```
double aveScore(double a, double b, double c){
  double ave = (a + b + c)/3.0;
  return ave;
}
```

More `examAve` and `aveScore`

The header file `aveScore.h` contains

```
#ifndef AVE_SCORE_H
#define AVE_SCORE_H
double aveScore(double, double, double);
#endif AVE_SCORE_H
```

We compile and link the program with

```
g++ -o examAve examAve.cc aveScore.cc
```

The input data file `studentScores.txt` might contain

```
1      73     65     68
2      52     45     44
3      83     85     91
```

etc. The example is trivial but we can generalize this to very complex programs.

Arrays

An array is a fixed-length list containing variables of the same type.

Declaring an array: ***`data-type variableName`*****`[`*****`numElements`*****`];`**

```
int score[10];
double energy[50], momentum[50];
const int MaxParticles = 100;
double ionizationRate[MaxParticles];
```

The number in brackets **`[]`** gives the total number of elements,e.g. the array **`score`** above has 10 elements, numbered 0 through 9. The individual elements are referred to as

```
score[0], score[1], score[2], ..., score[9]
```

The index of an array can be any integer expression with a value from zero up to the number of elements minus 1. If you try to access **`score[10]`** this is an error!

Arrays, continued

Array elements can be initialized with assignment statements and otherwise manipulated in expressions like normal variables:

```
const int NumYears = 50;
int year[NumYears];
for(int i=0; i<NumYears; i++){
  year[i] = i + 1960;
}
```

Note that C++ arrays always begin with zero, and the last element has an index equal to the number of elements minus one.

This makes it awkward to implement, e.g., n-dimensional vectors that are naturally numbered $\boldsymbol{x} = (x_1, \ldots, x_n)$.

In the C++ 98 standard, the size of the array must be known at compile time. In C99 (implemented by gcc), array length can be variable (set at run time). See also “dynamic” arrays (later).

Multidimensional arrays

An array can also have two or more indices. A two-dimensional array is often used to store the values of a matrix:

```
const int numRows = 2;
const int numColumns = 3;
double matrix[numRows][numColumns];
```

Again, notice that the array size is 2 by 3, but the row index runs from 0 to 1 and the column index from 0 to 2.

The elements are stored in memory in the order:

```
matrix[i][j], matrix[i][j+1], etc.
```

Usually we don't need to know how the data are stored internally. (Ancient history: in FORTRAN, the left-most index gave adjacent elements in memory.)

Initializing arrays

We can initialize an array together with the declaration:

```
int myArray[5] = {2, 4, 6, 8, 10};
```

Similar for multi-dimensional arrays:

```
double matrix[numRows][numColumns] =
                     { {3, 7, 2}, {2, 5, 4} };
```

In practice we will usually initialize arrays with assignment statements.

Example: multiplication of matrix and vector

```
// Initialize vector x and matrix A
const int n = 5;
double x[n];
double A[n][n];
for (int i=0; i<n; i++){
  x[i] = someFunction(i);
  for (int j=0; j<n; j++){
    A[i][j] = anotherFunction(i, j);
  }
}

// Now find y = Ax
double y[n];
for (int i=0; i<n; i++){
  y[i] = 0.0;
  for (int j=0; j<n; j++){
    y[i] += A[i][j] * x[j];
  }
}
```

Passing arrays to functions

Suppose we want to use an array **`a`** of length **`len`** as an argument of a function. In the function's declaration we say, e.g.,

```
double sumElements(double a[], int len);
```

We don't need to specify the number of elements in the prototype, but we often pass the length into the function as an **`int`** variable.

Then in the function definition we have, e.g.,

```
double sumElements(double a[], int len){
  double sum = 0.0;
  for (int i=0; i<len; i++){
    sum += a[i];
  }
  return sum;
}
```

Passing arrays to functions, cont.

Then to call the function we say, e.g.,

```
double s = sumElements(myMatrix, itsLength);
```

Note there are no brackets for **`myMatrix`** when we pass it to the function.

You could, however, pass **`myMatrix[i]`**, not as a matrix but as a **`double`**, i.e., the i^{th} element of **`myMatrix`**. For example,

```
double x = sqrt(myMatrix[i]);
```

Passing arrays to functions

When we pass an array to a function, it works as if passed by reference, even though we do not use the `&` notation as with non-array variables. (The array name is a "pointer" to the first array element. More on pointers later.)

This means that the array elements could wind up getting their values changed:

```
void changeArray (double a[], int len){
  for(int i=0; i<len; i++){
    a[i] *= 2.0;
  }
}

int main(){
  ...
  changeArray(a, len);    // elements of a doubled
```

Passing multidimensional arrays to functions

When passing a multidimensional array to a function, we need to specify in the prototype and function definition the number of elements for all but the left-most index:

```
void processImage(int image[][numColumns],
                  int numRows, int numColumns){
   ...
```

(But we still probably need to pass the number of elements for both indices since their values are needed inside the function.)

Pointers

A pointer variable contains a memory address. It ‘points’ to a location in memory. To declare a pointer, use a star, e.g.,

```
int* iPtr;
double * xPtr;
char *c;
float *x, *y;
```

Note some freedom in where to put the star. I prefer the first notation as it emphasizes that **`iPtr`** is of type “pointer to **`int`**”.

(But in **`int* iPtr, jPtr;`** only **`iPtr`** is a pointer--need 2 stars.)

Name of pointer variable can be any valid identifier, but often useful to choose name to show it’s a pointer (suffix **`Ptr`**, etc.).

Pointers: the `&` operator

Suppose we have a variable `i` of type `int`:

```
int i = 3;
```

We can define a pointer variable to point to the memory location that contains `i`:

```
int* iPtr = &i;
```

Here `&` means "address of". Don't confuse it with the `&` used when passing arguments by reference.

Initializing pointers

A statement like

```
int* iPtr;
```

declares a pointer variable, but does not initialize it. It will be pointing to some "random" location in memory. We need to set its value so that it points to a location we're interested in, e.g., where we have stored a variable:

```
iPtr = &i;
```

(just as ordinary variables must be initialized before use).

Dereferencing pointers: the `*` operator

Similarly we can use a pointer to access the value of the variable stored at that memory location. E.g. suppose `iPtr = &i;` then

```
int iCopy = *iPtr;     // now iCopy equals i
```

This is called 'dereferencing' the pointer. The * operator means "value stored in memory location being pointed to".

If we set a pointer equal to zero (or `NULL`) it points to nothing. (The address zero is reserved for null pointers.)

If we try to dereference a null pointer we get an error.

Why different kinds of pointers?

Suppose we declare

```
int* iPtr;         //  type "pointer to int"
float* fPtr;       //  type "pointer to float"
double* dPtr;      //  type "pointer to double"
```

We need different types of pointers because in general, the different data types (`int, float, double`) take up different amounts of memory. If declare another pointer and set

```
int* jPtr = iPtr + 1;
```

then the `+1` means “plus one unit of memory address for `int`”, i.e., if we had `int` variables stored contiguously, `jPtr` would point to the one just after `iPtr`.

But the types `float`, `double`, etc., take up different amounts of memory, so the actual memory address increment is different.

Passing pointers as arguments

When a pointer is passed as an argument, it divulges an address to the called function, so the function can change the value stored at that address:

```
void passPointer(int* iPtr){
  *iPtr += 2;           // note *iPtr on left!
}

...
int i = 3;
int* iPtr = &i;
passPointer(iPtr);
cout << "i = " << i << endl;    // prints i = 5
passPointer(&i);              // equivalent to above
cout << "i = " << i << endl;    // prints i = 7
```

End result same as pass-by-reference, syntax different. (Usually pass by reference is the preferred technique.)

Pointers vs. reference variables

A reference variable behaves like an alias for a regular variable.
To declare, place `&` after the type:

```
int i = 3;
int& j = i;          // j is a reference variable
j = 7;
cout << "i = " << i << endl;  // prints i = 7
```

Passing a reference variable to a function is the same as passing a normal variable by reference.

```
void passReference(int& i){
  i += 2;
}

passReference(j);
cout << "i = " << i << endl;  // prints i = 9
```

What to do with pointers

You can do lots of things with pointers in C++, many of which result in confusing code and hard-to-find bugs.

One of the main differences between Java and C++: Java doesn't have pointer variables (generally seen as a Good Thing).

One interesting use of pointers is that the name of an array is a pointer to the zeroth element in the array, e.g.,

```
double a[3] = {5, 7, 9};
double zerothVal = *a;        // has value of a[0]
```

The main usefulness of pointers for us is that they will allow us to allocate memory (create variables) dynamically, i.e., at run time, rather than at compile time.

Strings (the old way)

A string is a sequence of characters. In C and in earlier versions of C++, this was implemented with an array of variables of type **`char`**, ending with the character **`\0`** (counts as a single ‘null’ character):

```
char aString[] = "hello";  // inserts \0 at end
```

The **`cstring`** library (**`#include <cstring>`**) provides functions to copy strings, concatenate them, find substrings, etc. E.g.

```
char* strcpy(char* target, const char* source);
```

takes as input a string **`source`** and sets the value of a string **`target`**, equal to it. Note **`source`** is passed as **`const`** -- it can’t be changed.

You will see plenty of code with old “C-style” strings, but there is now a better way: the **`string`** class (more on this later).

Example with `strcpy`

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
  char string1[] = "hello";
  char string2[50];
  strcpy(string2, string1);
  cout << "string2:  " << string2 << endl;
  return 0;
}
```

No need to count elements when initializing string with `"   "`.

Also `\0` is automatically inserted as last character.

Program will print: `string2 = hello`

Classes

A class is something like a user-defined data type. The class must be declared with a statement of the form:

```
class MyClassName {
  public:
    public function prototypes and
    data declarations;
    ...
  private:
    private function prototypes and
    data declarations;
    ...
};
```

Typically this would be in a file called `MyClassName.h` and the definitions of the functions would be in `MyClassName.cc`.

Note the semi-colon after the closing brace.

For class names often use UpperCamelCase.

A simple class: `TwoVector`

We might define a class to represent a two-dimensional vector:

```
class TwoVector {
  public:
    TwoVector();
    TwoVector(double x, double y);
    double x();
    double y();
    double r();
    double theta();
    void setX(double x);
    void setY(double y);
    void setR(double r);
    void setTheta(double theta);
  private:
    double m_x;
    double m_y;
};
```

Class header files

The header file must be included (**`#include "MyClassName.h"`**) in other files where the class will be used.

To avoid multiple declarations, use the same trick we saw before with function prototypes, e.g., in **`TwoVector.h`** :

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {
  public:
    ...
  private:
    ...
};

#endif
```

Objects

Recall that variables are instances of a data type, e.g.,

```
double a;     // a is a variable of type double
```

Similarly, objects are instances of a class, e.g.,

```
#include "TwoVector.h"
int main() {
  TwoVector v;  // v is an object of type TwoVector
```

(Actually, variables are also objects in C++. Sometimes class instances are called "class objects" -- distinction is not important.)

A class contains in general both:

variables, called "data members" and

functions, called "member functions" (or "methods")

Data members of a `TwoVector` object

The data members of a `TwoVector` are:

```
 ...
private:
  double m_x;
  double m_y;
```

Their values define the "state" of the object.

Because here they are declared `private`, a `TwoVector` object's values of `m_x` and `m_y` cannot be accessed directly, but only from within the class's member functions (more later).

The optional prefixes `m_` indicate that these are data members. Some authors use e.g. a trailing underscore. (Any valid identifier is allowed.)

The constructors of a `TwoVector`

The first two member functions of the `TwoVector` class are:

```
...
public:
  TwoVector();
  TwoVector(double x, double y);
```

These are special functions called constructors.

A constructor always has the same name as that of the class.

It is a function that is called when an object is created.

A constructor has no return type.

There can be in general different constructors with different signatures (type and number of arguments).

The constructors of a `TwoVector`, cont.

When we declare an object, the constructor is called which has the matching signature, e.g.,

```
TwoVector u;    // calls TwoVector::TwoVector()
```

The constructor with no arguments is called the "default constructor". If, however, we say

```
TwoVector v(1.5, 3.7);
```

then the version that takes two `double` arguments is called.

If we provide no constructors for our class, C++ automatically gives us a default constructor.

Defining the constructors of a **`TwoVector`**

In the file that defines the member functions, e.g., **`TwoVector.cc`**, we precede each function name with the class name and **`::`** (the scope resolution operator). For our two constructors we have:

```
TwoVector::TwoVector() {
  m_x = 0;
  m_y = 0;
}
TwoVector::TwoVector(double x, double y) {
  m_x = x;
  m_y = y;
}
```

The constructor serves to initialize the object.

If we already have a **`TwoVector v`** and we say

```
TwoVector w = v;
```

this calls a "copy constructor" (automatically provided).

The member functions of **`TwoVector`**

We call an object's member functions with the "dot" notation:

```
TwoVector v(1.5, 3.7);    // creates an object v
double vX = v.x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
...
```

If the class had public data members, e.g., these would also be called with a dot. E.g. if **`m_x`** and **`m_y`** were public, we could say

```
double vX = v.m_x;
```

We usually keep the data members private, and only allow the user of an object to access the data through the public member functions. This is sometimes called "data hiding".

If, e.g., we were to change the internal representation to polar coordinates, we would need to rewrite the functions **`x()`**, etc., but the user of the class wouldn't see any change.

Defining the member functions

Also in **`TwoVector.cc`** we have the following definitions:

```
double TwoVector::x() const { return m_x; }
double TwoVector::y() const { return m_y; }
double TwoVector::r() const {
  return sqrt(m_x*m_x  + m_y*m_y);
}
double TwoVector::theta() const {
  return atan2(m_y, m_x);            // from cmath
}
...
```

These are called “accessor” or “getter” functions.

They access the data but do not change the internal state of the object; therefore we include **`const`** after the (empty) argument list (more on why we want **`const`** here later).

More member functions

Also in **`TwoVector.cc`** we have the following definitions:

```
void TwoVector::setX(double x) { m_x = x; }
void TwoVector::setY(double y) { m_y = y; }
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

These are "setter" functions. As they belong to the class, they are allowed to manipulate the **`private`** data members **`m_x`** and **`m_y`**.

To use with an object, use the "dot" notation:

```
TwoVector v(1.5, 3.7);
v.setX(2.9);       // sets v's value of m_x to 2.9
```

Pointers to objects

Just as we can define a pointer to type **`int`**,

```
int* iPtr;       //  type "pointer to int"
```

we can define a pointer to an object of any class, e.g.,

```
TwoVector* vPtr;  // type "pointer to TwoVector"
```

This doesn't create an object yet!   This is done with, e.g.,

```
vPtr = new TwoVector(1.5, 3.7);
```

`vPtr` is now a pointer to our object. With an object pointer, we call member functions (and access data members) with **`->`** (not with "**`.`**"), e.g.,

```
double vX = vPtr->x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
```

Forgotten detail: the **`this`** pointer

Inside each object's member functions, C++ automatically provides a pointer called **`this`**. It points to the object that called the member function. For example, we just saw

```
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

Here the use of **`this`** is optional (but nice, since it emphasizes what belongs to whom). It can be needed if one of the function's parameters has the same name, say, **`x`** as a data member. By default, **`x`** means the parameter, not the data member; **`this->x`** is then used to access the data member.

Memory allocation

We have seen two main ways to create variables or objects:

(1) by a declaration (automatic memory allocation):

```
int i;
double myArray[10];
TwoVector v;
TwoVector* vPtr;
```

(2) using **`new`**: (dynamic memory allocation):

```
vPtr = new TwoVector();        // creates object
TwoVector* uPtr = new TwoVector();  // on 1 line
double* a = new double[n];   // dynamic array
float* xPtr = new float(3.7);
```

The key distinction is whether or not we use the **`new`** operator.

Note that **`new`** always requires a pointer to the **`new`**ed object.

The stack

When a variable is created by a "usual declaration", i.e., without **`new`**, memory is allocated on the "stack".

When the variable goes out of scope, its memory is automatically deallocated ("popped off the stack").

```
...
{
  int i = 3;           // memory for i and obj
  MyObject obj;        // allocated on the stack
  ...
}                      // i and obj go out of scope,
                       // memory freed
```

The heap

To allocate memory dynamically, we first create a pointer, e.g.,

```
MyClass* ptr;
```

`ptr` itself is a variable on the stack. Then we create the object:

```
ptr = new MyClass( constructor args );
```

This creates the object (pointed to by `ptr`) from a pool of memory called the "heap" (or "free store").

When the object goes out of scope, `ptr` is deleted from the stack, but the memory for the object itself remains allocated in the heap:

```
{
  MyClass* ptr = new MyClass();    // creates object
  ...
}   // ptr goes out of scope here -- memory leak!
```

This is called a memory leak. Eventually all of the memory available will be used up and the program will crash.

Deleting objects

To prevent the memory leak, we need to deallocate the object's memory before it goes out of scope:

```
{
  MyClass* ptr = new MyClass();    // creates an object
  MyClass* a = new MyClass[n];     // array of objects
  ...

  delete ptr;  // deletes the object pointed to by ptr
  delete [] a; // brackets needed for array of objects
}
```

For every **`new`**, there should be a **`delete`**.

For every **`new`** with brackets **`[]`**, there should be a **`delete []`** .

This deallocates the object's memory. (Note that the pointer to the object still exists until it goes out of scope.)

Dangling pointers

Consider what would happen if we deleted the object, but then still tried to use the pointer:

```
MyClass* ptr = new MyClass();    // creates an object
...
delete ptr;
ptr->someMemberFunction();       //  unpredictable!!!
```

After the object's memory is deallocated, it will eventually be overwritten with other stuff.

But the "dangling pointer" still points to this part of memory.

If we dereference the pointer, it may still give reasonable behaviour. But not for long! The bug will be unpredictable and hard to find.

Some authors recommend setting a pointer to zero after the **`delete`**. Then trying to dereference a null pointer will give a consistent error.

Static memory allocation

For completeness we should mention static memory allocation. Static objects are allocated once and live until the program stops.

```
void aFunction(){
  static bool firstCall = true;
  if (firstCall) {
    firstCall = false;
    ...                   // do some initialization
  }
  ...
}     // firstCall out of scope, but still alive
```

The next time we enter the function, it remembers the previous value of the variable `firstCall`. (Not a very elegant initialization mechanism but it works.)

This is only one of several uses of the keyword `static` in C++.

Operator overloading

Suppose we have two **`TwoVector`** objects and we want to add them. We could write an **`add`** member function:

```
TwoVector TwoVector::add(TwoVector& v){
  double cx = this->m_x + v.x();
  double cy = this->m_y + v.y();
  TwoVector c(cx, cy);
  return c;
}
```

To use this function we would write, e.g.,

```
TwoVector u = a.add(b);
```

It would be much easier if would could simply use **`a+b`**, but to do this we need to define the **`+`** operator to work on **`TwoVector`**s.

This is called operator overloading. It can make manipulation of the objects more intuitive.

Overloading an operator

We can overload operators either as member or non-member functions. For member functions, we include in the class declaration:

```
class TwoVector {
  public:
    ...
    TwoVector operator+ (const TwoVector&);
    TwoVector operator- (const TwoVector&);
    ...
```

Instead of the function name we put the keyword **`operator`** followed by the operator being overloaded.

When we say **`a+b`**, **`a`** calls the function and **`b`** is the argument.

The argument is passed by reference (quicker) and the declaration uses **`const`** to protect its value from being changed.

Defining an overloaded operator

We define the overloaded operator along with the other member functions, e.g., in **`TwoVector.cc`**:

```
TwoVector TwoVector::operator+ (const TwoVector& b) {
  double cx = this->m_x + b.x();
  double cy = this->m_y + b.y();
  TwoVector c(cx, cy);
  return c;
}
```

The function adds the x and y components of the object that called the function to those of the argument.

It then returns an object with the summed x and y components.

Recall we declared **`x()`** and **`y()`**, as **`const`**. We did this so that when we pass a **`TwoVector`** argument as **`const`**, we're still able to use these functions, which don't change the object's state.

Overloaded operators: asymmetric arguments

Suppose we want to overload `*` to allow multiplication of a `TwoVector` by a scalar value:

```
TwoVector TwoVector::operator* (double b) {
  double cx = this->m_x * b;
  double cy = this->m_y * b;
  TwoVector c(cx, cy);
  return c;
}
```

Given a `TwoVector v` and a `double s` we can say e.g. `v = v*s;`

But how about `v = s*v;` ???

No! `s` is not a `TwoVector` object and cannot call the appropriate member function (first operand calls the function).

We didn't have this problem with `+` since addition commutes.

Overloading operators as non-member functions

We can get around this by overloading `*` with a non-member function.

We could put the declaration in `TwoVector.h` (since it is related to the class), but outside the class declaration.

We define two versions, one for each order:

```
TwoVector operator* (const TwoVector&, double b);
TwoVector operator* (double b, const TwoVector&);
```

For the definitions we have e.g. (other order similar):

```
TwoVector operator* (double b, const TwoVector& a) {
  double cx = a.x() * b;
  double cy = a.y() * b;
  TwoVector c(cx, cy);
  return c;
}
```

Restrictions on operator overloading

You can only overload C++'s existing operators:

Unary:
```
+   -   *   &   ~   !   ++   --   ->   ->*
```

Binary:
```
+   -   *   /   &   ^   &   |   <<   >>
+=  -=  *=  /=  %=  ^=  &=  |=  <<=  >>=
<   <=  >   >=  ==  !=  &&  ||  ,   []   ()
new    new[]    delete    delete[]
```

You cannot overload: `.` `.*` `?:` `::`

Operator precedence stays same as in original.

Too bad -- cannot replace `pow` function with `**` since this isn't allowed, and if we used `^` the precedence would be very low.

Recommendation is only to overload operators if this leads to more intuitive code. Remember you can still do it all with functions.

A different "static": static members

Sometimes it is useful to have a data member or member function associated not with individual objects but with the class as a whole.

An example is a variable that counts the number of objects of a class that have been created.

These are called static member functions/variables (yet another use of the word static -- better would be "class-specific"). To declare:

```
class TwoVector {
  public:
    ...
    static int totalTwoVecs();
  private:
    static int m_counter;
  ...
};
```

Static members, continued

Then in `TwoVector.cc` (note here no keyword `static`):

```
int TwoVector::m_counter = 0;  // initialize

TwoVector::TwoVector(double x, double y){
  m_x = x;
  m_y = y;
  m_counter++;    // in all constructors
}

int TwoVector::totalTwoVecs() { return m_counter; }
```

Now we can count our `TwoVector`s. Note the function is called with ***`class-name::`*** and then the function name. It is connected to the class, not to any given object of the class:

```
TwoVector a, b, c;
int vTot = TwoVector::totalTwoVecs();
cout << vTot << endl;          // prints 3
```

Oops #1: digression on destructors

The **`totalTwoVec`** function doesn't work very well, since we also create a new **`TwoVector`** object when, e.g., we use the overloaded **`+`**. The local object itself dies when it goes out of scope, but the counter still gets incremented when the constructor is executed.

We can remedy this with a destructor, a special member function called automatically just before its object dies. The name is **`~`** followed by the class name. To declare in **`TwoVector.h`**:

```
public:
  ~TwoVector();      // no arguments or return type
```

And then we define the destructor in **`TwoVector.cc`** :

```
TwoVector::~TwoVector(){  m_counter--;  }
```

Destructors are good places for clean up, e.g., deleting anything created with **`new`** in the constructor.

Oops #2: digression on copy constructors

The `totalTwoVec` function still doesn't work very well, since we should count an extra `TwoVector` object when, e.g., we say

```
TwoVector v;         // this increments m_counter
TwoVector u = v;     // oops, m_counter stays same
```

When we create/initialize an object with an assignment statement, this calls the copy constructor, which by default just makes a copy.

We need to write our own copy constructor to increment `m_counter`. To declare (together with the other constructors):

```
TwoVector(const TwoVector&);  // unique signature
```

It gets defined in `TwoVector.cc` :

```
TwoVector(const TwoVector& v) {
  m_x = v.x();  m_y = v.y();
  m_counter++;
}
```

Class templates

We defined the **`TwoVector`** class using **`double`** variables. But in some applications we might want to use **`float`**.

We could cut/paste to create a **`TwoVector`** class based on **`float`**s (very bad idea -- think about code maintenance).

Better solution is to create a class template, and from this we create the desired classes.

```
template <class T>        // T stands for a type
class TwoVector {
  public:
    TwoVector(T, T);      // put T where before we
    T x();                // had double
    T y();
    ...
};
```

Defining class templates

To define the class's member functions we now have, e.g.,

```
template <class T>
TwoVector<T>::TwoVector(T x, T y){
  m_x = x;
  m_y = y;
  m_counter++;
}

template <class T>
T TwoVector<T>::x(){ return m_x; }

template <class T>
void TwoVector<T>::setX(T x){
  m_x = x;
}
```

With templates, class declaration must be in same file as function definitions (put everything in `TwoVector.h`).

Using class templates

To use a class template, insert the desired argument:

```
TwoVector<double> dVec;  // creates double version

TwoVector<float> fVec;   // creates float version
```

`TwoVector` is no longer a class, it's only a template for classes.

`TwoVector<double>` and **`TwoVector<float>`** are classes (sometimes called "template classes", since they were made from class templates).

Class templates are particularly useful for container classes, such as vectors, stacks, linked lists, queues, etc. We will see this later in the Standard Template Library (STL).

The Standard C++ Library

We've already seen parts of the standard library such as **`iostream`** and **`cmath`**. Here are some more:

What you **`#include`**	What it does
`<algorithm>`	useful algorithms (sort, search, ...)
`<complex>`	complex number class
`<list>`	a linked list
`<stack>`	a stack (push, pop, etc.)
`<string>`	proper strings (better than C-style)
`<vector>`	often used instead of arrays

Most of these define classes using templates, i.e., we can have a vector of objects or of type **`double, int, float`**, etc. They form what is called the Standard Template Library (STL).

Using `vector`

Here is some sample code that uses the **`vector`** class. Often a **`vector`** is better than an array.

```
#include <vector>
using namespace std;
int main() {
  vector<double> v;          // uses template
  double x = 3.2;
  v.push_back(x);            // element 0 is 3.2
  v.push_back(17.0);         // element 1 is 17.0
  vector<double> u = v;      // assignment
  int len = v.size();
  for (int i=0; i<len; i++){
    cout << v[i] << endl;    // like an array
  }
  v.clear();                   // remove all elements
  ...
```

Sorting elements of a vector

Here is sample code that uses the **`sort`** function in **`algorithm`**:

```
#include <vector>
#include <algorithm>
using namespace std;

bool descending(double x, double y){ return (x>y); }

int main() {
...

// u, v are unsorted vectors; overwritten by sort.
// Default sort is ascending; also use user-
// defined comparison function for descending order.

   sort(u.begin(), u.end());
   sort(v.begin(), v.end(), descending);
```

Iterators

To loop over the elements of a vector **`v`**, we could do this:

```
vector<double> v = ...              // define vector v
for (int i=0; i<v.size(); i++){
  cout << v[i] << endl;
}
```

Alternatively, we can use an iterator, which is defined by the vector class (and all of the STL container classes):

```
vector<double> v = ...              // define vector v
vector<double>::iterator it;
for (it = v.begin(); it != v.end(); ++it){
  cout << *it << endl;
}
```

`vector`'s **`begin`** and **`end`** functions point to the first and last elements.

`++` tells the iterator to go to the next element.

`*` gives the object (vector element) pointed to (note no index used).

Using `string`

Here is some sample code that uses the **`string`** class (much better than C-style strings):

```
#include <string>
using namespace std;
int main() {
  string a, b, c;
  string s = "hello";
  a = s;                  // assignment
  int len = s.length();       // now len = 5
  bool sEmpty = s.empty();    // now sEmpty = false
  b = s.substring(0,2);       // first position is 0
  cout << b << endl;          // prints hel
  c = s + " world";           // concatenation
  s.replace(2, 3, "j!");      // replace 3 characters
                              // starting at 2 with j!
  cout << s << endl;          // hej!
  ...
```

Inheritance

Often we define a class which is similar to an existing one. For example, we could have a class

```
class Animal {
  public:
    double weight();
    double age();
    ...
  private:
    double m_weight;
    double m_age;
    ...
};
```

Related classes

Now suppose the objects in question are dogs. We want

```
class Dog {
  public:
    double weight();
    double age();
    bool hasFleas();
    void bark();
  private:
    double m_weight;
    double m_age;
    bool m_hasFleas;
    ...
};
```

`Dog` contains some (perhaps many) features of the **`Animal`** class but it requires a few extra ones.

The relationship is of the form "X is a Y": a dog is an animal.

Inheritance

Rather than redefine a separate Dog class, we can derive it from Animal. To do this we declare in **`Dog.h`**

```
#include "Animal.h"
class Dog : public Animal {
  public:
    bool hasFleas();
    void bark();
    ...
  private:
    bool m_hasFleas;
    ...
};
```

`Animal` is called the “base class”, **`Dog`** is the “derived class”.

`Dog` inherits all of the public (and “protected”) members of **`Animal`**. We only need to define **`hasFleas()`**, **`bark()`**, etc.

Polymorphism, virtual functions, etc.

We might redefine a member function of **`Animal`** to do or mean something else in **`Dog`**. This is function "overriding". (Contrast this with function overloading.)

We could have **`age()`** return normal years for **`Animal`**, but "dog years" for **`Dog`**. This is an example of polymorphism. The function takes on different forms, depending on the type of object calling it.

We can also declare functions in the base class as "pure virtual" (or "abstract"). In the declaration use the keyword **`virtual`** and set equal to zero; we do not supply any definition for the function in the base class:

```
virtual double age() = 0;
```

This would mean we cannot create an **`Animal`** object. A derived class must define the function if it is to create objects.

Compiling and linking with `gmake`

For our short test programs it was sufficient to put the compile and link commands in a short file (e.g. `build.sh`).

For large programs with many files, however, compiling and linking can take a long time, and we should therefore recompile only those files that have been modified.

This can be done with the Unix program `make` (gnu version `gmake`).

Homepage `www.gnu.org/software/make`

Manual ~150 pages (many online mini-tutorials).

Widely used in High Energy Physics (and elsewhere).

Why we use `gmake`

Suppose we have `hello.cc` :

```
#include "goodbye.h"
int main() {
  cout << "Hello world" << endl;
  goodbye();
}
```

as well as `goodbye.cc` :

```
#include "goodbye.h"
using namespace std;
void goodbye() {
  cout << "Good-bye world" << endl;
}
```

and its prototype in `goodbye.h` .

Simple example without `gmake`

Usually we compile with

```
g++ -o hello hello.cc goodbye.cc
```

which is really shorthand for compiling and linking steps:

```
g++ -c hello.cc
g++ -c goodbye.cc
g++ -o hello hello.o goodbye.o
```

Now suppose we modify **`goodbye.cc`**. To rebuild, really we only need to recompile this file.

But in general it's difficult to keep track of what needs to be recompiled, especially if we change a header file.

Using date/time information from the files plus user supplied information, **`gmake`** recompiles only those files that need to be and links the program.

Simple example with **`gmake`**

The first step is to create a "makefile". **`gmake`** looks in the current directory for the makefile under the names **`GNUmakefile`**, **`makefile`** and **`Makefile`** (in that order).

The makefile can contain several types of statements, the most important of which is a "rule". General format of a rule:

```
target : dependencies
         command
```

The target is usually the name of a file we want to produce and the dependencies are the other files on which the target depends.

On the next line there is a command which must always be preceded by a tab character (spaces no good). The command tells **`gmake`** what to do to produce the target.

Simple example with `gmake`, cont.

In our example we create a file named **`GNUmakefile`** with:

```
hello : hello.o goodbye.o
      g++ -o hello hello.o goodbye.o
hello.o : hello.cc goodbye.h
      g++ -c hello.cc
goodbye.o : goodbye.cc goodbye.h
      g++ -c goodbye.cc
```

If we type **`gmake`** without an argument, then the first target listed is taken as the default, i.e., to build the program, simply type

`gmake` or **`gmake hello`**

We could also type e.g.

`gmake goodbye.o`

if we wanted only to compile **`goodbye.cc`**.

`gmake` refinements

In the makefile we can also define variables (i.e., symbols). E.g., rather than repeating **`hello.o goodbye.o`** we can define

```
objects = hello.o goodbye.o

hello : $(objects)
      g++ -o hello $(objects)
...
```

When **`gmake`** encounters **`$(objects)`** it makes the substitution.

We can also make **`gmake`** figure out the command. We see that **`hello.o`** depends on a source file with suffix **`.cc`** and a header file with suffix **`.h`**. Provided certain defaults are set up right, it will work if we say e.g.

```
hello.o : hello.cc goodbye.h
```

`gmake` for experts

makefiles can become extremely complicated and cryptic.

Often they are hundreds or thousands of lines long.

Often they are themselves not written by "humans" but rather constructed by an equally obscure shell script.

The goal here has been to give you some feel for what `gmake` does and how to work with makefiles provided by others.

Often software packages are distributed with a makefile. You might have to edit a few lines depending on the local set up (probably explained in the comments) and then type `gmake`.

We will put some simple and generalizable examples on the course web site.

Debugging your code

You should write and test your code in short incremental steps. Then if something doesn't work you can take a short step back and figure out the problem.

For every class, write a short program to test its member functions.

You can go a long way with `cout`. But, to really see what's going on when a program executes, it's useful to have a debugging program.

The current best choice for us is probably `ddd` (DataDisplayDebugger) which is effectively free (gnu license).

`ddd` is actually an interface to a lower level debugging program, which can be `gdb`. If you don't have `ddd` installed, try `xxgdb`.

Using **`ddd`**

The **`ddd`** homepage is **`www.gnu.org/software/ddd`**

There are extensive online tutorials, manuals, etc.

To use **`ddd`**, you must compile your code with the **`-g`** option:

```
g++ -g -o MyProg MyProg.cc
```

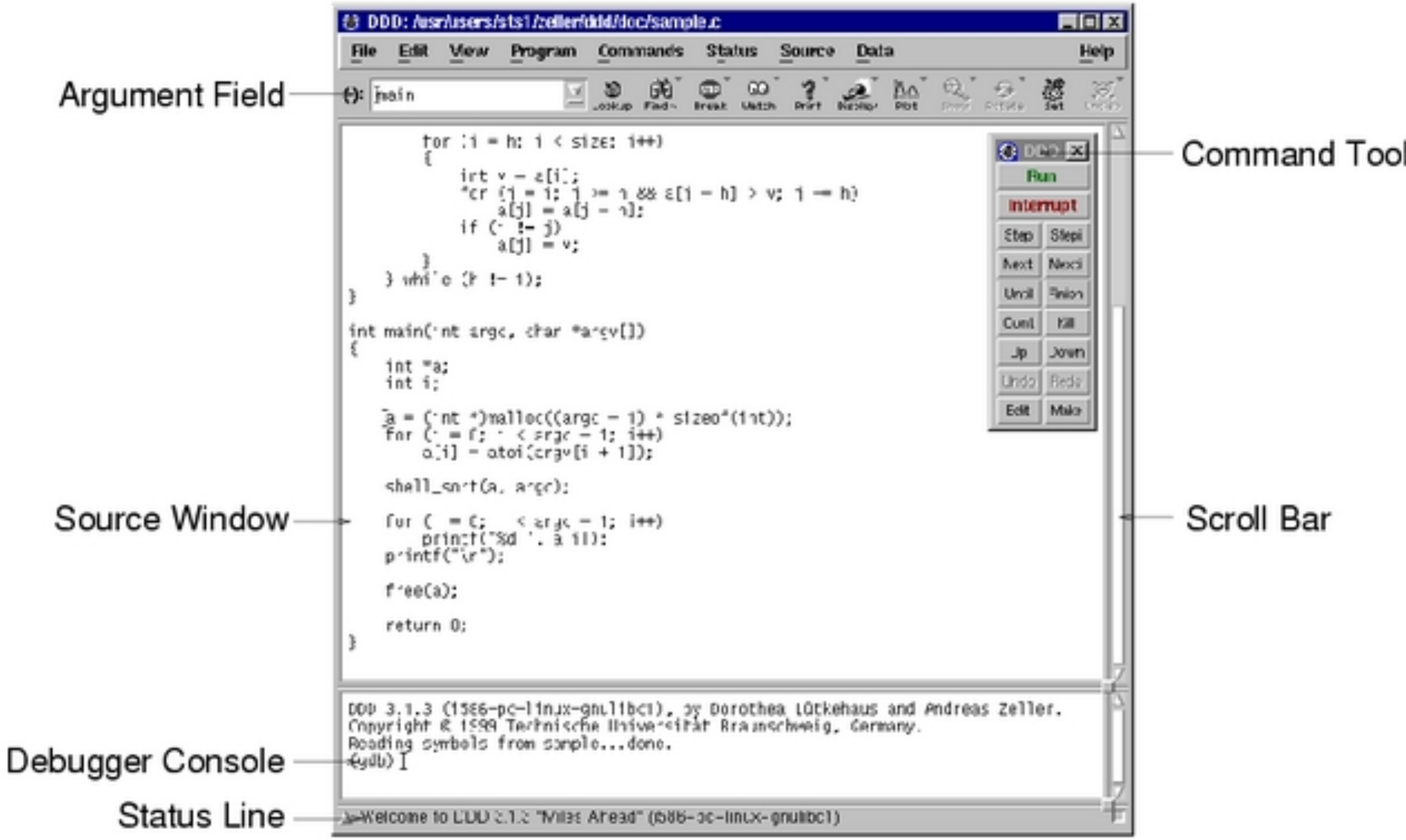
Then type

```
ddd MyProg
```

You should see a window with your program's source code and a bunch of controls.

When you start **ddd**

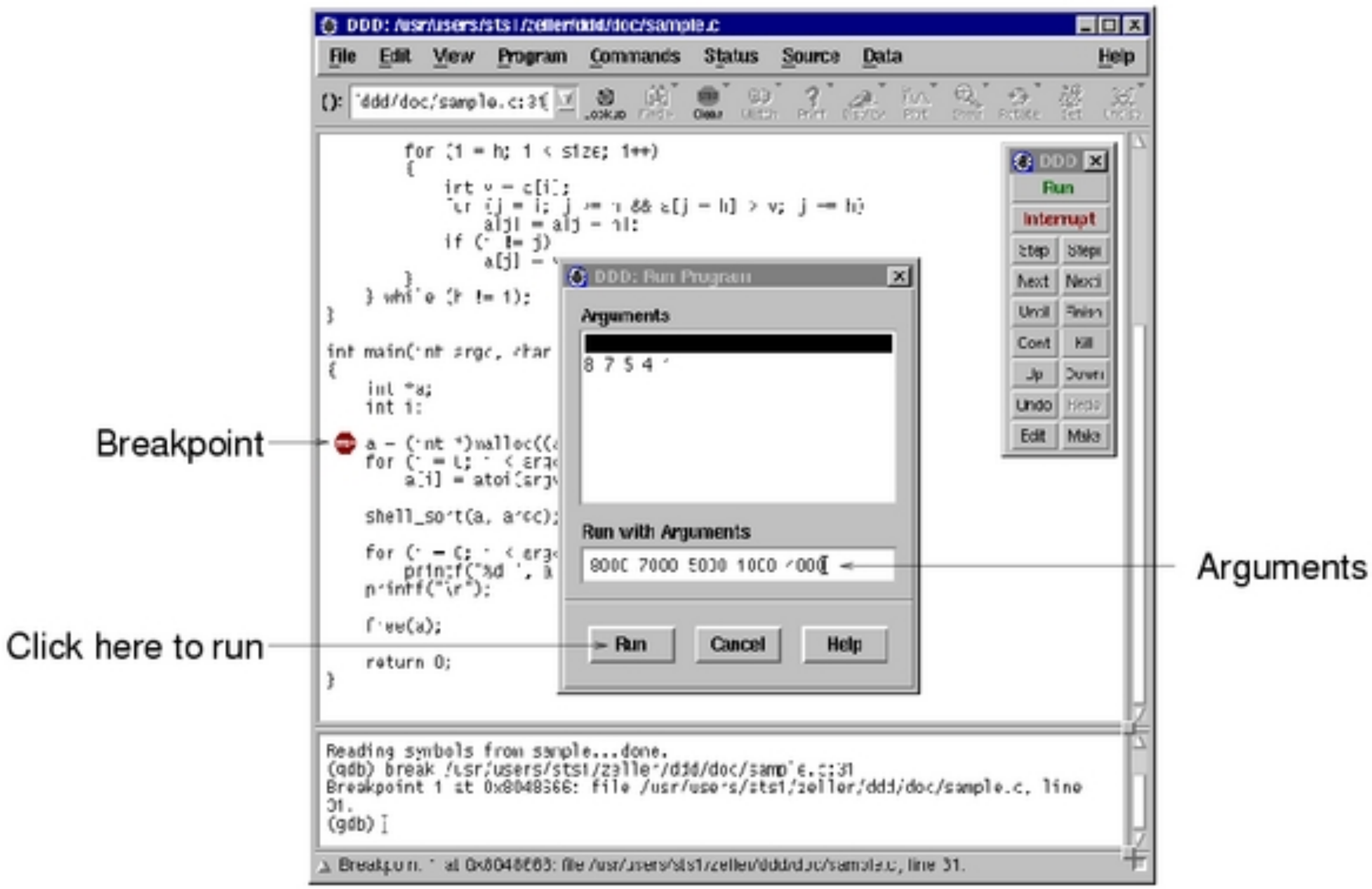
From the ddd online manual:



Initial DDD Window

Running the program

Click a line of the program and then on “Break” to set a break point. Then click on “Run”. The program will stop at the break point.



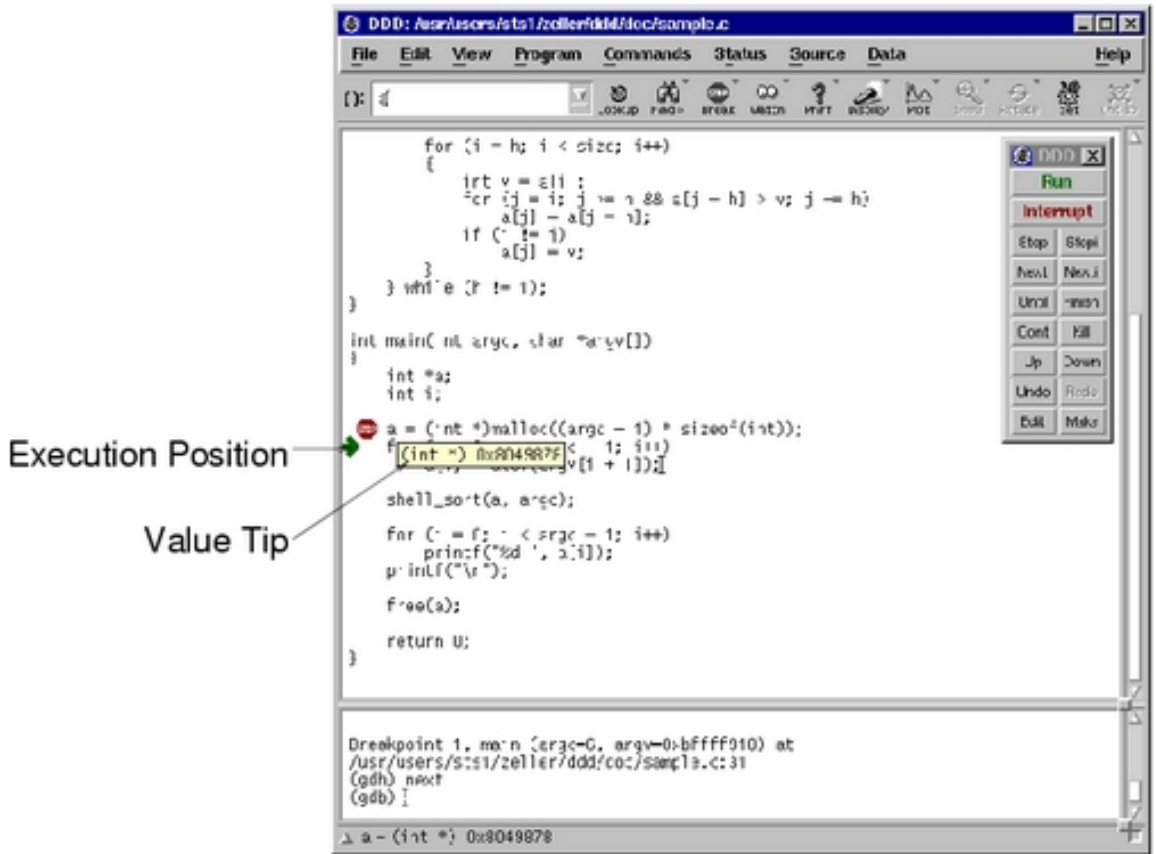
Running the Program

Stepping through the program

To execute current line, click next.

Put cursor over a variable to see its value.

For objects, select it and click Display.



Viewing Values in DDD

You get the idea.
Refer to the online
tutorial and manual.