Computing and Statistical Data Analysis
Lecture 8

STL and the Standard C++ Library

`vector, string`, ...

Inheritance (quick tour)

Some tools:

compiling/linking with `gmake`

debugging with `ddd`

Glen Cowan
RHUL Physics

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
  ...
```

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
```

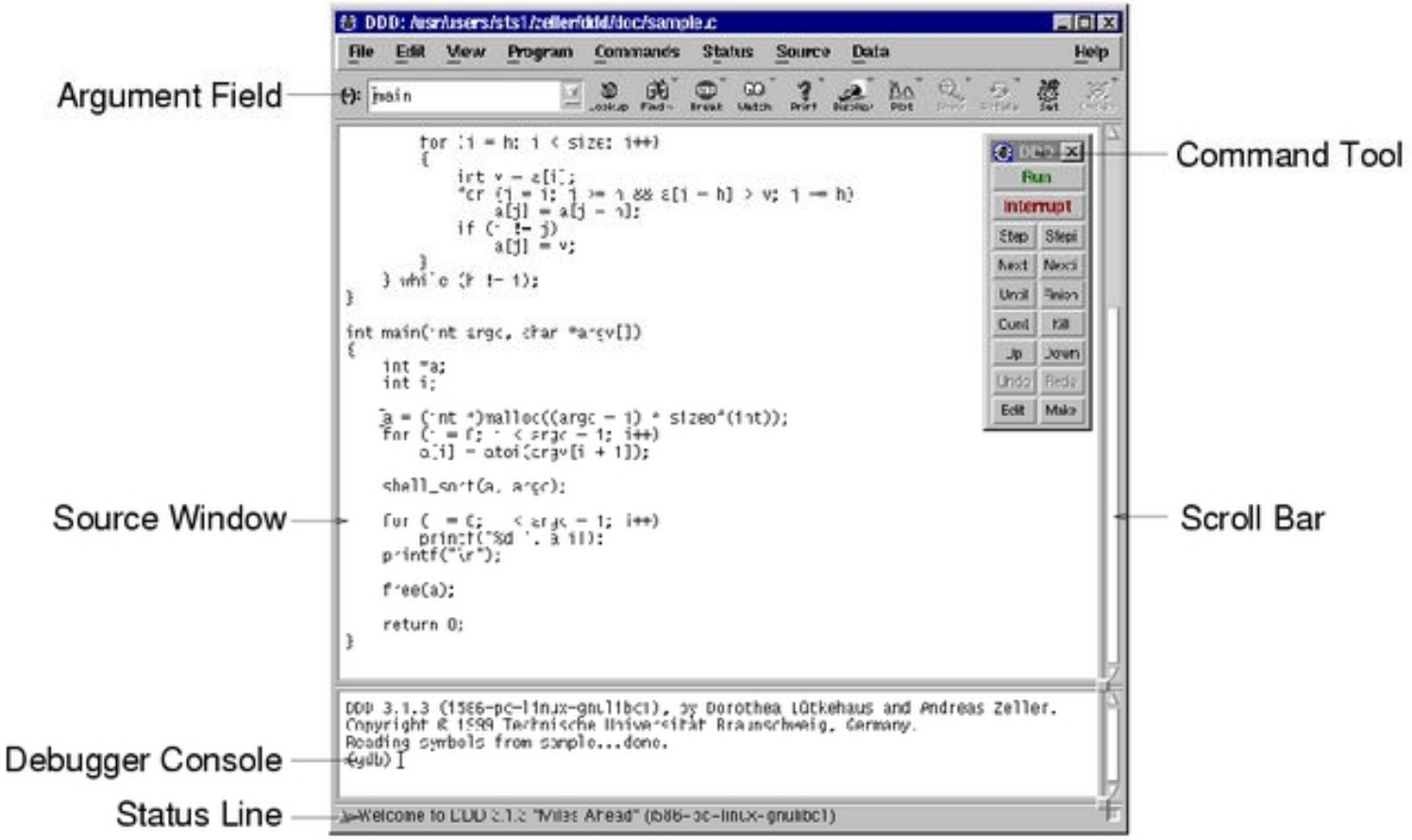
Then type

```
ddd MyProg
```

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

When you start **ddd**

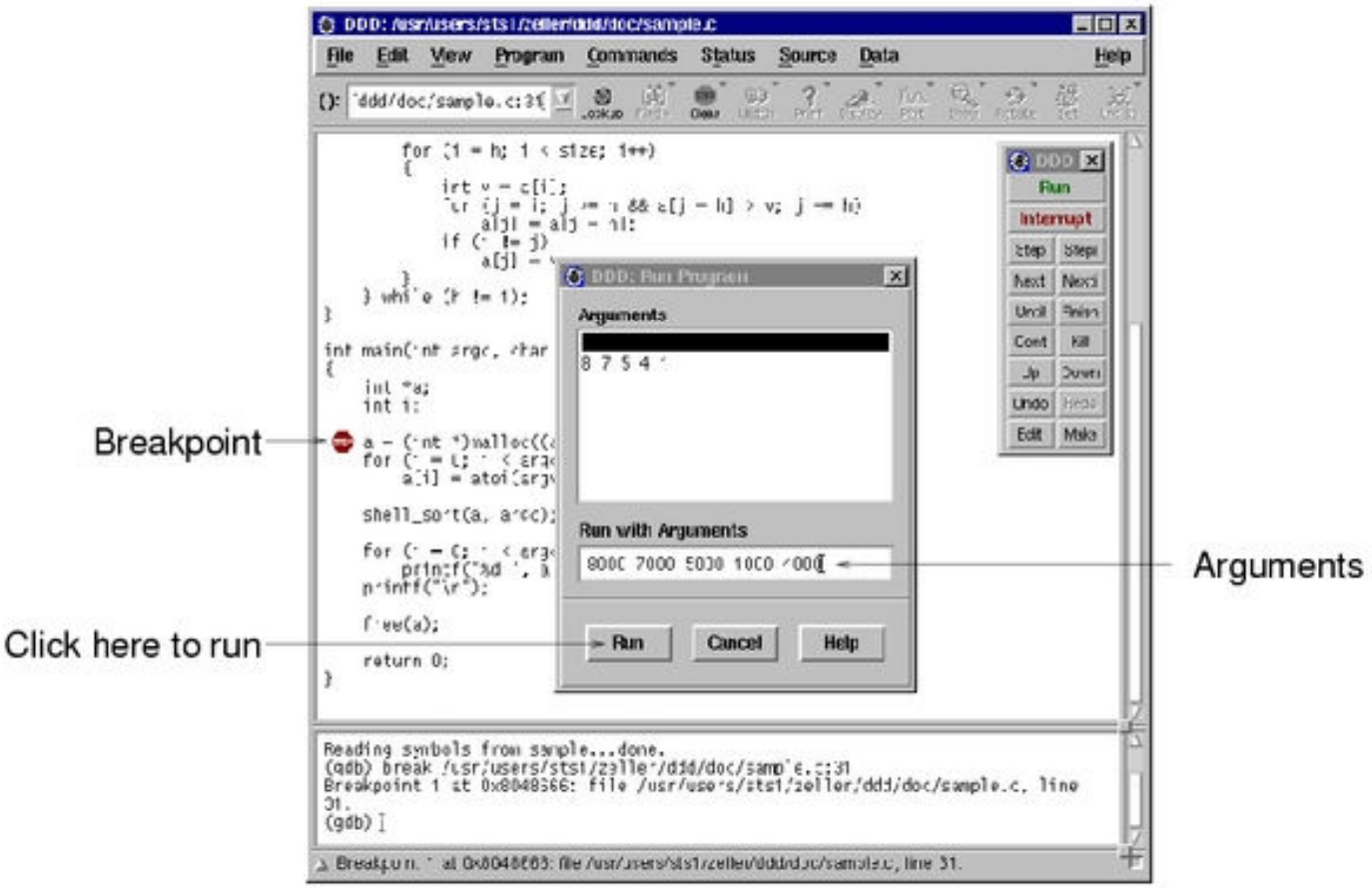
From the ddd online manual:



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.

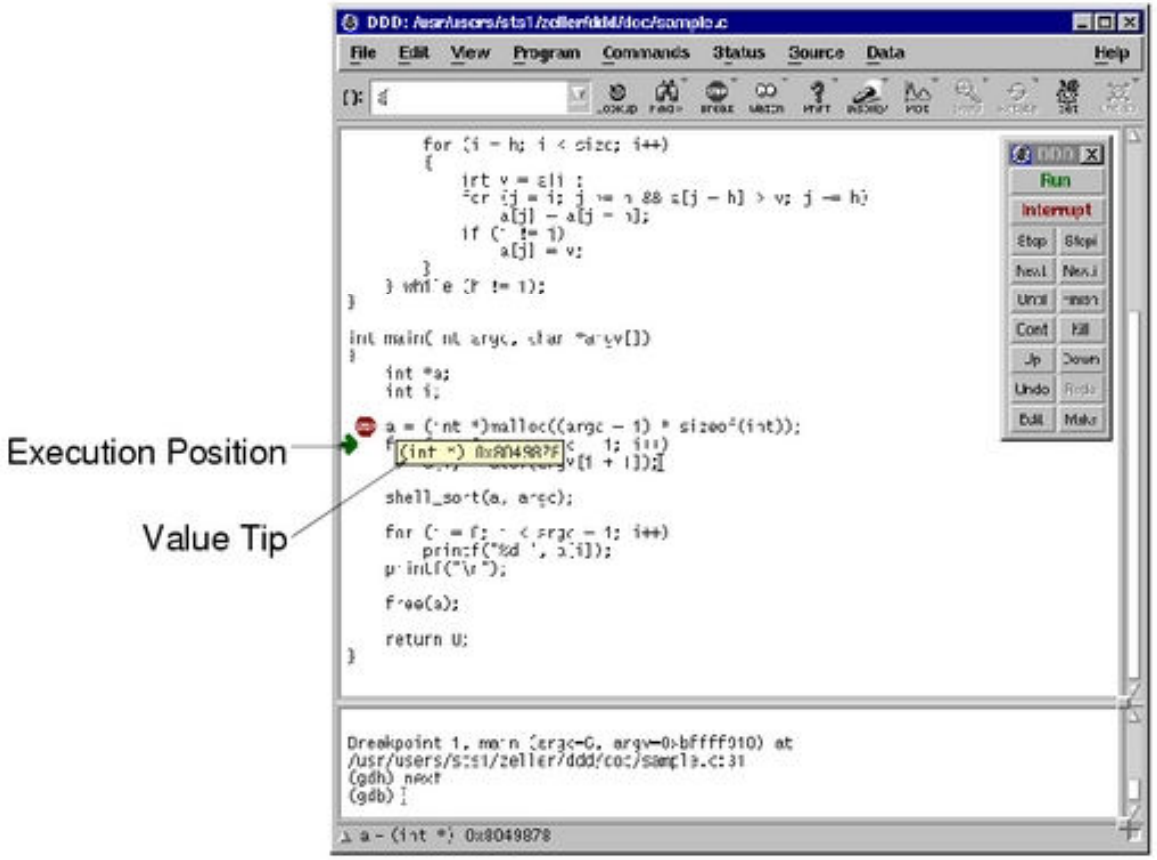


Stepping through the program

To execute current line, click next.

Put cursor over a variable to see its value.

For objects, select it and click Display.



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

Wrapping up the C++ course

Considering we've only been at it 4 weeks, we've seen a lot:

- All the main data types and control structures
- How to work with files
- Classes and objects
- Dynamic memory allocation, etc., etc., etc.

OK, we've glossed over many details and to really use these things you may have to refer back to the literature.

In addition we've seen the main elements of a realistic linux-based programming environment, using tools such as **`gmake`** and **`ddd`**.

Next week we start probability and statistical data analysis. This will give us many opportunities to develop and use C++ analysis tools.