Computing and Statistical Data Analysis
Comp 5: Object Oriented Programming

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

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 **`floats`** (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
```

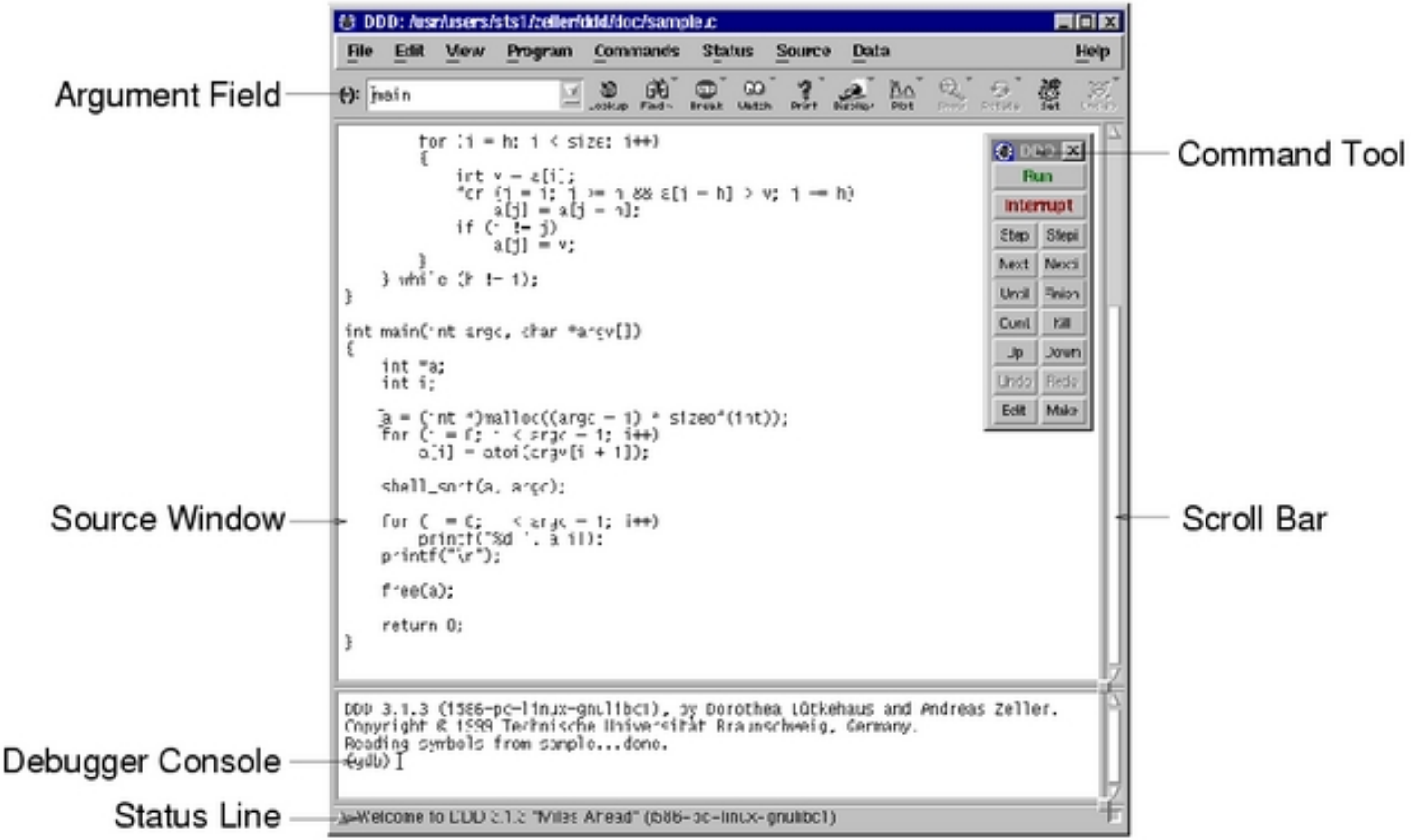
Then type

```
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:



```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

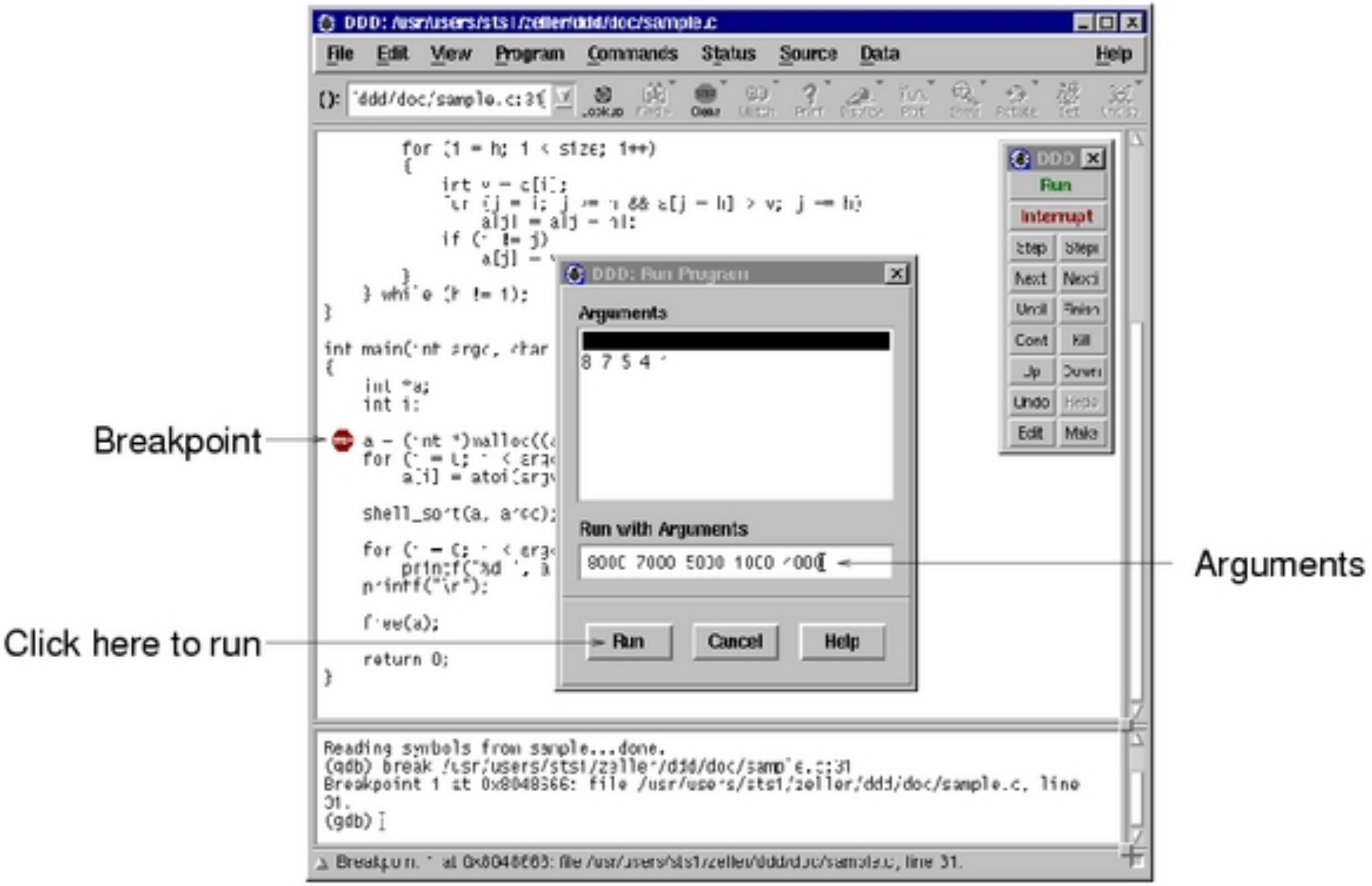
```
DDD 3.1.3 (i586-pc-linux-gnulibc1), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1999 Technische Universität Braunschweig, Germany.
Reading symbols from sample...done.
(gdb) 
```

```
Welcome to DDD 3.1.3 "Man Ahead" (i586-pc-linux-gnulibc1)
```

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.



```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v
        }
    } while (h != 1);
}

int main(int argc, char
{
    int *a;
    int i;

    a = (int *)malloc((
    for (i = 1; i < arg
        a[i] = atoi(arg

    shell_sort(a, argc);

    for (i = 0; i < arg
        printf("%d ", a
    printf("\n");

    free(a);

    return 0;
}
```

Run with Arguments: 8000 7000 5000 1000 4000

```
Reading symbols from sample...done.
(gdb) break /usr/users/sts1/zeller/ddd/doc/sample.c:31
Breakpoint 1 at 0x8048366: file /usr/users/sts1/zeller/ddd/doc/sample.c, line
31.
(gdb)
```

Breakpoint 1 at 0x8048666: file /usr/users/sts1/zeller/ddd/doc/sample.c, line 31.

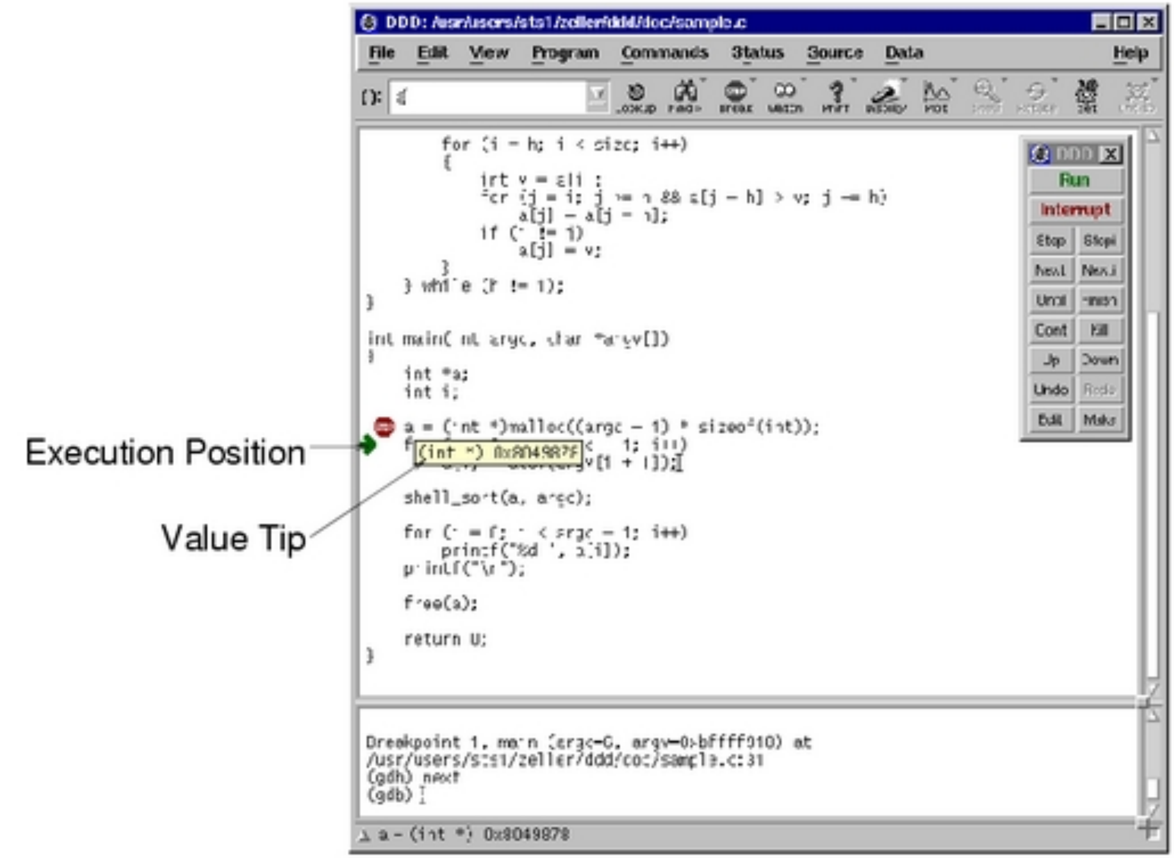
Running the Program

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.



```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

```
(int *) 0x8049878
```

```
Breakpoint 1, main (argc=6, argv=0xbffff910) at
/usr/users/sts1/zeller/ddd/doc/sample.c:31
(gdb) next
(gdb) 
```

```
a = (int *) 0x8049878
```

Viewing Values in DDD

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

Wrapping up the C++ course

Considering we’ve only been at it 5 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 we start probability and statistical data analysis. This will give us many opportunities to develop and use C++ analysis tools.