The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Lab #2 (08/01/2007)

Name:

Student ID:

Lab Instructor: Borja Sotomayor

Do not write in this area

1	2	3	**TOTAL**

Maximum possible points: 30

One of the goals of this lab is to introduce the Eclipse IDE, a software environment you can use to develop your C/C++ programs. We provide some basic instructions on how to write a program with Eclipse, and how to use Eclipse's debugging features. More complete documentation can be found in Eclipse's official website:

http://www.eclipse.org/
http://www.eclipse.org/documentation/
http://help.eclipse.org/

IDE: Integrated Development Environment

An IDE, or Integrated Development Environment, provides an *environment* that *integrates* all the tools necessary to *develop* a piece of software. IDEs typically include a text editor, a compiler, a debugger, and other tools to improve the productivity of the programmer. For example, many IDEs include tools that allow us to easily create graphical interfaces for our own applications ("UI designers")

The user can interact with all these tools (editor, compiler, etc.) using an interface which generally (but not always) will be an easy-to-use graphical interface. In fact, the main advantage of using an IDE is that the programmer can do all his work from a single interface, instead of having to constantly switch between different tools.

The following are some popular IDEs:

- **Free / Open source** (UNIX/Linux)
  - Eclipse (Multi-platform, Java, C/C++, other languages): http://www.eclipse.org/
  - KDevelop (mostly C/C++, includes UI designer): http://www.kdevelop.org/
  - Anjuta (mostly C/C++): http://anjuta.sourceforge.net/
  - Emacs (Multiple languages): http://www.gnu.org/software/emacs/emacs.html
  - NetBeans (mostly Java, also C/C++): http://www.netbeans.org/
- **Proprietary**
  - Microsoft Visual Studio (Windows-only, C#, Visual Basic.NET, Managed C++, Visual Basic, C++, includes UI designer): http://msdn.microsoft.com/vstudio/
  - Borland C++ Builder (Windows-only, C++, includes UI designer): http://www.borland.com/cbuilder/

A more complete list can be found here:
http://en.wikipedia.org/wiki/List_of_integrated_development_environments

The Eclipse IDE

Eclipse is an open-source IDE developed by The Eclipse Foundation. Actually, it is more correct to say that Eclipse as a whole is "an open platform for tool integration", where the IDE is only one of the many Eclipse projects. The IDE primarily supports Java, but can support additional languages thanks to its powerful plug-in architecture. In particular, C and C++ are supported thanks to the C/C++ Development Tools (CDT):

http://www.eclipse.org/cdt/

[If you want to install Eclipse at home, you must install Eclipse *and* CDT]

Starting Eclipse

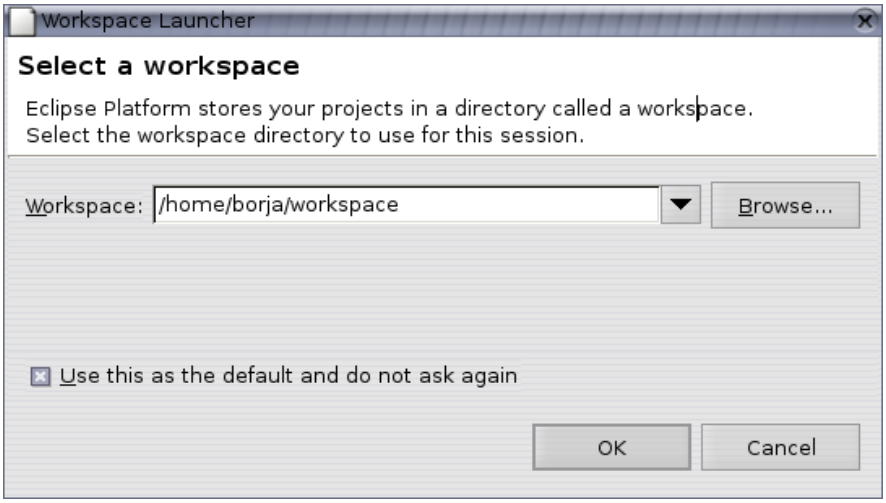
To start Eclipse in the CS Linux machines, run the following command:

```
/opt/eclipse/eclipse-3.2/eclipse
```

The first time you run Eclipse, you will be asked to choose a location for your *workspace*. The workspace is where Eclipse will store all your files. By default, the workspace is:

```
/home/$USER/workspace
```

You can use this default value or choose a different one inside your home directory.

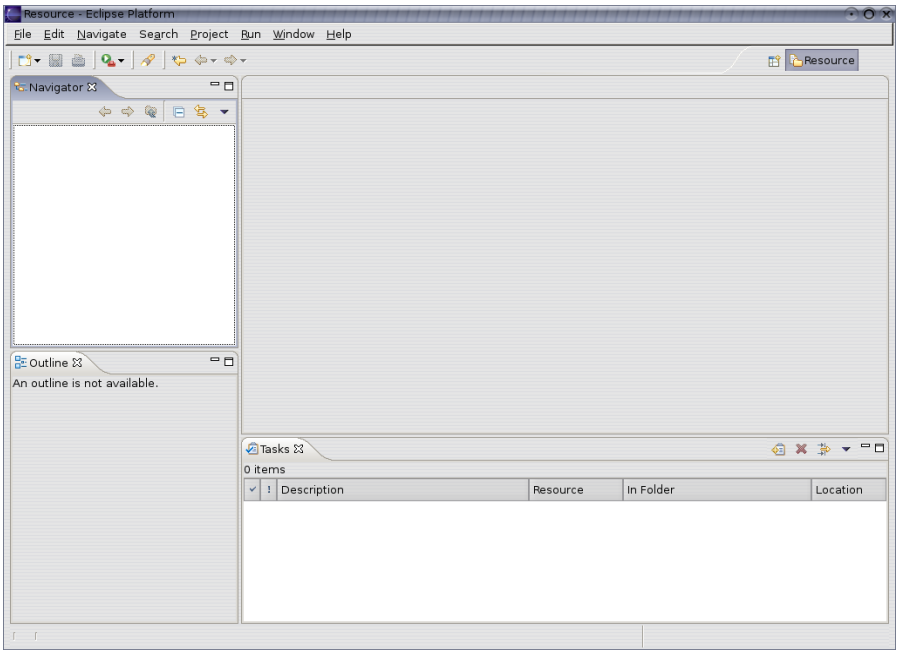


Unless you want to be asked for your workspace location every time you start Eclipse,

make sure you check "Use this as the default and do not ask again".

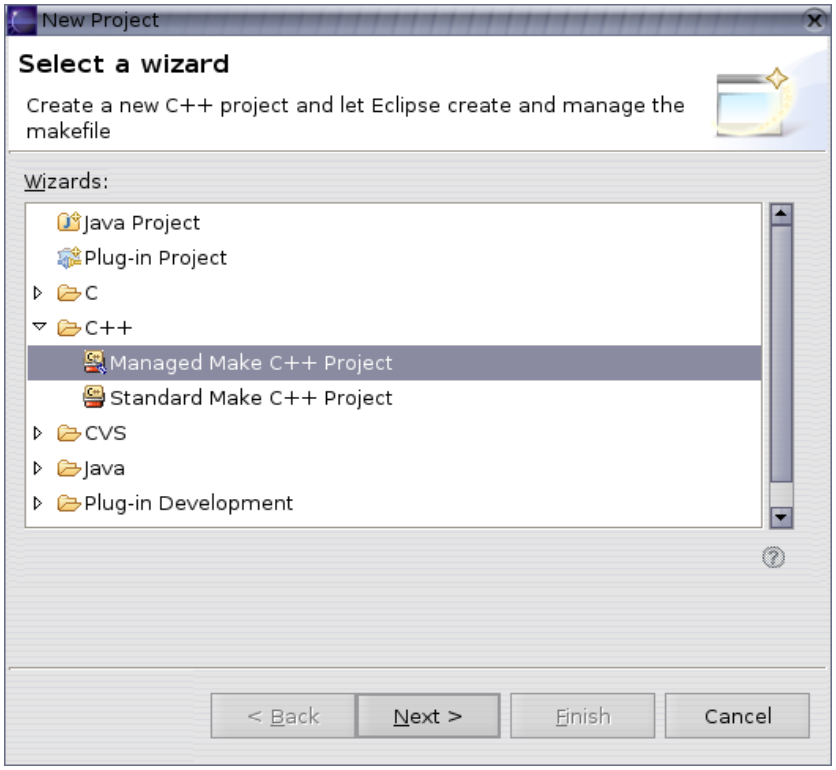
Next, you will be shown a "Welcome to Eclipse" screen where you will be able to access tutorials, help documents, ... Choose "Workbench" to enter the Eclipse IDE proper.

You should now see the following:

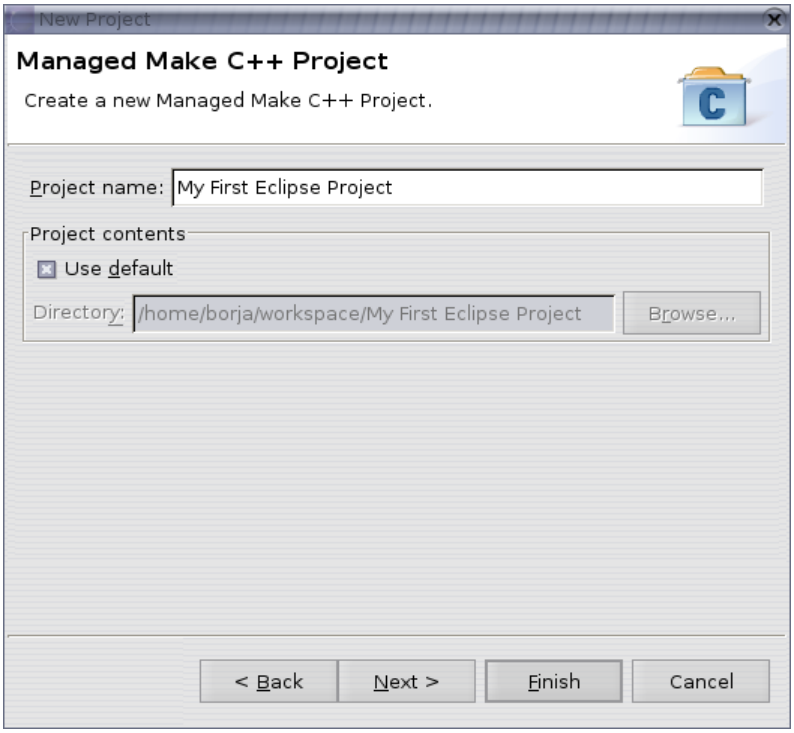


Creating a new project

The IDE is currently 'empty', and we need to create a new *project* to be able to start coding. To do this, select menu File -> New -> Project. You will be shown the following screen, where you need to choose "Managed Make C++ Project":



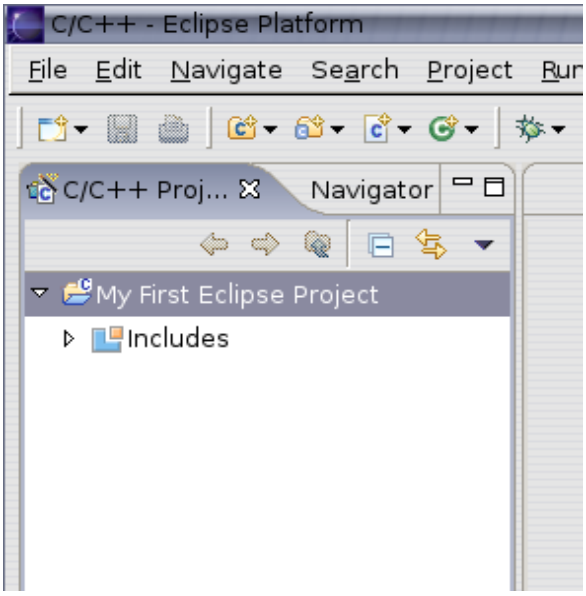
Next, you will be asked to specify a name for your project. You can choose any name, but you will generally have trouble with the debugger if you use any whitespace characters. So, use name "MyFirstEclipseProject" instead of "My First Eclipse Project" as shown in the screenshot:



Next, you will be shown two windows titled "Select a type of project" and "Additional Project Settings". You can safely click the "Next" button in each of them without changing anything.

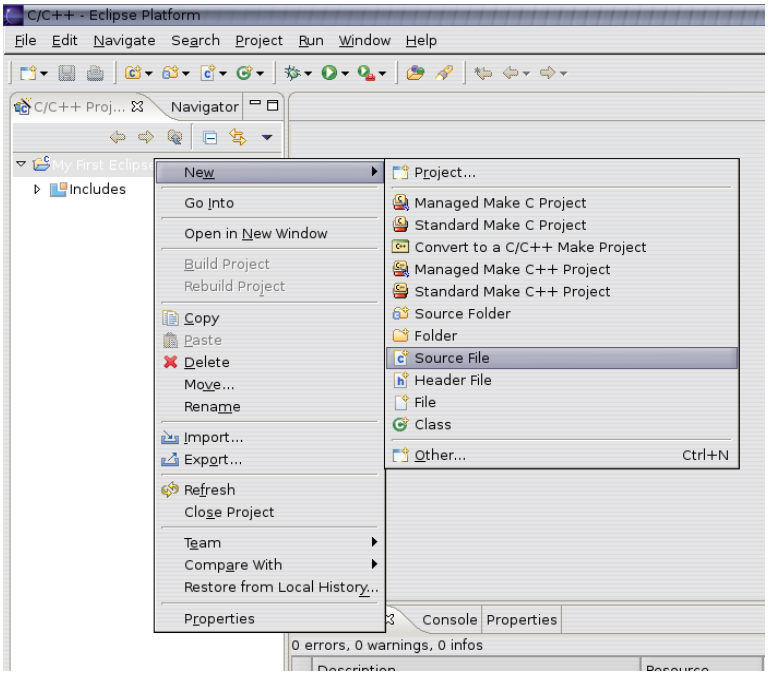
Once the project is created, Eclipse might ask you if you want to switch to the *C/C++ perspective*. If so, just answer "Yes".

Now, you will see that your project has been added to the "C/C++ Projects" tab:

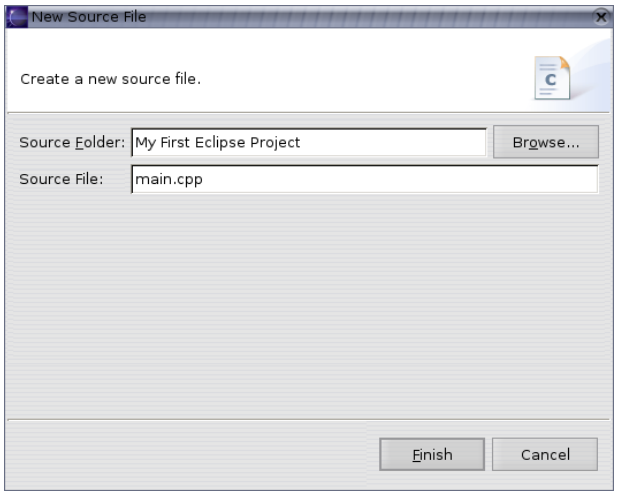


Adding a C++ source file

The project, however, contains no C++ source files. To add one, right-click on the project (in the C/C++ Projects tab) and choose New -> Source file.



You will be asked for the name of the source file. Enter the name "main.cpp"



A new tab will appear in the center of the IDE with your empty main.cpp file. This is where you will write your C++ code.

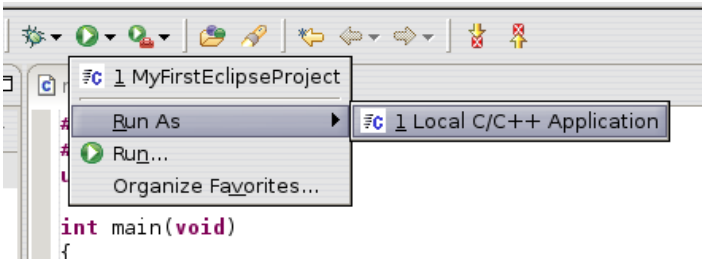
```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
	cout << "Hello, world!" << endl;
}
```

Now, do the following and observe what happens. Make sure you save the main.cpp file after each change (you can do this by pressing Control + S)

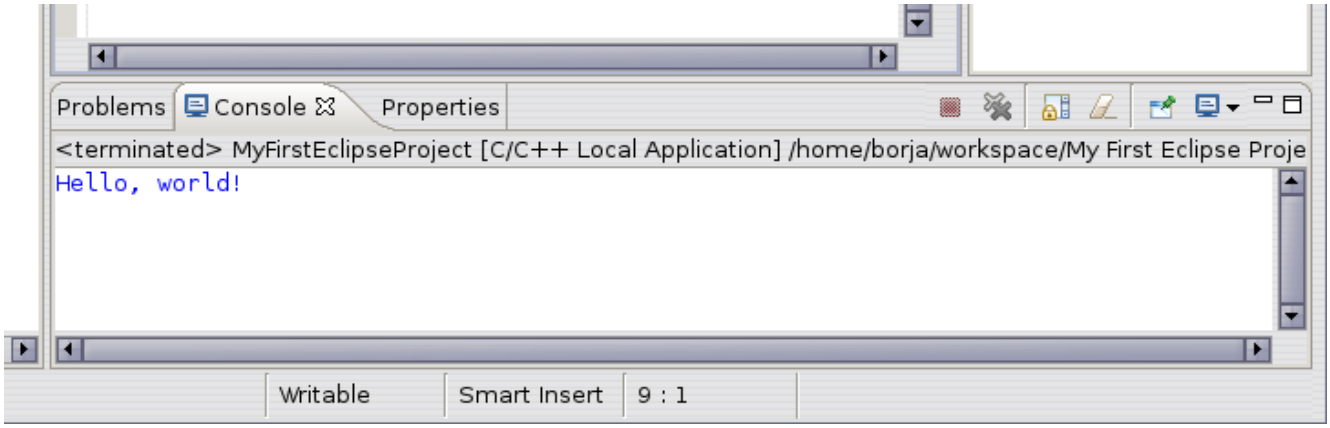
- Declare an integer variable (but don't use it anywhere in the code)
- Change "endl" to "end".
- Declare a variable s1 of type string. Start writing "s1." (the variable name followed by a period) and wait for a second or two.

Running your program

Undo the changes to the program (leave it as originally stated). The first time you want to run the program, you need to click on the "Run" icon in the toolbar (a green circle with a white triangle inside it), but making sure that you click on the small black triangle to the left of the "Run" icon to reveal the "Run" menu:



Choose Run As -> Local C/C++ Application. Next, you might be asked to choose a debugger. If so, just choose GDB Debugger. The program will now run, and you will be able to observe its output in the Console tab on the bottom of the screen:



For all subsequent runs of the program, you only need to click the "Run" button (without going through the "Run" menu).

Debugging with Eclipse

A debugger is a program that allows us to take a peek at what our program is doing internally while it is running. Without a debugger, we would have to rely solely on the output of the program to detect bugs in our code (besides manually inspecting the code, of course). A debugger provides a lot of other information we can use to *diagnose* problems in our program. Usually, we will use the debugger to run the program line by line and see how the values of the variables evolve, to detect (and correct) any bugs in our code.

The debugger most commonly used with GCC is the GDB (GNU DeBugger, http://www.gnu.org/software/gdb/). However, using this debugger directly can be difficult, so we will use Eclipse as a frontend to the GDB debugger. To showcase some of

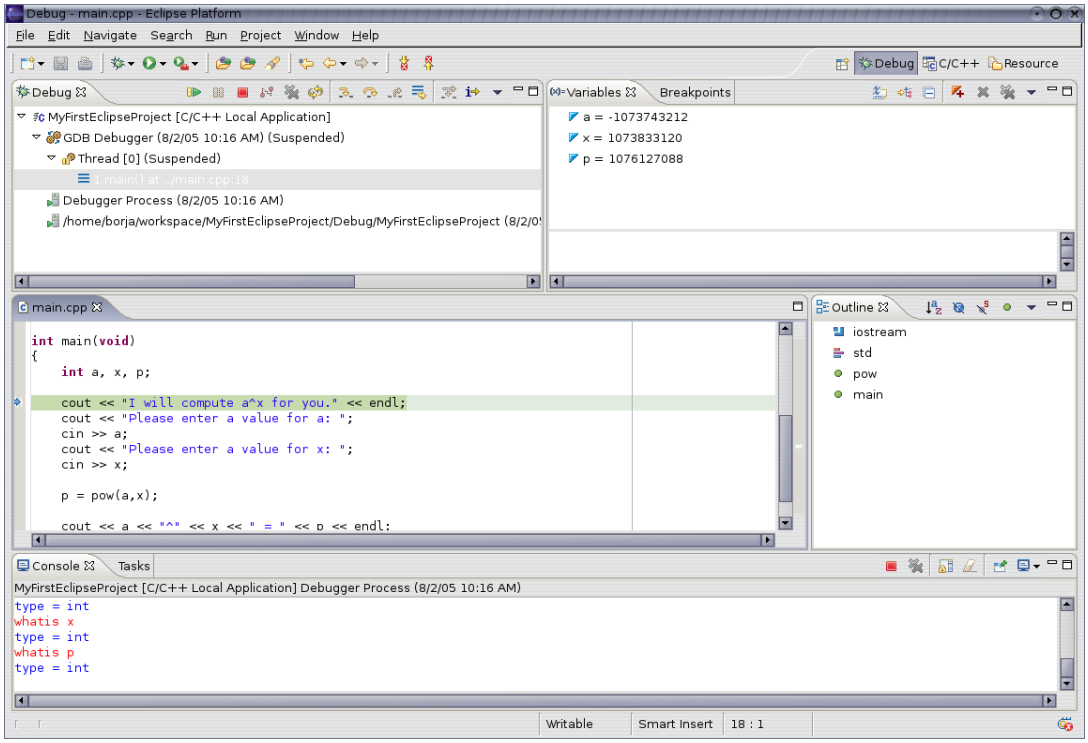
the debugging features in Eclipse, we will use the following program: (pow.cpp in the lab files)

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

int pow(int a, int x)
{
	int p=1;

	for(int i=0; i<x; i++)
		p = p * a;

	return p;
}

int main(void)
{
	int a, x, p;

	cout << "I will compute a^x for you." << endl;
	cout << "Please enter a value for a: ";
	cin >> a;
	cout << "Please enter a value for x: ";
	cin >> x;

	p = pow(a,x);

	cout << a << "^" << x << " = " << p << endl;
}
```

To debug this program, you will need to press the "Debug" button in the toolbar (the button with the drawing of a bug on it). As we did with the "Run" button, the first time you debug a program you will need to first click on the small black triangle to the right of the "Debug" icon, and choose Debug As -> Local C/C++ Application.

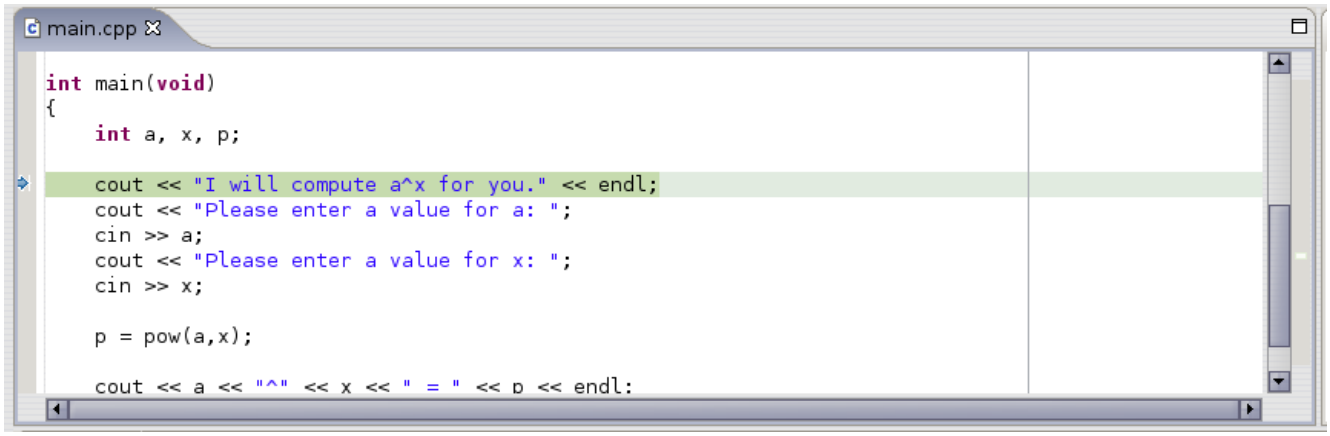
You might be asked if you want to switch to the Debug perspective. If so, answer "Yes".

You will see how the layout of the Eclipse window changes. This is the Debug perspective:

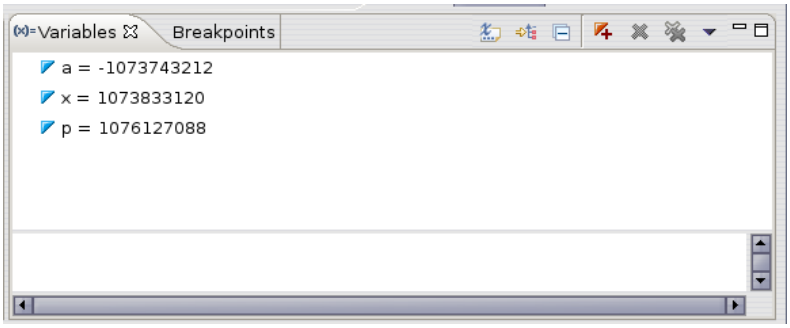


The Debug perspective offers several tabs with lots of interesting information. However, for basic debugging, we will only be interested in two tabs: the one actually containing our program and the "Variables" tab.

Let's start with the main.cpp tab. Notice how the first executable line is highlighted:



This means that the debugger is ready to process that line. However, it will not do so until we instruct it to. Before doing that, look at the Variables tab:

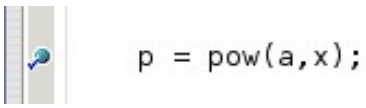


You will see a list of all the variables declared in the current scope. As expected in C/C++, uninitialized variables contain no significant value.

Now, to run through the program press the F6 key ("Step over"). Notice how each press of the F6 key makes the debugger run through a single line. If you look at the console (in the bottom of the screen), you will see the program's output (you will have to focus on the console whenever a program line includes an *input* operation). Keep on running through the program until you reach the end of the program, observing how the values of the variables change.

Breakpoints
Running through the entire program line by line (from beginning to end) can be cumbersome. Sometimes, we might want to debug a very specific piece of code. In that case, we can specify a *breakpoint*. When using breakpoints, the debugger runs the program as usual and only pauses (and switches to "line-by-line mode") when it encounters a breakpoint.

For example, we can place a breakpoint when the pow() function is called. To do this, double-click on the left margin of the line where you want to place the breakpoint. A little blue icon will appear:



When you start debugging the program, it will be paused in the first line as before. To instruct the debugger to start running the program until it encounters a breakpoint, click on the "Resume" button in the "Debug" tab:

The program will ask you for the values of *a* and *x*, but you will not have to run through that part of the program line by line. When you reach the call to pow(), the debugger will pause execution. At this point, if you press F6 as before, you will "Step Over" the function call. If you actually wanted to debug the pow() function, you can press the F5 key ("Step Into"), which instructs the debugger to *step into* the function that is being called.

Step into the function and notice how the contents of the "Variables" tab changes. Run through the function line by line and observe how the values of the variables change.

Exercise 1 <<20 points>>

You will write a program that will allow a user to specify an array of potentially infinite size. The program will repeatedly ask the user the following two questions:

```
Enter a position:
Enter a value:
```

The first time this pair of questions is asked (let's assume the user specifies pos=5, value=7), the program will need to dynamically allocate enough memory for an array with six positions (remember that C/C++ arrays are numbered from 0!), and assign value 7 to position 5.

For all subsequent entries, the program will behave the following way:

- If the current array can accommodate the requested assignment, then no change is necessary to the array. Simply perform the assignment. For example, if the user specifies pos=3 and value=17, we can do the assignment because our array has six positions.
- If the current array is too small to accommodate the requested assignment, then you need to create a *new* array with enough positions, copy all the data from the old array into the new array, and then do the assignment. For example, if the user specifies pos=25 and value=123, we need to create a new array of size 26, copy all the data from the old array, and then perform the assignment.

As stated, the program can run forever. However, for full credit (15 points otherwise), your program must ask the user if he wants to specify another assignment (y/n question, no need to validate this input). When the user is done assigning values, you must print the contents of the array like this:

```
array[0] = 3
array[1] = 37
array[2] = [No value assigned]
array[3] = [No value assigned]
array[4] = 1990
array[5] = [No value assigned]
array[6] = [No value assigned]
array[7] = [No value assigned]
array[8] = 9
```

Note: You will have to keep track of which positions have a value assigned to them. Clue: You are allowed to assume that the user can only introduce positive integers.

Exercise 2 <<10 points>>

The exponentiation algorithm shown earlier (to compute a^x)is not a very efficient one, as it requires *x* multiplications. In the case of *integer* exponentiation, a more efficient algorithm we could use is the *exponentiation by repeated squaring* method, which requires approximately log(*x)* multiplications. This algorithm breaks down the exponentiation into multiple squaring operations. For example:

- $5^8=((5^2)^2)^2$ (3 multiplications instead of 8)
- $5^9=5\cdot((5^2)^2)^2$ (4 multiplications instead of 9)
- $5^{10}=(5^2\cdot((5^2)^2)^2)$ (4 multiplications instead of 10; 5^2 is computed only once)
- etc.

Exponentiation by repeated squaring can be implemented easily with a recursive algorithm:

- $a^1=a$
- $a^x=a\cdot(a\cdot a)^{(x-1)/2}$ (if *x*>1 is odd)
- $a^x=(a\cdot a)^{x/2}$ (if *x* is even)

Write a program that asks for numbers *a, x* and computes a^x . To verify that your implementation works correctly, you can see if the output of your program matches the output of the inefficient exponentiation program used earlier.

Exercise 3 <<10 points>>

The following program (primes.cpp in the lab files) generates the list of all prime numbers up to a certain number, without having to check if each individual number is prime or not (these type of algorithms are called *sieve* algorithms).

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

// Maximum number to check
const int MAXPRIME  = 99;
// Denotes that a number is not prime
const int NOT_PRIME = -1;

int main(void)
{
    int primes[MAXPRIME];    // List of prime numbers
```

```
    primes[0] = NOT_PRIME; // 0 is not prime
    primes[1] = NOT_PRIME; // 1 is not prime

    for (int i=2; i<MAXPRIME; i++)
        primes[i]==i;

    for (int i=2; i<MAXPRIME; i++)
        if (primes[i] != NOT_PRIME)
            for (int j=i; j<MAXPRIME; j++)
                if (primes[j]!=NOT_PRIME && j%i==0)
                    primes[j] = NOT_PRIME;

    int numPrime=1;
    for (int i=0; i<MAXPRIME; i++)
        if (primes[i]!=NOT_PRIME)
            cout << "Prime #" << setw(4) << numPrime++ << " = " << i << endl;
}
```

However, this program has two bugs! One bug should be apparent by simply inspecting the code. However, the second bug might require some debugging to trace.

The algorithm the program is meant to implement works like this:

- Choose a number MAXPRIME (the largest number we will check for primality)
- Create a list of all numbers from 0 to MAXPRIME.
- Cross out 0 and 1, as we know they are not prime.
- Cross out all multiples of 2 (but not 2 itself).
- Cross out all multiples of 3 (but not 3 itself).
- Cross out all multiples of 5 (but not 5 itself).
- ...
- In general, we start at number 2, and keep going through the list of numbers. Each time we encounter an uncrossed number we go through the rest of the list and cross out any multiples of that number.
- Once we've processed all the numbers, any number which is left uncrossed will be prime.

For example, suppose we want to find out all the prime numbers up to 15. In the following list of steps, the number in boldface denotes the "current number":

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	**6**	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	**9**	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	**14**	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**

Cross out 0 and 1
Start at 2
Cross out multiples of 2
Next number: 3
Cross out multiples of 3
Next number: 4 (crossed out, move on)
Next number: 5
Cross out multiples of 5
Next number: 6 (crossed out, move on)
Next number: 7
Cross out multiples of 7
Next number: 8 (crossed out, move on)
Next number: 9 (crossed out, move on)
Next number: 10 (crossed out, move on)
Next number: 11
Cross out multiples of 11
Next number: 12 (crossed out, move on)
Next number: 13
Cross out multiples of 13
Next number: 14 (crossed out, move on)
Next number: 15 (crossed out, move on)