Getting Started With NetBeans

1 Introduction

An *Integrated Development Environment* is a graphical user interface that brings together (integrates) software development tools to make the developer's job simpler and more efficient. It provides a front end to a number of different tools (editor, compiler, debugger, and more) that enables them to work better together. While it is possible to do all this individually from the command line without an IDE, the convenience is well worth the effort to learn.

NetBeans is a free, open-source, cross-platform IDE that can be downloaded from the internet here[1]. It interfaces with the GNU Compiler Collection (GCC) and GNU Debugger (GDB), which are among the world's most popular development tools. The version of NetBeans installed on ECF is 8.2. The screen captures in this document are for version 7.1.2. However, there is little change between the two versions for your usage of NetBeans.

There is extensive online documentation to help you get started with NetBeans. You may want to look at NetBeans.org, particularly the C/C++ Getting Started Guide. There is also a YouTube video channel for Netbeans.

Code referred to in this tutorial document is available in the examples link on the lab documentation page. The examples subdirectory contains a number of folders, each of which is an example program with source files and a Makefile to build the example program.

2 Your first project

2.1 Importing an existing project

1. In the menu, select File → New Project.

2. Select C/C++ under Categories, and "C/C++ Application from Existing Sources" under Projects. The other options are more advanced: you can create libraries to be used and shared by other programs, or you can create graphical user interfaces with the Qt toolkit. For now, we are just interested in a basic C/C++ command-line (text only) application. Click next.

3. Browse to the folder containing the existing source files. Leave "build host"[2], "tool collection", and "configuration mode" alone for now. The tool collection should be GCC typically (unless you are using a different compiler). Whatever tool collection you choose must be installed for the project to compile. Click next.

4. After some time to load and parse the files, your project will appear in the editor and navigator windows for you to start browsing or building.

5. Click the green play button (see Fig 14 if this isn't clear) to run the program. It will compile if necessary, and run the program if compilation is successful.

[1] http://www.netbeans.org

[2] This option allows you to build on a different computer than you use to edit the files, sometimes done on large team projects

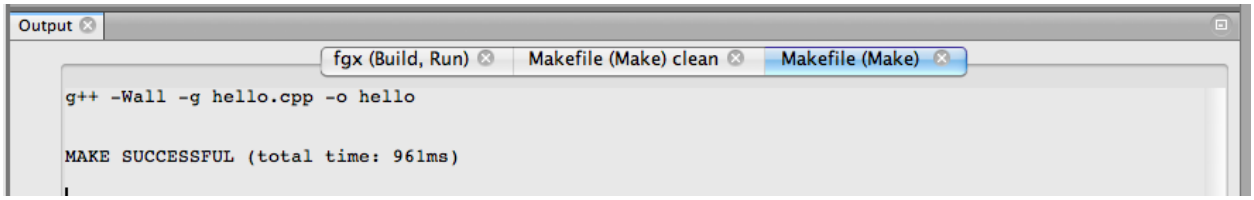

Figure 1: Succesfully importing and building `hello.cpp`

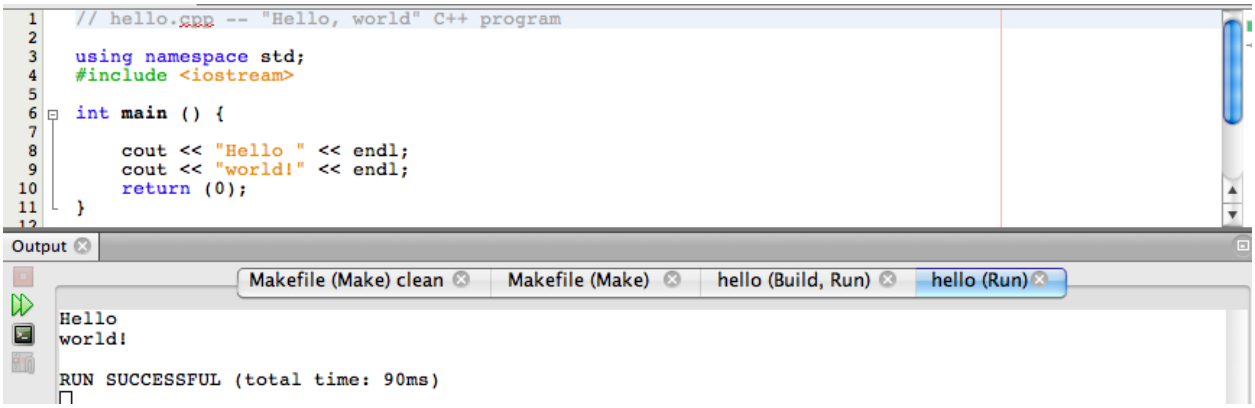

Figure 2: Results of running `hello`

Note that creating a project from existing sources requires a *Makefile* to provide the "recipe" for building the project. Without the Makefile, NetBeans does not know what to do with the sources to produce a working application. We will cover what the Make utility does in greater detail later in the course - initially, you will be provided with the necessary Makefiles to get your labs working.

2.1.1 Example

Try importing the `hello` program from the lab1/hello directory. The import process will automatically compile, leaving output as shown below in Fig 1. By clicking on the green run button, the program will be run giving output as shown in Fig 2.

2.2 Creating a new project

A new project can be created in a very similar way, except that you specify a new folder to create for the project instead of the one containing files as depicted in Fig 3. The "project location" field specifies where the project lives. Usually the project files will be placed in a folder under the one designated (ie. if you set project folder `/home/user/foo` and project name `bar`, the files will go in `/home/user/foo/bar`). Note also as you change one field, the other will automatically update. Instead of using a provided Makefile, the IDE will generate one for you, plus optionally a starter main source file called (by default) `main.cpp`. Generally, the automatically-generated Makefile will be far more complicated than one that is custom-created by hand. On the other hand, you never need to edit the file itself - all options can be set from within menus in the IDE (and any changes you make to the file itself would be overwritten the next time you change a menu option).

When you create a new project "from scratch" in this way, you should add the files you need to it using the NetBeans IDE, and NetBeans will automatically update the Makefile used to build

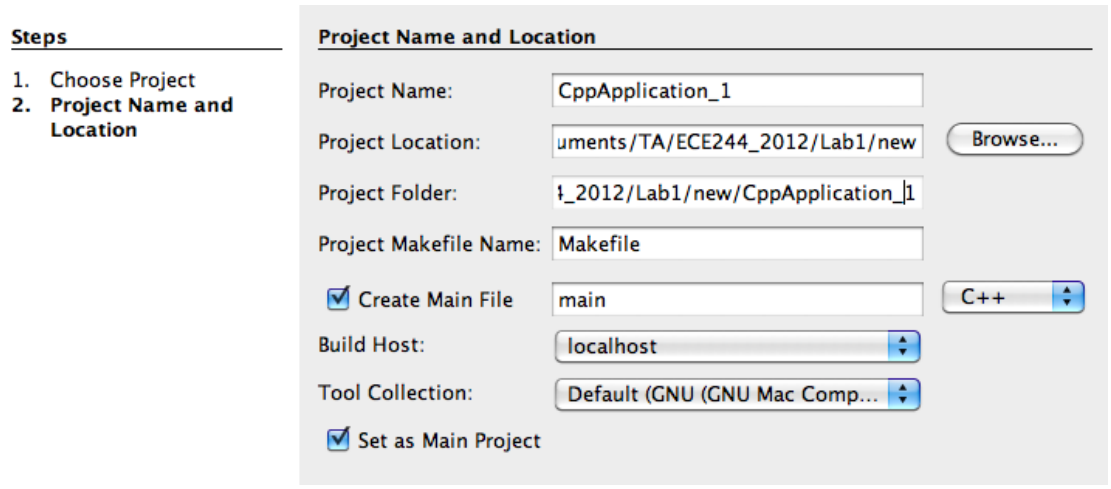

Figure 3: Creating a new project

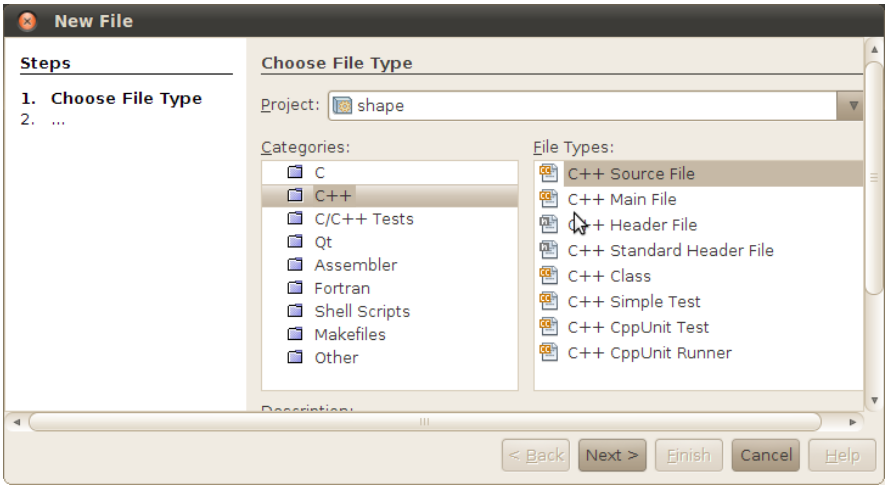

Figure 4: Adding a new file to a project.

the project. Use the "File → New File" command to add new C++ source (.cpp) and header (.h) files to your project, as shown in Fig 4.

Finally, when you create a new NetBeans project it is best to set the options that will be used to compile the project such that you receive "warnings" about any code in your program that the compiler detects is unusual or dangerous. Use the "File → Project Properties →" menu, and set the C++ Compiler Warning Level to "More Warnings" as shown in Fig 5.

2.3 Working with multiple projects

NetBeans has the ability to work with several projects at once. For large, complex projects this may mean several related applications, or an application plus the libraries on which it depends. One of the projects is always designated as the Main project (displayed in bold on the project browser - see Fig 6). The Run, Build, and Debug commands apply to the main project only. You can make a project main by right-clicking on it and selecting a menu option in the Projects pane of the main window.

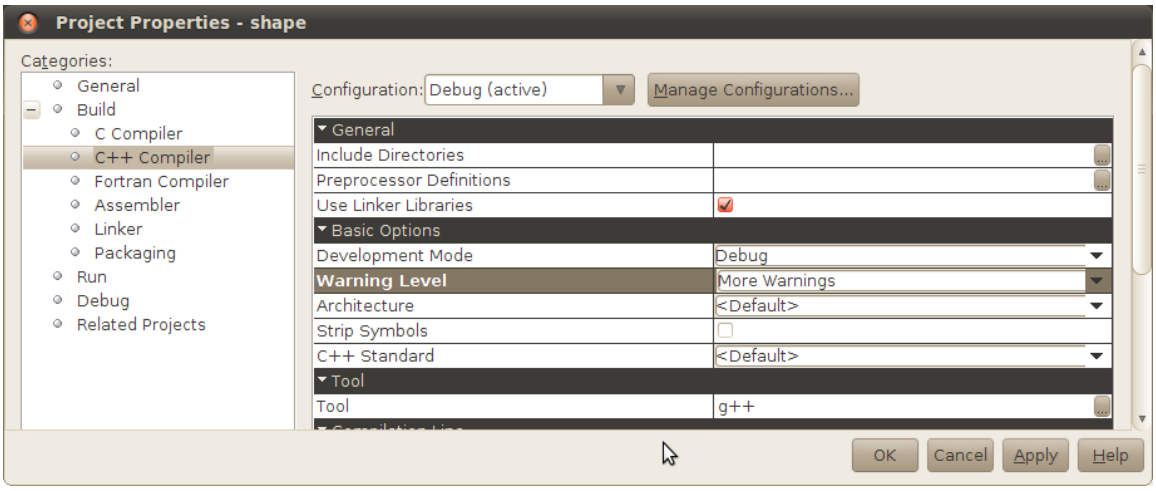

Figure 5: Select More Warnings in your C++ Compiler options so you are warned about dangerous or unusual code by the compiler.

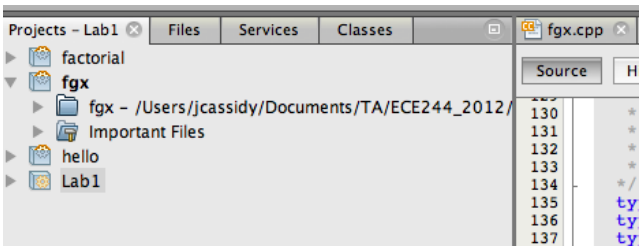

Figure 6: Handling multiple projects - with `fgx` as main project

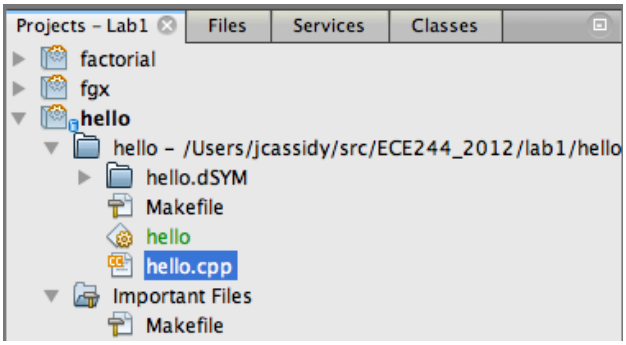

Figure 7: Open a file by double-clicking. If you're curious how the Makefile works, you can open it up and have a look too. If you type `man make` at the command prompt or Google "GNU make", it will give you more information on the Make utility.

3 Using the NetBeans editor

To start the editor, you must select a file to edit. You can do this from either the "Files" pane or the "Projects" pane at the top-left of the screen. The easiest is through the "Projects" pane - just expand the listing for your current project until you find the `.cpp` file (example in Fig 7). Double-click it to open. Alternatively, you can create a new file using File → New or the appropriate button on the toolbar (hover your mouse over to figure out which - it has a plus sign).

3.1 Completion Hints

Using control-P (command-P on Mac), the editor will show a list of possible arguments for the function call under the cursor (Fig 8. This is particularly handy if you forget what the arguments to a function are, or what order they come in. Note that unlike the example, the function prototype need not be in the same file. For standard library functions, this can be excellent because you don't even need to know where the definition is - the IDE will find it for you and tell you what you need to know[3].

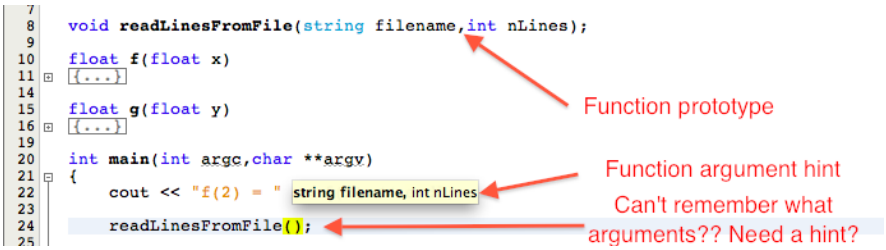

Figure 8: Forget what arguments a function takes? Put your cursor in the parentheses and press ctrl-P (or command-P on Mac)

Completion suggestions for a class member call can also be requested by typing control-\(command-\ on Mac). For a given context, it will examine the text under the cursor and list possible valid completions. One use is to view the list of class methods.

[3]Assuming the correct header file is `#include`-d

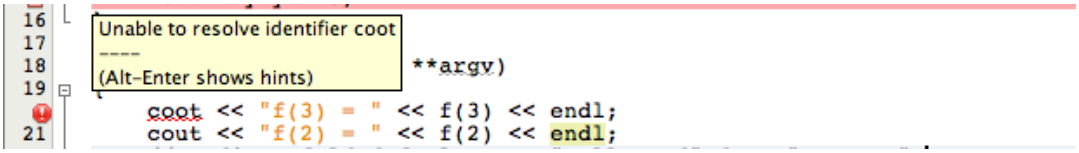

Figure 9: The editor has highlighted an undefined identifier here (press alt-enter to get details) - looks like a typo!

3.2 Syntax and identifier checking

The NetBeans editor will display a red stop sign with an exclamation point if it identifies an error in your code (Fig 9). These flags are an advance warning that the compiler will not be able to compile the source. The editor can't catch every possible error, but it does a fairly good job of the more common ones, which saves time.

3.3 Syntax highlighting

Another thing the editor does to help you in reading files is syntax highlighting. Language *keywords* (words with a special meaning such as `#include`, `int`, `class`, etc) are highlighted in a special colour to make them stand out. Parentheses, square brackets, and curly braces will also highlight in pairs when you place the cursor over them, to help you identify matching pairs. That feature gets handy when you have multiple levels of nesting, to make sure you're matched correctly.

3.4 Auto-indent and auto-complete

Source code readability is helped tremendously by proper indentation to denote nesting of classes, functions, loops, and conditional statements. The IDE helps you achieve this by automatically indenting the constructs listed above. It will also automatically close parentheses, and complete common constructs like `#include`. You will notice some of this happening as you type and use the editor. If you happen to find it annoying, you can turn off auto-completion by using the editor tab under the NetBeans → Preferences menu item.

3.5 Code folding

Sometimes when working on a large file[4], there is only a small portion of it that is of interest at a time. It can make your life easier to hide parts of the file so you can focus on what you're working on. In the editor, small boxed minus signs will appear in the left margin next to a function body, comment, or class definition. If you click, the whole block will *fold* into a single line (Fig 11; note the line numbers jump at the fold). Click again, and it comes back (Fig 11).

3.6 Code browser

One of the advantages of using an IDE over a normal text editor is that the IDE has functions to *parse* and understand the language you are writing in. As a result, it "knows" about function prototypes, function bodies, variable declarations, comments, identifiers, etc. It also understands *scope*, which is the set of variables/functions/classes visible at a given point in the program. If an

[4]Note if the file gets really big, convenience and good style would suggest that you split it into multiple files along some reasonable criteria. For instance, maybe have the methods for just one class in a file, or limit each file to just a few related functions.

```
 9  void readLinesFromFile(string filename,int nLines);
10
11  float f(float x)
12  {
13      return g(x-1) + 2*x;
14  }
15
16  float g(float y)
17  {
18      // suppose this was a really really big function
19      // ...
20      // ...
21      // ...
22      // ...
23      return y*y + 3;
24  }
25
26  int main(int argc,char **argv)
27  {
28      cout << "f(2) = " << f(2) << endl;
29
30      return 0;
31  }
32
```

Figure 10: Folding - before.

```
10
11  float f(float x)
12  {...}
15
16  float g(float y)
17  {...}
25
26  int main(int argc,char **argv)
27  {
28      cout << "f(2) = " << f(2) << endl;
29
30      return 0;
31  }
```

Figure 11: Folding - after. Note line numbers jump at the fold and the code appears much more compact.

identifier is seen within a given scope, the IDE can help you by navigating to the definition if you control-click on it (command-click on Mac). For instance, for any given variable or function you can have the IDE jump to its definition.

Another way to access this feature is through the navigator (Fig 12), typically found at bottom left or in Window → Navigating → Navigator as shown below.

This shows header files (`cmath`, `iostream`, `fstream`), function bodies (`f`, `g`, `main`), namespace references (`std`), and function prototypes (`readLinesFromFile`). Double-clicking on any of them will take you to the corresponding line in the source file.

3.7 Backwards and forwards browsing

The browser also maintains a history of where you've been while clicking through links and following definitions. This allows you to use a web-browser-like forward/backward command using control-left and control-right (Windows), or menu item Navigate → Back. For instance, you can control-click a function name to go to its definition, then use Back to return to where you were previously.

4 Examples

Try typing the following code into your editor (or load it from `stringex` in the examples folder). Note the differing colours indicating the language constructs that it recognizes. You should also see some red marks in the left margin. First, it will indicate some "unable to resolve identifier"

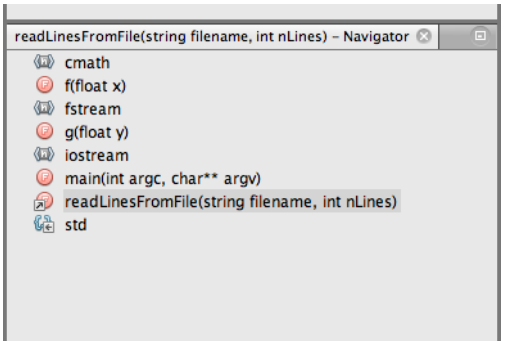

Figure 12: The navigator lets you jump easily to various definitions within your file or the project.

Figure 13: Hitting control-\without the function parenthesis brings up the function documentation and argument info

errors. These can be fixed by using the `std` namespace (see section 5.3.4). Now place the cursor at the end of the "`str.`" line, and use the completion suggestion button to see what options there are. It should list a number of different methods for the `string` class.

```
#include <string>

int main(int argc,char **argv)
{
        if (argc != 1)
        {
                cerr << "Needs one argument (a string)" << endl;
                return -1;
        }
        string str(argv[1]);

        count << "Length of string is " << str.                // this line needs completion
        return 0;
}
```

Another example, this time for a function call instead of a class member is shown in Fig 13.

5 Compiling

Compilation is the process of converting a human-readable source code text into a machine-readable binary *object file*. Most programs are too large to conveniently fit into a single source file - some may have thousands, millions, or even tens of millions of lines. Even very simple, single-file programs

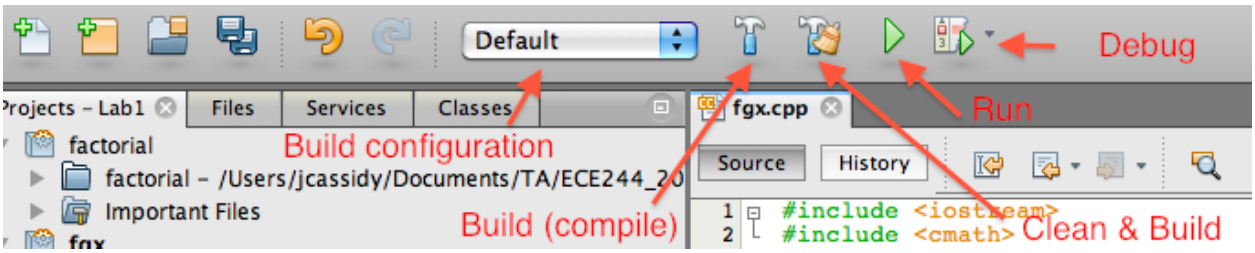

Figure 14: The build toolbar

typically rely on the *standard library* for functions like input/output: reading from and writing to the terminal. So, multiple files can be compiled independently, each leaving a list of *unresolved* entries which are functions or variables that it does not have a definition for within the file. The *linker* then "ties together" all the separate pieces in the object files with *libraries*. If successful, it produces a binary *executable file* that the computer can load and run. This is our goal - to create a working executable from source code.

5.1 Using NetBeans to compile

A compiler is an extremely complex piece of software to create, debug, and maintain. For this reason, NetBeans does not provide its own compiler, rather it "knows" how to call out to other existing compiler systems like LLVM, MinGW, and GCC. The user may specify which toolchain to use as we did during install (GCC on ECF machines). Such reuse of other components to avoid "reinventing the wheel" is an important principle in engineering. What NetBeans does is wrap the extremely powerful back-end compiler tools in a friendly, graphical, easy-to-use interface. The figure below show how to build a project once it has been imported.

Again reusing popular tried-and-true tools, NetBeans manages its build process using the *Make* utility to call the back-end compiler (in this case GCC). Project options and the Makefile take care of sending the correct flags to the compiler. We will cover Makefiles in greater depth later, but the brief explanation is that Make is a tool to save time by rebuilding only the parts of a program that have changed. If a given source file has not changed, then the compiler does not need to be re-run for it.

The build toolbar shown in fig 14 has the buttons needed to compile the project. Pressing the "Build" button will cause the compiler to run for all source files that are out of date, thus (if successful) producing the target program so it can be run. Status output is provided in the "Output" window which appears by default at the bottom of the screen. The "Clean & Build" option is a more thorough process which first deletes all the output files and re-builds from scratch. On a large project, this can be a very time-consuming process however it makes little difference on the scale of this course. The green "Run" arrow runs the program, building it if necessary due to a source file change. Command-line options provided to your program, if desired, can be set in the Run → Set Project Configuration → Customize window.

Note that when you build, the output window shows the commands and options issued to `g++` to build the project. You can verify here what exactly the IDE is doing. Also, if you were to type the same command at the terminal it would also build the project exactly the same way. One of the most useful things to check for is the `-g` option, which will be important when debugging.

Lastly, and just for completeness, the drop box labelled "Build configuration" allows the user to create and name several different configurations: sets of different compiler and run-time options. Imagine as is common practice that you have two versions with different compile options: one for

maximal speed (a *release build*), and one to allow debugging (a *debug build*). This box lets you flip quickly and easily between two different sets of compile, link, and run-time options to carry out those two functions without having to re-type them into menus each time. Generally for this course, we will provide the Makefile so you can just leave this on "Default".

5.2 Compile-time errors

Unfortunately, it is rare for a program to work correctly on the first try; there will be errors which can be classified by when they occur: *compile-time errors* prevent the compiler from producing an executable, whereas *run-time errors* or *"bugs"* permit compilation but produce undesired results or crashes when the program is run. We will deal with detecting and correcting run-time errors separately in a discussion of *debugging*.

A compile-time error occurs when the compiler can not correctly translate the source language (here C++) to machine code. Informally, a compile-time error occurs when the compiler cannot "understand" the code you've given it due to invalid use of the language or missing information. The syntax of a language is the set of rules saying what are valid elements (identifiers, punctuation, etc), and the order in which they can occur. As you will learn, working software code, unlike written or spoken language, must conform exactly to very strict rules of *syntax*. Any deviation from the rules will cause an error because computers are "dumb": they cannot understand your intent, context or meaning, instead relying on inflexible rules to translate exactly what you said rather than what you meant. This can be very frustrating: small typographical errors (spelling, case, punctuation) that don't hinder a human's understanding (and are therefore often overlooked) completely stump the compiler, causing one or more fatal errors. The NetBeans editor helps with this by finding and highlighting some errors as you go along, but there are often errors that pop up only when you try to compile.

It's also possible that what you've done is syntactically correct but refers to identifiers (variables, functions, classes) that are undefined, or uses the wrong arguments for a function, or does an invalid type conversion. In this case, the compiler can understand the structure of what you are trying to do, but it lacks some other information. For instance, you may refer to an undefined variable: the compiler can see that you're referring to a variable, but doesn't know where it's kept or what type it is. Likewise, it will recognize a function call if you type `f(a,b,c)`, but if it doesn't have information on what types of arguments `f` takes it will fail.

Don't be discouraged if you receive a huge list of errors. It's often the case that after a single syntax error, the compiler gets confused and starts thinking everything is wrong. Just start at the top and try to fix the errors one or two at a time, reading the error message carefully to try to understand the cause. Last but not least, the IDE helps you yet again by making it possible to click on compiler errors to go to the relevant line. Any error with a blue hyperlink permits you to follow the link into the source editor. Sometimes, the source browsing and completion-hint functions can help you fix this - or the IDE may offer you a hint if you press alt-enter when over an error.

5.3 Common compile-time errors

To help you get started, here is a list of common compile-time errors that you are likely to encounter:

5.3.1 Undefined symbol

Variables must be declared before being used. If you get an undefined symbol error, it could mean that you have either forgotton to declare the variable, the variable name is misspelled, or it is

not visible in the current scope. If you're trying to use a symbol from a standard library, see the sections below on case, header files, and namespaces.

5.3.2 Case

C++ is a case-sensitive language so `foo`, `Foo`, and `FOO` are all separate identifiers. Note that it's poor practice to use variables that differ only by case. You should choose a capitalization convention for identifiers (variable, function, class) and stick to it.

5.3.3 Missing header files

To use standard library functions, classes, and objects you must include a header file. The most common ones you will need are: `iostream` for IO (input/output); possibly `iomanip` for formatted output; `string` for anything related to the string class; consult documentation for any library functions you want to use to find out what header(s) are needed.

5.3.4 Namespace problems

If you have included the correct header files and it's still not finding a symbol (for instance `cout`), make sure you have the line `using namespace std;` near the top of your file after the `#include` statements. This will make all symbols in the `std` (standard library) *namespace* visible to your program. An alternative solution is to prefix `std::` to all library symbols that you use.

Namespaces are used primarily when creating libraries (such as the standard library). They support *encapsulation*, making definitions invisible unless the programmer wishes them to be visible. It avoids the risk of multiple libraries defining the same symbol in different ways which would be illegal. Normally when you refer to a symbol (variable, class, function), the compiler searches all currently-included namespaces to find its definition. If there is no match, it will generate an undefined symbol error. It is also possible to specify a *fully-qualified* symbol name such as `std::cout` to specify exactly the namespace (in this case `std`) to search.

5.3.5 Type mismatch

C++ is a *statically-typed* language, meaning each variable has a definite type which cannot be changed "on the fly". Once defined as a certain type, the variable has that type for the rest of its life, until "dies" by going out of scope. For convenience, certain conversions are permitted when unambiguous automatic conversions exist (for instance, you may assign an `int` value to a variable of type `double`). However, most are not. For instance, we cannot assign the character array `''2''` to an `int` even though the intended meaning may seem obvious. Make sure that you are assigning variables to like types, that you *cast* the type correctly (more on this later in the course), or that an appropriate conversion is called[5]. If you do not, you will see an error message like in Fig 15.

6 Running

Following successful compilation, you will want to run your program and test its functionality. This too can be done from within NetBeans. The IDE provides you a few handy ways of providing input to your program just like you would from the command line: terminal input (display text to screen & read from keyboard), command-line arguments, and environment variables.

[5] For the `char[]` example, `atoi` would be one way to do it.

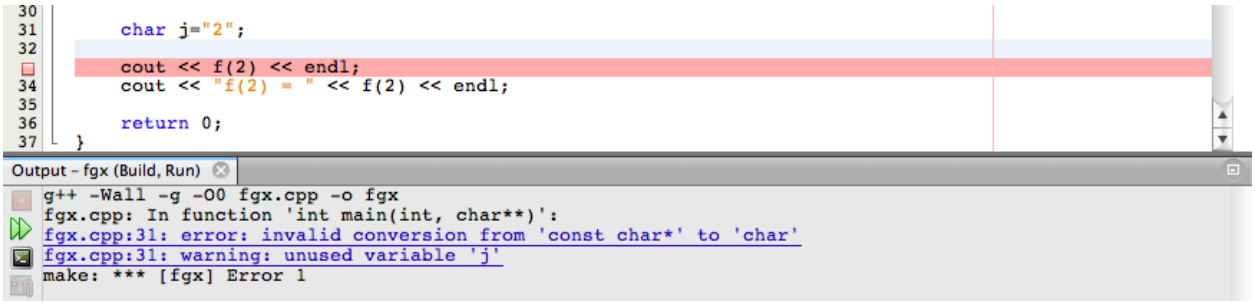

Figure 15: A type mismatch error. Note the blue error message allowing you to hyperlink to the source of the error. You will learn the (significant) difference between double and single quotes in C++.

Options controlling how the program is run are found under Run → Set Project Configuration → Customize menu item, which brings up the dialog shown in Fig 16 (click on Run at left). For each project configuration, you can set command-line arguments and environment variables (more on these in the sections below). This way, you can have multiple test cases (configurations) and switch between them easily without manually having to enter command-line arguments every time you run, or change menu options when switching between cases.

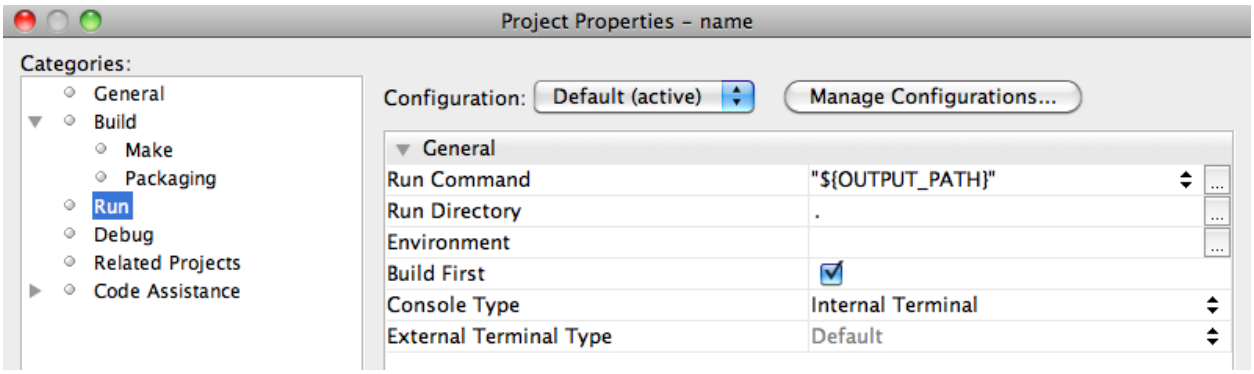

Figure 16: Run options dialog

6.1 Terminal input

The simplest case for reading and writing from the terminal is demonstrated in the example called `name`, which asks the user for their name and prints out a welcome message using the name given.

When you run this program, it should display a prompt message asking the user for their name. At this point, the program pauses (*blocks*) waiting for the user to respond. You can click in the Output pane and type on the keyboard to interact with the program as shown in Fig 18. When you type your name followed by Enter, the program will resume.

6.2 Command-line arguments

Another way of interacting with a program is via its command-line arguments. Arguments are additional information provided after the program name on the command line. The shell commands

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(int argc,char **argv)
6  {
7      string name;
8      cout << "Please type your name: ";
9      cin >> name;
10
11     cout << "Hello, " << name << ", and welcome to ECE244" << endl;
12     return 0;
13 }
```

Figure 17: `name.cpp` program to prompt user for input and use the input to display a message

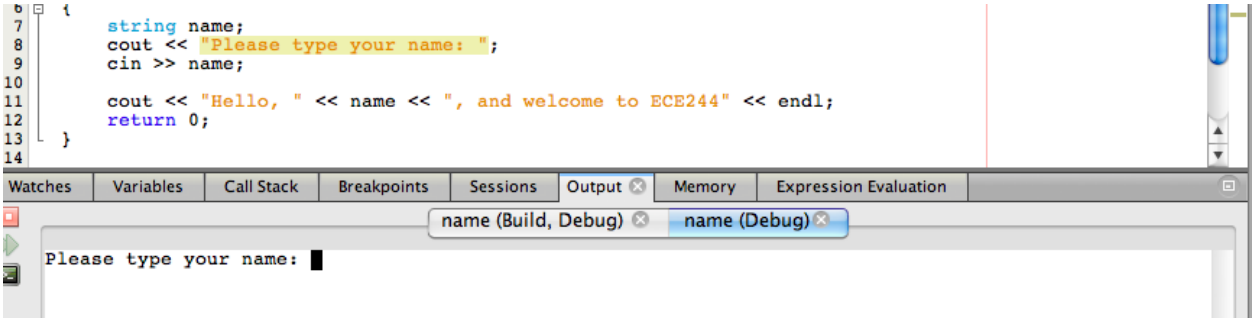

Figure 18: Interacting with the program via terminal

you have already learned often make use of arguments to tell the program what to do. For instance, when copying files you would type

```
> cp foo bar
```

This invokes the program `cp` with two arguments: `foo`, the source file; and `bar`, the destination file. Your program can also read its command-line arguments. They are passed to the `main` routine as two function arguments: an `int` and a `char**` (pointer to array of `char*`). These arguments are conventionally named `argc` (argument count plus one[6]) and `argv` (argument vector containing all the arguments). While the notion of pointer-to-array-of-pointers may seem complex, you can think of each of `argv[0]` through `argv[argc-1]` as a `char*` null-terminated character array (ie. a C-style string). The first value, `argv[0]`, is always the program name as input at the terminal. The `argc-1` arguments given (if any) start at `argv[1]`.

6.3 Exercises

1. Build and run `name`, providing input and seeing how the program responds
2. Build `args`; run it from both the NetBeans terminal and from the command-line. Try providing different argument values to see how it responds.

[6] This is always argument count plus one, because it marks the size of `argv` which always starts with the program path, hence it always has at least one element. The arguments are then appended, so the length is number-of-arguments-plus-one.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(int argc,char **argv)
6  {
7      cout << "There are " << argc << " arguments given" << endl;
8      for(int i=0;i<argc;++i)
9          cout << "argv[" << i << "] = \"" << argv[i] << '"' << endl;
10     return 0;
11 }
```

Figure 19: Source code showing use of command-line arguments

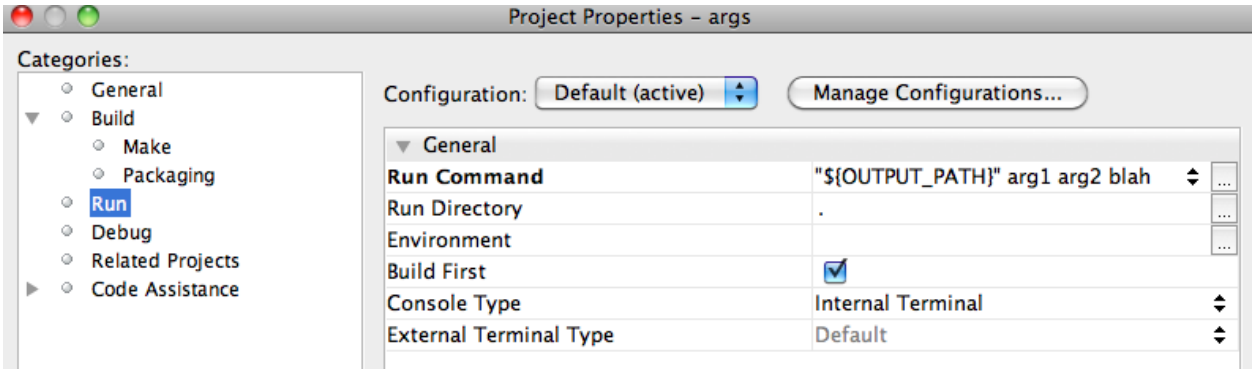

Figure 20: Setting command-line arguments by appending to the "Run command" field

Figure 21: Program output when provided with command-line arguments as shown in Fig 20