

CSCI131

Introduction to Computer Systems

Task Group 1: Basics

Complete the exercises, and try to get them marked, before attempting the assignment!

This first group of tasks introduces the use of the NetBeans IDE for C and C++ development, and gives you a first taste of C. Using NetBeans, you will explore aspects like source level debugging with breakpoints, and viewing the actual code that has been generated for your program. You will also revise work on the representation of different data types that you will have covered briefly in CSCI114.

In these early tasks, there will be little difference between the C++ that you have learnt and C - it will mainly come down to use of a different input-output library (C's `stdio` instead of C++'s `iostream`). Later in session, there will be a short lecture segment explaining more about the differences between the languages. C does not have classes (but you haven't learnt anything much about C++'s classes yet so that shouldn't worry you). C has some oddities with regard to declaring types (it's a little clumsier than C++); C has more restrictions on where local variables may be declared (recent revisions of the C standard have made it closer to C++). Probably the major challenge for you will relate to the use of pointers and dynamic data. Of course you haven't learnt anything about these yet in C++ - which is why the lecture segment on C doesn't come at the start of this subject. Fundamentally, C and C++ use pointers and dynamic data in exactly the same way but there are syntactic differences that can sometimes make C code a little more confusing than the corresponding C++ code for dynamic data and for passing arguments by reference.

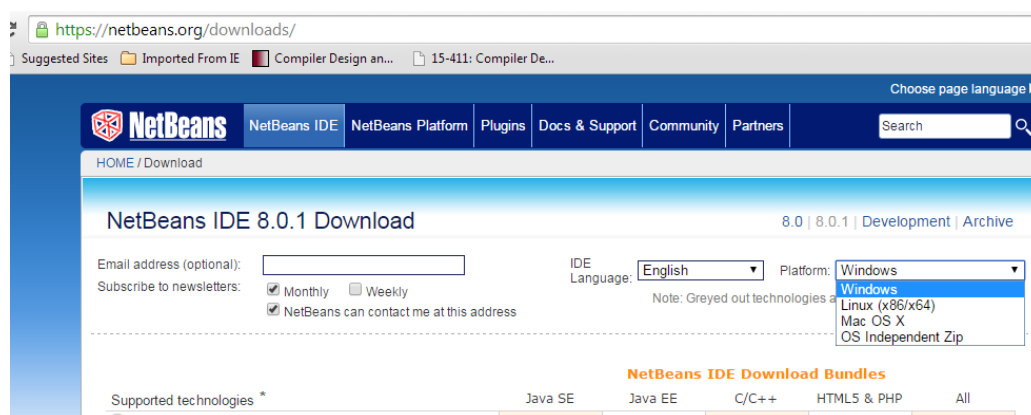
There are several exercise tasks in this task group as detailed below. At the end of this document there is a link to the assignment. You should try to complete all the exercise tasks before attempting the assignment task. When you have completed all the exercises, just show the working versions to a tutor so as to get marked off.

◀ NetBeans IDE - The Smarter and Faster Way to Code

NetBeans IDE lets you quickly and easily develop Java desktop, mobile, and web applications, as well as HTML5 applications with HTML, JavaScript, and CSS. The IDE also provides a great set of tools for PHP and C/C++ developers. It is free and open source and has a large community of users and developers around the world.

The NetBeans Integrated Development Environment is installed on the computers in the laboratory. This development environment has been used to support the practical work in a number of other CS subjects such as CSCI110, CSCI213, CSCI222, CSCI398, and CSCI399. (Use in these subjects depends on preferences of the current lecturers.) You are advised to use the laboratory. You can install all the software on your own machine and work independently, but you may find that setting up software systems consumes lots of time that would have been better spent on working on the tasks in the laboratory where everything is already installed and operational.

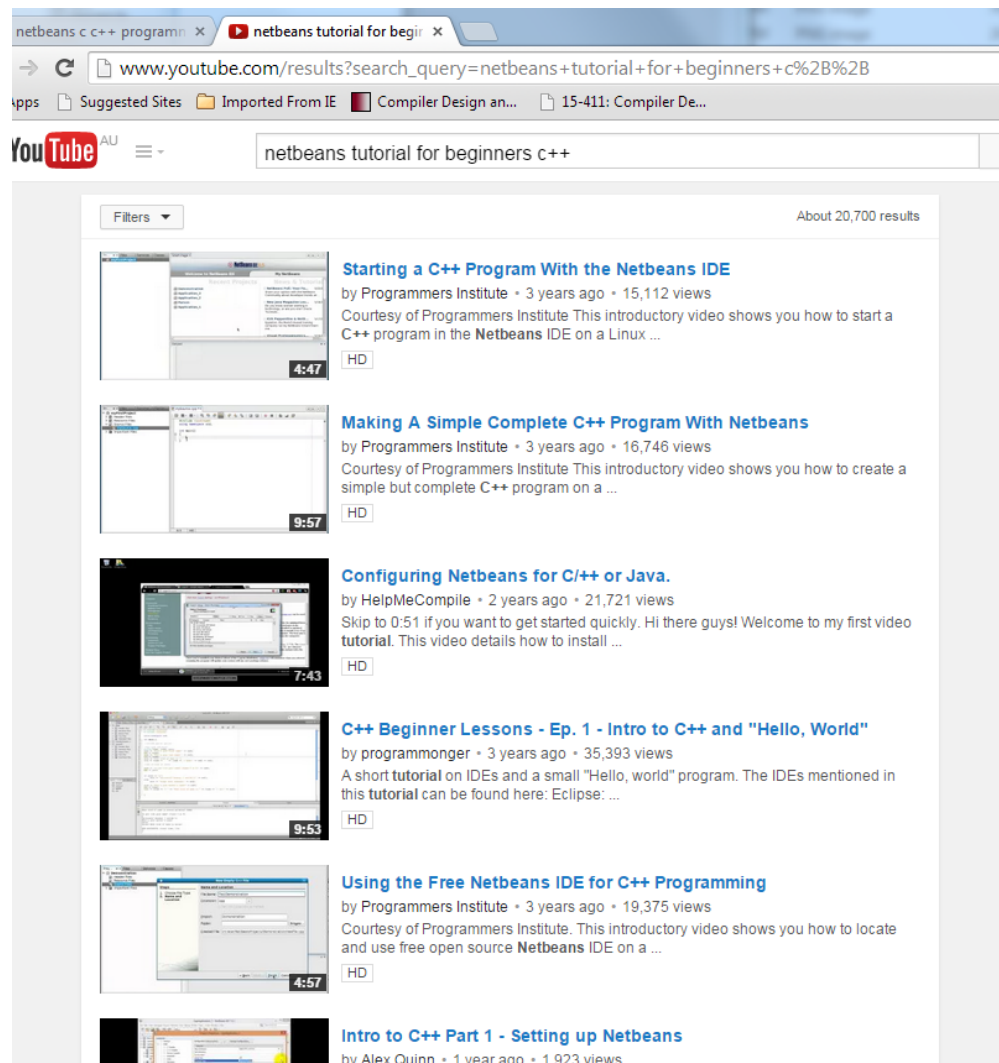
The NetBeans IDE is free. It can be downloaded from netbeans.org; there are versions for Windows, Linux, and MacOS. It is implemented in Java. If you wish to install NetBeans on your own operating system, **you must first install the current version of Java JDK**. (A Linux system will have a version of Java; this version is sponsored by IBM, it is usually out-of-date with respect to the Oracle release. You will need to get the current Java from Oracle.) For NetBeans, you should install the “All” bundle – support for web, Java, and C/C++ coding.



There are tutorials on YouTube that illustrate how to install NetBeans. **If you are working on Windows, you will also need MinGW or Cygwin** (these packages emulate the Unix/C development environment on Windows).



After installation on your own machine, you will need to do some configuration – “activating modules” etc. (Setting up NetBeans with Cygwin/mingw involves a number of steps including editing “PATH” entries in your Windows environment. There is a guide at [netbeans.org - https://netbeans.org/community/releases/80/cpp-setup-instructions.html](https://netbeans.org/community/releases/80/cpp-setup-instructions.html).) There are also tutorials at YouTube that may help.



In the lab, it will simply be a matter of starting NetBeans from the Ubuntu “Dash”:



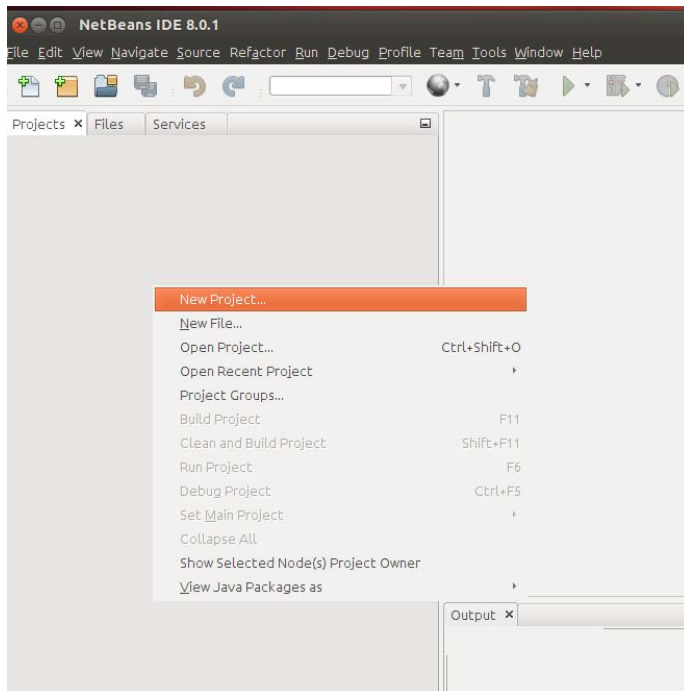
The screenshots that follow will use NetBeans version 8.01; the version in the lab will probably be more recent but there shouldn't be any major differences.



NetBeans works with “projects”. A NetBeans project will typically correspond to a single exercise or assignment. A NetBeans project will consist of a main directory containing the files that you create and numerous files and directories that NetBeans creates to support development. By default, NetBeans will place all these project folders inside “NetBeansProjects” - a sub-directory that it creates in your home directory. You will probably find it better to create your own sub-directories so as to keep work for different subjects quite separate.

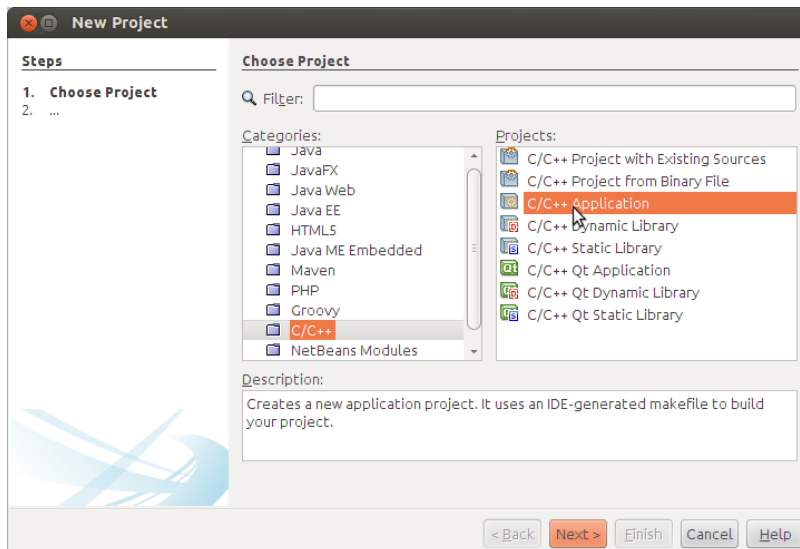
Task 1: A C++ project (2 marks)

Start by creating a new NetBeans project:

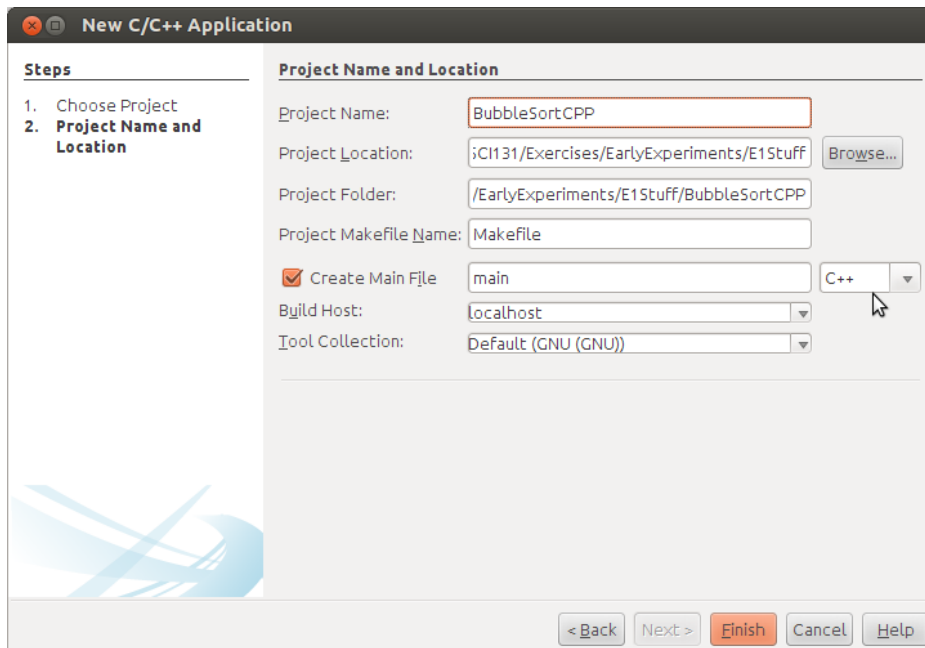


When it starts, NetBeans will display a window with some standard panes. There is a pane where you can view projects, files, and “services” (services are for things like connections to databases or web servers; they won’t be used much in CSCI131). (If you end up with some odd display where some standard elements seem to be missing, use the “Windows/Reset Windows” option from NetBeans’ menu-bar). Right-click in the “Projects” pane to get a pop-up menu that lets you create a new project.

Your first project should be a “C/C++ application”; and this one will be C++. This demonstration application illustrates one of the sorting algorithms that you might have studied in CSCI103. You first choose C/C++ from the “Categories” list (if there isn’t such an entry in your system at home then you haven’t configured your system completely, go and watch those YouTube tutorials again!). Then pick **C/C++ Application** from the updated Projects list:

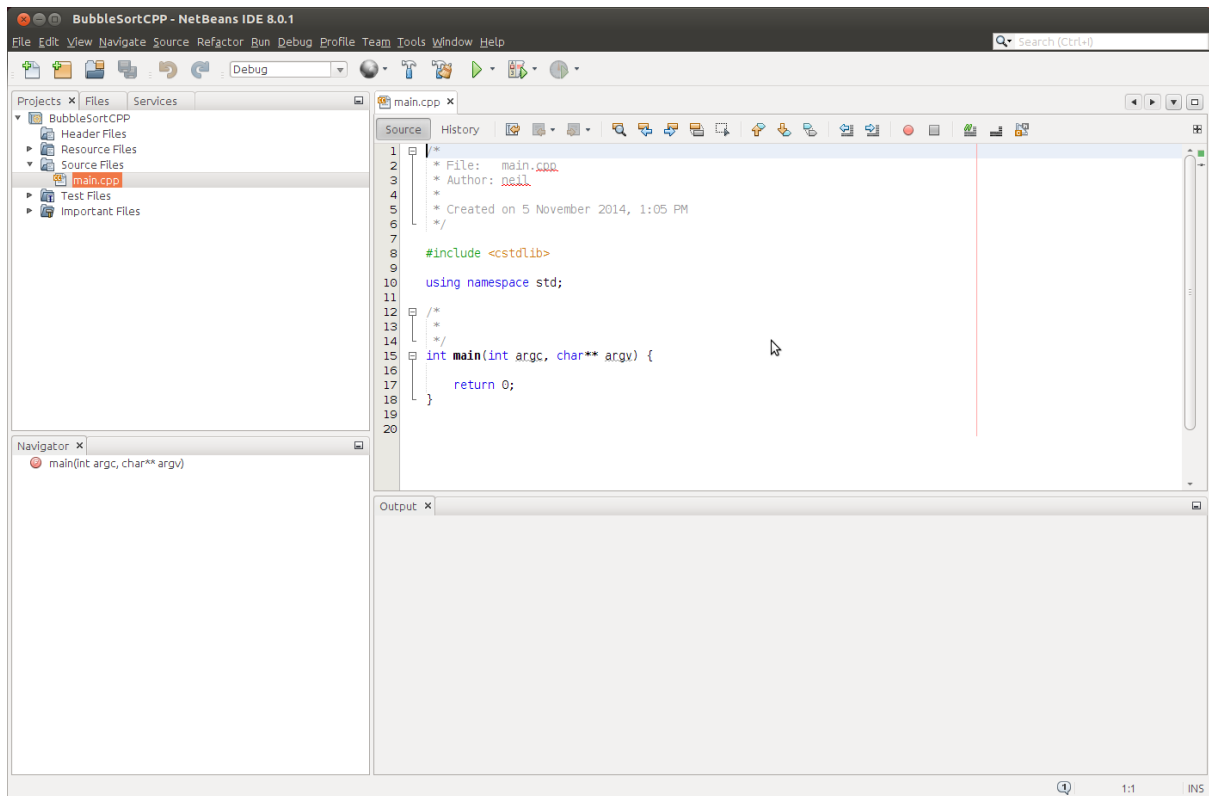


You then have to choose between C, C++, and Fortran. For this first part, make it C++, and give the project the name “BubbleSortCPP” and choose an appropriate directory to hold the files.



NetBeans will generate a standard C++ project, with a stub “main.cpp”, and a number of other files (configurations.xml, project.xml, Makefile) and directories (build, dist, nbproject). It shows the project structure in the project window. This project structure view has “folders” for “Source files”, “Header files”, etc; these don’t correspond to actual directories, they are just a conceptual device for helping clarify the role of files that you create. When you create “header” files, you should add them to the “Header Files” folder; C++ (or C) files should go in the “Source Files” folder; (if there are some data files that must be supplied with the program code, they can go in the “Resources” folder). The “Test Files” folder is for test code using a “Unit Test Framework” – you will learn about the use of unit test frameworks in CSC1222.

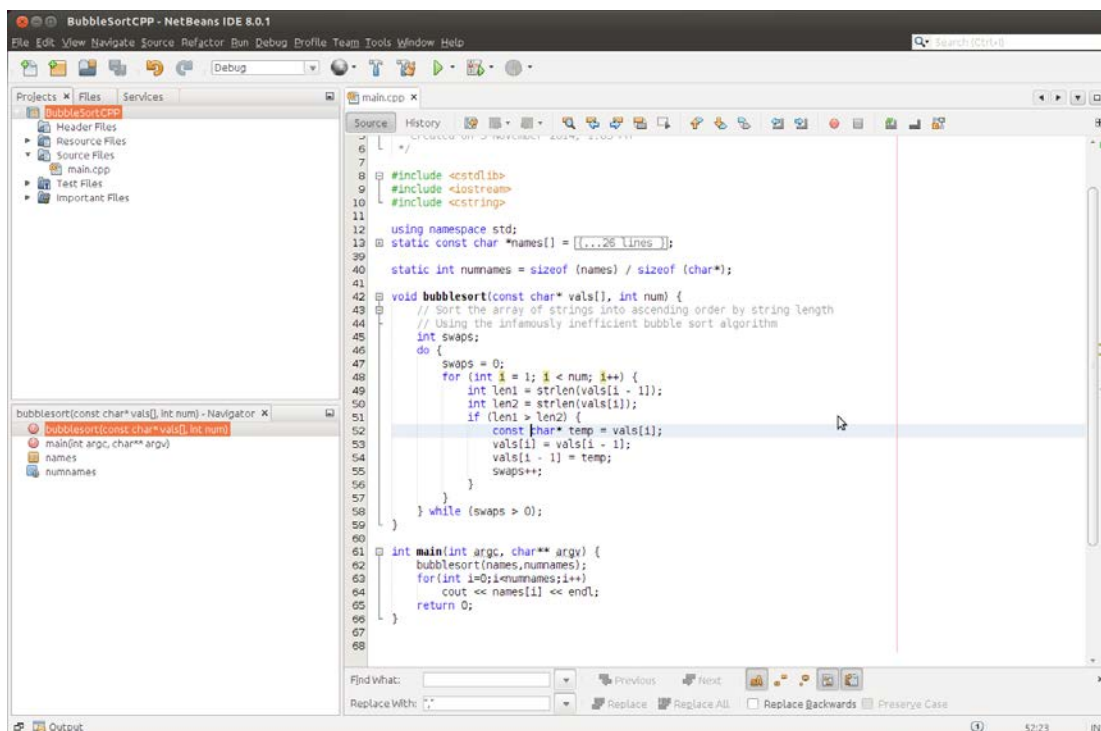
To the right of the “Projects/Files/Services” pane, there will be a main editor window.



You will also see a “Navigator” pane. This shows the structure of the file that is currently displayed in the editor window. For example, if you had a long C++ file with many functions and a few filescope variables open in the editor, the function names (and variable names) would be listed in the Navigator pane. Clicking on an entry in the Navigator pane will scroll the text in the editor to display the definition of the chosen entry.

This first exercise involves entering code and data for a simple sort demonstration program. The sort algorithm implemented is “BubbleSort”; you probably met this in CSCI103 where it would have been introduced as the worst performing of the $O(n^2)$ sort algorithms! The program will work on an array of strings, and will sort them into ascending order by string length.

Enter the code and data. (What do you mean when you say it's not readable? Zoom!)



The stub C++ program that NetBeans created would have had #includes for cstdlib, and iostream along with a “using namespace std;” directive. As the program needs the strlen() function, it will also need to include cstring.

```
#include <cstdlib>
#include <iostream>
#include <cstring>

using namespace std;
```

The data are the most popular names for children born in USA in 2012.

```
static const char *names[] = {...26 lines};

static int numnames = sizeof (names) / sizeof (char*);
```

Obviously it would require too much effort for students to enter 200 names, so here are the data in a form that you can cut and paste into the code:

```
static const char *names[] = {
    "Aiden", "Jackson", "Ethan", "Liam", "Mason", "Noah", "Lucas", "Jacob", "Jayden",
    "Jack", "Logan", "Ryan", "Caleb", "Benjamin", "William", "Michael", "Alexander",
    "Elijah", "Matthew", "Dylan", "James", "Owen", "Connor", "Brayden", "Carter", "Landon",
    "Joshua", "Luke", "Daniel", "Gabriel", "Nicholas", "Nathan", "Oliver", "Henry",
    "Andrew", "Gavin", "Cameron", "Eli", "Max", "Isaac", "Evan", "Samuel", "Grayson",
    "Tyler", "Zachary", "Wyatt", "Joseph", "Charlie", "Hunter", "David", "Anthony",
    "Christian", "Colton", "Thomas", "Dominic", "Austin", "John", "Sebastian",
    "Cooper", "Levi", "Parker", "Isaiah", "Chase", "Blake", "Aaron", "Alex", "Adam",
    "Tristan", "Julian", "Jonathan", "Christopher", "Jace", "Nolan", "Miles", "Jordan",
```



```

"Carson", "Colin", "Ian", "Riley", "Xavier", "Hudson", "Adrian", "Cole", "Brody",
"Leo", "Jake", "Bentley", "Sean", "Jeremiah", "Asher", "Nathaniel", "Micah",
"Jason", "Ryder", "Declan", "Hayden", "Brandon", "Easton", "Lincoln", "Harrison",
"Sophia", "Emma", "Olivia", "Isabella", "Ava", "Lily", "Zoe", "Chloe", "Mia", "Madison",
"Emily", "Ella", "Madelyn", "Abigail", "Aubrey", "Addison", "Avery", "Layla", "Hailey",
"Amelia", "Hannah", "Charlotte", "Kaitlyn", "Harper", "Kaylee", "Sophie", "Mackenzie",
"Peyton", "Riley", "Grace", "Brooklyn", "Sarah", "Aaliyah", "Anna", "Arianna", "Ellie",
"Natalie", "Isabelle", "Lillian", "Evelyn", "Elizabeth", "Lyla", "Lucy", "Claire", "Makayla",
"Kylie", "Audrey", "Maya", "Leah", "Gabriella", "Annabelle", "Savannah", "Nora",
"Reagan", "Scarlett", "Samantha", "Alyssa", "Allison", "Elena", "Stella", "Alexis",
"Victoria", "Aria", "Molly", "Maria", "Bailey", "Sydney", "Bella", "Mila", "Taylor",
"Kayla", "Eva", "Jasmine", "Gianna", "Alexandra", "Julia", "Eliana", "Kennedy", "Brianna",
"Ruby", "Lauren", "Alice", "Violet", "Kendall", "Morgan", "Caroline", "Piper", "Brooke",
"Elise", "Alexa", "Sienna", "Reese", "Clara", "Paige", "Kate", "Nevaeh", "Sadie", "Quinn",
"Isla", "Eleanor"
};

```

The bubblesort routine:

```

void bubblesort(const char* vals[], int num) {
    // Sort the array of strings into ascending order by string length
    // Using the infamously inefficient bubble sort algorithm
    int swaps;
    do {
        swaps = 0;
        for (int i = 1; i < num; i++) {
            int len1 = strlen(vals[i - 1]);
            int len2 = strlen(vals[i]);
            if (len1 > len2) {
                const char* temp = vals[i];
                vals[i] = vals[i - 1];
                vals[i - 1] = temp;
                swaps++;
            }
        }
    } while (swaps > 0);
}

```

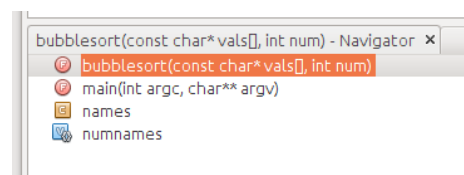
The main program:

```

int main(int argc, char** argv) {
    bubblesort(names, numnames);
    for(int i=0; i<numnames; i++)
        cout << names[i] << endl;
    return 0;
}

```

When you have entered the code, the Navigator window will display the structure of your main.cpp file – showing two function declarations and two filescope variable declarations:



If you enter code incorrectly, the NetBeans editor will highlight errors with red tags in the margin:

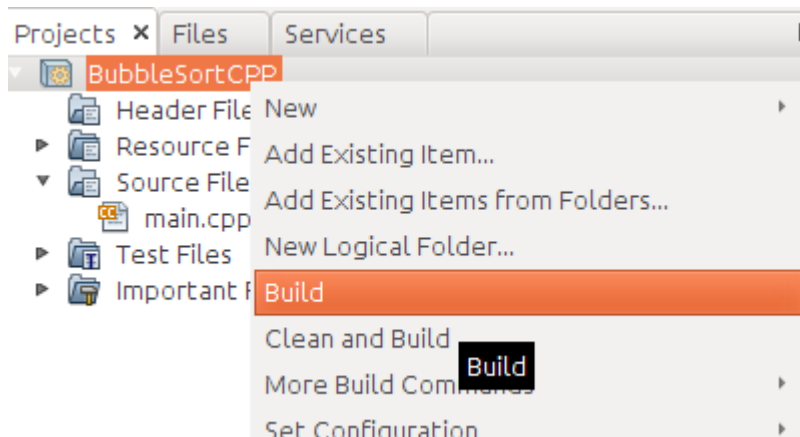
```
35     Ruby , Lauren , Alice , Violet , Kendall ,
36     "Elise", "Alexa", "Sienna", "Reese", "Clara", "E
37     "Isle", "Eleanor
38 };
39
40 static int numnames = sizeof (names) / sizeof (char*)
41
42 void bubblesort(const char* vals[], int num) {
43     // Sort the array of strings into ascending order
44     // Using the infamously inefficient bubble sort
45     int swaps;
46     do {
47         swaps = 0;
48         for (int i = 1; i < num; i++) {
49             int len1 = strlen(vals[i - 1]);
50             int len2 = strlen(vals[i]);
51             if (len1 > len2) {
52                 const char* temp = vals[i]
53                 vals[i] = vals[i - 1];
54                 vals[i - 1] = temp;
55                 swaps++;
56             }
57         }
58     } while (swaps > 0);
59 }
60
61 int main(int argc, char** argv) {
62     bubblesort(names, numnames);
63     for(int i=0; i<numnames; i++)
64         cout << names[i] << endl;
```

Fix all errors! There is no point trying to run a program that is known to have errors. Don't just stare at the screen and grumble. Don't call for a tutor and tell him/her that "My program doesn't work". Move the cursor over the red tag – an explanation of the error will pop-up.

```
48         for (int i = 1; i < num; i++) {
49             int len1 = strlen(vals[i - 1]);
50             int len2 = strlen(vals[i]);
51             if (len1 > len2) {
52                 const char* temp = vals[i]
53                 vals[i] = vals[i - 1];
54                 vals[i - 1] = temp;
55                 swaps++;
56             }
```

(NetBeans will use yellow tags for warnings; you should check any warnings that are flagged on your code, but mostly they are minor issues that can be ignored.)

When you have entered the code correctly, “build” the application (select the project in the projects pane, right-click to get the pop-up menu):



The program should be compiled and linked. A report on the build process is shown in the output pane:

```
Output - BubbleSortCPP (Build) x
"/usr/bin/make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJECTS= .build-conf
make[1]: Entering directory `/home/neil/Teaching/NewCSCI131/Exercises/EarlyExperiments/E1Stuff/Bubble
"/usr/bin/make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux-x86/bubblesortcpp
make[2]: Entering directory `/home/neil/Teaching/NewCSCI131/Exercises/EarlyExperiments/E1Stuff/Bubble
mkdir -p build/Debug/GNU-Linux-x86
rm -f "build/Debug/GNU-Linux-x86/main.o.d"
g++ -c -g -MMD -MP -MF "build/Debug/GNU-Linux-x86/main.o.d" -o build/Debug/GNU-Linux-x86/main.o m
mkdir -p dist/Debug/GNU-Linux-x86
g++ -o dist/Debug/GNU-Linux-x86/bubblesortcpp build/Debug/GNU-Linux-x86/main.o
make[2]: Leaving directory `/home/neil/Teaching/NewCSCI131/Exercises/EarlyExperiments/E1Stuff/Bubble
make[1]: Leaving directory `/home/neil/Teaching/NewCSCI131/Exercises/EarlyExperiments/E1Stuff/Bubble
```

The first invocation of g++ is the compilation step, it compiles the main.cpp file creating a main.o file; the second invocation of g++ completes the build step creating an executable program.

Of course, you may get a compilation error when compiling your own code. The following is a compilation error report that would result if you removed the “#include <cstring>” line:

```
Output - BubbleSortCPP (Clean) x BubbleSortCPP (Build) x
"/usr/bin/make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux-x86/
make[2]: Entering directory `/home/neil/Teaching/NewCSCI131/Exercises/Ear
mkdir -p build/Debug/GNU-Linux-x86
rm -f "build/Debug/GNU-Linux-x86/main.o.d"
g++ -c -g -MMD -MP -MF "build/Debug/GNU-Linux-x86/main.o.d" -o build/D
main.cpp: In function 'void bubblesort(const char**, int)':
main.cpp:49:42: error: 'strlen' was not declared in this scope
make[2]: *** [build/Debug/GNU-Linux-x86/main.o] Error 1
make[2]: Leaving directory `/home/neil/Teaching/NewCSCI131/Exercises/Ear
```

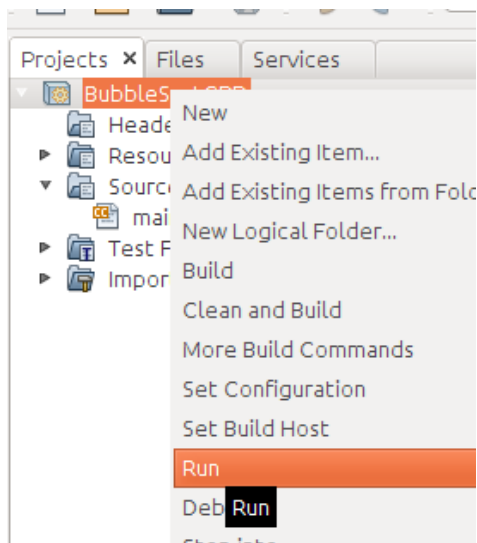
The error message acts as a hyperlink; click on it and the editor will display the file with the error, scrolling the text so that the line with the error is in view:

```

45     int swaps;
46     do {
47         swaps = 0;
48         for (int i = 1; i < num; i++) {
49             int len1 = strlen(vals[i - 1]);
50             int len2 = strlen(vals[i]);
51             if (len1 > len2) {
52                 const char* temp = vals[i];

```

If you entered the code correctly, you should not get any errors. The compilation should succeed allowing you to run the application (select project, right-click to get pop-up menu, select Run):



It should run correctly, producing output in the “Output” pane:



Task 2: Using the source level debugger (1mark)

Have you ever wished that you could somehow look inside your code and see what it was doing?

Have you ever been reduced to inserting dozens of tracer output messages?

Help is here – the source level debugger.

In one of the subjects in the main programming stream (CSCI114, CSCI124, CSCI204 – most likely CSCI124) you will be taught how to use the **gdb** source level debugging tool at the command line. Gdb is quite hard to use from the command line and consequently few students ever make much use of this approach.

But an IDE such as NetBeans can make it all simple.

Suppose we wanted to observe details of the actions in the bubblesort function – looking at the array entries that were getting swapped. We would want to see what was going on in this code:

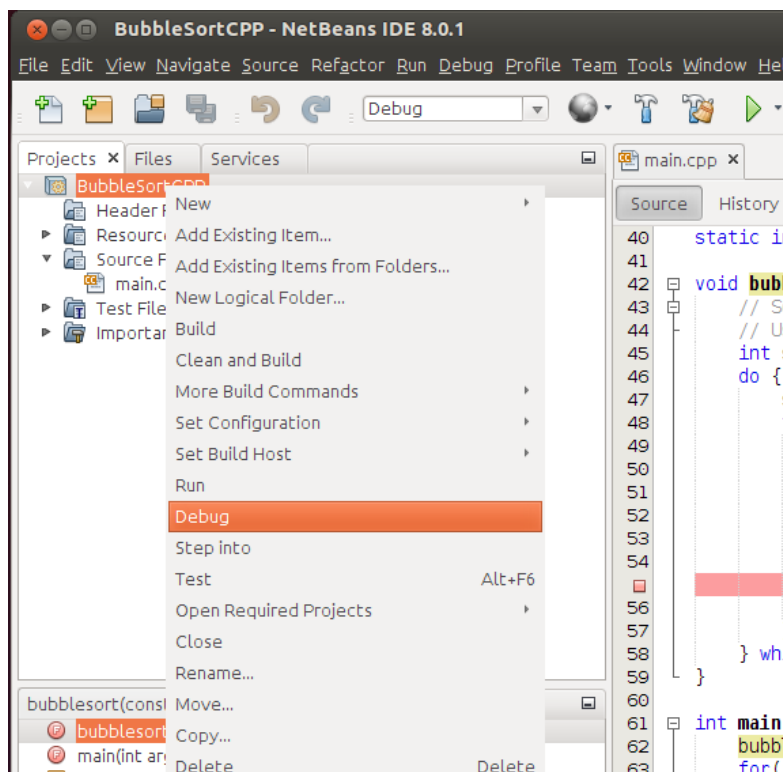
```
// USING THE INTRINSICALLY INEFFICIENT BUBBLE  
int swaps;  
do {  
    swaps = 0;  
    for (int i = 1; i < num; i++) {  
        int len1 = strlen(vals[i - 1]);  
        int len2 = strlen(vals[i]);  
        if (len1 > len2) {  
            const char* temp = vals[i];  
            vals[i] = vals[i - 1];  
            vals[i - 1] = temp;  
            swaps++;  
        }  
    }  
} while (swaps > 0);
```

With a debugger, we can set a “**breakpoint**” in the code. When execution reaches a breakpoint, the program is stopped; the debugger is started; the user can enter commands to get variable values displayed.

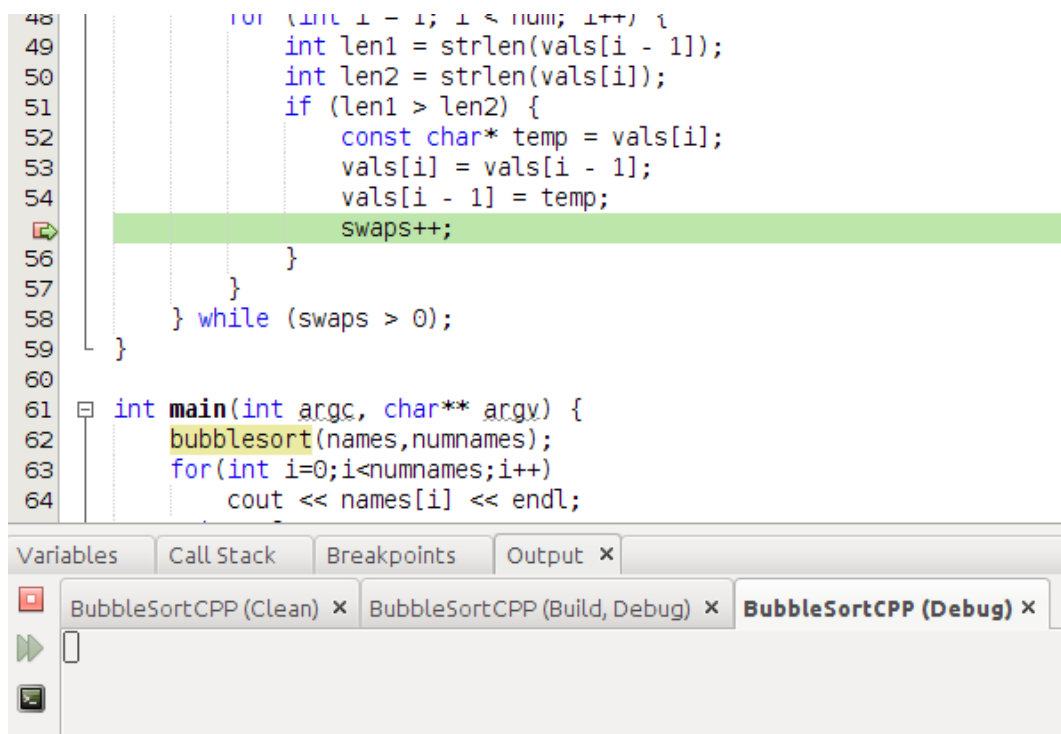
Using NetBeans, you start by adding one (or more) breakpoints; these are set by clicking in the left margin that shows the line numbers for the code:

```
52      const char* temp = vals[i];  
53      vals[i] = vals[i - 1];  
54      vals[i - 1] = temp;  
55      swaps++;  
56  }  
57  }  
58  } while (swaps > 0);  
59
```

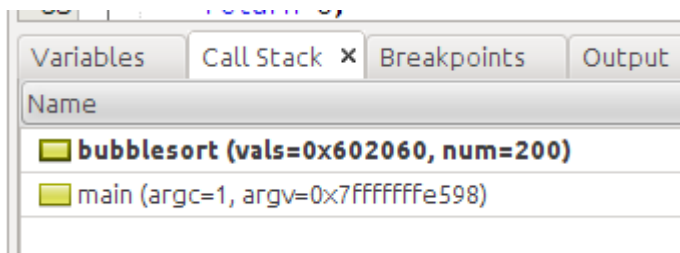
You then invoke “Debug” rather than “Run” when starting execution of the project:



When execution reaches the breakpoint, gdb will intervene. NetBeans will communicate with gdb and find where the program is stopped and provide means of interrogating gdb to view variable values. NetBeans provides interactive displays of gdb’s data. It first shows where execution has reached:

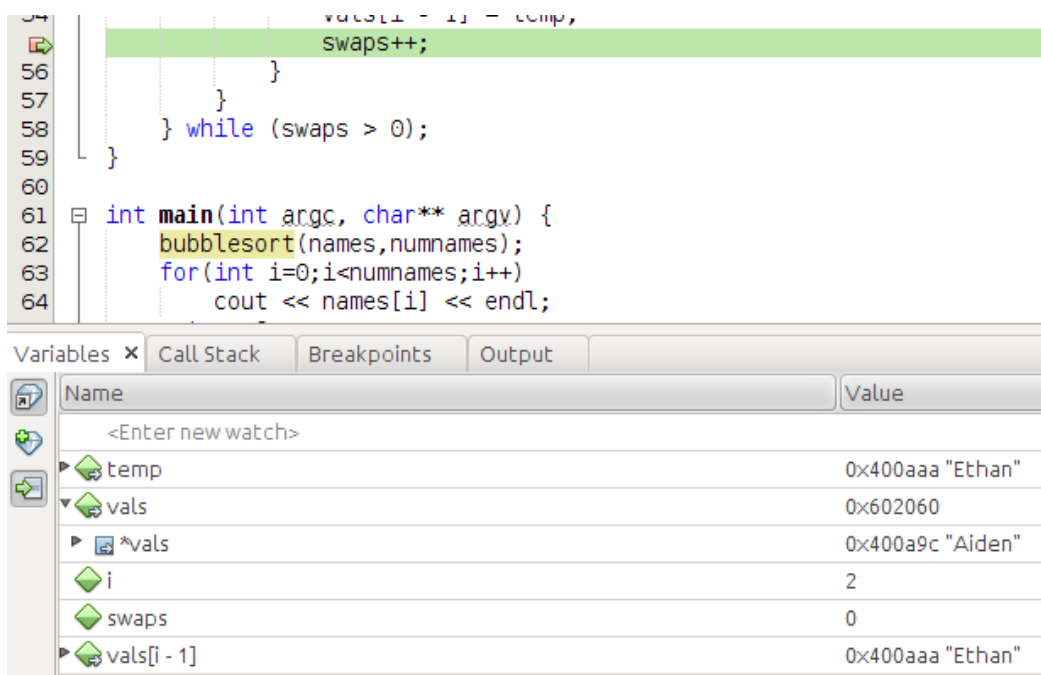


You would start by checking the call stack:



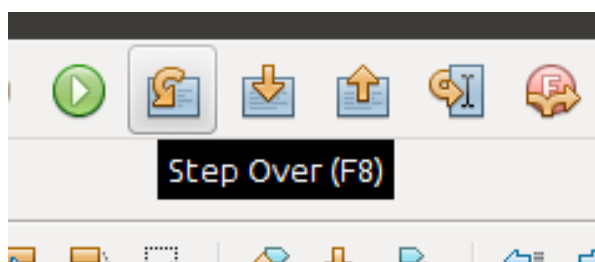
There is nothing of interest here in the call stack (it simply shows that bubble sort has been called from main – but we knew that). In more complex programs where functions get called from different points in the code, it is important to know the context for your debugging.

After viewing the call stack, you can check on variable values:




Bugs are often revealed by variables having implausible values (because you forgot to initialise things, or misused a pointer, or ...). In addition to looking at variable values, you can enter expressions, combining values in variables, and have gdb display the result.

You can then resume execution. You have several options available via the controls in one of NetBeans' menu bars.







The  control resumes execution. Your program will run on until it again encounters a breakpoint.

The other options let you step through your code ~ line by line. You can observe exactly what goes on, and you can check variable values at any stage.

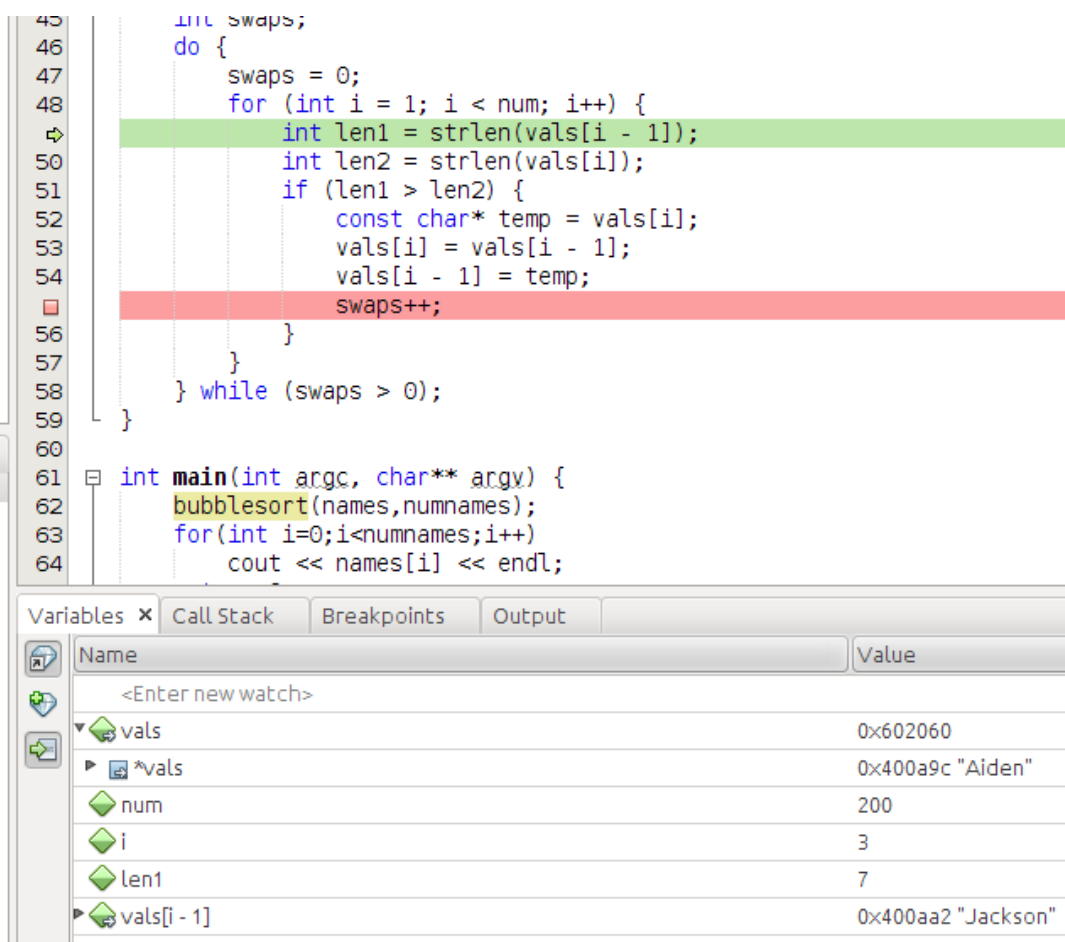


The  control “steps over” a statement. The statement is executed – you just don’t get to see the details. This is useful if the statement includes a call to some standard function like `strlen()`; you wouldn’t want to see the code that works out the length of a string.



The  control “steps into” a statement. The statement is executed. If it involves a function call, you will see the code of that function and have to step through the function’s code line-by-line.

Try stepping through some cycles of the loop in bubblesort. Here it is about to call `strlen` in the next iteration of the for loop:



```

45     int swaps;
46     do {
47         swaps = 0;
48         for (int i = 1; i < num; i++) {
49             int len1 = strlen(vals[i - 1]);
50             int len2 = strlen(vals[i]);
51             if (len1 > len2) {
52                 const char* temp = vals[i];
53                 vals[i] = vals[i - 1];
54                 vals[i - 1] = temp;
55                 swaps++;
56             }
57         }
58     } while (swaps > 0);
59 }
60
61 int main(int argc, char** argv) {
62     bubblesort(names, numnames);
63     for(int i=0; i<numnames; i++)
64         cout << names[i] << endl;

```

| Name | Value |
|-------------------|--------------------|
| <Enter new watch> | |
| vals | 0x602060 |
| *vals | 0x400a9c "Aiden" |
| num | 200 |
| i | 3 |
| len1 | 7 |
| vals[i - 1] | 0x400aa2 "Jackson" |

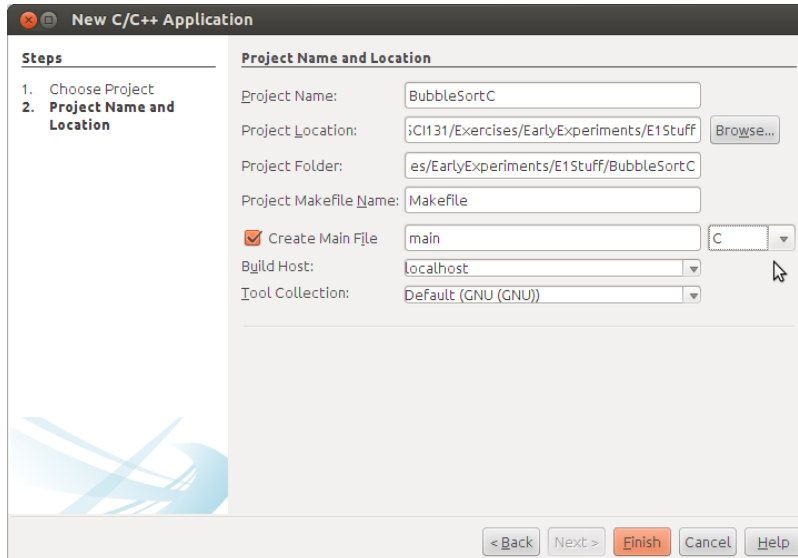
There is a more detailed tutorial on C++ debugging in the [netbeans.org](https://netbeans.org/kb/docs/cnd/debugging.html) documentation, at <https://netbeans.org/kb/docs/cnd/debugging.html>. There are also YouTube tutorials, e.g. http://www.youtube.com/watch?v=3o_OPAv2ea0

Detour: Create a new C++ project and cut and paste in the code of one of the exercises or assignments that caused you problems in CSCI114. Explore the use of source level debugging – would it have helped you back then?

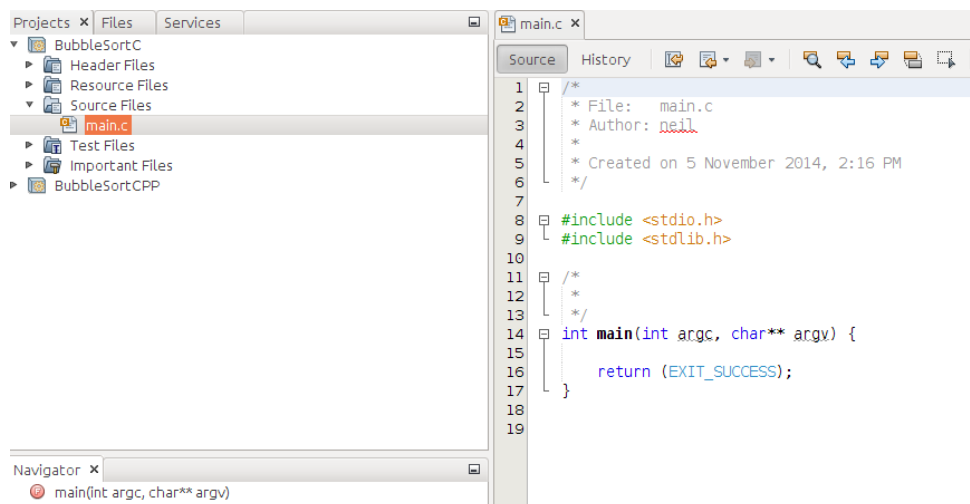
Task 3: C language version of bubble sort application (1 mark)

Time for C.

Create a new project, BubbleSortC, with a C language version of the application.



The generated project is basically similar – different #include statements is the only noticeable change really:



You will need to #include <string.h> for the string functions used from C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const char *names[] = {
```

There are no changes to the data declarations. There are some small changes needed in the bubblesort function:

```
1 void bubblesort(const char* vals[], int num) {
2     // Sort the array of strings into ascending order
3     // Using the infamously inefficient bubble sort
4     int swaps;
5     do {
6         swaps = 0;
7         for (int i = 1; i < num; i++) {
8             int len1 = strlen(vals[i - 1]);
9             int len2 = strlen(vals[i]);
10            if (len1 > len2) {
11                const char* temp = vals[i];
12                vals[i] = vals[i - 1];
13                vals[i - 1] = temp;
14                swaps++;
15            }
16        }
17    } while (swaps > 0);
18 }
19 }
```

C used to be very strict about where variables could be declared; they had to be defined at the head of a block. C's rules have been relaxed in recent versions. However, a C++ like declaration such as "for (int i=0; i<num; i++)" is still not valid in the default version of C for the gcc compiler (it is allowed in C-99, but as of 2014 that was not gcc's default C dialect). So, in the C code, the variable 'i' has to be declared before the for-loop (which gives it a different scope actually). Otherwise there are no changes in bubblesort.

The main() function uses printf() from the stdio library.

```
61 int main(int argc, char** argv) {
62     bubblesort(names, numnames);
63     for(int i=0; i<numnames; i++)
64         printf("%s\n", names[i]);
65
66     return (EXIT_SUCCESS);
67 }
```

This older input/output library is occasionally used from C++ (you will use it a little in one of the 200-level CS subjects using C, Java also offers a version of printf in one of its output libraries so you might use printf in CSCI213). Input uses the "scanf" function; output uses "printf". The "scanf" and "printf" functions take a "format string" argument and a variable number of data elements that are to be read or written. This first example is very simple. The format string for "printf" is simply "%s\n"; the %s means print a value as a string, the \n adds a newline character after the printed string. The other argument for printf is one of the entries in the char* array of strings.

There are reference summaries for stdio at :

http://www.tutorialspoint.com/c_standard_library/stdio_h.htm

<http://www.cplusplus.com/reference/cstdio/>

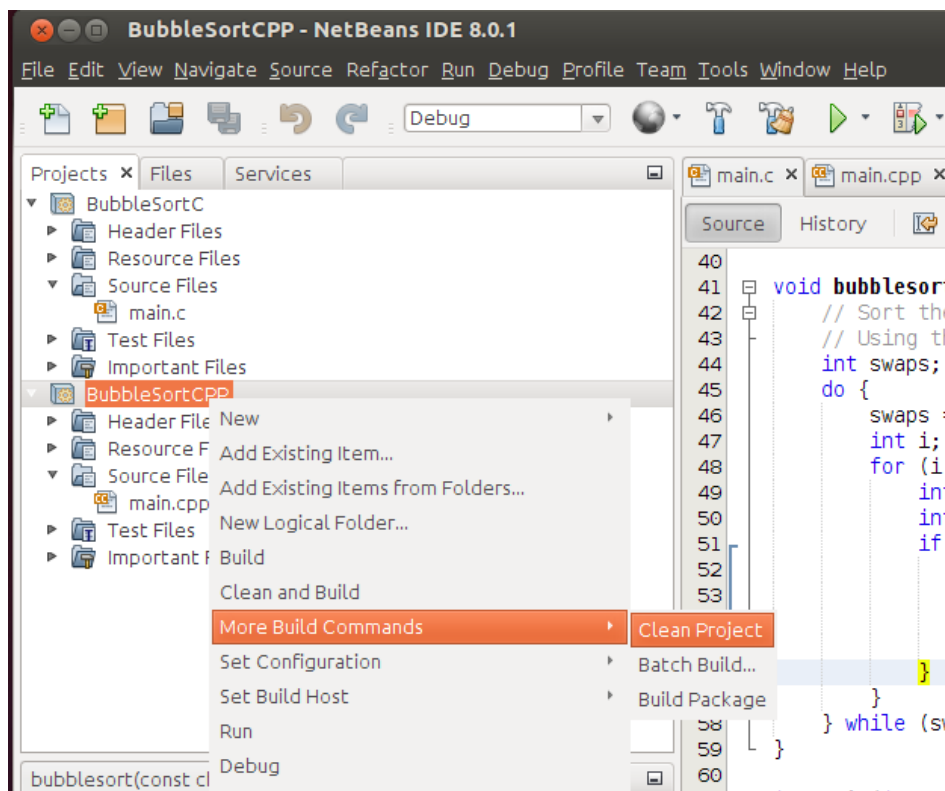
and

<http://www.cplusplus.com/reference/cstdio/printf/>

(the last reference lists all the different formats that you can use).

The BubbleSortC project should also compile and run without problems.

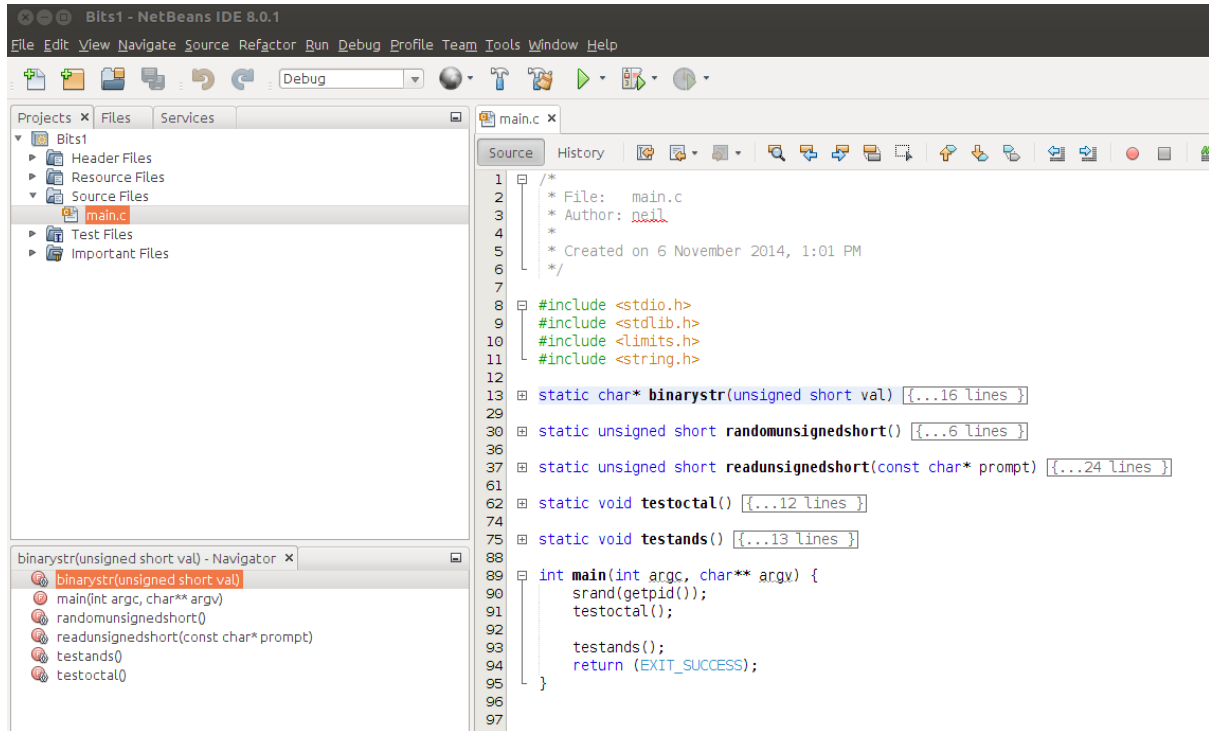
When you have completed and demonstrated all the exercises to a tutor, you should “tidy up” the projects. The generated .o and executable files can take up lots of your disk space. You should clear these away when no longer needed – use the “**More Build Commands/Clean**” option on the projects:



Task 4: A bit more with C(1 mark)

The next project, again in C, will give you a little practice in the use of octal as a means of representing bit patterns, and allow you to try some basic bit-manipulation operations.

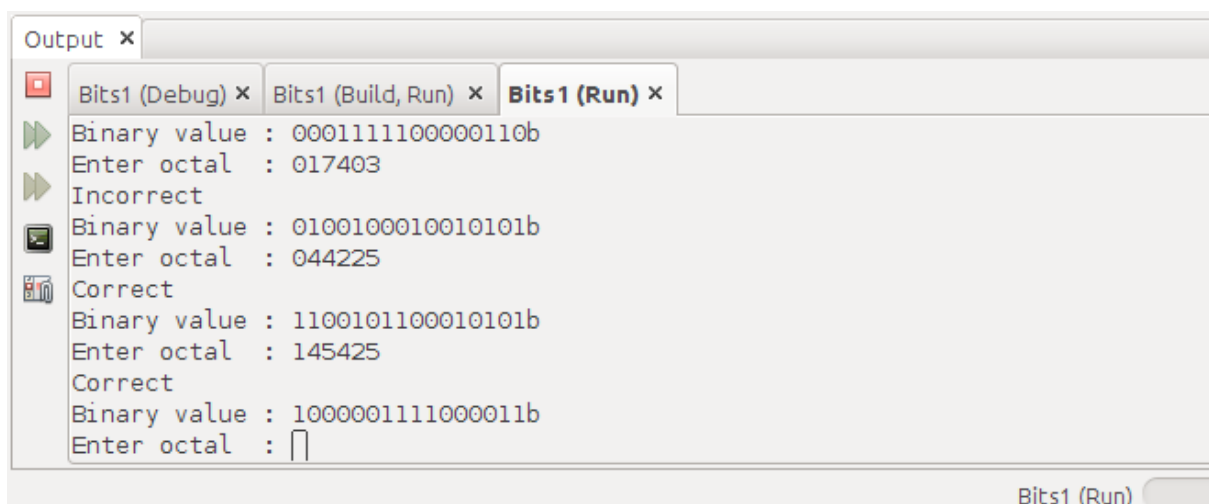
Create a new project, Bits1, as a C project. This application has a main() and five other functions:



The screenshot shows the NetBeans IDE 8.0.1 interface. The 'Projects' window on the left displays the 'Bits1' project structure, including 'Header Files', 'Resource Files', 'Source Files' (containing 'main.c'), 'Test Files', and 'Important Files'. The 'main.c' file is open in the 'Source' window, showing the following code:

```
1  /*  
2   * File:   main.c  
3   * Author: nall  
4   *  
5   * Created on 6 November 2014, 1:01 PM  
6   */  
7  
8  #include <stdio.h>  
9  #include <stdlib.h>  
10 #include <limits.h>  
11 #include <string.h>  
12  
13 static char* binarystr(unsigned short val) {...16 lines}  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29 static unsigned short randomunsignedshort() {...6 lines}  
30  
31  
32  
33  
34  
35  
36  
37 static unsigned short readunsignedshort(const char* prompt) {...24 lines}  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62 static void testoctal() {...12 lines}  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75 static void testands() {...13 lines}  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89 int main(int argc, char** argv) {  
90     srand(getpid());  
91     testoctal();  
92  
93     testands();  
94     return (EXIT_SUCCESS);  
95 }  
96  
97
```

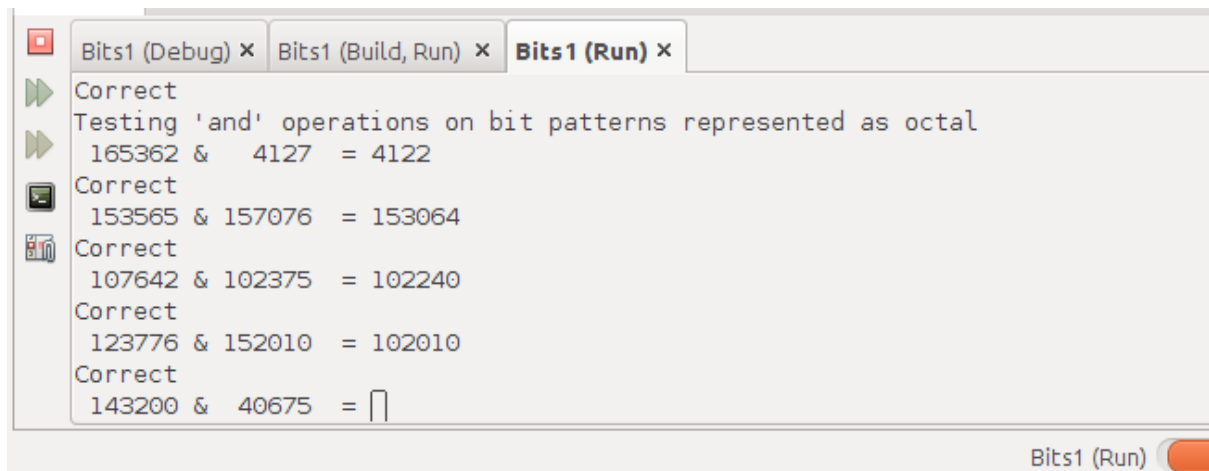
When correctly built and run, the application will test a user on their ability to encode bit patterns in octal and perform “and” operations on bit patterns that are represented in octal. First, it presents a number of bit patterns as strings of 0 and 1 characters; the user must enter the octal representation.



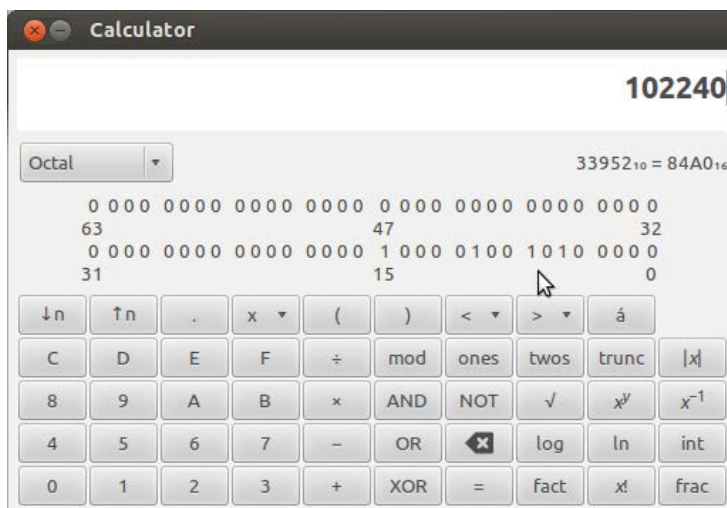
The screenshot shows the 'Output' window of the NetBeans IDE. It displays the execution of the 'Bits1' application. The output shows the application prompting the user to enter the octal representation of a given binary value. The user enters '017403' for the first binary value '0001111100000110b', which is marked as 'Incorrect'. The user enters '044225' for the second binary value '0100100010010101b', which is marked as 'Correct'. The user enters '145425' for the third binary value '1100101100010101b', which is marked as 'Correct'. The user enters an empty string for the fourth binary value '1000001111000011b', which is marked as 'Incorrect'.

```
Output x  
Bits1 (Debug) x Bits1 (Build, Run) x Bits1 (Run) x  
Binary value : 0001111100000110b  
Enter octal  : 017403  
Incorrect  
Binary value : 0100100010010101b  
Enter octal  : 044225  
Correct  
Binary value : 1100101100010101b  
Enter octal  : 145425  
Correct  
Binary value : 1000001111000011b  
Enter octal  :   
Bits1 (Run)
```

Next it shows to bit patterns as octal strings, and requires the user to enter the octal string equivalent to the bit pattern that would result:



If you have difficulty in doing the “&” operations, you can always cheat by using the Calculator that is installed on an Ubuntu system:



The code that you must enter

main():

Initialise C’s random number generator (using the process’s process identifier number as the seed), and then invoke the two test functions:

```
int main(int argc, char** argv) {
    srand(getpid());
    testoctal();

    testands();
    return (EXIT_SUCCESS);
}
```

testoctal():

Iterate a few times picking a random unsigned short integer value, displaying this as a bit string, asking the user for an octal input, and checking the user’s response:

```

1 static void testoctal() {
    printf("Testing understanding of binary and octal patterns\n");
    int i;
    int correct = 0;
    for (i = 0; i < 8; i++) {
        unsigned short sval = randomunsignedshort();
        printf("Binary value : %s\n", binarystr(sval));
        unsigned short sentry = readunsignedshort("Enter octal : ");
        if (sentry == sval) printf("Correct\n");
        else printf("Incorrect\n");
    }
}

```

If there are no values to be embedded, the “format string” for a printf can simply be the text that is to be printed – as here with the “*Incorrect\n*” string and other text elements.

(The variable “correct” was going to have been used to keep a count of the number of correct responses – but this somehow got forgotten! NetBeans uses a wavy underline to flag the variable declaration as suspicious code; so warning of a potential problem.)

testands():

Iterate a few times picking pairs of random unsigned short integer values, displaying these, asking the user to enter the octal representation of the bit pattern that would result from an “and” operation combining the values:

```

static void testands() {
    printf("Testing 'and' operations on bit patterns represented as octal\n");
    int i = 0;
    for (i = 0; i < 8; i++) {
        unsigned short one = randomunsignedshort();
        unsigned short two = randomunsignedshort();
        printf(" %6ho & %6ho ", one, two);
        unsigned short guess = readunsignedshort(" = ");
        unsigned short value = one & two;
        if(guess==value) printf("Correct\n");
        else printf("Incorrect\n");
    }
}

```

The printf() statement here has a more elaborate format string. There are two output elements, each with format %6ho. The % signs identify a format specification for a value. Here the “6” sets a field width (you may have used the setw io-manipulator in C++), the “h” flags the data as being a short value, and the “o” specifies octal output.

binarystr():

This function takes an unsigned short integer argument and generates a binary representation as a sequence of sixteen ‘0’ or ‘1’ characters in a static character buffer. It returns the address of this buffer.

Use of a static buffer like this is a common feature in many of the functions provided by an operating system. But there are potential problems. Another call to the function will overwrite any existing data. So if an application needs to keep a string, it must make a copy. In this example application, the string is immediately output by that `printf` statement in the `testoctal()` function; so there will not be any need to save it.

The function works by first creating a 1-bit “mask” in the left-most bit of a short integer (creating the bit pattern 1000000000000000). There is then a loop in which the mask and data value are “anded” together, if the result is non-zero a “1” should be added to the bit representation otherwise a zero; after the ‘and’ operation, the value is shifted left one place. (The value `SHRT_MIN` is in `limits.h`.)

```
static char* binarystr(unsigned short val) {
    // Assume that short is 16-bit
    // Uses static buffer, consumer must make copy of data; not thread safe
    static char buffer[18];
    memset(buffer, 0, 18);

    int mask = SHRT_MIN;
    int count;
    for (count = 0; count < 16; count++) {
        char ch = (val & mask) ? '1' : '0';
        val = val << 1;
        buffer[count] = ch;
    }
    buffer[count] = 'b';
    return buffer;
}
```

A final ‘b’ (for binary) character is added to the character array; the eighteenth element is always NUL guaranteeing that the result is a valid C string.

randomunsignedshort():

Uses C’s `rand` function, modifying the result to give a random 16-bit bit pattern:

```
static unsigned short randomunsignedshort() {
    int val = rand() % 65536;
    val -= 32768;
    short sval = (short) val;
    return (unsigned short) sval;
}
```

readunsignedshort():

This function repeatedly prompts the user to enter a short integer value, which must be entered as an octal string. If the data entered are invalid, they are discarded; the prompt is repeated. The code will loop until a valid data value has been entered or the program is terminated (`cntrl-C`) or input is aborted (`cntrl-D`).

```

static unsigned short readunsignedshort(const char* prompt) {
    // Prompts for and reads an unsigned short value
    // Keeps prompting until it gets a valid input
    for (;;) {
        printf("%s", prompt);
        unsigned short enteredval;
        int transferred = scanf("%ho", &enteredval);
        if (transferred != 1) {
            if (feof(stdin)) {
                printf("Input terminated, program will exit\n");
                exit(0);
            }
            printf("invalid input\n");
            // The following funny format string says
            // read all characters up to newline and simply ignore them
            // then read the newline character as a character but discard it
            // as well.
            // Serves to clear junk from the input line.
            scanf("%*[^\\n]%*c");
            continue;
        }
        return enteredval;
    }
}

```

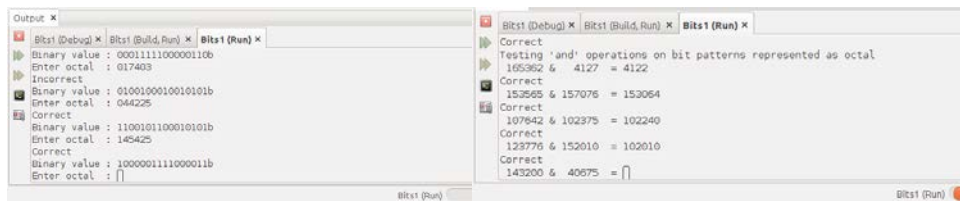
This is the first example with `scanf()`. `scanf` needs to be passed the addresses of the data elements whose values are to be read; it needs these addresses so that it can store values that have been input into the correct variables. The “&” address-of operator is used when passing arguments to `scanf`. Here we have `&enteredval` – pass the address of the variable `enteredval`. The format string is `%ho`; there is no width specifier, the “h” again flags a short value, and the “o” specifies octal conversion.

The `scanf` function returns a count of the number of data values successfully read. Here, we want one value read. If the input data are invalid (e.g. inputs like “hello world”, “9876”) `scanf` will stop reading at the first invalid character and return 0 (the input buffer would be left with “h” or “9” as the front character). (An input like “1295” will accept the “12” as a valid octal value, leaving “95” in the input buffer to disrupt the next read!) `scanf` will return 0 if end-of-file is encountered (cntrl-D).

Dealing with bad input data from the keyboard is a problem in both C++ and C. You need to empty the input buffer. This code attempts to read and discard all characters up to a newline character, then read and discard that new line character. This is done via the odd format string in the second `scanf`. (The “*” entries in the % format fields are the way to tell `stdio` to try to discard input. It is hard to deal with all possible bad inputs on `stdin`; there really is no good way to flush the input buffer. The code given will often help but may not always work.)

Enter all the code. Fix the typing errors that NetBeans flags with red tags. Fix any compiling errors such as those resulting from forgetting to `#include <string.h>`. Finally run the program. Practice a little with binary and octal.

Running the code:



Task 5: More gdb practice(1mark)

Modify the code to create a deliberate error by commenting out the return statement in the `binarystr` function (pretend for now that you simply forgot to type this line when copying my code):

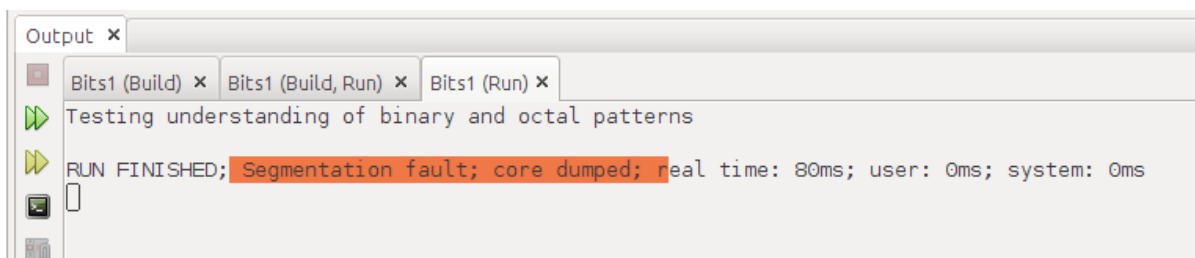
```
static char* binarystr(unsigned short val) {
    // Assume that short is 16-bit
    // Uses static buffer, consumer must make copy of data; not th
    static char buffer[18];
    memset(buffer, 0, 18);

    int mask = SHRT_MIN;
    int count;
    for (count = 0; count < 16; count++) {
        char ch = (val & mask) ? '1' : '0';
        val = val << 1;
        buffer[count] = ch;
    }
    buffer[count] = 'b';
    // return buffer;
}
```

Surprisingly, the modified code compiles without any error report. (You would think that the compiler would have noticed that the function says that it will return a pointer to character but returns nothing; but compilers tend to assume that the programmer knows what he/she is doing and so let through quite a lot of dubious code.)

You program compiled – *result happiness*.

You try to run:



Result misery.

What does the typical student do when his/her program crashes like this?

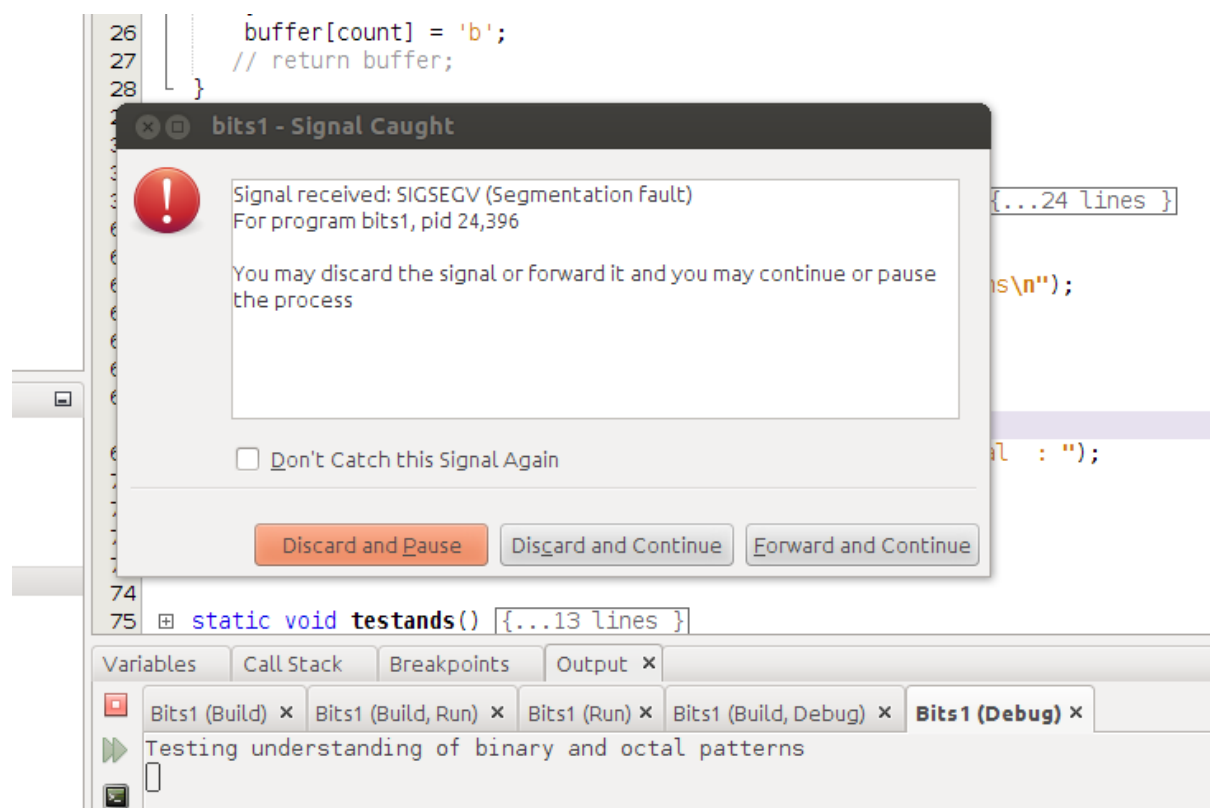
The typical student swears a bit, wonders how on earth such an error might be found, feeling bored and aggrieved seeks consolation via entertainment on Facebook/Twitter/YouTube, if in the lab calls the tutor and says “*My program doesn’t work*”, fidgets, stares at the screen, blames the lecturer, tells their colleagues that C and C++ are terrible programming languages, and then maybe inserts 100 tracer output lines and tries again.

What does the smart student do?

He/she uses the debugger.

Launch the program again using “Debug” rather than “Run”.

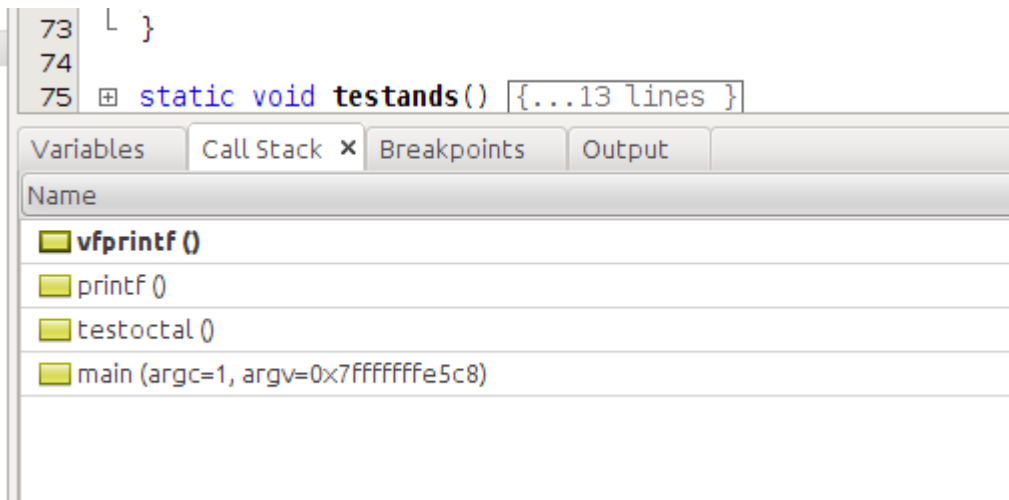
The program still dies, but gdb catches the problem and working with NetBeans reports details to the user.



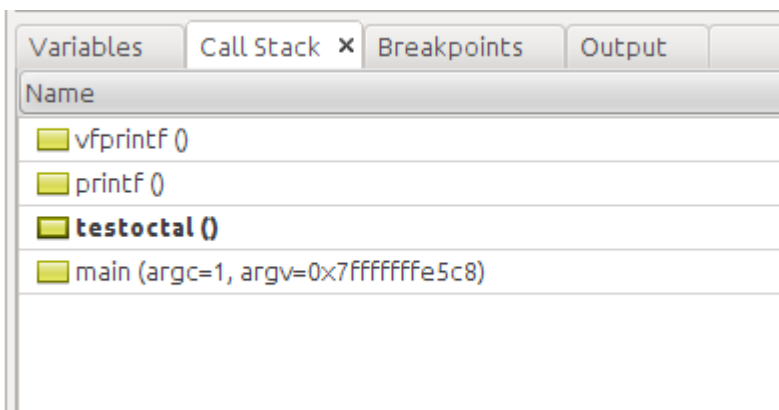
The line where the crash occurred is highlighted (it’s the purple shaded line in the back window); the dialog offers some options – the useful one is “Discard and Pause”.

Using this option allows the user to pick over the corpse of the program looking for clues as to why it died.

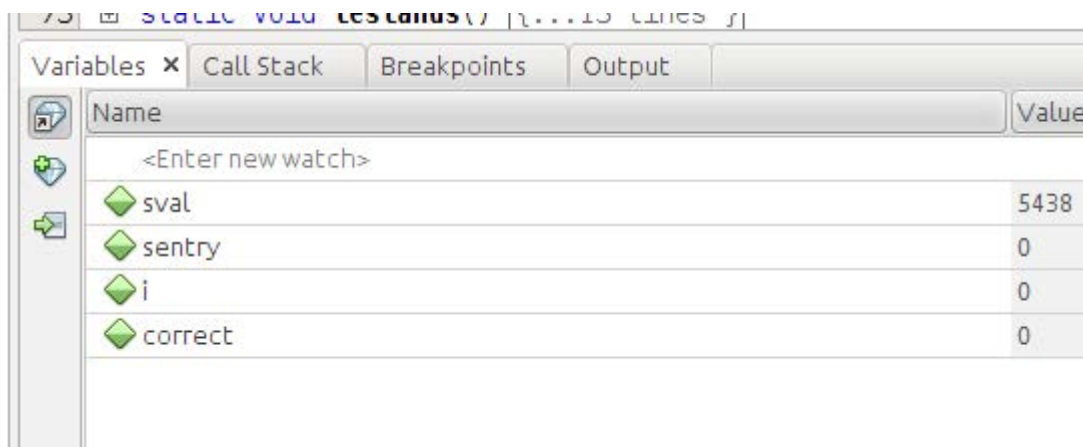
Look at the call-stack.



It shows that death occurred somewhere deep inside the code of the stdio library (in some function called `vprintf`, called from `printf`, called from `testoctal`). It's unlikely that there is an error in the stdio library code; it's more likely that the program has passed some inappropriate data to the string output functions. So move up the stack, and select the `testoctal` stack frame:



Could next look at variables:



Nothing much there; but at least have established that the problem occurs in the line with `printf`:

```

unsigned short sval = randomunsignedshort();
printf("Binary value : %s\n", binarystr(sval));
unsigned short sentry = readunsignedshort("Enter octal : ");

```

So would probably look next at the `binarystr()` function and realise that the return statement was missing. The function would still have returned *something* – random bits left in the stack; the crash occurred when those bits were used as an address in `vprintf()`.

The debugger can help a lot.

Dramatic crashes with segment faults are common with programs that use pointers (incorrectly). You will soon become familiar with the problem when you start using pointers in CSCI124.

Try a different error. Again rewrite the `binarystr` function, this time making the buffer array a local stack variable rather than a static variable. (This is actually a more plausible error than forgetting to return a result).

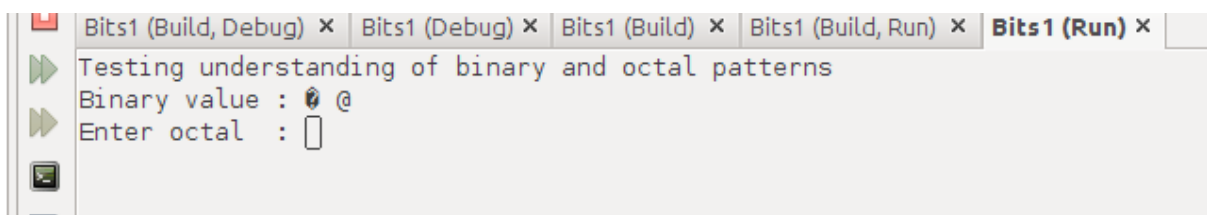
```

static char* binarystr(unsigned short val) {
    // Assume that short is 16-bit
    // Uses static buffer, consumer must make copy of data; not thread safe
    char buffer[18];
    memset(buffer, 0, 18);

    int mask = SHRT_MIN;
    int count;
    for (count = 0; count < 16; count++) {
        char ch = (val & mask) ? '1' : '0';
        val = val << 1;
        buffer[count] = ch;
    }
    buffer[count] = '\0';
    return buffer;
}

```

The program doesn't crash; but it's not happy. (This program didn't crash; a similar error in some other more complex program might have more dramatic consequences.)



The typical student will fail to notice that the output is corrupt and will randomly enter some data; the smart student guesses that the `binarystr` function has apparently generated the wrong result. Rather than going and adding lots of tracer output statements, he/she finds the problem by putting a break point where `binarystr` returns so that the value can be checked when the program is re-run with the gdb debugger in control. Try it.

```

21     for (count = 0; count < 16; count++) {
22         char ch = (val & mask) ? '1' : '0';
23         val = val << 1;
24         buffer[count] = ch;
25     }
26     buffer[count] = 'b';
27     return buffer;
28 }

```

| Name | Value |
|-------------------|--------|
| <Enter new watch> | |
| buffer | {...} |
| buffer[0] | 49 '1' |
| buffer[1] | 49 '1' |
| buffer[2] | 49 '1' |
| buffer[3] | 48 '0' |
| buffer[4] | 48 '0' |

The buffer appears to hold the correct characters; so why didn't they get printed?

This problem is caused by the function returning the address of a local array that had been created on the stack. The array "buffer" is in essence discarded when control leaves the binarystr function. When the program later tried to use that array, the character data that it contained had been overwritten.

Actually, in this case the compiler is good and does output a warning – but how often do students bother about compiler warnings?

```


gcc -c -g -MMD -MP -MF "build/Debug/GNU-Linux-x86/main.o.d" -o build/Debug/GNU-Linux-
main.o main.c: In function 'binarystr':
main.c:27:4: warning: function returns address of local variable [enabled by default]
main.c: In function 'readunsignedshort':

```

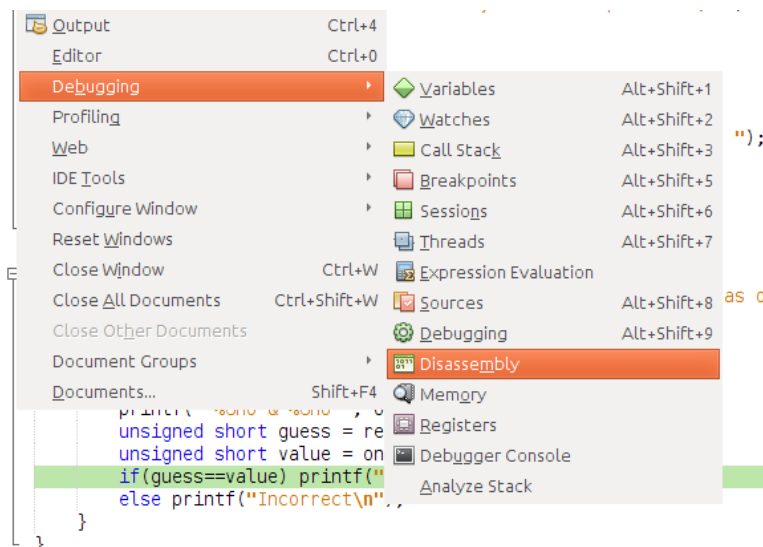
Gdb can also show you the instruction sequence that was generated for your code. This is more of novelty value; it's unlikely you would ever really understand the instructions. You can try as follows:

1. Set a breakpoint and run up to that point.

```
static void testands() {
    printf("Testing 'and' operations on bit patterns repre:
    int i = 0;
    for (i = 0; i < 8; i++) {
        unsigned short one = randomunsignedshort();
        unsigned short two = randomunsignedshort();
        printf(" %6ho & %6ho ", one, two);
        unsigned short guess = readunsignedshort(" = ");
        unsigned short value = one & two;
        if(guess==value) printf("Correct\n");
        else printf("Incorrect\n");
    }
}
```



2. Find the option to display code (it's in the menus):



3. View the code – at least the “and” instruction (for value = one & two) is obvious:

```
!          unsigned short value = one & two;
testands+104: movzwl  -0x6(%rbp),%eax
testands+108: movzwl  -0x8(%rbp),%edx
testands+112: and     %edx,%eax
testands+114: mov     %ax,-0x2(%rbp)
!          if(guess==value) printf("Correct\n");
testands()
testands+118: movzwl  -0x4(%rbp),%eax
testands+122: cmp     -0x2(%rbp),%ax
```

Another detour: Resurrect some more of your previous programs (particularly any programs that crashed or produced weird results) and explore further the use of the “source level debugging” approach using NetBeans and gdb.

Task 6: Some last bits (and hexes)(1 mark)

This tiny project is intended simply to get you to practice a bit more with bit data and octal and hexadecimal notations. Create another C application project – Bits2.

This program will test the user on binary, octal and hexadecimal notations. It has a simple main() driver function, and three other functions. The binarystr function can be simply copy-pasted from the existing code. The readunsignedshort function is a slightly more sophisticated version of that shown previously.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>

enum INPUTSTYLE {
    OCTAL, HEX
};

typedef enum INPUTSTYLE Style;

static char* binarystr(unsigned short val) {...16 lines }

static unsigned short readasbits() {...17 lines }

static unsigned short readunsignedshort(const char* prompt, Style s) {...26 lines }

int main(int argc, char** argv) {...21 lines }
```

The declaration of the enumerated type illustrates difference between C and C++. In C++, declarations of structs and enums are type declarations; you can define variables or function arguments that are instances of the new types. This is not the case in C. If you wanted an instance of the enum INPUTSTYLE in C (e.g. a variable “astyle”), you must define it as “enum INPUTSTYLE astyle;”. The constant repetition of the “enum” or “struct” type qualifiers is irritating. So, C programmers generally use a “typedef” statement to add a new type. The statement typedef enum INPUTSTYLE Style makes “Style” a type recognised by the C compiler. Code can declare arguments and variables of type “Style”; as in the second argument to the revised readunsignedshort() function.

The main() has a driver loop that gets the user to enter a binary pattern and the octal and hexadecimal equivalents.

```

int main(int argc, char** argv) {
    int correct = 0;
    do {
        printf("Enter bit string for short integer :\n");
        unsigned short valb = readasbits();
        printf("Value as 16-bit bit string : %s\n", binarystr(valb));
        printf("Enter hex and octal representations :\n");
        unsigned short valh = readunsignedshort("hex :", HEX);
        unsigned short valo = readunsignedshort("octal:", OCTAL);
        if ((valb == valh) && (valb == valo)) {correct++; printf("Correct\n"); }
        else {
            printf("Incorrect! Start again\n");
            printf("Value as octal          :          %6ho\n", valb);
            printf("Value as hex           :          %6hx\n", valb);
        }
        // discard any remaining input, newlines etc
        char discard[100];
        gets(discard);
    } while (correct < 5);
    return (EXIT_SUCCESS);
}

```

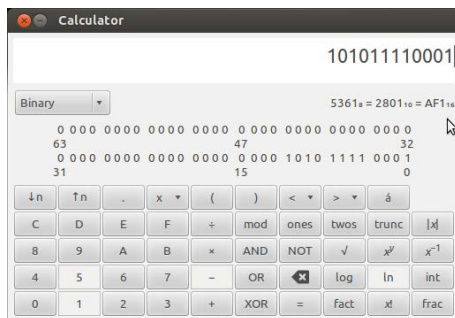
It keeps testing the user until he/she has made five successive successful attempts

```

Bits2 (Build, Run) × Bits2 (Run) ×
Enter bit string for short integer :
bits :11
Value as 16-bit bit string : 0000000000000011b
Enter hex and octal representations :
hex :3
octal:3
Correct
Enter bit string for short integer :
bits :101
Value as 16-bit bit string : 0000000000000101b
Enter hex and octal representations :
hex :7
octal:8
invalid input
octal:3
Incorrect! Start again
Value as octal          :          5
Value as hex           :          5
Enter bit string for short integer :
bits :101111
Value as 16-bit bit string : 0000000000010111b
Enter hex and octal representations :
hex :2f
octal:57
Correct
Enter bit string for short integer :
bits :101011110001
Value as 16-bit bit string : 0000101011110001b
Enter hex and octal representations :
hex :af1
octal:5361
Correct
Enter bit string for short integer :
bits :

```


Yes, you can cheat. Use the calculator on Ubuntu. Enter the bit sequence – 1010111 – as binary; use “=”; select octal or hex and the equivalent value is shown.



The readasbits function has the following definition:

```
static unsigned short readasbits() {
    char buf[256];
    printf("bits :");
    gets(buf);
    unsigned short val = 0;
    int len = strlen(buf);
    int i = 0;
    // skip any leading whitespace
    while ((i < len) && (isspace(buf[i]))) i++;

    while ((i < len) && ((buf[i] == '0') || (buf[i] == '1')) {
        val = val << 1;
        if (buf[i] == '1') val = val | 01;
        i++;
    }
    return val;
}
```

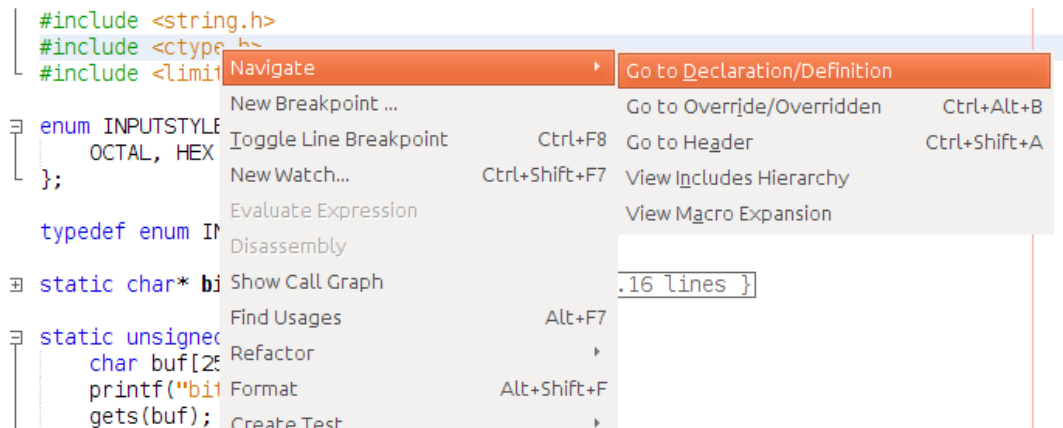
It reads a line using stdio's `gets()` function (you have probably used iostream's `cin.getline`). Of course you need some space where the input characters are to be stored; hopefully, a character array of size 256 will be enough. (If it isn't enough, well the program's stack gets corrupted. Sad. Or good if you are a hacker doing it deliberately.) The `gets()` function obviously needs the address of the array but the code doesn't use the "&" address of operator. *Arrays are always passed by reference in C and C++*; the value passed is the address of the [0] element of the array. You should already have encountered this usage in C++.

The code first has a loop that should let it step over any whitespace characters. *How do you test whether characters are whitespace?*

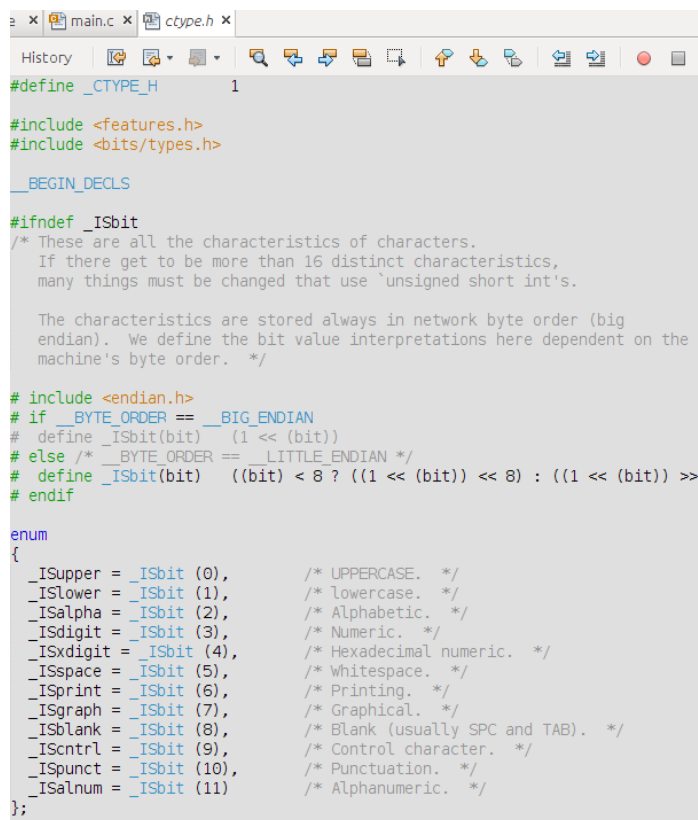
You might vaguely remember that there is a header file with functions that test character types – `cctype.h` for C, `cctype` in C++. But what was the function name?

You can get help in the IDE.

Enter the `#include` statement; then right click on the statement, you can get a link to the file with the declarations:



The file is opened (in a non-editable window) letting you check the actual function names:



After discarding any leading white space characters, the `readbits` function consumes all 0 and 1 characters. It builds the bit pattern by moving the existing value one place left via the `<<` shift operation and then inserting a 1 if needed.

The `readunsignedshort()` function is a variation of that used earlier. This version takes an additional “Style” argument that specifies whether data are to be entered in octal or hexadecimal.

```

static unsigned short readunsignedshort(const char* prompt, Style s) {
    // Prompts for and reads an unsigned short value
    // Keeps prompting until it gets a valid input
    for (;;) {
        printf("%s", prompt);
        unsigned short enteredval;
        int transferred;
        if (s == OCTAL) transferred = scanf("%ho", &enteredval);
        else transferred = scanf("%hx", &enteredval);
        if (transferred != 1) {
            if (feof(stdin)) {
                printf("Input terminated, program will exit\n");
                exit(0);
            }
            printf("invalid input\n");
            // The following funny format string says
            // read all characters up to newline and simply ignore them
            // then read the newline character as a character but discard it
            // as well.
            // Serves to clear junk from the input line.
            scanf("%*[^\\n]*%c");
            continue;
        }
        return enteredval;
    }
}

```

Get the program to work, and test yourself on your ability to convert between binary, hex, and octal representations.

Exercises complete

Show the tutor that you have completed all the parts and that you can use the debugger to stop a program part way through, single step execute a program, and inspect variable values.

(That was 7 very easy marks wasn't it – just copy the code. Hopefully, you learnt a little about how to use an IDE to increase your productivity as a programmer, and about how to exploit gdb to help resolve the inevitable bugs.)

Assignment

The [assignment](#) requires that you implement a more sophisticated version of the program that will test a user on their ability to manipulate bit data and use binary, octal, and hexadecimal notations.