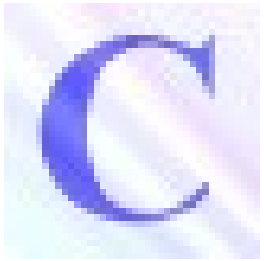
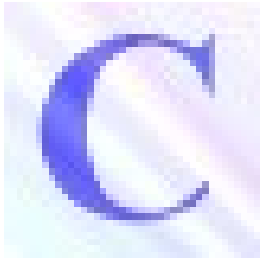


Computer Programming

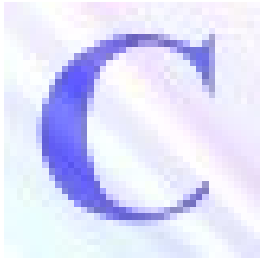
”The greatest gift you can give another is the purity of your attention”.

Richard Moss



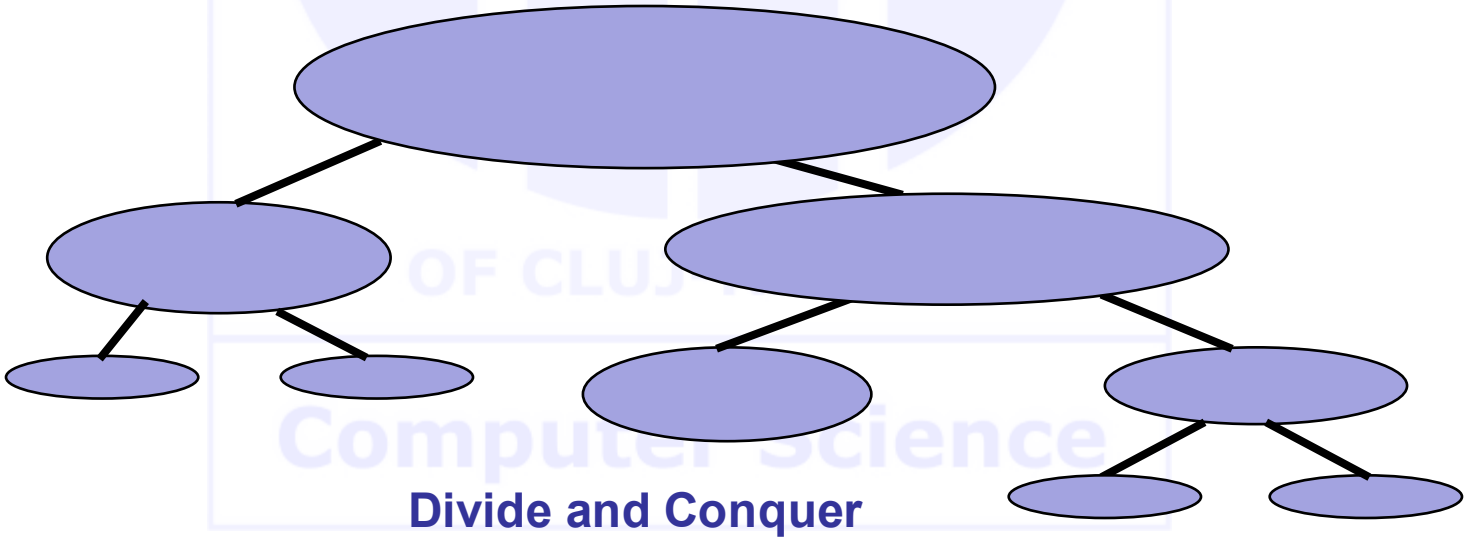
Outline

- Modular programming
  - Modularity
  - Header file
  - Code file
- Debugging
  - Hints
  - Examples

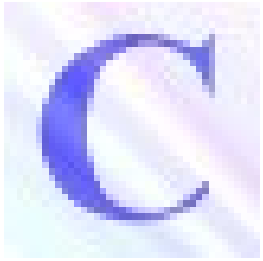


Modularity

- How do you solve a big/complex problem?
- Divide it into small tasks and solve each task. Then combine these solutions.

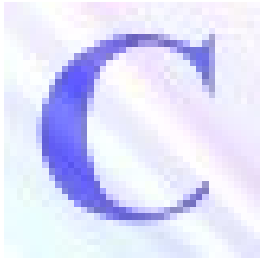


Divide and Conquer



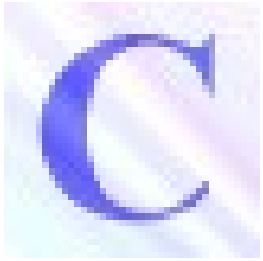
Advantages of using modules

- Modules can be written and tested separately
- Modules can be reused
- Large projects can be developed in parallel
- Reduces length of program, making it more readable
- Promotes the concept of **abstraction**
  - A module hides details of a task
  - We just need to know what this module does
  - We don't need to know how it does it



Module

- **Module**: a collection of functions that perform related tasks.
  - Example:
    - a module could exist to handle database functions such as lookup, enter, and sort.
    - Another module could handle complex numbers, and so on.

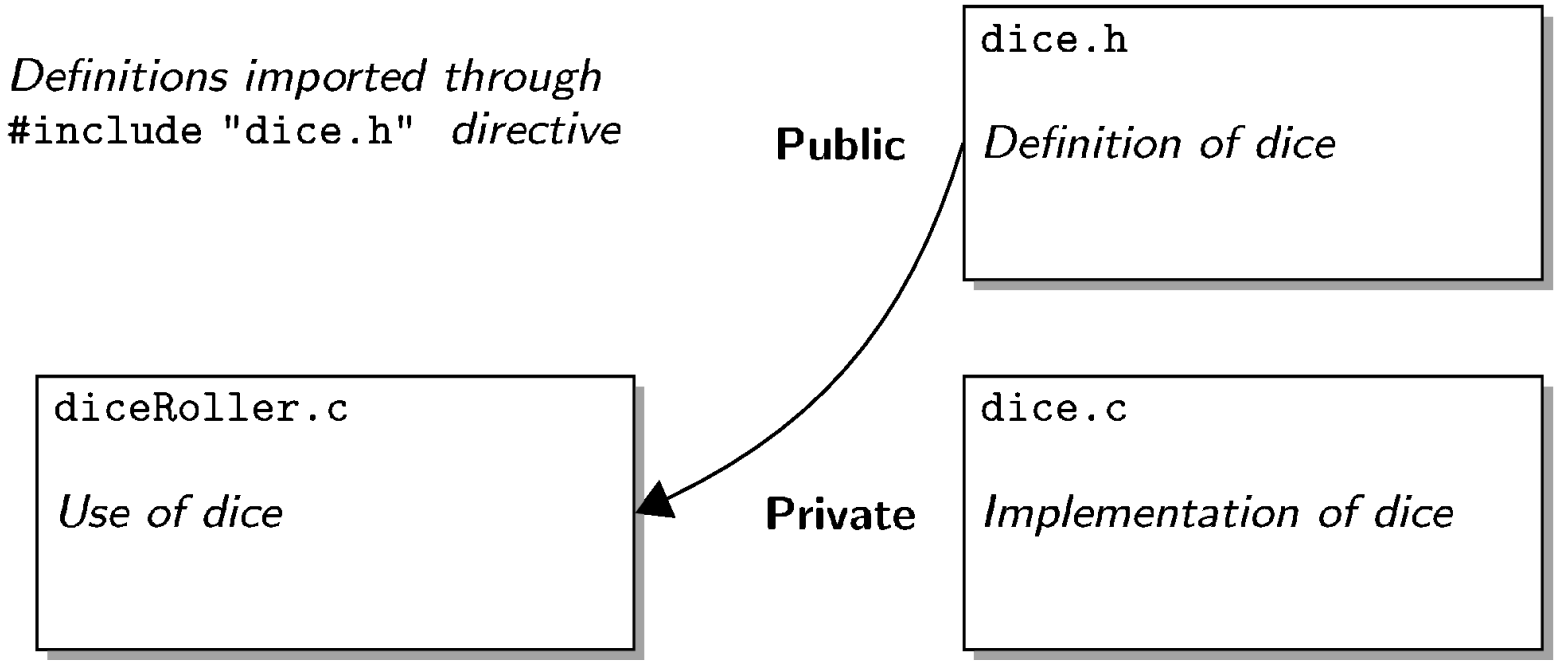


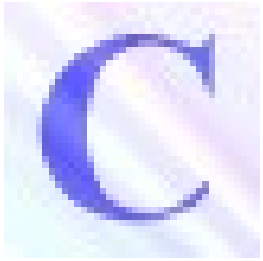
Modules

- Modules are divided into two parts: *public* and *private*.
  - *Public* part:
    - Tells the user how to call the functions in the module.
    - Contains the definition of data structures and functions that are to be used outside the module.
      - These definitions are put in a header file, and the file must be included in any program that depends on that module.

Modules

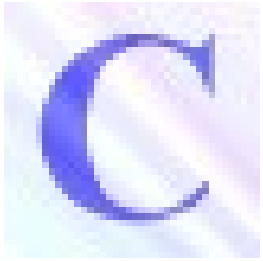
- *Private* part:
  - Anything that is internal to the module is private.
  - Everything that is not directly usable by the outside world should be kept private.





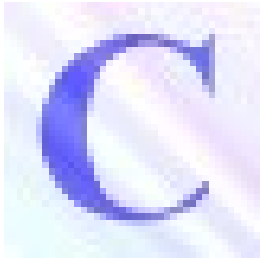
Modular Programming in C

- Means *splitting every source code* into:
  - a header file, e.g. module1.h and
  - a corresponding code file, e.g. module1.c .
- The header contains only what client program are allowed to see and to use:
  - declarations of constants, types, global variables and
  - function prototypes
  - Every other private item internal to the module must stay inside the code file.



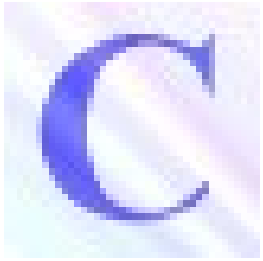
Header File

- Every header file should start with a brief description (as C comments) of:
  - its purpose,
  - author,
  - copyright information,
  - version and
  - how to check for further updates.
  - Proper C declarations must be enclosed between C preprocessor directives that prevent the same declarations from being parsed twice in the same compilation run



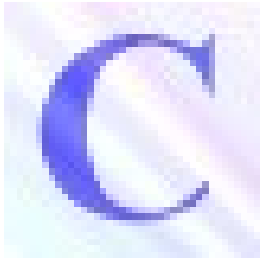
Header File. Skeleton example

```
/*
module1.h -- Skeleton example of a header file for a module
Copyleft 2013 by Marius Joldos
License: as you wish
Author: Marius Joldos <Marius.Joldos@cs.utcluj.ro>
Version: 2013-10-26
Updates: users.utcluj.ro/~jim/CP/modules.html
*/
#ifndef _MODULE1_H_
#define _MODULE1_H_
/* Include headers required by following declarations: */
#include <stdlib.h>
#include <math.h>
#include "otherHeader.h"
/* Constants declarations here. */
/* Types declarations here. */
/* Global variables declarations here. */
/* Function prototypes here. */
#endif
```



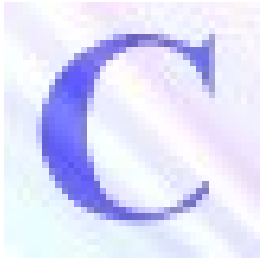
Code file

- The code module should
  - include the required headers, then
  - include its own header file.
    - Including its own header, the compiler grabs all the constants, types and variables it requires.
      - Including its own header file, the code file allocates and initialize the global variables declared in the header.
      - Another useful effect of including the header is that prototypes are checked against the actual functions, so that for example if you forgot some argument in the prototype, or if you changed the code missing to update the header, then the compiler will detect the mismatch with a proper error message.



Code file

- Macros, constants and types declared inside a code file cannot be exported, as them are implicitly always "private".
- **static** keyword
  - Tells to the compiler that these functions are not available for linking, and then them will not be visible anymore once the code file has been compiled in its own object file.
  - Global variables for *internal* use must have the *static* keyword in order to make them "private".
  - Declare as static all the functions that are private to the code module.
  - Private items are still available to the debugger, anyway.

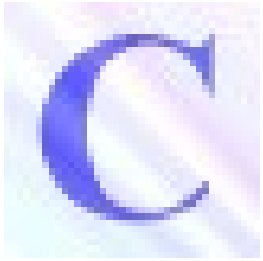


Code file

```
/* module1.c -- See module1.h for copyright and info */
#include <malloc.h>
#include <string.h>
/* Including my own header: */
#include "module1.h"
/* Private macros and constants: */
/* Private types: */
/* Private global variables: */
static module1_node * spare_nodes = NULL;
static int allocated_total_size = 0;
/* Private functions: */
static int *example1_opaque(void){ ... }
static void example2_opaque(module1_opaque * p){ ... }
/* Public functions: */
void module1_init(void){ ... }
module1_node * module1_add(char * key){ ... }
void module1_free(void){ ... }
```

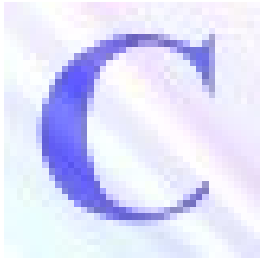
Main Program

- Main program
  - The name of our project will be programName and its source file is programName.c (or main.c) .
  - This source is the only one that does not require an header file, and it contains the only public function, main() , that does not have a prototype.
  - The main source includes and initializes all the required modules, and finally terminates them once the program is finished.



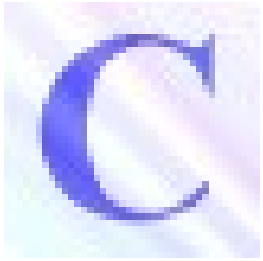
Main Program

```
/*
program_name.c -- Our sample program
Copyleft 2013 by Marius Joldos
License: as you wish
Author: Marius Joldos <Marius.Joldos@cs.utcluj.ro>
Version: 2013-10-26
Updates: users.utcluj.ro/~jim/CP/modules.html
*/
/* Include standard headers: */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```



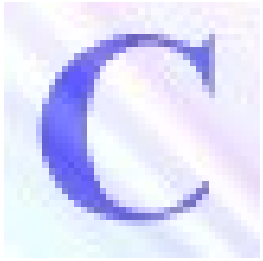
Main Program

```
/* Include our modules headers: */
#include "module1.h"
#include "module2.h"
#include "module3.h"
int main(int argc, char *argv[])
{
    /* Initialize modules: */
    module1_init();
    module2_init();
    /* Perform our job. */
    /* Properly terminate the modules, if required: */
    module2_free();
    module1_free();
    return 0;
}
```



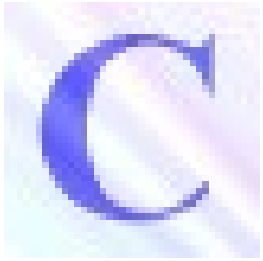
Example

- Demo and discussion of example



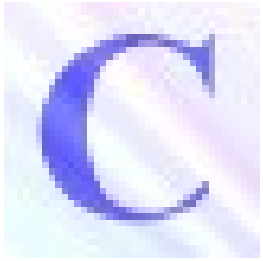
Debugging. What is it

- Debugging is not getting the compiler to compile your code without syntax errors or linking errors.
  - Although this can sometimes take some time when a language is unfamiliar.
- Debugging is what you do when your code compiles, but it doesn't run the way you expected.
  - "Why doesn't it work? What do I do now?"



Debugging. Hints

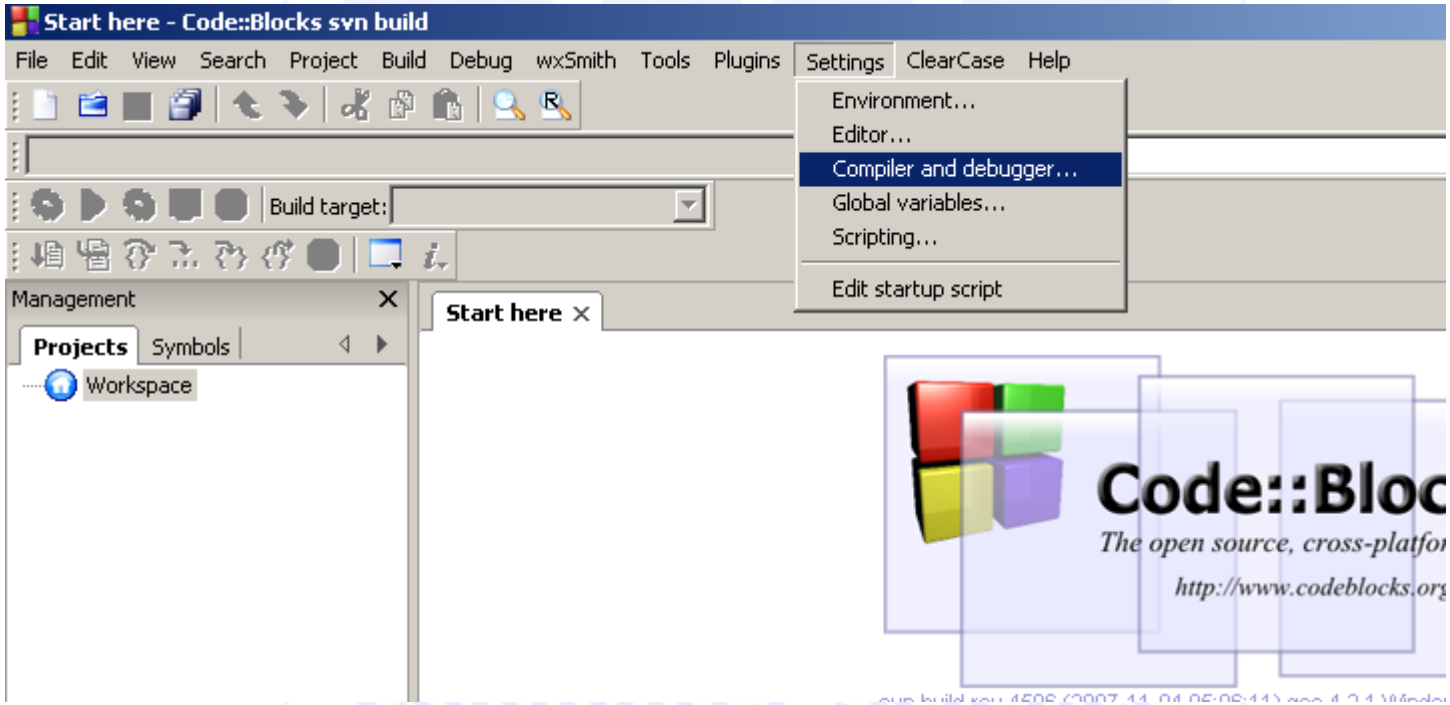
- To solve a problem, you must understand it.
- Make life easier for yourself:
  - Indent your code properly (observe style rules).
  - Use meaningful variable names.
  - Print debugging to stderr or stdout, but not both, or your debugging output may be reordered.
  - Develop incrementally. Test as you go.



Debugging. Make compiler help you

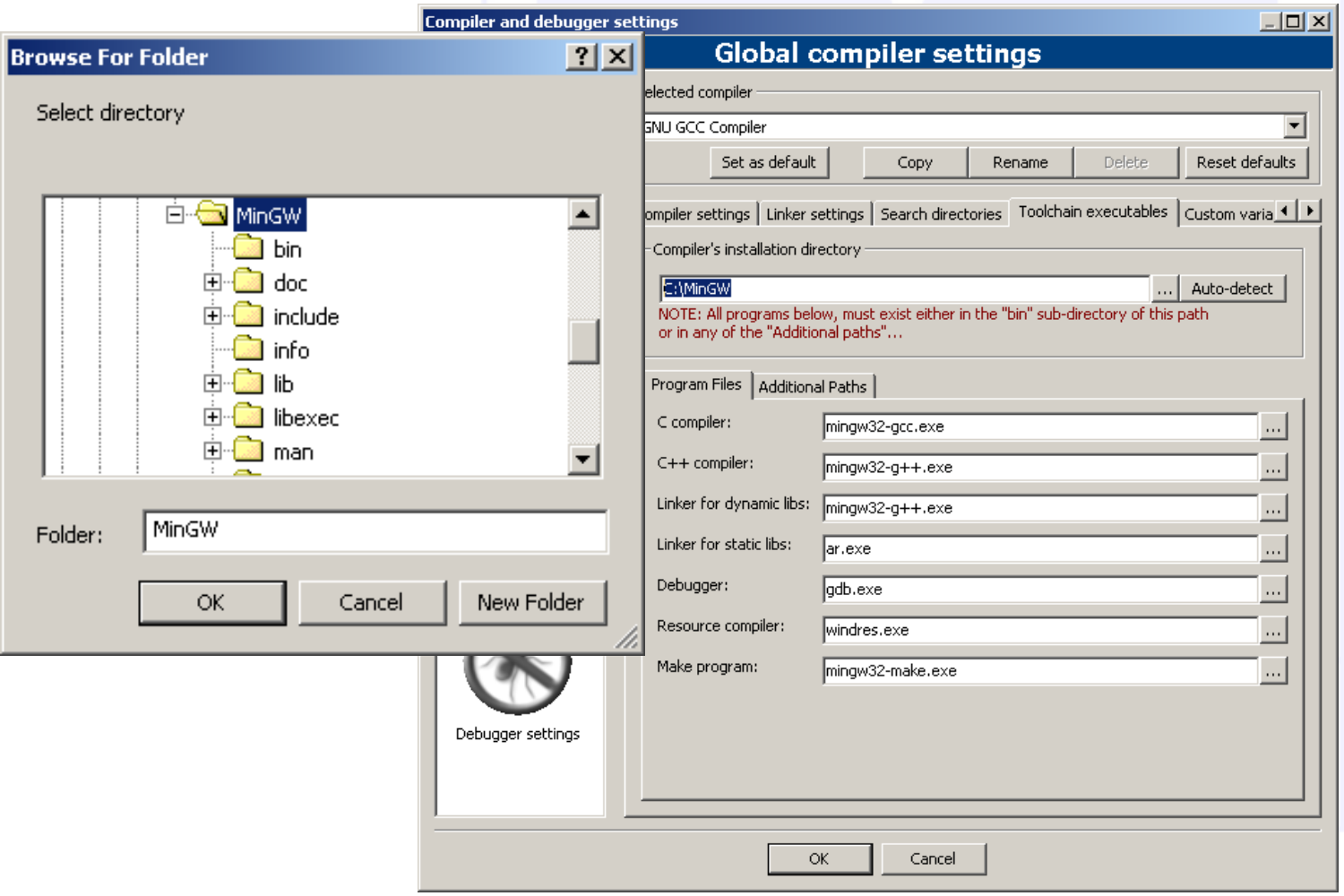
- The C compiler isn't as pedantic as a Java compiler, but it can still help you out.
- Use the compiler flags. With gcc:
  - -g includes debugging systems
  - -Wall turns on (almost) all compiler warnings.
- Use the help system. Use Web resources
  - on Dev-Cpp, CodeBlocks adjust help menu
  - Web links

Debugging. CodeBlocks. Make compiler help you

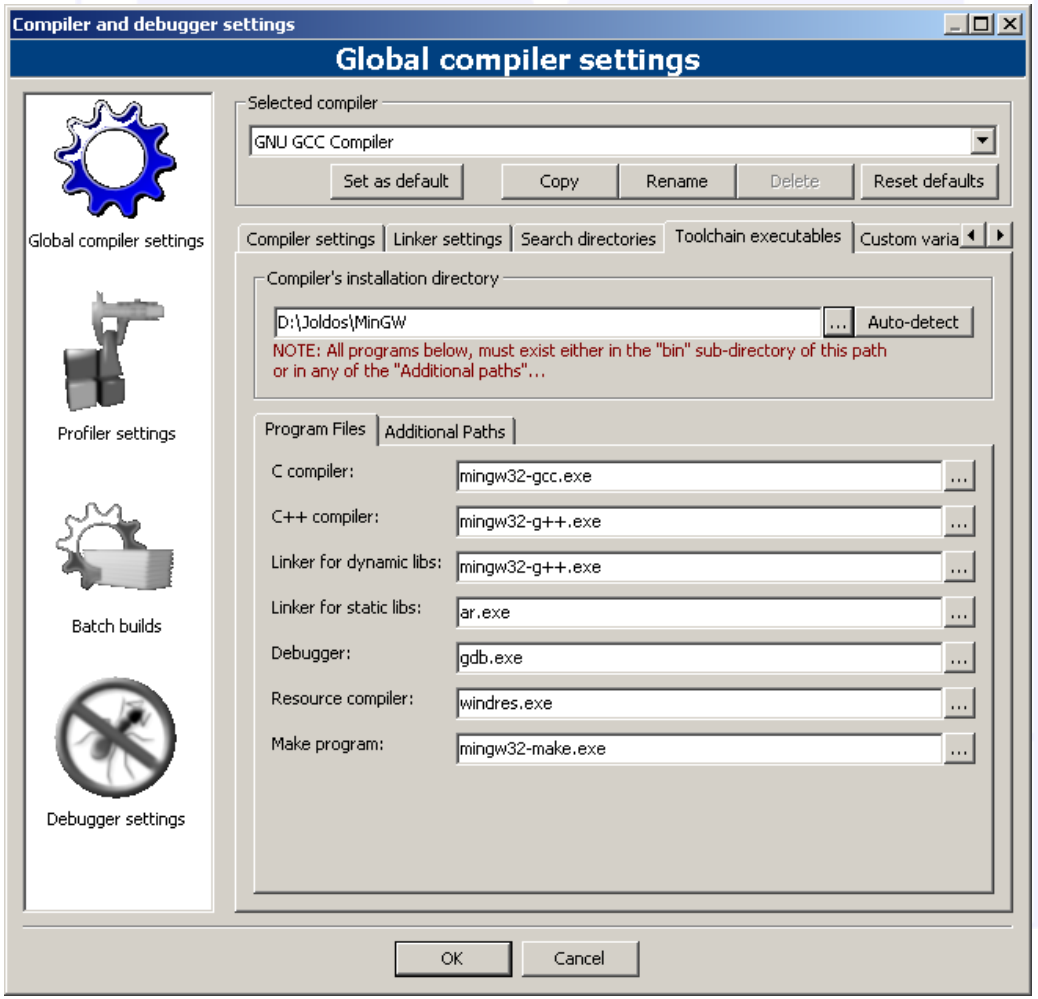


Debugging. CodeBlocks. Make compiler help you

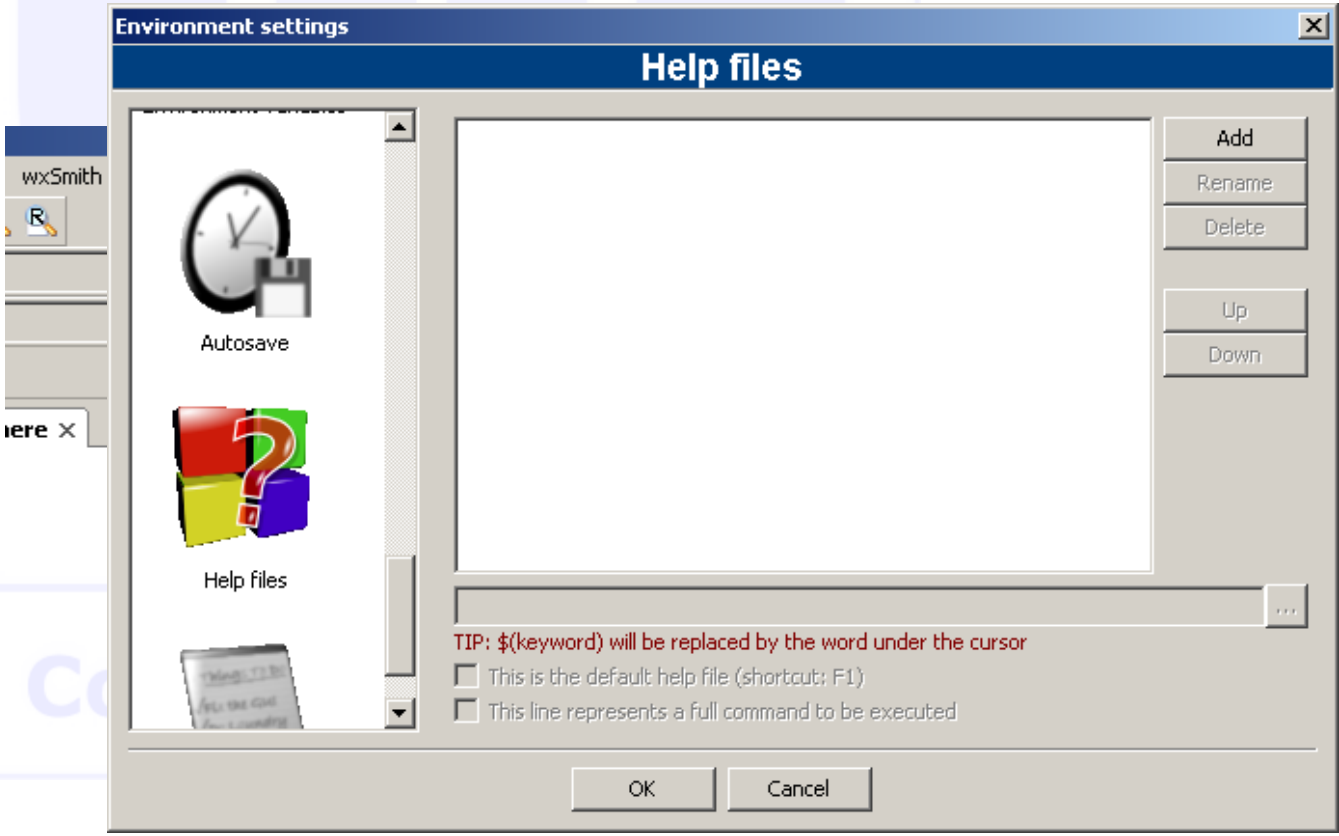
Debugging. CodeBlocks. Make compiler help you



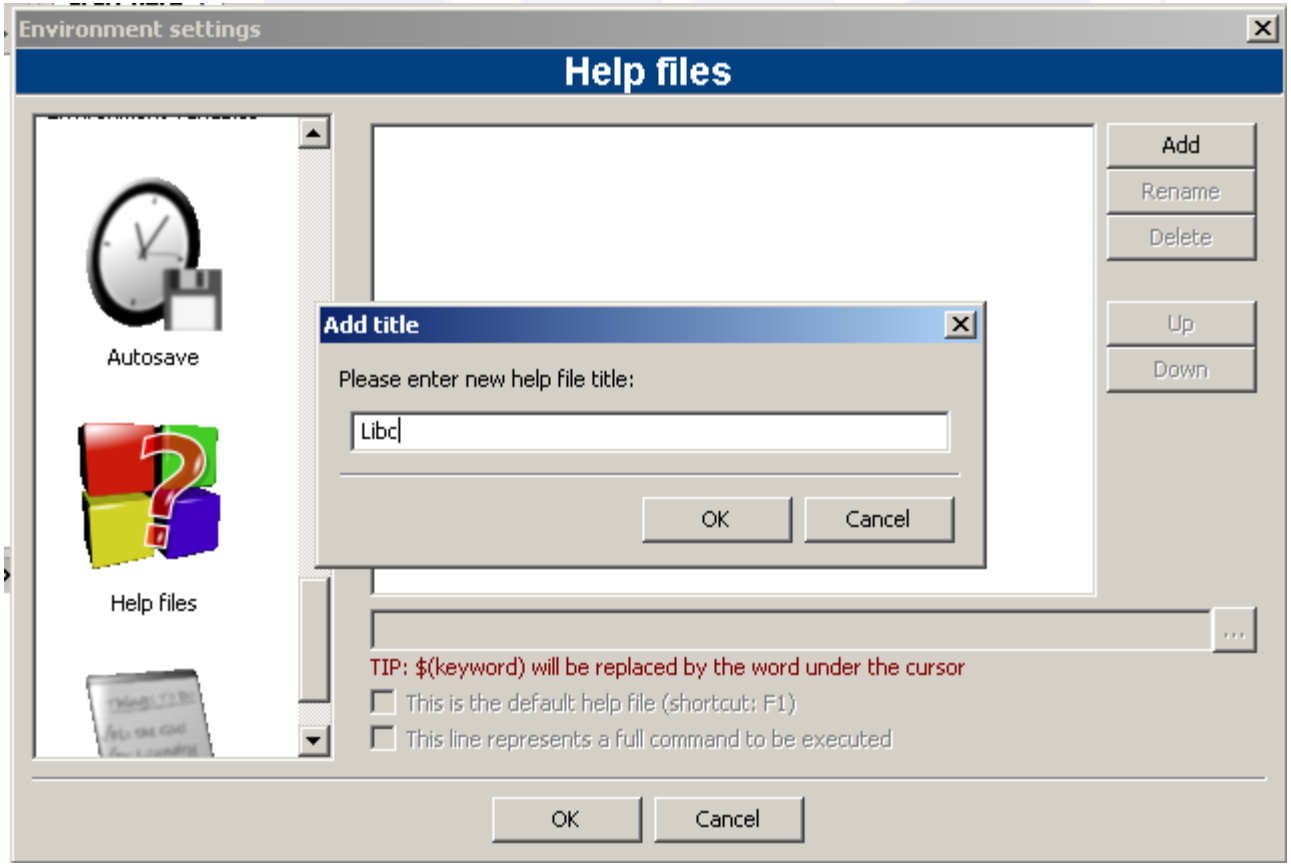
Debugging. CodeBlocks. Make compiler help you



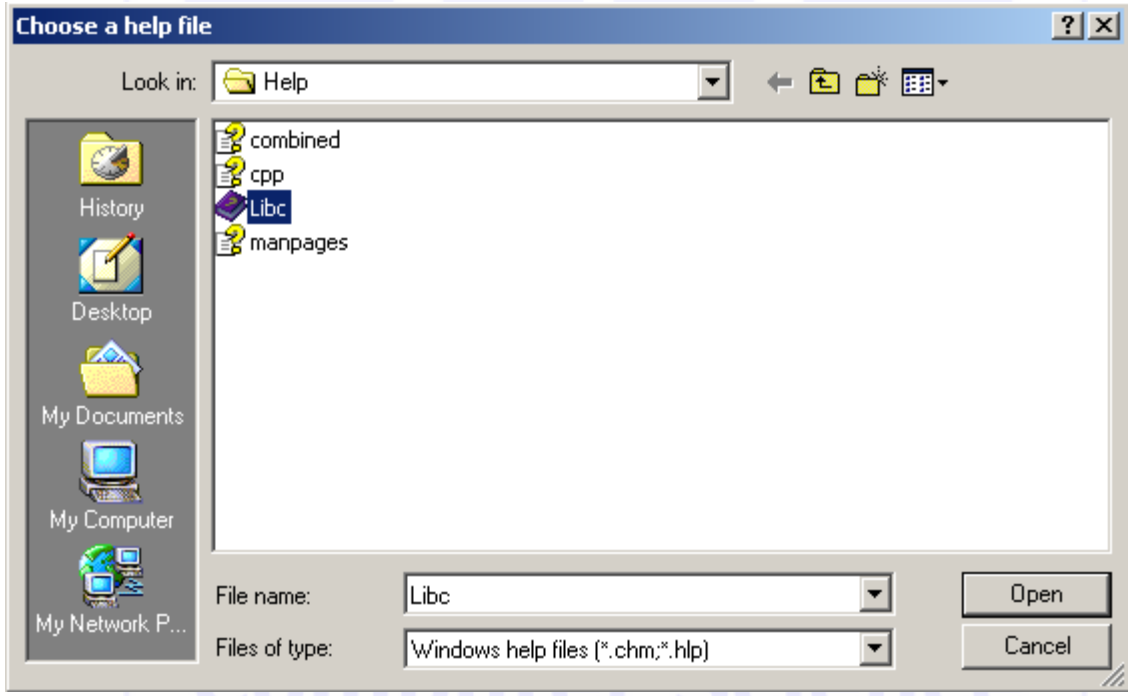
Debugging. CodeBlocks. Make IDE help you



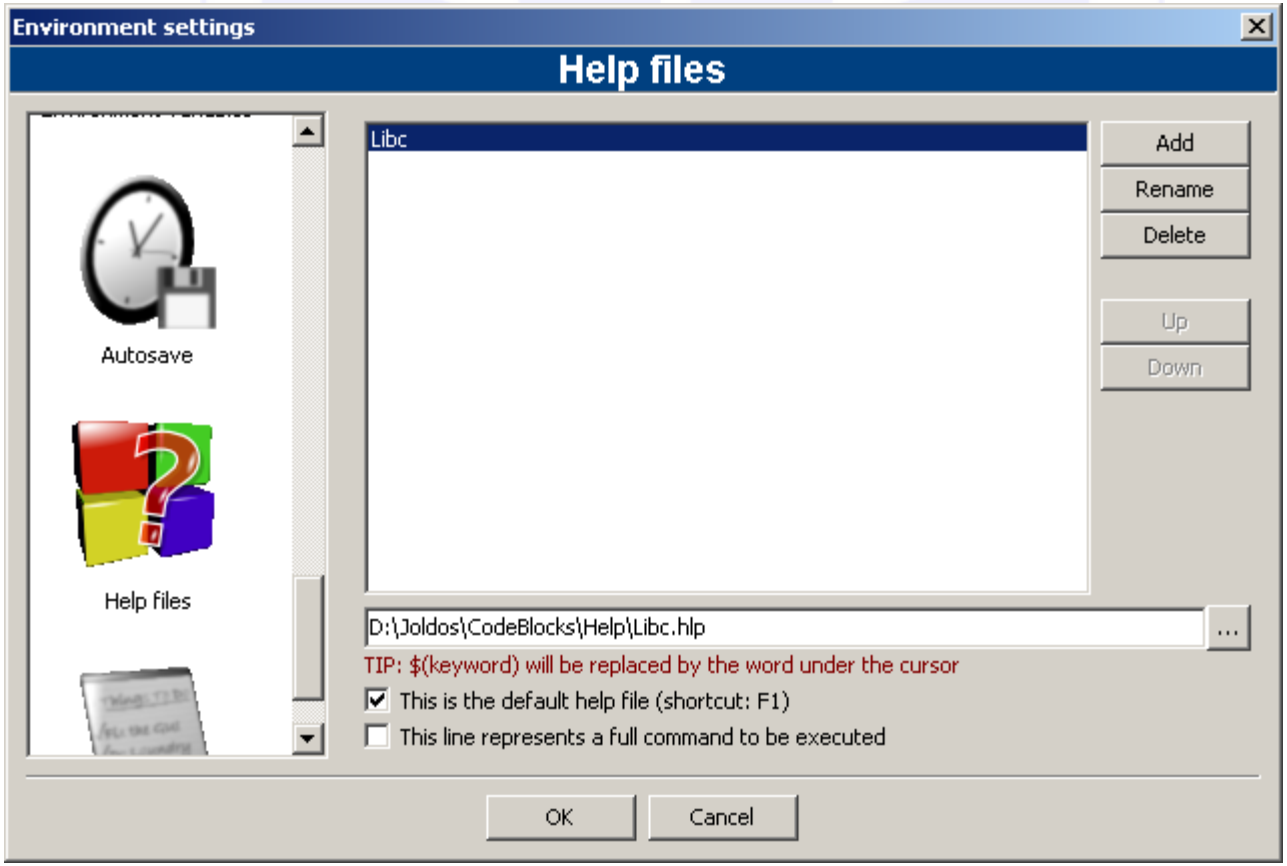
Debugging. CodeBlocks. Make IDE help you

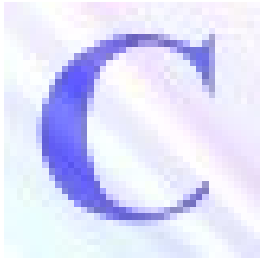


Debugging. CodeBlocks. Make IDE help you



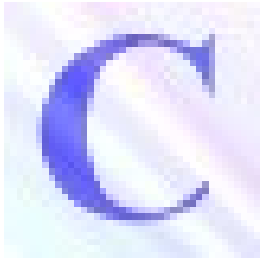
Debugging. CodeBlocks. Make IDE help you





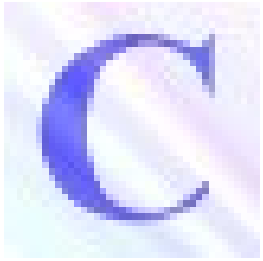
Useful Web links

- The C book:
  - http://publications.gbdirect.co.uk/c_book/
- C programming
  - http://www.eskimo.com/~scs/cclass/cclass.html
- How to debug programs
  - http://www.drpaulcarter.com/cs/debug.php
- Common C programming errors
  - http://www.drpaulcarter.com/cs/common-c-errors.php



Debugging. Hints

- When it doesn't work, pause, read the output very carefully, read your code, and think first.
  - It is way too easy to start hacking, without reviewing the evidence that's in front of your nose.
  - If there is not enough evidence to find and understand a problem, you need to gather more evidence.
  - This is the key to debugging.

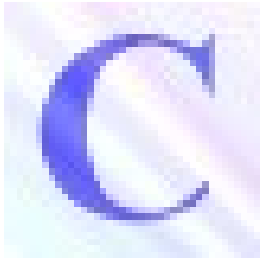


Debugging. Gathering evidence

- Two ways to fail:
  1. Program did something different from what you expected.
  2. Program crashed. In C, likely symptoms of a crash:

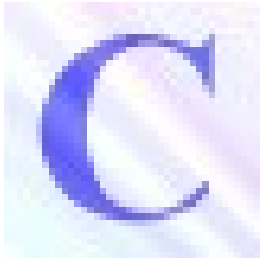
```
Segmentation fault: core dumped.
Bus error: core dumped.
Floating point exception: core dumped.
```

  - We'll deal with crashes a little later.



Debugging. Gathering evidence

- Program did something different from what you expected.
  - Task 1: figure out what the program did do.
  - Task 2: figure out how to make it do what you wanted it to do.
- Don’t jump in to task 2 before doing task 1. You’ll probably break something else, and still not fix the bug.
  - And there may be similar bugs elsewhere in your code because you made the same error more than once.
- Don’t jump into the debugger *first* - it stops you reading your code. A debugger is only one debugging tool.
  - Occasionally code behaves differently in a debugger.

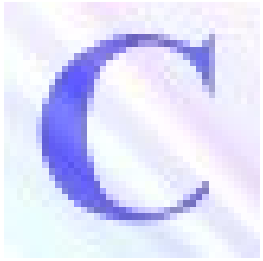


Debugging. Gathering evidence

- Did the program take the path through your code that you expected? How do you know?
  - Annotate the expected code path.

```
printf("here 7\n");
printf("entering factorial()\n");
printf("leaving factorial()\n");
```

- It's very common for your program to take a different path.
- Often when you see this, the bug will be obvious.



Debugging. Gathering evidence

- Print the values of key variables. What are you looking for?
  - Did you initialize them?
  - C won't usually notice if you didn't.
  - Were type conversions done correctly?
  - Did you pass the parameters in the wrong order?
  - Print out the parameters as they were received in the function.
  - Did you pass the right number of parameters?
  - C won't always notice.
  - Did some memory get overwritten?
  - A variable *spontaneously* changes value!

Common C Errors

- Missing break in a switch statement
- Using = instead of ==

```
if (a = 0) { ..... }
```

- Spurious semicolon:

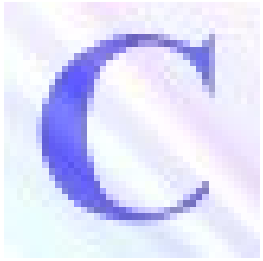
```
while (x < 10);
  x++;
```

- Missing parameters:

```
printf("The value of a is %d\n");
```

- Wrong parameter type in **printf, scanf**, etc:

```
double num;
printf("The value of n is %d\n", num);
```



Common C Errors

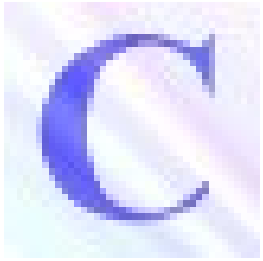
- Array indexing:

```
int a[10]; has indices from 0 to 9
```

- Comparing Strings:

```
char s1[] = "test";
char *s2 = "test";
if (s1 == s2)
  printf("Equal\n");
```

  - You must use `strcmp()` or `strncmp()` to compare strings.



Common C Errors

- Integer division:

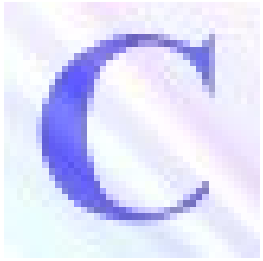
```
double half = 1/2;
```

  - This sets half to 0 not 0.5!
  - 1 and 2 are *integer* constants.
  - At least one needs to be floating point:

```
double half = 1.0/2;
```

  - Or cast one to floating point:

```
int a = 1, b = 2;
double half = ((double)1)/2.
```



Common C Errors

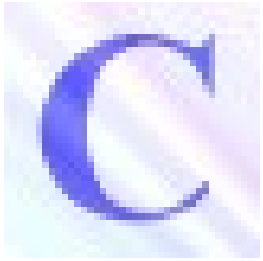
- Missing headers or prototypes:

```
double x = sqrt(2);
```

  - `sqrt` is a standard function defined in the maths library.
  - It won't work properly if you forget to include `math.h` which defines the function prototype:

```
double sqrt(double)
```

- C assumes a function returns an `int` if it doesn't know better.
  - Link with –lm may be necessary

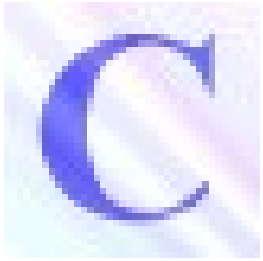


Common C Errors

Spurious pointers:

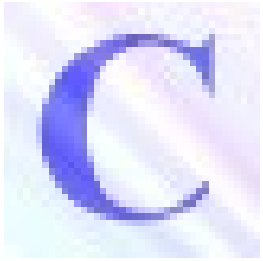
```
char *buffer;
fgets(buffer, 80, stdin);
```

- With pointers, always ask yourself :
  - "*Which memory is this pointer pointing to*?".
- If it's not pointing anywhere, then assign it the value NULL (0)
- If your code allows a pointer to be NULL, you *must* check for this before following the pointer.



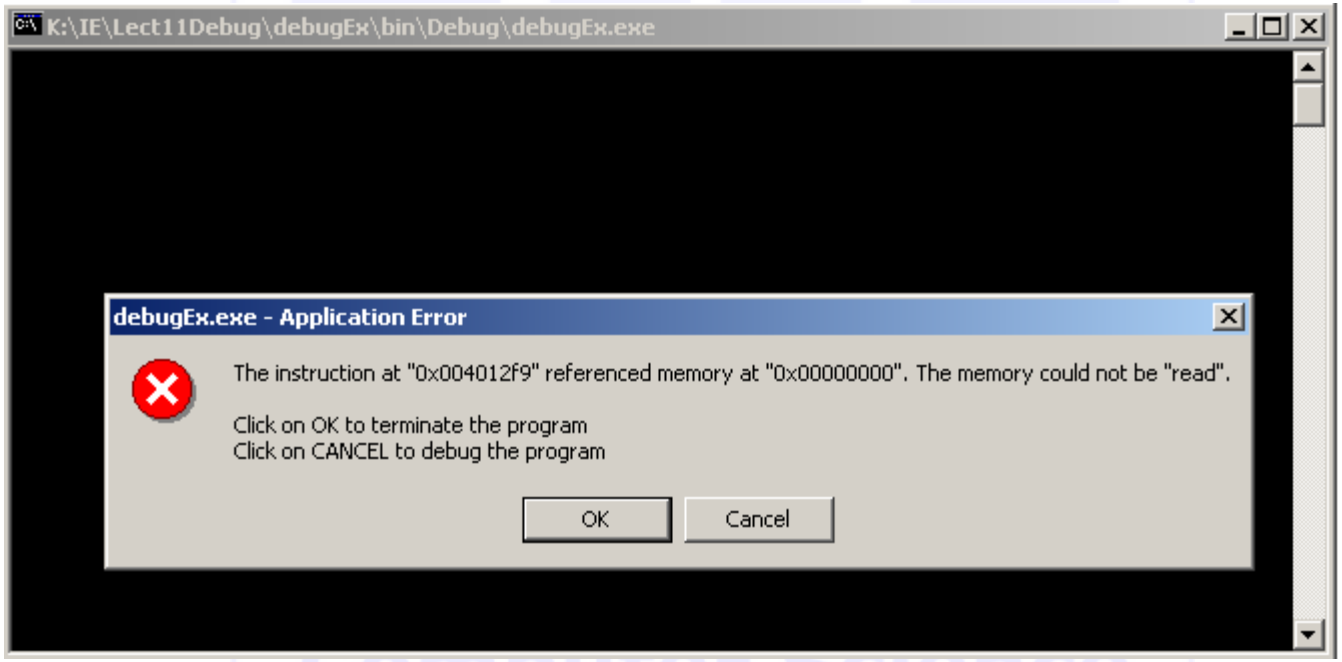
Crashes. Core files

- Segmentation fault: core dumped.
  - The program tried to access memory that did not belong to it.
  - The error was detected by the OS.
- Bus error: core dumped.
  - The program tried to access memory that did not belong to it.
  - The error was detected by the CPU.
- Floating point exception: core dumped.
  - The CPU's floating point unit trapped an error.

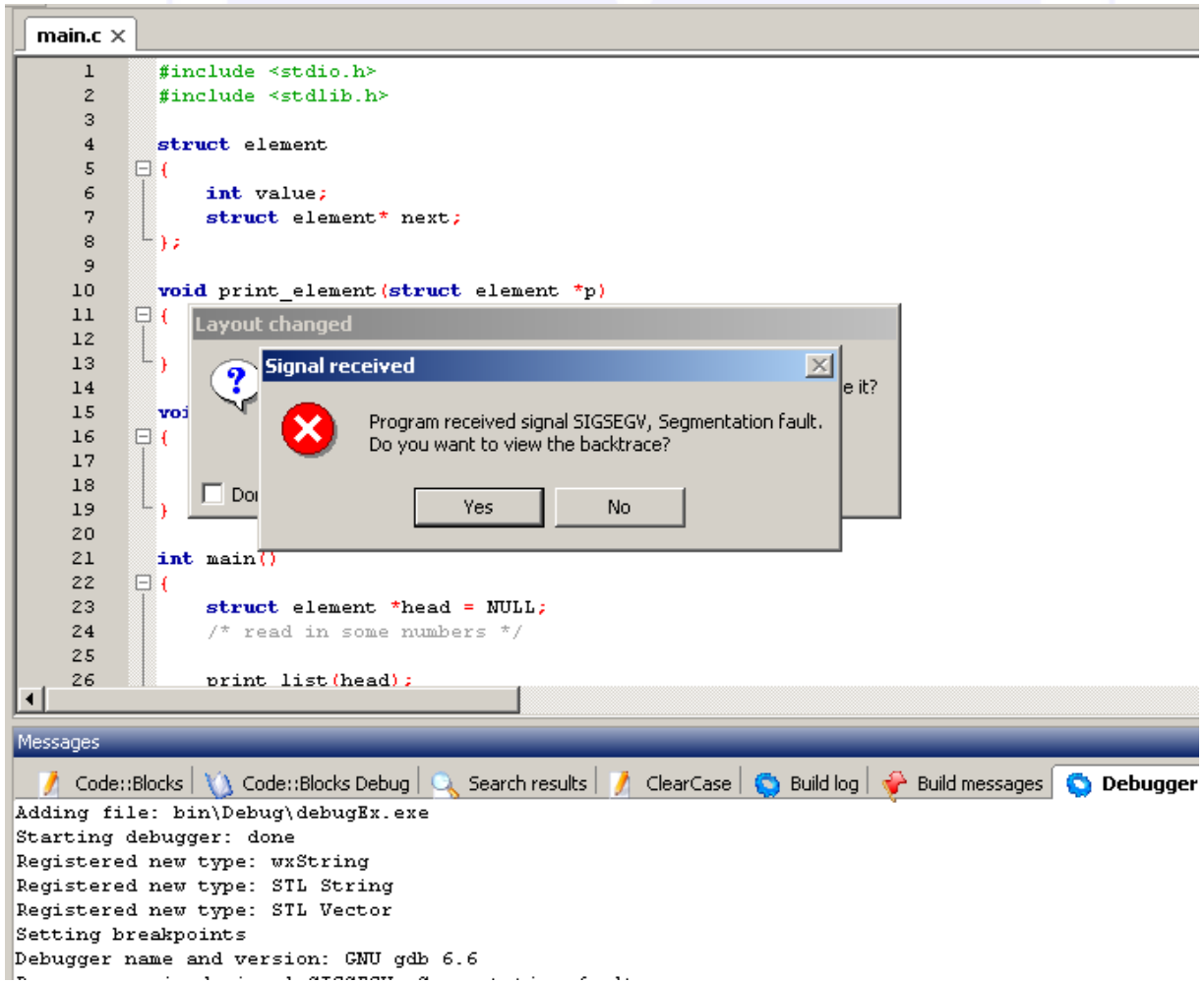


CodeBlocks

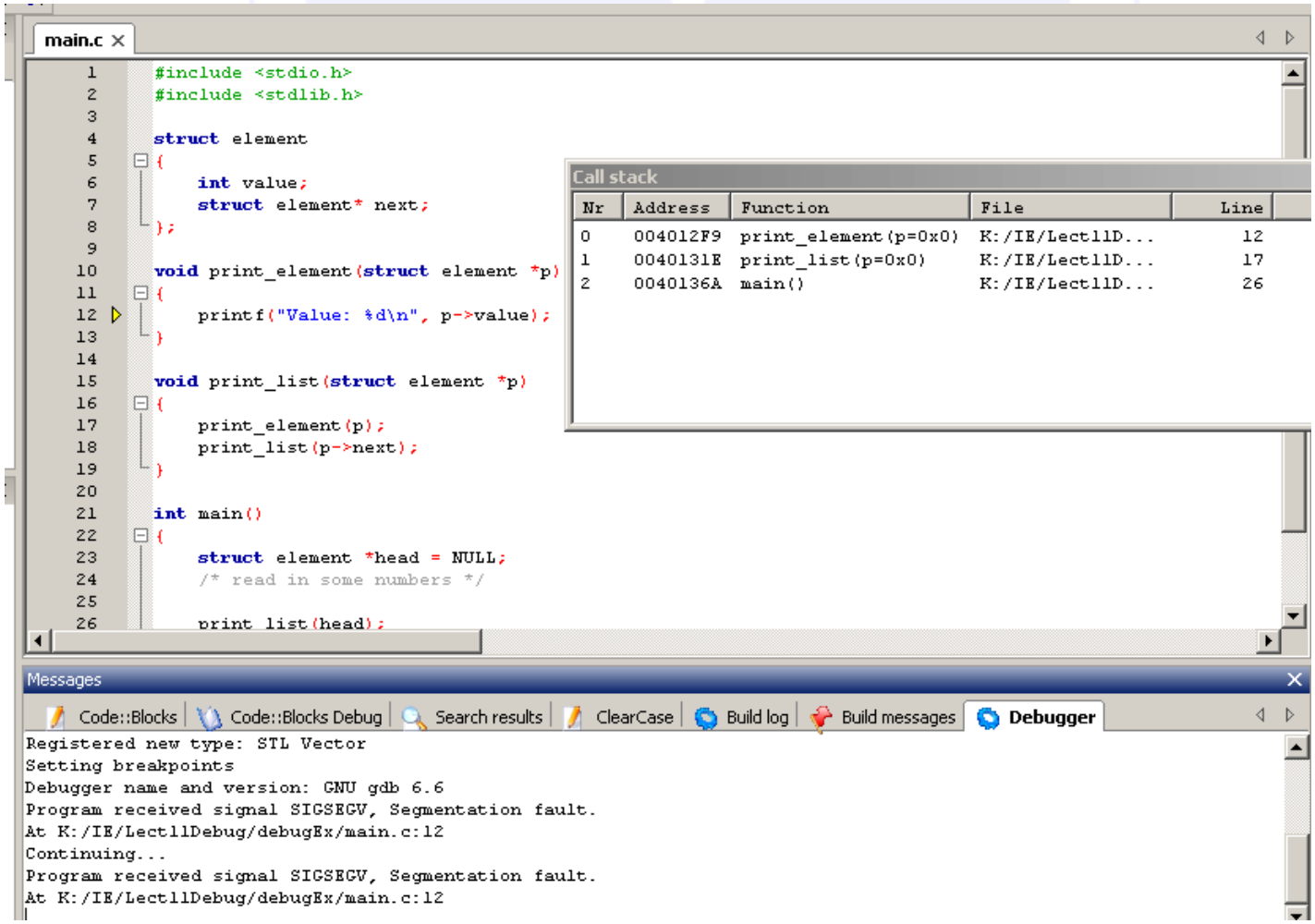
- Running program ends with execution error.

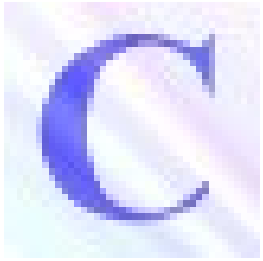


CodeBlocks. Run program under debugger



CodeBlocks. Run program under debugger



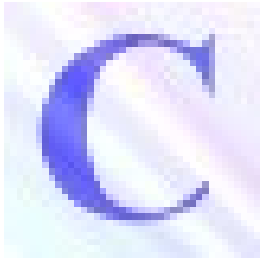


Debugging Tips (Philip Guo)

- Use print statements (with endlines) to hone in on a problem, then switch into a debugger
- Write complex conditional breakpoint conditions in your source code. E.g.

```
void foo(int a, int b, int c) {
  ...
  if (((a % 3) == 0) || ((b < (c * 2)) && ((c + a) < 0))) {
    printf("BREAK!!!");
  }
  ...
}
```

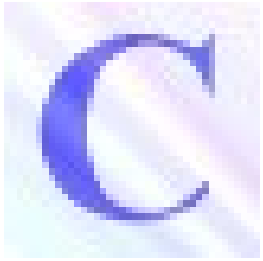
- Debug memory corruption errors using watchpoints
  - Memory corruption can occur when some data structure in your program retains a pointer to a region of memory that is freed without its knowledge (via an aliased pointer), and then some other part of your program re-allocates that memory to be used for another purpose



What is wrong here?

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int numbers[10];
int i;
  for (i=1;i<=10;i++)
     numbers[i]=i;
  for (i=1;i<=10;i++)
     printf("numbers[%d]=%d\n",
 i, numbers[i]);
}
```

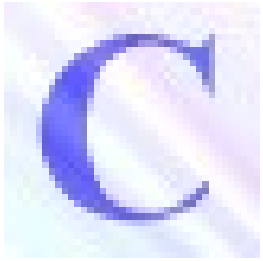
```
#include <stdio.h>
#include <stdlib.h>
main()
{
int i;
for (i=0; i<10; i=i+1)
  if (i=2)
    printf("i is 2\n");
  else
    printf("i is not 2\n");
}
```



What is wrong?

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int i;
for (i=0; i<10; i=i+1)
  if (i<2)
     printf("%d is less than
  2\n",i);
     printf("and %d is not
  equal to, 2 either\n",i);
}
```

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

main()
{
int i;
 i = 0;
 while (i < 10);
   i = i + 1;
 printf("Finished. i =
  %d\n",i);
}
```



What is wrong?

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int i;
for (i=0; i<10; i=i+1)
    switch(i){

    case 0: printf("i is 0\n");
    case 1: printf("i is 1\n");
    default: printf("i is more
   than 1\n");
    }
}
```

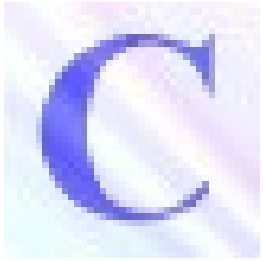
```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i = 0;
   char buff[128];
   char *arg1 = argv[1];
   while (arg1[i] != '\0' )
   {
     buff[i] = arg1[i];
     i++;
   }
   buff[i] = '\0';
   printf("buff = %s\n", buff);
}
```

Common Programming Errors in the C Language (Microsoft KB22321)

- Using an automatic variable that has not been initialized
- Omitting a closing comment delimiter
- Using an array index greater than the length of the array (In C, array indexes run from zero to <length>-1.)
- Omitting a semicolon or a closing brace
- Using an uninitialized pointer
- Using a forward slash when a backslash is required (for example, substituting "/n" for "\n.")
- Using "=" in a comparison where "==" is desired
- Overwriting the null terminator for a string
- Prematurely terminating a function declaration with a semicolon (The compiler often flags the "orphan" block of code as a syntax error.)
- Specifying the values of variables in a scanf() statement instead of their addresses
- Failing to declare the return type for a function
- Assuming an expression evaluation order when using an expression with side effects (For example, a[i] = i++; is ambiguous and dangerous.)
- Failing to account that a static variable in a function is initialized only once
- Omitting a "break" from a case in a switch statement (Execution "falls through" to subsequent cases.)
- Using "break" to exit a block of code associated with an if statement (The break statement exits a block of code associated with a for, switch, or while statement.)
- Comparing a "char" variable against EOF (-1).

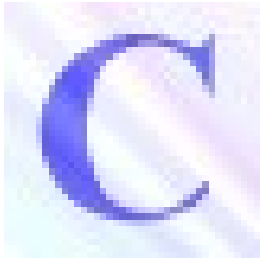
Questions to answer (M. R. Murphy)

- Is the argument count correct?
- Is the spelling correct?
- Is the argument type correct?
- Is the variable initialized correctly?
- Do "ends" match "do"s?
- Is the correct register used?
- Is a previous equate, using, or define in effect?
- Is the level of indirectness correct?
- Are the defaults correct?
- Is the array large enough to hold the data?
- Is the index out of range?
- Is reentrancy, reusability, refreshability preserved?
- Did you let something go by that you can't explain? Fix it now.
- Does the program have anything remotely to do with the requirement for the program?
- Did you start to code before thinking?
- Did you read the manual?



Advice from Andy Oram and Greg Wilson "Beautiful Code"

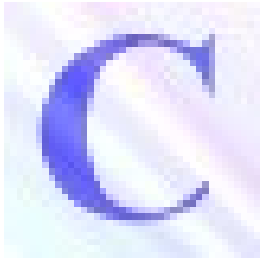
- The general process for finding causes is called the **scientific method**. Applied to program failures, it works as follows:
  - Observe a program failure.
  - Invent a **hypothesis** for the failure cause that is consistent with the observations.
  - Use the hypothesis to make **predictions**.
  - Put your predictions to the test by experiments and further observations:
    - If experiment and observation satisfy the prediction, refine the hypothesis.
    - If not, find an alternate hypothesis.
  - Repeat steps 3 and 4 until the hypothesis can no longer be refined.



Info sources (web)

- C Programming Techniques

- Debugging with Code::Blocks - CodeBlocks
- Debugging in Code::Blocks – YouTube
- How to use **Debug in Code::Blocks** - YouTube



Summary

- Modular programming
  - Modularity
  - Header file
  - Code file
- Debugging
  - Hints
  - Examples