C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 1

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- The History of C
- Basics of C Program Writing
- Creating an Executable Program
- Coding Style
- References

`http://www.physics.gla.ac.uk/~kaiser/`

Programming Languages

Computers operate on *binary numbers*, both the data and the program are that is executed are a long series of `1` and `0`. The program in this case is called machine code

```
1001 1100
0011 1011
1100 1111 ... and so on ...
```

Unlike computers, people don't think in numbers. Therefore *assembly language* was developed, and at first used by programmers to write programs that were then translated by hand:

```
MOV A,47  1010 1111
ADD A,B   0011 0111
HALT      0111 0110
```

As the cost of computers went down and that of programmers went up, programs (*assemblers*) took over the translation.

Programming Languages

- The next step were *high level* languages, that are even easier to understand for humans, leaving more translation work for the computer. For this translation the computer uses a program called a *compiler*.
- Examples of high level languages are
  - FORTRAN FORMula TRANslation
  - Pascal
  - COBOL (still used in some banks)
  - Java
- C is also a high level programming language. Like most programming languages it is based on English (thankfully).

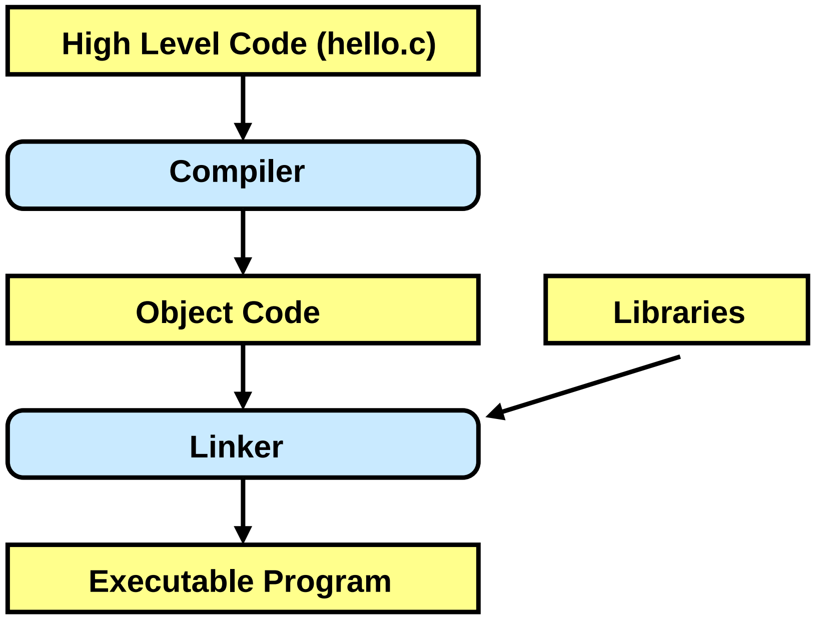
A Short History of C

- BCPL (Basic Computer Programming Language), Martin Richards, 1967
- B, Bell Labs, Ken Thompson, 1970
- C, Bell Labs, Dennis Ritchie, 1970+
- *The C Programming Language*, B.Kernighan/D.Ritchie, 1978
- C++, Bjarne Stroustrup, 1980
- ANSI C, American National Standards Institute, 1989
- ISO/IEC 9899 C, International Organisation for Standardization, 1999, the current Standard C
- C#, Anders Hejlsberg, Microsoft, 2000

Which C are we using ?

- There is a large number of different C compilers for different operating systems. Many of them today are C++ compilers that also compile C code: Borland C++, Microsoft Visual C++, Turbo C++.
- We will use gcc, the Gnu C Compiler, open source software from the Free Software Foundation. gcc supports Standard C, but contains some extended features (that can be disabled). Details at `http://gcc.gnu.org/`.
- C code is portable, i.e. the source code (as long as it complies with Standard C) can be compiled by any compiler on any platform. The compilers are platform-dependent.
- Unix, including Linux, has a special connection with C, because Unix was written in C.

From High-Level Code to Executable Program



- source code written by programmer in high-level language, in our case in C
- compiler translates it into an object file
- linker combines object code with predefined routines from libraries and produces the executable program
- This is usually not done ‘by hand’, but using a wrapper program that combines the functions of compiler, assembler and linker.

Basic Building Blocks of C

- Data
  are stored in variables that can be of different types. The type of a variable defines the set of values it can have. Variables have to be declared in a declaration statement before they can be used:

```
type name;
int variable1;
int variable2;
int sum;        /* sum of variable1 and variable2 */
```

- Instructions
  tell the computer what to do with the data. Operators act on one or connect two variables to form an expression that can be part of an assignment statement. In addition there area also control statements that regulate the flow of the program. All C statements end with a semi-colon, not with the end of the line.

```
sum = variable1 + variable2
```

Basic Building Blocks of C

- Functions
  are the building blocks of a C program. They have a type (like variables), arguments or parameters listed in round brackets and a function body enclosed in curly braces:

```
type function(parameter1,parameter2){
    first statement;
     ...
    last statement;
}
```

- Files
  The C code is contained in source files that conventionally have the ending `.c`, e.g. `test.c`. One file can contain more than one function and several files can be compiled together into a single executable. This helps with being organised.
- We will later look at variable types, operators and functions in a lot of detail with many examples.

Hello World

Let’s now write our very first program. Using a text editor of your choice (e.g. `emacs` or `xemacs`), write a file `hello.c`:

```
#include <stdio.h>
int main()
{
    printf ("Hello␣World\n");
   return (0);
}
```

- `int main(){}`
  is the main function, the function that is always executed first, it returns a value of type `int` and has no arguments
- `#include <stdio.h>`
  includes a header file that contains the definitions of standard input/output functions, e.g. `printf()`.
- `printf("Hello World\n");`
  prints the words ’Hello World’ to the screen and then goes to the next line
- `return (0);`
  sets the return value of the function `main` to zero. Note the semi-colon at the end of each statement.

Compiling a Program

- There are two types of compilers that are very different in look-and-feel: command-line compilers and Integrated Development Environments (IDEs).
- Under Linux command-line compilers are more common; IDEs exist (e.g. KDevelop), but they typically are front-ends for command-line compilers and assume some knowledge of the command-line compiler and Linux. Almost every Windows compiler contains an IDE.
- We will use the command-line compiler `gcc`. To compile our example `hello.c` we type

```
gcc -o hello hello.c
```

- This tells the `gcc` compiler to read, compile and link the the source file `hello.c` and write the executable code into the output file `hello`.

gcc command-line options

- Like many Linux programs also `gcc` has a (large) number of command-line options, or flags, that turn features off/on. To compile `hello.c` we might actually use something like

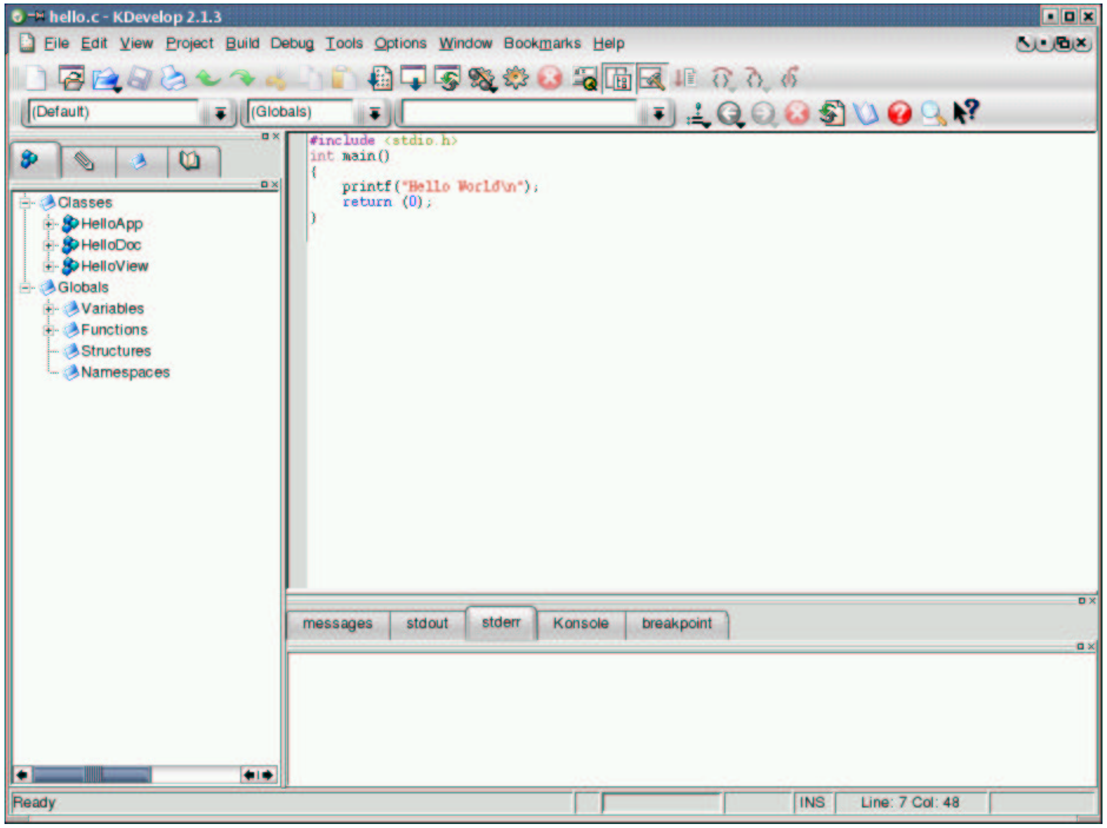
  ```
  gcc -g -Wall -ansi -pedantic -o hello hello.c
  ```

`-g`	enables debugging
`-Wall`	turns on warnings
`-ansi`	turns off `gcc` features that are incompatible with ANSI C
`-pedantic`	issue warnings for any non-ANSI feature

- We will get back to `gcc` flags later. We also will see that the `make` utility saves us from having to remember all the flags we need.

KDevelop IDE

One example for a C programming IDE under Linux is KDevelop, which is part of the KDE desktop. Unfortunately KDevelop assumes some familiarity with Linux, `make`, C (and C++), so that it is perhaps a useful tool after completion of this course ...



```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return (0);
}
```

Coding Style - Comments

- A lot more time is spent on upgrading, maintaining, debugging and adapting code than on actually writing new code ‘from scratch’. For example, it’s three years ago that I wrote the last completely new C program - a simple detector simulation based on the Bethe-Bloch equation.
- For this reason code must be commented and the absence of comments will be treated as a programming error in the marking of assignments.
- Comments in C start with `/*` and end with `*/`, for example

```
/* Say hello to the world */
printf("Hello World\n");
```

- It is customary that longer comments are put into boxes, e.g.

```
/*=======================================*\
 * this could be a section header        *
\*=======================================*/
```

Coding Style - Comments

Besides the comments in the code, that explain the meaning of variables, functions and sections of the code, there should also be a comment box at the begin of each file that contains all relevant information:

- name of the program and what it does
- author and how to reach him/her
- usage: how do you call it, what are the options
- revision history: who edited the file when and why
- file formats, input/output files
- references, i.e. from whom did you copy what
- restrictions: what the program doesn’t do
- known bugs and anything else that’s relevant

Coding Style - Comments

This is what our `hello.c` looks like when it's properly commented:

```
/********************************************************
 * hello -- program to print out "Hello World".         *
 *                                                      *
 * Ralf Kaiser, September 2003                          *
 *                                                      *
 * Reference:  Steve Oualline, Practical C Programming, *
 *             O'Reilly                                 *
 *                                                      *
 * Purpose:  Demonstration of comments                  *
 *                                                      *
 ********************************************************/

#include <stdio.h>

int main()
{
    /* Say Hello to the World */
    printf("Hello World\n");
    return (0);
}
```

Coding Style - Comments

Since C99 also C++ style comments are possible in C. In this case the comment starts with // and extends to the next line break.

```
#include <stdio.h>

// C++ Style Comment
// program that says hello to the world

int main()
{
    printf("Hello World\n");
    return (0);
}
```

This is typically already no problem if the compiler is a C++ compiler that also compiles C (C being a subset of C++). However, it is not recommended for use in C programs in this course, because `gcc -ansi -pedantic` will give an error message and not compile.

Coding Style - Indentation

To make programs easier to read, most programmers indent their programs according to the level of the statement. This is not necessary for the program to compile, but it makes the code easier to understand. Two styles of indentation are frequently used:

```
int main(){
    if (morning) {
        printf("Hello World\n");
    } else {
        printf("Good Night\n");
    }
    return (0);
}
```

```
int main()
{
    if (morning)
    {
        printf("Hello World\n");
    }
    else
    {
        printf("Good Night\n");
    }
    return (0);
}
```

The style of curly braces is up to you. Editors like `emacs` and `xemacs` will help you with the indentation.

References

- By now you know that all the information you need is available online, either directly on your PC or on the web. There are the `man` and `info` pages for each Linux command, including `gcc`
- Manuals in postscript or .pdf format are available on the web for `emacs` and other programs.
- There are websites with ’Frequently Answered Question’ (FAQ) sections and Usenet newsgroups that you can turn to with questions.

References

Books that have been used in the preparation of the C-programming part of this course:

- Practical C Programming, S.Oualline, O’Reilly
  (O’Reilly specialises in Unix related programming handbooks for professionals; took examples from this book, can be found on the web)
- The C Programming Language, B.Kernighan, D.Ritchie
  (the original C bible)
- Programming with C, U.Glasgow
  (C handouts from previous P2T courses)
- ISO 9899, the actual standard definition

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 2

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Computer Memory Organisation
- Number Systems
- Variables
- Simple Operators
- Printing to the Screen

`http://www.physics.gla.ac.uk/~kaiser/`

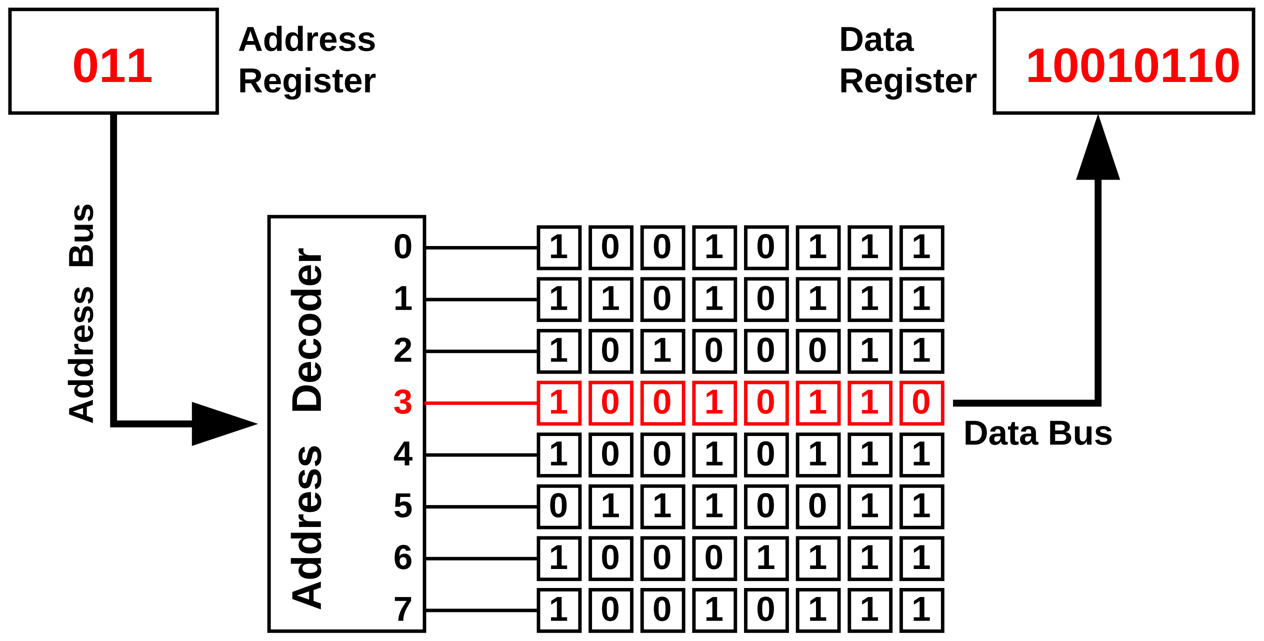
Computer Memory

- Computer memory is classified as either ROM (read only memory) or RAM (random access memory).
- Static RAMs (SRAM) use flip-flops as storage elements and can therefore store data indefinitely as long as DC power is applied.
- Dynamic RAMs (DRAM) use capacitors as storage elements and cannot retain data very long without the capacitors being recharged.
- Data can be read much faster from SRAMs than from DRAMs. DRAMs can store much more data for a given physical size and cost of the memory, because the DRAM cell is much simpler. Therefore you may find that your PC uses DRAM as main memory (e.g. EDO DRAM, extended data out DRAM), but the cache better be SRAM or your PC will be slower than it could be.

Computer Memory cont.

- One bit is the unit of information, i.e. one 1 or 0. Eight bits are one byte.
- A complete unit of information is called a word and generally consists of one or several bytes. As a simplified general rule we can say that memories store data in bytes.
- The memory can be imagined as an array of cells (flip-flops or capacitors) that has 8 columns and a large number of rows. In this picture a 64 MByte memory has 8 columns and 64 million rows (actually , more like 67 million rows, as 1 k is 1024 in memory).
- The location of data in a memory array is called its address.

Memory Organisation



- Data units go in and out of the memory on a set of lines called the data bus.
- For any read or write operation an address is selected by placing the corresponding binary code on a set of lines called the address bus.

Number Systems

- Our regular, every day number system is the decimal system. It's probably so successful because of the 10 fingers we have; in fact the word *digit* comes from the Latin word for finger.
- In the decimal system, the position of the digit indicates it's value in powers of 10:
  $2003 = 2 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$
- In other number systems the position of the digit indicates it's value in powers of another value: powers of 2 for binary, powers of 8 for octal and powers of 16 for hexadecimal numbers.
- Octal number only need the symbols 0..7, hexadecimal numbers use A..F for the numbers 10..15, e.g. 2003 (decimal) becomes 7D3 (hexadecimal).
- Binary, octal and hexadecimal are the natural number systems for computers.

Variable Types

In C data are stored in variables and there are three basic types of variables:

- `int`
  integer variables, whose values are integer numbers such as 4, -128, 147238
- `float`
  floating point numbers, corresponding to non-integer numbers with a decimal point. 5.0 is a floating point number, while 5 is an integer.
- `char`
  character variables with values ’a’ to ’z’, ’A’ to ’Z’, ’0’ to ’9’ plus punctuation marks, parentheses etc.

Variable Types - `int`

- size typically reflects the natural size of integers on the host machine. The C standard does not define the size of numbers. Under Linux (or another Unix) integers are typically 32 bits (4 bytes), providing a range from 2147483647 $(2^{31} - 1)$ to -2147483648.
- The standard header file `limits.h` defines constants for the various numerical limits.

```
/* Minimum and maximum values a ‘signed int’ can hold.  */
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647
```

- for comparison: in Turbo C++ integers are only 16 bit, with a range from 32767 to -32768. The example of 147238 from the last slide would not be possible.

Variable Types - `float` and `double`

- usually `float` variables use 4 bytes, corresponding to a value range of $3.4 \cdot 10^{-38}$ to $3.4 \cdot 10^{+38}$ with a corresponding range of negative numbers and approximately 7 significant digits.
- double variables use 8 bytes, corresponding to a value range of $1.7 \cdot 10^{-308}$ to $1.7 \cdot 10^{+308}$ with a corresponding range of negative numbers and approximately 15 significant digits.
- floating point numbers can be given in exponential form, e.g. 1.2e34 is $1.2 \cdot 10^{34}$
- While it is possible to write .5 instead of 0.5 and 12. instead of 12.0, this should be avoided.
- computers do integers better than floating point numbers, and in floating point operations rounding errors can accumulate.

Variable Types - `char`

- single characters are represented by an 8 bit number, this length is also defined in `limits.h`:

```
/* Number of bits in a ‘char’.  */
#  define CHAR_BIT      8
```

- characters are enclosed in single quotes

```
char char1;    /* first character  */
char char2;    /* second character */
char char3=’z’ /* third character, directly initialised */

char2 = ’9’;   /* assignment statement for second character */
```

- only 7 bits are actively used to encode the character in the ASCII code (American Standard Code for Information Interchange), bit 7 can be used as a parity bit.
- characters can also be specified by \nnn where nnn is their octal code according to ASCII (e.g. \`100` is @)

Variable Types - Qualifiers

- The variable type can be combined with the qualifiers `short` or `long` to limit or extend their range and with the qualifiers `signed` or `unsigned`.
- where `int` is 32 bit, `short int` is only 16 bit, `long int` is 32 bit, but `long long int` is 64 bit
- `long`, `long long` and `unsigned` numbers are marked with `U` and `L`, `LL` at the end. The largest integer number in `gcc` on a Linux machine is an `unsigned long long int`:

```
/* Maximum value an ‘unsigned long long int’ can hold.  (Minimum is 0.)  */
#   define ULLONG_MAX   18446744073709551615ULL
```

- `long double` has a maximum value of `1.18973149535723176502e+4932L` - as defined in `float.h`.

Variable Names

- names in C start with a letter or an underscore (_), followed by any number of letters, numbers or underscores
- for an internal name at least the first 31 characters are significant
- special signs and spaces are not allowed; C commands are reserved words and can’t be used as variables names
- uppercase is different from lowercase, so `max`, `Max` and `MAX` specify three different variables
- names starting with underscores are conventionally used only for internal and systems variables
- lowercase variable names are typical for C, so are lowercase_and_underscore, but some also use thisStyle.

Variable Names - Examples

- examples for valid variable names and declarations are

```
int number_of_students = 47;  /* number of students in this class */
float pi = 3.1415927;         /* pi to 7 decimal places           */
int dataRecoil;               /* Recoil detector data               */
```

- examples for invalid variable names are

```
3rd_entry    /* starts with a number */
all$done     /* contains a '$'       */
int          /* reserved word        */
home phone   /* contains a space     */
```

- Use variable names that describe the contents. I once knew a programmer who had used the names of rivers in Armenia as variables in a high voltage control program. Beautiful, but not very practical.
- Don't forget to comment the variables, even if the names are chosen well.

Simple Operators

There are 5 simple arithmetic operators in C:

Operator	Meaning
*	multiply
/	divide
+	add
-	subtract
%	modulus (return remainder after integer division)

- multiply (*), divide (\) and modulus (%) have precedence over add (+) and subtract (-)
- parentheses () may be used to group terms
- these operators are also refered to as binary operators, because they have two operands, e.g. `a + b`

Simple Operators - Examples

```
int term1;                  /* first term                           */
int term2;                  /* second term                          */
int sum;                    /* sum of first and second term         */
int difference ;            /* difference of first and second term */
int modulo;                 /* term1 modulus term2                  */
int product;                /* term1 * term2                        */
int ratio ;                 /* term1 / term2                        */

int main()
{
term1 = 1 + 2 * 4;          /* yields 2*4=8  8+1=9                  */
term2 = (1 + 2) * 4;        /* yields 1+2=3  3*4=12                 */
sum = term1 + term2;        /* yields 9+12=21                       */
difference = term1 − term2  /* yields 9−12=−3                       */
modulo = term1 % term2      /* yields 9/12=0, remainder is 9        */
product = term1 * term2     /* yields 9*12=108                      */
ratio = 9 / 12              /* yields 9/12=0                        */
return(sum);
}
```

Floating Point vs Integer Divide

- There is a vast difference between an integer divide and a floating point divide. In an integer divide the result is truncated.
- C allows the assignment of an integer expression to a floating point variable. C will automatically do the conversion. Similarly, a floating point number can be assigned to an integer variable; the value will be truncated.
- The result of a division is integer only if both factors are integers or if the result is assigned to an integer variable. Otherwise it is floating point.

Expression	**Result**	**Result Type**
19 / 10	1	integer
19.0 / 10	1.9	floating point
19.0 / 10.0	1.9	floating point

Floating Point vs Integer Divide - Example

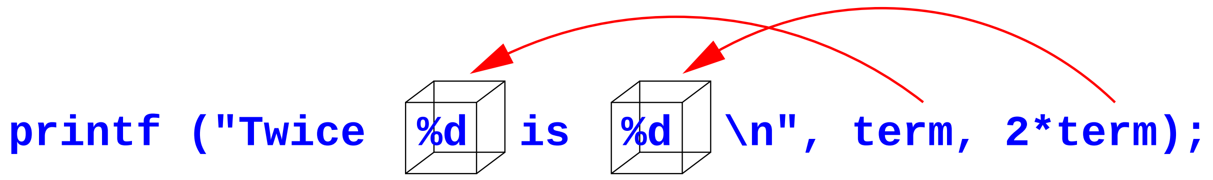
```
/********************************************************
 * Question:                                            *
 *      Why does the following program print:           *
 *      "The value of 1/3 is 0.0" ?                     *
 ********************************************************/
#include <stdio.h>

float answer;   /* The result of our calculation */

int main()
{
    answer = 1/3;
    printf("The value of 1/3 is %f\n", answer);
    return (0);
}
```

Answer: 1 and 3 are both integers, so the problem is the integer divide where the result is truncated. The expression should be written as

```
answer = 1.0 / 3.0
```

The Function `printf`

- The standard function in C used to print something to the screen is `printf`, included in the library `<stdio.h>`.
- The standard form of the `printf` statement is `printf(format, expression-1, expression-2, ...)` where `format` is the string describing what to print, e.g.
- `printf("Twice %d is %d", term, 2*term);` where the special characters %d are the integer conversion specification. Their place is filled with the values of `expression-1` and `expression-2`; everything else is printed verbatim.



- You have to check that conversions and expressions match !

Escape Characters

Special characters or escape characters starting with '\' move the cursor or represent otherwise reserved characters.

Character	**Name**	**Meaning**
\b	backspace	move cursor one character to the left
\f	form feed	go to top of new page
\n	newline	go to the next line
\r	return	go to beginning of current line
\a	audible alert	‘beep’
\t	tab	advance to next tab stop
\’	apostrophe	character ’
\"	double quote	character "
\\\\	backslash	character \\
\nnn		character number nnn (octal)

Format Statements

Besides %d there are other conversion specifications, here is an overview of the most important ones:

Conversion	**Argument Type**	**Printed as**
%d	integer	decimal number
%`f`	float	[-]m.dddddd (details below)
%X	integer	hex. number using A..F for 10..15
%c	char	single character
%s	char *	print characters from string until '\0'
%e	float	float in exp. form [-]m.dddddde$\pm$xx

In addition, the precision and additional spaces can be specified:

`%6d`	decimal integer, at least 6 characters wide
`%8.2f`	float, at least 8 characters wide, two decimal digits
`%.10s`	first 10 characters of a string

Example for `printf`

Why does 2 + 2 = 1075031184 ? (on my laptop; results may vary)

```
#include <stdio.h>

/* Variable for computation results */
int answer;

int main()
{
    answer = 2 + 2;

    printf("The answer is %d\n");
    return (0);
}
```

Answer: The `printf` statement is trying to print a decimal number (%d), but no value is specified - so C makes one up. The proper statement would be

```
printf("The answer is %d\n", answer);
```

ASCII and beyond

- ASCII was defined in 1968 (i.e before most of you were born). It uses 7 bit and works well with American English, which is after all what it was meant to do.
- ASCII already doesn't work with Danish, French or German - but for this there is an extension to 8 bit called Latin1 (aka ISO-8859-1).
- `printf` will typically understand Latin1, so you can use `printf("One beer is 1.95 \xA3");` to print `One beer is 1.95 £`.
- If you're German (like me) you might want to print `'Schöne Grüße'`, which you can do like this: `printf("Sch\366ne Gr\374\337e\n");`
- Of course this still doesn't help you much if you're Thai or Armenian. Have a look at `www.unicode.org` for Unicode (ISO-10646), a 32 bit code for all the letters in the world.

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 3

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Arrays
- Strings
- Reading in from the Keyboard

`http://www.physics.gla.ac.uk/~kaiser/`

Arrays

- An array is a set of consecutive memory locations used to store data.
- Each item in the array is called an element. The number of elements in an array is called the dimension of the array.
- A typical array declaration is

  ```
  /* List of data to be sorted and averaged */
  int data_list[3];
  ```

  In this case the array `data_list` contains the 3 elements `data_list[0]`, `data_list[1]` and `data_list[2]`. The number in square brackets [] is the index.
- C starts counting at 0, not at 1.

Arrays - Example

Calculate sum and average of five numbers. (`array.c`)

```
#include <stdio.h>

float data[5];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

int main()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average =  total / 5.0;
    printf("Total %f Average %f\n", total, average);
    return (0);
}
```

Output:

```
Total 210.000000 Average 42.000000
```

Multidimensional Arrays

- Arrays can have more than one dimension. The declaration for a two-dimensional array is
  `type variable[size1][size2]; /* comment */`
- Example:
  `int matrix[2][4]; /* a typical matrix */`
- C does not follow the notation used in other languages, e.g. `matrix[10,12]`.
- to access an element of the two-dimensional array `matrix` we use the notation
  `matrix[1][2] = 10;`
- C allows to use as many dimensions as needed, limited only by the amount of memory available. A four-dimensional array `four_dimensions[10][12][9][5]` is no problem.

Multidimensional Arrays - Example

(p_array.c)

```
#include <stdio.h>

/* Array of numbers */
int array[3][2];

int main()
{
    int x,y; /* Loop indices */

    array[0][0] = 0 * 10 + 0;
    array[0][1] = 0 * 10 + 1;
    array[1][0] = 1 * 10 + 0;
    array[1][1] = 1 * 10 + 1;
    array[2][0] = 2 * 10 + 0;
    array[2][1] = 2 * 10 + 1;

    printf("array[%d] ", 0);
    printf("%d ", array[0,0]);
    printf("%d ", array[0,1]);
    printf("\n");

    printf("array[%d] ", 1);
    printf("%d ", array[1,0]);
    printf("%d ", array[1,1]);
    printf("\n");

    printf("array[%d] ", 2);
    printf("%d ", array[2,0]);
    printf("%d ", array[2,1]);
    printf("\n");

    return (0);
}
```

What's the problem with this example ?

Multidimensional Arrays - Example cont.

The program on the previous slide produces the following output (again, on my laptop):

```
kaiser@npl03:~/oreilly/pracc/p_array> p_array
array[0] 134518256 134518264
array[1] 134518256 134518264
array[2] 134518256 134518264
```

The problem is the use of the expression `array[x,y]` in the `printf` statement, instead of using the correct expression `array[x][y]`, which would have resulted in

```
kaiser@npl03:~/oreilly/pracc/p_array> p_array
array[0] 0 1
array[1] 10 11
array[2] 20 21
```

More detail: `x,y` is equivalent to `y`, therefore `array[x,y]` is really `array[y]`, which is a pointer to row `y` of the array. But we will only later learn about pointers...

Initialising Arrays

- C allows variables to be initialised in the declaration statement
  `int counter = 0; /* number counted so far */`
- This is especially practical for arrays, where a list of element enclosed in curly braces {} can be assigned:
  `int product_codes[3] = {10, 972, 45};`
- If no dimension is given, C will determine the dimension from the number of elements in the initialisation list:
  `int product_codes[] = {10, 972, 45};`
- The same kind of initialisation at declaration can also be used for multidimensional arrays:

```
int matrix[2][4] =
    {
        {1, 2, 3 ,4},
        {10, 20, 30, 40}
    };
```

Strings

- Strings are sequences of characters. C does not have a built-in string type; instead, strings are created out of character arrays.
- Strings are character arrays with the additional special character \0 (NUL) at the end.
- String constants consist of text enclosed in double quotes, i.e. `"Linux"`. The first parameter to `printf` is a string constant.
- C does not allow to assign one array to another, instead, to fill an array with a string constant we have to copy it into the variable using the standard library function `strcpy`:

```
#include <string.h>
char system[6];
int main(){
    strcpy(system, "Linux");   /* Legal way to fill variable system */
    return(0);
}
```

Strings cont.

- The array can also be filled element by element:

```
system[0] = ’L’;
system[1] = ’i’;
system[2] = ’n’;
system[3] = ’u’;
system[4] = ’x’;
system[5] = ’\0’;
```

- Because C allows variables to be initialised at declaration, this can be used to fill a string in a convenient way. In this case you don’t even have to specify the length of the array. C will determine the dimension of the array itself.

```
char system[] = "Linux"
```

- The above declaration is equivalent to the following initialisation:

```
char system[] = ’L’,’i’,’n’,’u’,’x’,’\0’;
```

Strings cont.

- String and character constants are different: `"X"` is a one-character string, taking up two bytes, one for X, the other one for \0. ’Y’ is just a single character, taking up one byte.
- A string should never be copied into an array that is shorter than the string. Otherwise you are writing into memory that you shouldn’t access and the program can behave unexpectedly.
- The most common string functions are

Function	**Description**
`strcpy(string1, string2)`	copy `string2` into `string1`
`strcat(string1, string2)`	concatenate `string2` onto the end of `string1`
`length = strlen(string)`	get the length of a `string`
`strcmp(string1, string2)`	0 if `string1` equals `string2`, otherwise nonzero

Strings - Example

Putting strings together using `strcat(full.c)`.

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

char first[100];        /* first name */
char last[100];         /* last name */
char full_name[200];    /* full version of first and last name */

int main()
{
    strcpy(first, "John");        /* Initialize first name */
    strcpy(last, "Lennon");       /* Initialize last name  */

    strcpy(full_name, first);     /* full = "John"         */
    /* Note: strcat not strcpy */
    strcat(full_name, " ");       /* full = "John "        */
    strcat(full_name, last);      /* full = "John Lennon"  */

    printf("The full name is %s\n", full_name);
    return (0);
}
```

Output:

```
 The full name is John Lennon
```

Reading in Strings with `fgets`

The standard function `fgets` can be used to read a string from the keyboard. The general form of an `fgets` statement is:
`fgets(name, size, stdin);`

- `name`
  is the name of a character array, aka a string variable. The line, including the end-of-line character, is read into this array.
- `size`
  `fgets` reads until it gets a line complete with ending \n or it reads `size` - 1 characters. It is convenient to use the`sizeof` function: `fgets(name, sizeof(name), stdin);`
  because it provides a way of limiting the number of characters read to the maximum number that the variable can hold.
- stdin
  is the file to read. In this case the 'file' is the standard input or keyboard. Other files will be discussed later under file input/output.

`fgets` - Example 1

Read in a string and output it's length (`length.c`).

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

char line[100]; /* Line we are looking at */

int main()
{
    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    printf("The length of the line is: %d\n", strlen(line));
    return (0);
}
```

Output:

```
Enter a line: test
The length of the line is:  5
```

`test` has only 4 characters - but `fgets` also gets the \n character.

`fgets` - Example 2

Read in first and last name, print out full name (`full1.c`).

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

char first[100];        /* first name of person we are working with */
char last[100];         /* last name                                */
char full[200];         /* full name of the person (computed)       */

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

`fgets` - Example 2 cont.

Output of `fgets` - Example 2:

```
kaiser@npl03:~/oreilly/pracc/full1> full1
Enter first name: John
Enter last name: Lennon
The name is John
 Lennon
```

What happened ? Why is the last name in a new line ?

- The `fgets` command gets the entire line, including the end-of-line. For example, "John" gets stored as {'J','o','h','n','\n','\0'}.
- This can be fixed by using the statement
  `first[strlen(first)-1] = '\0';`
  which replaces the end-of-line with an end-of-string character and so end the string earlier.

The Function `scanf`

- The direct equivalent to the output function `printf` is the input function `scanf`. The syntax of a `scanf` statement is
  `scanf(format, &variable-1, &variable-2, ...)`
  where `format` specifies the types of variables and `&variable-1` is the address of variable-1.
- A typical `scanf` statement would be
  `scanf("%d%d%f", &a, &b, &x)`
  reading in integer values for the variables a and b and a floating point value for the variable x, entered from the keyboard.
- The `%s` conversion in `scanf` ignores leading blanks and reads until either the end-of-string '\0' or the first blank after non-blank characters. For example, if the input from the keyboard is "   ABCD   EFG ", %s will read "ABCD".

Reading Numbers with `fgets` and `sscanf`

- To quote my favorite C book: 'The function `scanf` provides a simple and easy way of reading numbers that almost never works'. `scanf` is not very good at end-of-line handling and you may find yourself having to hit 'return' a couple of times.
- One way around these problems is to use a combination of `fgets` and `sscanf` instead. `sscanf` stands for 'string `scanf`' and works like `scanf`, but acts on strings rather than on keyboard input.
- The code to read in and process a line from the keyboard then looks like this:

```
char line[100];
fgets(line, sizeof(line), stdin);
sscanf(line, format, &variable-1,...);
```

`fgets/sscanf` - Example 1

Read in a number from the keyboard and double it (`double.c`).

```
#include <stdio.h>
char  line[100];   /* input line from console */
int   value;       /* a value to double */

int main()
{
    printf("Enter a value: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &value);

    printf("Twice %d is %d\n", value, value * 2);
    return (0);
}
```

Output:

```
kaiser@npl03:~/oreilly/pracc/double> ./double
Enter a value: 21
Twice 21 is 42
```

Actually, `scanf("%d", &value);` worked as well.

fgets/sscanf - Example 2

Input width and height, output area of triangle (tri.c).

```
#include <stdio.h>
char line[100];/* line of input data */
int  height;   /* the height of the triangle
int  width;    /* the width of the triangle */
int  area;     /* area of the triangle (computed) */

int main()
{
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);

    area = (width * height) / 2;
    printf("The area is %d\n", area)
    return (0);
}
```

fgets/sscanf - Example 2 cont

If we are trying to compile the program tri.c we get the following error messages:

```
kaiser@npl03:~/oreilly/pracc/tri> make
This program fails to compile
gcc -g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi -o tri tri.c
tri.c:18:16: warning: "/*" within comment
tri.c: In function ‘main’:
tri.c:26: ‘width’ undeclared (first use in this function)
tri.c:26: (Each undeclared identifier is reported only once
tri.c:26: for each function it appears in.)
tri.c:30: parse error before "return"
make: *** [tri] Error 1
```

The source code of tri.c contains two of the most common errors: One missing */ at the end of a comment:

```
int height; /* the height of the triangle */
```

and one missing semi-colon at the end of the second printf statement:

```
printf("The area is %d\n", area);
```

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 4

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- More Operators
- Control Statements

`http://www.physics.gla.ac.uk/~kaiser/`

Operators for Shortcuts

- C includes a large number of special-purpose operators, some of which allow to use shortcuts for frequently used operations.
- The most popular one is the increment operator `++`. It allows to replace a statement like `counter = counter + 1;` with the shortcut `counter++`.
- Similar is the decrement operator `--` that decreases the value of a variable by 1.

Operator	**Shortcut**	**Equivalent Statement**
`++`	`x++ ;`	`x = x + 1;`
`--`	`x-- ;`	`x = x - 1;`
`+=`	`x += 2;`	`x = x + 2;`
`-=`	`x -= 2;`	`x = x - 2;`
`*=`	`x *+= 2;`	`x = x * 2;`
`/=`	`x /+= 2;`	`x = x / 2;`
`%=`	`x %+= 2;`	`x = x % 2;`

The Wonderful World of ++

- The increment and decrement operators come in two different flavours, the prefix form `++variable` and the postfix form `variable++`. (By now you also know where C++ comes from...)
- The two forms lead to different results; the prefix first increments and then evaluates the expression, the postfix first evaluates the expression and then increments:

```
number = 5;                          number = 5;
result = number++;                   result = ++number;

result is 5                          result is 6
```

- Easier to read would be:

```
number = 5;
number++;
result = number;
```

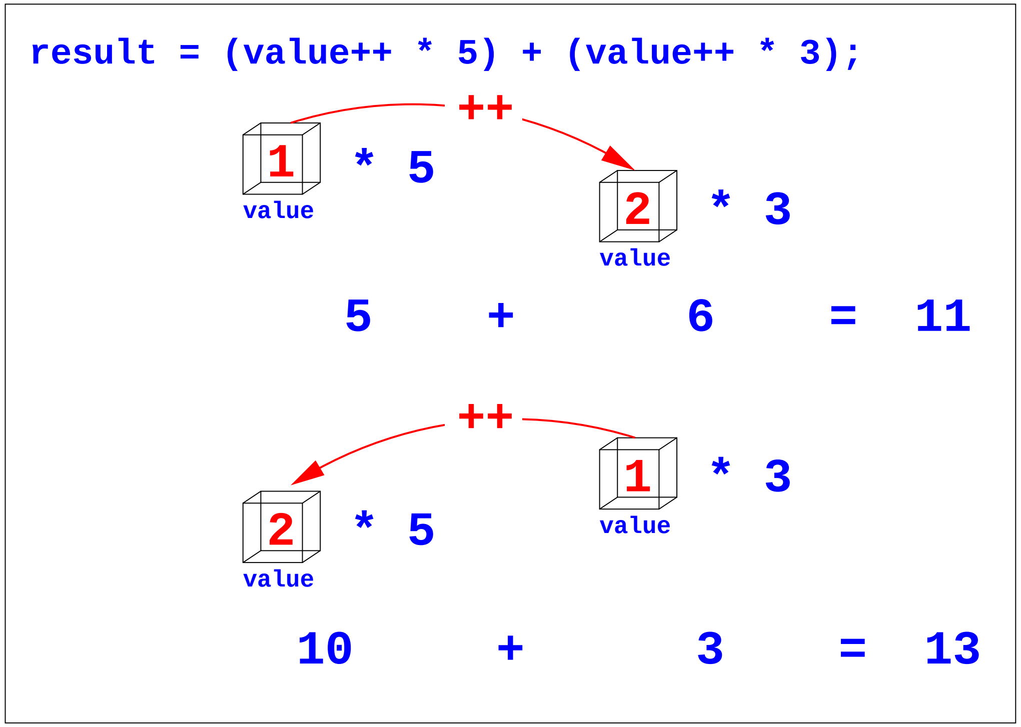
- Compact code is a holdover from the time when storage space cost a lot of money. And some people think it's cool.

The Wonderful World of ++

- Consider the code fragment

```
number = 1;
result = (number++ * 5) + (number++ * 3)
```

The result (11 or 13) actually depends on the compiler:



- Try to avoid using ++ in the middle of expressions, then you don't have to worry about this.

Control Statements

- So far our programs have been linear, i.e. they execute in a straight line, one statement after another.
- The statements we were dealing with were assignment statements.
- Now we are going to introduce control statements,i.e. branching statements and looping statements that change the control flow of a program.
- Branching statements cause a section of code to be executed or not.
- Looping statements are used to repeat a section of code a number of times or until some condition occurs.

The `if` Statement

simple `if` statement:

```
if (condition)
    statement;
```

`if - else if - else` statement:

```
if (condition1){
    statement1;
    statement2;
} else if (condition2){
    statement3;
    statement4;
} else
    statement5;
```

- If the condition is true (non-zero), the statement will be executed.
- If the condition is false (0), it will not be executed.
- Multiple statements may be in curly braces.
- `else` allows a statement to be executed if the condition is not fulfilled.

Relational Operators

- Now how do we formulate the condition for an `if` statement ? In principle, everything that returns a value that is 1 (true) or 0 (false) will work. (Actually, non-zero for true).
- Practically, this brings us to yet another set of operators, because we will mostly use relational operators:

Operator	**Meaning**
<=	less than or equal
<	less than
>	greater than
>=	greater or equal than
==	equal
!=	not equal

- A simple `if` statement might look like this:

```
if (total_owed <=0)
    printf("You owe nothing.\n");
```

`if` - Example

Read in an amount and print out a message (`owe0.c`)

```
#include <stdio.h>
char  line[80];          /* input line */
int   balance_owed;      /* amount owed */

int main()
{
    printf("Enter number of dollars owed:");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &balance_owed);

    if (balance_owed = 0)
        printf("You owe nothing.\n");
    else
        printf("You owe %d dollars.\n", balance_owed);

    return (0);
}
```

Output:

```
Enter number of dollars owed:42
You owe 0 dollars.
```

`if` - Example cont.

- The `fgets` and `sscanf` statements are alright, there is no problem with the length of the character array, so why does the program return `0` ?
- The program illustrates one of the most common errors in C programming.

  ’=’ is not the same as ’==’.

  ’`=`’ is the assignment operator. ’`==`’ is a relational operator that compares two values and returns true/false.
- In the case of our program the statement `if (balance_owed = 0)` assigns the value 0 which is then always printed.
- Correct would be the statement `if (balance_owed == 0)`

How not to use `strcmp`

- The function `strcmp` compares two strings. It returns 0 if they are equal or non-zero if they are different. Obviously, it can be used in the condition of an `if` statement.

```
if (strcmp(string1, string2) == 0);
    printf("Strings equal\n");
else
    printf("Strings not equal\n");
```

- It may be tempting to write compact code and leave the `== 0` bit out:

```
if (strcmp(string1, string2));
    ...
```

However, `strcmp` is counter-intuitive in this case, because it returns 0 (i.e. false, not true), when both string are equal.

The `while` Statement

- The `while` statement is used when a program needs to perform repetitive tasks. The general syntax is:
  ```
  while (condition)
     statement;
  ```
- The program will repeat the execution of the statement inside the `while` until the condition becomes false (0).
- If the condition is initially false, it will never be executed.
- If the condition is always true, i.e. `while (1)`, the loop will loop forever (when nothing else happens...).
- As with `if`, multiple statements can be enclosed in curly braces.

`while` - Example

Print out all Fibonacci Numbers below 100 (`fib.c`).

```
#include <stdio.h>
int   old_number;     /* previous Fibonacci number */
int   current_number; /* current Fibonacci number */
int   next_number;    /* next number in the series */

int main()
{
    /* start things out */
    old_number = 1;
    current_number = 1;

    printf("1\n");    /* Print first number */

    while (current_number < 100) {

        printf("%d\n", current_number);
        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;
    }
    return (0);
}
```

`while` - Example cont.

- Fibonacci numbers are a series that is defined by

$$f_n = f_{n-1} + f_{n-2} \qquad f_0 = 1, f_1 = 1$$

  so the first Fibonacci numbers are 1, 1, 2, 3, 5, 8,.... They appear in nature e.g. in the number of left- and right-turning spirals on the bottom of some pine cones.

- In our C code, the above equation is implemented as

```
old_number = 1;
current_number = 1;

next_number = current_number + old_number;
old_number = current_number;
current_number = next_number;
   }
```

- We could also chose more 'mathematical' names, like `f_n`, `f_n_1` and `f_n_2` - the question is: What will be easier to understand for somebody else or after some time.

The Statements `break` and `continue`

- A `while` loop can be exited when the condition after the `while` becomes false (0). Alternatively, any loop can be exited at any point through the use of a `break` statement.
- Usually, `break` appears in combination with `if`:

```
if (condition)
    break;
```

- Often, `break` is used in an endless `while` loop, to exit once a condition is fulfilled:

```
while (1) {
    if (condition)
        break;
{
```

- The `continue` statement is similar to `break`, but instead of terminating the loop, `continue` starts re-executing the body of the loop from the top.

`break` - Example

Add numbers until '0' is entered, print total (`total.c`).
(Listing leaving out #include statements and variable declarations.)

```
int main()
{
    total = 0;
    while (1) {
        printf("Enter # to add \n");
        printf("  or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    return (0);
}
```

Output:

```
Enter # to add
  or 0 to stop:20
Total: 20
Enter # to add
  or 0 to stop:1
Total: 21
Enter # to add
  or 0 to stop:678
Total: 699
Enter # to add
  or 0 to stop:0
Final total 699
```

`continue` - Example

Now only add positive numbers, count negative ones (`totalb.c`).
(Code fragment, only `while` loop and `printf` statements from.)

```
while (1) {
    printf("Enter # to add\n");
    printf("  or 0 to stop:");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &item);

    if (item == 0)
        break;

    if (item < 0) {
        ++minus_items;
        continue;
    }
    total += item;
    printf("Total: %d\n", total);
}
printf("Final total %d\n", total);
printf("with %d negative items omitted\n",
              minus_items);
```

Output:

```
Enter # to add
  or 0 to stop:-2
Enter # to add
  or 0 to stop:22
Total: 22
Enter # to add
  or 0 to stop:1459890
Total: 1459912
Enter # to add
  or 0 to stop:0
Final total 1459912
with 1 negative items omitted
```

The `for` Statement

- The `for` statement allows to execute a block of code for a specified number of times. The general form of the `for` statement is:

```
for (initial-statement; condition; iteration-statement){
    statement-1;
    ...
    statement-n;
}
```

- This is equivalent to a `while` loop of the shape

```
initial-statement;
while (condition) {
    statement-1;
    ...
    statement-n;
    iteration-statement;
}
```

- The `iteration-statement` is most of the time a counter like `++counter`. It is conventional in C to start counters with 0 (like the numbering of array elements); i.e. count from 0 to 4, not from 1 to 5.

`for` - Example

Print Celsius to Fahrenheit conversion chart for 0 to 100 Celsius (fahrenheit.c).

```
#include <stdio.h>
/* the current Celsius temperature we are working with */
int celsius;
int main() {
    for (celsius = 0; celsius <= 100; ++celsius)
        printf("Celsius:%d Fahrenheit:%d\n",
            celsius, (celsius * 9) / 5 + 32);
    return (0);
}
```

What would happen if we accidentally add a semi-colon at the end of the `for`-statement ?

```
    for (celsius = 0; celsius <= 100; ++celsius);
```

Answer: The program would only print

```
Celsius:101 Fahrenheit:213
```

The `switch` Statement

The `switch` statement is similar to a chain of `if/else` statements. The general form of the `switch` statement is:

```
switch (expression) {
    case constant1:
        statement
        ...
        break;
    case constant2:
        statement
        ...
        /* Fall through */
    case constant3:
        statement
        ...
        break;
    default:
        statement
        ...
        break;
}
```

- `switch` evaluates the value of an expression and branches to one of the `case` labels. Duplicate labels are not allowed. The expression must evaluate an integer, character or enumeration.
- The `case` labels can be in any order and must be constants.
- The `default` label can be put anywhere in the `switch`.
- If no `case` matches and no `default` exists, the `switch` does nothing.

The `switch` Statement cont.

- A `break` statement inside a `switch` tells the computer to continue execution after the `switch`. If a `break` statement is not there, it will continue with the next statement, or fall through to the next statement.
- To make sure that this is intentional, it should be marked by a comment.
- The last `case` statement does not need a `break`, but should get one anyways.
- While a `default` is not necessary and if present can be anywhere, it should always be there and it should be the last statement. At least it should be present as an empty statement that contains only a `break`.

`switch` - Example

Select cases of different operators; code fragment only.

```
switch (operator) {
        case ’+’:
            result += value;
            break;
        case ’-’:
            result -= value;
            break;
        case ’*’:
            result *= value;
            break;
        case ’/’:
            if (value == 0) {
                printf("Error:Divide by zero\n");
                printf("   operation ignored\n");
            } else
                result /= value;
            break;
        default:
            printf("Unknown operator %c\n", operator);
            break;
        }
```

`break`, `continue` and `exit`

- `break` has two uses: Inside a `switch`, `break` causes the program to go to the end of the switch. Inside a `for` or `while` loop, `break` causes a loop exit.
- `continue` is valid only inside a loop and will cause the program to go to the top of the loop.
- In the case of a `continue` statement inside a `switch` that in turn is inside a loop, the `continue` will act on the loop.
- If a `break` statement is inside a `switch` that in turn is inside a loop, it will act on the `switch`.
- The `exit` function requires the inclusion of the header `<stdlib.h>`. It is normally called as `exit(0);` and causes the normal termination of the program, equivalent to reaching the closing curly brace of the `main` function. It will exit from anywhere in the program.

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 5

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- A Short Introduction to ’make’
- The Scope of Variables
- Functions
- Recursion
- Structured Programming

`http://www.physics.gla.ac.uk/~kaiser/`

The `make` Utility

- So far we have compiled our programs by hand using a command line for `gcc`. Before we move on to functions and more complex structures of programs, e.g. multiple source files, we will learn to use the `make` utility.
- `make` handles the details of compiling and linking for us and only does those steps that are necessary by checking which source file has been edited when.
- `make` reads in the file `Makefile` in the same directory.
- `make` exists for Windows as well as for Unix. The version typically used under Linux is GNU make `gmake`; usually `make` will be defined as alias for `gmake`.

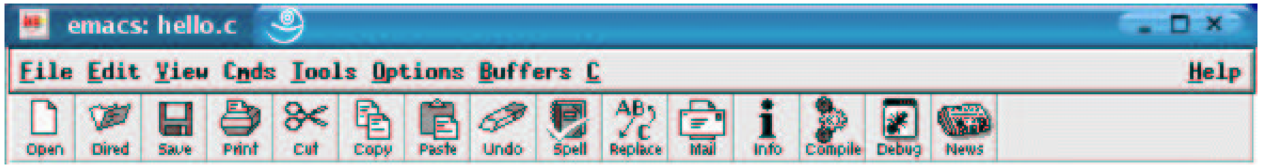
Simple Makefile for gcc

A simple Makefile for gcc that can be used with C programs contained in a single file by just editing the name of the source file:

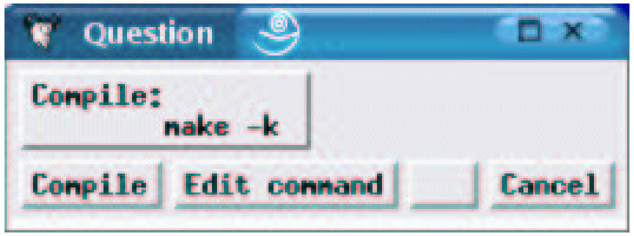
```
#-----------------------------------------------#
#       Makefile for unix systems               #
#    using a GNU C compiler                     #
#-----------------------------------------------#
CC=gcc
CFLAGS=-g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi
#
# Compiler flags:
#       -g      -- Enable debugging
#       -Wall   -- Turn on all warnings
#       -D__USE_FIXED_PROTOTYPES__
#               -- Force the compiler to use the correct headers
#       -ansi   -- Don't use GNU extensions.  Stick to ANSI C.

hello: hello.c
        $(CC) $(CFLAGS) -o hello hello.c

clean:
        rm -f hello
```

Using xemacs with make

- If you are using xemacs to edit your source code, you can configure the ‘Compile’ button easily to call make.



Upon first clicking at the ‘Compile’ button, a configuration window opens that allows to configure the compilation command:

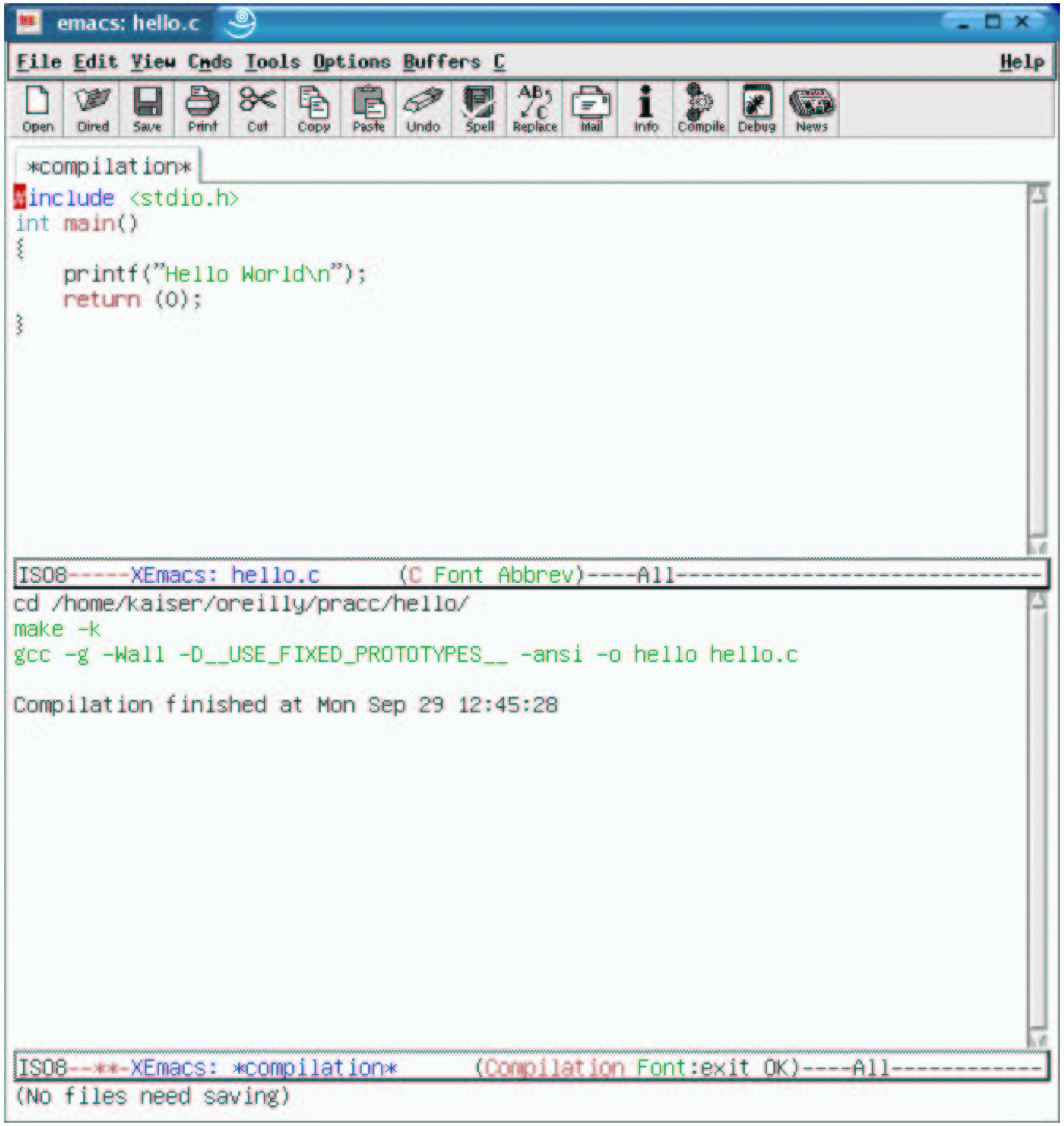


```
Compile:
        make -k
```

Afterwards, simply clicking at ’Save’ and ’Compile’ will compile and link the program.

- The xemacs window will split vertically into two, with the bottom one displaying the compiler command and warning messages that otherwise go to stdout.

Using `xemacs` with `make`


emacs: hello.c
File Edit View Cmds Tools Options Buffers C Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
compilation

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return (0);
}
```

```
ISO8-----XEmacs: hello.c        (C Font Abbrev)----All---------------------------
cd /home/kaiser/oreilly/pracc/hello/
make -k
gcc -g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi -o hello hello.c

Compilation finished at Mon Sep 29 12:45:28
```

```
ISO8--**-XEmacs: *compilation*       (Compilation Font:exit OK)----All------------
(No files need saving)
```

Using `xemacs` with `make` cont.

- Depending on the version of `emacs` or `xemacs` there may be an entry 'Compile' in a pull down menu instead and the configuration of the command may happen not in an extra window but in the command line at the bottom of emacs.
- The `make` command itself of course also has a number of command line options (flags). You can use `'man make'` to find out more. A frequently used option is `make -k`:

```
-k   Continue as much as possible after an error.  While the tar
     get that failed, and those that  depend  on  it,  cannot  be
     remade,  the other dependencies of these targets can be pro
     cessed all the same.
```

- Warning on Makefile syntax:
  The line `'$(CC) $(CFLAGS) -o hello hello.c'` must start with a tab, not eight spaces.

Scope and Class of Variables

- Variables we have used so far were global variables. Now we will have a look at local variables and the scope of variables in general.
- All variables have two attributes besides their type: scope and class.
- The scope of a variable is the area of the program in which the variable is valid.
- A global variable is valid everywhere in the program.
- A local variable has a scope that is limited to the blockin which it is declared. It cannot be accessed outside that block.
- A block is a section of code enclosed in curly braces{}.
- The class of a variable is either permanent or temporary. (We'll get back to that.)

Scope of Variables cont.

- Between a global and a local variable with identical names, the local one takes precedence. However, this is generally not good practice.

```
int global;
main()
{
    int local;

    global = 1;
    local = 2;

    {
        int very_local;

        very_local = global+local;
    }

}
```

```
int total;
int count;

main()
{
    total = 0;
    count = 0;
    {
        int count;

        count = 0;

        while (1) {
            if (count > 10)
                break;
            total += count;
            ++count;
        }
    }
    ++count;
    return(0);
}
```

Class of Variables

- Global variables are always permanent. They are created and initialised before the program starts and remain until it is terminated.
- Temporary variables are allocated from a section of memory called the stack at the beginning of the block. If you try to allocate too many local variables, you will get a ’Stack overflow’ error. Each time the block is entered, the temporary variables are initialised.
- The size of the stack depends on system and compiler. under MS-DOS/Windows the stack space must be less than 65,536 bytes, on Linux systems it may be larger, typically up to 8 MByte.
- gcc has compiler options that help dealing with the stack size.
- Local variables are temporary unless they are declared static, in which case they may be local, but permanent.

Scope and Class Overview

Declared	**Scope**	**Class**	**Initialised**
outside all blocks	global	permanent	once
`static` outside all blocks	global	permanent	once
inside a block	local	temporary	each time block is entered
`static` inside a block	local	permanent	once

- A `static` declaration made outside blocks indicates that the variable is local to the file in which it is declared. (More about this if/when we get around to multiple files.)

Class of Variables - Example

Demonstrate the difference between permanent and temporary local variables (`vars.c`).

```
#include <stdio.h>

int main() {
    int counter;                                    /* loop counter */
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;                          /* A temporary variable */
        static int permanent = 1;                   /* A permanent variable */
        printf("Temporary %d Permanent %d\n",
            temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

Output:

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Functions

- Functions allow to group commonly used code into a compact unit that can be used repeatedly. We have already encountered one function, the function `main()`.
- The generic syntax for a function is

```
return-type function-name(parameters)
{
   declarations;
   statements;
   return(result);
}
```

- A function consists of a header (the first line) and a body (the part in curly braces).
- The return-type is not strictly required, if missing it defaults to `int`. However, you should always use one, keeping in mind that a later maintainer otherwise might wonder if you intended `int` or just forgot.

Functions cont.

- Assume we have a function

  ```
  float triangle(float width, float height)
  ```

  It is called e.g. as `triangle(1.3, 8.3)`, in which case C copies the values (1.3 and 8.3) into the functions parameters (width and height). This is known as 'call by value'.

- The alternative would be 'call by reference', where not the value of a variable is passed to the function, but it's address. (This is e.g. the case in FORTRAN.)

- A C function cannot pass data back to the caller using parameters. (Well, it can with some tricks using pointers, but we'll only learn that later.)

- The `return` statement is used to give the result to the caller. The return statement can contain a statement, e.g. `return(width*height/2.0);`

Function Prototypes

- If we want to use a function before we define it - because we like to order the functions in the file to be in a particular order, or we just want to be sure and not have to check in which order the functions are - we must declare the function just like a variable to inform the compiler about it.
- We use a declaration like

  ```
  float triangle (float width, float height);
  ```

  This declaration is called the function prototype.
- In the function prototype variable names are not required, only their type, so it would also be ok to just write

  ```
  float triangle (float, float);
  ```
- Strictly speaking, if no prototype is specified the C compiler assumes the function returns an `int` and takes any number of parameters. But we don't want to get into bad habits here.

Functions and `void`

- A function my have no parameters, but the function prototype should not have an empty parameter list. (At least not if it's not `main()` which typically has just that.) In this case the keyword `void` is used to indicate an empty parameter list:

```
int next_index(void)
```

- In an assignment statement you only have to use empty brackets:

```
value = next_index()
```

- A function also does not have to have a `return` value. In this case it gets the return-type `void`, indicating that no value will be returned.

```
void print_answer(int answer){
    if (answer < 0) {
        printf("Answer corrupt\n");
        return;
    }
    printf("The answer is %d\n", answer);
}
```

Functions and Comments

- This is a good moment to come back to the importance of comments. In the example used over the last few lectures we have most of the time left most of the comments out - simply because otherwise not all of the code would fit onto the slide. However, comments are crucial. (And expected in your laboratory assignments and exams....).
- Each function should begin with a comment block, containing
  - Name of the function
  - Description of what the function does
  - Description of each of the parameters of the function
  - Description of the return value of the function
  - Formats, references, anything else that would be really useful

Function - Example

Compute area of a triangle (tri-sub/tri-prog.c).

```
#include <stdio.h>

/********************************************
 * triangle -- compute area of a triangle   *
 *                                          *
 * comments, comments, comments...          *
 ********************************************/
float triangle(float width, float height)
{
    float area;     /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}

int main()
{
    printf("Triangle #1 %f\n", triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", triangle(1.2, 2.0));
    return (0);
}
```

Recursion

- Recursion occurs when a function calls itself, directly or indirectly. Some programming functions, e.g. the factorial, lend themselves naturally to recursive algorithms.
- A recursive function must follow two basic rules
  - It must have an ending point.
  - It must make the problem simpler.
- Example: Factorial $n!$
  A definition of factorial is

$$0! = 1 \qquad n! = n * (n-1)!$$

- In C this turns into

```
int fact(int number){
    if (number == 0)
        return (1);
    return (number * fact(number-1));
}
```

Structured Programming

- There are quite a number of different programming methodologies, some use simple flow-charts, others object-oriented design (OOD), one of the latest fads is ‘extreme programming’ that involves two programmers working together (and some other stuff). In this course we’re not going to get into any of these.
- However, now that we know functions, we can go and structure our programs.
- Simple rules are to divide the code into functions, that each should be short enough to be not to hard to understand. Three A4 pages is a practical limit. Shorter is easier.
- Also, have an idea of the overall structure that the program has.
- And start with simple versions of the functions and make them more complex when the simple versions work.
- Don’t forget about comments.

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 6

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- The C Preprocessor
- Bit Operations

`http://www.physics.gla.ac.uk/~kaiser/`

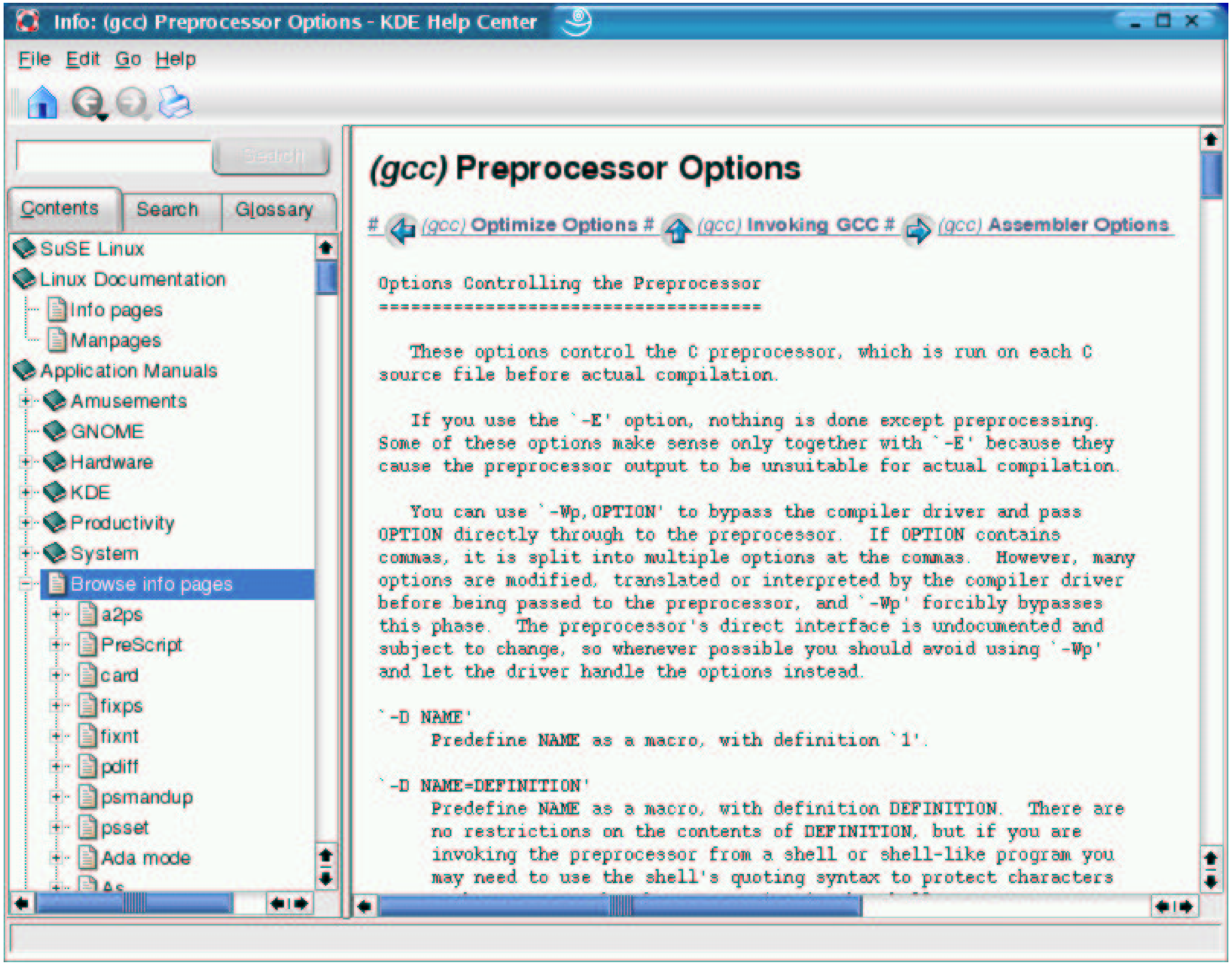
The C Preprocessor

- The C preprocessor is a separate program called by `gcc` before the compiler and it does just that: it acts on the source code and alters it before it is compiled.
- We have already (without pointing it out) made use of the preprocessor. Preprocessor instructions are those that start with a hash (#) in the first column, like `#include <stdio.h>`.
- The syntax of the preprocessor is completely different from the syntax of C; it has no understanding at all of C constructs.
- In fact, it is one of the most common errors of new programmers to try to use C constructs in a preprocessor directive.
- Where C has in principle a free format, the preprocessor does not: The hash mark (#) always must be in the first column.
- A preprocessor directive ends at the end-of-line, not with a semi-colon. A line may be continued by putting a backslash (\) at the end.

`gcc` Preprocessor Options

- `gcc -E program.c`
  puts program.c only through the preprocessor and sends the results to `stdout`. This can be invaluable to track down errors that appear to be caused by preprocessor instructions. Under Linux the output can be re-routed to a file via `gcc -E program.c > program_pp.c` or to a display tool like `less` via `gcc -E program.c | less`.
- `gcc -DNAME program.c`
  allows to define a constant `NAME` on the command line. If there are conditional compilation options this saves you from having to edit the file every time.
- `gcc -DNAME=value program.c`
  the constant `NAME` can also get a value assigned to it.
- `gcc -o FILE program.c`
  writes the compiled code into `FILE` instead of `a.out`, the C default.

gcc Preprocessor Options



(gcc) Preprocessor Options

```
Options Controlling the Preprocessor
====================================

   These options control the C preprocessor, which is run on each C
source file before actual compilation.

   If you use the `-E' option, nothing is done except preprocessing.
Some of these options make sense only together with `-E' because they
cause the preprocessor output to be unsuitable for actual compilation.

   You can use `-Wp,OPTION' to bypass the compiler driver and pass
OPTION directly through to the preprocessor.  If OPTION contains
commas, it is split into multiple options at the commas.  However, many
options are modified, translated or interpreted by the compiler driver
before being passed to the preprocessor, and `-Wp' forcibly bypasses
this phase.  The preprocessor's direct interface is undocumented and
subject to change, so whenever possible you should avoid using `-Wp'
and let the driver handle the options instead.

`-D NAME'
     Predefine NAME as a macro, with definition `1'.

`-D NAME=DEFINITION'
     Predefine NAME as a macro, with definition DEFINITION.  There are
     no restrictions on the contents of DEFINITION, but if you are
     invoking the preprocessor from a shell or shell-like program you
     may need to use the shell's quoting syntax to protect characters
```

The `#include` Instruction

- The `#include` directive allows the program to source code from another file. The syntax is simply

```
#include file-name
```

- Files that are included in other programs are called header files and it is customary to give them the ending .h.
- If the file name is in angle brackets (<>), like

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

  the file is a standard header file. Under Linux, these files are located in `/usr/include`. Standard include files define data structures and macros used by library routines. E.g. `printf` is such a library routine.
- Local include files may be specified by using double quotes (" ") around the file name, e.g. `#include "defs.h"`. Absolute pathnames should be avoided in this case, because it makes the code less portable. (Well, that's generally true...)

The `#include` Instruction cont.

- Anything can be put into a header file. However, good programming practice allows only definitions and function prototypes.
- Include files may be nested, i.e. they may contain `#include` instructions themselves. This may lead to problems, if two include files (`one.h, two.h`) each include the same third file (`three.h`). In this case the contents of `three.h` would be included twice before the compiler is called. Depending on the contents of `three.h` this may lead to fatal errors.
- A way around this problem is to build a check into `three.h` to see if it has been included already. This can be done using the instruction `#ifndef symbol` which is true if `symbol`is not defined:

```
#ifndef _THREE_H_INCLUDED_
......
#define _THREE_H_INCLUDED_
#endif /* _THREE_H_INCLUDED _ */
```

The #define Instruction

- The general form of a simple #define statement is

  ```
  #define name substitute-text
  ```

  where name can be any valid C identifier and substitute-text can be anything.
- The preprocessor will then take any occurrence of name in the source code and replace it with substitute-text before handing it over to the compiler.
- The #define instruction can be used to define constants. For example, the constant PI can be defined by

  ```
  #define PI 3.1415926
  ```

- By convention, constants defined in this way are given names in upper case letters to distinguish them from variables, which are given names in lower case letters.
- #define can also be used to define macros, i.e. statements or groups of statements

`#define` vs `const`

- The keyword `const` is relatively new; older code only uses `#define` to define constants.
- However, `const` has several advantages:
  C checks the syntax of `const` statements immediately and `const` uses C syntax and follows normal C scope rules.
- So the two ways to define the same constant look like this:

```
#define MAX 10        /* Define a value using the preprocessor */

const int MAX = 10;   /* Define a C constant integer  */
```

- `#define` can only define simple constants, while `const` can define almost any type of C constant, including things like structure classes. (That we don't know yet and will learn about a bit later.)

The `#define` Instruction - Example 1

Simple `while` loop illustrating the use of a `#define` statement.

```
 1 #define BIG_NUMBER 10 ** 10
 2
 3 main()
 4 {
 5     /* index for our calculations */
 6     int   index;
 7     index = 0;
 8     /* syntax error on next line */
 9     while (index < BIG_NUMBER) {
10         index = index * 8;
11    }
12     return (0);
13 }
```

- Line 9 expands to

  ```
  while (index < 10 ** 10)
  ```

  and ’**’ is a FORTRAN operator that is illegal in C
- The preprocessor does not check for correct C syntax.

Output (trying to compile):

```
kaiser@npl03:~> make -k
gcc -g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi -o big big.c
big.c:4: warning: return type defaults to ‘int’
big.c: In function ‘main’:
big.c:9: invalid type argument of ‘unary *’
make: *** [big] Error 1
```

The #define Instruction - Example 2

`if` construction illustrating the use of a `#define` statement.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define DIE \
 5   printf("Fatal Error:Abort\n");exit(8);
 6
 7 int main() {
 8     /* a random value for testing */
 9    int value;
10    value = 1;
11    if (value < 0)
12        DIE
13    printf("We did not die\n");
14    return (0);
15 }
```

Output: Nothing. Program exits.

- Properly indented, line 11 and 12 expand to

  ```
  if (value < 0)
      printf("Fatal Error:Abort\n");
  exit(8);
  ```

  the program exits always.
- The cure is to put curly braces ({})around all multistatement macros.

Parameterised Macros

- So far we have only discussed simple `define` statements or macros. But macros also can take parameters. For example, the following computes the square of a number:

```
#define SQR(x) ((x) * (x))
```

- When used, the macro will replace x by the text of the following argument:

`SQR(5)` expands to `((5) * (5))`

- Always put parentheses () around the parameters of a macro, otherwise unexpected problems may occur.
- Note that there must be no space between the macro name and the parentheses that enclose the parameter.

Parameterised Macro - Example

Illustrating possible problems with a parameterised macro (`sqr.c`, correct in `sqr-i.c`).

```
#include <stdio.h>
#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter)
    {
        printf("x %d, x squared %d\n",
            counter+1, SQR(counter+1));
    }
    return (0);
}
```

Output:

```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

- `gcc -E` shows that `SQR(counter+1)` was expanded to `(counter+1 * counter+1)`.
- With `x` being `counter+1` this actually becomes `x-1 + x = 2x-1`.
- The problem are the missing parentheses in the macro definition.

Conditional Compilation

- The preprocessor allows conditional compilation, i.e. sections of the source code are marked and if they are compiled or not depends on whether a condition is fulfilled.
- This is typically done through a combination of the `#define` instruction with `#ifdef`, `#else`, `#ifndef` and `#endif`.
- For example, to print a message indicating that we are dealing with a debugging version of the code or with the production version could be done by switching a variable `DEBUG` on with `#define DEBUG` or off with `#undef DEBUG` and preprocessor code like

```
#ifdef DEBUG
    printf("Test version. Debugging is on.\n");
#else DEBUG
    printf("Production version\n");
#endif /* DEBUG */
```

- This feature is of great use for the portability of C code to different machines.

Bits and Bytes Revisited

- A bit is the smallest unit of information, represented by the values 0 and 1. At the machine level it corresponds to on/off, high/low, charged/discharged.
- Bit manipulations are used to control the machine at the lowest level, closest to the hardware. They are needed for low-level coding, like writing device drivers, pixel-level graphic programming or custom made data acquisition systems. In nuclear or particle physics experiments this will typically be needed somewhere.
- Eight bits form a byte. One byte can be represented by the C data type `char`.
- Instead of binary numbers, hexadecimal numbers can be used to represent bits and bytes. In this case each hexadecimal number represents 4 bits, or two hexadecimal numbers one byte.

Binary and Hexadecimal Numbers Revisited

Hexadecimal	Binary	Hexadecimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

- Remember that there is a difference between the number and it’s representation. By changing from one system to another the number stays the same, only the representation changes.

Bitwise Operators

Bit operators, or bitwise operators, allow to work on individual bits. They work on any integer or character data type.

Operator	**Meaning**
&	bitwise and
|	bitwise or
^	bitwise exclusive or
∼	complement
«	shift left
»	shift right

The Bitwise And Operator (&)

The bitwise and operator & compares two bits. If they are both 1 the result is 1, otherwise it is 0:

Bit1	**Bit2**	**Bit1&Bit2**
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
printf("%x & %x = %x\n", 0x45, 0x71, (0x45 & 0x71));
```

outputs `45 & 71 = 41`.

This is because:

```
        0x45    binary 01000101
    &   0x71    binary 01110001
    ---------------------------
    =   0x41    binary 01000001
```

The Bitwise Or Operator

The inclusive or (or simply the or) operator (|) compares its two operands and if one or the other bit is 1, the result is 1.

Bit1	**Bit2**	**Bit1&Bit2**
0	0	0
0	1	1
1	0	1
1	1	1

Example:

	0x45	binary 01000101
&	0x71	binary 01110001
=	0x75	binary 01110101

It is tempting to assume that there is a simple rule for (&) and (|) with hexadecimal operands, i.e. that the result of the bitwise and & is the combination of the smaller digits, the result of the bitwise or | is the combination of the larger digits. However, this is unfortunately not true.

Bitwise Operators and Hexadecimal Numbers

Bitwise Operators and Hexadecimal Numbers (band.c).

```
int main()
{
  int i, j;

  printf("  &");
  for (j=0; j<16; j++)
{
  printf("%3d", j);
}
  printf("\n\n");
  for (i=0; i<16; i++)
    {
      printf("%3d", i);
      for (j=0; j<16; j++)
{
  printf("%3d", i&j);
}
      printf("\n");
    }
  printf("\n\n");
  return(0);
}
```

Output:

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

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

The Bitwise Exclusive Or Operator

The exclusive or operator ^ (also know as xor) compares its two operands and is 1 if one bit is 1, but not both.

Bit1	Bit2	Bit1 ^ Bit2
0	0	0
0	1	1
1	0	1
1	1	0

Example:

	0x45	binary 01000101
^	0x71	binary 01110001
=	0x34	binary 00110100

The Ones Complement Operator (∼)

The ones complement or not operator ∼ (also know as xor) is a unary operator that returns the inverse of its operand.

Bit	**∼Bit**
0	1
1	0

Example:

c=	0x45	binary 01000101
∼c=	0xBA	binary 10111010

The Left and Right Shift Operators ($<<,>>$)

The left-shift operator « moves the data to the left by a specified number of bits. Any bits that are shifted out on the left side disappear; new bits coming in from the right are zeros. The right-shift » does the same in the other direction. For example:

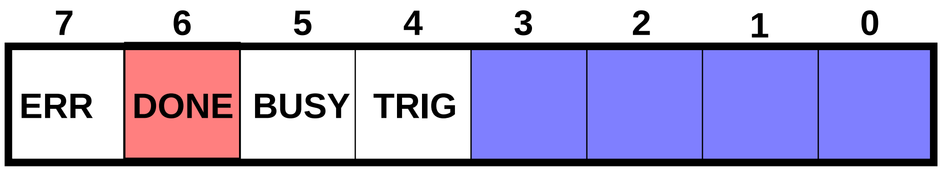
	hexadecimal	**binary**	**decimal**
	c=0x1C	00011100	28
c«1	c=0x38	00111000	56
c»2	c=0x07	00000111	7

Shifting left/right by one bit is the same as multiplying/dividing by 2. `q = i >> 2` is the same as `q = i / 4`. And shifting is much faster than division.

This might give you the idea to use this trick to speed up your code. Thankfully you don't have to do this, because your compiler is smart enough to do just this for you. So don't.

Registers

- Registers can be seen as a special kind of computer memory. They have two basic functions: data storage and data movement.
- This means that they are used to hold data that are being manipulated, that are about to be send somewhere, or just received from somewhere, or they indicate e.g. a status.
- Registers in principal can have any number of bits, but 8 bit registers (1 byte) are typical.
- In registers bits can be shifted left and right, therefore one also finds the term shift register.
- As an example, take the following status register, where bit 6 is the `DONE` bit, indicating that an operation, e.g. a data transfer, is done.



Setting, Clearing and Testing Bits

- Now let’s assume that we have a variable `status` of type `char`, where bit 6 is the `DONE` bit. The hexadecimal form of only bit 6 being set is 0x40. We want to be able to set this bit, test if it is set and clear it.
- To set a bit we use the bitwise or operator |:

  `status = status|0x40` or `status |= 0x40`

- To test if a bit is set we use the bitwise and operator &. This is also know as masking out the bit:

  `if ((status & 0x40) != 0)` or `if (status & 0x40)`

- To clear a bit we create a mask that has all bits set except the one we want to clear. This is done using the not operator ∼. Then the variable/register is anded with the mask to clear the bit:

  `status = status & ∼0x40` or `status &= ∼0x40`

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 7

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Pointers, Pointers, Pointers

`http://www.physics.gla.ac.uk/~kaiser/`

People and Addresses

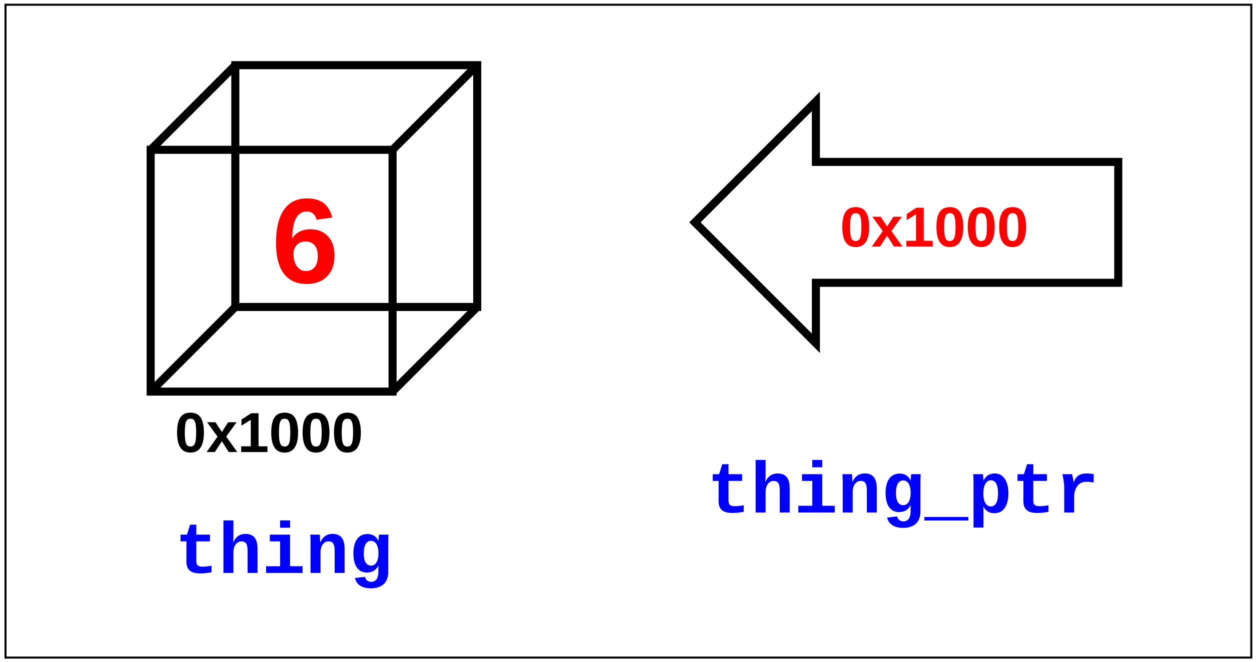
- You may remember receiving a letter addressed to ’the occupier’ of your flat. This may e.g. have been a reminder to register for the next election or it may have been a letter from the electricity supplier.
- They clearly meant you, without knowing you name. When you answered them you would have included your name, e.g. by signing the letter.
- Another example would be that instead of ’Tony Blair’, i.e. the name of the prime minister, the news often just refer to ’10 Downing Street’ and everybody know that this means Tony Blair.
- More precisely, it means Tony Blair at the moment - in the future there will likely be a different name at the same address while Tony Blair will still be around but at a different address.
- Now comes the relevance to C programming: Any variable has (at a given time) a particular value that is stored at a specific address.

Variables, Values and Addresses

- Each variable has several attributes that belong to it and that define it:
  - name
  - type
  - value
  - address

  Usually the variable address is something that the system uses implicitly and that we don't have to worry about.
- However, we can also have a variables where the value is a memory address.
- These variables known as address variables and in C they are called pointers, because they point to the location (address) of something (the value of the variable).
- Pointers are typical for C, not very intuitive, but very powerful. And they are a bit harder to grasp. That's why we will spend a bit more time on them.

Things and Pointers



- Assume we have a variable called `thing`. The value of the variable is 6. The address of `thing` is `0x1000`.
- Our pointer `thing_ptr` contains the address `0x1000`. Because this is the address of `thing`, we say that `thing_ptr` points to `thing`.

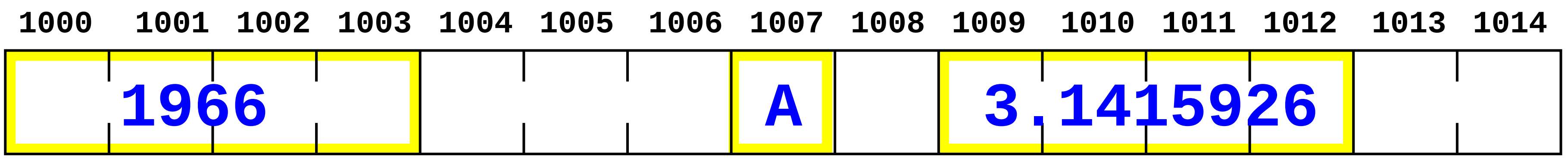
Pointers

- Pointers are linked to a specific type of variable. A pointer is declared by putting an asterisk (*) in front of the variable name in the declaration statement:

```
int thing;           /* define a thing */
int *thing_ptr;      /* define a pointer to an integer */
float *result_ptr;   /* define a pointer to a float */
```

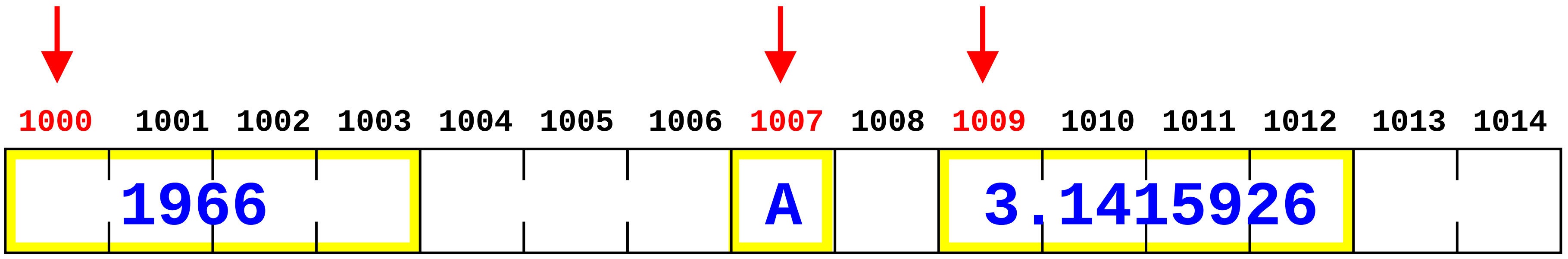
- Different types of variables occupy different amounts of storage space in memory (e.g. 1 byte for `char` and 4 byte for `float`)
- The compiler "remembers" the size of the variable that the pointer points to.
- It is a practical convention (but not required) to give pointer variables names with the extension `_ptr`. This helps with keeping pointers and variables apart. (You may find different conventions elsewhere, e.g. using p_ as a prefix.)

Visualising Pointers



```
int a = 1966;      char b = ’A’; float c = 3.1415926;
```

```
int *a_ptr;     char *b_ptr;     float *c_ptr;
 a_ptr = &a;     b_ptr = &b;      c_ptr = &c;
```



Pointer Operators

- Two unary operators are used in conjunction with pointers:
- The operator ampersand (&) returns the address of a thing - which is a pointer.
- The operator asterisk (*) returns the object to which a pointer points - i.e. what is found at the address that is the value of the pointer.

C Code	**Description**
`thing`	the integer variable named "thing"
`&thing`	address of the variable "thing" (a pointer)
`*thing`	is illegal, the operation is invalid
`thing_ptr`	pointer to an integer
	(may or may not be the specific integer `thing`)
`*thing_ptr`	integer variable at the address `thing_ptr` points to
`&thing_ptr`	is legal, but odd, because it's a pointer to a pointer

Pointer Operators cont.

```
int thing;  /* declare an integer (a thing) */
thing = 4;
```

The variable `thing` is a thing. The declaration `int thing` does not contain an *, so `thing` is not a pointer.

```
int *thing_ptr;  /* declare a pointer to a thing */
```

The variable `thing_ptr` is a pointer, indicated by the * in the declaration (and by the extension `_ptr`).

```
thing_ptr = &thing;  /* point to the thing */
```

The expression `&thing` is a pointer (the address of the variable `thing`). This is now assigned to `thing_ptr`.

```
*thing_ptr = 5;  /* set "thing" to 5 */
```

The expression `*thing_ptr` indicates a thing, because the * tells C to look at the data pointed to (an integer in this case), not the pointer itself. We now have set `thing` to 5. Note that `thing_ptr` points at any integer, it may or may not point to the specific variable `thing`.

Pointer Operators - Example

printf using variable and pointer to variable. (thing.c)

```
#include <stdio.h>
int main()
{
    int   thing_var;                  /* define a variable for thing */
    int  *thing_ptr;                  /* define a pointer to thing */

    thing_var = 2;                    /* assigning a value to thing */
    printf("Thing %d\n", thing_var);
    thing_ptr = &thing_var;           /* make the pointer point to thing */
    *thing_ptr = 3;                   /* thing_ptr points to thing_var so */
                                      /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    printf("Thing %d\n", *thing_ptr); /* another way of doing the printf */
    return (0);
}
```

Output:

```
Thing 2
Thing 3
Thing 3
```

Pointers as Function Arguments

- C passes parameters to a function using call by value, i.e. the parameters go only one way into a function.
- Not the parameter itself, only it’s value is handed to the function. The only result of a function is a single return value.
- Pointers can be used to get around this restriction.
- Instead of passing a variable to a function (which would only pass the value of the variable) we pass a pointer to the function (which passes the value of the pointer).
- The value of the pointer is an address, the address of the variable that we can change now.
- The parameter handed to the function (the address) is not changed, but what it points to (the value of the variable at the address) is changed.

Pointers as Function Arguments - Example

Function that demonstrates the use of pointers to pass parameters that can be changed (`call.c`).

```
#include <stdio.h>
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}

int main()
{
    /* number of times through */
    int  count = 0;

    while (count < 10)
        {
        inc_count(&count);
        printf("%d\n", count);
        }

    return (0);
}
```

- `main` calls the function `inc_count` to increment the variable `count`.
- Passing `count` would only pass it's value (0).
- So the address `&count` is passed instead, as a parameter specified as a pointer to an integer `(int *count_ptr)`.
- Note that the parameter (the address) is not changed, but what it points to (the value at the address) is changed.

`const` Pointers

- Pointers can be constant, but this is a little tricky, because either we can have a constant pointer or a pointer to a constant.
- `const char *answer_ptr = "Forty-Two";`
  does not mean that the variable `answer_ptr` is a constant, but that the data pointed to by `answer_ptr` is a constant. The data cannot change, but the pointer can.
- If we put `const` after the `*` we tell C that the pointer is constant, e.g. `char *const name_ptr = "Test";`.
  In this case the pointer cannot be changed, but the data it points to can.
- Finally, if we really want to, we can create a `const` pointer to a `const` variable:
  `const char *const title_ptr = "Title";`.

Pointers and Arrays

- C allows pointer arithmetic (addition and subtraction). Because the elements of an array are assigned to consecutive addresses, this allows to navigate an array.
- C automatically scales pointer arithmetic so that it works correctly, by incrementing/decrementing by the correct number of bytes.
- If we create an array and a pointer to it's first element
  `char array[5];`
  `char *array_ptr = &array[0];`
  we can then refer to the nth element of the array `array[n]` as `*(array_ptr+n)`.
- The brackets () are important; `(*array_ptr)+n` is the same as `array[0]+n`, not `array[n]`.
- C provides a shorthand for dealing with arrays:
  `array_ptr = array;` instead of `array_ptr = &array[0];`

Pointers and Arrays - Example

Scanning an array using array index and pointer increment in comparison (`ptr2.c, ptr3.c`).

```
#include <stdio.h>

int array[] = {9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("%d elements before zero\n",
                  index);
    return (0);
}
```

```
#include <stdio.h>

int array[] = {9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("%d elements before zero\n",
                  array_ptr - array);
    return (0);
}
```

The index operation required for the `while` loop on the left side takes longer than the pointer dereference on the right side.

Passing Arrays to Functions

- You may want to pass an array to a function. If you do so, C will automatically change the array into a pointer - because you can only hand single values over to a function. In this case the address of the first element of the array.
- If you want your function to know how many elements the array has, you may want to pass on the number of elements as a second (integer) variable.

  ```
  int some_function(int *data_ptr, int nelements);
  ```

- If you pass a single variable to a function you only pass a copy of the variable's value - so the original variable cannot be changed. This is different if you pass an array. Because you are passing it as a pointer, the code in the function is working with the actual array elements.

Passing Arrays to Functions - Example

Passing an array to a function and initialising it (`init-a.c`).

```
#define MAX 10

void init_array_1(int data[])
{
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
```

```
int main()
{
    int  array[MAX];

    void init_array_1();
    void init_array_2();

    /* 4 ways of initializing
                    the array */

    init_array_1(array);
    init_array_1(&array[0]);
    init_array_1(&array);
    init_array_2(array);

    return (0);
}
```

When passing an array to a function, C will automatically change the array into a pointer. In fact, C will issue a warning if you put a & before the array, i.e. as in version 3.

Pointers-to-Pointers

- Pointers are variables whose value is an address.
- According to the type of variable at the address, we can have e.g. a pointer-to-integer or a pointer-to-float.
- Like any other variable, also pointers are stored at a specific place in the computer's memory - they also have an address.
- We can therefore define a variable that has as it's value the address of a pointer - a pointer-to-pointer.

```
int x = 12;                /* x is an integer variable with value 12 */
int *x_ptr = &x;           /* x_ptr is a pointer to the integer x    */
int **ptr_to_ptr = &x_ptr; /* ptr_to_ptr is a pointer to a pointer   */
                           /* to type int     */
```

- Note the double indirection operator (**) when declaring the pointer-to-pointer.
- There is in principle no limit to the level of multiple indirection - you could point and point and point and point. But there is no real advantage to anything with more than two levels of pointing.

Arrays of Pointers

- Because pointers are one of C’s data types you can declare and use arrays of pointers.
- One possible use for this is an array of pointers to type `char`, aka an array of strings.
- It’s much easier to pass an array of pointers to a function than to pass a series of strings.
- In fact, you are of course passing a pointer to the function - this pointer is a pointer-to-pointer.
- This actually is one of the main applications of multiple indirection.

Arrays of Pointers - Example

Passing an array of pointers to a function (p2p.c).

```
#include <stdio.h>

void print_message(char *ptr_array[], int n) {
  int count;
  for (count = 0; count < n; count++)
    {
      printf("%s ", ptr_array[count]);
    }
  printf("\n");
}

int main() {
  char *message[9] = {"Dennis", "Ritchie", "designed", "the", "C",
                      "language", "in", "the", "1970s"};
  print_message(message, 9);
  return (0);
}
```

Output:

```
kaiser@npl03:~/linuxc/oreilly/pracc/p2p> p2p
Dennis Ritchie designed the C language in the 1970s
```

Pointers to Functions

- A pointer holds an address. This may be the address of a variable or the address of the first element of an array - the cases we looked at so far.
- However, a pointer may also hold the starting address of a function, i.e. the address where the function is stored in memory.
- Pointers to functions provide another way to call functions.
- The general form (and some examples) for the declaration of a pointer to a function:

```
type (*ptr_to_func)(parameter_list);
int (*func1)(int x);
char (*func2)(char *p[]);
```

- The parentheses around the function name are necessary because of the relatively low precedence of the indirection operator (*).

How not to Use Pointers

- Pointers in themselves are already confusing enough. However, the combination of pointers with increment and decrement operators (++/–) can easily make matters worse.
- Have a look at these lines of code as examples of how not to use pointers:

```
data_ptr = &array[0]; /* Point to the first element of the array.      */
value = *data_ptr++;  /* Get element #0, data_ptr points to element #1.*/
value = *++data_ptr;  /* Get element #2, data_ptr points to element #2.*/
value = ++*data_ptr;  /* Increment element #2, return it’s value       */
                      /* Leave data_ptr alone.                         */
```

- While it’s not impossible to figure out what is going on, it is not the point of programming to provide challenging little puzzles to the programmers that have to look at your code in the future.
- Here is another bad example:

```
void copy_string(char *p, char *q)
{
   while (*p++ = *q++);
}
```

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 8

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- File I/O
- Graphics with C

`http://www.physics.gla.ac.uk/~kaiser/`

Streams

- Input and Output, aka I/O, in C is based on the concept of the stream.
- A stream is a sequence of characters, more precisely, a sequence of bytes of data.
- The advantage of streams is that I/O programming becomes device independent. Programmers don't have to write special I/O functions for each device (keyboard, disk, etc.). The program 'sees' I/O as a continuous stream of bytes, no matter where it's coming from or going to.
- Every C stream is connected to a file, where file does not (only) refer to a disk file. Rather, it's an intermediate step between the stream that your program deals with and the actual physical device used for I/O.
- We will in the following typically not distinguish between streams and files and just use 'file'. And we will deal only with disk files, but keep in mind that a 'file' could also be e.g. a printer.

Standard I/O

- There are three predefined streams, also referred to as the standard I/O files:

Name	Stream	Device
stdin	standard input	keyboard
stdout	standard output	screen
stderror	standard error	screen

- The C library contains a large number of routines for manipulating files. The declarations for the structures and functions used by the file functions are e stored in the standard include file `<stdio.h>`.
- So, before doing anything with files, you must include the line

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

  at the beginning of the program.

FILEs

- Files are handled using the data type `FILE`, which is defined in `<stdio.h>` together with the rest of the I/O facilities.
- The declaration for a file, actually for a file variable is:

  `FILE *file-variable; /* comment */`

- Example:

```
FILE *in_file;   /* file containing the input data */
```

- To open a file you must create a pointer to type `FILE` and use the `fopen` function. The prototype of this function is located in `stdio.h` and reads

```
FILE *fopen(const char *filename, const char *mode)
```

  (ok, have a look, it's a bit more complicated...)

- For example

```
FILE *in_file;
in_file = fopen("data.txt","r");
```

  opens the file 'data.txt' in the present working directory for read access.

Opening and Closing Files

- When opening a file with `fopen` the `name` that you use depends on the operating system under which you are working.
- Under Linux, the rules for Linux filenames apply. This means e.g. that you can open the file 'data.txt' in the present working directory or the file '/usr/local/stdio.h' using the absolute path of that file.
- The mode specifies if you want to open the file for reading, writing etc:

Mode	Meaning
`r`	reading
`w`	writing
`a`	appending
`r+`	reading and writing, overwrites from the start
`w+`	reading and writing, if the file exists, it's overwritten
`a+`	reading and appending

Opening and Closing Files cont.

- If you fopen a file for write access and it doesn’t yet exist, a file with the specified name will be created. If a file with the same name already exists, it will be overwritten. (That is, if you have the permission to do so.)
- The function `fopen` returns a file handle that will be used in subsequent I/O operations.
- If there is an I/O error, e.g. if you try to open a file for reading that doesn’t exist, the value NULL is returned. This can be used in an `if`-statement to check if the file was opened correctly:

```
in_file = fopen("input.txt", "r");
if (in_file == "NULL")
   fprintf(stderr, "Error: unable to open ’input.txt’\n");
   exit(8);
```

- The function `fclose` will close the file, e.g.

```
fclose(in_file);
```

File Access - Example

Count the number of characters in `input.txt` (`copy.c`).

```
#include <stdio.h>
const char FILE_NAME[] = "input.txt";
#include <stdlib.h>

int main(){
    int  count = 0;
    FILE *in_file;
    int ch;
    in_file = fopen(FILE_NAME, "r");
    if (in_file == NULL) {
        printf("Cannot open %s\n", FILE_NAME);
        exit(8);
    }
    while (1) {
        ch = fgetc(in_file);
        if (ch == EOF)
            break;
        ++count;
    }
    printf("Number of characters in
       %s is %d\n", FILE_NAME, count);
    fclose(in_file);
    return (0);
}
```

- open file for read access
- exits if file does not exist or cannot be read
- `fgetc` gets a single character from the file
- more about `fgetc` later

ASCII and Binary Files

- There are two types of files: ASCII and Binary files.
- ASCII, the American Standard Code for Information Interchange, encodes characters as (7-bit) integer numbers. Terminals, keyboards and printers deal with character data.
- Computers, i.e. the CPU and the memory, work on binary data. When reading numbers from an ASCII file, the character data must be processed through a conversion routine like `sscanf`. This takes (some) time. Binary files require no conversion.
- To access a binary file you have to add a ’b’ to the mode of `fopen`.

ASCII	Binary
reading requires conversion	no conversion required
human readable	not human readable
portable	(mostly) not portable
small/medium amounts of data	large amounts of data

Different Types of I/O

- There are three different ways to write data to a disk file: formatted output, character output and direct (binary) output. There are of course the corresponding ways of reading them in.

Type of I/O	I/O statements	should be used for
Formatted	`fprintf` `fscanf`	files with text and numerical data to be read in by spreadsheets, databases or data analysis programs
Character	`fputc` `fgetc` `fputs` `fgets`	text files, to be read e.g. by word processors
Direct	`fwrite` `fread`	binary files, best way to save for later use by a C program

Character I/O

- `int fgetc(FILE *file_ptr)`
  returns a single character from the file, but has return type integer.
- Successive calls get successive characters.
- If no more data exist, `fgetc` returns the constant `EOF`
- `EOF` defined in `<stdio.h>` as -1 (which is why the return type has to be `int`).
- `int fputc(int c, FILE *file_ptr)`
  writes a single character to the file.
- `char fgets(char *s, int n, FILE *file_ptr)`
  reads in a string of n characters. We have already met `fgets` before: `file_ptr` can also be `stdin` to read a string from the keyboard.
- `char fputs(const char *s, FILE *file_ptr)`
  writes out a string.

`fprintf` - Formatted File Output

- The function `fprintf` converts data to characters and writes them in a defined format to a file. The general form of `fprintf` is:

  ```
  count = fprintf(file, format, parameter-1,
                      parameter-2,...);
  ```

  where
  - `count` is the return value of `fprintf`: the number of characters sent or -1 if an error occurred,
  - `format` is a format statement of the same type as used with `printf`.
- `fprintf` to `stdout` is identical to `printf`.
- `printf` and `fprintf` have another sister function `sprintf` that does the same for formatted writing to a string.
- Example:

  ```
  fprintf(file_ptr, "The year was %d", year);
  ```

Formatted File Output - Example

Write the table from `band.c` to file (`band_plot.c`).

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

int main()
{
  int i, j;
  FILE  *f_ptr;    /* file handle */

  f_ptr = fopen("binary_and.txt", "w");

  fprintf(f_ptr, "  &");
  for (j=0; j<16; j++)
        {
          fprintf(f_ptr, "%3d", j);
        }
  fprintf(f_ptr, "\n\n");

  for (i=0; i<16; i++)
    {
      fprintf(f_ptr, "%3d", i);
      for (j=0; j<16; j++)
        {
          fprintf(f_ptr, "%3d", i&j);
        }
      fprintf(f_ptr, "\n");
    }

  fclose(f_ptr);  /* close file */
  return(0);
}
```

The only difference between `band.c` and `band_file.c` is the introduction of the file handle and the `fprintf` instead of `printf`.

Formatted File Input - `fscanf`

- `fscanf` is similar to `scanf`, except that the file (represented by the file handle) has to be specified, while `scanf` assumes `stdin` as input stream.
- The syntax for `fscanf` is

```
fscanf(file, format, &parameter-1, ...);
```

  where the return value of the function `fscanf` is the number of parameters that were read in successfully.
- For example

```
fscanf(f_ptr, "%f %f", &para1, &para2);
```

- Instead of using `fscanf`, we can also use the combination of `fgets` and `sscanf` already introduced in C-lecture 3.

Binary I/O

- Binary I/O is accomplished through the routines `fread` and `fwrite`. The syntax for both commands is similar:

```
  read_size = fread(data_ptr, 1, size, file);
 write_size = fwrite(data_ptr, 1, size, file);
```

  - where `read_size/write_size` is the size of the data that was read/written,
  - `data_ptr` is the pointer to the data to be read/written. This pointer must be cast to a character pointer `(char *)`.
  - size is the size of the data to be read/written in byte,
  - file is the input file (the file pointer).
- `fread` and `fwrite` are originally meant for arrays of objects, so the second parameter (that we set to 1) is the size of an object in bytes and the third parameter is the number of objects in the array.

Binary I/O - Example

Plot a sinus function to screen and postscript file (`sin_plot.c`).

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

#define SIZE 10

int main()
{
  int count, array1[SIZE], array2[SIZE];
  FILE *fp;

  /* Initialise array1 */
  for (count = 0; count < SIZE; count++)
    array1[count] = 2*count;

  /* Save array1 to the binary file */
  fp = fopen("direct.txt", "wb");
  fwrite(array1, sizeof(int), SIZE, fp);
  fclose(fp);

  /* Read data from binary file into array2 */
  fp = fopen("direct.txt", "rb");
  fread(array2, sizeof(int), SIZE, fp);
  fclose(fp);

  /* Display both arrays */
  /* to show they are the same */
  for (count = 0; count < SIZE;
                           count++)
    printf("%d\t%d\n",
     array1[count], array2[count]);

  return(0);
}
```

Output:

```
 0       0
 2       2
 4       4
 6       6
 8       8
10      10
12      12
14      14
16      16
18      18
20      20
```

Graphics with C

- Graphics capabilities are not part of Standard C. Sorry, but that's just the way it is.
- Some versions of C, e.g. Turbo C, have libraries like `graphics.h` that come as part of the distribution, but they are non-standard.
- However, there are libraries available for almost anything, and a lot of quite fancy graphics programs are actually written in C. Just that drawing a line on the screen is not trivial and requires almost the same work in setting up the right infrastructure as more complicated applications.
- Let's have a look at some of the different applications that you actually may want to use graphics for:
    - Windows programming and Graphical User Interfaces,
    - drawing/plotting data,
    - producing picture files as output, e.g. .gif or .pdf files.
- Unfortunately, graphics is mostly beyond the scope of this

Windows and GUIs with C

- X Window System
  `http://www.xfree86.org/` and `http://www.x.org/`
  Linux uses the X Window system. You also can find instructions in X/Motif programming under C on the web, but it's a bit outdated.
- Qt
  `http://www.trolltech.com/`
  Actually a C++ development tool for X Window programming, available also for Linux. KDevelop uses Qt.
- GGI - General Graphics Interface
  `http://www.ggi-project.org/`
  A project that aims at a graphics system that works everywhere. The GGI project provides various libraries, the most important ones being LibGGI and LibGII.

Drawing/Plotting Data with `gnuplot`

- `gnuplot` is one of many data analysis and function plot programs.
- It's advantage is that it is available free under Linux and that there is a library (gnuplot_i) that provides an interface for C to gnuplot. http://ndevilla.free.fr/gnuplot/gnuplot_i/
- `gnuplot` website: `http://www.gnuplot.info/`
  Tutorials can be found at
  `http://www.cs.uni.edu/Help/gnuplot/` and
  `http://www.duke.edu/~hpgavin/gnuplot.html`
- One nice feature of `gnuplot_i` is that `gnuplot` commands can be 'piped' from a C-program to `gnuplot`. There are very few additional commands to learn once one is familiar with `gnuplot`.

Basic Functionality of `gnuplot`

- Start `gnuplot` by simply typing `’gnuplot’`. You will get a few lines of text and a command-line interface with the prompt `’gnuplot>’`.
- To quit type `’exit’`.
- To plot a function, type `’plot sin(x)’`. A graphics window will open and the function will be displayed.
- `plot[x1:x2][y1:y2]<function>`
  will plot the specified function within the given x- and y-limits.
- `splot[x1:x2][y1:y2][z1:z2]<function>`
  will plot a function of 2 parameters within the given x-, y- and z-limits.
- `set isosamples x-rate, y-rate`
  sets the number of points in x and y, (`set samples x-rate` for a 1-d function.
- `replot` does just that after a change of parameters

Basic Functionality of `gnuplot` cont.

- `save "work.gnu"`
  saves the present settings and commands to file
- `load 'work.gnu'`
  loads a 'macro' that saves you from typing lots of lines every time. Comments in macros and data files are marked by `#`.
- `plot "<filename>" using x:y`
  Assuming `<filename>` is an ASCII file with data organised in columns separated only by spaces, this command will plot the specified columns (e.g. 1:3) versus one another.
- `help` accesses the built-in online help facility of gnuplot.
- `set output file.ps`
  `set terminal postscript`
  `replot`
  will allow you to plot to a postscript file instead of the screen.
- `set terminal x11`
  to get back to the screen.

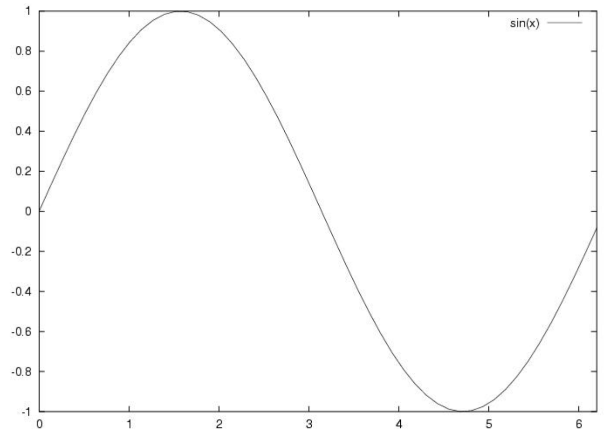
Using `gnuplot_i`

- `#include "gnuplot_i.h"` to include the `gnuplot_i` header file
- `gcc -o program program.c gnuplot_i.o`
  to compile, or better, use a `Makefile` and the flag `-I` to add the right directory for `gnuplot_i.h` to the path.
- `gnuplot_ctrl *h1;`
  `h1 = gnuplot_init() ;`
  opens a new gnuplot session, referenced by a handle of type (pointer to) `gnuplot_ctrl` and initialised by calling `gnuplot_init()`.
- `gnuplot_cmd(handle, "gnuplot_command")`
  'pipes' a `gnuplot` command from the C-program to `gnuplot`.
- For some commands there are C versions, see `gnuplot_i.h`.
- `gnuplot_close(h) ;` to close the session.

gnuplot_i - Example

Plot a sinus function to screen and postscript file (sin_plot.c).

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

#define SLEEP_LGTH  5
#define NPOINTS     50

int main(int argc, char *argv[])
{
    gnuplot_ctrl   *h1;
    double         x[NPOINTS] ;
    int            i ;

    /* Initialize gnuplot handle */
    h1 = gnuplot_init() ;

    /* Plot sinus to screen */
    printf("sine in points\n") ;
    gnuplot_setstyle(h1, "points") ;
    gnuplot_cmd(h1, "set samples 50");
    gnuplot_cmd(h1, "plot [0:6.2] sin(x)");
    sleep(SLEEP_LGTH) ;

    /* Plot sinus to postscript file */
    gnuplot_cmd(h1, "set terminal postscript");
    gnuplot_cmd(h1, "set output \"sinus.ps\"");
    gnuplot_cmd(h1, "replot");
    gnuplot_cmd(h1, "set terminal x11");

    /* close gnuplot handle */
    printf("\n\n") ;
    printf("*** end of gnuplot example\n") ;
    gnuplot_close(h1) ;
    return(0) ;
}
```



C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 9

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Command-Line Arguments
- Casting
- Advanced Types

`http://www.physics.gla.ac.uk/~kaiser/`

Command-Line Arguments

- You may have noticed that your typical Linux application may take one or several command line arguments, e.g. a filename and that it also often can be started with different flags. One example is the C-compiler itself: `gcc -o test test.c`. You also may have wondered how C is going to make this possible.
- This is the moment to reveal that the function `main` actually takes two arguments:

```
int main(int argc, char *argv[])
```

- The parameter `argc` is the number of arguments on the command line, including the name of the program. You don't enter it explicitly - C will count the arguments itself.
- The array `argv` contains the actual arguments.
- To remember the names you can use 'argument counter' (`argc`) and 'argument vector' (`argv`). These names are a convention that is almost always followed, not a part of the C specifications.

Command-Line Arguments - Example

Accessing command-line arguments (`c-line.c`).

```
#include <stdio.h>

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

  printf("Program name: %s\n", argv[0]);

  if (argc > 1)
    {
      for (count = 1; count < argc; count++)
        printf("Arg %d: %s\n", count, argv[count]);
    }
  else
    printf("No command line arguments entered.");

  return(0);
}
```

- The program takes the command-line arguments and prints them out again.
- This illustrates in a simple way how the command-line arguments can be accessed and used.

Parsing Command-Line Options

- Almost all Linux commands use a standard command-line format:

  `command options file1 file2 file3...`

- Also the options (or 'flags') have a standard format (well, two): The one form starts with a dash (`-`) and is usually a single letter that maybe followed by an argument to the option. Example: `gcc -o outfile`.
- The other form starts with a double-dash (`--`) usually followed by a word. Example `gcc --help`.
- The dash or double-dash is meant to help `parse` the command-line.
- In the following we'll consider a simple parser for single dash, single letter flags with no additional space to the argument, e.g. `-ooutfile`.

Parsing Command-Line Options cont.

- To cycle through the command-line options we can use a loop like `while ((argc > 1) && (argv[1][0] == '-')){}`
- At the end of the loop is the code

```
--argc;
++argv;
```

- This 'consumes' one argument: The number of arguments is decremented and the pointer to the first option is incremented, shifting the list to the left by one place.
- Character `0` of each argument is the dash (`-`), character `1` is the option character. So we can use the expression `switch (argv[1][1])` to decode the option.
- If the option has an argument, character 2 is the first character of the argument. If e.g. this is a filename we can use `out_file = &argv[1][2]` to assign the address of the begin of the output filename to a character pointer named `out_file`.

Parsing Command-Line Arguments - Example

Parsing Command-Line Arguments (`print.c`).

```
int main(int argc, char *argv[])
{
    program_name = argv[0];

    while ((argc > 1) && (argv[1][0] == '-')) {

        switch (argv[1][1]) {
            case 'v':
                verbose = 1;
                break;
            case 'o':
                out_file = &argv[1][2];
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
    return (0);
}
```

Casting

- Sometimes you will have to convert one type of variable to another type. This is done with a cast or typecast operation.
- The general syntax for a cast is

```
(type) expression
```

- This operation tells C to compute the value of the expression, and then convert it to the specified type.
- This is particularly useful when you work with integers and floating point numbers or for converting pointers from one type to another.
- Example:

```
int won, lost;
float ratio;
ratio = ((float) won)/((float) lost);
```

Structures

- If your are writing a program for an application in the 'real world' you will find that often information logically belongs together that is naturally represented by different variable types in C. Some examples could be:
  - Address book: names(strings), streets(strings), street numbers(integers), phonenumbers(integers) and postal codes (integers or strings, depending on the country).
  - Particle physics event: for each track a response for each detector (floats or integers), fit results from the reconstruction (floats), timestamps (integers, probably)
  - Warehouse inventory: names of items (strings), quantity in store (integer), price (float).
- So far we have only dealt with individual variables and with arrays of variables, where each element has the same type.
- Now we'll get to know a new data type, called a structure. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name.

Defining and Declaring Structures

- The general form of a structure definition is

```
struct structure-name {
    field-type field-name; /* comment */
    field-type field-name; /* comment */
    ......
} variable-name;
```

- For example, we want to define a bin to hold printer cables. The structure definition is

```
struct bin {
    char name[30];   /* name of the part */
    int quantity;    /* how many are in the bin */
    float price;     /* cost of a single part */
} printer_cable_bin;
```

- This defines a structure `bin` and also declares the variable `printer_cable_bin` as type `bin`.
- We can now also declare other variables of type `bin`:

```
struct bin terminal_cable_bin; /* Place to put terminal cables */
```

Defining and Declaring Structures cont.

- It is possible to declare a structure variable, without at the same time defining a data type. This is known as an anonymous structure:

```
struct {
    char name[30];   /* name of the part */
    int quantity;    /* how many are in the bin */
    float price;     /* cost of a single part */
} printer_cable_bin;
```

- It is also possible to define a structure data type without immediately declaring a structure variable:

```
struct coord{
    int x;  /* x coordinate */
    int y;  /* y coordinate */
    int z;  /* z coordinate */
};
```

- While it is also possible (and the code will compile) to create an anonymous structure without declaring a variable, this would be a completely pointless exercise...

Accessing Structure Members

- Structure members are accessed using the structure member operator '.'. We use the syntax `variable.field`.
- If we have a variable `printer_cable_bin` defined as above, we can assign a value to the field `cost` using

  `printer_cable_bin.cost = 12.95; /* Price in $*/`.
- And we can calculate the total value of the cables in the bin:

```
total_cost = printer_cable_bin.cost * printer_cable_bin.quantity;
```

- Structures may be initialised at declaration time by putting the list of elements in curly braces:

```
struct bin {
    char name[30];   /* name of the part */
    int quantity;    /* how many are in the bin */
    float price;     /* cost of a single part */
} printer_cable_bin = {
    "Printer Cables",
    100,
    12.95
};
```

Structures Containing Structures

- A C structure can contain any of C's data types. Naturally, it can also contain structures.
- Let's assume we have structure `coord`

```
struct coord {
    int x;
    int y;
};
```

- We can define a structure `box` and create an instance `mybox` of this structure by

```
struct box {
   struct coord topleft;
   struct coord bottomright;
} mybox;
```

- To access the actual data, we must now apply the membership operator twice:

```
mybox.topleft.x = 10;
```

Structures Containing Structures - Example

box structure containing `coord` structures (`structs.c`).

```
#include <stdio.h>

int width, height, area;

struct coord {
  int x;
  int y;
} topleft, bottomright;

struct box {
  struct coord topleft;
  struct coord bottomright;
} mybox;
```

```
int main()
{
  printf("Top left x coordinate:");
  scanf("%d", &mybox.topleft.x);
  printf("Top left y coordinate:");
  scanf("%d", &mybox.topleft.y);
  printf("Bottom right x coordinate:");
  scanf("%d", &mybox.bottomright.x);
  printf("Bottom right y coordinate:");
  scanf("%d", &mybox.bottomright.y);

  width = mybox.bottomright.x - mybox.topleft.x;
  height = mybox.bottomright.y - mybox.topleft.y;
  area = width * height;

  printf("\nThe area is %d units.\n\n", area);

  return(0);
}
```

Output:

```
Top left x coordinate:22
Top left y coordinate:33
Bottom right x coordinate:444
Bottom right y coordinate:555

The area is 220284 units.
```

Arrays of Structures

- We have already seen, or implicitly assumed that structures can have arrays as members (when we put `char name[30]` into a structure). Can we also have an array with elements of type structure ? Yes. We can.
- Let's assume that we want to maintain a list of phone numbers. We might define a structure like

```
struct entry {
   char fname[20];   /* First name   */
   char lname[20];   /* Last  name   */
   char phone[20];   /* Phone number */
};
```

- Now we can define our phone list as an array of structures:

```
struct entry list[1000]; /* phone list */
```

- Now `list[1]` refers to a structure, `list[1].fname` to a member of that structure (that itself is an array) and `list[1].fname[2]` to the third letter of the first name of the second entry in our list. (C starts to count at zero !).

Arrays of Structures - Example

Enter names and numbers and print them again (a_structs.c).

```
#include <stdio.h>

struct entry {
  char fname[20];
  char lname[20];
  char phone[20];
};

struct entry list[4];

int main()
{
  int i;
  for (i = 0; i < 4; i++)
    {
      printf("First Name: ");
      scanf("%s", list[i].fname);
      printf("Last Name: ");
      scanf("%s", list[i].lname);
      printf("Phone Number: ");
      scanf("%s", list[i].phone);
    }
  for (i = 0; i < 4; i++)
    {
      printf("Name: %s %s",
             list[i].fname,
             list[i].lname);
      printf("\t\tPhone: %s\n",
             list[i].phone);
    }
  return(0);
}
```

- Please remember: there should be comments here everywhere and there only left out so that the code fist onto the page.

Pointers as Structure Members

- We have already said that structures can have members of any type, so it is only logical that they can also have members that are pointers to any type.
- For example

  ```
  struct data {
     int *value1;
     int *value2;
  } first;
  ```

  creates an instance `first` of the structure `data` with two pointers to int as members.
- Assuming that the integer variables `adc1` and `adc2` have been defined we can initialize the pointers as

  ```
  first.value1 = &adc1;
  first.value2 = &adc2;
  ```
- Now we can use the indirection operator (`*`) in the same way as for regular pointers, i.e. `*first.value1` evaluates to the value of `value1`.

Pointers to Structures

- As you probably already suspected, we can also declare pointers to structures.
- This is in particular practical if we want to pass a structure to a function. (And we remember that this implies that the values in the structure can be changed by the function.)
- Let's assume we have the structure `coord` again:

```
struct coord {
   int x;
   int y;
} point1;
```

- Then we can declare a pointer to the structure `coord` by

```
struct coord *point_ptr;
```

- To initialize the pointer to `point1` we use

```
point_ptr = &point1;
```

Pointers to Structures cont.

- There are now three ways to access a structure member:
- `point1.x` - the structure name.
- `(*point_ptr).x` - a pointer to the structure with the indirection operator (*). The brackets () are necessary because the precedence of the membership operator '.' is higher than that of the dereference operator `*`. (Precedence is something we have mostly ignored so far, and we'll use brackets whenever necessary and keep it that way.)
- `point_ptr->x` - a pointer with the indirect membership operator `->`.
- The structure pointer operator `->` is also known as the indirect membership operator.
- Now think about a structure with a member that is a pointer to the same structure. Thankfully that's beyond the scope of this course.

Unions

- While a structure is used to define a data type with several fields where each field takes up a separate storage location, a `union` defines a single location that can be given many different field names.
- Example of a `union`:

```
union value {
   long int i_value;  /* integer version of value */
   float f_value;     /* float version of value   */
} data;
```

- Because all fields occupy the same space, assigning a value to `f_value` wipes out a possibly existing value of `i_value`.
- Using a `union`:

```
data.f_value = 5.0;
data.i_value = 3;   /* data.f_value overwritten */
i = data.i_value;   /* legal, */
f = data.f_value;   /* illegal, will generate unexpected results */
data.f_value = 5.5; /* put something into f_value / clobber i_value */
i = data.i_value;   /* illegal, will generate unexpected results */
```

The `typedef` Statement

- C allows the programmer to define his/her own variable types through the `typedef` command. The general form is

  `typedef type-declaration`

- For example, the declaration typedef int count; defines a new type `count` that is identical with `int`. Therefore the declaration `count flag;` is identical to `int flag;`.
- One frequent use of `typedef` is the definition of a new structure as a variable type.
- Example:

```
struct complex_struct {
   double real;
   double imag;
};
typedef struct complex struct complex;

complex voltage1 = {3.5, 1.2};
```

C Programming under Linux

P2T Course, Semester 1, 2004–5
C Lecture 10

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Memory Allocation
- Random Numbers
- Bits and Pieces
- Famous Last Words

`http://www.physics.gla.ac.uk/~kaiser/`

Dynamic Memory Allocation

- So far we have always declared all the variables we would need during the running of the program right at the beginning. This can also be referred to as static memory allocation.
- What happens if the program finds out that it needs a bit more storage space than we allocated ? Let's say we have an array that we are going to fill as the result of a calculation. What if we don't know from the beginning how long it should be ?
- We can of course allocate just a lot of space, i.e. make the array huge from the beginning. However, this wastes a lot of space and once it's not one array but 100 arrays, we may run out of space.
- The solution lies in dynamic memory allocation. The C library contains functions that can allocate storage space at runtime.
- This allows the program to react to demands from the outside and makes it more flexible, e.g. also when it communicates with other programs while running.

Allocating Memory with `malloc`

- The most basic memory allocation function is `malloc`.
- The function prototype for `malloc` is

```
void *malloc(unsigned int);
```

- The argument of malloc is the number of bytes to allocate. If malloc runs out of memory it returns a null pointer.
- `malloc` returns a pointer to type `void`, i.e. a generic pointer that can point to anything, e.g. a string or a structure. What `malloc` doesn't give us is a normal variable with a variable name.
- The returned pointer can be cast to a specific type, but implicit conversion on assignment to a previously defined pointer is allowed in Standard C.
- In C++ the type cast is required.
- In old books you may find that `malloc` returns a pointer to character. That's outdated.

Allocating Memory with `malloc` - Example

- Suppose we are working on a data base program, using a structure called `person`:

```
struct person {
    char name[30];     /* name of the person       */
    char address[30];  /* where he/she lives       */
    int age;           /* how old he/she is        */
    float height;      /* how tall is he/she in cm */
}
```

- Instead of using an array of structures `person` we can now use `malloc` to create a `person` when we need one:

```
/* Pointer to a new person structure to be allocated from the heap */
struct person *new_item_ptr;
new_item_ptr = malloc(sizeof(struct person));
```

- We don't even have to know the size of a `person` in bytes, the `sizeof` function will figure this out for us.
- The heap is the total of the available memory space, which is large but not infinite. When malloc runs out of space it returns a NULL pointer - and one should check if `malloc` was successful.

De-allocating Memory with `free`

- If we keep allocating memory we will eventually run out. That's why there also has to be a function that de-allocates or frees the memory again. This function is called `free`.
- The function prototype is void free(void *ptr);
- In addition the pointer should be set to `NULL`, but doesn't have to. However, it does save us from trying to use freed memory.
- Here is an example using `malloc` to get memory and `free` to dispose of it:

```
cont int DATA_SIZE = (16 * 1024) ; /* Number of bytes in the  buffer */
void copy(void)
{
    char *data_ptr;   /* pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);   /* get the buffer */
    /*
     * use the buffer to do something, i.e. copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

Other Memory Allocation Functions

- The memory allocation functions are, among others, declared by the header file `stdlib.h`. So if you want to use any of them you have to include it. The other allocation functions are:
  - `calloc(count, size)` allocates a region of memory large enough for an array of `count` elements, each of `size` bytes. The region of memory is declared bitwise to zero.
  - `realloc` changes the size of an already existing pointer while preserving it's contents. If necessary, the contents are copied to a new memory region; a pointer to the (possibly new) memory region is returned.
  - `mlalloc` - same as `malloc` but parameter is `unsigned long`
  - `clalloc` - same as `calloc` but parameters are `unsigned long`
  - `cfree` frees memory allocated by `calloc` or `clalloc`.
- For details please have a look at `man` pages, `info` pages, the web or a C book.

Random Numbers

- On occasion you will want to be able to generate random numbers, lots of them.
- A typical application in particle and nuclear physics are so-called Monte Carlo simulations that use large numbers of simulated physics processes with a statistical distribution of parameters. They are important to model the performance of a detector or to understand the background under the peak that signifies a new particle.
- C offers a function in stdlib.h that can be used for this purpose.
- The function prototype is simply

  ```
  int rand(void);
  ```

  It returns a pseudo-random integer number between 0 and `RAND_MAX`.
- `RAND_MAX` is defined in `stdlib.h` as 2147483647.

Random Numbers cont.

- The numbers returned by `rand` are pseudo-random, meaning that the distribution is flat, i.e. the probability for any given interval of the same size is about equal.
- However, the sequence is deterministic, i.e. repeated calls to `rand` in one program give different numbers, but if the program writes out 1000 'random' numbers they will always be the same in the same order.
- The function `srand` can be used to change the so-called seed, i.e. the input value of the algorithm that works behind `rand`.
- The function prototype here is

```
void srand(unsigned seed);
```

- The same argument in `srand` will produce the same chain of values coming from `rand`. So for 'really' random numbers one can use `rand` to produce a seed for `srand` and then produce a large set of numbers.

Integer and Float Random Numbers - Example

Integer and floating point random numbers (`random.c`).

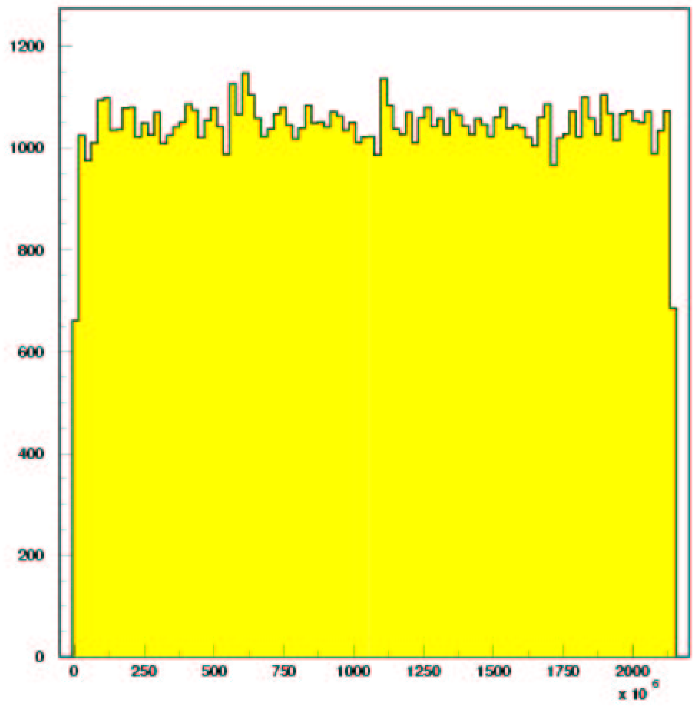
```
/* integer random numbers */

f_ptr = fopen("random1.dat", "w");

for (i=0; i<100000; i++) {
  fprintf(f_ptr, "%d\n", rand());
}
fclose(f_ptr);  /* close file */
```

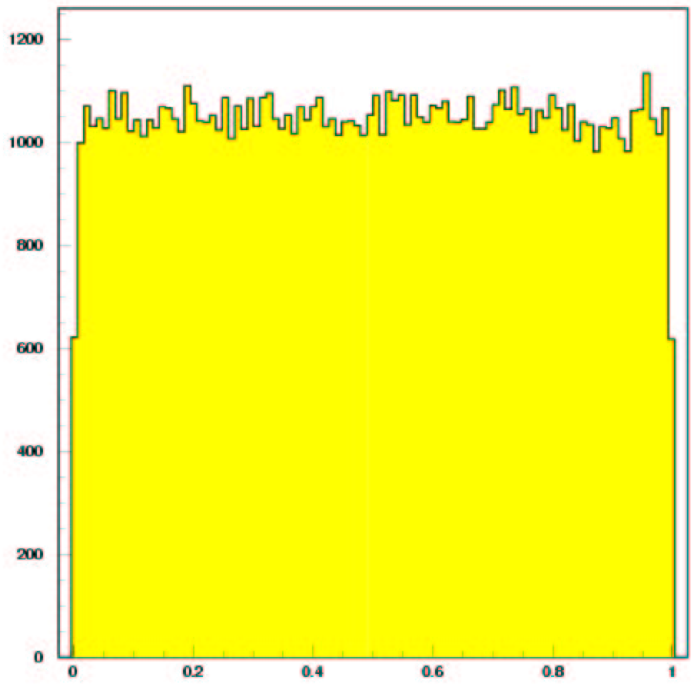
```
/* random numbers between 0 and 1 */
/* flat distribution              */

f_ptr = fopen("random2.dat", "w");

for (i=0; i<100000; i++) {
  number = (float) rand()/RAND_MAX;
  fprintf(f_ptr, "%f\n", number);
}
fclose(f_ptr);  /* close file */
```

- The standard output of `rand` consists of integers. However, you may want to have floating point values between 0 and 1 instead.
- This can be constructed in a simple fashion (see example).
- `stdlib.h` also contains additional functions that do this for you, but `rand` is Standard C.

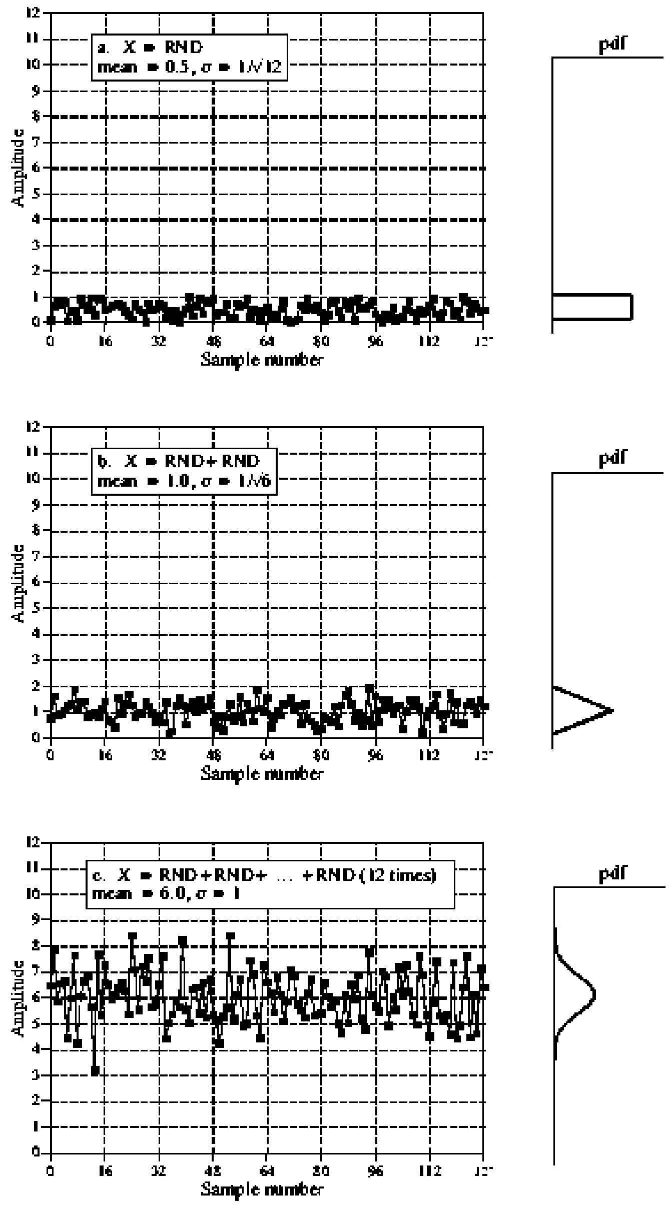
Integer and Float Random Numbers - Example

Integer and floating point random numbers (`random.c`), histograms of random number output.





Gaussian Random Numbers

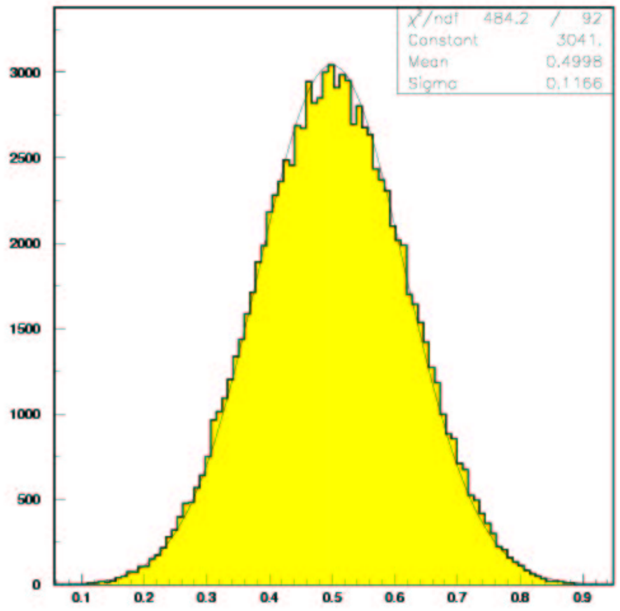


- The key to a Gaussian distribution is that it can be created from a flat distribution by sampling.
- Already the sum of 6 random numbers from a flat distribution is very close to a Gaussian.

Gaussian Random Numbers - Example

Gaussian random numbers (`random.c`), histogram of random number output with Gaussian fit.

```
/* random numbers between 0 and 1   */
/* Gaussian distribution around 0.5 */

f_ptr = fopen("random3.dat", "w");

for (i=0; i<100000; i++) {
  number = 0.0;
  for (j=0; j<6; j++) {
    number += (float) rand()/RAND_MAX;
  }
  fprintf(f_ptr, "%f\n", number/6.0);
}
fclose(f_ptr);  /* close file */
```



(The histograms on this slide and the previous one were produced using PAW, a data analysis package from CERN which is available for Linux, but unfortunately based on FORTRAN. The successor is called root and based on C++.)

The Ternary Operator '?:'

- We have seen unary operators (like '++') that act on one variable and binary operators (like '+') that require two.
- There is also a single ternary operator, the operator '?:'. It's use is similar to an `if/then/else` construction, but in contrast to `if/then/else` '?:' can be used inside of an expression.
- The general form is

  (expression) ? value1 : value2

- For example, the following construct assigns to `amount_owed` the value of the balance or zero, depending on the amount of the balance:

  ```
  amount_owed = (balance < 0) ? 0 : balance;
  ```

- The following macro returns the minimum of it's two arguments:

  ```
  #define min(x,y) ((x) < (y) ? (x) : (y))
  ```

The do/while Statement

- The do/while statement has the following syntax:

```
do {
      statement;
      statement;
} while (expression);
```

- The program will loop, test the expression, and stop if the expression is false (0).
- This construct will always execute at least once.
- It’s not frequently used in C programming. Most programmers prefer to use a while/break combination.

The Comma (,) Operator

- The comma operator (,) can be used to group statements, for example instead of

```
if (total < 0) {
   printf("You owe nothing\n");
   total = 0;
}
```

  we cam also write

```
if (total < 0)
   printf("You owe nothing\n"), total = 0;
```

- In most cases curly braces {} should be used instead. The only places where the comma operator is useful is for the declaration of a couple of simple variables in one line

```
int i, j, k;  /* a couple of counters */
```

  and in `for` statements:

```
for (two = 0, three = 0;
     two < 10;
     two += 2, three +=3)
         printf("%d %d\n", two, three);
```

Finally: The `goto` Statement

- You thought perhaps that `goto` has been banished to forever live only in damp dungeons of BASIC. You were wrong. It exists, but you don't every have to use it.
- For those infrequent times when you actually want to use a `goto`, the correct syntax is

  `goto label;`

  where `label` is a statement label.
- Example:

```
for (x = 0; x < X_LIMIT; x++) {
    for (y = 0; y < Y_LIMIT; y++) {
       if (data[x][y] == 0)
          goto found;
    }
}
printf("Not found\n");
exit(8);

found:
   printf("Found at (%d,%d)\n", x, y);
```

Things not to do

- '=' is not the same as '==' ! Don't mix them up !
- C starts counting at '0' not at '1' ! Stick to this, don't start at 1 !
- `matrix[10,12]` is not the correct C syntax !
- Don't forget the `\0` at the end of a string !
- Don't use 8 spaces at the begin of a line in a Makefile instead of a tab !
- Never put an assignment inside another statement !
- Don't make a block of code inside or a function longer than a few (about 3) pages !

Things to do

- Use `++` and `--` on lines by themselves.
- One variable declaration per line.
- Use comments. Lots of them. Useful ones.
- Make variable names explicit. ’total’ is better than ’t’.
- Always put a `default:` case into a `switch` statement.
- Use `const` instead of #define wherever possible.
- Put () around each argument of a preprocessor macro.
- Generally use enough () if you don’t know the precedence rules. (And you haven’t learned them here...).
- Use a Makefile.
- Make a backup of your work. Then make another one. If it’s important make one more. For my thesis I made one on another disk, one on another machine, one on another continent and one on tape - every evening.