Introduction to C and C^{++}
A PHY201 Reference Material

Aleksandar Donev

October 1999
Physics Department
Michigan State University

Abstract

This handout should serve you as an introduction to C and object-oriented programming in C^{++} and FORTRAN 90/95. The material follows much the same format as the previous *Inroduction to FORTRAN*.

Here is a list of web sites leading to some very useful on-line courses. Use these links to learn more about C and object-oriented programming and to find out details not present in this handout:

- A very nice introduction to C:
  http://math.nmu.edu/Dept/web/courseware/c/cstart.htm
- A higly detailed course in C:
  http://www.le.ac.uk/cc/iss/tutorials/cprog/cccc.html
- A very pedagogical introduction to C and object-oriented programming in C^{++}:
  http://devcentral.iftech.com/Learning/tutorials/subcpp.asp
- An overview of object-oriented programming in FORTRAN 90/95:
  http://www.cs.rpi.edu/~szymansk/oof90.html
- And our web-site (still under development):
  http://www.pa.msu.edu/~donev/PHY201

Contents

Chapter 1

Introduction to C

In this chapter we give a brief overview of some of the important feautures of the C programming language in the ANSI standard. It is assumed that you are already familiar with programming in FORTRAN, so that more time will be spent on the unique and important (thus more advanced) features of C. We begin as usual with a description of the C program.

1.1 Program Layout

There are several important things concerning the structure of a C program:

- Most commands end with a semicolon ;.
- C is case-sensitive. All commands are in lower-case letters. Also, avoid using upper-case names for variables that are not processor constants.
- Use a new line for each command and enclose program units using curly braces `{}`.
- Enclose comments between `/*` and `*/`. Multi-lined comments are thus possible and require only two comment delimiters.

In fact, the actual program layout can have quite a lot of "freedom of expression" so long as the above guidelines are observed. It is worth mentioning that C does not have a stiff syntax and the compilers allow the programmer a great deal of flexibility. This is good for more advanced programmers, but means that you should make C programs with more caution in the beginning.

The structure of a C program looks like:

```
# Preprocessor statement
Declaration of global variables and function prototypes
main(command-line arguments)
{
```

```
Declaration of variables
Body of program...
}
Body of auxilliary functions
```

One thing warrants explanation. In C, the main program is itself a function (a subroutine) called `main`. This function is in most respects equivalent to other functions with the difference that the execution starts from it and that it takes arguments from the command line. We will not use command-line arguments and results, so all of our main programs will begin with `main()`.

1.2 Variables

The data types in C are similar to those in FORTRAN, with a few differences. It should be stressed that C is very type-oriented, so that care must be taken about the type of all variables involved in expressions and function argument lists and proper conversions done where needed.

1.2.1 Variable Declaration

All variables must be declared at the beginning of the C program—either before the main program, in which case these variables are global and visible to all other functions, or just at the beginning of the main program. Like in FORTRAN, values can be assigned to variables during declaration. Declaration is done with:

[*modifier* **]** *Data_type { List_of_variables }*

The modifier tells more about the size of the data type. For example, `long` for floating point numbers means double precission. Several important modifiers apply to all data types. If a variable is used very often (for example a counter), we can store it in the registers (and not the memory) of the computer by including the modifier `register`. Also, a convenient and reliable way of making a variable global is by using the modifier `extern`, which means that memory for a given variable has already been allocated in an external source module (which needs to be linked to this one during compilation). Another way to make global variables is to declare the variable outside of all functions. To have a local variable retain its values between different calls of a function (SAVE in FORTRAN), we declare that variable with the modifier `static`.

For example, to declare an integer register variable called `counter` and initialize it to zero, we write

```
register int counter=0;
```

1.2.2 Data Types

The data types supported in C are:

- `int`–integer variables (2 bytes). The possible modifiers are `long` (4 bytes), `short` (1 byte), `unsigned` (if the integer is only positive).

- `float`–floating-point variables with single precision (2 bytes). The modifier `long` simply makes it a `double`.
- `double`–floating-point numbers with double precision (4 bytes)
- `char`-character or character strings (1 byte per character). In C these are almost equivalent to integers, with the standard ASCII code of the characters used for conversion.

Conversion can be done between any of the data-types by using:
(*New_data_type*)*Variable*
For example, `(int)number` represents the whole part of a number.
You can also make a new name for a data-type by using `typedef`. For example, after typing,

```
typedef int counter;
```

we can declare a new integer through:

```
counter i;
```

1.2.3 Enumerated Data Types

Variables that take values only from a fixed set of values can be defined conveniently as enumerated data types. For example, to define a variable type for a month, which can only have 12 different values, we type,

```
enum month {jan=1,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};
```

where now each three-letter acronim replaces a number, 1 for January, 2 for February, etc. (we could have given these values explicitly, but only the first number is sufficient). We can now declare a new varible of the type month, which can only get values from the set of 12 months, and assign it a value of November, which in fact translates to 11:

```
enum month my_birthday;
my_birthday=nov;
```

1.2.4 Structures

Structures are equivalent to FORTRAN's defined data-types and serve as a collection of different data types used in conjuction. For example, to define a structure for a variable representing a two dimensional vector we would type (note the semicolon after the curly brace since this is a declaration statement):

```
struct vector  {
float x_component, y_component;
};
```

We can later declare two new vectors and add them ($c = a + b$) with:

```
struct vector a, b, c;
c.x_component=a.x_component+b.x_component;
c.y_component=a.y_component+b.y_component;
```

where we accessed individual variables in the structure using a period (# in FORTRAN).

1.2.5 Arrays

Arrays are treated very similarly in C and FORTRAN 77. C does not have any advanced array feautures as does the 90 standard, and in fact power-array programming in C implies the use of pointers, which we will treat later (in fact arrays in C are just pointers to the first element of a memory block). It is important to note that C does not store any kind of size information in the array, and does not perform any checking about an index exceeding the array size. This implies that beginners must exercise greater care when using arrays. Also, it is worth knowing that C stores arrays row-wise, unlike FORTRAN. Finally, it is very important to notice that the indices in C *always* begin from 0, and not 1!

Arrays are declared as usual, by appending the dimensions in square brackets (note that a separate bracket is used for each dimension):

Data_type Array_name[*First_dimension*][*Second_dimension*]...[n^{th}*_dimension*]

For example, to declare an integer array of size $[50 \times 100]$ use:

```
int array[50][100];
```

We can also initialize an array at declaration, in which case we need not specify the dimension and use lists of values in (nested) curly braces:

```
int values[]={1,2,3,4,5,6,7,8,9,10};
```

1.2.6 Strings

Characters were already mentioned as one of the basic data types in C. They are enclosed in single quotes, like `'a','b','c'` etc. *Strings in C are arrays of characters that end with the null character* `'\0'`. Note that we actually use a backslash and a zero to write this string-end delimiter in a program. Strings in C are enclosed in quotes, which automatically places a (hidden) null delimiter at the end of the string (this is not displayed when the string is printed). Due to this very specific behaviour of C, most effective C programs declare strings as pointers to `char`. C supports many string functions through the header file `string.h` (see the on-line help).

For example, here are ways to declare a constant string that says "Hello" three times (note that there is no `&` in `scanf`. Think why after reading about pointers):

```
char name[10];
printf(''Enter your name \n'');
scanf(''%s'',name);
char first_way[] = {'H','e','l','l','o','\0'};
char second_way[]=''Hello'';
char *best_way=''Hello''; /* this uses a char pointer */
printf(''%s %s %s %s'',first_way,second_way,best_way,name);
```

1.3 Preprocessor Statements

Preprocessor statements are commands that are placed outside of the functions and that the compiler begins compilation from. There are several of these, but only a few of them are essential. It is important to remember that all preprocessor statements begin with the symbol # and do not end with a semicolon.

1.3.1 `#include`

A very important thing about C is that the language itself is very small; it has only about twenty commands for performing basic tasks. Most of the functions for doing more advanced operations are stored in so called *headerfiles*, which end with the extension `.h` and have libraries of utility functions stored in them.

Thus, C by itself does not even have the sine function. But the header file `math.h` comes with all C compilers. If you want to use mathematical functions in your program, the header file needs to be included via,

```
#include <math.h>
```

where the square brackets denote that the header file is in the compiler directory. When using your own header file enclose its name and path in quotes:

```
#include ''my_file.h''
```

Other important header files that you will almost always use in your programs are `stdio.h` (Standard Input and Output) and `stdlib.h` (Standard Library).

1.3.2 `#define`

This preprocessor statement is used to define constants. Constants should be named with names in uppercase letters. The constansts that are defined this way are substituted into the program before it is compiled. For example, we can define π via:

```
#define PI 3.14159
```

When we later type in the program,

```
radians=degrees/180*PI;
```

the compiler turns this into,

```
radians=degrees/180*3.14159;
```

and then compiles the program. The `define` preprocessor statement can be used to define more elaborate pre-compilation substitutions (called macros), but you are not likely to use these features.

1.4 Expressions

Expressions between variables are formed in C almost identically to FORTRAN. There are a few differences worth mentioning.

1.4.1 Arithmetic Operators

The substraction, addition, multiplication and division operators are the standard `+`,`-`,`*` and /. Oddly enough, C does not have an exponentiation operator. The header file `math.h` has a function `pow(base,exponent)` which can be used for that purpose, although this function works for floating-point numbers only, so for integers it is wise to write `i*i*i` instead of `pow(i,3)`.

Two arithmetic operators specific to C only are the increment and decrement unary operators, `++` and `--`. These increment the variable they are attached to in two different ways. If they stand in front of the variable, the variable is incremented by 1 before the expression is evaluated; if they are after the variable, the incrementation is performed after the expression is evaluated. For example, `y=++x` is equivalent to a succession of two statements, `x=x+1` and then `y=x`. On the other hand, `y=x++` is equivalent to a succession of two statements, `y=x` and then `x=x+1`. It can be quite tricky trying to decifer expressions involving these, so use them with caution (other operators, like `+=` or `-=` are likely to cause confusion at this point, so we don't mention them). The most famous example of a post-increment is the name C^{++}.

1.4.2 Relational Operators and Logical Expressions

Relational expressions are formed and evaluated in C the same way as in FORTRAN. The only (positive) difference is in the names. The relational operators in C have the standard notation `<`,`>`,`<=`,`>=` and `==` (do not forget the double equal for comparison as opposed to a single equal for assignment!). It is also important to note that C does not support boolean values by default. Instead, integers are used for logical values, with 0 being "false", and any other number being "true".

The logical operators have the following non-standard notation:

- `&&` for logical and
- || for logical or
- `!` for logical negation

For example,
`a!=0 && b>=0`
is true if `a` is not zero and `b` is positive.

1.5 Input and Output

Input and output in C are supported at a higher level in the header file `stdio.h` which needs to be included if you wish to use any of the functions described in this section. Basic C supports only two low level functions, `putchar(Character)` and `getchar()`, which display a single character or get the next keyboard key

that is pressed. Unfortunately, in C there is no standard functions for unformatted I/O, that is, all I/O in C is formatted and the user must specify the type and format of the variables being diplayed or entered. This is corrected in C++.

1.5.1 `printf`

This is the main function for displaying information on the screen. It's syntax is:

printf("*Format_sequence*" , { *List_of_variables* } **);**

The format sequence specifies the kind and possibly the format of the variables that are being displayed, with any additional text for messages. It consists of three types of entries:

- *Text* for messages
- *Control commands* beginning with a backslash. The more important ones are:

Command	*Action*
\n	new line
\f	new screen
\t	tab space
\''	print a quote
\b	backspace

- *Data format specifiers* beginning with a percentage sign. They have the general form,
  %[Width][.Precision]Type
  where the width specifies the minimum number of digits to be printed (spaces are added) and the precision specifies the maximum number of decimal places to be printed. The type can not be omitted and specifies the type of the data that is being displayed. The most important data types have similar notation as in FORTRAN:

Specifier	*Data type*
d	integer
f, lf	float in decimal-point notation (lf is double)
e,E	float in exponential notation
g,G	float in notation depending on size
c, s	character or string

For example, to print the value of your income for the last year, use:

```
int year;
double income;
printf(''\n My income for the year 19%d was %5.2lf dollars'',year,income);
```

1.5.2 scanf

`Scanf` is used for formatted user input. Unfortunately, although the compiler parses the input according to the format specification, it does not perform a check on whether the actual user input conforms to the format specification. The syntax is,

scanf("*Format_sequence*" , &*Variable_name* **);**

where the `&` is used to get the pointer of the variable that is being entered. For example, to have the user enter the value of his income, use:

```
scanf(``%lf'',&income);
```

1.5.3 File I/O

As in FORTRAN, before data are written to a file, the file has to be opened. This is done by first including the header file `stdio.h`, then declaring a pointer (more on that later) of type `FILE` and assigning the pointer to a specific file using the command `fopen`. The command `fopen` thus effectively has the syntax,

**FILE ***_File_pointer_;

File_pointer = **fopen(** "*File_name_with_path*" , " *Mode* " **);**

where the mode is `w` for writing to a new file, `a` for appending to an existing file of that name, and `r` for reading (also, `w+r` is for a file used both for data input and output).

After that, reading and writing to the file is done in the same way as to the screen, with the commands `fprintf` and `fscanf`. These commands are almost identical to `printf` and `scanf`, with the change that before the format specification we specify the pointer to the file we wish to read or write to:

printf(*File_pointer* , "*Format_sequence*" , *{ List_of_variables }* **);**

scanf(*File_pointer* , "*Format_sequence*" , &*Variable_names* **);**

Files are closed with:

fclose(*File_pointer* **);**

This will become clear with the next example, where we open a file, write a message and the number 2^{10}, close the file, and then read the number from the file:

```
#include <math.h>
#include <stdio.h>
main()
{
int i;
FILE *fp;
i=(int)pow(2,10);
fp=fopen(``Test.dat'',``w'');
fprintf(fp,``This file has only one number:  \n %10d'',i);
fclose(fp);
fp=fopen(``Test.dat'',``r'');
fscanf(fp,``\n %d'',&i);
printf(``Two to the power of 10 is %d'',i);
```

```
fclose(fp);
}
```

1.6 Control Structures

Just as in FORTRAN, in C control structures are used to branch or loop the control flow in a program. The control structures are almost identical in their functions in the two languages, but with different names and syntax.

1.6.1 The `if/else` Structure

The `if/else` branching statement is almost identical to the FORTRAN one, aside from some syntax differences, so we just give the syntax without further explanations:

if(*first_condition* **)**
{ *First_block_of_statements* **}**
else if(*second_condition* **)**
{ *Second_block_of_statements* **}**
...
else
{ *Default_block_of_statements* **}**

The curly braces around the blocks can be omitted if only one statement is present.

1.6.2 The `switch` statement

The `switch` statement is equivalent to the FORTAN 90 SELECT CASE, and is used to branch the execution into many different directions depending on the value of an expression. Without giving the syntax, an example will make clear the usage:

```
switch ( menu_choice )
{
  case ``add'':
    c=a+b;
    break;
  case ``substract'':
    c=a-b;
    break;
  case ``multiply'':
    c=a*b;
    break;
  default:
    c=a/b;
    break;
}
```

This code will assign a different value to the variable `c` depending on the value of a menu choice string. Only integers or character constants can be used as the switch expression, and each case block must end with a `break` statement (to be descibed later). The default statement is optional and will be executed if neither case is satisfied.

1.6.3 The `for` looping structure

FORTRAN's DO structure is replaced in C by the `for` structure, which performs the same task as DO, but with a syntax that allows greater freedom and more possibilities:

for(*Initialization_statements* **;** *Test_expression* **;** *Increment_statements* **)**
{ *Body_of_loop* **}**

Notice that there is no semicolon after the `for` line if the loop has a body. In fact, the body is optional, or it may consist of only one line in which case the curly braces are optional. Several statements can go into the initialization and increment sections, provided they are separated by commas, but it is easier to read code that puts the main parts of the code in the body, delimited by curly braces.

The execution of the this loop goes as follows. First the initialization statements are executed, then the test expression is evaluated and if it is true, the body of the loop is executed, and after that the increment statements are executed. The whole process repeats without the initialization part so long as the test statement is true. Once the test statement becomes false the loop finishes. For example, to calculate the sum of the square roots of the positive elements of a matrix of size $[n \times m]$, we use:

```
sum=0.0;
for( i=0 ; i<n ; i++)
{
  for( j=0 ; j<m ; j++)
  {
    if( matrix[i][j]>0.0 )
      sum=sum+sqrt(matrix[i][j]);
  }
}
```

1.6.4 The `while` and `do/while` structures

In FOTRAN it was difficult to make semi-infinite loops because DO was the only repetition statement. In C however, the `while` statement is used quite often for loops that do not execute a fixed number of times. The `for` loop is in fact a simpler syntax of `while` suitable for array-like looping:

while(*Test_expression* **)**
{ *Body_of_loop* **}**

The loop is executed repetitively so long as the test expression is true. The only difference between this and the `do/while` construct,

do
{ *Body_of_loop* **};**
while(*Test_expression* **)**

is that in the later case the body is executed at least the first time since the test expression is evaluated only at the end of the loop body. It is recommended to use a simple `while` whenever possible. For example, to have C calculate the square root of a number the user enters and print it, so long as the number is positive, we use (this is not the best way to do this, it is just an example):

```
d=1.0;
while( d>0 )
{
  printf(``\n The square root of %f is %f.'',d,sqrt(d));
  scanf(``%f'',&d);
}
```

1.6.5 The `break` and `continue` utilities

These two very useful commands are equivalents to EXIT and CYCLE in FORTRAN and there is therefore no need of explanations. The `break` command simply exists the innermost control structure (other than `if/else`) enclosing it, while the `continue` command simply takes the execution of the loop to the next cycle.

1.7 Pointers

Pointers in C have a central place and are truly necessary tools for a good C programmer. We only briefly introduce them since you are not likely to need them often.

1.7.1 Pointer Declaration

A pointer is a variable that holds the (hexadecimal) computer memory address of a given variable. The pointer also has a size associated with it, that is, it points to as many memory units (bites) as are needed to store the variable type it points to. Therefore, it is necessary to declare the pointers as ordinary variables. The only difference is that an asterics * is placed in front of the pointer name. For example, a pointer that can point to an integer can be declared via:

```
int *pointer,number;
```

Pointers in C, unlike in FORTRAN, are not treated as mere substitutes for the real variables they point to, but are rather real variables that can be manipulated in expressions. For example, it is perfectly valid to increment a pointer by one, as in `pointer++`, which simply means that the pointer is moved for as many bites in the main memory of the computer as is the size of the pointer.

1.7.2 Pointer referencing and de-referencing

In FORTRAN, we only need to *reference*, or associate a pointer with a given variable, which makes the pointer point to that given variable. In C this is done with the unary operator `&` which returns the address of the variable. For example,

```
pointer=&number;
```

associates the pointer to the variable `number`. In FORTRAN, when we write `pointer`, it would be treated as an alias (a substitute) for `number`. But in C the pointer has a meaning by itself. In order to get the value that the pointer points to, in C we need to *de-reference* the pointer. This is done with the operator *, which returns the actual value that the pointer points to:

```
number=*pointer;
```

It is important to notice that the compiler can notice the difference between the pointer * and multiplication (since the pointer operator has higher precedence), but it is wise to use parenthesis in order to be on the safe side.

1.7.3 Pointer Usage

Pointers are used in C in a variety of ways. The usage important for scientific computing is for array manipulation, memory manipulation, and passing arguments to functions (explained later). In terms of array manipulation, we already mentioned that an array is absolutely equivalent to a pointer. In fact, the array name is simple a *constant (fixed) pointer* to the first element in the array. For example, here is a way to to find the squares of the values of a one dimensional array of length `N`:

```
float *pointer,*array_end,array[N],squares[N];
array_end=array+N;
for(pointer=array,array_end=array+N;pointer<=array_end;pointer++)
  *pointer=(*pointer)*(*pointer);
```

You might be asking yourselves why is this important. We could just as well have done the same (albeit somewhat slower!) with:

```
for(i=0;i<N;i++)
  array[i]=array[i]*array[i];
```

1.7.4 Dynamic Memory Allocation

The reason for using pointers in the context of array manipulation is dynamic memory allocation. Remember that dynamic memory allocation, meaning using arrays and other variables of variable size, does not exist in FORTRAN 77, and in the 90 standard we can very easily allocate arrays of variable size.

C existed long before FORTRAN 90, and it is worth repeating that it is not a language intented for numerical manipulation (which is the same as numeric-array manipulation). Therefore, although C has very powerful memory allocation routines, these routines simply allocate blocks of memory, not arrays. The way to turn them into arrays is to use incremented pointers instead of array

indices, as already explained. It is worth noting that C gives little supervision over what the programmer does, so use memory allocation with caution to avoid system crashes.

Dynamic memory allocation in C is done primarily with the function `calloc` (or `malloc`) in the header file `alloc.h`. We have not dealt with functions yet, and so we won't try to give the general syntax of `calloc` (as C's help would, for example), but rather here is the main usage format,

Pointer = **(***Type* ***) calloc(** *Number_of_elements*, *Bytes_per_element* **);**

which allocates a block of memory of the given number of elements, with each element of a given number of bytes per element and returns a pointer of the a certain type that points to the first element in the memory block. If you don't know how many bites the data type uses, it is recommended (for portability purposes as well) to use the function `sizeof`. For example, the declaration

```
int array[100];
```

can be done dynamically with:

```
int integer=1;
array=(int *)calloc(100,sizeof(integer));
```

If there is not enough memory to allocate, the returned pointer is of type `NULL`. When the memory is no longer needed, it can be freed with the command `free`. For example, we would free the memory allocated in the previous example via:

```
free(array);
```

1.8 Functions

Unlike in FORTRAN, in C all procedures are called functions and have the same structure. As usual, it is difficult for beginners to fully understand the scoping and lifetime issues of the dummy arguments and local arguments to functions, but here are a few important examples.

1.8.1 Scope and Lifetimes of Variables

First and most important, in C the dummy arguments of functions are (in most cases) totally independent of the actual variables in the other procedures or the main program, *unless they are pointers.* In other words, when a function is called it makes its own copies of the actual arguments which are stored in the dummy arguments. These copies are independent of the actual arguments in the calling program unit. So, modifying the values of the dummy arguments *does not change* the actual values of the arguments.

This is not true for pointers, and one of the main usage of pointers is to change the values of the calling arguments. Also, note that since arrays are simply pointers, they are not copied when used as arguments and any changes made to the array in a function reflect in the main program as well. Think carefully about these issues and talk to someone (or refer to other sources) to

understand this well, since it truly is important even though you may not use it in the near future. Also, contrast this behaviour with FORTRAN.

As for local variables, the behaviour is the same as in FORTRAN. They are created when the function is called and deleted when it is finished, and may not be used later on. The way to modify this was already explained, and that is to use the `static` modifier when declaring the local variables. Also, we already mentioned that it is possible to extend the scope or visibility of certain variables to all of the functions (make them global) by declaring them after the preprocessor statements.

1.8.2 Declaring Functions

The exact way functions are declared in C depends on the compiler, but today the ANSI standard of C is used most often. In this standard, each function needs to have its *function prototype*, which is the same as the INTERFACE block in FORTRAN, and simply tells the compiler what type of a variable the function returns and what type of arguments it accepts. The prototype should be placed after the preprocessor statements and outside of all other functions (including `main`). The syntax for prototypes is:

Returned_type Function_name(*{List_of_argument_types}*)

For example, we know that the function `pow` accepts two `double`'s, the base a and the exponent b, and returns another `double` $c = a^b$. Its prototype would thus be:

```
double pow(double,double)
```

If a function does not return a value (a subroutine in C), it should be declared as type `void`. Also, if a function does not accept any argumens, the type of the argument should be declared as `void`, as in:

```
void message(void)
{
 printf(``Just a message...\n'');
}
```

1.8.3 Function Body

The body of the functions can be placed in a separate file, before, or after the main program (it is bad practice to place the main function in-between other functions). The body of the function begins with a declarative statement very similar to the function prototype, with the difference that names for the dummy arguments are given along with the argument types. The actual body of the function is as usuall enclosed in curly braces. The function returns a value through the command `return` followed by the value to be returned.

For example, the body of the `pow` function can be written using the identity $a^b = e^{b \ln a}$ (notice that there is no semicolon after the first line):

```
double pow(double a, double b)
{
  return exp(b*log(a));
```

```
}
```

Arrays can be passed as arguments by leaving the dimension in the declaration empty. A very illustrative example is given in the next subsection. For now, here is an important example of using pointers in a function that changes (swaps) the values of two strings, up to the end of one of the strings (remember that strings are themselves pointers):

```
/* The prototype:  */
void swap(char *, char *);
/* The body:  */
void swap(char *string_a,char *string_b)
{
  char *temp,*a,*b;
  for(a=string_a,b=string_b ; (*a)!='\0' && (*b)!='\0' ; a++,b++)
 {
    *temp=*a;
    *a=*b;
    *b=*temp;
 }
}
```

1.8.4 An Example–Using Arrays in Functions

At the end we give a more complicated example of a program with a function that finds the maximum element of a numeric array:

```
/* Finding the maximum in a numeric array */
#include <stdio.h>
int maximum( int [], int );
main()
{
  int array[] = { 1, 4, 5, -25, 38, 92, 0, 5, 7, 9 };
  printf(''The maximum element is %d \n'', maximum(array, 10));
 }
int maximum( int numbers[], int n_elem )
{
  int largest_value, i;
  largest_value = numbers[0];
  for( i = 0; i<n_elem ; ++i )
  {
    if( numbers[i]>largest_value )
    largest_value = numbers[i];
  }
  return largest_value;
}
```

Chapter 2

Object-Oriented Programming in C^{++} (and FORTRAN 90/95)

In this chapter, we will only briefly and rudimentary explore the concepts that make C^{++} the most widely used programming language among the computer science community. It is very helpful for every programmer to understand the advantages of structured and object-oriented programming. However, C^{++} compilers today are still too slow to be used widely for scientific computing[1], and so we will not go into many details. FORTRAN 90/95 also supports some aspects of object oriented programming (although it is *not* considered an object-oriented programming language) and these were not discussed in the *Introduction to FORTRAN* reference manual. Therefore, throughout this chapter we will reference the FORTRAN equivalents to C^{++}'s object-oriented commands.

2.1 Important Syntactic Changes in C^{++}

Before we begin discussing object-oriented programming, it is worth mentioning a few of the more or less syntactic improvements in C^{++} over standard C.

2.1.1 Comments, Variable and Function Declarations

C++ supports the old-style multi-line comments enclosed between /* and */, as well as a new single line comment that begins with the // symbol. Programmers usually have their own commenting standards that use both styles.

A more important change is the fact that variables in C^{++} can be declared *anywhere* in the program. In fact, a variable declared in a program part eclosed with curly braces ceases to exist after the closed curly brace. This avoids the

[1] We will discuss this issue in a separate section.

need to go back to the beginning of the program to declare variables and avoids the overuse of dynamic memory allocation.

Functions can be declared in C^{++} with default values for the parameters, and some parameters can be optional, just as in FORTRAN. Also, a concept called *function overloading* allows to use the same function name for different types, just like in FORTRAN where, for example, SIN can be used for both single and double precision variables. Further more, operator overloading is also supported in both C^{++} and FORTRAN, so that one can use `+` or `*` for, let's say, strings (or any other user-defined data type). We will not give examples of how that is done here for this is a (almost purely) syntactic improvement and slows down things in scientific computing (due to the additional choices that the operator or function has to make).

Another such change is the introduction of *passing arguments by reference*, which is the standard way in FORTRAN. This means that arguments passed by reference are actually passed by using their pointer, so that any changes made to the variable inside the function is reflected in the calling subprogram as well. For example, here is a function that can swap the values of two integers passed by reference using the pointer operator `&`:

```
void swap(int& i, int& j)
 {
 int temp = i;
 i = j;
 j = t;
 }
```

2.1.2 Input and Output

The I/O standard routines have been completely renovated using the new object-oriented features of C. The new standard I/O header file is the `iostream.h` library. Printing things on the screen no longer requires long format lines and type specifications. By using the command `cout` and the symbol <<, one can print any standard data object on the screen. For example,

```
int i = 2;
float f = 3.14;
char *s = ``2*Pi is'';
times='*';
cout<<s<<''\t''<<i<<times<<f<<endl;
```

displays the message "2*Pi is [tab] 2*3.14 [new line]" on the screen. Input is done in an identical manner, with the command `cin` and the symbol >>. For example,

```
int i,j,k;
cin>>i>>j>>k;
```

will read three integers from the keyboard into `i`, `j` and `k`. White space is automatically used as a separator and ignored. A wonderful feature of the iostream library is that it can be extended to non standard types using object-oriented language features (in other words, you can print matrices or vectors

with the same degree of simplicity). Also, the same changes apply to file I/O as well, but we will not go into the details here.

2.1.3 Dynamic Memory Allocation

Dynamic memory allocation is very important in scientific computing where large matrices of variable sizes need to be allocated quickly and easily. C++ makes memory allocation almost as simple as it is in FORTRAN 90 with an almost identical syntax. C++ replaces the C memory allocation function `malloc` and the deallocation function `free` with `new` and `delete` respectively, and these are much easier to use. For example, allocating a large array and then deleting it is trivial,

```
int *array;
array=new int[10000];
...
delete [] array;
```

where the `[]` is needed for arrays.

2.2 Object-Oriented Programming Languages

Object-oriented programming languages were developed as an answer to problems seen in *very large programs* in the early 70's. In this class, and throughout your education, you are unlikely to ever develop large programs, but you are likely to use other people's large libraries that are based on object-oriented language features. In fact, you've already used Mathematica, and such a complex programming environment could not be developed without the advent of languages like C++.

The three main features of object-oriented languages are:

- Object oriented languages all implement "**data abstraction**". Data abstraction is a concept that enables programmers to talk about data and operations performed on the data without referring to implementation details. If you ever read a good book on data structures and algorithms, you will notice that all algorithms are given in a so called pseudocode which is really not written in any existing programming language. How that pseudocode translates (implements) into a specific programming language is completely irrelevant, so long as the data and operations performed on the data in the pseudocode are available to the programmer. And we will see how C++ and FORTRAN both give us a way to make the data and the associated operations readily available, reusable and extensible by using *classes* and *modules* respectively.

- All object-oriented languages try to make parts of programs easily reusable and extensible. Programs are thus broken down into **objects**. These objects can then be grouped together in different ways to form new programs or extended to form new objects.

- Object-oriented languages create **easily modifiable code**–code that can be changed without actually changing it. At first it doesn't seem possible to change something without actually changing it. Using C^{++}'s *inheritance* and *polymorphism* it is possible to do that. The existing object stays the same, and any changes are layered on top of it.

The last two properties are most important for software companies that create large programs that need to be upgraded and improved continuosly in as a bug-free manner as possible. Data abstraction on the other hand is a fundamental and important concept even in scientific computing, so we begin by describing it.

2.3 Classes

A class is simply an extension of a C structure. A class allows you to create a structure, and then permanently bind all closely related functions to that structure.This process is known as encapsulation and is the heart of object oriented programming:

$$\boxed{\text{data} + \text{functions} = \text{class object}}$$

To explain how this works, we need an example. We first introduce the problem by describing a data structure that we call a *list*.

2.3.1 Why a Class : A List

In many applications involving applications like databases, sorting, searching etc., there is a need to create a structure than can hold an arbitrary number of elements of a certain type in a certain order. This implies that such a data structure has to be extensible–we must be able to add new elements. Also, we must have a mechanism for deleting some of the elements and accessing all the elements in this data structure.

Examples are very easy to think of. For example, let's say that a nutty professor wants to keep a record of all the strange names that he encounters in his class. The basic element that we need to store in our data structure is thus a name (a string). But notice that the professor needs to be able to add new names to the list of names and to delete old entries if he no longer finds those last names strange. Examples like this come about very, very often. Let's call this data structure a list.

The idea od object-oriented programming is to enable us to create such a data structure once, and then use this data structure in as many problems as we want to. Notice that we did not mention anything about the specific implementation of this list. It can be an array of strings, with let's say, a maximum of 100 elements. This is obviously not the best solution since if we have less than a 100 entries in the list lots of memory will be wasted, and we also can not have more than 100 entries. One way to solve this problem is to

create a data structure called a link list, which we will not try to implement at this point.

The point to understand is that whatever the specific implementation of the list, what is essential is the way that list is used. In other words, the main program only needs to be able to access the elements of the list, to delete an element and add a new element. It is also very helpful to be able to locate a certain name in the list, if it is a member of the list. These several operations are intimately related to the list, and so we can think of the list and these operations (functions, procedures) as being one data structure, which we call a list *class*:

Class: *List* = Data: *A list* + Functions: [*add* + *delete* + *access* + *locate*]

In a specific implementation we will need a way to create the whole list and delete it, but these will come naturally as we try to actually implement the list class.

At this point it should already be clear why we would want to create such a class in the first place. But let's answer this question one last time. We do not really have to implement this class, but implementing it will make all the programs that use lists much more portable, simpler, better organized, easily maintainable. All we need to change, improve, modify etc. is the list class. The programs that use the list will not change and suffer at all, if we implemented our class well. So, when you become a better programmer and can implement a linked list, you can improve the list class below without changing the main program at all.

2.3.2 Classes in C^{++} : Implementing an Unordered List

Now we show how to implement the list class in C^{++}. We will not give any syntax or go into any details, just give the example and show how it works. The rest will naturally become clear. At this point, we will only implement a list of names, or a list of strings. Also, we will not worry about efficiency and simply make our list an array of 100 strings.

At this point we should confess that the implementation is really not that of a list, but rather an *unordered list*. We do this only to make the `delete` function simple, and since an ordered list can only be implemented efficiently as a linked list. Here is the class that implements an unordered list of strings:

```
#include <stdio.h>
#include <iostream.h>
#define MAX 100
class StringList
{
  private:
  char *array[MAX];
  int element;
  public:
```

```
  StringList() // constructor
  { element=-1; }
  ~StringList() // destructor
  { }
  int locate(char *);
  int add(char *);
  int remove(int);
  char * access(int);
  int num_elements();
}
```

A few things deserve explanation. The first line is a declaration line, just like `struct StringList`, and it creates a new class. The reserved word `private` means that whatever comes after it is accessible only to functions inside the class, and this simply protects our list from being inadvertently modified from outside the class. The word `public` means that whatever follows it is accessible from outiside the class as well. The special function `StringList()` has the same name as the class and is called a *constructor*. This function is called when the class is first used from the main program and in this case it simply sets our element counter `element` to zero. The other special function, `~StringList()`, is called a *destructor* and is called when the class stops being used in the main program. In this case we do not need to to anything (this function is empty), but we may need to close a file, or something like that, at the end.

The rest are function prototypes for the functions that are a part of the class. We now need to give the actual body of these functions (best to be done outside of the class) and decide what exactly we want them to do. The function `locate` returns the location of a given string in the list if it is present or -1 if it is not. We need to add a double colon when giving the body of this function to tell the compiler (and our selves) that this function is a part of the class `StringList`:

```
int StringList ::  locate(char *name)
{
  int i,found=-1;
  for(i=0;i<=element;i++)
  {
    if(array[i]==name)
    {
      found=i;
      break;
    }
  }
  return found;
}
```

The function `add` needs to add a new surname at the end of the list `array`, so long as we don't go over the size of the list. It returns 1 on success or 0 if the list is full (it is useful to add this feature so that a program that uses these functions can determine whether the operation was successful or not):

```
int StringList ::  add(char *name)
```

```
{
  if(element<MAX)
  {
    element=element+1;
    array[element]=name;
    return 1;
  }
  else
    return 0;
}
```

The function that deletes (removes) an entry is `remove` (`delete` is a reserved word in C++):

```
int StringList ::  remove(int del_element)
{
  if (del_element<=element)
  {
    array[del_element]=set[element];
    array[element]='\0';
    element=element-1;
    return 1;
  }
  else
    return 0;
}
```

A few explanations of how `remove` works. Since we agreed that order is not important in our list, when we delete a surname at the location `del_element` we copy the surname from the end of the list to that position, so that there are no empty entries in the list. Again, the function returns 1 upon success and 0 upon failure.

The `access` function is even simpler and only returns the name at position `access_element`. We could have of course chosen different methods of accessing the elements of the list, and usually careful planning is needed of what exactly we want the class to support before embarking on actually coding it:

```
char * StringList ::  access(int access_element)
{
  if (access_element<=element)
  {
    return array[access_element];
  }
  else
    return '\0';
}
```

We have also added a utility function `num_elements` which will be handy to anyone using the list of strings, and this function simply returns the number of elements in the list. We could have alternatively made the variable `element`

public and available to external functions, but this method is unsafe and may lead to program errors much more easily:

```
int StringList ::  num_elements()
{ return element; }
```

2.3.3 Using Classes in C^{++} : The `main` Function

After having created the class `StringList` (usually placed before the `main` function), we can now use it in a program. The only thing that you need to remember and understand is that a class is used just like a structure, but with functions in it as well. Here is a simple program and its output. Try to understand exactly what is happening by yourself:

```
void print_names(StringSet);
main()
{
  int i;
  StringList strange_names;
  strange_names.add(``Mikke'');
  strange_names.add(``Mikky'');
  strange_names.add(``Mikey'');
  print_names(strange_names);
  strange_names.add(``Mikky'');
  print_names(strange_names);
  i=strange_names.locate(``Mikke'');
  strange_names.remove(i);
  print_names(strange_names);
}
void print_names(StringList names)
{
  int i;
  cout<<''The listed strange names are:''<<endl;
   for (i=0;i<=names.num_elements();i++)
  {
    cout<<names.access(i)<<endl;
  }
}
```

Note that we made a separate function to print the list since this was not a part of the class. The output of the program looks like:

```
The listed strange names are:
Mikke
Mikky
Mikey
The listed strange names are:
Mikke
Mikky
```

Mikey
Mikky
The listed strange names are:
Mikky
Mikky
Mikey

2.3.4 Inheritance in C^{++} : Implementing a Set

Now we are going to look into a way of extending the previously developed class `StringList` without actually changing it. We do this by making another class that inherits the properties of the old one, while adding some new changes to it as well. The concept of *inheritance* is a very important one and central to the idea of object oriented programming. It enables the programmer to extend and build upon existing structures with almost no effort. We can even inherit classes that we do not have the code for (commercial compiled libraries). It is said that this adds a third dimension to programming not present in non object-oriented languages[2].

We modify the previous class that implemented an unordered list of strings so as it implements a set of strings. A *set* is an unordered (unsorted–these two terms are slightly different) list which *does not contain repeated entries.* It is reasonable to expect that out nutty professor does not need to store the same name twice, and that it does not matter what order the names are in. Here is a class `StringSet` implementing a set of strings by building upon the existing class `StringList`:

```
class StringSet :  public StringList
{
  public:
  StringSet() // constructor
  { StringList(); }
  ~StringSet() // destructor
  { }
  int add(char *);
};
```

A few explanations. The colon after the declaration of the new class means that we are inheriting an existing class. We also have to change the `add` function, since we do not want to add repeated entries. We do this by simply writing a new procedure with the same name `add` that checks whether a given name is in the list before it adds it. It may be strange that we can in fact use the same name for a different function, but this is one of the strenghts of object-oriented programming called *function overloading* and is an example of *polymorphism* (same words for different meanings). In C^{++} we can even overload the standard operators and functions, but we will not dicuss that right now:

[2]The first dimension is regular sequential programming and the second dimension is formed by functions and subroutines.

```
int StringSet ::  add(char *surname)
{
  int found;
  found=locate(surname);
  if(found==(-1))
    return StringList::add(surname)
  else
    return 0;
}
```

The double colon in `StringList::add` means that we are using a function from the previous hierarchy level in the inheritance list, and in most occassions one level of inheritance will be sufficient, but there is no limit posed by C^{++}.

We can use the same `main` function again, by simply changing all appearences of `StringList` with `StringSet`, and repeated entries will no longer be entered into the list. To see this, here is the actual output of the compiled program with the new class (compare this to the previous output):

```
The listed strange names are:
Mikke
Mikky
Mikey
The listed strange names are:
Mikke
Mikky
Mikey
The listed strange names are:
Mikey
Mikky
```

2.4 Object-Oriented Programming in FORTRAN 90/95

It is important to note that the new standards of FORTRAN support some features of object-oriented programming, although it is not until the new 2000 standard that FORTRAN is expected to become a true object-oriented programming language.

The substitution that FORTRAN offers for classes are **modules**. Modules are often used for holding global variables or data needed in different subprograms. However, modules can also contain procedures, called *module procedures*, just like a class contains functions associated with the data in the class. These module procedures are declared in much the same way as regular procedures, and in some instances require the attribute MODULE. There are, however, some important differences between classes (in C^{++}) and modules:

- Modules are *not* equivalent to derived data types, while classes are equivalent to structures. In other words, in C^{++} one can declare an array of 10 sets of strings `sets` with a mere `StringSet sets[10]`, while in FORTRAN only one USE statement can be used per module, and the programmer of the module has to facilitate another way of declaring more than one sets of strings.
- FORTRAN does *not* (at this point) support automatic inheritance with modules, so that more effort is needed to build hierarchies of modules. This is viewed by many as the biggest downfall od modules as object-oriented structures.

FORTRAN supports some other interesting features as well, like *overloading* operators and functions (a kind of polymorphism). It is best not to go into details of implementations of object-oriented features in FORTRAN. This is done very well in the websites listed in the abstract of this manual, and some more or less useful ways of emulating C^{++} features are described. It should be greatly stressed that object-oriented features greatly slow down programs (we discuss this important feature in the next subsection) and should not be the first choice of a scientific programmer. In many cases though, the advantages of object-oriented programming are more then worthwhile.

2.5 FORTRAN, C^{++}, Compilers, Speed and Scientific Computing

The long title of this section tells by itself that this is one of the most important sections in these reference manuals. However, we can not say a lot about the issues of speed in scientific computing without going into "rough" waters. The scientific community does not agree on many of the questions discussed here, and things are changing all the time. However, we do want to stress the importance of considering the programming language, compiler (and/or compiler switches) and speed issues when approaching a real scientific computing task.

2.5.1 FORTRAN versus C : The Battle Goes On

With the multitude of programming languages available, it may be questionable why we chose to dicuss only FORTRAN and C^{++}, and why we discussed the old FORTRAN 77 and C. The answer is due to an often asked question: *What programming language is the "best"?* Of course, the answer to this question strongly depends on the type of application being discussed. However, an *approximate* answer can go as follows:

- For *general-purpose programming*, C^{++} is the language of choice[3]. What is the relevance of this statement to scientific programmers? Well, all sci-

[3]We will not even mention things like system-level programming since these are not relevant. The word general-purpose can be interpreted as anything a physicst might need.

entific programmers are advised to learn at least the basics of C^{++}, since most do more than just scientific computing. The flexibility and functionality of C^{++} make it an irreplacable programmers's tool, and it is exactly these characteristics that are needed in general-purpose programming.

- For *scientific computing* FORTRAN ?? is the language if choice. Scientific computing can largely be taken to be a synonym for *number crunching* and *array* (matrix) *manipulation*. Speed is the most important attribute to scientific computing, and FORTRAN is unsurpassed in this respect. Notice that we have placed a ?? for the best version of FORTRAN, since this is still a debated issue. We may however say that the 90 standard is winning the battle with certainty, and this is why you will be asked to follow this standard most of the time. We explain this claim next.

2.5.2 Compilers and Speed

The speed of execution is the single most important parameter to consider in scientific programming. As already mentioned, the speed of execution (on a *fixed* computer architecture) depends mostly on the way the program handles numbers (integers and floats), and the way it handles arrays. The way the computer handles numbers is in many respects set by the processor since most processors today come optimized for certain types of flops, although the programming language and the compiler used do matter.

Handling Arrays

The way a programming language and a compiler handle arrays, however, is a distinguishing speed factor. Here are a couple of famous examples.

- When passing arguments to a function, many programming languages (including C) physically copy the passed arguments to the function memory space. Fortunately, C does not do this for arrays (because they are pointers, not real variables). Imagine how slow the execution of a program that calls a function (let's say FFT) with a very large numeric array (say 2^{15} elements) would be if the function copied this huge array whenever it was called. In C, functions copy their arguments to facilitate structured programming in which functions don't interfere a lot with the "outside world", and this is indeed very important. FORTRAN never does this, but that is why the new standard provides the INTENT attribute for function arguments, which enables the compiler to check for possible missuses of a given argument.

- When accessing an element of an array, say `array[100]` in C or `array(100)` in FORTRAN, it is a very important question whether the compiler should check to make sure the requested element is within the array bounds (that the array has 100 entries in this case). If the compiler does not do this, errors and system crashes can occur; if it does it every time, then this

will slow the execution considerably. C never does this sort of checking. FORTRAN on the other hand leaves this to the compiler, so that most compilers have compilation options (switches) for specifying whether we want array-bound checking or not. An experienced programmer would put this switch on when testing the program, but *always* turn it off when compiling the final executable.

Compiler Optimization

The importance of the compiler and the various compilation options that it offers was already stressed. One thing thay you may not be familiar with is the so called *compiler-level optimization.* We can easily explain this on the following example. Suppose you write a program that needs to evaluate an expression like:

$S = \sqrt{x^2 + \sqrt{x^2 + x^4}} + x^{-2}$

If the compiler is good, it will notice that the expression x^2 occurs very often in this expression and optimize, or expand this expression to something like:

$t_1 = x^2$, $t_2 = t_1^2$, $t_3 = \sqrt{t_1 + t_2}$, $t_4 = \sqrt{t_1 + t_3}$, $S = t_4 + \frac{1}{t_1}$

Most compilers are fairly good at this, and they offer compilation options for various levels and types of optimizations. A good programmer will carefully study these before embarking on an important project. In most UNIX compilers, including `g77`, the generic optimization switch is `-O`.

The best optimization in scientific computing today is still among FORTRAN 77 compilers. This, and the fact that there is a lot of legacy from previous scientific programmers in this language, make the 77 standard very widely used. It takes a lot of years of hard work and research to find stable and robust ways of optimizing code, so that the FORTRAN 90 compilers are just about catching up (95 compilers are still rare). The flexibility of C^{++} greatly limits the ability of the compiler to do automatic optimizations, and much effort is being put into making optimized C^{++} compilers, with small chances of ever catching up with FORTRAN. As a rule of thumb, *C^{++} programs are twice slower in scientific computing that FORTRAN 90 programs* (compiled with a good compiler, which is hard ot find), and these are anywhere from 10-50% slower than 77 programs, depending on the compiler and the problem at hand (in some cases, dicussed next, a well optimized 90 program should run much faster than a 77 program).

Parallel Processing

The future of scientific computing lies in parallel processing. Current semiconductor technology has a limit on the possible processor speed (in the range of about 1.7 GHz), and we are slowly approaching this limit. Scientific programming is the perfect candidate for effective multi-processor (parallel) computing because it often entails performing similar operations on a multitude of data. FORTRAN 90 is a pioneer in the attempt to introduce parallel constructs as a

fundamental part of the language, and we already discussed the importance of array syntax, FORALL and WHERE constructs in the previous manual.

However, compilers have still not caught up with the universality requirements of the standard, but this is changing every day. Another major addition to the 90 standard is the introduction of the *High Performance Fortran* (HPF) "semi-standard", which introduces the possibility of helping the compiler by introducing commands that give directions for parallelization of the code. HPF, together with another standard called the *Message Passing Interface* (MPI) or other variations of MP, make the major contestors for the future of scientific computing, and we have therefore incouraged FORTRAN 90/95 throughout these manuals.