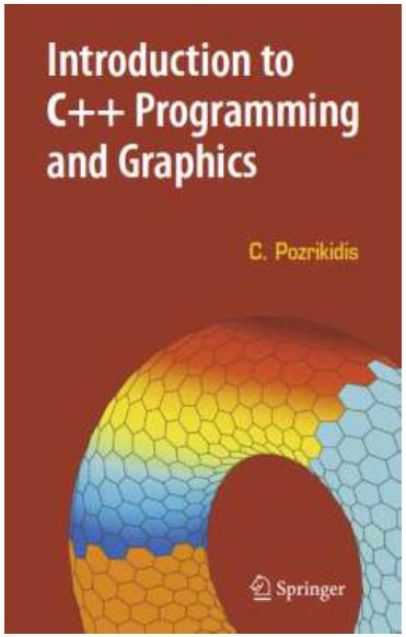
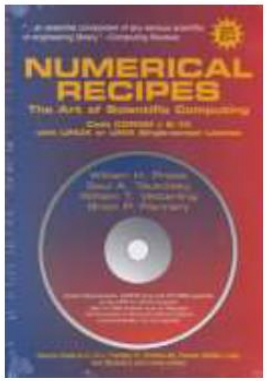
Programming and numerical analysis -user defined-functions

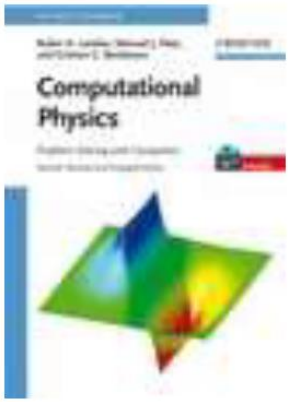
Hiroshi Sagara

Oct. 13, 2020
Tokyo Tech

Text Book







Numerical recipes in C++ : the art of scientific computing / William H. Press ... [et al.], ISBN: 9780521750332 [0521750334]

Introduction to C++ Programming and Graphics / Constantine Pozrikidis, Springer eBooks Computer Science, ISBN: 9780387689920 [0387689923], https://link.springer.com/10.1007/978-0-387-68993-7

Computational Physics - Problem Solving with Computers / Rubin H. Landau, Cristian C. Bordeianu, Manuel José Páez Mejía Wiley Online Library Online Books ISBN: 9783527406265 [3527406263] 9783527618835 [352761883X], https://onlinelibrary.wiley.com/book/10.1002/9783527618835

Remind: Operators

Arithmetic operators

The basic implementation of C++ supports the following arithmetic operators:

- Addition (+): We may write

```
c=a+b;
```

- Subtraction (-): We may write

```
c=a-b;
```

- Multiplication (*): We may write

```
c=a*b;
```

- Division (/): We may write

```
c=a/b;
```

- Modulo (%): We may write

```
c=a%b;
```

This operator extracts the remainder of the division a/b. For example

$$5\%3 = 2$$

Remind: Operators

Compound assignation

Other unconventional statements mediated by compound assignation operators are listed in Table 3.1.1.

Operation	*Meaning*
a +=b;	a=a+b;
a -=b;	a=a-b;
a *=b;	a=a*b;
a /=b;	a=a/b;
a *= b+c;	a=a*(b+c);
a++;	a=a+1;
++a;	a=a+1;
a--;	a=a-1;
--a;	a=a-1;

Table 3.1.1 Unconventional statements mediated by compound assignation operators in C++. The language name C++ translates into C+1, which subtly indicates that C++ is one level above C. Alternatively, we could have given to C++ the name C and rename C as C−−.

Remind: Operators

Relational and logical operands

Relational and logical operands are shown in Table 3.1.2. For example, to find the maximum of numbers `a` and `b`, we write:

```
max = (a>b) ?  a :  b;
```

If $a > b$ is true, the variable max will set equal to a; if $a > b$ is false, the variable max will set equal to b.

Equal to	a == b
Not equal to	a != b
Less than	a<b
Less than or equal to	a<=b
Greater than	a>b
Greater than or equal to	a>=b
And	A && B
Or	A || B
Boolean opposite or true or false	!A
Conditional operator	A ? a : b;

Table 3.1.2 Relational and logical operands in C++; a, b are variables, and A, B are expressions. The conditional operator shown in the last entry returns the value of the variable a if the statement `A` is true, and the value of the variable b if the statement `A` is false.

4 User-Defined Functions

Text p.91 ~

The use of main programs and subprograms that perform modular tasks is an essential concept of computer programming. In MATLAB and C++ we use functions, in FORTRAN 77 we use functions and subroutines. In C++, even the main program is a function loaded in memory on execution. Large application codes and operating systems may contain dozens, hundreds, or even thousands of functions.

In mathematics, a function is a device that receives one number or a group of numbers in the input, and produces a new number or groups of numbers in the output. The input and output may contain vectors and matrices collectively called arrays. Computer programming functions work in similar ways. In addition, the input and output may contain characters, strings of characters, words, sentences, and even more complex objects.

The individual variables comprising the input and output of a C++ function are communicated through the function arguments enclosed by parentheses following the function name. We will see that, for reasons of efficient design, single variables (scalars) are communicated differently than arrays.

4.1 Functions in the main file

The following C++ code contained in the file *ciao.cc* prints on the screen the greeting "Ciao" by invoking the function `ciao`:

```
#include <iostream>
using namespace std;

//--- function ciao:

void ciao()
{
cout << "Ciao\n";
}

//--- main:

int main()
{
ciao();
return 0;
}
```

4.1 Functions in the main file

The statement `ciao()` in the main program prompts the execution of the function `ciao`, which prints on the screen "Ciao" and returns nothing to the main program; that is, the return is *void*. The mandatory parentheses () enclose the function arguments after the function name; in this case null, the arguments are null.

Note that the function `ciao` has been defined before the main program. If the order is transposed, the compiler will complain that the function is attempted to be used before declaration. To satisfy the compiler, we duplicate the function prototype *before* the main function, as shown in the following code:

4.1 Functions in the main file

```
#include <iostream>
using namespace std;

void ciao();

//--- main:

int main()
{
ciao();
return 0;
}

//--- function main:

void ciao()
{
cout << "Ciao\n";
}
```

4.1 Functions in the main file

A function can call another function. An example is implemented in the following code:

```
#include <iostream>
using namespace std;

void greet1();

//--- main:

int main()
{
greet1();
return 0;
}

void greet2();

//--- function greet1:

void greet1()
{
cout << "bye now" << " ";
greet2();
}

//--- function greet2:

void greet2()
{
cout << "come again" << endl;
}
```

4.1 Functions in the main file

Tokyo Tech

Running this program prints on the screen:

```
bye now come again
```

The second function declaration `void greet2()` could have been stated before the main program.

4.2 Static variables

Text p.93 ~

Suppose that a variable is defined inside a function. When the function is exited, the value of the variable evaporates. To preserve the variable, we declare it as static using the qualifier `static`.

4.2 Static variables

As an example, running the program:

```
#include <iostream>
using namespace std;

void counter();

//--- main:

int main()
{
for(int i=1;i<3;i++)
  {
  counter();
  }
cout << endl;

return 0;
}

//--- function counter:

void counter()
{
static int n=0;
n++;
cout << n << " ";
}
```

4.2 Static variables

prints on the screen:

```
1 2
```

Every time the function *counter* is entered, the static variable `n` increases by one unit. A static variable can be used to count the number of times a function is called from another function.

4.2 Static variables

In another application, we use the `static` declaration to prevent a variable from being reinitialized:

```
for (i=1;i<=5;i++)
{
static int count = 0;
count++;
}
```

At the end of the loop, the value of `count` will be 5, not 1.

4.3 Function return

Text p.95 ~

A function can return to the main program a scalar, a character, or a string by means of the `return` statement. Thus, running the program:

```
#include <iostream>
using namespace std;

double piev();

//--- main:

int main()
{
double pi = piev();
cout << pi << endl;
return 0;
}

//--- function piev:

double piev()
{
double pidef=3.14157;
return pidef;
}
```

prints on the screen:

```
pi=3.14157
```

4.3 Function return

A function may contain more than one **return** statement at different places, evaluating the same variable or different variables. When a **return** statement is executed, control is passed to the calling program.

As an example, a function may contain the following structure:

```
...
if(index==1)
  {
  return value1;
  }
else if(index==2)
  {
  return value2;
  }
...
```

where the three dotes indicate additional lines of code.

4.3 Function return

A scalar, character, or string computed in a function is customarily passed to the calling program through the `return` statement. A less common alternative is to pass it through a function argument enclosed by the parentheses following the function name, as will be discussed later in this chapter. Groups of scalar variables and arrays *must* be communicated as function arguments.

Tokyo Tech

Prime numbers

As an application, we discuss a code that decides whether a given integer is prime. By definition, a prime number is divisible only by itself and unity.

The prime-test is based on the observation that the remainder of the division between two integers is an integer only if the two numbers are divisible.

The following code contained in the file *prime.cc* assesses whether an integer entered through the keyboard is prime:

4.3 Function return

```
#include<iostream>
using namespace std;

int GetN();

//--- main:

int main()
{
int k,l,m,n=1;

while (n>0)
{
  n=GetN();

  for(m=2; m<=n-1; m++)
  {
    l=n/m;   //--- Test for the remainder
    k=l*m;

    if(k==n)   //--- Not a prime:
      {
      cout<<"\n"<<n<<" is not a prime number;";
      cout<<" the highest divisor is:  "<<l<<"\n";
      break;
      }
  }

  if( k!=n && n!=0 || n==2)   //--- Found a prime:
  {
  cout<<"\n"<<n<<" is a prime number";
  }
}
return 0;
}

//---- Function GetN:

int GetN()
{
int n;
cout<<"\nPlease enter the integer to be tested:  \t";

while (cin>>n)
  {
  if (n<0)
    {
    cout<<"\nThe integer must be positive; try again\n";
    }
  else
    {
    goto dromos;
    }
  }

  dromos:

  return n;
}
```

4.3 Function return

The user-defined function *GetN* is used to solicit input in a `while` loop. If an entered number is negative, a request is made for a repeat. The input is then returned to the main program. If the input is not an integer, the `while` loop in the main program is exited and the execution terminates. A typical session follows:

```
Please enter the integer to be tested:  -10
The integer must be positive; try again
897
897 is not a prime number; the highest divisor is:  299
Please enter the integer to be tested:  q
```

Problem

4.3.2. Using a home computer, twenty-nine-year-old programmer Joel Armengaud discovered that $2^{1398269} - 1$ is a prime number. Confirm his finding.

4.3 Function return

Combinatorial

Imagine that we are given n identical objects and are asked to select from these a group of m objects, where $m = 0, 1, \ldots, n$. The number of possible combinations is given by the combinatorial,

$$p = \left(\begin{array}{c} n \\ m \end{array} \right) = \frac{n!}{m!\,(n-m)!}, \tag{1}$$

where the exclamation mark denotes the factorial,

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot n, \qquad & m! &= 1 \cdot 2 \cdot \ldots \cdot m, \\ (n-m)! &= 1 \cdot 2 \cdot \ldots \cdot (n-m), & & \end{aligned} \tag{2}$$

and the centered dot designates multiplication; by convention, $0! = 1$. When $m = 0$, we select no object, and $p = 1$; when $m = n$, we select all objects, and $p = 1$; when $m = 1$, we select one object, and $p = n$; when $m = n - 1$, we select all but one objects, and $p = n$.

4.3 Function return

The following main program contained in the file *combinatorial.cc* receives the pair (n, m) from the keyboard and calls a function to compute the combinatorial:

```
#include <iostream>
using namespace std;

int combin(int, int);

//--- main:

int main()
{
int n,m;
cout<< endl <<"Please enter n and m (n>=m);";
cout<<"'q' for either one to quit" << endl;

while(cin>>n && cin>>m)
{
if(m>n|n<0|m<0)
  {
  cout<<"Invalid input; please try again\n";
  }
else
  {
  int p = combin(n,m);
  cout<<"Combinatorial:" << p << endl;
  cout<< endl << "Please enter a new pair" << endl;
  }
}
return 0;
}
```

4.3 Function return

If a non-integer is entered for either n or m, the while loop is exited due to the false read and the execution terminates.

It would appear that the combinatorial requires the computation of three factorials. Even when f n and m are moderate, the factorial can be huge leading to memory overflow. To prevent this, we use the expression

$$p = n \cdot \frac{n-1}{2} \cdot \ldots \cdot \frac{n-k+1}{k} \cdot \ldots \frac{n-l+1}{l}, \tag{3}$$

where l is the minimum of m and $n - m$. This formula is implemented in the following function:

4.3 Function return

```
int combin(int n, int m)
{
int l,p;

//--- Find the minimum of m and n-m:

l = m;
if(n-m<l)
  {
  l=n-m;
  }

//--- Apply the formula:

p=1;
for(int k=1; k<=l;k++)
  {
  p=p*(n-k+1)/k;
  }
return p;
}
```

4.4 Function in individual files and header files

Text p.100 ~

Medium-size and large codes are split into a number of source files hosting the main program and various functions. Each file is compiled independently to produce the corresponding object file, and the object files are linked to build the executable. In C++, each file containing user-defined functions must be accompanied by a *header file* that declares these functions.

As an example, consider a code that has been split into two files. The first file named *greetings_dr.cc* contains the following main program:

```
#include "greetings.h"
using namespace std;

int main()
{
greetings();
return 0;
}
```

4.4 Function in individual files and header files

The second file named *greetings.cc* contains the following user-defined function:

```
#include <iostream>
using namespace std;

void greetings()
{
cout << "Greetings\n";
}
```

The first line of the main program is a compiler preprocessor directive requesting the attachment of the header file *greetings.h*. The significance of the double quotes in the syntax `#include "greetings.h"` is illustrated in Table 4.4.1.

4.4 Function in individual files and header files

#include <file>	*file* is a system header file provided with the compiler
#include "file"	*file* is either a user-defined header file or a system header file

Table 4.4.1 Syntax of the `include` directive system and user-defined header files. The current directory is searched first for a user-defined header file.

4.4 Function in individual files and header files

The content of the header file *greetings.h* is:

```
#ifndef GREETINGS_H

  #define GREETINGS_H

  #include<iostream>
  using namespace std;
  void greetings();

#endif
```

The "if not defined" loop checks whether the variable `GREETINGS_H` has been defined. If not, the enclosed block of commands is executed:

- The first of these commands defines the variable `GREETINGS_H`, so that the loop will not be executed if the header file is linked for a second time. By convention, the name `GREETINGS_H` arises by capitalizing the name of the header file and then appending `_H`.
- The rest of the statements in the "if not defined" loop duplicate the function preamble and declaration.

The overall procedure ensures that a function is not defined multiple times.

4.5 Function with Scalar arguments

An important concept in C++ is the distinction between global and local variables. The former are pervasive, whereas the latter are private to the individual functions.

Global variables

Global variables are defined outside the main function and user-defined functions. Because their memory addresses are communicated implicitly, their values do not need to be passed explicitly through the function argument list enclosed by parentheses.

4.5 Function with Scalar arguments

The following code employs three global variables:

```
#include <iostream>
using namespace std;

void banana(); // function declaration

double a = 2.0;
double b = 3.0;
double c;

//---- main ---

int main()
{
banana ();
cout << a << " " << b << " " << c << endl;
return 0;
}

//---- banana ---

void banana ()
{
 a = 2.0*a;
 b = -b;
 c = a+b;
 }
```

4.5 Function with Scalar arguments

Running the executable prints on the screen:

```
4 -3 1.
```

Numerical global variables do not have to be initialized, and are set to zero by the compiler.

4.5 Function with Scalar arguments

Tokyo Tech

Local variables

Local variables are defined inside the main function or user-defined functions, and are private to the individual functions.

4.5 Function with Scalar arguments

The following code employs only local variables:

```
#include <iostream>
using namespace std;

double pear (double, double); // function declaration

//---- main

int main()
{
double a = 2.5;
double b = 1.5;
double c = pear (a, b);
cout << a << " " << b << " " << c << endl;
return 0;
}

//--- pear ----

double pear (double a, double b)
{
a = 2.0*a;
b = 3.0*b;
double c = a+b;
return c;
}
```

4.5 Function with Scalar arguments

Running the executable prints on the screen:

```
2.5 1.5 9.5
```

Note that the function `pear` is unable to permanently change the values of the variables `a` and `b` defined in the main program. When these variables are communicated to the function `pear`, they are assigned new memory addresses, all calculations are done locally, and the temporary variables disappear when control is passed to the main program.

Thus, *scalar variables passed to a function are not automatically updated.* This feature of C++ represents an important difference from MATLAB and FORTRAN 77.

4.5 Function with Scalar arguments

Tokyo Tech

Referral by address

To allow a function to change the value of a communicated scalar, we specify that the scalar is *not* stored in a new memory address, but is referred instead to the original address pertinent to the calling program. This is done by employing the *reference declarator* "&", which causes the argument of the scalar to be the *memory address* instead of the actual *memory content*.

4.5 Function with Scalar arguments

The implementation of the reference declarator is illustrated in the following code:

```
#include <iostream>
using namespace std;

void melon (double, double, double&);

//--- main ---

int main()
{
double a = 2.0;
double b = 3.0;
double c;
melon (a, b, c);
cout << a << " " << b << " " << c << endl;
return 0;
}

//--- melon ----

void melon (double a, double b, double& c)
{
a = 2.0*a;
c = a+b;
cout << a << " " << b << " " << c << "; ";
}
```

4.5 Function with Scalar arguments

The reference declarator & has been appended to the variable type definition "double" both in the function `melon` prototype and in the function implementation. Running the executable prints on the screen:

```
4 3 7; 2 3 7
```

In contrast, if the reference declarators were omitted, the output would have been:

```
4 3 7; 2 3 1
```

The reference declarator must be used when a function returns one variable or a group of scalar variables through the function arguments.

4.6 Function with array arguments

Text p.109 ~

Unlike scalar variables, array variables communicated to a function are referred to the memory address allocated in the calling program. Thus, array variables are called by *reference* or by *address*. By default, user-defined functions are able to change the values of the elements of a communicated array.

4.6 Function with array arguments

The following main program contained in the file *prj.cc* calls a user-defined function to compute the inner product (projection) of two vector arrays:

```
#include <iostream>
using namespace std;

void prj (double[], double[], int, double&);

//--- main ---

int main()
{
const int n=2;
double a[n] = {0.1, 0.2};
double b[n] = {2.1, 3.1};
double prod;
prj (a,b,n,prod);
cout << "inner product:  " << prod << endl;
return 0;
}

//--- prj ---

void prj (double a[], double b[], int n, double& prod)
{
prod = 0.0;
for (int i=0;i<=n-1;i++)
  prod = prod + a[i]*b[i];
}
```

4.6 Function with array arguments

Tokyo Tech

Running this program prints on the screen:

```
inner product:  0.83
```

4.6 Function with array arguments

Constant arguments

To deny a user-defined function the privilege of changing the values of a communicated array, we declare the function attempts to change during compilation. Consider t

```
#include <iostrean
#include <cmath>

using namespace st
```

```
void squirrel (const double[], double[], int);

/*-------main---------*/

int main()
{
const int n=3;
double a[n] = {1, 2, 3};
double b[n] = {1, 4, 9};
squirrel (a, b, n);

for (int i=0; i<=n-1; i++)
  {
  cout << a[i] << " " << b[i] << endl ;
  }
return 0;
}

/*-------squirrel---------*/

void squirrel(const double a[], double b[], int n)
{
for (int i=0; i<=n-1; i++)
  {
  //   a[i] = sqrt(a[i]);
  b[i] = sqrt(b[i]);
  }
}
```

4.6 Function with array arguments

```
#include <iostream>
#include <cmath>

using namespace std;
void squirrel (const double[], double[], int);

/*-------main---------*/

int main()
{
const int n=3;
double a[n] = {1, 2, 3};
double b[n] = {1, 4, 9};
squirrel (a, b, n);

for (int i=0; i<=n-1; i++)
  {
  cout << a[i] << " " << b[i] << endl ;
  }
return 0;
}

/*-------squirrel---------*/

void squirrel(const double a[], double b[], int n)
{
for (int i=0; i<=n-1; i++)
  {
  //   a[i] = sqrt(a[i]);
  b[i] = sqrt(b[i]);
  }
}
```

4.6 Function with array arguments

If the fourth line from the end is uncommented, the compiler will produce the error message:

```
squirrel.cc:22:  assignment of read-only location
```

Tokyo Tech

Matrix arguments

In the case of vectors, we do not have to specify the length of the arrays in the argument of a function. This is not true in the case of matrices where only the length of the first index can be omitted. The reason is that C++ must know the row width in order to assess the memory addresses of the first-column elements and store the row elements in successive memory blocks.

The following code calculates and prints the sum of the diagonals (trace) of a square matrix:

4.6 Function with array arguments

```
#include <iostream>
using namespace std;

const int n=2;
void trace (double[][n], double&);

//---------------- main -----------------

int main()
{
double a[n][n] = {{0.1, 0.2}, {0.9, 0.5}};
double Trace;
trace (a, Trace);
cout << "Trace:  " << Trace << endl ;
return 0;
}

//---------------- trace -----------------

void trace (double a[][n], double& Trace)
{
Trace= 0.0;
for(int i=0; i<=n-1; i++)
  {
  Trace = Trace + a[i][i];
  }
}
```

4.6 Function with array arguments

Running the core prints on the screen:

```
Trace:  0.6
```

An alternative implementation of the `trace` function that passes the trace through the function return is:

```
double trace (double a[][n])
{
Trace=0.0;
for(int i=1; i<=n-1; i++)
  {
  Trace = Trace + a[i][i];
  }
return Trace;
}
```

In this case, the function call is:

```
double Trace = trace (double a[][n]);
```

4.7 Function with array arguments

Text p.117 ~

Assume that a code has been split into two files, one file containing the main program and the second file containing a function. Moreover, assume that the global integer variable `kokoras` is defined and possibly evaluated in the first file before the implementation of the main function.

The same global variable cannot be defined in the second file, or the linker may throw an exception on multiple variable definitions. However, if the variable is not defined in the second file, the individual compilation of this file will fail.

To circumvent this difficulty, we declare the variable in the second file as external by issuing the statement:

```
extern int kokoras;
```

which reassures the compiler that the value of this variable will be supplied externally.

4.7 Function with array arguments

As an example, the main program contained in the file *kotoula.cc*, and a function named *kalaboki* are implemented, respectively, as:

```
#include <iostream>
#include "kalaboki.h"
using namespace std;

int kokoras = 10;

int main()
{
kalaboki();
cout << kokoras << endl;
return 0;
}
```

and

```
using namespace std;

extern int kokoras;

void kalaboki()
{
kokoras++;
return kokoras;
}
```

4.7 Function with array arguments

The header file of the function `kalaboki.cc` is:

```
#ifndef KALABOKI_H
#define KALABOKI_H

using namespace std;
extern int kokoras;
void kalaboki();

#endif
```

If more than two files are involved, a variable may be declared as external in all but one file where it is defined and possibly evaluated. This may be the file hosting the main program or another file hosting a function.

4.8 Function overloading

Text p.119 ~

With the exception of the main function, two entirely different functions are allowed to have the same name, provided they have distinct lists of arguments. The compiler will realize that these are distinct functions, distinguished by the list or type of their arguments.

For example, the following code computes the inverse-distance potential of two charged particles along the x axis. When the particles coincide, the potential is infinite and a warning is issued:

4.8 Function overloading

```
#include <iostream>
using namespace std;

/*---------- regular potential ----------*/

double potential(double a, double b)
{
return 1/(a-b);
}

/*---------- singular potential ----------*/

string potential()
{
return "Warning:  singular potential";
}

/*---------- main----------*/

int main()
{
double a=1.1;
double b=2.2;
```

```
if(a!=b)
{
double V = potential(a, b);
cout << "Potential:  " << V << endl;
}
else
{
string message = potential();
cout << message << endl;
}

return 0;
}
```

4.8 Function overloading

In this case, we implement the function *potential* twice, the first time with two arguments and the second time with no arguments. The return of these functions is also different, though this is not necessary for the functions to be distinguished by the compiler.

4.9 Recursive calling

Text p.120 ~

C++ functions are allowed to call themselves in a recursive fashion that is reminiscent of a nested sequence.

For example, recursive function call can be used to compute the factorial of an integer $n! = 1 \cdot 2 \ldots \cdot n$, as implemented in the following algorithm:

```
int factorial (int n)
{
if(n==1)
  int fact = 1;
else
  fact = n * factorial(n-1);
return fact;
}
```

Recursive calling is ideal for computing self-similar objects such as fractals containing an infinite cascade of geometrical patterns. On the down side, recursive calling carries the risk of prolonged execution time.