Functions

- In C++ there is no distinction between functions, procedure, and subroutine; we use the term function for all
- Consider a function `fun`
  - with four parameters `x, y, i,` and `j`, of type `float, float, int,` and `int`, respectively
  - which computes and returns
    - » (x-y)/(i-j) , $i \neq j$
    - » 10^{20}, i = j and $x \neq y$ (with sign of (x-y))
    - » 0, i = j and $x = y$

Functions

```
// FDEMO1: Demonstration program with a function
#include <iostream.h>

int main()
{  float fun(float x, float y, int i, int j);
   float xx, yy;
   int ii, jj;
   cout << “Enter two real numbers followed by two integers:
\n”;
   cin >> xx >> yy >> ii >> jj;
   cout << “Value returned by function: “
        << fu(xx, yy, ii, jj) << endl;
   return 0;
}
```

Functions

```
 float fun(float x, float y, int i, int j)
{  float a = x - y;
   int   b = i - j;
   return b != 0 ? a/b :
          a > 0  ? +1e20 :
          a < 0  ? -1e20 : 0.0;
}
```

Functions

```
float fun(float x, float y, int i, int j)//ALTERNATIVE
{  float a, result;                          //FORMULATION
   int   b;
   a = x - y;  b = i - j;
   if (b != 0)
      result = a/b;          // non-zero denominator
   else
      if (a > 0)
         result = +1e20;     // +ve numerator
      else
         if (a < 0)
            result = -1e20; // -ve numerator
         else
            result = 0.0;   // zero numerator
   return result;
}
```

Functions

- Key points
  - `fun(xx, yy, ii, jj)` is a function call
  - `xx, yy, ii, jj` are called arguments
  - the parameters `x, y, i, j` are used as local variables within the function
  - the initial values of `x, y, i, j` correspond to the passed arguments `xx, yy, ii, jj`
  - the `return` *expression*`;` statement assigns the value of expression to the function and then returns to the calling function (`main`, in this case)

Functions

- Key points
  - The type of each parameter must be specified:

```
float fun(float x, float y, int i, int j) // correct
float fun(float x, y, int i, j) // incorrect
```

  - We can omit the parameter names in the function *declaration* (but not the *definition*) but it's not good practice

```
float fun(float, float, int, int) // declaration in
                                  // main()
```

  - A function may be delared either inside a function that contains a call to it or before it at the global level

Functions

- Key points
  - A function may be delared either inside a function that contains a call to it or before it at the global level

```
float fun(float x, float y, int i, int j);
...
main()
{
}
```

  - Global declarations are valid until the end of the file
  - Can have many declarations
  - Can have only one definition (which must not occur inside another function)

Functions

- Key points
  - If the definition occurs before the first usage of the function, there is no need to declare it (as a definition is also a declaration)
  - Function arguments can be expressions

```
float result = fun(xx+1, 2*yy, ii+2, jj-ii);
```

» NOTE: the order in which the arguments are evaluated is undefined

```
/* ill defined function call */
float result = fun(xx, yy, ++ii, ii+3);
```

The `void` Keyword

- Some functions do not return a value
- Similar to procedures and subroutines
- The functions are given the type `void`

```
void max(int x, int y, int z)
{  if (y > x) x = y;
   if (z > x) x = z;
   cout << “the maximum is “ << x << endl;
}
// Poor programming style; why?
```

The `void` Keyword

- Functions with no parameters are declared (and defined) with parameters of type `void`

```
/* read a real number */
double readreal(void)
{  double x;  char ch;
   while (scanf(“%lf”, &x) != 1)
   {  // skip rest of incorrect line
      do ch = getchar(); while (ch != ‘\n’);
      printf(“\nIncorrect. Enter a number:\n”;
   }
   return x;
}
```

The `void` Keyword

- In C omission of `void` would have implied that the function could have had any number of parameters
- In C++ omission of information about parameters is not allowed
  - no parameters means NO parameters
  - `double readreal(void)` = `double readreal()`

The `void` Keyword

- In C one normally writes

```
main()
```

- This is equivalent to:

```
int main()
```

not `void main()` and implies `return 0;` at the end of the main function.

The `void` Keyword

- It makes sense to adopt the `int` formulation (with the required return `statement`) since the operating system often picks up the main return value as a run-time error code

Global Variables

- Local variables
  - defined within a function
  - visible only within that function (local scope)
- Global variables
  - defined outside functions
  - visible from the point of definition to end of the file (global scope)
  - modification of a global variable by a function is called a side-effect ... to be avoided

Global Variables

- Scope Rules
  - If two variables have the same name and are both in scope, then the one with local scope is used
- Scope resolution operator ::
  - C++ enables us to explicitly over-ride the local scope rule
  - Indicate that the global variable is meant by writing the scope-resolution-operator in front of the variable name

Scope Resolution

```
#include <iostream.h>;
int i = 1;

int main()
{  int i=2;
   cout << ::i << endl; // Output: 1 (global variable)
   cout << i   << endl; // Output: 2 (local variable)
   return 0;
}

// more on :: in the section on structures and classes
```

Functions
Altering Variables via Parameters

- C++ allows reference parameters

```
void swap1(int &x, int &y)
{  int temp;
   temp = x;
   x = y;
   y = temp;
}
```

- The arguments of `swap1` must be lvalues

Functions
Altering Variables via Parameters

- C does not allow reference parameters
- Instead of passing parameters by reference
  - we pass the address of the argument
  - and access the parameter indirectly in the function
- & (unary operator)
  - address of the object given by the operand
- * (unary operator)
  - object that has the address given by the operand

Functions
Altering Variables via Parameters

```
void swap1(int *p, int *q)
{  int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}
```

- `swap(&i, &j); // function call`

Functions
Altering Variables via Parameters

- Pointers
  - `*p` has type `int`
  - But `p` is the parameter, not `*p`
  - Variables that have addresses as their values are called *pointers*
  - `p` is a pointer
  - The type of `p` is *pointer-to-int*

Functions
Types of Arguments and Return Values

- Argument Types
  - Function arguments are automatically converted to the required parameter types, if possible
  - If it is not, an (compile) error message is given
  - Thus it is valid to pass an integer argument to a function with a float parameter
  - The same rules for type conversion apply as did in assignment statements

Functions
Types of Arguments and Return Values

- Types of return values
  - The conversion rules also apply to return-statements

```
int g(double x, double y)
{  return x * x - y * y + 1;
}
```

  - Since the definition says that `g` has type `int`, the value returned is `int` and truncation take place

Functions
Types of Arguments and Return Values

- Types of return values
  - It would be better to explicitly acknowledge this with a cast

```
int g(double x, double y)
{  return int (x * x - y * y + 1);
}
```

Functions
Initialization

- Variables can be initialized when the are declared. However:
  - Variables can be initialized only when memory locations are assigned to them
  - In the absence of explicit initialization, the intial value 0 is assigned to all variables that are `global` or `static`

Functions
Initialization

-
  - `global`
    - » variables that are declared outside functions
    - » have a permanent memory location
    - » can only be intialized with a constant expression
  - `static`
    - » a keyword used in the declaration to enforce the allocation of a permanent memory location
    - » static variables local to a function are initialized ONLY the first time the function is called
    - » can only be intialized with a constant expression

Functions
Initialization

-
  - `auto`
    - » a keyword used in the declaration to enforce the (default) allocation of memory from the stack
    - » such variables are called automatic
    - » memory associated with automatic variables is released when the functions in which they are declared are exited
    - » unless initialized, you should not assume automatic variables have an initial value of 0 (or any value)
    - » can be intialized with any valid expression (not necessarily a constant expression

Functions
Initialization

```
#include <iostream.h>

void f()
{  static int i=1;
   cout << i++ << endl;
}

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

Functions
Initialization

```
#include <iostream.h>

void f()
{  static int i=1;
   cout << i++ << endl;
}

int main()
{  f();           // prints 1
   f();           // prints 2
   return 0;
}
```

Functions
Initialization

- Uses of local static variables
  – For example, as a flag which can indicate the first time a function is called

```
void f()
{  static int first_time = 1;
   if (first_time)
   {  cout <<
      “f called for the first time\n”;
      first_time = 0;  // false
   }
   cout << “f called (every time)\n”;
}
```

Functions
Initialization

- Initialization of arrays
  - write the initial values in braces
  - There must not be more initial values than there are array elements
  - There can be fewer (but at least one!)
  - Trailing elements are initialized to 0

```
float a[a100] = {23, 41.5};
// a[0]=23; a[1]=41.5; a[2]= ... = a[99]= 0

char str[16] = “Charles Handy”;
```

Functions
Initialization

- Default arguments
  - C++ allows a function to be called with fewer arguments than there are parameters
  - Must supply the parameters with default argument values (i.e. initialized parameters)
  - Once a parameter is initialized, all subsequent parameters must also be initialized

```
void f(int i, float x=0, char ch='A')
{  ...
}
```

Functions
Initialization

```
void f(int i, float x=0, char ch='A')
{  ...
}
...
f(5, 1.23, 'E');
f(5, 1.23);  // equivalent to f(5,1.23,'A');
f(5);        // equivalent to f(5,0,'A');
```

Functions
Initialization

- Default arguments
  - functions which are both defined and declared can also have default argument
  - Default value may only be specified once, either in the declaration or in the definition

```
// declaration
void f(int i, float, char ch};

// definition
void f(int i, float x=0; char ch='A'}
{  ...
}
```

Functions
Separate Compilation

- Large programs can be split into modules, compiled separately, and subsequently linked

Functions
Separate Compilation

```
// MODULE1
#include <iostream>
int main()
{  void f(int i), g(void);
   extern int n; // declaration of n
                 // (not a definition)
   f(8);
   n++;
   g();
   cout << “End of program.\n”;
   return 0;
}
```

Functions
Separate Compilation

```
// MODULE2
#include <iostream>

int n=100;      // Defintion of n (also a declaration)
static int m=7;

void f(int i)
{  n += i+m;
}

void g(void)
{  cout << “n = “ << n << endl;
}
```

Functions
Separate Compilation

- Key points
  - `n` is used in both modules
    - » defined in module 2
    - » declared (to be extern) in module 1
  - `f()` and `g()` are used in module 1
    - » defined in module 2
    - » declared in module 1
  - A variable may only be used after it has been declared
  - Only definitions reserve memory (and hence can only be used with initializations)

Functions
Separate Compilation

- Key points
  - we defined a variable only once
  - we can declare it many times
  - a variable declaration at the global level (outside a function) is valid from that declaration until the end of the file (*global scope*)
  - a declaration inside a function is valid only in that function (*local scope*)
  - we don't need to use the keyword `extern` with functions

Functions
Separate Compilation

- Key points
  - `static int m=7;`
    - » `m` is already global and so its memory location is permanent
    - » Thus, the keyword `static` might seem unnessary;
    - » However, `static` global variables are the private to the module in which they occur
    - » cannot write

```
extern int m;  // error
```

Functions
Separate Compilation

- Key points
  - Static can also be used with functions
    - » This makes them private to the module in which they are defined
  - The keyword `static`, both for global variables and for functions, is very for
    - » avoiding name-space pollution
    - » restricting scope and usage to instances where usage is intended
    - » Avoid global variables (and make them static if you must use them)
    - » make functions static if they are private to your code

Functions
Standard Mathematical Functions

- Declare standard maths functions by

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

- `double cos(double x);`
- `double sin(double x);`
- `double tan(double x);`
- `double exp(double x);`
- `double ln(double x);`
- `double log10(double x);`
- `double pow(double x, double y); // x to the y`
- `double sqrt(double x);`
- `double floor(double x); // truncate`
- `double ceil(double x);  // round up`
- `double fabs(double x);  // |x|`

Functions
Standard Mathematical Functions

- `double acos(double x);`
- `double asin(double x);`
- `double atan(double x); // -pi/2 .. +pi/2`
- `double atan2(double y, double x);`
- `double cosh(double x);`
- `double sinh(double x);`
- `double tanh(double x);`

- `abs` and `labs` are defined in `stdlib.h` but return integer values

Functions
Function Overloading

- C++ allows the definition of two or more functions with the same name
- This is known as *Function Overloading*
  - number or types of parameters must differ
  - ```
    void writenum(int i)   // function 1
    {  printf(“%10d”, i);
    }

    void writenum(float x) // function 2
    {  printf(%10.4f”, x);
    }
    ```
    ```

Functions
Function Overloading

```
writenum(expression);
```

- function 1 is called if expression is type int
- function 2 is called if expression is type float

- The functions are distinguished by their parameter types and parameter numbers

Functions
Function Overloading

- Allowable or not? ....

```
int g(int n)
{ ...
}

float g(int n)
{ ...
}
```

Functions
Function Overloading

- Allowable or not? ....

```
int g(int n)
{ ...
}

float g(int n)
{ ...
}
```

- Not! parameters don’t differ.

Functions
Function Overloading

- Type-safe linkage
  - Differentiation between functions is facilitated by name mangling
    - » coded information about the parameters is appended to the function name
    - » all this information is used by the linker

Functions
Function Overloading

- Type-safe linkage
  - Of use even if not using function overloading:

```
void f(float n) // definition in
{ ...           // module 1
}               // only one defn. of f

void f(int i);  // declaration in
}               // module 2
```

- C++ compilers will catch this; C compilers won’t

Functions
References as Return Values

- The return value can also be a reference (just as parameters can be reference parameters)

Functions
References as Return Values

```
//REFFUN: Reference as return value.
#include <iostream.h>
int &smaller(int &x, int &y)
{  return (x < y ? x : y);
}
int main()
{  int a=23, b=15;
   cout << “a = “ << a << “ b = “ << b << endl;
   cout << “The smaller of these is “
   << smaller(a, b) << endl;
   smaller(a, b) = 0; // a function on the LHS!
   cout << “ The smaller of a and b is set to 0:”;
   cout << “a = “  << a << “ b = “ << b << endl;
   return 0;
```

Functions
References as Return Values

- Key points about the assignment

  `smaller(a, b) = 0;`
- Function `smaller` returns the argument itself (i.e. either `a` or `b`)
- This gets assigned the value 0
- The arguments must be variables
- The `&` must be used with the parameters
- The returned value must exist outside the scope of the function

Functions
Inline Functions and Macros

- A call to a function causes
  - a jump to a separate and unique code segment
  - the passing and returning of arguments and function values
- This trades off time efficiency in favour of space efficiency
- Inline functions cause
  - no jump or parameter passing
  - duplication of the code segment in place of the function call

Functions
References as Return Values

```
 inline int sum(int n)
{  return n*(n+1)/2; // 1+2+ ... n
}
```

- Should only use for time-critical code
- and for short functions
- Inline functions are available only in C++, not C

Functions
Inline Functions and Macros

- In C we would have used a macro to achieve the effect of inline functions
  – define a macro
  – macro expansion occurs every time the compiler preprocessor meets the macro reference
  – for example

```
#define sum(n) ((n)*((n)+1); // note ()
```

Functions
Inline Functions and Macros

– The following macro call

```
y = 1.0 / sum(k+1)/2;
```

– expands to

```
y = 1.0 / ((k+1) * ((k+1)+1)/2);
```

Functions
Inline Functions and Macros

– If we had defined the macro without full use of parentheses

```
#define sum(n) n*(n+1)/2;
```

– the expansion would have been

```
y = 1.0 / k+1 * (k+1+1)/2;
```

– which is seriously wrong ... why?

Functions
Inline Functions and Macros

- Some macros have no parameters

```
#define LENGTH 100

#define ID_number(i) array_id[i];
```

- Since macro expansion occurs at preprocessor stage, compilation errors refer to the expanded text and make no reference to the macro definition per se

Functions
Inline Functions and Macros

```
#define f(x) ((x)*(x)+(x)+1);
...
  y = f(a) * f(b);
```

produces the syntactically incorrect code
(and a possibly confusing “invalid indirection” error)

```
y = ((a)*(a)+(a)+1); * ((b)*(b)+(b)+1);;
```

Functions
Inline Functions and Macros

- Previously defined macros can be used in the definition of a macro.
- Macros cannot call themselves
  - if, in a macro definition, its own name is used then this name is not expanded

```
 #define cos(x) cos((x) * PI/180)
//cos (a+b) expands to cos((a+b))*PI/180)
```

Functions
Inline Functions and Macros

- A macro can be defined more than once
  - The replacement text MUST be identical

```
#define LENGTH 100
...
#define LENGTH 1000 // not allowed
```

  - Consequently, the same macro can now be defined in more than one header file
  - And it is valid to include several such header files in the same program file

Functions
Inline Functions and Macros

- The string generating character #
  - In macro definitions, parameters immediately preceded by a # are surrounded by double quotes in the macro expansion

```
#define print_value(x) printf(#x “ = %f\n”, x)

...

print_value(temperature);
// expands to printf(“temperature” “ = %f\n”, temperature);
```

  - Consequently, the same macro can now be defined in more than one header file

Functions

Other Preprocessor Facilities

- Header files
  - The preprocessor also expands #include lines

```
#include <stdio.h>
#include “myfile.h”
```

  - The two lines are logically replaced by the contents of these header files

`<...>` search for the header file only in the general include directories

`“...”` search in the current directory first, then search in the general include direct.

Functions

Other Preprocessor Facilities

- Header files
  - normally used to declare functions and to define macros
  - included in several module files
  - header files can also include files
  - function definition should NOT be written in header files (except, perhaps, inline functions)

Functions

Other Preprocessor Facilities

- Conditional compilation
  - compile a program fragment (A) only if a certain condition is met

```
#if constant expression
    program fragment A
#else
    program fragment B
#endif
```

  - The #else clause is optional

Functions

Other Preprocessor Facilities

- Conditional compilation
  - a useful way to ‘comment out’ large sections of text which comprises statements and comments
  - (remember, we can’t nest comments)

```
#if SKIP
    /* lots of statements */
   a = PI;
   ...
#endif
```

Functions

Other Preprocessor Facilities

- Tests about names being known

```
#if !defined(PI)
#define PI 3.14159265358979
#endif
```

  - `defined()` can be used with the logical operators `!`, `||`, and `&&`
  - Older forms:

    `#ifdef` *`name`* is equivalent to `#if defined (`*`name`*`)`

    `#ifndef` *`name`* is equivalent to `#if !defined (`*`name`*`)`

Functions

Other Preprocessor Facilities

- Tests about names being known

```
#undef PI
```

undefines a name (even if it hasn't been defined)

Functions

Other Preprocessor Facilities

- Making the compiler print error messages

```
#include "myfile.h"
#if !(defined(V_B)
#error You should use Ver. B of myfile.h
#endif
```

Compilation terminates after printing the error message

Functions
Other Preprocessor Facilities

- Predefined names
  - can be used in constant expressions

`__LINE__`	integer: current line number
`__FILE__`	string: current file being compiled
`__DATE__`	string: date in the form M*mm dd yyyy* *(date of compilation)*
`__TIME__`	string: date in the form M*mm dd yyyy* *(time of compilation)*
`__cplusplus`	a constant defined only if we are using a C++ compiler

Functions

Exercises

7. Write and test a function sort4, which has four parameters. If the integer variables a, b, c, and d are available and have been assigned values, we wish to write:

   ```
   sort4(&a, &b, &c, &d);
   ```

   to sort these four variables, so that, after this call, we have a<= b <= c <= d
8. Write and test a function sort4_2 which uses reference parameters

Functions

Exercises

9. Investigate (on paper and then with a computer) the effect of the following recursive function and calling program with values k=0,1,2,...5

```
sort4(&a, &b, &c, &d);
```

Functions

Exercises

```
#include <iostream.h>
void f(int n)
{  if (n > 0)
   {  f(n-2); cout << n << “ “; f(n-1);
   }
}

int main()
{  int k;
   cout << “Enter k: “; cin >> k;
   cout << “Output:\n”;
   f(k);
   return 0;
}
```

Functions

Exercises

10. Write and test a (recursive) function gcd(x, y) which computes the greatest common divisor of the integers x and y. These two integers are non-negative and not both equal to zero. Use Euclid’s algorithm:

```
gcd(x,y)      =     x                    if y =0
                    gcd(y, x%y)          if y!=0
```