22.5HV2
SOFTWARE ENGINEERING II

Functions in C++

Aim

- This unit will consider the following topics:
  - The use of abstraction and top-down design in programming.
  - The need for functions.
  - Declaring and defining functions.
  - Returning values from functions
  - Call-by-value functions
  - Call-by-reference functions
  - Default parameters
  - Function overloading
  - Operator overloading
  - Inline functions

Abstraction

- **An important concept in programming is that of *abstraction*.**
  - **Abstraction means that we don't have to know or understand the details of a program component in order to use it.**
  - **The concept of abstraction means that when tackling problems we don't have to consider all the details that would otherwise drive us to distraction.**
- **C, the programming language that C++ was developed from was built around the concept of functional abstraction.**
  - **To consider what is meant by this, we will consider the example of the Apollo space program undertaken by NASA in the 1960's.**

Abstraction in the Apollo program

- One of the largest projects undertaken by man, was the Apollo moon landing program.
  - The "top" in this problem was specified by President Kennedy as *"… to land a man on the moon and return him safely"*
  - The actual implementation of this project required billions of details regarding the design of each individual component, planning, logistics etc.
  - No one individual could comprehend the project in its entire complexity - each participant used abstraction.
    - The capsule designers thought of the rocket as a source of propulsion.
    - The rocket designers simply considered the capsule as a mass.

Abstraction in daily life

- Consider a housewife (or househusband!) that has the following tasks to achieve in a day:

Buy stamps
Pay TV licence
Buy onions, tomatoes, mushrooms, rice
Collect shoes from repair shop
Hoover living room
Clean bathroom
Clean windows
Do washing
Chop onions and garlic
Fry in oil until golden
Chop mushrooms
Add to pot
etc . . .

Abstraction in daily life

- This list clearly explains all that needs to be done in the day, but a better approach would be to have a simple list:

Do shopping
Do housework
Make dinner

  - Each task could then be described in more <u>detail</u>:

Buy stamps
Pay TV licence
Buy onions, tomatoes, mushrooms, rice
Collect shoes from repair sh

shopping list

Hoover living room
Clean bathroom
Clean windows
Do washing

housework

Chop onions and garlic
Fry in oil until golden
Chop mushrooms
Add to pot
etc . . .

recipe

Advantages of abstraction

- **Both approaches will achieve the same goal, but the second has the following advantages:**
  - **It is clearer and easier to understand.**
  - **It is easier to modify - for example if you wanted a different meal you can simply change the recipe. The concept of "make dinner" is unchanged.**
  - **It is easier to reuse. You may have the same housework and meal the next day, but different shopping - simply alter the shopping list.**
- **Some people write programs like the first version:**
  - **Consequently, they are difficult to understand, modify and reuse.**

Advantages of abstraction

- One of the biggest advantages of abstraction is that it allows us to use something without understanding the details:
  - People can drive cars without having to understand the operation of the internal combustion engine.
  - Once you have learnt to drive a car, you can drive one with a petrol, diesel or electric engine.
  - The method by which the different engines achieve motive power may change, but the interface to the driver remains unchanged (accelerator, brake, clutch, gears, steering wheel).
- As we can see from the above, this allows us to change the way something is done, without changing how it is used.

Abstraction in programming

- In programming, this kind of abstraction is achieved using **functions**.
  - We have already used functions that are contained in the `math.h` library, e.g. `sin()`.
- A function is a small unit of a program that contains its own statements and variables.
  - A function can be *called* from other parts of the program.
- Each C++ program must contain at least one function which is called when the program is executed.
  - This function is called `main()`.
  - Other functions can be called from `main()` to implement specific parts of the program.

Top-down design using functions

- A wise programmer once wrote:

  "*writing a program is like eating an elephant: it is easier if you break it up into chunks first*"

- When we perform a top-down design of a program:
  - We start with an abstraction about the operation of the whole program.
  - We divide this into a number of smaller abstractions and iterate until we arrive at the actual details of the implementation.
  - This process divides the problem up until they become a size that is manageable by a single programmer.
  - The thing(s) that the programmer writes will most likely be a **function**.

Top-down design using functions

- A good candidate for a function is:
  - A piece of code that performs a self-contained task.
    - This increases the idea of *abstraction*, as it hides the details of an easily identifiable sub-task.
    - This also enhances the ability to *modify* the program.
  - A piece of code that is used frequently in a program.
    - This increases the idea of *reuse*.
    - It reduces the size of the program.

Libraries of functions

- Some pieces of code are used so frequently, that they are used in many different programs.
  - To do this, we create a *library of functions*, and include this library into our code.
  - We have already used function libraries such as `math.h`.

- A team of programmers involved on the development of a large project will have different specialisations.
  - By creating libraries of functions, they can use the work of other specialists, in their own area of expertise.
  - e.g. a financial expert programmer could use the functions created by a network specialist in his financial transactions.

The mathematical function library in C++

- We have already used the function `sin()` from the `math.h` library. e.g.:

```
cout << degree << " " << sin(radian) << endl;
```

- The statement `sin(radian)` *calls* the function `sin()` with a single *parameter* `radian`. This function *returns* the floating point value corresponding to the sine of this angle.
  - A function *call* passes control of execution into the body of code contained in the function.
  - A function *parameter* allows the transfer of data to the function.
  - A function may *return* a value which can be used in an arithmetic or logical statement, an output statement or the rhs of an assignment.

The mathematical function library in C++

- Other functions in `math.h` include:

`acos(x)`	inverse cosine (in radians)
`asin(x)`	inverse sine (in radians)
`atan(x)`	inverse tangent (in radians)
`cos(x)`	cosine of `x` (`x` in radians)
`sin(x)`	sine of `x` (`x` in radians)
`tan(x)`	tangent of `x` (`x` in radians)
`exp(x)`	exponential function of `x`
`log(x)`	natural log of `x`
`sqrt(x)`	square root of `x`
`fabs(x)`	absolute value of `x`
`floor(x)`	largest integer not greater than `x`
`ceil(x)`	smallest integer not less than `x`

User defined functions

- C++ allows programmers to define their own functions:
  - Below is the definition of a function that takes as parameters, the coords of a point (x,y) and will return its distance from the origin:

```
float distance(float x, float y)
{
   float dist;  // local variable
   dist = sqrt(x*x + y*y);
   return dist;
}
```

  - The function has 2 *parameters* `x` and `y` which are of type `float`.
  - The function has a *local variable* `dist`.
  - The function *returns* this value as a `float`.

User defined functions

- The general form of a function definition in C++ is:

```
return_type function_name( parameter_list )
{
   local_definitions
   function_implementation
}
```

  - If the function returns a value then the function must be given the *return_type* of the returned value.
  - The *function_name* follows the same rules as for variable identifiers.
  - The *parameter_list* lists the function parameters and their types.
  - The *local_definitions* are the definitions of variables that are used in the function. These variables only "exist" within the function.
  - The *function_implementation* consists of the C++ statements that implement the function operation.

Functions with no parameters

- The simplest type of function is one that has no parameters and does not return anything.
  - A function that does not return anything is given the type `void`.
  - The function `main()` in most of our examples is such a function:

```
void main()
{
   // program statements
}
```

  - `main()` is the function that is called when the program is first executed.
  - It is possible for `main()` to accept parameters (called command line arguments) from the system, and return a value to the system.

Functions with no parameters

- A user defined function with no parameters and no returned value is shown below:

```
void skipthree() // skips 3 lines on output
{
   cout << endl << endl << endl;
}
```

  - The return type of the function is given as `void`, as nothing is returned from the function.
  - The parameter list is empty as no parameters are required.
  - There are no local variables required by this function.

Calling the function

- This function may be called from within another function (for example `main()`) as follows:

```
cout << "Hello";
skipthree();
cout << "there";
```

output

```
Hello


there
```

  - The function is called as a single statement containing the name of the function and the empty brackets, terminated by a semi-colon.

  - As the function does not return anything, it cannot be used as part of an arithmetic or logical expression or output statement.

Declaring functions

- **In the same way that it is necessary to declare variables before their use, we must *declare* a function prior to calling.**
  - **The reason for this is that the compiler must know the name of valid functions, their parameter lists and return types before their use.**
  - **The simplest way to achieve this, is to insert the function definition before the calling function (in this case `main()`):**

```
#include <iostream.h>
void skipthree() // function definition
{
   cout << endl << endl << endl;
}
void main()
{
   cout << "Hello";
   skipthree();  // function call
   cout << "there";
}
```

A word about declarations and definitions

- A *declaration* tells the compiler about a variable or function:
  - the type and identifier in the case of a variable.
  - the return type, name and parameters in the case of a function.

  declarations generate no code and reserve no memory

- A *definition* creates the variable or function:
  - the memory required to store the variable is reserved.
  - the code associated with a function is specified.

- In single file programs, the variables are declared and defined at the same time.

- The previous slide shows a function being declared and defined at the same time.

Why we don't do it this way?

- In general we will not place the function definition before `main()` for the following reasons:
  - The functions contain the detail of the program implementation, whilst `main()` contains the overall operation of the program at a higher level of abstraction. Hence anyone reading the program will read `main()` first to understand the program operation, so it makes sense to have `main()` at the top of your program listing.
  - When we use functions from a library, the main program is linked with the pre-compiled object code of the functions. Hence we only require the function declarations and not the function definitions. Hence the file `math.h` contains the declarations of the math library functions. The functions themselves are pre-compiled into object code and linked by the compiler.

Function prototypes

- Another name for a function declaration is a *function prototype*.

- The function prototype for our simple function is:

```
void skipthree();
```

  - This tells the compiler of the return type, the function name and the (empty) parameter list.

  - As this is a statement, it must end in a semicolon.

Using function prototypes

- We have the function prototype first, then `main()` including the function call and lastly the function definition.

```
#include <iostream.h>
void skipthree(); // function prototype
void main()
{
   cout << "Hello";
   skipthree();   // function call
   cout << "there";
}
void skipthree()  // function definition
{
   cout << endl << endl << endl;
}
```

Using several functions

- If we had a program that required several functions, we would place all of the function prototypes at the top of the file (after `#include <iostream.h>` etc), and all the function definitions after `main()`.

- It is often more convenient to place all of the function prototypes (and other declarations) in a *header file* (with a `.h` file extension), and place the function definitions in a separate file that may be compiled into object code independently and then linked to the main program code.

  - More of this later.

Functions with parameters and no return type

- The previous function is not very useful - suppose we wished to skip 4 lines or 2 lines.
  - It is more useful to be able to tell the function the number of lines to skip.
  - In other words, the function should have an *input parameter* that indicates the number of lines to be skipped.

```
void skip(int n)
{
   int i;  // local variable
   for (i=0;i<n;i++)
      cout << endl;
}
```

Functions with parameters and no return type

- The function `skip()` takes a single integer parameter `n` that specifies the number of lines to be skipped.
  - The parameter list consists of the type and identifier for this parameter.
  - As nothing is returned, the function has a return type of `void`.

```
void skip(int n)
{
   int i;  // local variable
   for (i=0; i<n; i++)
      cout << endl;
}
```

Functions with parameters and no return type

```
#include <iostream.h>
void skip(int);  // function prototype
void main()
{
   int  m=6, n=3;
   skip(m);       // parameter integer variable
   skip(m+n);     // parameter integer expression
   skip(4);       // parameter integer constant
   //...
}
void skip(int n) // function definition
{
   int i;  // local variable
   for (i=0; i<n; i++)
      cout << endl;
}
```

Functions with parameters and no return type

- The function prototype must now include the type of each parameter, but not the identifier.

```
void skip(int);
```

- The parameter used in the function call could be:
  - an integer variable.

```
skip(m);
```

  - an expression evaluating to an integer.

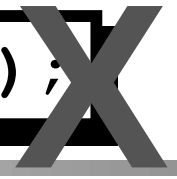
```
skip(m+n);
```

  - an integer value.

```
skip(4);
```

- The parameter must not be any other type:

```
skip("4");
```



Functions that return values

- A useful type of function is one that returns a value that is a function of its parameters.
  - Each of the mathematical functions in math.h library is such a function, e.g. `sin(x)`, `fabs(x)`.

- Such functions must be given the same type as the value that is returned.
  - Consider the previous example function for the computation of the distance of a point from the origin:

```
float distance(float x, float y)
{
   float dist;  // local variable
   dist = sqrt(x*x + y*y);
   return dist;
}
```

Functions that return values

- The function prototype of this function will be:

```
float distance(float, float);
```

- The function has a return type of `float`, as it is going to return a `float` value.

- The parameter list has two parameters: `x` and `y`. The parameters are declared by listing the type and identifier with each parameter separated by commas.

```
float distance(float x, float y)
```

Functions that return values

- The local variable `dist` is declared within the function, to temporarily hold the computed distance.

```
dist = sqrt(x*x + y*y);
```

- The `return` statement is used to return this computed value back to the statement from which the function was called.

```
return dist;
```

- We could have avoided using the local variable `dist`, by simply following the `return` keyword with the expression that computes the distance:

```
return sqrt(x*x + y*y);
```

Functions that return values

- As the function returns a value, it can only be used in an expression or output operation:

```
float  a, b, c, d, x, y;
a = 3.0;
b = 4.4;
c = 5.1;
d = 2.6;
x = distance(a,b);
y = distance(c,d);
cout << distance(a,d) << endl;
if ( distance(4.1,6.7) > distance(x,y) )
   cout << "Message 1" << endl;
```

```
distance(a,b);
```

Functions with several `return` statements

- As well as returning the value back to the calling statement, the `return` statement also stops execution of the function.
  - This means that we can have more than one `return` statement when used in conjunction with conditional statements:
  - EXAMPLE: Opening file from within `main()` :

```
int main()
{
   ifstream ins;
   ins.open("infile.dat");
   if ( ins.fail() ) {
      cerr << "Error opening infile.dat" << endl;
      return 1;  // exit program in error
   }
   // rest of program
   return 0;      // exit program correctly
}
```

EXAMPLE: Iterative evaluation of square root

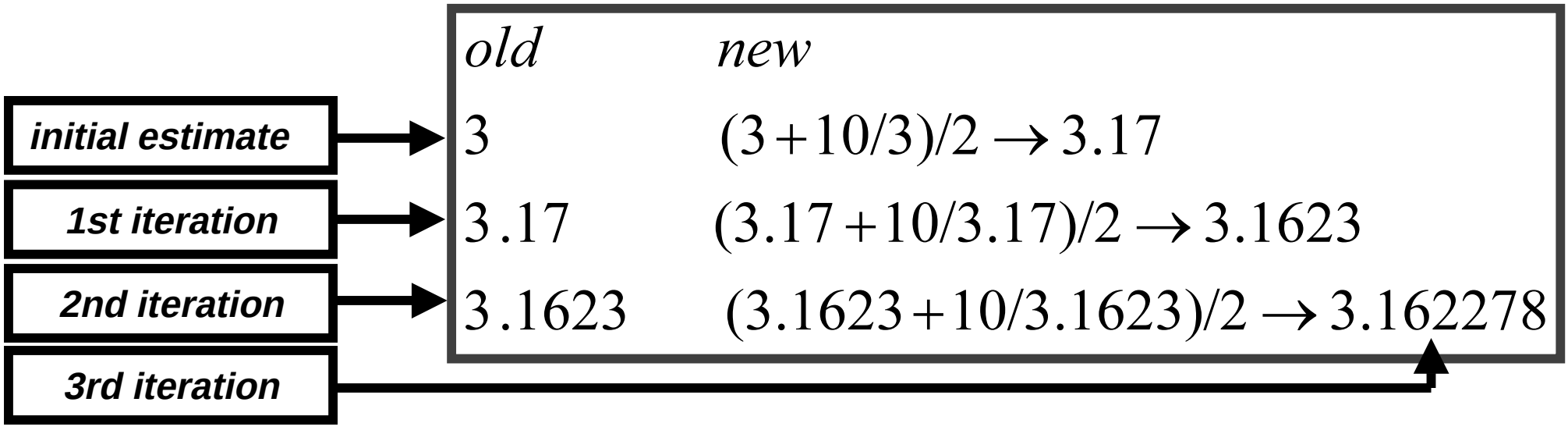
- We can find the square root of a number by performing the following iteration:
  - If *old* is an approximation to the square root of x, then a better approximation *new* can be found as:

$$new = (old + \frac{x}{old}) / 2$$

  - By repeating this expression, we can arrive at a solution with the required accuracy.
  - This method will work for all positive values of x, as long as the first approximation chosen is positive.

EXAMPLE: Iterative evaluation of square root

- For example we can compute the square root of 10 as follows (actual answer 3.16227766):



  - Before completing our algorithm, it is necessary to consider how we can determine if two successive estimates are "close enough".

EXAMPLE: Iterative evaluation of square root

- For an accuracy of 3 decimal places, we could stop once 2 successive approximations differ by less than 0.0005.
  - We may attempt to write this in C++ as:

```
while ( (new_app - old_app) > 0.0005 )
```

X

  - However, this ignores the fact that a new approximation may be <u>less than</u> the previous by more than 0.0005. Hence we test the absolute value of the difference using the `fabs()` function:

```
while ( fabs(new_app - old_app) > 0.0005 )
```

  - The `fabs()` function returns the absolute value of a floating point nunber, e.g. `fabs(1.2)` equals 1.2, `fabs(-3.7)` equals 3.7.

EXAMPLE: Iterative evaluation of square root

- A further problem exists, due to specifying an accuracy in decimal places:

exact value	approximation
100	100.1
0.1	0.2

  - Both examples have an absolute error of 0.1, but this represents 0.1% of the first value and 100% of the second value!

  - This would cause problems in finding the square root of small numbers. To compensate for this we shall test the relative error against the specified number of places:

```
while ( fabs(new_app-old_app)/new_app > 0.0005 )
```

EXAMPLE: Computing the square root

```
double mysqrt(double x)
{
   const double  tol=1.0e-5;          // 5 sig. figures
   double        xold, xnew;
   if (x == 0.0)  return 0.0;         // in case x=0.0
   else {
      xold = x;                       // first approx
      xnew = 0.5*(xold+x/xold);       // refine approx
      while ( fabs( (xold-xnew)/xnew ) > tol ) {
         xold = xnew;
         xnew = 0.5*(xold+x/xold); // refine approx
      }
      return xnew;                    // square root
   }
}
```

EXAMPLE: Sum of squares of integers

```
int sumsq(int n)
{
   int  sum=0;
   int  i;
   for (i=1; i<=n; i++)
      sum += i*i;
   return sum;
}
```

definition

use

```
int  sumsquare;
int  number;
cout << "Enter number (>= 0): ";
cin >> number;
sumsquare = sumsq(number);
```

EXAMPLE: Raising to the power

- This function returns the value of the first parameter raised to the power of the second parameter.

```
float power(float x, int n)
{
   float product = 1.0;
   int   absn, i;
   if ( n == 0 ) return 1.0;
   else {
      absn = abs(n);
      for (i=1; i<=absn; i++)
         product *= x;
      if (n < 0) return 1.0/product;
      else return product;
   }
}
```

EXAMPLE: Raising to the power

```
float power(float x, int n)
{
   float product = 1.0;
   int   absn, i;
   if ( n == 0 ) return 1.0;
   else {
      absn = abs(n);
      for (i=1; i<=absn; i++)
         product *= x;
      if (n < 0) return 1.0/product;
      else return product;
   }
}
```

```
float  x,y;
int    p;
cout << "Enter a float and an integer: ";
cin >> x >> p;
y = power(x,p);
```

Call-by-value parameters

- Consider the following function and code:

```
void add2(int x)
{
   x += 2;
}
```

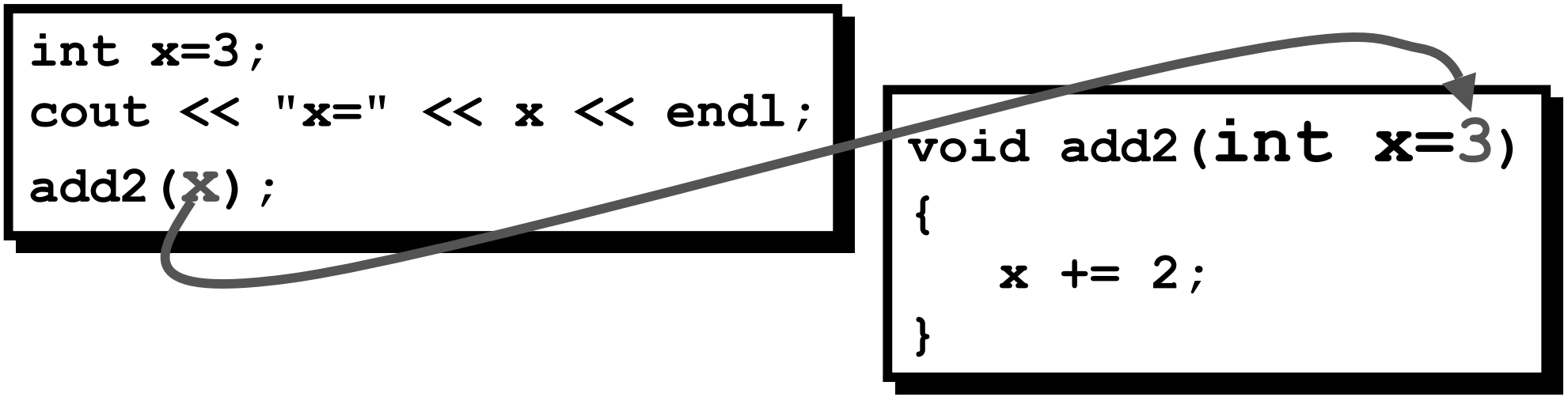
```
int x=3;
cout << "x=" << x << endl;
add2(x);
cout << "x=" << x << endl;
```

- Q: What is output?

```
x=3
x=3
```

Call-by-value parameters

- When we call the function `add2()` with the parameter `x`, the value of `x` (in this case 3) is simply copied to the function's parameter variable, also called `x`.
  - This is equivalent to initialising the parameter variable `x` with 3:



  - The statement `x+=2` inside the function will change the value of the function's local variable `x`, but cannot change the value of the variable `x` from the calling function.

Call-by-value parameters

- To explain this better, let us rename the function parameter as `y`, and include an output statement in `add2()`:

```
void add2(int y)
{
   y += 2;
   cout << "y=" << endl;
}
```

```
int x=3;
cout << "x=" << x << endl;
add2(x);
cout << "x=" << x << endl;
```

- OUTPUT:

```
x=3
y=5
x=3
```

Call-by-value parameters

- **Call-by-value parameters are used when we wish to transfer some data into a function.**
  - **Call-by-value parameters are declared by simply providing the type and identifier in the parameter list.**
  - **Anything that happens inside the function to the copy of the value of the parameter, cannot affect the original parameter.**

Call-by-reference parameters

- **So far we have seen that we can transfer data *into* a function by using a call-by-value parameter, and we can return a value *from* a function using the `return` statement.**
  - **It is possible to pass a number of separate values into a function, by using a separate parameter for each piece of data.**
  - **However, we can only ever return a single data item.**
  - **In many applications, we need to return several pieces of information from a function. E.g. a function that takes an amount of money in pence and returns the equivalent in pounds and pence.**
- ***Call-by-reference parameters* allow a function to return more than one piece of data.**

Reference variables

- Before we discuss call-by-reference functions, we must introduce the idea of a reference variable.
  - A variable can be declared to be a *reference* to another variable by prefixing the variable's name with the reference operator (`&`), and initialising it with the variable to which it is to refer to.

```
int i=1, &ref=i;  // ref is a reference to i
```

  - The reference variable can then be used in place of the variable to which it is the reference.

```
ref++;          // i is now 2
```

  - A reference variable may be considered to be an alias to another variable.

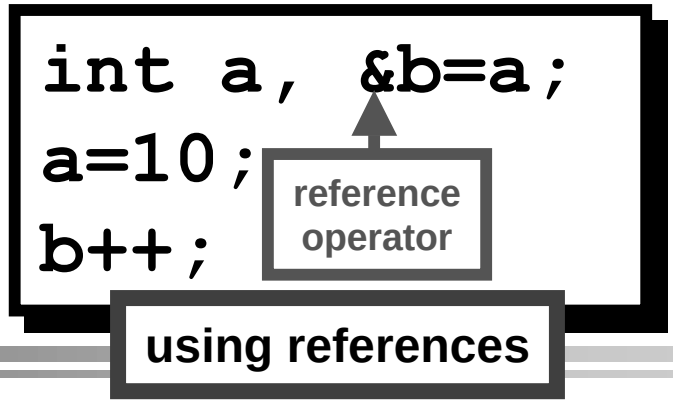
Reference variables

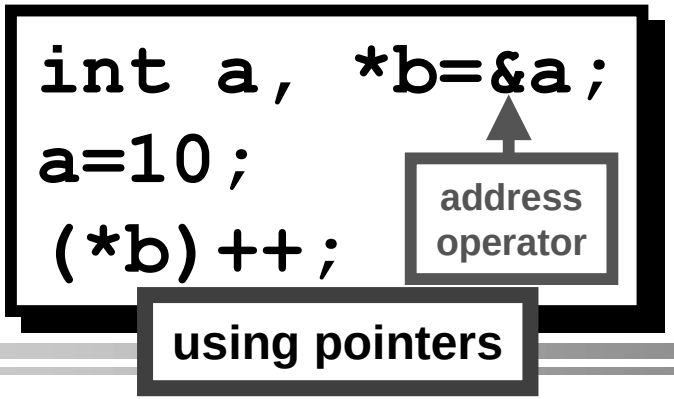
```
#include <iostream.h>
void main()
{
  int i=1,j=10,&ref=i;  //reference variables
                        //must be intialised
  cout << "i=" << i << ", j=" << j << endl;
  ref=j;
  ref--;
  cout << "i=" << i << ", j=" << j << endl;
}
```

```
i=1, j=10
i=9, j=10
```

The reference operator (&)

- The reference operator and the address operator both use the ampersand (&).
  - Q: How can the compiler tell them apart?
  - A: The reference operator is only used in declarations.
- The reference operator is used to indicate that one variable is to be used as an alias for another.
  - The same effect can be achieved using pointers, but only at the expense of a more complex syntax:





Call-by-reference parameters

- As we can only return a single item using the `return` statement we must pass the data via the parameter list.
- To do this we prefix the identifier of a parameter in the function parameter list by the reference operator `&`.
  - This has the effect that the address of the parameter used in the function call is passed to the function.
  - The function parameter is known as a reference to the parameter used in the function call.
  - What ever we do to this reference parameter, will be done to the parameter passed by reference.

Call-by-reference parameters

- To illustrate this idea, consider the previous example function `add2()`.
  - We now declare the parameter `x` to be a reference parameter:

```
void add2(int& x)
{
   x += 2;
}
```

```
int x=3;
cout << "x=" << x << endl;
add2(x);
cout << "x=" << x << endl;
```

- Q: What is output?

```
x=3
x=5
```

Call-by-reference parameters

- Why the difference in behaviour of the function `add2()` ?:
  - Previously the function call `add2(x)` , passed the *value* of `x` to the function parameter (also called `x`).
  - In the second example, the function parameter was declared to be a reference variable, and hence the function call `add2(x)` actually passes the address of the variable `x` to the function, allowing this value to be changed from within the function.
- Reference variables can be thought of as an alias, i.e. we can manipulate a variable by using a variable declared as a reference to it.
  - Anything that we do to the reference variable will in fact be done to the variable to which it is a reference.

Call-by-reference: C vs C++

- The idea of a reference variable is new to C++. As C did not have this capability, to implement call by reference functions we needed to explicitly use pointers:

- **C version:**

```
void add2(int* x)
{
   x += 2;
}
```

```
int x=3;
cout << "x=" << x << endl;
add2(&x);
cout << "x=" << x << endl;
```

- **C++ version:**

```
void add2(int& x)
{
   x += 2;
}
```

```
int x=3;
cout << "x=" << x << endl;
add2(x);
cout << "x=" << x << endl;
```

EXAMPLE: Quadratic root finding

```
bool quadsolve(float a,       // call-by-value
               float b,       // call-by-value
               float c,       // call-by-value
               float& root1, // call-by-reference
               float& root2) // call-by-reference
{
   float disc;                // local variable
   disc = b*b - 4*a*c;
   if (disc < 0.0) return false;
   else {
      root1 = (-b + sqrt(disc))/(2*a);
      root2 = (-b - sqrt(disc))/(2*a);
      return true;
   }
}
```

EXAMPLE: Quadratic root finding

- The function prototype of this function would be as follows:

```
bool quadsolve(float, float, float, float&, float&);
```

- The function may be called as follows:

```
float  c1, c2, c3, r1, r2;
cout << "Please enter coefficients: ";
cin >> c1 >> c2 >> c3;
if ( quadsolve(c1,c2,c3,r1,r2) )
   cout << "Roots are " << r1
        << " and " << r2 << endl;
else
   cout << "Complex roots" << endl;
```

EXAMPLE: Quadratic root finding

- There are a few of things to note about this function:
  - The function uses a return type of `bool` to indicate success. i.e. it returns `true` if the specified quadratic has real roots and `false` if the roots are complex.
  - We do not need any special code other than the ampersand (`&`) in the declaration of the reference parameters `root1` and `root2`. Assigning a value to these parameters is sufficient to pass this information back to the parameters used in the function call.
  - The function call does not discriminate between call-by-value and call-by-reference parameters, i.e. there is nothing in the function *call* that indicates the type of passing used.

EXAMPLE: Quadratic root finding

```
#include <iostream.h>
#include <math.h>
bool quadsolve(float, float, float, float&, float&);
void main()
{
   float  c1, c2, c3, r1, r2;
   cout << "Please enter coefficients: ";
   cin >> c1 >> c2 >> c3;
   if ( quadsolve(c1,c2,c3,r1,r2) )
      cout << "Roots are " << r1 << " and " << r2 << endl;
   else   cout << "Complex roots" << endl;
}
```

```
bool quadsolve(float a, float b, float c, float& root1, float& root2)
{
   float disc;                // local variable
   disc = b*b - 4*a*c;
   if (disc < 0.0) return false;
   else {
      root1 = (-b + sqrt(disc))/(2*a);
      root2 = (-b - sqrt(disc))/(2*a);
      return true;
   }
}
```

A quick word about scope

- The *scope* of a variable is determined by where the variable is declared.
  - A variable declared outside any function (including `main()`) is global and hence is accessible anywhere in the program. Most functions are declared as global, allowing them to be used anywhere.
  - A variable declared within a function is local to that function, and cannot be accessed outside that function. This explains why different functions can have local variables with the same identifier.

A dirty secret about functions

- We have learnt to pass information into a function by value parameters, and return data using the `return` statement and reference parameters.
- In fact, we can write all our functions with no parameters and no returned values !
- To do this, we simply declare all our data to be *global*.
  - Any declaration made outside a function means that this part of the program is accessible from anywhere in the program, i.e. global.
  - To illustrate this, consider the following implementation of the quadratic root finding function.

The `quadsolve()` function using global data

```
#include <iostream.h>
#include <math.h>
void quadsolve();                    // global function declaration
float c1, c2, c3, r1, r2, disc;      // global data declaration
bool  flag;                          // global data declaration
void main()
{
   cout << "Please enter coefficients: ";
   cin >> c1 >> c2 >> c3;
   quadsolve();                      // function call
   if ( flag ) cout << "Roots are " << r1 << " and " << r2 << endl;
   else cout << "Complex roots" << endl;
}
```

```
void quadsolve()
{
   disc = c2*c2 - 4*c1*c3;
   if (disc < 0.0) flag = false;
   else {
      r1 = (-c2 + sqrt(disc))/(2*c1);
      r2 = (-c2 - sqrt(disc))/(2*c1);
      flag = true;
   }
}
```

Q: Well, why don't we do it this way?

- A: We used to write programs this way, but these programs were VERY fragile.
  - By making all data global, it is possible for this data to be corrupted very easily.
  - A programmer working in a different part of the program could inadvertently affect the operation of your part of the program and vice versa.
  - Imagine if each lecture theatre had light switches that controlled all lecture theatres - a confused lecturer could cause havoc to the entire department.
  - By declaring data to be local to a function, we greatly reduce the potential for catastrophic errors.

Functions of type reference

- An interesting type of function that can be used on the lhs of assignments is one whose *return type* is a *reference*:

```
#include <iostream.h>
int array[10];  //global declaration of array
int arrayval(int i) {
   return array[i];
}
int& arrayref(int i) {
   return array[i];
}
void main()
{
   arrayval(7) = 24;                 // WRONG!
   cout << arrayval(7) << endl;      // OK
   arrayref(3) = 19;                 // OK
   cout << arrayref(3) << endl;      // OK
}
```

Functions of type reference

- Q: What is wrong with this function?

```
int& wrong(void)
{
    int i;
 // something
    return i;
}
```

- A: The variable `i` is local to the function `wrong()` - hence `i` only exists during the function call.
  - the caller to this function will receive a pointer to a variable that no longer exists.

Default parameters

- C++ allows you to write functions with optional arguments:
  - Suppose that you modify the function `mysqrt()` to include a parameter specifying the required tolerance.
  - Each time you call this function you would have to specify the required tolerance.

```
double mysqrt(double x, double tol)
{
   double  xold, xnew;
   if (x <= 0.0)  return 0.0;       // in case x=0.0
   else {
      xold = x;                     // first approx
      xnew = 0.5*(xold+x/xold);     // refine approx
      while ( fabs( (xold-xnew)/xnew ) > tol ) {
         xold = xnew;
         xnew = 0.5*(xold+x/xold); // refine approx
      }
      return xnew;                  // square root
   }
}
```

```
cout << mysqrt(17.9,0.00001);
```

Default parameters

- If this option was useful, but rarely required, we could define the parameter `tol` with a default value.
  - This is simply achieved by initialising it with the required value.
  - We can choose to use the default value or specify our own:

```
double mysqrt(double x, double tol=1.0e-5)
{
   double  xold, xnew;
   if (x <= 0.0)  return 0.0;       // in case x=0.0
   else {
      xold = x;                     // first approx
      xnew = 0.5*(xold+x/xold);     // refine approx
      while ( fabs( (xold-xnew)/xnew ) > tol ) {
         xold = xnew;
         xnew = 0.5*(xold+x/xold); // refine approx
      }
      return xnew;                  // square root
   }
}
```

```
cout << mysqrt(17.9) << endl;  // uses default value
cout << mysqrt(0.0003,1.0e-7); // specified tolerance
```

Default parameters

- There are some rules about which parameters are taken:
  - Consider a function that has three parameters, each of which are default, (assume we have a structure `time` with three members):

```
time set(int hour=12, int minute=0, int second=0)
{
   time t={hour,minute,second};
   return t;
}
```

  - A single parameter in the function call will be taken as `hour`. The `minute` and `second` parameters will take the default values.
  - Calling with two parameters, will take the first as `hour`, and the 2nd as `minute`. The parameter `second` will take its default value.
  - The compiler will "fill" the parameter list from left to right.
  - Place any non-default parameters at the left of the parameter list.

Function overloading

- C has strict rules about functions being uniquely identified with an individual name.

- C++ relaxes these rules to allow several functions of the same name - called *function overloading*.
  - The reason behind this will be better explained when we consider the object-oriented philosophy.
  - Basically it is more flexible to allow the programmer to use the same name of function for similar operations.
  - This prevents having to think up a different name for each operation.
  - Imagine if we had to invent a new word to represent the action of pushing, for each object that we wanted to push. e.g. car, door, button, etc.
  - The word "push" can be considered to be overloaded.

Function overloading

- Consider a program that requires functions to display a sum of money, a time, and a date:

```
void display(float amount)
{
   cout << setiosflags(ios::fixed|ios::showpoint);
   cout << setprecision(2) << "£" << amount;
}
```

```
void display(int hour, int minute)
{
   cout << setw(2) << setfill('0') << hour << ":";
   cout << setw(2) << setfill('0') << minute;
}
```

```
void display(int day, int month, int year)
{
   cout << day << "/" << month << "/" << year;
}
```

Function overloading

- **Without function overloading, we would be required to provide individual names for each function.**
  - **`E.g. display_money(),display_time(),display_date().`**

- **The question remains: *"how does the compiler tell which `display()` function we mean?".***

The answer is in the *type* and *number* of the arguments.

```
cout << "The sum of ";
display(withdrawal);
cout << " was removed from your account at ";
display(hour,min);
cout << " on ";
display(d,m,y);
```

Function overloading

- In this example, we have 3 functions called `display()`.
  - The compiler can determine which function to use by comparing the number of arguments in each call to the function declarations.

```
cout << "The sum of ";
display(withdrawal);              // 1 argument
cout << " was removed from your account at ";
display(hour,min);                // 2 arguments
cout << " on ";
display(d,m,y);                   // 3 arguments
```

```
The sum of £7.85 was removed from your account at 17:05 on 2/6/98
```

Function overloading

- Using structures called `time` and `date`, would still be OK as the *types* of the single arguments will differ:

```
void display(float amount)
{
   cout << setiosflags(ios::fixed|ios::showpoint);
   cout << setprecision(2) << "£" << amount;
}
```

```
void display(time t)
{
   cout << setw(2) << setfill('0') << t.hour << ":";
   cout << setw(2) << setfill('0') << t.minute;
}
```

```
void display(date d)
{
   cout << d.day << "/" << d.month << "/" << d.year;
}
```

```
cout << "The sum of ";
display(withdrawal);   // float argument
cout << " was removed from your account at ";
display(t);            // time argument
cout << " on ";
display(day);          // date argument
```

Function overloading

- An important point to note is that the return type is not sufficient to discriminate between overloaded functions:

```
time input()
{
   time t;
   cin >> t.hour >> t.minute;
   return t;
}
```

```
date input()
{
   date d;
   cin >> d.day >> d.month >> d.year;
   return d;
}
```


```
cout << "Enter time: ";
t = input();
cout << " Enter date: ";
day = input();
```

Operator overloading

- **A useful feature of C++ is the ability to *overload* the arithmetic and relational operators to work with user defined data types.**

> **Note that at least one of the operands (arguments) of the operator must be a user defined data type as we cannot change the *number* of operands, so we must change the *type*.**

- **In C++ operators can be treated as functions with a special naming convention:**
  - **A function to overload an operator is given the name `operator` followed by the operator's symbol, e.g:**

```
operator+()  // addition operator
operator>=() // greater than or equal to operator
```

Operator overloading

- Consider a structure called rational that represents rational numbers:

```
struct rational {
   int num, den;
};
```

  - We could overload the multiplication operator as follows:

```
rational operator*(rational a, rational b)
{
   rational product;
   product.num = a.num * b.num;
   product.den = a.den * b.den;
   return product;
}
```

Operator overloading

- We could then use this operator in our code as follows:

```
void main()
{
   rational a={7,22}, b={5,9}, c;
   c = a*b;
   // rest of code
}
```

  - Operator overloading increases our idea of abstraction, as we can use mathematical notation for our own data types.
  - As well as arithmetic operators, we can overload compound assignment, increment/decrement, relational, type conversion, subscript and stream input/output operators.
  - We will consider operator overloading further later in this module when we discuss classes.

Inline functions

- Another useful feature introduced by C++ is the inline function.

- Although functions were a keystone of the methodology behind C, sometimes programmers are reluctant to use them:
  - The reason for this is that there is a performance penalty in using a function.
  - When a function is called, the computer must store where it is in the main program and the values of the local variables, allocate space for the function's local variables and pass parameters to it.
  - As a result, if a piece of code is used repeatedly in a program then placing this code in a function will slow the program down.
  - Hence the irony - the bits of the program that are most obvious function candidates will cause the greatest decrease in speed.

Inline functions

- C++ overcomes this problem by allowing the programmer to declare such functions as being `inline`.
  - This is achieved by placing the `inline` keyword before the function return type in the declaration and prototype:

```
inline void display(date);
```

  - This will cause the compiler to insert the function code at each point in the program where the function is called.

- This will have two effects:
  - Firstly, the program will increase in execution speed.
  - Secondly, the size of the executable code will increase.

- As a result most programmers restrict the use of `inline` status to relatively small functions.

Summary

- The top-down approach to program design splits the initial problem into several sub-problems which in turn can be further sub-divided. Once a problem is simple enough to solve, it can be implemented as a function.

- A function should be self-contained. That is it should communicate with the calling program via supplied input parameters and output parameters. The user of a function should not have to know any details of how the function is implemented.

- Functions encourage the re-use of code and can encapsulate techniques of which the user has no understanding.

Summary

- A C++ function can return a value. The function must be declared to have the same type as this value. If the function does not return a value it must be given the type `void`.

- Information is passed into a function via the parameter list. Each parameter must be given a type and identifier. Unless otherwise indicated, parameters are call-by-value.

- If a parameter is a value parameter, then the function operates on a copy of the value of the actual parameter. Hence the value of the actual parameter cannot be changed by the function.

Summary

- A function prototype provides information to the compiler about the return type of a function and the number and type of parameters. The function prototype must appear in the program before the function is used.

- Any variable declared inside a function is local to that function, and has existence and meaning only inside the function. Hence it can use an identifier already used in the main program or in any other function without confusion with that identifier.

- Information is passed back from a function via reference parameters. A parameter is declared to be a reference parameter by appending `&` to its type.

Summary

- **Default parameters allow you to provide *optional* parameters for functions which are not always required. These optional parameters should be placed at the end of the parameter list.**

- **Functions may be overloaded, by declaring two or more functions to have the same name. This allows similar operations on different data to have the same name.**

- **Overloaded functions can be distinguished on the number and/or type of arguments but not on their return type alone.**

Summary

- C++ operators can also be overloaded, which allows your user defined data types to enjoy the same operations as the supplied data types.

- Overloaded operators must have the same number of operands and hence must have at least one operand of a user defined data type.

- Inline functions can be used to increase the speed of program execution at the expense of executable code size.