Introduction to C/C++

This document is a continuation of the tutorials “IntrocutionTo.NET” and “IntroductionToVisualStudio6”.

2. Introduction to C:

In this section we introduce some basic data types and programming constructs defined in the *ANSI C* standard. We also mention some frequently used *ANSI C* library functions.

2.1 Structure of a C program

All C programs must contain one function called main. The `main` function is the first function called when the program executes and may in turn call other functions. C makes use of only 32 keywords, which combine with the formal syntax to form the C programming language. The keywords are

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

In this tutorial we will require only a subset of these keywords. Note that all keywords are written in lower letters. Like UNIX, C is case-sensitive, ie. lower and upper letters mean different things. If you are not sure what to use its a good idea to always use lower letters.

The overall layout of a C program is as follows:

```
preprocessor directives
global declarations
main()
{
   local variables to function main ;
   statements associated with function main ;
}
other function definitions
```

Let’s consider the example from section 1:

```
// My first C-program

#include <stdio.h>

int main( int argc, char* argv[] )
{
   printf( "Hello World!\n" );
   return 0;
}
```

The first line is just a comments. The second line is a preprocessor directive which includes the header file `stdio.h`. The result of the `#include` statement is to insert the content of the entire file `stdio.h`. The file `stdio.h` contains constants and function declarations required for input and output. The actual definitions of the functions are contained in another file which is automatically included when compiling your program.

The next five lines represent the main function of our program. Note that the main function always has the return type `int` and that it always requires two arguments `argc`, and `argv`, which contain the command line arguments used when executing the program. The number of command line arguments is stored in `argc` and the arguments themselves are stored as an array of strings called `argv`. More explanations on arrays follow in later sections. In this lecture we won't be using command line arguments, ie. `argc` will always be zero.

As in Java the function body is enclosed in brackets ({ and }). Inside the body we have two statements. The `print` statement writes its argument to the standard output (in our case the console window) and the `return` statement returns the result of the function (in the main function we can return any integer value since the result is not used). Note that all C statements are terminated with a semicolon.

2.1.1 Preprocessor Directives

Preprocessor directives are instructions which are processed before compilation. The most common ones are:

1. **Include statements**, which during preprocessing are replaced with the contents of the corresponding header file. As a result all constants and function prototypes defined in that header file are known. Example:

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

2. **Symbolic constants**. During preprocessing all occurrences of the symbolic constant are replaced with the associated numeric value. Example:

   ```
   #define PI 3.14
   ```
   During preprocessing all occurrences of `PI` in the file are replaced by `3.14`.
   NOTE: Constant variables are defined by using the keyword `const`, e.g.

   ```
   const double PI = 3.14;
   ```

2.2 Data Types

2.2.1 Basic Types

Similar to Java, C has got a couple of predefined basic types. The following types are the most important ones:

type	#bytes	range of values	place holder
`int`	4	-2147483648 to 2147483647	%d
`char`	1	-128 to 127	%c
`float`	4	$\approx -3.4*10^{38}$ to $3.4*10^{38}$	%f, %e

`double`	8	≈$-1.7*10^{308}$ to $1.7*10^{308}$	%lf, %e
`short`	2	-32768 to 32767	%d

Note that the number of bytes for each type, and hence its range of values, depends on the *C* implementation used. The values given above are typical and apply to our lab machines. The last column titled 'place holder' refers to the expression used when outputting (or inputting) a value using `printf` or similar functions (see section 2.7).
Note that a float and double constant can be specified either in decimal or exponential notation. Float constant must be identified by adding `f` at the end, e.g.

```
float a = 345000.0f;
float b = 3.15e5f;              // b = 315000.0f
double c = 1.3e-4;              // c = 0.00013
double d = 345000.0;
```

2.2.2 Type conversion

Type conversion occurs in two cases:

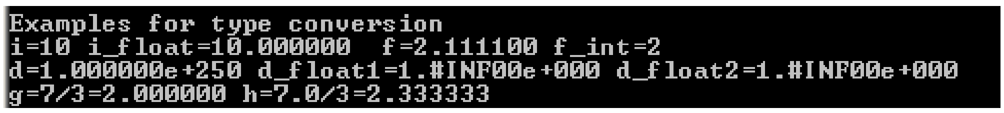
1. if a variable of constant of a certain type (source type) is assigned to a variable of another type (target type) or
2. if two variables or constants of different types (source types) are used as arguments of an arithmetic or logical operation. The resulting (target) type is the type of the argument highest in the following order: `char, short → int → float → double`. For example, when adding a `short` and a `float` number than the result is a `float`.

In order to avoid confusion it is recommended to always perform explicit type conversion. This is achieved by putting the target type in brackets before the value you want to convert.

Example: The program fragment

```
// type conversion
int i = 10;
float f = 2.1111f;
printf( "\nExamples for type conversion" );
float i_float = i;      // implicit conversion
int f_int = (int) f;    // explicit conversion, decimal digits lost
printf( "\ni=%d i_float=%f \tf=%f f_int=%d\n", i, i_float, f, f_int );

double d = 1.0e250;
float d_float1 = (float) d;   // Number to big for a float variable!
float d_float2 = 1.0e250;     // Number to big for a float variable!
printf( "d=%e d_float1=%e d_float2=%e \n", d, d_float1, d_float2 );

float g = 7 / 3;        // integer division then conversion to float
float h = 7.0f / 3;     // convert 3 to float then float division
printf( "g=7/3=%f h=7.0/3=%f\n", g, h );
```

results into the following output



```
Examples for type conversion
i=10 i_float=10.000000  f=2.111100 f_int=2
d=1.000000e+250 d_float1=1.#INF00e+000 d_float2=1.#INF00e+000
g=7/3=2.000000 h=7.0/3=2.333333
```

2.2.3 Enumerations

An *enumeration* is a user-defined integer type consisting of a number of symbolic names representing constant integer values. For exsample,

```
enum Days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};
```

Defining an enumeration is equivalent to defining all symbolic names as integer constants (starting with 0). This means above statement is equivalent to

```
#define MONDAY 0
#define TUESDAY 1 etc.
```

Note that arithmetic and Boolean operations can be performed between two enumeration variables but not between an enumeration variable and an integer. Example:

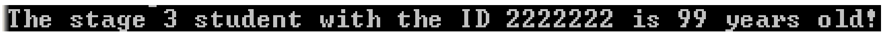
```
Days day = THURSDAY;
day = day + 1;                        // ERROR!
day = (Days) ((int) day + 1);         // CORRECT!
printf("numWeekdays=%d\n", SATURDAY-MONDAY);  // → numWeekdays=5
```

2.2.4 Structures

A structure is similar to a class in Java except that all attributes are public and it has no methods. A structure is created by using the keyword `struct` followed by the name of the structure and in brackets a list of variables (called *components*) separated by semicolons.
If we define a variable having a structure as its type then we can access its components by using the dot operator (analogously to the access of class attributes in Java).

Example:

```
struct Student{
      int id;
      int age;
      int stage;
};

struct Student stud1, stud2;
stud1.id = 2222222;
stud1.age = 99;
stud1.stage = 3;
stud2 = stud1;
printf( "The stage %d student with the ID %d is %d years old!\n",
        stud2.stage, stud2.id, stud2.age );
```

Output: 

```
The stage 3 student with the ID 2222222 is 99 years old!
```

Since it is inconvenient to use the keyword `struct` each time a variable of this structure type is defined, it is recommendable to define the structure as a new type. This is achieved by using the keyword `typedef` followed by the type and its new name. The structure name can be omitted in this definition.

```
typedef struct Student{
     int id;
     int age;
     int stage;
} StudentType;

StudentType stud3;
```

or better

```
typedef struct {
     int id;
     int age;
     int stage;
} StudentType;

StudentType stud3;
```

2.3 Operators

Operators in C are the same as in Java:

Arithmetic operations: +, - , * , / , % , ++ , -- , += , -= , *= , /= , %=

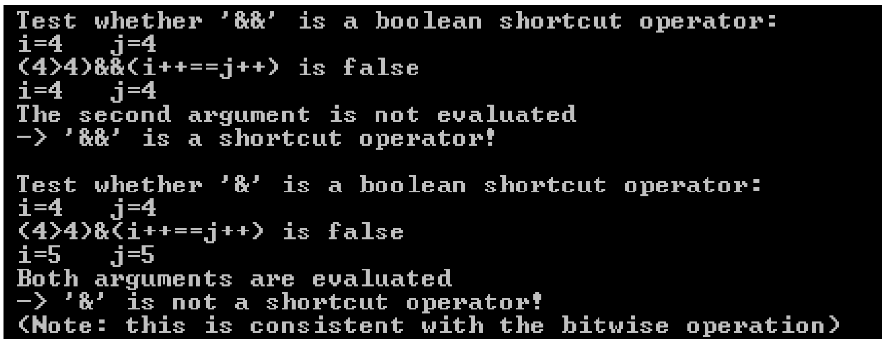
Relational and logical operations: <, > , <= , >= , == , != , &&, || , & , | , !

Note 1: *C* does not have a boolean data type. Instead an integer of zero is interpreted as false and any other value as true. ANSI *C++* standard does have a type `bool`!)

Note 2: The operators '&&' and '||' are shortcut operators. This means, if the result of the expression is determined by evaluating the first argument than the second argument is not evaluated. The operators '&' and '|' are bitwise operators (see below) and therefore always evaluate both arguments. Since booleans are represented by integers such a bitwise operation gives the same result as the logical operation.

Bitwise operations: & , | , ^ , ~ , >> , <<

Example:



```
Test whether '&&' is a boolean shortcut operator:
i=4   j=4
(4>4)&&(i++==j++) is false
i=4   j=4
The second argument is not evaluated
-> '&&' is a shortcut operator!

Test whether '&' is a boolean shortcut operator:
i=4   j=4
(4>4)&(i++==j++) is false
i=5   j=5
Both arguments are evaluated
-> '&' is not a shortcut operator!
(Note: this is consistent with the bitwise operation)
```

2.4 Control Structures

The control structures (loops and conditional statements) in *C* have the same syntax as in Java:

- `if (cond) body1 else body2`
-
```
switch (var) {
      case value1: body1
      case value2: body2
      …
      default: bodyN }
```
- `for(init,cond,iteration) body`
- `do body while (cond);`
- `while (cond) body`
- `continue;`
- `break;`

In the above listing *`cond`* is a boolean expression and *`body`* is either a single statement terminated with a semicolon or a series of statements between curly brackets. *`var`* is a variable and *`value1`*, *`value2`* are values of the same type as the variable.

2.5 Arrays

Arrays in *C* are similar to arrays in Java, but they do not have a length attribute (i.e. the programmer is responsible for keeping track of the array length). Arrays can be allocated either statically (i.e. their length is defined during declaration) or dynamically (only the name of the array is declared and memory is allocated at a later stage). The memory for statically allocated arrays is taken from the stack space during linking. Note that the stack space is usually restricted to 1MByte. If you want to have more memory you have to set a corresponding link option. In contrast dynamically allocated arrays reside in the heap space. Dynamically allocated arrays are just pointers to the memory location of the first array element and are explained in the following section.

A static array is defined either explicitly by specifying its size, e.g.

```
      int a[10];                    // array of 10 integers
```

or implicitly by specifying the array entries, e.g.

```
      int b[] = { 5, 10, 1134 };    // array of 3 integers
```

Note that unlike in Java statically allocated arrays are defined without the new operator and can not be empty, i.e. the definitions

```
      "int c[];" and "int c[0];"
```

cause syntax errors.
An array element can be accessed by using the bracket operator. The first index of an array is always 0, e.g.

```
      printf( "b[0] = %d", b[0] );        // outputs "b[0] = 5"
```

For multidimensional arrays the size must be always specified explicitly, e.g.

```
    int d[10][10];
    int e[2][3] = { {1,2,3}, {4,5,6} }; // [nRows][nColumns]
    printf( "e[0,2] = %d\n", e[0][2] );
```

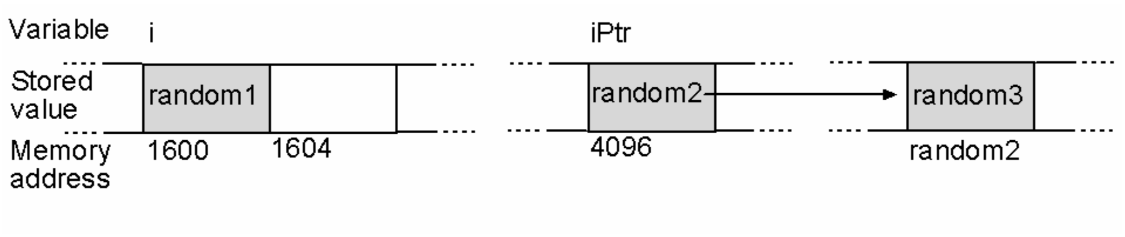
2.6 Pointers

A *pointer* is a variable that contains the address of another variable. Without initialisation the pointer variable contains a random value, which may or may not be a valid address space. Many severe errors are the result of not properly initialised pointer variables:

After executing:

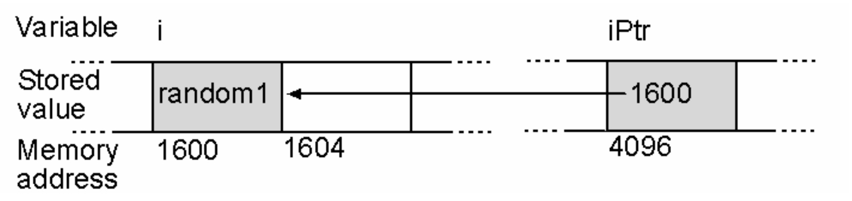
```
    int i;             // integer variable
    int *iPtr;         // pointer variable (to int)
```

the memory might look as follows:



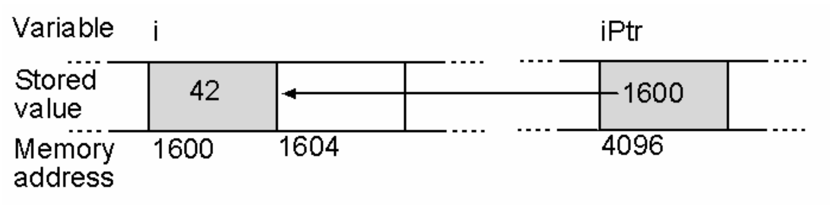
We now set the pointer variable to the address of the variable `i`:

```
    iPtr = &i;                // & is the address operator
```



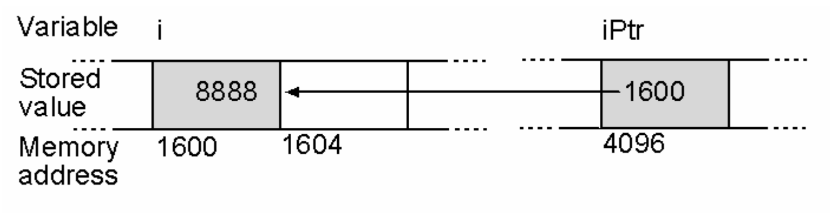
Using the *indirection operator* '`*`' we can access the variable pointed to by the pointer `iPtr`:

```
    *iPtr = 42;                      // indirection operator
```



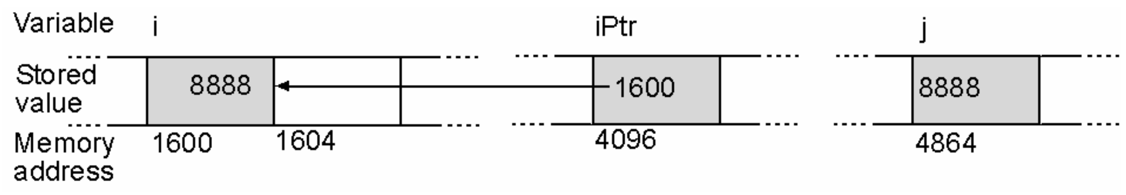
Of course we can still modify the variable i directly. If we access the variable using our pointer we get the modified value:

```
      i = 8888;
      printf( "%d", *iPtr );  // outputs 8888
```



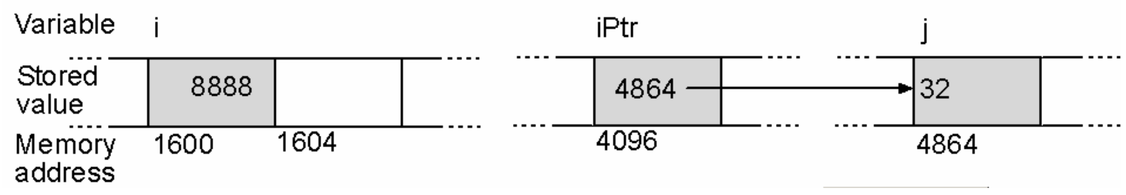
A pointer with an indirection operator can be used exactly like a variable of the corresponding type:

```
      int j = *iPtr;                 // same as "int j = i;"
```



We can also modify the address of the pointer so that it points to a different variable:

```
      j = 32; iPtr = &j;
      printf( "%d", *iPtr );  // outputs 32
```



If a pointer doesn't point to any particular address than the convention is to set it to NULL:

```
int *jPtr = NULL; // a NULL pointer
```

Note that this convention is extremely useful. For example, when implementing a search operation you can return a pointer to an object if it was found and a NULL pointer if no object fulfilled the search criteria. NULL pointers are also used to indicate the end of a linked list or the children of leaf nodes in a binary tree (see 220 lecture). It's a good idea to initialise all pointers to NULL especially if the pointer represents an array (see next section).

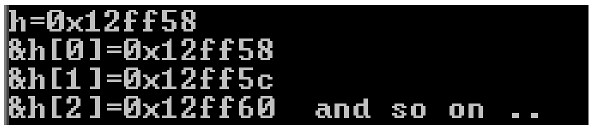
2.6.1 The Relationship between Pointers and Arrays

Arrays are just pointers and hence can be declared as such, e.g.

```
int *a;      // An array of integers (without memory allocated)
```

More precisely, the name of an array is a variable containing the address of the first element of the array. Array elements are stored consecutively in the memory. For example, if we define an array of 10 integers and examine the value of `h` and the addresses of its elements we get the following result:

```
int h[10];
printf( "\nh=%#x", h );          // %#x is used for hexadecimal output
printf( "\n&h[0]=%#x", &h[0] ); // First element has the same address
                                // as the array name
printf( "\n&h[1]=%#x", &h[1] );
printf( "\n&h[2]=%#x  and so on ..\n", &h[2] );
```



```
h=0x12ff58
&h[0]=0x12ff58
&h[1]=0x12ff5c
&h[2]=0x12ff60  and so on ..
```

i.e. the value of h is the address of the first array element. Since an array is just a pointer we can create an array of type *T* and size *n* dynamically by allocating *m*=*n**`sizeof`(*T*) bytes for it. The function `sizeof` returns the size of the type *T* in bytes (e.g. an integer is four bytes long).

The memory is allocated using the command

`Type` `*h=(`*`Type`*`*) malloc(`*`n`*`*sizeof(`*`Type`*`))`

where *`Type`* is the desired data type and *`n`* is the size of the array. If no memory is available a `NULL` pointer is returned.

After using the array the memory must be freed using the function `free`. Example:

```
int numEntries;
scanf( "%d", &numEntries );          // input size of array
int *a=(int*) malloc( numEntries * sizeof(int) ); // allocate
for ( i = 0; i < numEntries; i++ ) a[i] = i; // insert some values
for ( i = 0; i < numEntries; i++ )  // output values
      printf( "a[i]=%d\n", a[i] );
free( a );                           // free memory
```

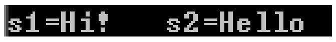
A nicer way to dynamically allocate and free arrays (more similar to Java) is to use the C++ functions `new` and `delete`:

```
int numEntries;
scanf( "%d", &numEntries );          // input size of array
int *a = new int[numEntries]         // allocate memory
for ( i = 0; i < numEntries; i++ ) a[i] = i;    // insert some values
for ( i = 0; i < numEntries; i++ )        // output values
      printf( "a[i]=%d\n", a[i] );
delete[] a;                          // free memory
```

2.6.2 Strings

Strings in *C* are represented as arrays of characters (`char`). The last character is always ‘\0’.

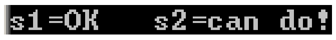
```
char s1[4] = "Hi!";       // s1=’H’’i’’!’’\0’
char *s2 = "Hello";       // s2=’H’’e’’l’’l’’o’’\0’
```



```
s1=Hi!   s2=Hello
```

We can change the content of a string by changing its characters. However, we have to be careful that the new string is again terminated with ‘\0’ and that it fits into the allocated memory. If the string has been declared as a pointer than we can assign a new string constant (e.g. “bla”) to it. In this case the value of the pointer is the address of the assigned string constant.

```
s1 = "can't do this";    // Syntax error, since s1 has been allocated
                         // statically using s1[4] and therefore has a
                         // constant address.
s1[0] = 'O'; s1[1] = 'K'; s1[2] = '\0';// ok, since only content of the
                                       // allocated memory is changed.
s2 = "can do!";          // address of “can do!” goes into s2
                         // possible since s2 is a pointer to a
                         // character (i.e. allocated dynamically)
printf( "s1=%s   s2=%s\n", s1, s2 );
```



```
s1=OK    s2=can do!
```

2.7 Functions

Functions in *C* have a similar syntax to methods in Java:

```
      <returnType> FunctionName(<argType> argName, ..)
      {
            // function body
      }
```

i.e. a function has a return type, a name and a list of arguments with associated types. Functions are only visible in the file where they are defined and they can only be used after they have been defined. An example is given below.

```
int add( int a, int b ){ return a + b; }

int main( int argc, char* argv[] )
{
      int i = 5, j = 7;
      printf( "%d+%d=%d\n", i, j, add( i, j ) );
      return 0;
}
```

Note, that if you define the function `add` after the main function, i.e.

```
int main( int argc, char* argv[] )
{
      int i = 5, j = 7;
      printf( "%d+%d=%d\n", i, j, add( i, j ) );
      return 0;
}

int add( int a, int b ){ return a + b; }
```

then your compiler will generate the error messages:

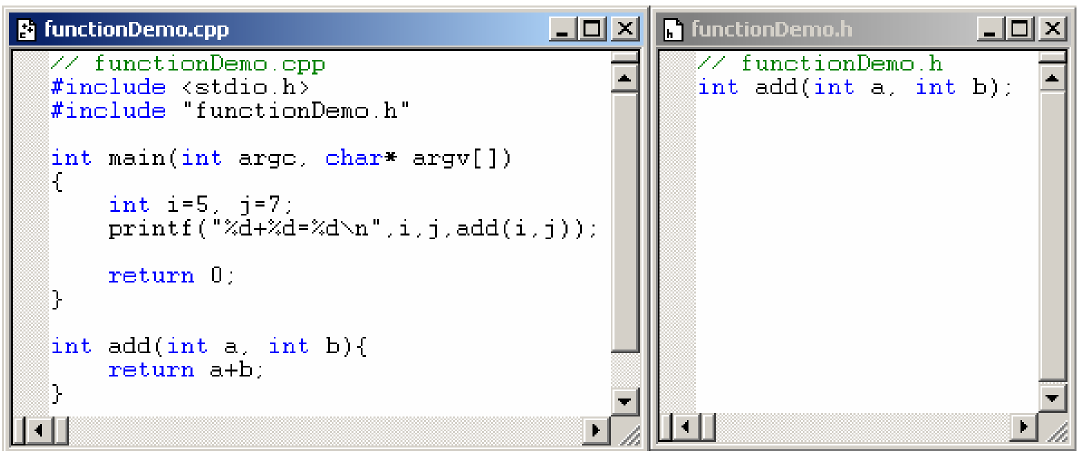
```
error C2065: 'minus' : undeclared identifier
error C2373: 'minus' : redefinition; different type modifiers
```

The first error message is generated when encountering the word `add` in the `printf` function, which at this point hasn't been declared yet. By default the compiler interprets the unknown identifier `add` as an integer. Hence, when encountering the definition of the function `add` the compiler complaints that there is a redefinition of `add` with a different type (i.e. a function type rather than an integer). Note that it is quite common in C that a simple error generates multiple error messages.

Since it is very inconvenient to define functions in the order they are used (and in the case of functions, which call each other, such an order is impossible) the language C allows the declaration of function prototypes. A function prototype contains the return type, the function name, and the argument types (argument names are not necessary) and is terminated with a semicolon. After declaring the prototype a function can be used even if it is only defined later (possible in a different file). Here is an example:

```
int add( int a, int b );      // Prototype
                              // Could also use "int add( int, int );"

int main( int argc, char* argv[] )
{
      int i = 5, j = 7;
      printf( "%d+%d=%d\n", i, j, add( i, j ) );
      return 0;
}

int add( int a, int b ){ return a + b; }  // Definition
```

For large files it's recommendable to include all function prototypes into a header file and to include that header file into the source file. That way you don't have to worry about in which order the functions are used and defined.


functionDemo.cpp

```
// functionDemo.cpp
#include <stdio.h>
#include "functionDemo.h"

int main(int argc, char* argv[])
{
    int i=5, j=7;
    printf("%d+%d=%d\n",i,j,add(i,j));

    return 0;
}

int add(int a, int b){
    return a+b;
}
```

functionDemo.h

```
// functionDemo.h
int add(int a, int b);
```

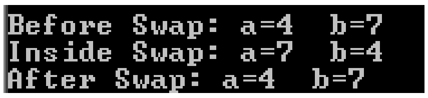
2.7.1 Call-by-value and call-by-reference

If not specified otherwise an argument is given to the called function by '*call by value*'. This means, that the arguments of the function are local copies of the variables in the function call. Consequently changes to these variables are not visible outside the function and the only way to return a value is `return`.

In the example below the variables `a` and `b` are swapped inside the function, but because they are local copies the variables in the function call are not swapped.

```
void badSwap( int a, int b ){
      int temp = a; a = b; b = temp;
      printf( "\nInside Swap: a=%d  b=%d", a, b ); }
int a = 4, b = 7;
printf( "\nBefore Swap: a=%d  b=%d", a, b );
badSwap( a, b );
printf( "\nAfter Swap: a=%d  b=%d\n", a, b );
```

Output:

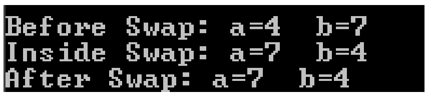


```
Before Swap: a=4  b=7
Inside Swap: a=7  b=4
After Swap: a=4  b=7
```

The variables in a function call can be changed by using '*call by reference*'. Using this calling convention the function's arguments are pointers (references) to variables. Consequently any change to the variable is visible outside the function:

```
void goodSwap( int *aPtr, int *bPtr ){
      int temp; temp = *aPtr; *aPtr = *bPtr; *bPtr = temp;
      printf( "\nInside Swap: a=%d  b=%d", *aPtr, *bPtr ); }
int a = 4, b = 7;
printf( "\nBefore Swap: a=%d  b=%d", a, b );
goodSwap( &a, &b );
printf( "\nAfter Swap: a=%d  b=%d\n", a, b );
```

Output:



```
Before Swap: a=4  b=7
Inside Swap: a=7  b=4
After Swap: a=7  b=4
```

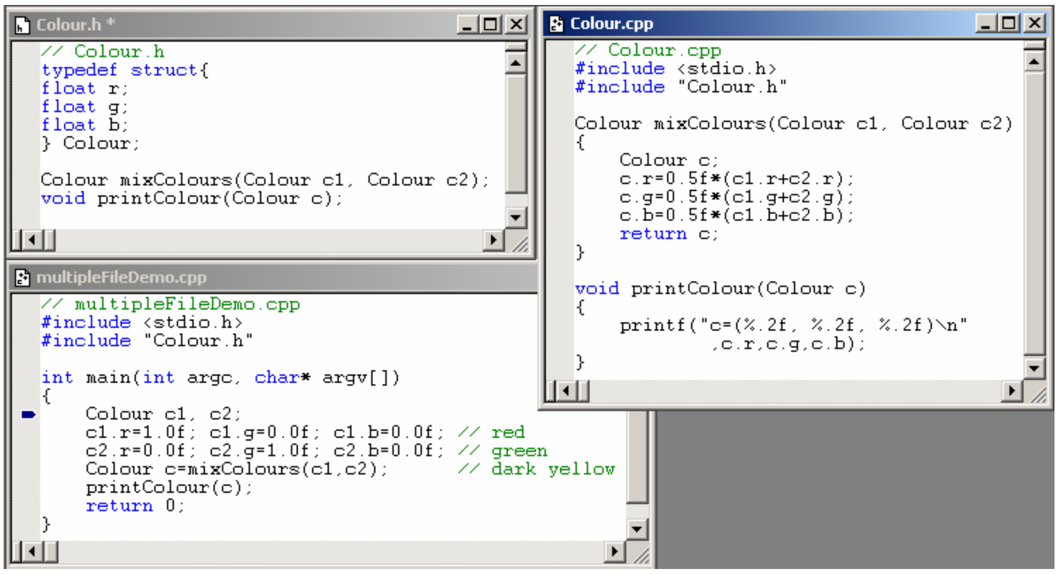
2.7.2 Using multiple files

For large projects it is recommendable to create separate files for groups of related functions and types. For example, when creating a graphics application you might want to have one file containing vector and matrix structures and operations and one file containing functions drawing geometric objects.

The advantages of such an organisation are:

- more readable programs
- improved collaborative software development
- improved reuse
- higher efficiency (only modified files need recompiling)

For each source file we have to create a h*eader file* containing constants, type definitions and function prototypes. The *source file* contains the actual function definitions and 'local' (not visible outside the file) constants and types. Usually the header file is included into the source file, e.g.:



```
// Colour.h
typedef struct{
float r;
float g;
float b;
} Colour;

Colour mixColours(Colour c1, Colour c2);
void printColour(Colour c);
```

```
// Colour.cpp
#include <stdio.h>
#include "Colour.h"

Colour mixColours(Colour c1, Colour c2)
{
    Colour c;
    c.r=0.5f*(c1.r+c2.r);
    c.g=0.5f*(c1.g+c2.g);
    c.b=0.5f*(c1.b+c2.b);
    return c;
}

void printColour(Colour c)
{
    printf("c=(%.2f, %.2f, %.2f)\n"
            ,c.r,c.g,c.b);
}
```

```
// multipleFileDemo.cpp
#include <stdio.h>
#include "Colour.h"

int main(int argc, char* argv[])
{
    Colour c1, c2;
    c1.r=1.0f; c1.g=0.0f; c1.b=0.0f; // red
    c2.r=0.0f; c2.g=1.0f; c2.b=0.0f; // green
    Colour c=mixColours(c1,c2);      // dark yellow
    printColour(c);
    return 0;
}
```

2.8 The ANSI C Standard Library

The ANSI C standard defines a set of functions, as well as related types and macros, to be provided with any implementation of ANSI C. This collection of functions is called 'The ANSI C Standard Library'. The corresponding function prototypes together with the related types are grouped into header files according to the following topics application:

Diagnostics (<assert.h>)	
Character Processing (<ctype.h>)	
Error Codes (<errno.h>)	
ANSI C Limits (<limits.h> and <float.h>)	
Localization (<locale.h>)	
Mathematics (<math.h>)	// essential for Graphics :-)
Nonlocal Jumps (<setjmp.h>)	
Signal Handling (<signal.h>)	
Variable Arguments (<stdarg.h>)	
Common Definitions (<stddef.h>)	
Standard Input/Output (<stdio.h>)	// every program needs I/O
General Utilities (<stdlib.h>)	// random numbers, memory allocation
String Processing (<string.h>)	
Date and Time (<time.h>)	

For our purposes the header files in red should be sufficient. The next three subsections present some of the functions defined in those files. For more information please consult the 'ANSI standard library' documentation http://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_085.HTM#library_chap .

2.8.1 <math.h>

The `<math.h>` header file includes a variety of common mathematical functions. Examples are:

```
double sin(double x);    // sine
double asin(double x);  // inverse of sine
double exp(double x);    // exponential function
double log(double x);    // logarithm with basis e
double log10(double x); // logarithm with basis 10
double pow(double x, double y);     // x^y
double sqrt(double x);  // square root
double ceil(double x);  // rounding up
double fabs(double x);  // absolute value
```

2.8.2 <stdlib.h>

The `<stdlib.h>` header file declares several types, constants and functions of general use. The functions perform string conversion, random number generation, searching and sorting, memory management, and similar tasks. Some examples are listed below:

Constants:

`NULL`	an implementation-defined null pointer constant.
`RAND_MAX`	the maximum value returned by the rand function (is an integer)

String conversion functions:

```
double atof(const char *nptr); // Converts the string pointed to by nptr to double
                               // representation and returns the converted value.
```

Pseudo-random number generation:

```
int rand(void);               // Returns a sequence of pseudo-random integers in
                              // the range 0 to RAND_MAX .
```

Functions for memory allocation:

```
void *malloc(size_t size);    // Allocates a contiguous area in memory for an
                              // object of size size. The area is not initialized.
                              // Returns a pointer to the allocated area, or a null
                              // pointer if unable to allocate.
void free(void *ptr);         // Deallocates the memory area pointed to by ptr.
```

2.8.3 <stdio.h>

The `<stdio.h>` header file several types and constants, and many functions for performing text input and output. A text stream consists of a sequence of lines; each line ends with a new-line character

Types:

FILE	type for a data stream.

Constants:

`NULL`	an implementation-defined null pointer constant.
`EOF`	constant that is returned by several functions to indicate end-of-file (an integer)

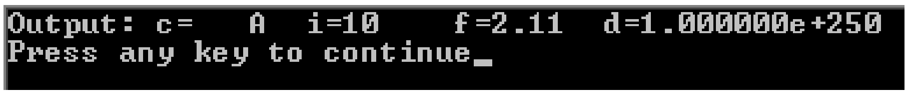
Standard input/output:

```
int printf(const char *format, ...);
```

Writes output to the standard output stream `stdout` under control of the string pointed to by *format*, which specifies how subsequent arguments are converted for output. If there are an insufficient number of arguments for the format, the behavior is undefined. The return value is the number of outputted characters or a negative number if an output error occurred. The format string uses the following special characters to output variables or to format output.

%d - integer	%8d - integer right-aligned in a block of 8 characters
%c - character	%s - string
%f - float	%.3f - float with three decimal digits
%lf – double	%.3lf - double with three decimal digits
%x - integer in hexadecimal notation	
%e – float or double in exponential notation	
'\n' new line	'\t' – tab

```
char c = 'A';              // same as c=65
int i = 10;
float f = 2.1111f;
double d = 1.0e250;
printf( "Output: c=%4c \ti=%d \tf=%.2f \td=%e\n", c, i, f, d );
```

Output: 

```
Output: c=   A  i=10    f=2.11  d=1.000000e+250
Press any key to continue_
```

```
int scanf(const char *format, ...);
```

Reads from the standard input stream `stdin`. The same rules as for `sscanf` apply, which is explained in the next paragraph.

String scanning:

```
int sscanf(const char *s, const char *format, ...);
```

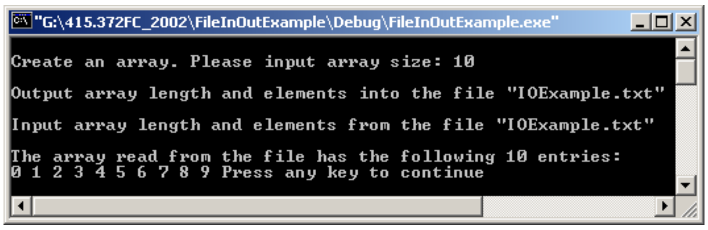
Reads input from string *s*, under control of the string pointed to by *format*, which specifies the allowable input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Note that the address of the input variable is given as argument.

```
int n; char* s[5];
sscanf( "15", "%d", &n );            // ⇒ n = 15
sscanf( "Hi 15", "Hi %d", &n );      // ⇒ n = 15
sscanf( "Hi 15", "Hid%d", &n );      // ⇒ n = 0 (can't match argument)
sscanf( "Hi 15", "%s %d", &s, &n ); // ⇒ s = "Hi", n = 15
```

File Output/Input Example:

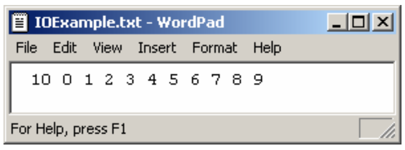
`int fprintf(FILE *stream, const char *format, ...);` // same as `printf`
`int fscanf(FILE *stream, const char *format, ...);` // same as `scanf`
`int fclose(FILE *stream);` // flushes *stream* and closes the associated file.
`FILE *fopen(const char *filename, const char *mode);`
// Opens the file pointed to by *filename* and associates a stream with it. The argument *mode* points is
// a string specifying the fiel access mode: “r” reading, “w” writing, “r+w” read and write.

```
int numEntries, i;
// read size of array
printf( "\nCreate an array. Please input array size: " );
scanf( "%d", &numEntries );                // read from standard input
// create array and fill with consecutive integers
int *f = (int*) malloc( numEntries * sizeof(int) ); // allocate array
for ( i = 0; i < numEntries; i++ ) f[i] = i;
// output array into the file IOExample.txt
char *filename = "IOExample.txt";
printf( "\nOutput array length and elements into the file
        %s\"\n", filename );
FILE *ostream = fopen( fileName, "w" ); // open file for writing
if ( ostream == NULL )
      fprintf( stderr, "Error opening the file %s", filename );
else {
      fprintf( ostream, "%d ", numEntries );// Write array to a file
      for ( i = 0; i < numEntries; i++ )
            fprintf( ostream, "%d ", f[i] );
}
fclose( ostream );

// Read array from file
printf( "\nInput array length and elements from the file
        \"%s\"\n", filename );
FILE *istream = fopen( fileName, "r" ); // open file for reading
if ( ostream == NULL )
      fprintf( stderr, "Error opening the file %s", filename );
else {
      fscanf( istream, "%d", &numEntries );
      f = (int*) malloc( numEntries * sizeof(int) ); // allocate array
      for ( i = 0; i < numEntries; i++ )
            fscanf( istream, "%d", &f[i] );
}
fclose( istream );

// Output the array read
printf( "\nThe array read from the file has the following %d
        entries:\n",numEntries );
for ( i = 0; i < numEntries; i++ ) printf( "%d ", f[i] );
```


"G:\415.372FC_2002\FileInOutExample\Debug\FileInOutExample.exe"

```
Create an array. Please input array size: 10
Output array length and elements into the file "IOExample.txt"
Input array length and elements from the file "IOExample.txt"
The array read from the file has the following 10 entries:
0 1 2 3 4 5 6 7 8 9 Press any key to continue
```

Output of the program fragment above



```
10 0 1 2 3 4 5 6 7 8 9
```

The resulting output file

2.9 Frequent Mistakes

Be careful not to make the following common mistakes:

- **Using '=' instead of '=='**

  For example, the code fragment

  ```
  int a;
  if ( a = 5 ) printf( "a equals 5 ?" );
  ```

  is syntactically valid. However, the expression 'a=5' is always true (think about why!) so that this code is most likely not what you want! To help the compiler detect when you make such mistakes you can write the constant on the left hand side of the equals sign. In that case if you do make a mistake like this:

  ```
  if ( 5 = a ) printf( "a equals 5 ?" );
  ```

  The compiler will complain and you can fix the error immediately.

- **Not initialising variables**

  In Java all variables are initialised to zero. Hence it is possible to write "for (int i; i < 10; i++)". In C the an integer variable contains a random value after declaration so that the result of above program fragment is unpredictable.

- **Using pointers after freeing them**

  After freeing a pointer the memory space formerly associated with it may be overwritten by other parts of the program. Hence using the pointer leads to unpredictable results.

- **Returning a local variable by reference**

  Local variables exist only within the function scope. A reference to a local variable is not valid outside the function because the associated memory space is freed when returning from the function call. However, it is possible to create objects and arrays with 'new' and to use them outside the function (note, however, that both of these cases involve pointer variables).

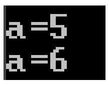
3. Introduction to C++:

This section presents a minimal subset of C++ sufficient for basic graphics programming. Note that OpenGL itself is not object oriented and in general we utilize the OO capabilities of C++ only to create supporting classes such as arrays, vectors, and drawable objects.

3.1 Language Extensions

The language *C* implements 'call-by-reference' passing of arguments by using pointers. *C++* offers additionally so-called *reference parameters*, which are denoted by '&' after the type name. A reference parameter looks and is used like a normal 'call-by-value' parameter but represents a reference to the calling variable so that any changes to it are visible outside the function.

```
void myIncrementFunction( int& a ){ a++; }
…
int k = 5;
printf( "\na=%d", k );
myIncrementFunction( k );
printf( "\na=%d", k );
```

Output: 

```
a =5
a =6
```

3.2 C++ Libraries

MS Visual C++ offers additional C++ libraries. In order to avoid compatibility problems we are using only the C++ I/O libraries. Everything defined within the ISO C++ standard is within the namespace "`std`". If you want to use a function or variable from this namespace you must put "`std::`" in front of it. Namespaces allow us to group code and prevent the occurrence of name clashes between various libraries. For a more complete explanation of namespaces please refer to online documentation (the Thinking in C++ online book is a good start).

The header file <iostream> includes functions and types for standard input and output using streams, all of those reside in the `std` namespace. The standard input and standard output streams are named `cin` and `cout`, respectively, and to use them they must be explicitly qualified as `std::cin` or `std::cout`. This gets a bit tedious, so we import all the names in the `std` namespace into the global namespace via the using directive: "`using namespace std`".

In order to perform the input/output of a certain type a corresponding input and output operator '>>' and '<<', respectively, must be defined. The left-hand side of these operators is a stream and the right hand side a variable or a constant of a specified type. Since the return values of the I/O operators are streams chains of input/output variables can be formed.

The I/O operators are implemented for all basic data types. They can be implemented for user-defined classes using operator overloading (see next subsection).

```
#include <iostream>
using namespace std; // using directive
int a = 5;
cout << "\nThe value of a is: " << a;
cout << "\nInput a new value for a: ";
cin >> a;
cout << "\nThe new value of a is: " << a;
```

The header file <fstream> includes functions and types for file input and output. Similar to *C* the first step involves the creation of an instance of the corresponding class `ofstream` or `ifstream`. After performing input or output (using >> or <<) the streams are closed by calling the `close` method. Note that everything in <fstream> resides in the std namespace.

```
#include <fstream>
using namespace std; // using directive
int a = 6;
ofstream outFile( "fstreamDemo.txt", ios::out );
outFile << a;           // output to a file
outFile.close();
ifstream inFile( "fstreamDemo.txt", ios::in );
inFile >> a;            // input from a file
inFile.close();
```

NOTE: Almost all compilers, even those complying with ANSI standard, allow the use of the traditional header files (like **iostream.h**, **stdlib.h**, etc). If you use these header files you don't have to use the 'using' directive. Nevertheless, the ANSI standard has completely redesigned these libraries taking advantage of the templates feature and following the rule to declare all the functions and variables under the namespace **std**. As explained in this subsection the standard has specified new names for these "header" files, basically using the same name for C++ specific files, but without the ending **.h**. For example, **iostream.h** becomes **iostream**. For compatibility reasons it's a good ides to use this standard (though you will probably soon find out that I often use the old standard myself!)

3.3 Classes

Classes are defined similar as in Java. However, no 'public' keyword is required in order to make them visible to another file. Instead we can write a *specification* (*interface*) into a header file and an *implementation* into a source file.

The specification of the class describes what the class does, but not how it is done. It contains the class name, class attributes and the prototypes of all class methods. The implementation of the class contains the implementations of all class methods. The name of the source and header file must be the same, but the file name may be different from the class name. A file may contain an arbitrary number of classes.

Attributes and methods can be public, private or protected. Private methods and arguments are only visible within the class whereas protected methods and arguments are additionally visible in inherited classes.

Classes may have a destructor, which is called before an object is deleted. It is used to free resources such as arrays.

Below is an example for a class definition and its usage in a different file:

The interface of the Colour class:

```
// Colour.h: interface of the CColour class.

#ifndef LECTURE372_CCOLOUR
#define LECTURE372_CCOLOUR
class CColour{
public:
      CColour();
      CColour( float r, float g, float b );
      virtual ~CColour();
      void setColour( float r, float g, float b );
      void print();
      CColour mixColour( CColour c2 );
private:
      float r, g, b;
};
#endif // LECTURE372_CCOLOUR
```

The implementation of the Colour class:

```
// Colour.cpp: implementation of the CColour class
#include “Colour.h”
#include <iostream>
using namespace std;

CColour::CColour(){ r = 0.0f; g = 0.0f; b = 0.0f; }
CColour::CColour( float r, float g, float b ){ setColour( r, g, b ); }
CColour::~CColour(){}
void CColour::setColour( float r, float g, float b ){
      this->r = r; this->g = g; this->b = b;
}
void CColour::print(){
      cout << "(" << r << ", " << g << ", " << b << ")";
}
CColour CColour::mixColour( CColour c2 ){
      return CColour( 0.5f*(r+c2.r), 0.5f*(g+c2.g), 0.5f*(b+c2.b) );
}
```

Using the Colour class in a program:

```
// classDemo.cpp
#include "Colour.h"
#include <iostream>
using namespace std;

int main( int argc, char* argv[] )
{
      CColour c1;                         //default constructor
      c1.setColour( 1.0f, 0.0f, 0.0f );         // red
      CColour c2( 0.0f, 1.0f, 0.0f );           // green
      CColour c = c1.mixColour( c2 );           // dark yellow
      cout << "\nc="; c.print();
      return 0;
}
```

Note that an object is created by calling its constructor. The keyword new is only used when creating a new object for a pointer.

When using a pointer to an object the object's attributes and methods can be accessed using the '->' operator. This is easier and more readable than using '*' and '.' as demonstrated in the following example:

```
CColour *cPtr;
cPtr = new CColour( 0.0f, 0.0f, 0.1f );
(*cPtr).print();  // original notation
cPtr->print();    // simplified notation
```

Rather than writing an interface and an implementation it is also possible to combine these two into a 'private' class which is only visible in the file where it is defined. For example, in the above example we could have defined the class CColour in the file `classDemo.cpp` as follows:

```
class CColour
{
public:
      CColour(){ r = 0.0f; g = 0.0f; b = 0.0f; }
      CColour( float r, float g, float b ){ setColour( r, g, b ); }
      virtual ~CColour(){}
      void setColour( float r, float g, float b ){
            this->r = r; this->g = g; this->b = b; }
      void print(){ cout << "(" << r << ", " << g << ", " << b << ")";}
      CColour mixColour( CColour c2 ){
            return CColour(0.5f*(r+c2.r),0.5f*(g+c2.g),0.5f*(b+c2.b));}
private:
      float r, g, b;
};
```

3.3.1 Inheritance

A subclass inheriting the attributes and methods from a super class is defined by putting a colon and the name of the super class after the class name. Example:

```
class CcolourWithTransparency : public CColour
{
public:
      CColourWithTransparency();
      CColourWithTransparency( float r, float g, float b, float t );
      virtual ~CColourWithTransparency();
      void setColour( float r, float g, float b, float t );
      void print();
      CColourWithTransparency mixColour( CColourWithTransparency c2 );
private:
      float t;
};
```

This course will only use a little bit of inheritance and if necessary, inheritance will be explained in more detail later.

3.3.2 The `friend` keyword

In some instances it is useful to define methods or even complete classes that have access to the private members of another class `T`. This can be achieved by declaring the methods or the class with the keyword friend in the class `T`. Example:

```
// In Colour.h

class CColour{
public:
      …
      friend CColour mixColour( CColour c1, CColour c2 );
      …
};
```

```
//In Colour.cpp

CColour mixColour( CColour c1, CColour c2 ){
    return CColour(0.5f*(c1.r+c2.r),0.5f*(c1.g+c2.g),0.5f*(c1.b+c2.b));
}
```

3.3.3 Operator overloading

Unlike Java it is possible in C++ to overload operators, which helps creating readable code (e.g. define '+' for vectors and matrices). Please take care that an overloaded operator has a meaningful implementation (e.g. what would be the meaning of '+' for two cars?).

Example: Overloading the stream output operator for the Colour class:

```
// In Colour.h
class CColour{
public:
      …
      friend ostream& operator<<( ostream& s, const CColour& c );
      …
};
```

```
//In Colour.cpp
ostream& operator<<( ostream& s, const CColour& c ){
      cout << "(" << c.r << ", " << c.g << ", " << c.b << ")";
      return s;
}
```

```
// Example of using the operator in some other file
CColour c( 0.0f, 0.0f, 1.0f );           // blue
cout << "\nc=" << c;
```