3. Introduction to C/C++

- 3.1 Reference Material
- 3.2 Background
- 3.3 Introduction to ANSI C
- 3.4 The ANSI C Standard Library
- 3.5 Introduction to C++

3.1 Reference Material

C References:

- C Language Reference & ANSI-C Standard library
  - See COMPSCI 372 Resources page
  - man-pages of any UNIX implementation (eg. type ‘man printf’)

C++ References:

- Microsoft Visual C++ Help
- Bruce Eckel - Thinking in C++
  - free online: http://mindview.net/Books/TICPP/ThinkingInCPP2e.html
  - A local copy is on the COMPSCI 372 Resources page

3.2 Background

- **C** is a general-purpose language originally developed for the UNIX operating system.
- Many of the important ideas of **C** stem from the language **BCPL**, developed by Martin Richards, and the language **B**, which was written by Ken Thompson in 1970 at Bell Labs. **BCPL** and **B** are "typeless" languages whereas **C** provides a variety of data types.
- In 1972 Dennis Ritchie at Bell Labs writes **C** and in 1978 the now famous publication “The C Programming language”.
- In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of **C**. The resulting *ANSI standard*, or "ANSI C", was completed late 1988.

Why learn C (in addition to Java)?

- Advantages
  - Both high-level and low-level language.
  - Better control of low-level mechanism (direct access to hardware).
  - Better performance.
- Disadvantages
  - Explicit memory management (programmer must allocate and free memory).
  - Explicit initialisation and error detection.
  - Higher probability and severity of mistakes!!
    - Pointer errors, memory leaks, …

3.3 Introduction to ANSI C

- Example
- The Structure of a *C* program
- Data types
- Operators
- Arrays

Example

```
/* My first C-programm for Compsci 372FC */

#include <stdio.h>                    // include I/O library

int main(int argc, char* argv[])
{
   printf("Hello World!\n");          // output “Hello World” and
                                      // go to new line
   return 0;
}
```

Example (cont'd)

NOTES:

1. Every *C* program has a `main` function which is executed first.
   - Return type is `int` (ANSI C standard)
     - *MS C/C++* allows return type `void` (try it!). However, in order to make a program portable it is recommended to follow the ANSI standard.
   - The arguments `argc` and `argv` are the number and values of the command line arguments used when executing the application.
     - Example: The UNIX command "`diff <file1> <file2>`"
     - In this lecture we will not use command line arguments.
2. All *C* statements are terminated with a semicolon.

Example (cont'd)

3. Two types of comments
   - `// These single line comments extend`
     `// only to the end of the line`
   - `/* This is a multiline comment, which is often used for excluding program segments during debugging */`

4. The *C* language makes use of only 32 key words. In order to obtain higher-order functionalities we use libraries (as in Java).
   - Many basic libraries are specified by the ANSI C standard and are by default part of your system (see *372 Resources* page).
   - If you use a constant or function from such a library you have to make them known to your program by including the *header file* as `<name.h>`.
   - For example, the output function `printf` is part of the library `stdio`.

The structure of a C program

- The overall layout of a C program is as follows:

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

other function definitions
```

The structure of a C program (cont'd)

- Preprocessor directives are instructions which are processed before compilation. The most common ones are:
  - Include statements, e.g

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

    During preprocessing the statement is replaced with the content of the file `stdio.h`. As a result all constants and function prototypes defined in that file are known.
  - Symbolic constants, e.g.

    ```
    #define PI 3.14
    ```

    During preprocessing all occurrence of PI are replaced with the number 3.14.

    NOTE: Constant variables are defined by using the keyword `const`, e.g.

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

Data Types

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

- The number of bytes depends on the *C* implementation used. The given values apply to the lab machines and are typical.
- 'place holder' refers to the expression used when outputting a basic type using the `printf` function (see the following example).
- Variables in *C* must be explicitly initialised.

Data Types (cont'd)

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

- float values are specified by adding an `f` at the end.
- float and double values can be specified either in decimal or in exponential form. For output in exponential notation use `%e`.
- `printf` allows formatted output, e.g.
  - `%4c` prints a character right aligned in a block of 4 characters.
  - `%.2f` prints a float value with 2 decimal places.
  - `\n` starts a new line and `\t` forwards one tab.

Type conversion

Explicit conversion is recommended if the target type has a lower accuracy then the source type.

```
int i=10;
float f=2.1111f;

int f_int=(int) f;       // explicit conversion
float i_float=i;         // implicit conversion
printf("i=%d i_float=%f\tf=%f f_int=%d\n",i,i_float,f,f_int);
float g=7/3;             // integer division then conversion
float h=7.0f/3;          // conversion then float division
printf("g=%f h=%f\n",g,h);
```

```
i=10 i_float=10.000000  f=2.111100 f_int=2
g=2.000000 h=2.333333
```

Enumerations

- Nothing to do with the Enumeration interface in Java!
- Improve readability of code.
- User-defined integer type consisting of a number of symbolic names representing constant integer values.

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

- Is equivalent to defining all constants as integers (starting with 0), i.e. `#define MONDAY 0`
  `#define TUESDAY 1` etc.
- Can perform operations between enumerations but not between an enumeration and an integer:

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

Structures

Like classes, but they have no methods and all attributes are public:

```
struct Colour {          // RGB colour
    float r;             // red component in range (0,1)
    float g;             // green component in range (0,1)
    float b;             // blue component in range (0,1)
};

struct Colour white, myColour;  // type is struct Colour
white.r=1.0; white.g=1.0; white.b=1.0;
myColour=white;
printf("The RGB coordinates of white colour are (%f, %f,
        %f)\n",myColour.r, myColour.g, myColour.b);
```

```
The RGB coordinates of white colour are (1.000000, 1.000000, 1.000000)
```

Structures (cont'd)

- Have to use the keyword `struct` in each variable definition.
- More convenient to define structure as a new type:
  - Use keyword `typedef`, followed by the structure and name of the new type.
  - Structure name may be omitted

```
typedef struct{ // RGB colour
    float r;          // red component in range (0,1)
    float g;          // green component in range (0,1)
    float b;          // blue component in range (0,1)
} ColourType;

int main(){ …
    ColourType myColour;
    … }
```

Operators

- Same as in Java:
  - Arithmetic operations
    - +, - , * , / , % , ++ , --, += , -= , *= , /= , %=
    - Example: `int i=0; i++;`
  - Relational and logical operations
    - <, > , <= , >= , == , !=
    - &&, || , !
    - Note: *C* does not have a boolean data type. Instead an integer of zero is interpreted as false and any other value as true.
    - Example:

```
int isBigger=11>4;          // isBigger=1
if (isBigger) printf("11>4 is true");
```

    - Remark: ANSI/ISO *C*++ standard does have a type `bool`!

Operators (cont'd)

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

```
int myInt=0x00C3, bitMask = 0x000F; //  11000011 (=195)
int lowFourBits = myInt & bitMask;  //& 00001111 (=15)
                                    //= 00000011 (=3)
```

```
#define FLAG1 1   // 0x0001
#define FLAG2 2   // 0x0010
#define FLAG3 4   // 0x0100
…
int state=FLAG1 | FLAG3;      // state=0x0101
if (state & FLAG3) printf("FLAG3 is set");
```

Control Structures

- Same syntax as in Java:
  - `if (cond) body1 else body2`
  - `switch (var) {`
    `case 1: body1 …`
    `default: bodyN }`
  - `for(init,cond,iteration) body`
  - `do body while (cond);`
  - `while (cond) body`
  - `continue;`
  - `break;`

  where *cond* is a boolean expression and *body* is either a single statement terminated with a semicolon or a series of statements between curly brackets.
- Also have a conditional operator `(cond)?rvalue1:rvalue2`

```
int maximum=(a>b)?a:b;
```

Arrays

- Similar as in Java, but
  - arrays don't have a length.
    - You are responsible for keeping track of the array length (with a separate variable)
  - no empty arrays (instead have NULL pointers - do them later).
    "`int c[];`" or "`int c[0];`" generates a syntax error.
  - The size of an array can be allocated explicitly or implicitly.

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

Arrays (cont'd)

- access to array elements by using the bracket operator. The first index of an array is always 0.

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

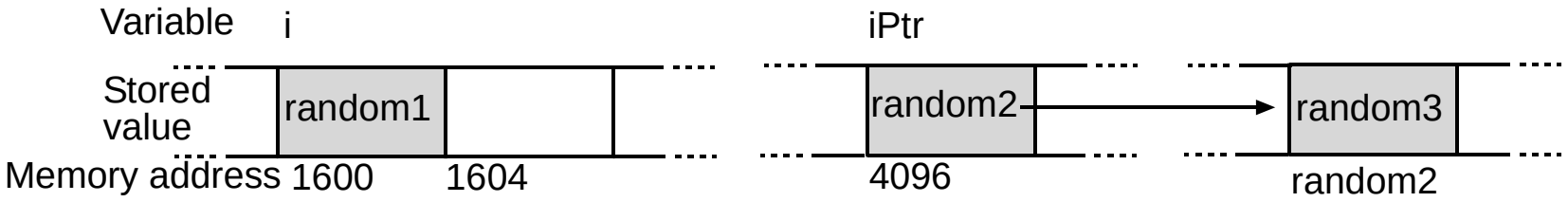
- for multidimensional arrays size must be specified explicitly

```
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]);
```

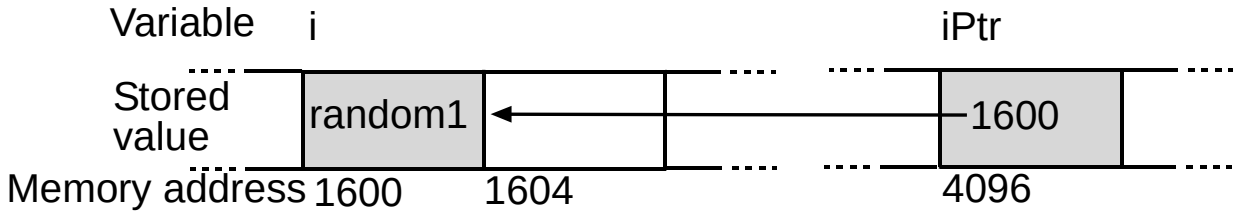
Pointers

- A *pointer* is a variable that contains the address of another variable.

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



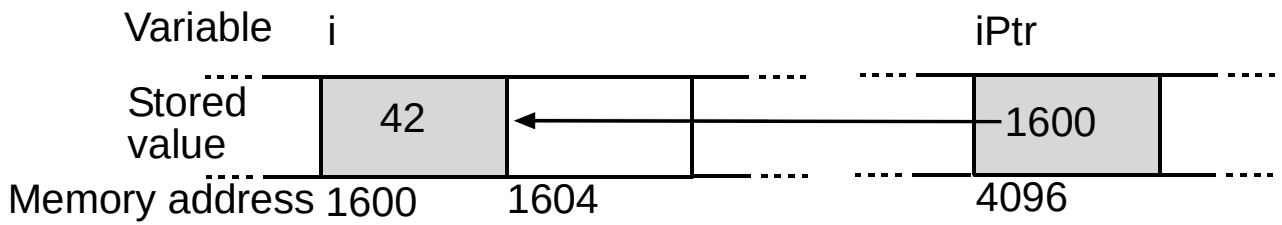

```
iPtr=&i;                // & is the address operator
```



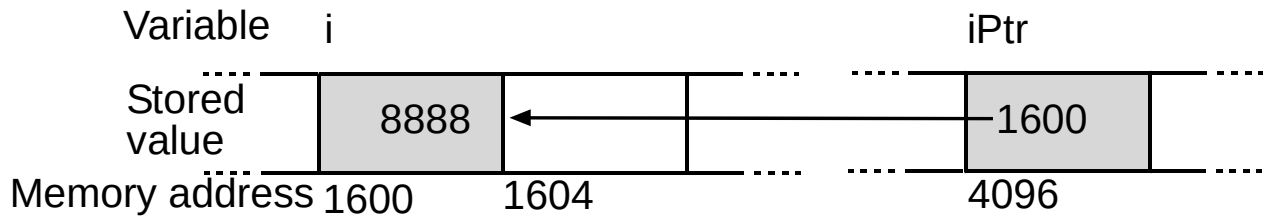
Pointers (cont'd)

- Using the indirection operator we can access the variable pointed to by the pointer.

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



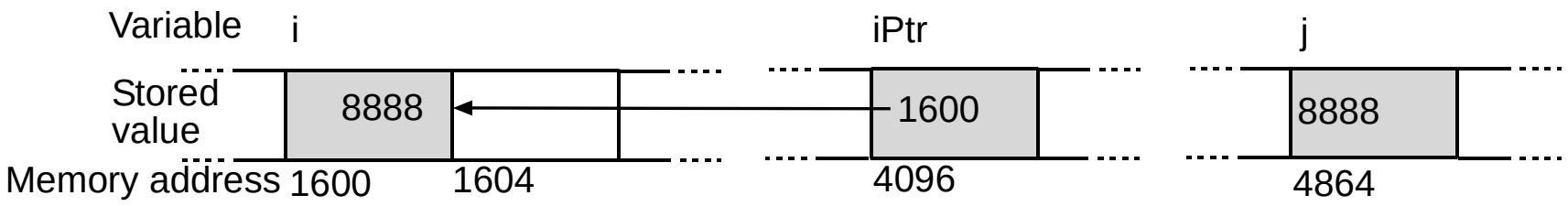

```
i=8888;
printf("%d",*iPtr);    // outputs 8888
```



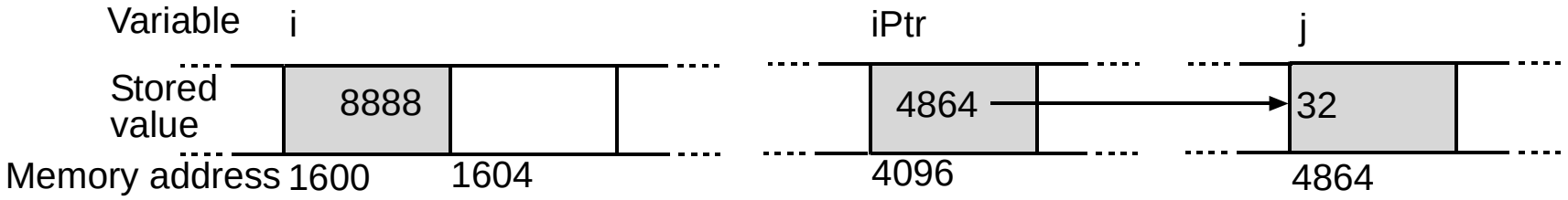
Pointers (cont'd)

- A integer pointer with an indirection operator can be used like an integer variable

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




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



Pointers (cont'd)

If a pointer points to nothing than this is indicated by

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

It'a a good idea to initialise all pointers to NULL especially when creating arrays (see next section).

Pointers and Arrays

- An array name is a variable containing the address of the first element of the array. The address is constant, i.e. it can not be changed.
- Consecutive array elements are stored consecutively in the memory.

```
int h[10];
```

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

An array of type *T* and size *n* can be created 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.

Pointers and Arrays (cont'd)

- After using the array the memory must be freed using `free`:

```
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
```

Exercise: Here is a more efficient (and nastier looking) version of the loop outputting the array. How does it work and why is it more efficient?

```
int *b=a;
while (b!=&a[numEntries]) printf("a[i]=%d\n",*(b++));
```

Pointers and Arrays (cont'd)

- A 'nicer' way to dynamically allocate arrays (and 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
```

- WARNING: The compiler does not check whether an array element is valid or whether a pointer is indeed an array at all:

```
i=5;
int *iPtr=&i;
iPtr[4]=7;      // may cause a crash!
```

Strings

- Strings in *C* are represented by arrays of characters (`char`). The last character is always '\0'.

```
char s1[4]="Hi!";     // s1=['H','i','!','\0']
char *s2="Hello";
printf("s1=%s   s2=%s\n",s1,s2);
```

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

```
s1="can't do this";   // Syntax error, since s1 has a
                      // constant address.
s1[0]='O'; s1[1]='K'; s1[2]='\0';  // ok, since only
                      // allocated memory positions
                      // are changed.
s2="can do!";         // address of "can do!" goes into s2
printf("s1=%s   s2=%s\n",s1,s2);
```

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

Functions

- Similar to methods in Java

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

- Functions are only visible in the file where they are defined. They can only be used after they have been defined.

```
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;
}
```

Functions (cont'd)

- If you we want to use a function before its definition we must define a function prototype first. The function prototype contains the return type, the function name, and the argument types (argument names are not necessary) and is terminated with a semicolon.

```
int add(int a, int b);      // or "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
```

Functions (cont'd)

- The function calls in the previous slides were examples of '*call by value*':
  - The arguments are local copies of the variables in the function call.
  - Changes are not visible outside the function.
  - The only way to return values is by `return`.

```
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);
```

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

Functions (cont'd)

- Variables in a function call can be changed by using '*call by reference*':
  - The arguments are pointers (references) to variables.
  - Obtain pointers by using the address operator in the function call.
  - Changes to the referenced variables are 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);
```

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

Functions (cont'd)

- Exercise: Why does this code not work?

```
void badSwap2(int *aPtr, int *bPtr){
    int *tempPtr;
    tempPtr=aPtr; aPtr=bPtr; bPtr=tempPtr;
    printf("\nInside Swap: a=%d  b=%d",*aPtr,*bPtr); }

printf("\nSwapping pointers:");
printf("\nBefore Swap: a=%d  b=%d",a,b);
badSwap2(&a,&b);
printf("\nAfter Swap: a=%d  b=%d\n",a,b);
```

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

Using multiple files

- Why?
  - Makes program more readable
  - Improves collaborative software development
  - Improves reuse
  - Efficient - only modified files need recompiling
- How?
  - Define *header file* containing constants and types and function prototypes.
  - Define *source file* containing the actual function definitions and ‘private’ constants and types. It’s a good idea to include the header file into the source file.

Using multiple files (cont'd)

Example:

Colour.h *

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

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

Colour.cpp

```
// 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

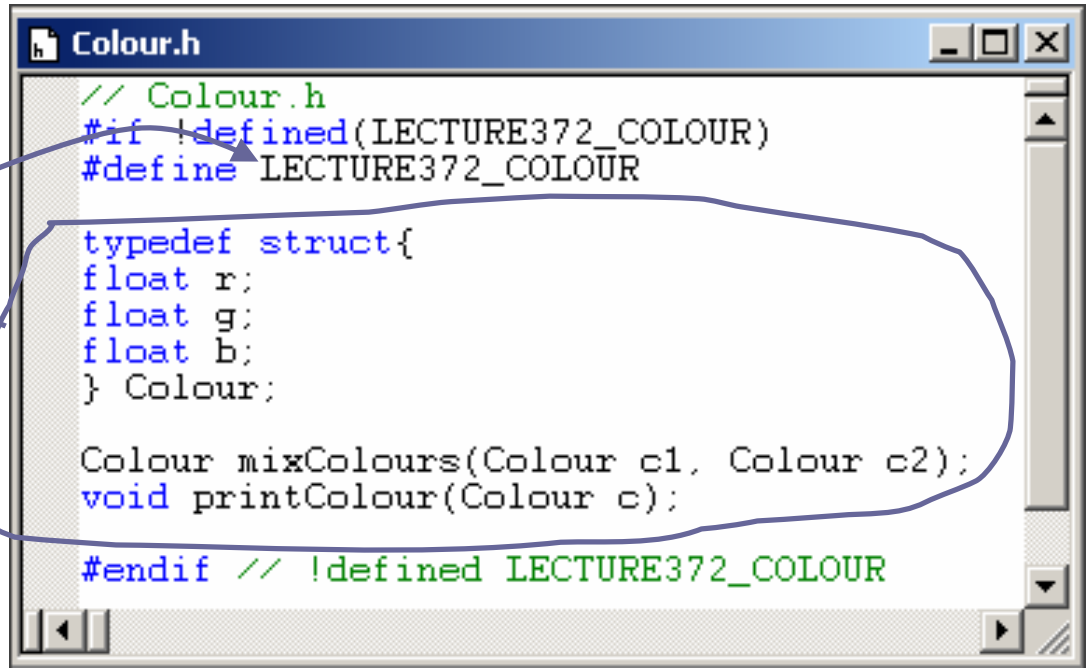
```
// multipleFileDemo.cpp
#include <stdio.h>
#include "Colour.h"

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;
}
```

Using multiple files (cont'd)

- If several files *file_1.h*, …, *file_n.h* including the same file *commonFile.h* are included into another file *bigFile.h* than this file contains multiple versions of the content of *commonFile.h.*

- This can be avoided by defining for each file a unique constant and only including the file content if this constant hasn't been defined yet during compilation.



3.4 The ANSI C Standard Library

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>)

<math.h>

- Defines common mathematical functions. Note that all functions take double arguments and return double-precision values.
- Read documentation for more info.

Examples:

```
double sin(double x);   // sinus
double asin(double x);  // inverse of sinus
double exp(double x);   // exponential function
double log(double x);   // logarithm basis e
double log10(double x); // logarithm 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
```

<stdlib.h>

- String conversion functions, e.g.

```
double atof(const char *nptr);
```

  - Converts the string pointed to by *nptr* to double representation and returns the converted value.

- Pseudo-random number generation, e.g.

```
int rand(void);
```

  - Returns a sequence of pseudo-random integers in the range 0 to RAND_MAX .

- Functions for memory allocation, e.g.

```
void free(void *ptr);
void *malloc(size_t size);
```

<stdio.h>

Standard input/output:

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

- Writes output to the standard output stream `stdout`. The use of the format string has been explained in previous slides. Here's a quick summary
  - %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
  - '\n' new line '\t' - tab

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

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

<stdio.h> (cont'd)

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.

Example:

```
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
```

<stdio.h> (cont'd)

File input/output:

```
FILE *fopen(const char *fname, const char *mode);
```

- Opens the file pointed to by *fname* and associates a stream with it.
- Mode is "r" for reading from a file and "w" for writing into a file.
- If file can not be opened the function returns a NULL pointer.

```
int fclose(FILE *stream);
```

- Closes the file.

```
int fprintf(FILE *stream, const char *format,...);
```

- Writes output to the stream pointed to by *stream.* The same rules as for `printf` apply.

```
int fscanf(FILE *stream, const char *format, ...);
```

- Reads input from the stream pointed to by *stream.* The same rules as for `scanf` apply.

<stdio.h> (cont'd)

File Output/Input Example

```
int numEntries, i;
printf("\nCreate an array. Please input array size: ");
scanf("%d",&numEntries);                          // read from standard input
int *f=(int*) malloc(numEntries*sizeof(int));     // allocate numEntries integers
for(i=0;i<numEntries;i++) f[i]=i;

char *fileName="IOExample.txt";
printf("\nOutput array length and elements into the file \"%s\"\n",fileName);
FILE *ostream=fopen(fileName,"w");
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);
```

<stdio.h> (cont'd)

File Output/Input Example

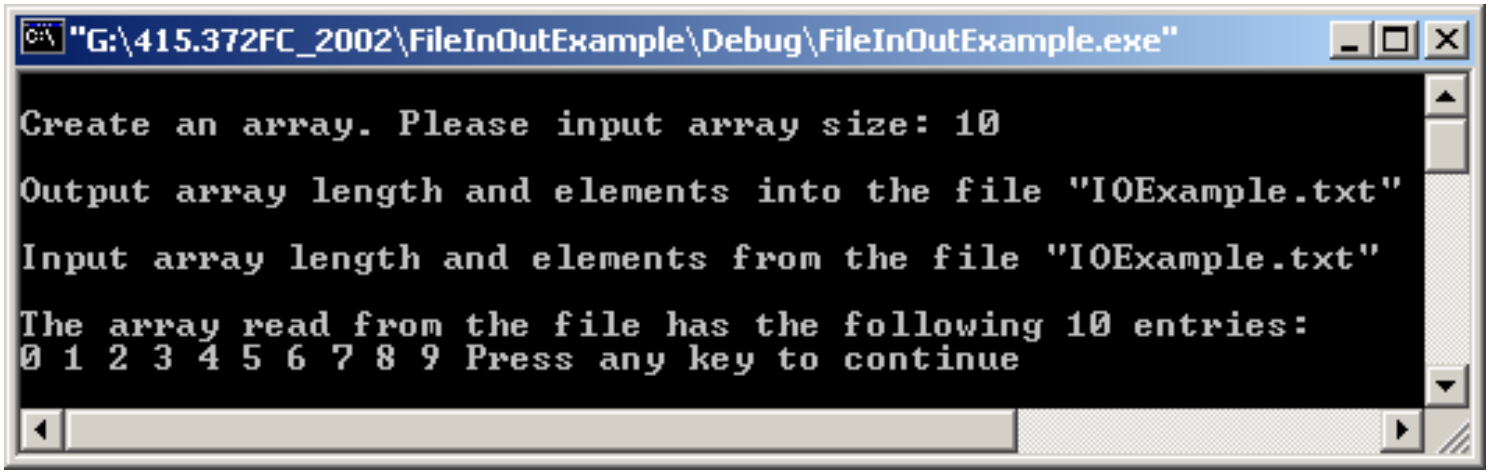
```
printf("\nInput array length and elements from the file \"%s\"\n",fileName);
FILE *istream=fopen(fileName,"r");
if (istream==NULL) fprintf(stderr,"Error opening the file %s",fileName);
else {
        fscanf(istream,"%d",&numEntries);
        f=(int*) malloc(numEntries*sizeof(int));    // allocate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

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

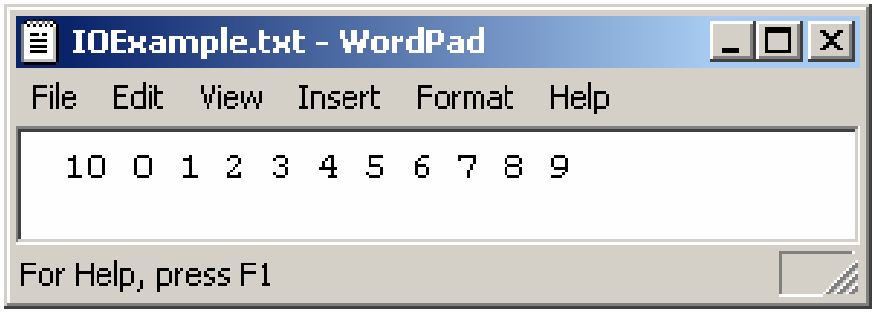
<stdio.h> (cont'd)

File Output Example (cont'd):

Program output:



The input/output file:



3.5 Introduction to C++

- Reference variables
- I/O functions
- Classes
- Inheritance
- Friends
- Operator overloading

Reference variables

- C++ allows parameters to be passed by reference to functions without using pointers.
- A reference parameter is denoted by '&' after the type name.

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

Output:

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

Name spaces

- Namespaces allow us to group code and prevent the occurrence of name clashes between various 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.
  - Can avoid this by using the `using` directive

```
#include <iostream.h>          // old header file
```

```
#include <iostream>                    // ISO C++ standard
std::cout << "\n Hello world";
```

```
#include <iostream>                    // ISO C++ standard
using namespace std;
cout << "\n Hello world";
```

I/O functions

- The input and output streams in *C++* are `cout` and `cin`.
  - Defined in <iostream>.
  - Input and output is performed by using the operators '`<<`' and '`>>`', which are defined for all basic types.
  - The operators can implemented for user-defined classes by using operator overloading (explained later).

```
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;
```

I/O functions (cont'd)

- Functions and classes for file input and output
  - Defined in <fstream>.
  - Similar as in C we have to create a file stream first by creating 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.

```
#include <fstream>
using namespace std;
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();
```

Classes

- Similar to Java:
  - Classes do not have a ‘public’ keyword
  - For each class we (usually) write a *specification* (*interface*) in 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 name may be different from the class name. The files may contain an arbitrary number of classes.
  - Attributes and methods can be public, private or protected.
  - Classes have a “destructor” which is called before an object is deleted. Used to free system resources, e.g. to free memory allocated for an array.

Classes (cont'd)

```
// Colour.h: interface of the CColour class.
#if !defined(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
```

Classes (cont'd)

```
// 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));}
```

Classes (cont'd)

- Using the class:

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

int main(int argc, char* argv[])
{
   CColour c1;                          //default
   constructore
   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;
}
```

Classes (cont'd)

- Can combine implementation and interface of a class:
  - For example, write a 'private' class in classDemo.cpp.

```
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;
};
```

Classes (cont'd)

- Remarks
  - An object is created by calling the constructor (don't use `new`).
  - However, must use `new` when creating a new object for a pointer.
  - When using pointers to objects a more readable notation for accessing attributes and methods is to use the '->' operator:

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

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.

```
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;
};
```

Friends

- 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 method or class with the keyword friend in the class `t`.

```
// 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));}
```

Operator overloading

- Unlike Java it is possible in C++ to overload operators
  - makes the code more readable (e.g. define '+' for vectors and matrices)
  - use only in instances where the operator is meaningful, e.g what would be the meaning of '+' for two cars?

- Example: Defining the output operator for the Colour class:

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

Operator overloading (cont'd)

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

```
// Using the operator
CColour c(0.0f, 0.0f, 1.0f);        // blue
cout << "\nc=" << c;
```