



Practical Issues

- Course registration now works via the webform but you can also use res checkin oopk
- DELFI has obtained the list of students that need a UNIX account.
- You can log into your UNIX account from an external terminal using telnet, for more information see <http://www.sgr.nada.kth.se/unix/docs/inloggnig.html>
- In order to get access to the terminal rooms DELFI can update your existing student card, sign in on the list.
- A module has been created for the course which you can load with the command
module load /info/ooopk01/module/ooopk01
- The slides for the first two lectures and the description of the lab exams are available on the course webpage <http://www.nada.kth.se/kurser/kth/2D1358/00-01/E/>



C/C++ Programming Basics

- basic data types in C/C++
- variables
- type conversions
- standard input/output
- arithmetic, relational, logical operators
- header files and libraries
- loops and decisions
- scope of variables
- functions, call by value, call by reference
- overloaded functions



Hello World

```
#include <iostream.h> // input-output library
int main()           // function main
{
    cout << "Hello World" << endl; // standard output stream
}
```



Comments

- comments are **always** a good thing to use because
 - not everyone is as smart as you are and needs more explanation in order to understand your program
 - you may not be as smart next month when you have to change your program
- comments should clarify your code and explain the rationale behind a group of statements
- comments start with a double slash "//" and terminate at the end of line
 - // this is a comment in C++
- alternative comment syntax (old C style) /* */
 - /* this is an old-style comment which can go across multiple lines of code */

Simple Data-Types

- integer data-types :
char, short, int, long, unsigned char, unsigned short,...
- floating point data-types :
float, double, long double
- logical data-type :
bool
bool constants : true, false
- character data-type :
char
character constants in single quotes : 'a', '\n'
- text data-type :
string
string constants in double quotes : "Hello world"

Data-Types

Type	Low	High	Digits of Precision	Bytes
char	-128	127	-	1
short	-32768	32767	-	2
int	-2147483648	2147483647	-	4
long	-2147483648	2147483647	-	4
float	3.4×10^{-38}	3.4×10^{38}	7	4
double	1.7×10^{-308}	1.7×10^{308}	15	8
long double	3.4×10^{-4932}	3.4×10^{4932}	19	10

Variable Definitions

- A *declaration* introduces a variable's name (such as var1) into a program and specifies its type (such as int)
- A *definition* is a *declaration* which also sets aside memory for that variable (which is usually the case)
- In C++ it is possible to initialize a variable at the same time it is defined, in the same way one assigns a value to an already defined variable.
- Examples of variable definitions: Conventional C

```

int var1;
double x = 5.9;
char answer = 'n';
string str = "Hello world";
double y=x;
double x;
x = 5.9;
char answer;
answer = 'n';
    
```

Constant Variables

- the keyword *const* for constant precedes the data type of a variable and specifies that the value of this variable will not change throughout the program
- an attempt to alter the value of a variable defined as constant results in an error message by the compiler
- example:
`const double PI = 3.14159;`
- constants can also be specified using the preprocessor directive `#define` (old C style constants)
- example:
`#define PI 3.14159`
the preprocessor replaces the identifier PI by the text 3.14159 throughout the program
- the major drawback of `#define` is that the data type of the constant is not specified

String Data Type

- string data-type in C++ different from classical string implementation in C (array of char)
- the operator + can be used for concatenation of multiple strings
- the operator [] can be used to index individual characters

```
string str = "Hello"; // definition with initialization
str = str + " World"; // results in "Hello World";
str += " !"; // results in "Hello World!";
char c = str[2]; // extracts the character 'e' from str
int n = str.length(); // returns the length of a string
```

Type Conversions

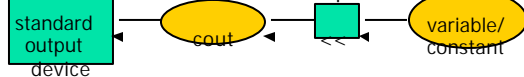
- C++ is a hard-typed language, meaning that each variable has a fixed type that does not change and which determines the possible assignments and applicable operators

```
double pi=3.14; // ok
double x="Hello"; // ooops,...fails
• Some type conversions occur automatically for example int to float or float to double
int i = 17;
float x = i; // assigns 17 to x
int j = 2;
float y = i/j; // assigns 17/2 = 8 to y not 8.5 !!!
• Type conversions can be forced by a programmer through a type cast
float z = static_cast<float>(i)/j; // casts i into float before division
```

Output Using Cout

- The identifier *cout* is actually an object. It is predefined in C++ and corresponds to the *standard output stream*.
- A *stream* is an abstraction that refers to a flow of data.
- The operator << is called the insertion or put operator and can be cascaded (like arithmetic operators +, -, *, /)
- It directs the contents of the variable to the right to the object to the left.

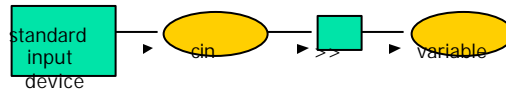
```
string str="Hello world";
int i=8;
cout << str << endl; // endl inserts a new line like \n
cout << "i=" << i << endl; // cascade operator <<
```



Input Using Cin

- The object *cin* is predefined in C++ and corresponds to the *standard input stream*.
- The >> operator is called the *extraction* or *get* operator and takes the value from the stream object to the left and places it in the variable on its right.
- The stream represents data coming from the keyboard

```
int temperature;
cout << "Enter temperature in Celsius: ";
cin >> temperature;
```



Arithmetic Operations

- multiplication, summation, subtraction, division
- ```
int i = 1/3; // integer division result 0
float x = 1.0/3; // floating point division result 0.3333
int j = 7 % 3; // modulo operator remainder of 7/3
```
- prefix and postfix-increment operator ++
- ```
int i=3;
int j=7;
cout << 10 * i++; // outputs 30, i has value 4 afterwards
cout << 10 * ++j; // outputs 80, j has value 8 afterwards
```
- arithmetic assignment operators
- ```
float x=6.0;
x+=3.5;
• is equivalent to
x=x+3.5;
```

## Library Functions

- Many activities in C/C++ are carried out by *library functions*.
- These functions perform file access, data conversion and mathematical computations.

```
#include <math.h> // includes the declaration file for
 // mathematical functions

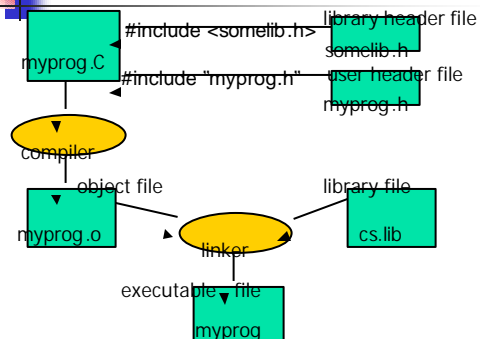
int main()
{
 double x=3.14;
 cout << "sin(3.14)= " << sin(x) << endl;
}
```

## Header Files

- a header file contains the declaration of functions you want to use in your code
- the preprocessor directive #include takes care of incorporating a header file into your source file
- example:

```
#include <math.h>
#include "myprog.h"
```
- the brackets <> indicate that the compiler first searches the standard *include* directory which contains the standard C/C++ header files first
- the quotation marks indicate that the compiler first searches for header files in the local directory
- if you do not include the appropriate header file you get an error message from the compiler

## Header and Library Files



## Relational Operators

- a relational operator compares two values of C++ built in data types such as char, int, float or user defined classes
- typical relationships are equal to, less than and greater than
- the result of a comparison is either true or false, where 0 is false and any value unequal to 0 is true
- C++ provides the following relational operators  
<, >, ==, !=, <=, >=
- example:
 

```
int x=44;
int y=12;
(x == y) // false
(x >= y) // true
(x != y) // true
```

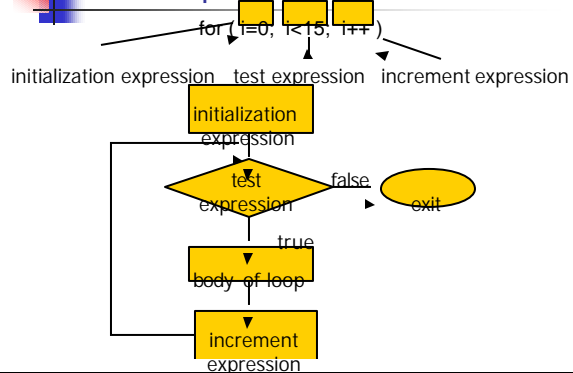
## Loops

- a loop causes a section of the program to be repeated multiple times while the *loop condition* remains true
- C++ knows three kinds of loops
  - for loop
  - while loop
  - do loop
- the for loop repeats a code segment a fixed number of times
- the for statement contains three expressions usually referring to the same *loop variable* separated by semicolons

```
for (i=0; i<15; i++)
```

initialization expression    test expression    increment expression

## For Loop



## For Loop

```
int i;
for (i=1; i<=15; i++) // for loop with a single statement
 cout << i*i << " ";
cout << endl;

for (i=1; i<=15; i++) // for loop with multiple statements
{
 cout << setw(4) << i << " "; // setw(n) sets the width
 // of the next output field to n
 int cube = i*i*i;
 cout << setw(6) << cube << endl;
}
```

## Indendation and Loop Style

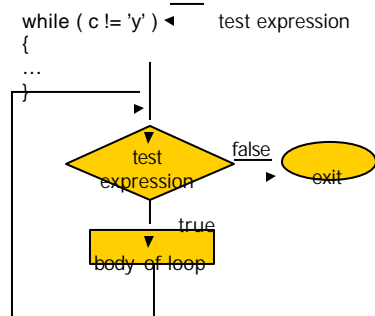
- it is good programming practice to indent loops
  - there is some variation of the style used for loops  
whatever style you use do it consistently
- ```
for (i=0; i<10; i++) //indent body but not the brackets
{
    cout << i*i << endl;
}
for (i=0; i<10; i++) //indent body and brackets
{
    cout << i*i << endl;
}
for (i=0; i<10; i++) { // opening bracket after loop statement
    cout << i*i << endl;
}
```

While Loop

- the while loop is used when the number of iterations is unknown before the loop is started
- the while loop is repeated as long as the test expression remains true

```
char c='n';
while ( c != 'y')
{
    cout << "Do you want to continue: (y/n)" << endl;
    cin >> c;
}
```

While Loop



Do Loop

- in the do loop the test expression is evaluated at the end of the loop, therefore the body is executed at least once

```
char c;
do
{
    cout << "Do you want to continue: (y/n)" << endl;
    cin >> c;
}
while ( c != 'y');
```

Do Loop

```
do
{ ... }
while ( c != 'y' );
```

test expression

```

graph TD
    Body[body of loop] --> Test{test expression}
    Test -- true --> Body
    Test -- false --> Exit([exit])
  
```

If ... Else Statement

- depending on whether the test condition is true or false either the if or the else branch is executed

```
int x;
cout << "Enter a number. ";
cin >> x;
if ( x > 100)
  cout << x << " is greater than 100" << endl;
else
  cout << x << " is not greater than 100" << endl;
```

If ... Else Statement

```
if ( x > 100)
{
...
}
else
{
...
}
```

test expression

```

graph TD
    Test{test expression} -- true --> BodyIf[body of if]
    Test -- false --> BodyElse[body of else]
    BodyIf --> Exit([exit])
    BodyElse --> Exit
  
```

Switch Statement

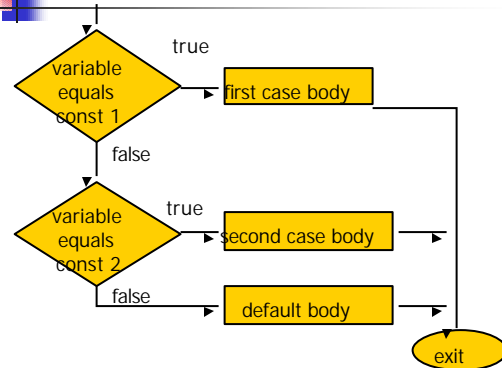
- the switch statement is used if there are more than two alternatives and the decision depends on the value of the same variable
- the switch statement can replace a ladder of multiple nested if..else statements

```
switch(<variable>)
{
  case <constant1>:
    ...
    break;
  case <constant2>:
    ...
    break;
  default :
    ...
}
```

Switch Statement

```
char c;
cout << "Enter your choice (a/b/c) : ";
cin >> c;
switch (c)
{
  case 'a':
    cout << "You picked a!" << endl;
    break;
  case 'b':
    cout << "You picked b!" << endl;
    break;
  case 'c':
    cout << "You picked c!" << endl;
    break;
  default:
    cout << "You picked neither a,b,c!" << endl;
}
```

Switch Statement



Logical Operators

- logical and : &&
(x >= 5) && (x <= 15) // true if x in [5,15]
- logical or : ||
(x == 5) || (x == 10) // true if x=5 or x=10
- logical negation : !
!(x == 5) // true if x is not equal to 5
x != 5 // is equivalent
- conditional operator : ? ... :
<condition> ? <>true expression> : < false expression>
if (alpha < beta)
min = alpha;
else
min = beta;
min = (alpha < beta) ? alpha : beta; // substitutes if ...else

Scope of Variables

- a block is a section of code delimited by a pair of brackets { ... }
- a declaration introduces a variable into a scope (a specific part of program text)
- the scope of a variable is the block within which the variable is declared
- a variable declared outside a function is global
- a declaration of a variable in an inner block can hide a declaration in an enclosing block or a global variable

Scope of Variables

```
int x; // global variable

int main()
{
    int i=3; // local variable
    {
        int j=5; // local variable
        int i=7; // local i hides outer i
        cout << i; // outputs 7
    } // end of scope of j and inner i
    cout << i; // outputs 3
} // end of scope of outer i
```

visibility of outer i
lifetime of outer i

Functions

- a function groups a number of program statements into a unit and gives it a name
- the function can be invoked from other parts of the program
- dividing a program into functions is one way to structure your program (structured programming)
- a function *declaration* specifies the name of the function the type of the value returned and the number and type of arguments that must be supplied in a call of the function
- a function *definition* contains the body of the function
- typically function *declarations* occur in header files (.h) whereas the function *definitions* are stored in source files (.c)

Functions

```
int fac(int n); // function declaration
int x=7;

cout << "fac(" << x << ")=" << fac(x); // call to a function;

int fac(int n) // function definition
{
    int result=1;
    for (int i=1; i<=n; i++)
        result*=i;
    return result;
}
```

Passing by Value

- when passing arguments by value, the function creates new local variables to hold the values of the variable argument
 - the value of the original variable are not changed
- ```
void f(int val) // the parameter val holds a local copy of the
 // variable argument
{
 val++;
}

int x=4;
f(x); // call function f passing x by value
cout << x; // x still has the value 4
```

## Passing by Reference

- a *reference* provides an alias –a different name– for a variable
- when passing arguments by reference the local variable is an alias for the original variable
- the memory address of the variable is passed such that the function can access the actual variable in the calling program

```
void swap (int &a, int& b) // call by reference a,b are aliases
{
 int tmp;
 tmp = a;
 a = b;
 b = tmp;
}
int x=3; int y=5;
swap(x,y); // call by reference : a,b are aliases for x,y
```

## Const Function Arguments

- passing by reference is efficient when passing large data structures as it avoids copying the variable
- a const reference guarantees that the function cannot modify the value of the passed argument
- an attempt to change the value of a const argument is caught by the compiler with an error message

```
void f(int& a, const int& b) // argument b is constant
{
 a=5; // ok
 b=7; // error: cannot modify const argument
}
```

## Overloaded Functions

- an overloaded function performs different activities depending on the number and type of arguments passed

```
void print(int a) // prints an integer
{
 cout << "integer : " << a;
}

void print(string s) // prints a string
{
 cout << "string : " << s;
}
```

## Overloaded Functions

- in C++ an overloaded function can be called with a variable number of arguments

```
void draw_line() { //draws a line with char 'x' of length 20
 for (int i=0; i<20; i++)
 cout << "x";
 cout << endl;
}

void draw_line(char c) { //draws a line with char c of length 20
 for (int i=0; i<20; i++)
 cout << c;
 cout << endl;
}

void draw_line(char c, int n) { // draws a line with char c of length n
 for (int i=0; i<n; i++)
 cout << c;
 cout << endl;
}
```



## Default Arguments

- in C++ functions can be called with a variable number of arguments

```
#include <math.h>
double logn (double x, double base=10) // default base is 10
{
 return log(x)/log(base);
}
```

```
double y=5.6;
cout << "log(y) = " << logn(y) << endl; // uses default base
cout << "ln(y) = " << logn(y,2.71828) << endl; //base e
cout << "ld(y) = " << logn(y,2) << endl; // base 2
```



## Recommended Reading

- Stroustrup : chapters 3.4-3.6, 4, 6 and 7
- Lippman : chapters 3.1-3.7, 4.1-4.7, 5, 7.1-7.4 and 9.1-9.2
- Lafore : chapters 2, 3 and 5