Your first C++ Program

```
#include <iostream>
using namespace std;

void main()
{
  cout << "Hello, world!“ << endl;

}
```

include library <iostream>

use std namespace

output statement

Components of the Program

- “`#include…`” - called *include directives*, tells the compiler to include the header of the library `<iostream>`
- The line “`using namespace std;`” tells the compiler to include the `std` namespace
- The lines “`int main(){` ” and “`return 0; }`” tell the compiler where the main body of program starts and ends
- The line “`cout << …`” is an *executable statement* (or simply *statement*). It causes “`Hello, world!`” to be printed on the screen.

Library

- Library is a collection of classes, objects and functions *(see p14, more detail in Chapter 4, 5…)*
- We can use the pre-written classes, objects and functions inside a library after including it using "`#include`…"
- For example, `cout` and `cin` are objects inside the iostream library. So, we can use them after having the directive: "`#include <iostream>`"

Namespace

```
// without “using namespace std;”
#include <iostream>
void main (){
  std::cout << "Hello world” << endl;
}
```

```
// with “using namespace std;”
#include <iostream>
using namespace std;
void main () {
   cout << "Hello world“ << endl;
}
```

Further Details

- C++ is case-sensitive:
  - E.g. “`Main`” is different from “`main`”.
- Syntax of include directives:
  - no space after < and before >
  - no semi-colon at the end, each include directive must be on its own line
- Statements are ended with semicolons
  - spacing is not important
  - programmer can put in suitable spacing and indentation to increase readability

- Output statement:
  - `<< -` the *output* (or *insertion*) *operator*
  - `cout << "Hello…" -` apply the output operator on the objects `cout` and `"Hello…"`
  - The object `cout` is called *standard output stream* and is defined in `<iostream>`
  - `endl` – causes the cursor to move to the beginning of a new line

Layout of a Simple C++ Program

```
#include <iostream>
using namespace std;

void main()
{
  statement-1
  statement-2
  statement-3
    ...
  statement-last
}
```

Chapter 2

- Input-Process-Output
  - The temperature program
  - Variables, constants, operators & expressions
  - I/O statements

The Temperature Program

```
// convert temperature in Celsius to Fahrenheit
#include <iostream>
using namespace std;

void main()
{
  double C_temp, F_temp;
  cin >> C_temp;
  F_temp = C_temp*9/5 + 32;
  cout << F_temp << “\n”;

}
```

- Two ways to specify comment in C++:
  - enclosed with the pair: /* and */
  - preceded by: // for in-line comment (refer to line *1*)
- Variable definition: (refer to line *7*)
  - "`double C_temp, F_temp`" defines two variables with names `C_temp` and `F_temp`, both of type `double`; and allocates sufficient memory for them
  - End with semi-colons
- Input statement: (refer to line *8*)
  - `>>` - *input* (or *extraction*) *operator*
  - `cin` - *standard input stream*, defined in `<iostream>`
- Assignment statement: (refer to line *9*)
  - Value of expression "`C_temp*9/5+32`" stored in `F_temp`.

A User-friendly Version

```
// convert temperature in Celsius to Fahrenheit
#include <iostream>
using namespace std;

int main()
{
   double C_temp;
   cout << “Enter temperature in Celsius: ”;
   cin >> C_temp;
   cout << “Temperature in Fahrenheit is: ”
      << C_temp*9/5 + 32  << “\n”;
   return 0;
}
```

Variables

- To store data (e.g. user input, intermediate result)
- Implemented as memory locations
- Each variable must be of a *data type*; there are 7 basic data types in C++:
  - `char, wchar_t`: characters/wide characters
  - `int` : integers
  - `float, double`: single/double precision nos.
  - `void` : no value
  - `bool` : Boolean values (true/false)
- Each variable must have a *name*

Identifiers

- Names of variables, functions, and various other user-defined objects are called *identifiers*
- Syntax:
  - first character must be a letter or an underscore
  - subsequent characters must be either letters, digits, or underscores
  - can be of any length but not all of them may be significant
    $\Rightarrow$ Don't use excessively long names
  - cannot be a keyword or be already defined in library
- Legal e.g.: `x1   _abc   X_cor   Big_bonus`
- Illegal e.g.: `2x   ab%c   myfirst.c`

Data Types

- Data of different types are stored in computer memory using different encoding schemes and require different numbers of memory bytes.
- Exact schemes differ from system to system

Typical Encoding Schemes

- `int`
  - 2's complement
  - 4 bytes (32 bits)
  - $[-2^{31}, 2^{31}-1] = [-2147483648, 2147483647]$
- `double`
  - floating point representation
  - 8 bytes
  - approx. $\pm 1.7 \times 10^{308}$ and 15 decimal places
- `char`
  - ASCII
  - 1 byte

Variable Definitions

- Each variable must be defined before use
- Syntax: *`type variable_list;`*
- Examples:
  - `int  nr_cans, i, count;`
  - `double total, average;`
  - `char ch;`
- Depending on the type, an appropriate amount of memory locations will be allocated for each declared variable.
- During execution, the content in these locations will be interpreted appropriately.

Integer and Floating Point Constants

- A sequence of digits (without decimal points) is recognized as an integer constant
  - e.g. 2001
- A constant of type `double` can be specified by:
  - having a decimal point, e.g., 2.0
  - using scientific notations, e.g., 3.67e-1 which stands for $3.67*10^{-1}$
- Do not put comma in the number
  - e.g. `2,001` is wrong

Character and String Constants

- A character constant is specified by
  - a single character enclosed with a pair of single quotes, e.g. `‘A’`   `‘2’`   `‘%’`
  - an escape sequence enclosed with a pair of single quotes, e.g., `‘\n’` newline

    `‘\t’` horizontal tab

    `‘\\’` backslash

    `‘\“’` double quote
- A sequence of character(s) enclosed with a pair of double quotes is recognized as a string constant
  - e.g. `“Hello, world\n”`   `“A”`

Operators & Expressions

- An *operator* specifies some operation to be performed on its operand(s)
- E.g., `x+3` is an *expression* consisting of the addition operator and the operands `x` and `3`.

  `x` is a variable and `3` is a constant
- An expression is a meaningful combination of variables, constants, operators and function calls (to be covered later)
- Each expression has a value and a type.
- We construct expressions to generate the appropriate output values from input values.

Operators

- Some types of operators:
  - Arithmetic
  - Assignment
  - Relational
  - Boolean (Logical)
  - Increment/Decrement

  and many more…

Arithmetic Operators

- `+ - * /` carry the usual meaning: addition, subtraction (or minus sign), multiplication and division
- E.g., the expression `7+4` has value `11`.
- E.g., the expression `7/4` has value `1`; the decimal part (`0.75`) is discarded.
- E.g., the expression `7.0/4` has value `1.75`; the decimal part is kept.
- `%` is the modulo operator
  - e.g. `17%5 = 2`
  - (Question: try `-17%5`)

Level of Precedence & Associativity

- The usual precedence rules apply: `*`, `/` and `%` have higher *precedence* than `+` and `-`.
  - E.g., the expression `24+4*3` has value `36`.
- The *associativity* of them are from left to right
  - E.g. the expression `24/4/2` has value `3`;

    the expression `24/4*3` has value `18`.
- Parenthesis `(  )` can be used to
  - override default precedence rules, e.g. `(x+y)*z`
  - group sub-expressions within a larger expression for clarity.

Assignment Operator

- Symbol: =
- Syntax of an assignment *statement*:

  `variable = expression;`

- E.g.: `unit_price = price/quantity;`

```
F_temp = C_temp * 9/5 + 32;
```

- This is illegal:

```
F_temp–32 = C_temp * 9/5;
```

  because the left side of = must be a variable.

Efficient / Shorthand Assignments

- The statement: `cnt += 2;`
  is equivalent to: `cnt = cnt+2;`
- The same applies to the operators: `- * / %`
  - `cnt *= x+y;      // cnt = cnt*(x+y)`

Variable Initializations

- Variables can be initialized when defined
- E.g. the integer variable `i` will get an initial value of 10:
  - `int i=10;`
- Alternative notations:
  - `int i(10);`
- You can have both initialized and un-initialized variables in the same declaration:
  - `double rate=0.07, time, balance=0.0;`

Type Conversions

- General rule: do not assign data to variables of a different type
- `int i = 2.99;   // problem`
- `double x = 2;   // OK`
- `char c = 65;`

  `// c gets the character with ASCII`

  `// value 65`
- `int i = ‘Z’;`

  `// i gets the ASCII value of ‘Z’`
- Conversions involving other types will be covered later.

Input Using `cin`

- You can input data to more than one variable in the same input statement:
  - e.g. `cin >> x >> y;`
- `cin` is logically an object called *standard input stream*, and is physically linked to the keyboard by default
- Excess input from `cin` not “consumed” by the variables are kept in `cin` and left for the next cin-statements

Output Formatting

- The core C++ language does not specify how I/O's are done
  - I/O operations are usually rather machine dependent
  - I/O facilities are provided by standard libraries, e.g. `<iostream>`
- The library `<iomanip>` provides a number of handy I/O functions for formatting I/O's:
  - `fixed, scientific, setfill(int),`
  - `setprecision(int), setw(int)`

Fixed-point Notation

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(){
  cout << fixed    << setprecision(2)
       << 2343     << endl
       << 2343.0   << endl
       << 2343.346 << endl;
  return 0;
}
```

```
2343
2343.00
2343.35
```

Scientific Notation

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(){
  cout << scientific << setprecision(2)
       << 2343       << endl
       << 2343.0     << endl;
  return 0;
}
```

```
2343
2.34e+003
```

Chapter 3

- Flow of Control
  - Compound & empty statements
  - If/if-else statements
  - Boolean expression
  - While, do-while & for statements
  - Comma
  - Break & continue
  - Switch
  - Structured programming

Flow of Control

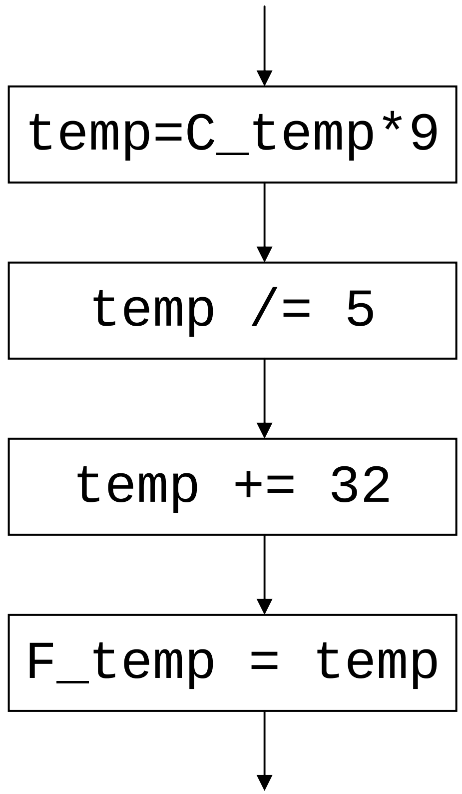
- The *order* of statement execution
- 3 major types of control flow:
  - Sequential
  - Branching
  - Looping

Sequential Structure

- Simply put the statements one after another
- E.g.,

```
temp = C_temp*9;
temp /= 5;
temp += 32;
F_temp = temp;
```

The 4 statements are executed sequentially.



Compound Statements

- A list of statements enclosed in a pair of braces

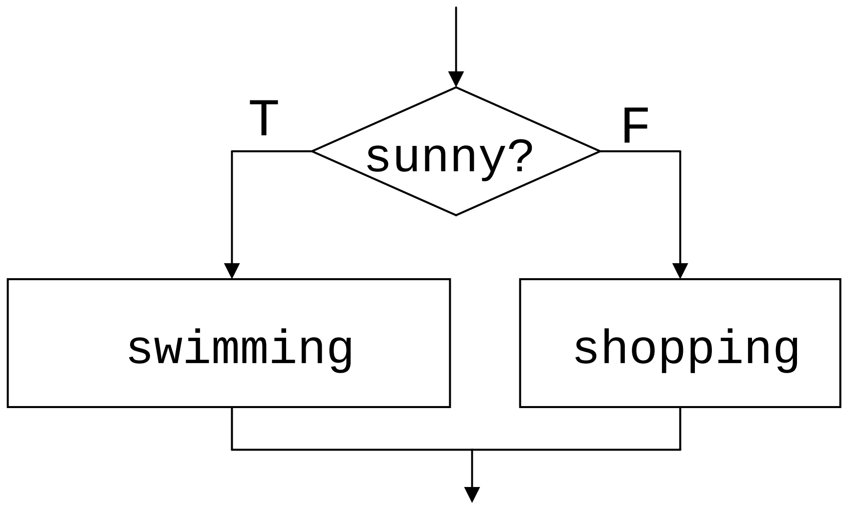
```
{
    statement1
    statement2
      …
    statementn
}
```

- No need to put semicolon after the `}` symbol.
- Treated like a single statement
- Statements inside the braces are executed sequentially.

Branching: if and if-else statements

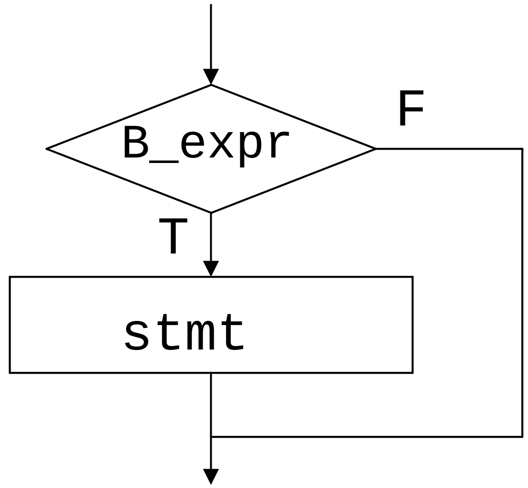
- Suppose we have an "algorithm" to determine what we will do depending on the weather:

```
if it is sunny
   go swimming
else
   go shopping
```



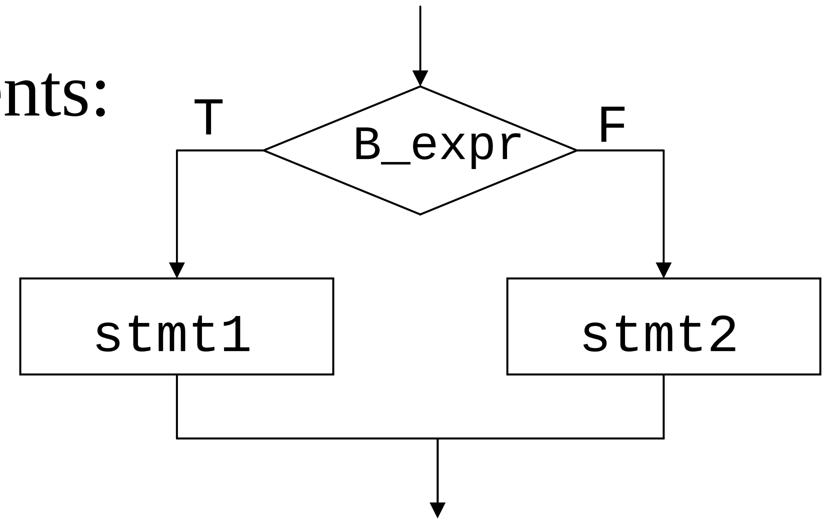
- Syntax of if-statements:

```
if (Bool_expr)
    statement
```



- Syntax of if-else statements:

```
if (Bool_expr)
    statement1
else
    statement2
```



Matching the if- and else-parts

```
if (temperature < 100)
      if (temperature <= 0)
            cout << “Water is frozen\n”;
else
      cout << “Water is all boiled away\n”;
```

- What will happen if temperature = 90?
  - The else-part is matched with the nearest unmatched if-part above it
  - Indentation doesn’t matter
- How can the code be corrected?
  - Use brackets: `{ }`.

Boolean (Logical) Expressions

- Any expression that is either true or false
- Relational operators: `>  <  >=  <=  ==  !=`
  - Do not insert spaces between the 2 symbols
  - Do not reverse the order of the 2 symbols
  - e.g. `x>7` is a Boolean expression
- Boolean (logical) operators: `&&  ||  !`
  - Do not insert spaces between the 2 symbols
  - e.g. `(x>7) && (x<10)`

    or simply, `x>7 && x<10` but not `7<x<10`
  - e.g. `!(x<y)    // parenthesis necessary`

Mixing up = and ==

- The following if-statement does not generate any compile-time error:

```
if (x=12)
  cout << “It is twelve\n”;
```

- The assignment expression `x=12` has a value of `12` (this is by definition);

  value `12` is non-zero and treated as `true` (type conversion)
- So, the string `“It is twelve\n”` is always printed.
- A small trick:

```
if (12==x) ...
```

Short-circuit Evaluation

- Evaluation of expressions containing && and `||` stops as soon as the outcome is known
- Improve program efficiency
- Easier to write programs
  - E.g.
    ```
    cin >> x >> y;
    if (y!=0 && x/y>10)
        ...
    ```
  - Without short-circuit evaluation, you need to write:
    ```
    cin >> x >> y;
    if (y!=0)
        if (x/y>10) ...
    ```

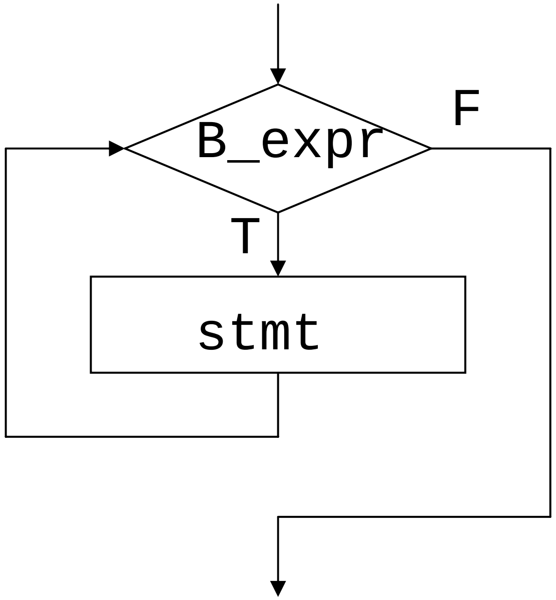
Looping: while-statement

- Syntax of while statement:

  `while` (*`Bool_expr`*)

  `statement`

- If *`Bool_expr`* is false, the loop body *`statement`* is never executed
- If *`Bool_expr`* is true, *`statement`* is executed
- Then *`Bool_expr`* is checked again and *`statement`* is executed repeatedly until *`Bool_expr`* is false
- *`statement`* can be a compound statement



Example: Class Average (fixed size)

- Problem Specification:

  Write a program to calculate the class average of a quiz. There are 10 students in the class and the grades are integers ranging from 0 to 100.

- Pseudocode:

```
set total to 0
set counter to 1
while counter <= 10
   input the next grade
   add grade to total
   increase counter by 1
calculate total/10 and output the result
```

Program: Class Average (fixed size)

```
#include <iostream>
using namespace std;

int main(){
   int total, count, grade;
   total = 0;  count = 1;
   while (count <=10) {
      cout << “Enter marks: ”;
      cin >> grade;
      total += grade;
      count +=1;
   }
   cout << “Average of 10 students = "
        << total/10.0 << endl;
   return 0;
}
```

Increment & Decrement Operators

- Can be applied on variables
- E.g., `count++;`
  increases `count` by 1. The ++ operator called *post-increment operator*
- E.g., `++count;`
  also increases `count` by 1. The ++ operator called *pre-increment operator*
- E.g., `cout << count++;`
  outputs the old value of `count`
- E.g., `cout << ++count;`
  outputs the new value of `count`
- The decrement operator `--` is to decrease the content of a variable

Ex: Class Average (variable size)

- Problem Specification:

  Write a program to calculate the class average of a quiz. The grades are integers ranging from 0 to 100. The class size is arbitrary. The end of data is marked by a –1.

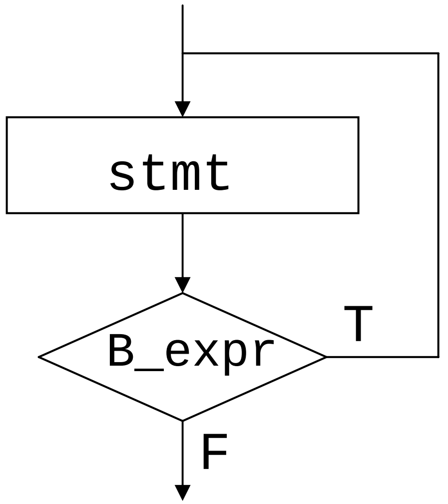
- Idea:

```
(1) initialize variables
(2) input, sum and count the grades
(3) calculate and output the class average
```

Looping: do-while-statements

- When you know that the loop body has to be executed at least once, use a do-while-statement
- Syntax:

```
do
   statement
while (Bool_expr);
```



- Semantics:
  - *statement* is executed once and then *Bool_expr* is evaluated.
  - The above is repeated as long as *Bool_expr* is true.

Looping: for-statements

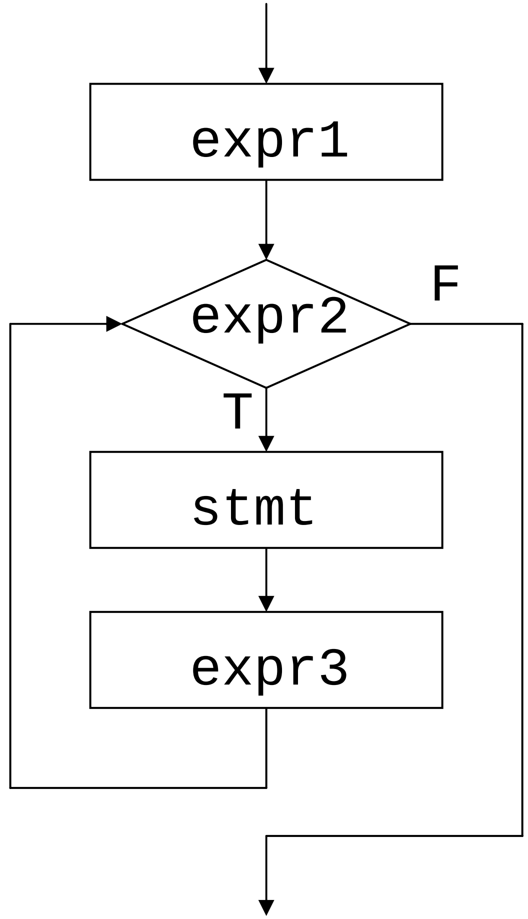
- Syntax:

```
for (expr1; expr2; expr3)
    statement
```

- Semantics equivalent to:

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

provided `expr2` exists, and `statement` doesn't contain a continue-statement (discussed later)



Looping: for-statements

- Usually,
  - `expr1` is to initialize some loop controlling variables, e.g. `count=1;`
  - `expr2` is a loop test, e.g. `count<=10;`
  - `expr3` is to update the loop controlling variable, e.g. `count++;`

Example (c.f. p13)

```
#include <iostream>
using namespace std;

int main(){
   int total, count, grade;
   total = 0;
   for (count=1; count <=10; count++) {
      cout << “Enter marks: ”;
      cin >> grade;
      total += grade;
   }
   cout << “Average of 10 students = "
        << total/10.0 << endl;
   return 0;
}
```

Comma Operator (,)

We can also place the assignment: `total=0` into the for-loop:

```
#include <iostream>
using namespace std;

int main(){
   int total, count, grade;
   for (total=0, count=1; count <=10; count++) {
      cout << “Enter marks: ”;
      cin >> grade;
      total += grade;
   }
   cout << “Average of 10 students = "
        << total/10.0 << endl;
   return 0;
}
```

Un-intentional Empty Statements

- Never insert any excessive semi-colons.
- E.g.:

```
int total, count, grade;
total = 0;
for (count=1; count <=10; count++);{
   cout << “Enter marks: ”;
   cin >> grade;
   total += grade;
}
cout << “Average of 10 students = "
     << total/10.0 << endl;
```

Empty Statements

- No action is done by an empty statement
- An empty statement can be specified by a semi-colon or a pair of `{}` without any statement inside
- Wherever it is syntactically correct to place a statement, it is also syntactically correct to place an empty statement.

Continue-statement

- Causes the current iteration of a loop to stop and the next iteration to begin immediately
- Applicable to a while, do-while and for-statement

```
// read in 10 numbers
// and sum only the positive ones
   sum = 0.0;
   for (cnt=0; cnt<10; cnt++) {
     cin >> x;
     if (x<=0)
        continue;
     sum += x;
   }
```

- A continue-statement can often be avoided:

```
// read in 10 numbers
// and sum only the positive ones.
   sum = 0;
   for (cnt=0; cnt<10; cnt++) {
     cin >> x;
     if (x > 0)
       sum += x;
   }
```

Another Example (c.f. p23)

```
// read in 10 positive numbers and sum them
   sum = 0.0;
   cnt = 0;
   while (cnt<10) {
     cin >> x;
     if (x <= 0)
       continue;  // skip non-positive values
     sum += x;
     cnt++;
     // continue transfers control here
  }
```

Break-statement

- Causes an exit from the innermost enclosing loop or switch-statement (discussed later)
- Applicable to while, do-while, for and switch-stmt

```
while (true) {
  cin >> x;
  if (x < 0)
    break;    // exit loop if x negative
  //  compute square root of x
  }
// Break transfers control here
```

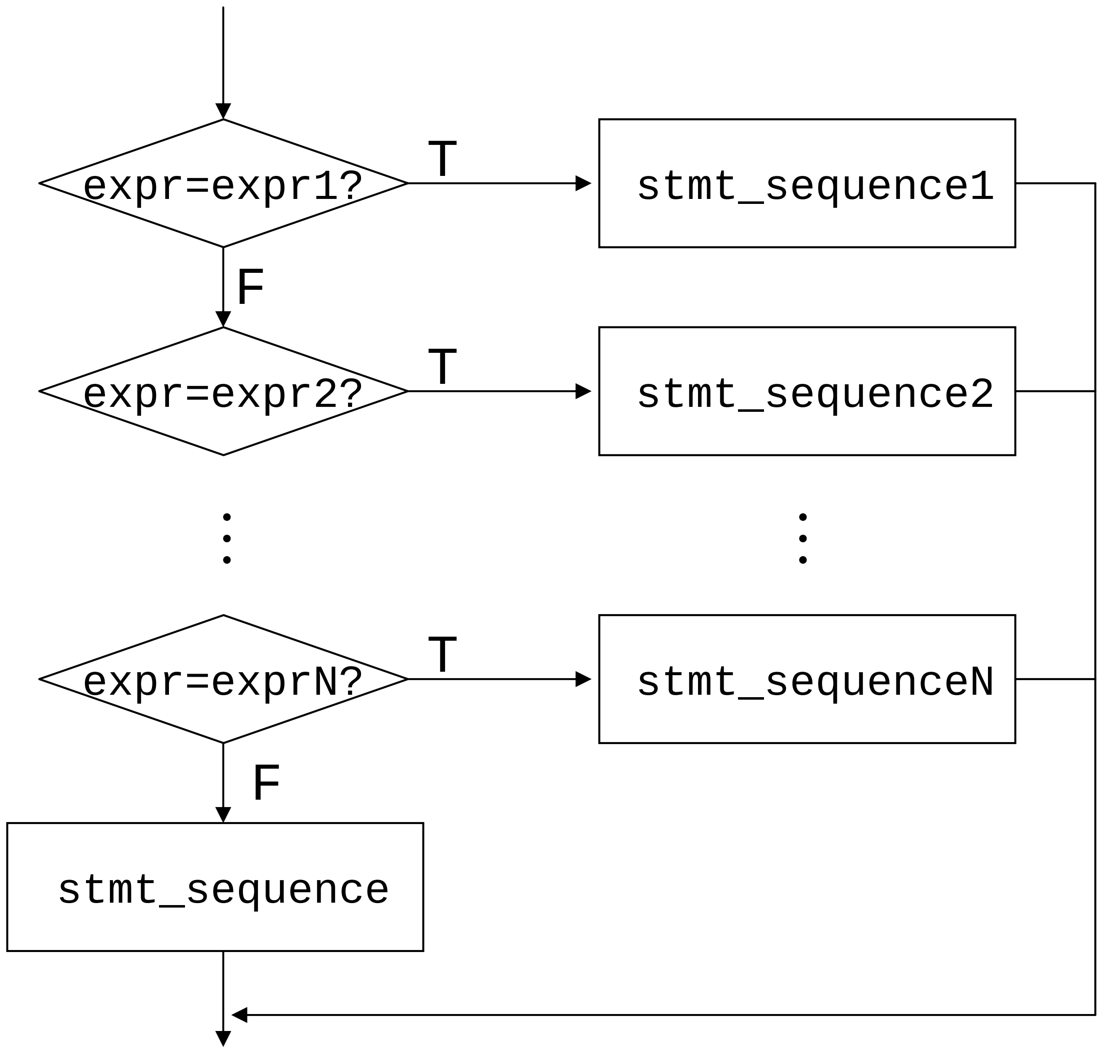
Branching: switch-statements

- Useful for multi-way branching
- Syntax:

```
switch (expr) {
  case expr1: stmt_sequence1
  case expr2: stmt_sequence2
   ...
   ...
  case exprN: stmt_sequenceN
  default: stmt_sequence
}
```

- If all `stmt_sequence1`, `stmt_sequence2`, etc. ended with a `break` statement, control flow shown in next page:

Flowchart:


expr=expr1?
T
stmt_sequence1
F
expr=expr2?
T
stmt_sequence2
expr=exprN?
T
stmt_sequenceN
F
stmt_sequence

```
//  To print no. of days in a month
cin >> month;   // month is int variable
switch (month) {
  case 1:  cout << “31 days”; break;
  case 2:  cout << “28 or 29 days”; break;
  case 3:  cout << “31 days”; break;
  case 4:  cout << “30 days”; break;
  case 5:  cout << “31 days”; break;
  case 6:  cout << “30 days”; break;
  case 7:  cout << “31 days”; break;
  case 8:  cout << “31 days”; break;
  case 9:  cout << “30 days”; break;
  case 10: cout << “31 days”; break;
  case 11: cout << “30 days”; break;
  case 12: cout << “31 days”; break;
  default: cout << “input error”; break;
}
```

- Some rules:
  - `expr` must have integral data type (`int`, `char` and `bool`)
  - `expr1`,…`exprN` must be constant expressions and must be all different
  - The default case is optional
  - If the sequence of statements for the matched case does not end with a `break` statement, execution will “fall through” to the next case.

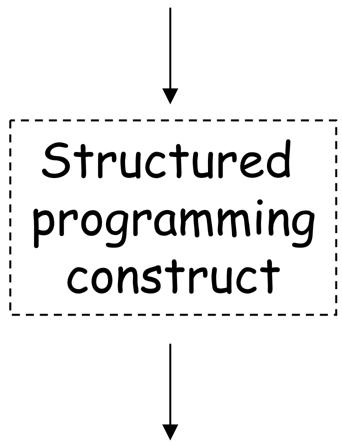
```
//  To print no. of days in a month
cin >> month;   // month is int variable
switch (month) {
  case  1:
  case  3:
  case  5:
  case  7:
  case  8:
  case 10:
  case 12: cout << “31 days”; break;
  case  4:
  case  6:
  case  9:
  case 11: cout << “30 days”; break;
  case  2: cout << “28 or 29 days”; break;
  default: cout << “input error”; break;
}
```

Structured Programming

- The flow of control of a structured program is easier to follow
- How?
  - Apply stepwise refinement to decompose a problem into smaller problems repeatedly until they are simple enough to be coded
  - In each refinement, apply one of the 3 control flow structures (p2). In C++, these are realized by using sequence of statements, branching (if, if-else, switch) statements and looping (while, do-while, for) statements

Structured Programming (cont')

- We generally prefer our structures to have single-entry and single-exit



- A loop containing break or continue statements will have more than one entry/exit.
- We should use continue/break statements in a loop with great care.

Chapter 7

- Class
  - Defining classes
  - Defining member functions & scope resolution operator
  - Accessing members & dot operator
  - Public & private members
  - Accessors
  - Constructors

Classes

- A class is a data type whose variables are objects
- An object is a variable with member functions and data values
- `ifstream`, `ofstream` are classes defined in header `<fstream>`
- `cin`, `cout` are objects defined in header `<iostream>`
- C++ has great facilities for you to define your own class and objects

Defining classes (example)

```
#include <iostream>
using namespace std;
class DayOfYear
{
public:
  void output(); //member func. prototype
  int month;
  int day;
};
void main()
{
  DayofYear today, birthday;
  cin >> today.month >> today.day;
  cin >> birthday.month >> birthday.day;
```

Defining class (example cont'd)

```
  cout << "Today’s date is: ";
  today.output();
  cout << "Your birthday is: ";
  birthday.output();
  if (today.month == birthday.month
      && today.day == birthday.day)
      cout << "Happy Birthday!\n";
}

void DayOfYear::output()
{
  cout << "month =" << month
       << ", day =" << day << endl;
}
```

Remarks

- The class `DayOfYear` has 1 member function: `output`, and 2 member variables: `month` and `day`.
- The keyword `public` states that all the three members `output`, `month`, `day` can be accessed freely by other parts of the program, e.g. the main function
- A class definition contains only the prototypes of its member functions (except for *inline functions*)
- A member function of an object is called using the dot operator: `today.output()`

- When defining a member function, the class name has to be specified because different classes can have member functions of the same name
- The operator `::` is called the *scope resolution operator* (Don’t confuse it with dot operator.)
- Inside the function definition of `DayOfYear::output`, the member names `month` and `day` are used without specifying the object.
- This creates no confusion because when you call `output`, you have to specify the object:

```
today.output();
```

Public and private members

- By default, all members of a class are private
- You can declare public members using the keyword `public`
- You can explicitly declare private members using the keyword `private` (*a good programming style*)
- Private members can be accessed only by member functions (and *friend* functions) of that class
- E.g. using the new class definition next page,

```
DayOfYear today;   // OK
today.month = 13;  // illegal
```

A new class definition for DayOfYear

```
class DayOfYear
{
public:
  void input();
  void output();
  void set(int new_m, int new_d);

  int get_month();
  int get_day();
private:
  bool valid(int m, int d); // check if m,d valid
  int month;
  int day;
};
```

Member function definitions

```
void DayOfYear::input()
{
  int m, d;

  // input and validate
  do {
    cout << "Enter month and day as numbers: ";
    cin >> m >> d;  // local var. of input()
  } while (!valid(m,d));

  month = m;  // accessing private members
  day = d;
}
```

Member function definitions (cont'd)

```
void DayOfYear::set(int new_m, int new_d)
{
  if (valid(new_m, new_d)) {
     month = new_m;
     day   = mew_d;
     }
}

int DayOfYear::get_month()
{  return month;
}

int DayOfYear::get_day()
{  return day;
}
```

Member function definitions (cont'd)

```
bool DayOfYear::valid(int m, int d)
{
  if (m<1 || m>12 || d<1) return false;
  switch(m){
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
         return d<=31; break;
    case 4: case 6: case 9: case 11:
         return d<=30; break;
    case 2:
         return d<=29; break;
    }
}
```

A new main program

```
void main()
{ DayOfYear today, birthday;

  today.input();
  birthday.input();
  cout << "Today's date is:\n";
  today.output();
  cout << "Your birthday is:\n";
  birthday.output();

  if (today.get_month()==birthday.get_month()
                      &&
      today.get_day() == birthday.get_day())
    cout << "Happy Birthday!\n";
}
```

Access functions

- A private member variable can only be accessed through one of the member functions
- Member functions that give you access to the values of the private member variables are called *access functions*, e.g., `get_month`, `set`
- Useful for controlling access to private members:
  - E.g. Provide data validation to ensure data integrity.
- Unnecessary access functions should not be defined. It is not a must for each member variable to have a pair of get and set functions (see “Day of Year” example version 2).

Why private members?

- A class definition should separate the rules for using the class (the *interface*) and the details of the class *implementation* as much as possible
- An analogy: You use predefined type `double` without worrying (much) which compiler you are using, even though different compilers have slightly different implementation.
- Ideally, you should be able to change the details of a class implementation by changing the member function definitions and private member variables only

- This is often not possible if your program accesses the member variables directly.
- Therefore, the strongly suggested style of class definitions is:
    - To have all member variables private
    - Provide enough access functions to get and set the member variables
    - Supporting functions used by the member functions should also be made private

Syntax of class definitions

```
class class_name
{
public:
  member_spec_1
  ...
  member_spec_n
private:
  member_spec_n+1
  member_spec_n+2
  ...
};
```

- `member_spec_1` is either a member variable declaration or a member function prototype.

Assignment operator for objects

- It is legal to use assignment operator = with objects or with structures
- E.g. `DayOfYear due_date, tomorrow;`

```
tomorrow.input();
due_date = tomorrow;
```

Constructors for initialization

- A *constructor* is a member function that is automatically called when an object of that class is declared
- Special rules:
  - A constructor must have the same name as the class
  - A constructor definition cannot return a value
- E.g., Suppose we want to define a bank account class which has member variables `balance` and `interest_rate`. We want to have a constructor that initializes the member variables.

```
class BankAcc
{
public:
  BankAcc(int dollars, int cents, double rate);
  ...
private:
  double balance;
  double interest_rate;
};
...
BankAcc::BankAcc(int dollars, int cents,
                 double rate)
{  balance = dollars + 0.01*cents;
   interest_rate = rate;
}
```

- No return type is specified in the function header (not even the type `void`)
- When declaring objects of `BankAcc` class:

```
BankAcc account1(10,50,2.0),
        account2(500,0,4.5);
```

- 2 objects of `BankAcc` class are declared and the constructor is called to initialize the member vars.
- A constructor cannot be called in the same way as an ordinary member function is called:

```
account1.BankAcc(10,20,1.0); // illegal
```

- Constructors are usually overloaded so that objects can be initialized in more than one way, e.g.

```
class BankAcc
{
public:
  BankAcc(int dollars, int cents, double rate);
  BankAcc(int dollars, double rate);
  BankAcc();
  ...
private:
  double balance;
  double interest_rate;
};
```

```
BankAcc::BankAcc(int dollars, int cents,
                 double rate)
{  balance = dollars + 0.01*cents;
   interest_rate = rate;
}

BankAcc::BankAcc(int dollars, double rate)
{  balance = dollars;
   interest_rate = rate;
}

BankAcc::BankAcc()
{  balance = 0;
   interest_rate = 0.0;
}
```

- When the constructor has no arguments, don’t include any parentheses in the object declaration.
- E.g.

```
BankAcc acc1(100, 50, 2.0), // OK
        acc2(100, 2.3),     // OK
        acc3(),           // error
        acc4;           // correct
```

- The compiler thinks that it is the prototype of a function called `acc3` that takes no arguments and returns a value of type `BankAcc`

- Once you have a good set of constructors, there is no need for other member functions to set the private member variables
- Alternative way to call a constructor:

  `obj = constr_name(arguments);`
- E.g.,
```
BankAcc account1;
account1 = BankAcc(200, 3.5);
```
- Mechanism: calling the constructor creates an anonymous object with new values; the object is then assignment to the named object
- A constructor behaves like a function that returns an object of its class type

Default constructors

- A constructor with no parameters
- Sometimes you want to declare an object without giving any arguments
- E.g.

```
class SampleClass {
public:
  SampleClass(int param1);
  SampleClass();
...
};
SampleClass::SampleClass(){}
```