Lecture 3

C++ Programming

Arne Kutzner

Hanyang University / Seoul Korea

`const`

- You can add the **`const`** modifier to the declaration of a variable to tell the compiler that the value cannot be changed:

```
const double factor = 5.0/9.0;
const double offset = 32.0;
celcius = (fahr - offset)*factor;
```

What if you try to change a `const`?

- The compiler will complain if your code tries to modify a const variable:

```
const foo = 100;
…
foo = 21;
```

Error: l-value specifies const object

Why use **`const`**?

- const tells the compiler that a variable should never be changed.
- Can be used for increasing code quality
- Can help the compiler creating more performant executables

Integer vs. floating point math

- How does C++ *know* whether to use floating point or integer math operators?
  - If either operand is floating point, a floating point operation is done (the result is a floating point value).
  - If both operand are integer the result is an integer (even division).

Math Operator Quiz

What are the values printed?

```
const int five = 5;
int i = 7;
float x = 7.0;
cout << five + i/2 << endl;
cout << five + x/2 << endl;
```

Control Structures

- Unless something special happens a program is executed sequentially.
- When we want something special to happen we need to use a control structure.
- Control Structures provide two basic functions: selection and repetition

Selection

- A Selection control structure is used to choose among alternative courses of action.
- There must be some *condition* that determines whether or not an action occurs.
- C++ has a number of selection control structures:
  - If
  - if/else
  - switch

Repetition Control Structures

- Repetition control structures allow us to repeat a sequence of actions (statements).
- C++ supports a number of repetition control structures:
  - while
  - for
  - do/while

`if`

- The `if` control structure allows us to state that an action (sequence of statements) should happen only when some condition is true:

```
if (condition)
    action;
```

Conditions

- The *condition* used in an **`if`** (and other control structures) is a Boolean value - either **`true`** or **`false`**.
- In C++ (like in C) Boolean values are represented via integers:
  - the value **0** is **`false`**
  - **any other value** is **`true`**

`if` examples

```
if (5) // not 0 -> true
   std::cout << "I am true!\n";

if (0) // 0 -> false
   std::cout << "I am false!\n";
```

Relational and Equality Operators and Conditions

- Typically a condition is built using the C++ relational and equality operators.
- These operators have the values `true` (1) and `false` (0).
- So the expression `x==x` has the value `true`.
- and `7 <= 3` has the value false.

More **ifs**

```
if (foo)
  std::cout << "foo is not zero\n";

if (grade>=90)
  lettergrade = 'A';

if (lettergrade == 'F')
  std::cout << "The system has failed you\n"
```

Common Mistake

- It is easy to mix up the assignment operator "=" with the equality operator "==".
- What's wrong with this:

```
if (grade = 100)
   std::cout << "Your grade is
     perfect ...\n";
```

Compound Statements

- Inside an **`if`** you can put a single statement or a compound statement.
- A compound statement starts with "{", ends with "}" and can contain a sequence of statements (or control structures)

```
if (grade>=90) {
   cout << "Nice job - you get an A\n";
   acnt = acnt+1;
}
```

A word about style

- C++ doesn't care about whitespace (including newlines), so you can arrange your code in many ways.
- There are a couple of *often-used* styles.
- All that is important is that the code is easy to understand and change!

Some common styles

```
if (foo>10) {
  x=y+100;
  cout << x;
}
```

```
if (foo>10)
  {
  x=y+100;
  cout << x;
  }
```

```
if(foo>10){x=y+100;cout<<x;}
```

`if else` Control Structure

- The **`if else`** control structure allows you to specify an alternative action:

```
if ( condition )
     action if true
else
     action if false
```

if else example

```
if (grade >= 90)
  lettergrade = 'A';
else
  lettergrade = 'F';
```

Another example

```
if (grade >= 99)
  lettergrade = 'A';
else if (grade >= 98)
  lettergrade = 'B';
else if (grade >= 97)
  lettergrade = 'C';
else if (grade >= 96)
  lettergrade = 'D';
else
  lettergrade = 'F';
```

switch Statement

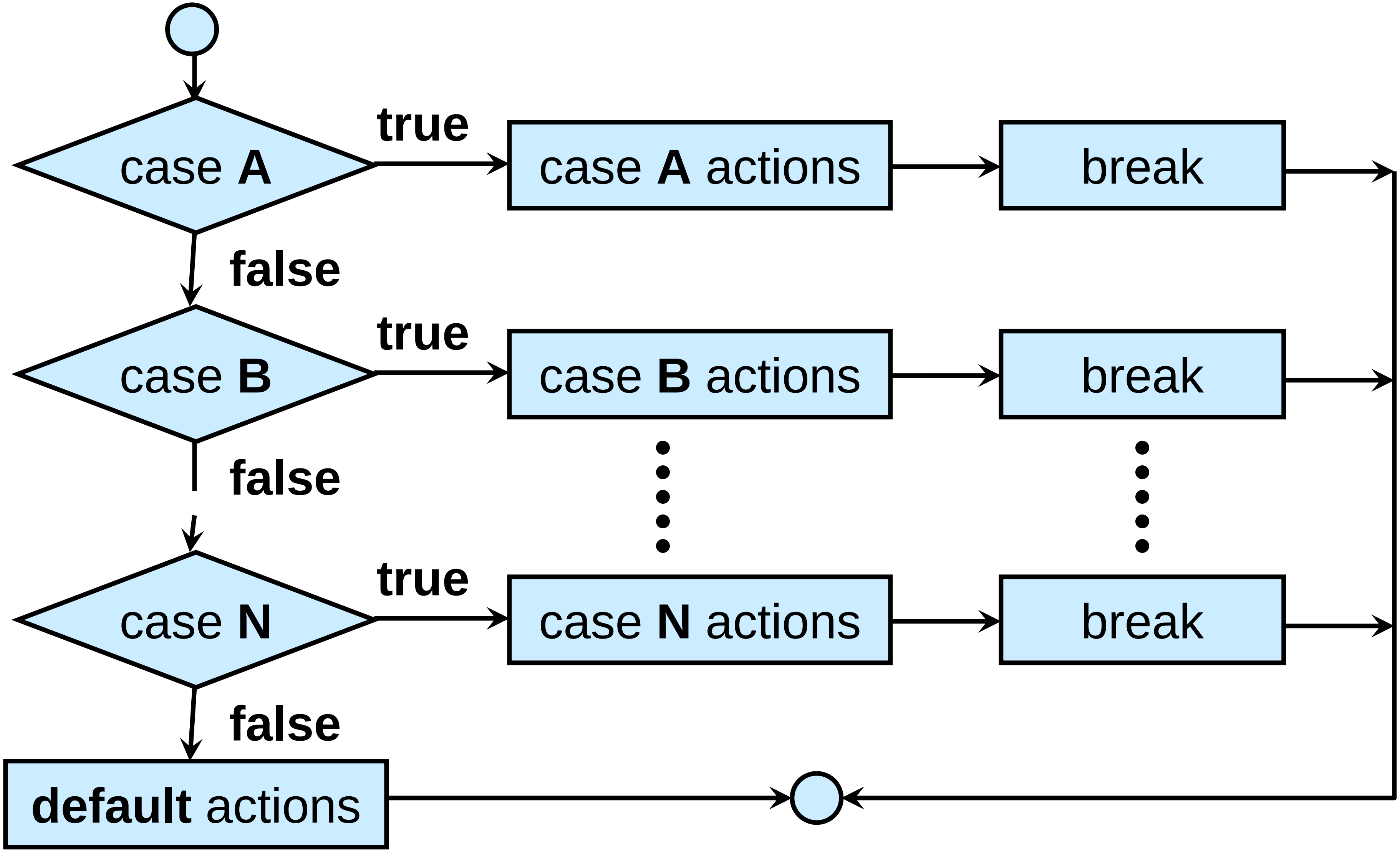
- To select one of several alternatives.
- Selection is based on the value of an expression.
- Expression could be a single value.
- The type of expression can be either **`int`** or **`char`**, but not **`double`**.

switch Statement

- **Syntax**

```
switch (expression) {
   case const-expr :
      statements;
      break;
   case const-expr :
      statements;
      break;
   …
   default
      statements;
      break;
}
```

switch Statement / Flow diagram



while Control Structure

- The **while** control structure supports repetition - the same statement (or compound statement) is repeated until the condition is false.

```
while (condition)
  do something;
```

the inside is called the "body of the loop"

`while` example

```
lettergrade = 'A';
cutoff = 90;
while (grade < cutoff) {
  lettergrade = lettergrade + 1;
  cutoff = cutoff - 10;
}
if (lettergrade > 'F')
  lettergrade = 'F';
```

`break` and **`continue`** in loops

- The keyword **break** can be used to terminate a loop
- The keyword **continue** can be used to jump immediately to evaluation of the condition

Increment and decrement operators

- You can increment an integer variable like this:

```
// same as lettergrade = lettergrade + 1;
lettergrade++;
```

- You can also decrement:

```
// same as lettergrade = lettergrade - 1;
lettergrade--;
```

- There is prefix and suffix notation
  - **`lettergrade++;`**
    Here we read first the value and increment after reading.
  - **`++lettergrade;`**
    Here we increment first and read the value after incrementing.

Special assignment operators

- This C++ statement:

```
foo += 17;
```

  - is shorthand for this:

```
foo = foo + 17;
```

- You can also use:

```
-=   *=   /=
```

`while` example modified

```
lettergrade = 'A';
cutoff = 90;
while (grade < cutoff) {
  lettergrade++;
  cutoff -= 10;
}
if (lettergrade > 'F')
  lettergrade = 'F';
```

`do while`

- The **`do while`** control structure also provides repetition, this time the condition is at the bottom of the loop.
  - the body is always executed at least once

```
do
  somestuff;
while ( condition );
```

do while example

```
i=1;
do
  std::cout << "i is " << i++ << endl;
while (i <= 10);
```

`for` loops

- The `for` control structure is often used for loops that involve counting.
- You can write any `for` loop as a `while` (and any `while` as a `for`).

```
for (initialization; condition; update)
   dosomething;
```

```
for (initialization; condition; update)
```

- initialization is a statement that is executed at the beginning of the loop (and never again).
- the body of the loop is executed as long as the condition is true.
- the update statement is executed each time the body of the loop has been executed (and before the condition is checked)

for example

```
for (i=1; i<10; i++)
  cout << "i is " << i << endl;

for (i=10; i>=0; i--)
  cout << "i is " << i << endl;
```

Another **`for`** example

initialization

condition

update

```
for (lettergrade = 'A', cutoff = 90;
    grade < cutoff; lettergrade++)
  cutoff -= 10;

if (lettergrade > 'F')
  lettergrade = 'F';
```

Yet another “odd” example

```
for (i=1; i<100;i++) {
  std::cout << "Checking " << i << std::endl;
  if ( i%2 )
     std::cout << i << " is odd" << std::endl;
  else
     std::cout << i << " is even" << std::endl;
}
```

More about **`for`**

- You can leave the initialization, condition or update statements blank.
- If the condition is blank the loop never ends!

```
for (i=0; ;i++)
   Std::cout << i << endl;
```

Complex Conditions

- You can build complex conditions that involve more than one relational or equality operator.
- The **&&** (means "and") and **||**(means "or") operators are used to create complex conditions.
- More operators means another precedence table...

Updated Precedence Table

Operators	Precedence
`()`	highest (applied first)
`++ --`	
`* / %`	
`+ -`	
`< <= > >=`	
`== !=`	
`&&`	
`||`	
`=`	lowest (applied last)

`&&` Operator

- **`&&`** is a boolean operator, so the value of an expression is **`true`** or **`false`**.

`( `*`cond1`*` && `*`cond2`*` )`

is true only if both ***`cond1`*** and ***`cond2`*** are true.

&& Example

```
lettergrade = 'A';
cutoff = 90;
while ((grade<cutoff) && (lettergrade!='F')) {
  lettergrade++;
  cutoff -= 10;
}
```

|| Operator

- `||` is a boolean operator, so the value of an expression is `true` or `false`.

```
( cond1 || cond2 )
```

is true if either of *`cond1`* or *`cond2`* is true.

|| Example

```
if ((test1==0) || (test2==0))
  std::cout << "You missed a test!\n";

if ((hw1==0) || (hw2==0))
  std::cout << "You missed a
  homework!\n";
```

The `!` operator

- The ! operator is a *unary* boolean operator
  - unary means it has only 1 operand.
- **`!`** negates it's operand.
- **`!`** means "not".

`(!` ***`condition`*****`)`**

is true only when `condition` is false

! example

```
bool done = false;
int i=1;

while (!done) {
  std::cout << "i is " << i << "\n";
  i++;
  if (i==100)
     done=true;
}
```