Module 3 Guidance Notes

C++ Basics

ENGG1340
Computer Programming II

COMP2113
Programming Technologies

Estimated Time of Completion: 3 Hours

Important Note: C++ 11 Standard

- We will deal with C++ only in this module. and will leave the C counterparts on I/O (i.e., input/output) handling to latter modules.
- Important: We will be using the C++ 11 standard, so make sure that your compiler option is set appropriately. We suggest to use the following command to compile your C++ program in Linux:

```
g++ -pedantic-errors -std=c++11
your_program.cpp
```

The -pedantic-errors flag is to make sure that your code conforms to the ISO C/C++ standard. We will enforce this in your assignment submission too.
For more information about C/C++ standards, you may read https://en.wikipedia.org/wiki/ANSI_C and https://isocpp.org/std/the-standard

Important Note: Guidance Notes

- Our guidance notes aim at leading you through the learning of the materials. It also defines the scope of our course (say what we expect that you should know for the purpose of this course).
- Pages marked with “Reference Only” or “Optional” mean that they are not in the scope of assessment for this course.

Important Note: Guidance Notes

- We suggest you to copy every code segment in the notes to the coding environment and try run the program yourself.
- Also, try make change to the code, then observe the output and deduce the behavior of the code. This way of playing around with the code can help give you a better understanding of the programming language.

Outline

(P. 6 – 11) Part I: Program Compilation & Execution

(P. 13 – 59) Part II: Basic Operations

- Variables & Constants
- Operators
- Expressions
- Data Types & Type Conversions
- Basic Input/Output

(P. 60 – 129) Part III: Flow of Control

- Branching
- Looping

Part I

PROGRAM COMPILATION & EXECUTION

The First C++ Program

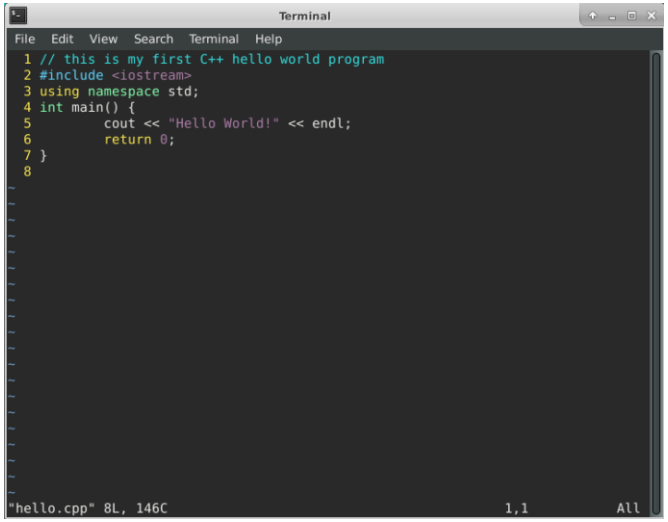
As usual, we will start with the Hello World program.

```
// this is my first C++ hello world program
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Now, copy the code and save it in a file named hello.cpp in your home directory.

Program Editing

In the Ubuntu (Linux) environment that you have been working on for the previous modules, you may use vi or any other text editor to edit your program.



hello.cpp in the vi editor

Compiling and Execution

In Linux environment, we can rely on command line (via the terminal) for compiling and executing your program.

Now, suppose you already have hello.cpp in your current working directory.

```
twchim@academy11 module3> ls
hello.cpp
twchim@academy11 module3> cat hello.cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
twchim@academy11 module3>
```

1. Use this command line to compile hello.cpp:
   `g++ -pedantic-errors -std=c++11 hello.cpp -o hello`
2. If the compilation is successful, you should find another file "hello" in the working directory.
3. Run the executable "hello" by typing "./hello" at the prompt

```
twchim@academy11 module3> g++ -pedantic-errors -std=c++11 hello.cpp -o hello
twchim@academy11 module3> ls -l
total 17
-rwx------ 1 twchim ta 8920 Jul 20 17:29 hello*
-rw------- 1 twchim ta  103 Jul 20 17:29 hello.cpp
twchim@academy11 module3> ./hello
Hello World!
twchim@academy11 module3>
```

Compiling and Execution

Now try again to mess up with your code.

1. Delete line 3 “using namespace std;”
2. Compile and run the executable, and note what the error message is.

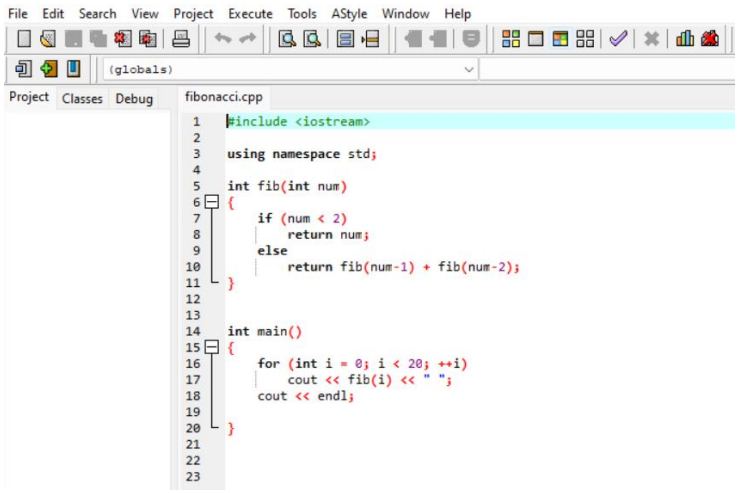
```
twchim@academy11 module3> ls
hello.cpp
twchim@academy11 module3> cat hello.cpp
#include <iostream>

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
twchim@academy11 module3> g++ -pedantic-errors -std=c++11 hello.cpp -o hello
hello.cpp: In function 'int main()':
hello.cpp:4:3: error: 'cout' was not declared in this scope
   cout << "Hello World!" << endl;
   ^~~~
hello.cpp:4:3: note: suggested alternative:
In file included from hello.cpp:1:0:
/usr/include/c++/7/iostream:61:18: note:   'std::cout'
   extern ostream cout;  /// Linked to standard output
                  ^~~~
hello.cpp:4:29: error: 'endl' was not declared in this scope
   cout << "Hello World!" << endl;
                            ^~~~
hello.cpp:4:29: note: suggested alternative:
In file included from /usr/include/c++/7/iostream:39:0,
                 from hello.cpp:1:
/usr/include/c++/7/ostream:590:5: note:   'std::endl'
     endl(basic_ostream<_CharT, _Traits>& __os)
     ^~~~
twchim@academy11 module3>
```

Hints on Debugging

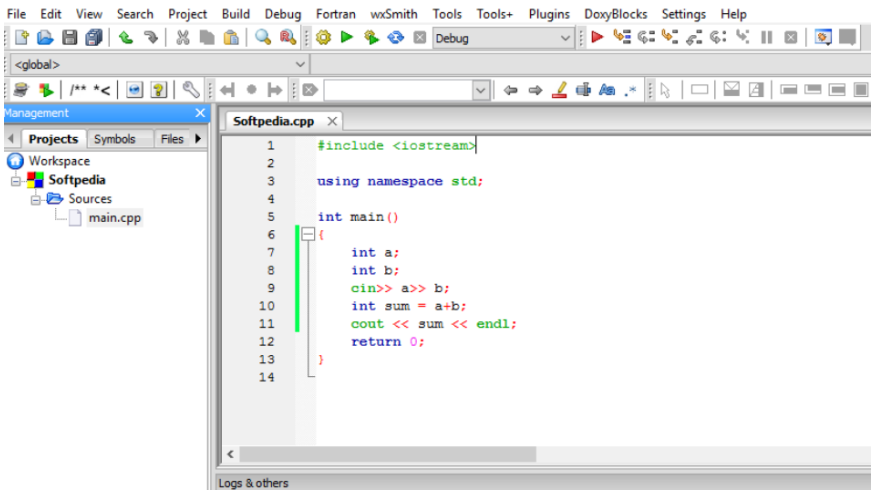
- Hint 1: The line number of an error reported by the compiler may be incorrect. It is possible that the error is located before the reported line. After all, the compiler can only try its best to guess what you meant to write down.
- Hint 2: For the same above reason, the nature of an error reported by the compiler may be incorrect.
- Hint 3: If your source code has multiple errors, always fix the first error and recompile, and repeat the process until the compilation is successful. This is because error messages subsequent to the first one have a higher likelihood of being incorrect.

Offline options for Windows users

- If you are using Windows computer, you can use IDEs like Dev-C++ (https://www.bloodshed.net/) or Code::Blocks (https://www.codeblocks.org/) to write simple C++ programs locally.
- They support simple C++ programming only. Advanced functions like input / output redirection and separate compilation are not supported.
- Before assignment submission, you **MUST** test your programs on Linux platform.



Dev-C++ IDE for Windows



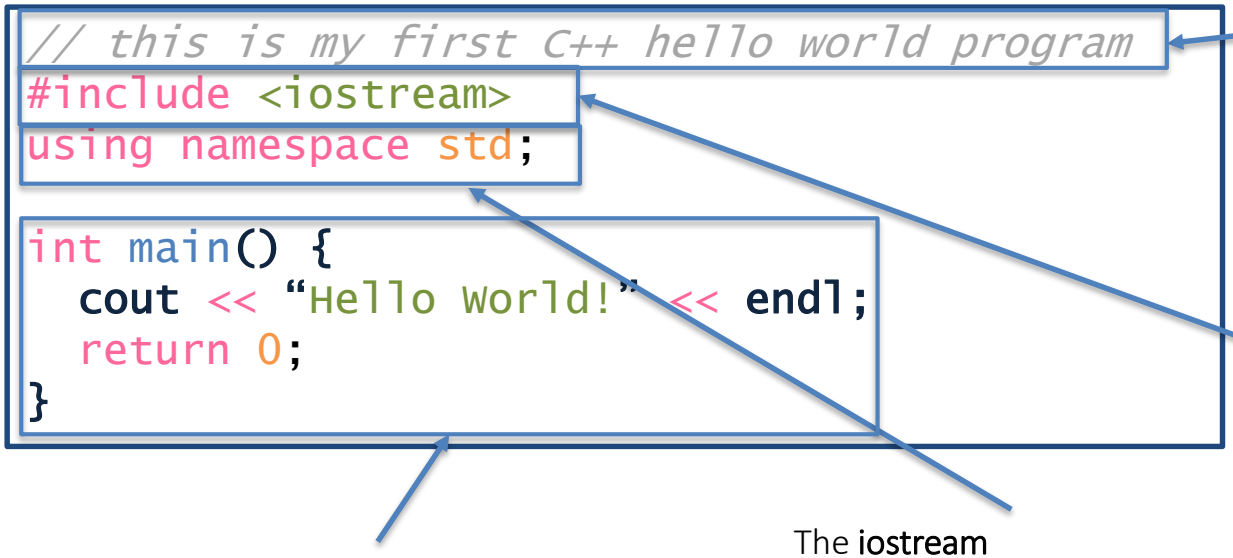
Code::Blocks IDE for Windows

Part II

BASIC OPERATIONS

The First C++ Program

The Hello World program gives the basic structure of a C++ program.



A line starting with // is called a comment line, any text after // till the end of line is ignored by the compiler.

This is the include directive which tells the compiler where to find information about certain routines used by the program; **iostream** is the name of a library that contains the declarations of the routines (**cout**/**endl**) that handle input from the keyboard and output to the screen;
Later, you may also use other libraries (e.g., the math library by #include <math>.

This is the main function which contains the main body of the C++ program. In this case, we have two statements "**cout**…" and "**return** .." in the main body. The main function is also the starting point of the program execution of all C++ program: the program is executed statement by statement starting from the first statement in this main function.

The **iostream** object/operation **cout** and **endl** are under the namespace **std**. If this line is removed, then you will need to write **std::cout** and **std::endl** without raising a compilation error.
(You can try and look for the error yourselves.)

The First C++ Program

```
// this is my first C++ hello world program
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

By looking at the output of this program, you probably can guess what this program does. How would you change the program so that it can output Hello ENGG1340!
on the screen?

The last statement return 0; in the main function indicates (to the operating system) that the program ended successfully. Note that on C++ compilers and more recent C compilers (C99 onwards), the compiler will add this statement for you if you omit it.

The First C++ Program

```
// this is my first C++ hello world program
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

cout is the standard output stream object defined in the iostream library. The standard output is the screen by default.

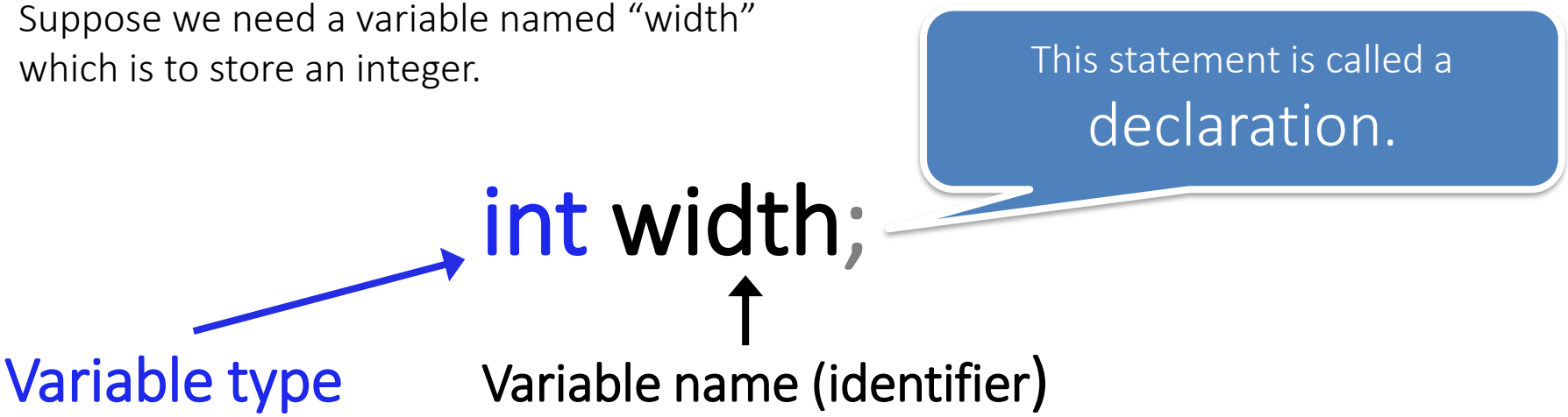
We will come back to the basic I/O afterwards.

www.cplusplus.com is a good place to look for the definition and usage of the C++ constructs and functions.

You are highly recommended to go through the related topics in their tutorial as well: http://www.cplusplus.com/doc/tutorial/

Variables

Let's start with how a variable can be defined in C/C++.
Suppose we need a variable named "width"
which is to store an integer.



- Used to store data.
- Data stored in a variable may change over time.
- When we declare a variable, the computer will assign an appropriate number of memory cells in the main memory to each variable according to the type of data to be stored.

Variables

What happens in the computer?

1

```
int main ( ) {
    int width;
    int height;
    …
    return 0;
}
```

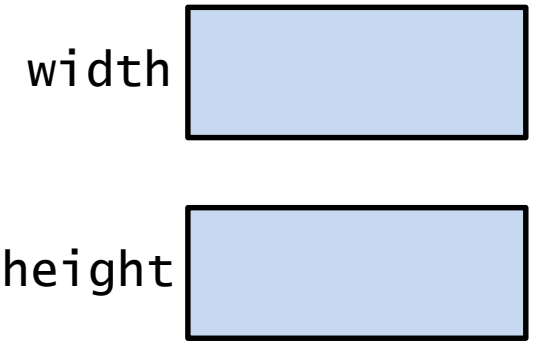
Main Memory

A memory chunk for storing an integer will be created in the main memory and associated with the name “width”

2

```
int main ( ) {
    int width;
    int height;
    …
    return 0;
}
```

Main Memory



Recall that the execution starts from the main() function

Variable Names (aka Identifiers)

- An identifier must start with either
  - a letter (i.e., A to Z, and a to z), or
  - the underscore symbol (i.e., _)
- The rest of the characters may be
  - letters (i.e., A to Z, and a to z),
  - digits (i.e., 0 to 9), or
  - the underscore symbol (i.e., _)
- Meaningful identifiers make a program more readable
- C++ is case-sensitive
  - e.g., radius, RADIUS, Radius, etc., are different
- Cannot be a **reserved keyword** in C++

```
int area;
int length;
int area = length*length;
```

vs.

```
int a;
int b;
int b= a*a;
```

C++ Reserved Keywords

- Reserved words in C++ with predefined meanings.
- CANNOT be used as names for variables or anything else.

asm	do	inline	return	typedef
auto	double	int	short	typeid
bool	dynamic_cast	log	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	extern	new	struct	virtual
class	false	operator	switch	void
const	float	private	template	volatile
const_cast	for	protected	this	wchar_t
continue	friend	public	throw	while
default	goto	register	true	
delete	if	reinterpret_cast	try	

You are not required to memorize all these names. You will get to recognize most of them later on.

Valid identifiers

- Which of the following identifiers are valid in C++?

`a_man` ✔	`2008` ✗	`program.cc` ✗
`const` ✗	`year1-student` ✗	`_oOOo_` ✔
`an integer` ✗	`change%2` ✗	`ABCx123` ✔
`string` ✔	`Days_of_Week` ✔	`friend` ✗
`cout` ✔	`delete` ✗	`cos` ✔

Words like **`cin`**, **`cout`**, **`string`**, and **`cos`** are NOT keywords in C++. They are defined in libraries required by the C++ language standard. Redefining these words, though allowed, can be confusing and thus should be avoided.

Data Type of a Variable

Data type is an important concept when using a variable. This concept is stricter in C++ than in Python.

- Tells the computer how to interpret the data stored in a variable
- Determines the size of storage needed to store the data
- Some basic data types in C++:

Name	Description	Size	Range
`char`	Character or small integer	1 byte	0 to 255
`bool`	Boolean value	1 byte	True(1) or False(0)
`int`	Integer	4 bytes	-2147483648 to 2147483648
`double`	Double precision floating point number	8 bytes	1.7e-308 to 1.7e+308 (~15 digits) -1.7e-308 to -1.7e+308 (~15 digits)

** The size and range of a particular data type depend on the system under which a program is compiled. The values shown above are those found on most 32-bit systems. Also **bool** occupies 1 byte because it is the smallest addressable size of CPU.

Declarations

- All variables must be declared before use.
- A declaration specifies a type, and contains a list of one or more variables of that type.

Syntax

```
type_name     variable_name;
type_name     variable_name_1, variable_name_2, …;
```

- Examples:

To declare two integer variables named “age” and “steps”

```
int age, steps;
char c;
bool win;
double height, width, length;
```

Assignment Statement

- A variable may be initialized or its value can be changed at a later time after its declaration using an assignment statement.
- An assignment statement consists of a variable on the left-hand side of an equal sign, and a value or an expression on the right-hand side.

Syntax

```
variable_name = expression;
```

Example

```
int age;
double heights;
age = 5;
heights = 8 * age + 20.5;
```

a constant value

an expression

Assigning Values to Variables

1

```
int main ( ) {
  int width, height, area;
  width =  5;
  height =  4;
  area = width * height;
  return 0;
}
```

Main Memory

width	5
height	4
area	20

2

```
int main ( ) {
  int width, height, area;
  width =  5;
  area = width * height;
  return 0;
}
```

The variable **height** is uninitialized before use and the result is unpredictable.

Main Memory

width	5
height	???
area	???

Initializations

- A variable that has not been given a value is said to be uninitialized, and will simply contain some "garbage value"
- Using uninitialized variables in computations will give unexpected results, and thus should be avoided
- A variable may be initialized in its declaration:

```
int age = 5, steps = age + 10;
char c = 'Y';
bool win = true;
double height = 120.5, length = 1.5e3;
```

A character constant is written as a character within **single quotes**.

Scientific notation
(floating point notation)
1.5e3 = 1.5 * 10^3 = 1500

Strings – The Very Basics

- Very often we need to work on textual information, and this can be done in C++ using strings (C has a different handling of strings and we will discuss that later).
- A string variable is just a variable containing a sequence of characters.
- Strings are not one of the fundamental C++ data types but are so frequently needed that they are defined as a class within the standard library.
- Include the <string> header when using strings in your program.

Sometimes you got no compilation error even if you don't include the <string> header; it's because it might be included in some standard libraries already, however, this depends on the implementation of the standard libraries and so it's always a good practice to include it when using strings.

Strings – The Very Basics

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

int main() {
  string greeting = “Hi”,  name = “ENGG1340”;
  cout << greeting << “ “ << name << endl;
  greeting = “Good morning”;
  cout << greeting << “ “ << name << endl;
  return 0;
}
```

Can you guess what the output is?

```
Hi ENGG1340
Good morning ENGG1340
```

We will come back to the more interesting operations of strings later.

Constants

- Constants are expressions with fixed values.
- Integers: `65` (decimal), `0101` (octal), `0x41` (hexadecimal)
- Floating point numbers: `3.14159`, `6e23`, `1.6e-19`, `3.0`
- Characters: `'A'`, `'z'`,
  `'\n'` (newline), `'\''` ('), `'\\'` (\), `'\?'` (?),
  `'\101'` ('A', octal ASCII code for 65),
  `'\x41'` ('A', hex ASCII code for 65)
- Strings: `"This is a string"`, `""` (empty string)
- Boolean: `true`, `false`

Note that a character is enclosed within the single quotes ' ' while a string is enclosed by the double quotes " ". We will come back to the differences between characters and strings in later modules. For now, you may think of a character as a single letter and a string as a sequence of letters.

Constants

- Sometimes we want to assign a fixed value to a variable

```
double PI = 3.14159265359;
```

- Add a constant **modifier** in front of a variable declaration
- The compiler will make sure that the variable remains a constant

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

int main() {
  const double PI = 3.14159265359;
  double r = 5.0, area = PI * r * r;

  PI = 3.14159;
}
```

gcc-make-run: Compile Error

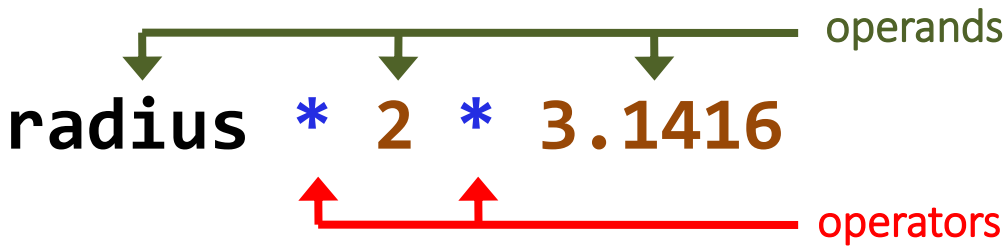
```
hello.cpp: In function 'int main()':
hello.cpp:8:8: error: assignment of read-only variable
'PI'
   PI = 3.14159;
        ^~~~~~~
```

Line 8 "PI = 3.14159;" generates a compile error since PI is declared as a constant variable in line 5, but here we attempt to change its value

You can see that this helps to ensure the value of a variable will not be changed accidentally.

Expressions

- Combine variables and constants to produce new values (i.e., to evaluate an expression)
- Composed of operators (instructions) and operands (data)



- Operators
  - Specify what is to be done on the operands
  - E.g., arithmetic operators, relational operators, logical operators
- Operands
  - Data on which the computation is performed
  - May be variables and/or constants

Operators

- Arithmetic operators (+, –, *, /, %)
- Relational operators (>, >=, <, <=, ==, !=)
- Logical operators (&&, ||, !)
- Increment and decrement operators (++, --)
- Assignment operators (=, +=, -=, *=, /=)

Arithmetic Operators

Arithmetic Operators	Sign in the expression
Addition	+
Subtraction	–
Multiplication	*
Division	/
Modulus	%

- The modulus operator % produces the remainder.
  - E.g., 13 % 3 results in 1

    because 13 = (3 * 4) + 1

Arithmetic Operators

- Note: when both operands of the / operator are of integer types, the / operator performs integer division which truncates any fractional part of the division result.

```
int a = 3, b = 2;
int c = a / b, d = 8 / 3;
cout << "The value of c is " << c << endl ;
cout << "The value of d is " << d << endl ;
```

What is the screen output?

```
The value of c is 1
The value of d is 2
```

- The operator % cannot be applied to double (i.e., floating point numbers).

Division by Zero

- If the divisor of the / operator is 0, a division by zero error will be generated during runtime.

```
twchim@academy11 module3> cat division_by_zero.cpp
#include <iostream>
using namespace std;

int main() {
        int x = 3, y = 0;
        cout << x/y << endl;
        return 0;
}
twchim@academy11 module3> g++ -pedantic-errors -std=c++11 division_by_zero.cpp -
o division_by_zero
twchim@academy11 module3> ./division_by_zero
Floating point exception (core dumped)
twchim@academy11 module3>
```

Note that no compilation error will be generated.

Precedence

- In evaluating an expression with mixed operators, those operators with a **higher priority** will be carried out before those with a **lower priority**.

1 **+** 2 ***** 3 Result: 9 or 7?

- The operator ***** has a higher precedence than the operator **+** (same as what you learned before ☺).
- The order of evaluation is equivalent to `1 + (2 * 3)`.

Precedence

- In evaluating an expression with mixed operators, those operators with a **higher priority** will be carried out before those with a **lower priority**.

12 – 11 **%** 3 Result: 1 or 10?

- The operator **%** has a higher precedence than the operator **–**.
- The order of evaluation is equivalent to 12 – (11 % 3).

Precedence & Associativity

Operator types	Operators	Associativity
unary	`+, -, ++, --, !`	-
binary arithmetic	`*, /, %`	left to right
binary arithmetic	`+, -`	left to right
relational	`<, <=, >, >=`	left to right
relational	`==, !=`	left to right
logical	`&&`	left to right
logical	`||`	left to right
assignment	`=, +=, -=, *=, /=, %=`	right to left

High precedence

Lower precedence

- The order of evaluation may be overridden by inserting parentheses **()** into the expressions
  - e.g., (1 **+** 2) ***** 3 = 9

Arithmetic Operator for Characters

We may perform arithmetic operation with characters. In this case, the numerical representation as in the ASCII code for each character will be used in the calculation. The following program also shows a common technique in converting a letter from upper case to lower case.

```
#include <iostream>
using namespace std;
int main()
{
      char c = 'Y';
      // convert a letter from upper case to lower case
      c = c + ('a' - 'A');
      cout << "1: " << c << endl;

      // advance to the next character
      c = c + 1;
      cout << "2: " << c << endl;
      return 0;
}
```

```
1: y
2: z
```

Screen output

Can you convert a letter from lower case to upper case then?

The ASCII table: www.asciitable.com

Relational Operators

Relational Operators	Sign in the expression
Greater than	>
Greater than or equal	>=
Smaller than	<
Smaller than or equal	<=
Equal	==
Not equal	!=

same higher precedence

lower precedence

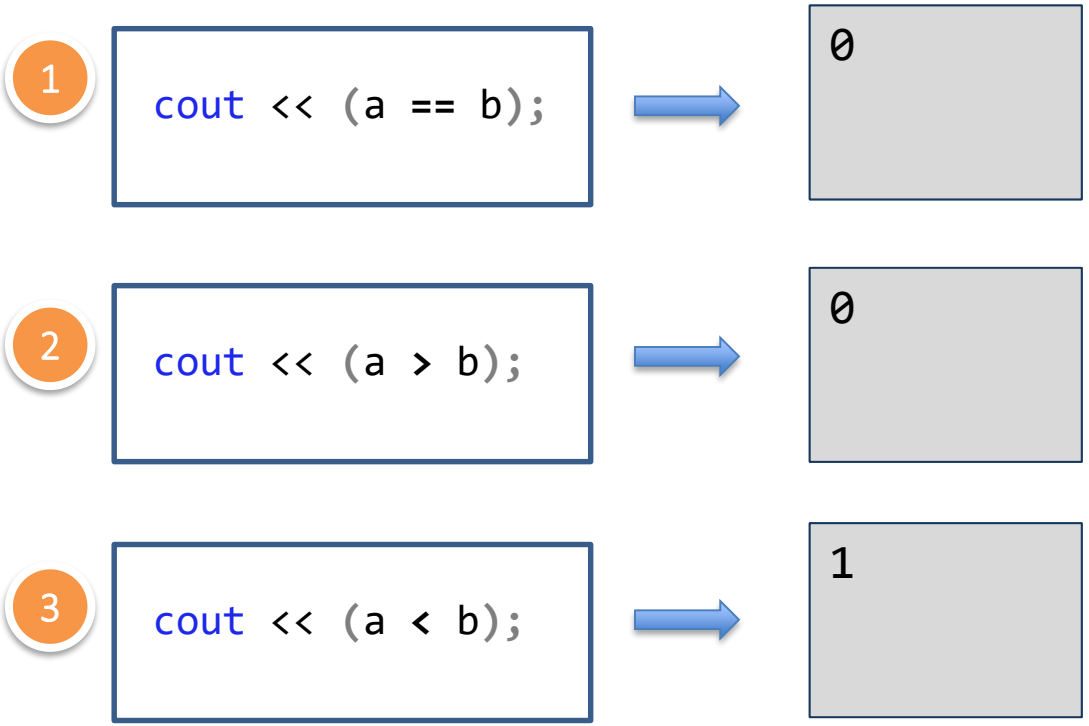
- For comparing the operands.

Relational Operators

- In C/C++, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

Suppose all 3 examples start with:

```
int a = 1, b = 2;
```



Relational Operators

4

```
int i = 1, lim = 2;
cout << (i < lim - 2) ;
```

```
0
```

The "–" operator is of **higher** precedence than the "<" operator, so "`lim - 2`" is executed first

5

```
int i = 1, lim = 2;
cout << ( (i < lim) - 2 );
```

```
-1
```

The bracket `()` **overrides** precedence and associativity,
hence `(i < lim)` is first evaluated to yield the intermediate result **1**

Logical Operators

Recall truth tables for computer logics.

Operands		AND (&&)	OR (||)	NOT (!)
A	B	A && B	A || B	! A
0	0	0	0	1
0	1	0	1	
1	0	0	1	0
1	1	1	1	

0: False
1: True

- Precedence: (High) ! > && > || (Low)
- C++ treats any non-zero value as true, and zero as false
  - Hence (3 && 0) is false, and (-5 || 0) is true
- The unary negation operator ! converts a non-zero operand into 0, and a zero operand into 1 (e.g., ! 3 is evaluated to 0)

Logical Operators

1

```
int x = 5;
bool in_range = ! (x < 0 || x > 50);
cout << in_range << endl;
```

```
1
```

Both expressions connected by || evaluate to a `false` value (`0`)

2

```
bool  i_am_cool =
      (gals != 0) && ((gifts / gals) >= 2);
```

What if **gals** is **0**? Will **gifts/gals** generate a runtime error?

No! Because: C/C++ evaluates a logical expression from left to right, and stops evaluating once the truth or falsehood of the result is known. (a.k.a. short-circuit evaluation)

Hence, if `gals` is `0`, the expression ((gals != 0) && ???) must be false anyway, so the expression `(gifts / gals) >=2` will NOT be evaluated, and thus not generating a runtime error.

There is similar short-circuit evaluation for the || operator:

```
bool omg = (gals == 0) || ((gifts / gals) < 2);
```

Increment & Decrement Operators

Assignment Operators	Sign in the expression
Increment	++
Decrement	--

- The increment operator ++ adds 1 to its operand.

```
int i = 0;
++i ;
```

is equivalent to

```
int i = 0;
i = i + 1;
```

- The decrement operator -- subtracts 1 from its operand.

```
int i = 0;
--i ;
```

is equivalent to

```
int i = 0;
i = i - 1;
```

Increment & Decrement Operators

- The operators **++** and **--** may be used either as prefix (e.g., **++i**) or postfix (e.g., **i++**) operators.
  - When used as **prefix**, increment/decrement is done **before** the value is used.

```
int c = 0, i = 0;
c = ++i ;
```

is equivalent to

```
int c = 0, i = 0;
i = i + 1;
c = i;
```

c = ???
i = ???

  - When used as **postfix**, increment/decrement is done **after** the value is used.

```
int c = 0, i = 0;
c = i++;
```

is equivalent to

```
int c = 0, i = 0;
c = i;
i = i + 1;
```

c = ???
i = ???

Assignment Operators

- Expression such as `i=i+2` in which the variable on the left-hand side is repeated immediately on the right can be written in the compressed form `i+=2`
- Most binary operators have a corresponding compound assignment operator, e.g., `-=`, `*=`, `/=`, and `%=`

Examples

```
x *= y + 1;
```
is equivalent to
```
x = x * (y + 1);
```

```
x %= y % 3;
```
is equivalent to
```
x = x % (y % 3);
```

Type Conversions

- When an operator has operands of different types, they are converted to a **common type** according to a small number of rules.

1. "lower" type promoted to "higher" type

```
3.0 / 2;
```

2 (int) is promoted to 2.0 (double), and the result is 1.5

Important: Compare this with

```
3 / 2;
```

No type conversion because both 3 and 2 are integers, therefore integer division is carried out, and the result is 1

Type Conversions

2. In assignment statements, the value of the right side is converted to the type of the left

```
double x = 5;
```

x stores the value 5.0

```
int x = 2.8;
```

Converting a double value to an int value causes truncation of any fractional part
x stores the value 2

* The compiler may issue a warning as there is inform

```
int x = (int) 2.8;
```

Explicit type casting tells the compiler it is an intended type conversion and prevents the compiler from producing a warning.
x stores the value 2

* The compiler generates no warning

This also shows that you, as the programmer, can control how values are stored.

Type Conversions

3 Type conversions that don't make sense are not allowed.

e.g., assigning a string literal to an int variable generates a compilation error:

```
int main() {
  int x = "abc";
}
```

gcc-make-run: Compile Error

```
hello.cpp: In function 'int main()':
hello.cpp:3:11: error: invalid conversion from 'const
char*' to 'int' [-fpermissive]
   int x = "abc";
           ^~~~~
```

Basic I/O (Input/Output)

- A stream is an object where a program can either insert or extract characters to/from it.
- We may use streams to perform input and output operations in sequential media such as the screen or the keyboard.
- The standard **C++ library** includes the header file iostream where the standard input and output stream objects are declared.
- We need to include the header file by the #include directives before using any objects and functions in the iostream library.

Include the **iostream** library to use **cin** and **cout**. The **iostream** library is some existing object codes developed by others. As this is so useful, it is regarded as standard C++ library.

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

int main () {
   …
}
```

Basic I/O

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

int main () {
    cout << “Hello!” << endl;
}
```

This statement is important!
Because **cout** and **endl** are provided under the namespace (i.e., a container of names) **std**.
Their names are indeed **std::cout** and **std::endl**.

```
#include <iostream>

int main () {
    cout << “Hello!” << endl;
}
```

```
a.cpp: In function int main():
a.cpp:4: error: 'cout' was not declared in this scope
a.cpp:4: error: 'endl' was not declared in this scope
```

Compiler error

```
#include <iostream>

int main () {
    std::cout << “Hello!”
        << std:: endl;
}
```

Standard Output

- By default, the standard output of a program is the screen, and the C++ stream object defined to access it is **cout**.
- The insertion operator << is used to insert data into the stream, which may be used more than once in a single statement.

```
int a = 1 , b= 2, c = 3;
cout << "Hello ";
cout << "World!" << endl;
cout << 1 << a << endl;
cout << "b = " << b << " and c = "
   << c << endl;
```

```
Hello World!
11
b = 2 and c = 3
```

Screen output

Note that there is no line break after "Hello " and "World!"

Also there is no space between 1 and the value of a in the 2nd output line.

Standard Output

- There are some escape sequences that have special usage in the output.

\a	alert (bell) character	\v	vertical tab
\b	backspace	\\	backslash
\n	newline	\?	question mark
\r	carriage return	\’	single quote
\t	horizontal tab	\”	double quote

```
cout << a << endl;
cout << “Hi!” << endl;
```

is equivalent to

```
cout << a << ‘\n’;
cout << “Hi!\n”;
```

Try out these escape sequences in a program and see the result!

Standard Input

- From time to time, we need to obtain user input to our program.
- The standard input device is usually the keyboard, and the C++ stream object defined to access it is **`cin`**.
- The extraction operator **`>>`** is used to extract data from the stream
- The type of the variable will determine the type of data that is extracted from the stream.

```
int x;
cin >> x;
```

An integer is expected to be input

vs.

```
char x;
cin >> x;
```

A character is expected to be input

- Note that `cin` can only process the input from the keyboard once the RETURN key has been pressed.

A Sample Program on I/O

Be careful about the directions of the << and >> operators!

```
#include <iostream>
using namespace std;
int main(){
  int age;
  double height, weight;
  cout << "Please input your age, height and weight: ";
  cin >> age >> height >> weight;
  cout << endl << "Your age is " << age << endl;
  cout << "Your height is " << height << endl;
  cout << "Your weight is " << weight << endl;
  return 0;
}
```

Screen output

```
Please input your age, height and weight:
```

A Sample Program on I/O

```
#include <iostream>
using namespace std;
int main(){
  int age;
  double height, weight;
  cout << "Please input your age, height and weight: ";
  cin >> age >> height >> weight;
  cout << endl << "Your age is " << age << endl;
  cout << "Your height is " << height << endl;
  cout << "Your weight is " << weight << endl;
  return 0;
}
```

user input from keyboard

Screen output

```
Please input your age, height and weight: 20 175.5 132
```

A Sample Program on I/O

```
#include <iostream>
using namespace std;
int main(){
  int age;
  double height, weight;
  cout << "Please input your age, height and weight: ";
  cin >> age >> height >> weight;
  cout << endl << "Your age is " << age << endl;
  cout << "Your height is " << height << endl;
  cout << "Your weight is " << weight << endl;
  return 0;
}
```

Screen output

```
Please input your age, height and weight: 20 175.5 132

Your age is 20
Your height is 175.5
Your weight is 132
```

Using File Redirection as Standard Input to Your Program

- Sometimes it is just too tiring to enter the input values to your program again and again, especially during the testing and debugging stages. In this case, you may execute your program using command line and file redirection so that the contents of a file will be fed into your program as if they are from the standard input (i.e., by default the keyboard)

```
twchim@academy11 module3> g++ -pedantic-errors -std=c++11 simple_input.cpp -o si
mple_input
twchim@academy11 module3> ./simple_input
Please input your age, height and weight: 19 168 55

Your age is 19
Your height is 168
Your weight is 55
twchim@academy11 module3>
```

User input from keyboard

```
twchim@academy11 module3> cat info.txt
20 170 58
twchim@academy11 module3> ./simple_input < info.txt
Please input your age, height and weight:
Your age is 20
Your height is 170
Your weight is 58
twchim@academy11 module3>
```

User input stored in a file “info.txt” and use file redirection to fed file contents to the program as input.

Part III

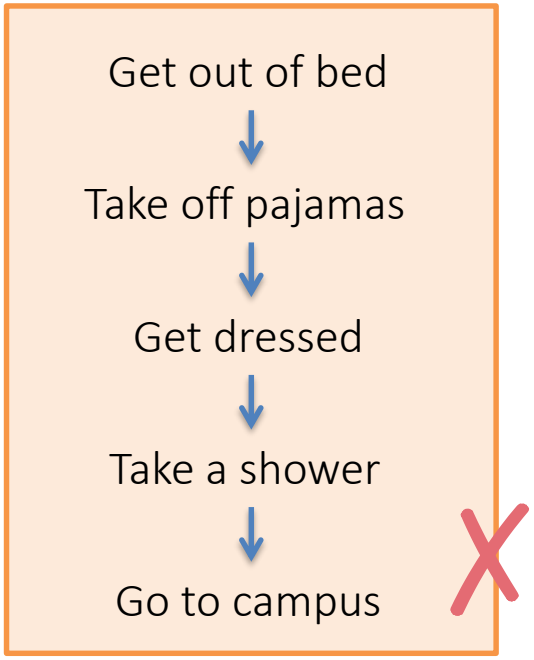
FLOW OF CONTROL

What we are going to learn?

- Making decisions in your program (branching)
  - the if selection statement
  - the if…else double selection statement
  - the switch multiple-selection statement
- Doing something repeatedly (looping)
  - while loop
  - for loop
  - break and continue in loops

Algorithms

- An algorithm is a procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which the actions execute (**flow of control**)



Flow of control is important. The **correctness** of your algorithm determines whether you can get the desired result.

Reference Only

Pseudocode

- "fake" code — An artificial and informal language similar to everyday English for developing an algorithm
- Helps you think out a program without worrying the syntax of a programming language.

Problem: Adding two input integers

Pseudocode

Prompt the user to enter the 1st integer
Input the 1st integer

Prompt the user to enter the 2nd integer
Input the 2nd integer

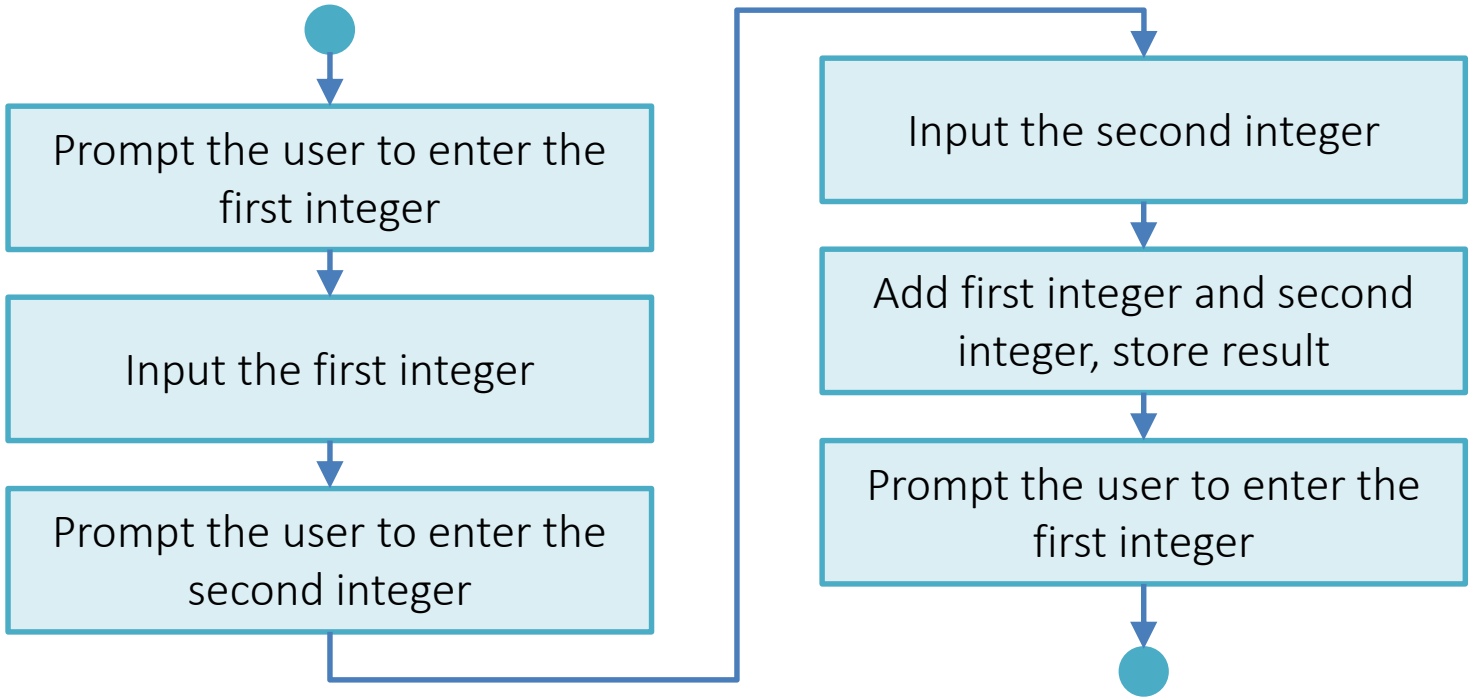
Add 1st integer and 2nd integer, store result
Display result

A C++ Program

```
cout << "Please input the 1st integer:";
cin >> x;

cout << "Please input the 2nd integer:";
cin >> y;

int res = x + y;
cout << res << endl;
```

Flowchart

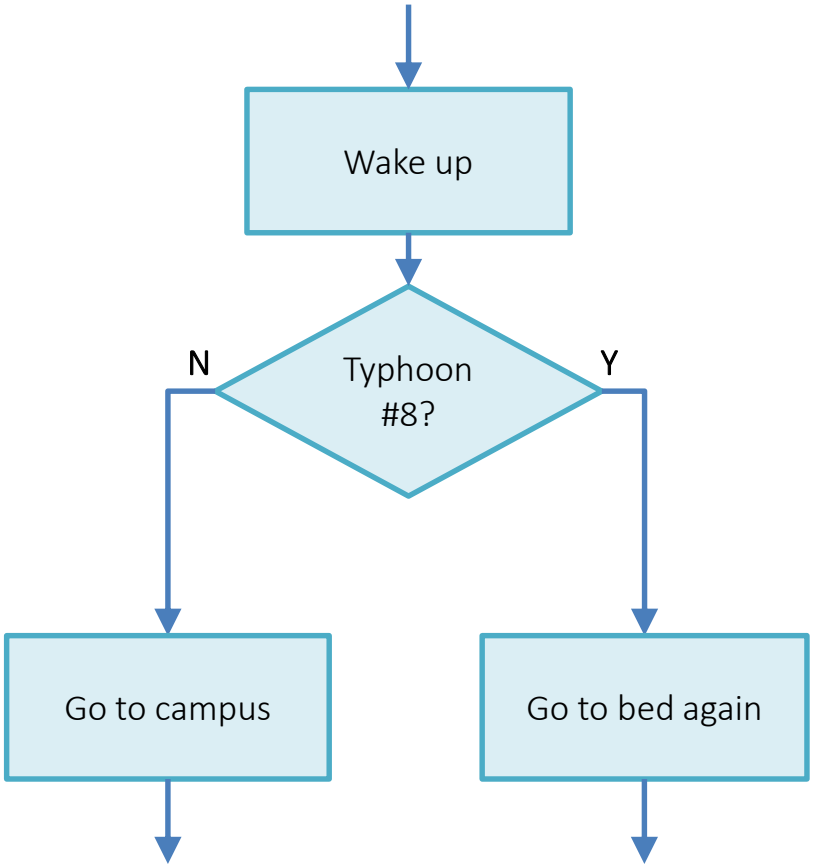
- A diagram to illustrate program flow (program logic).
- Used in analyzing, designing, documenting or managing a program.



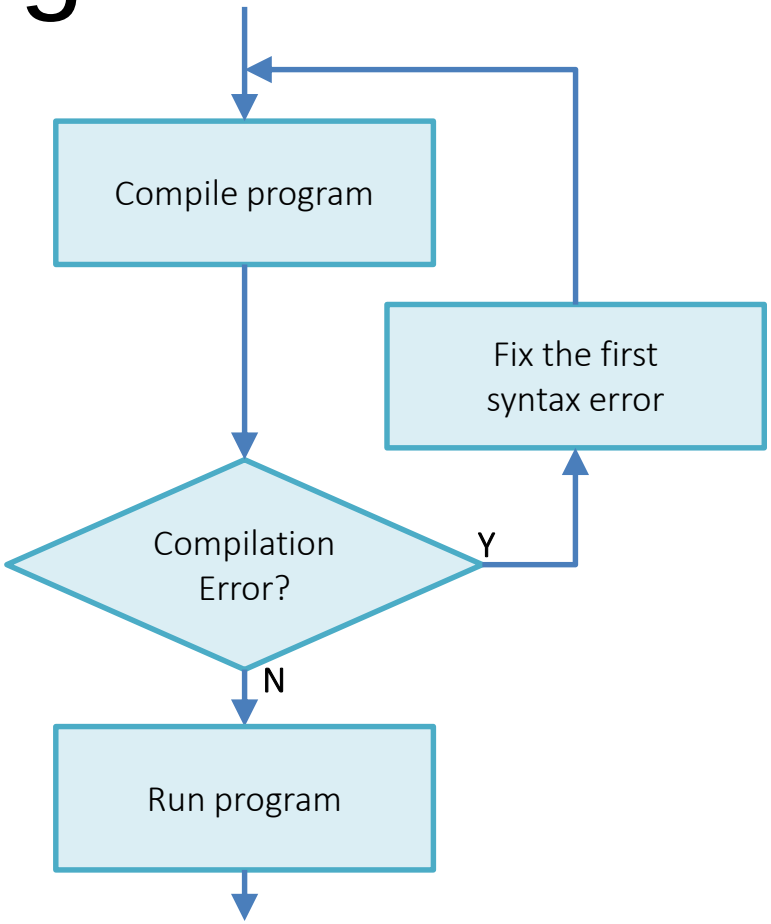
Flow of Control

- Recall that statements in the main function are executed sequentially.
- In more complex programs, however, it is often necessary to alter the order in which statements are executed, e.g.,
  - Choosing between two alternative actions – branching
  - Repeating an action a number of times – looping
- The order in which statements are executed is often referred to as flow of control.

Branching & Looping



Branching



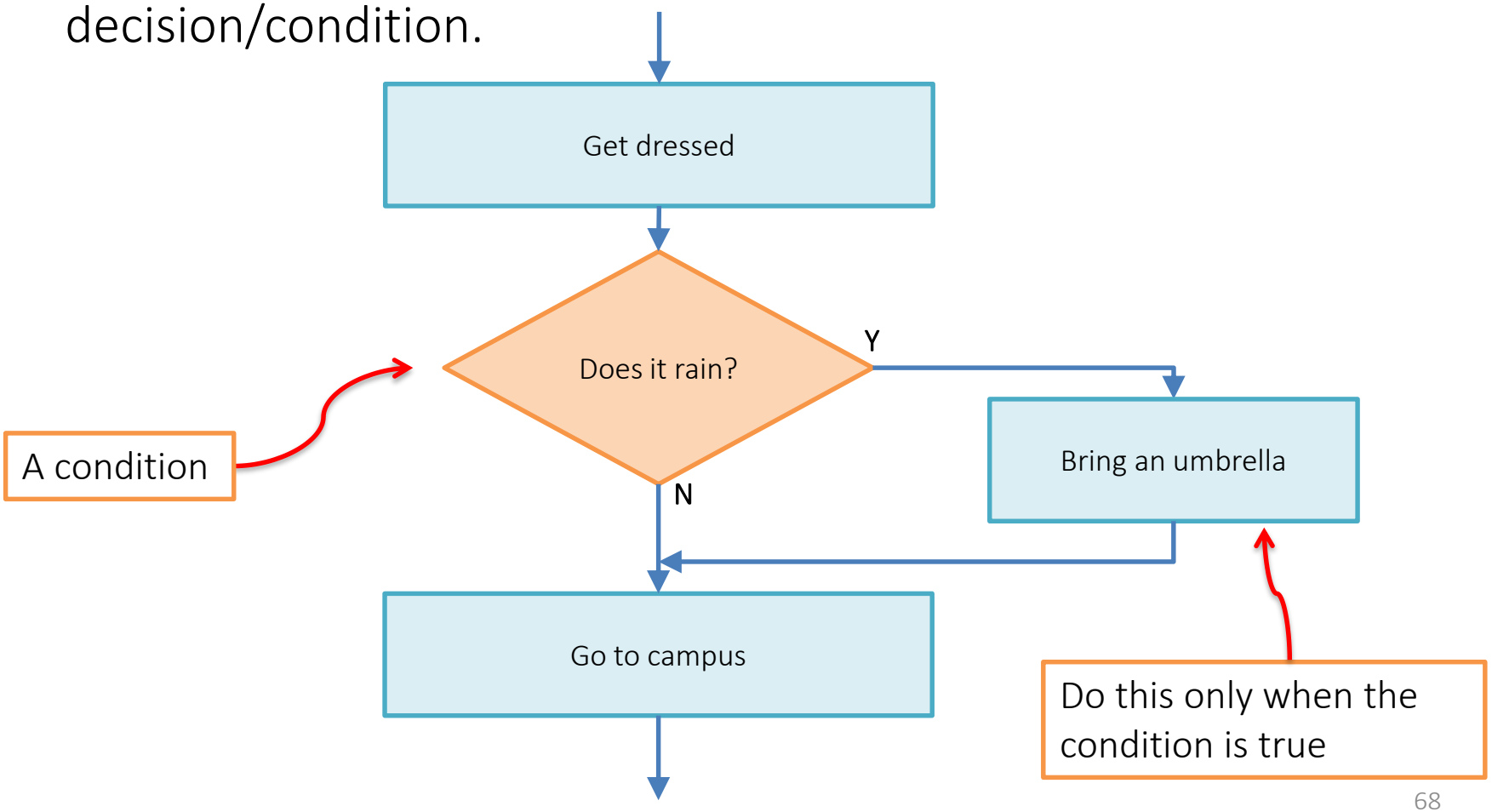
Looping

Making a decision

BRANCHING

Branching – Making a Decision

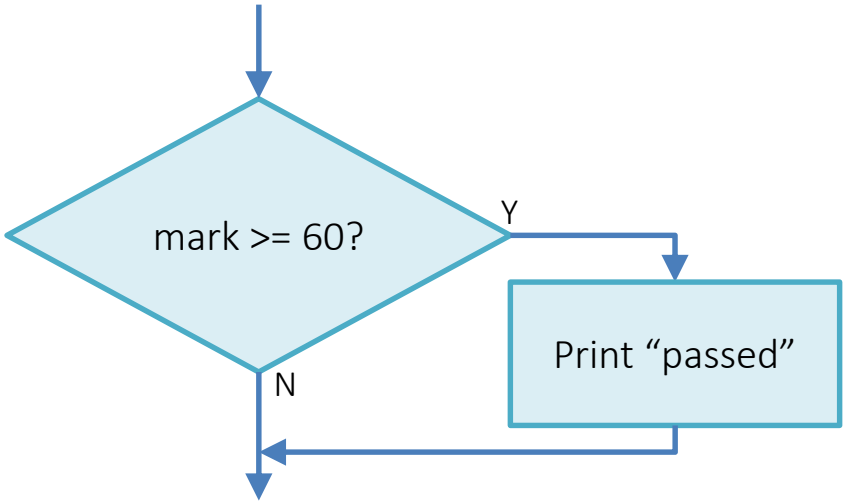
- Sometimes an action is taken **selectively** based on a decision/condition.



The **if** statement

Pseudocode

If student's mark is greater than or equal to 60
    print "passed"



Flowchart

C++ code

```
if (mark >= 60)
  cout << "passed";
```

The **if** statement

Syntax

```
if (condition)   statement;
```

- condition: an expression that evaluates to true or false

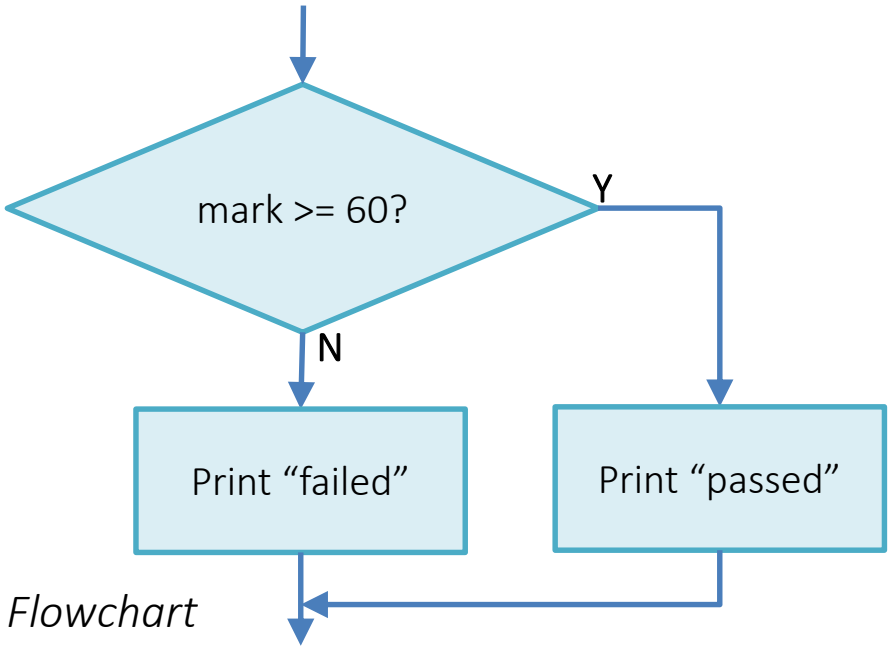
mark > 60	'A' == 'a'	3 – 2 != 0	3 – 2

- statement: a statement to execute if condition is true

The **if…else** statement

Pseudocode

```
If student's mark is greater than or equal to 60
      print "passed"
Else
      print "failed"
```



Flowchart

C++ code

```
if (mark >= 60)
  cout << "passed";
else
  cout << "failed";
```

The **if…else** statement

Syntax

```
if (condition)
    statement1;
else
    statement2;
```

- condition: an expression that evaluates to true or false
- statement1 is executed if condition is true; and if condition is false, statement2 is executed.

Example 1

- Write a program that reads 2 input integers and outputs the bigger one.

```
#include <iostream>
using namespace std;
int main() {









  return 0;
}
```

How would you solve the problem?

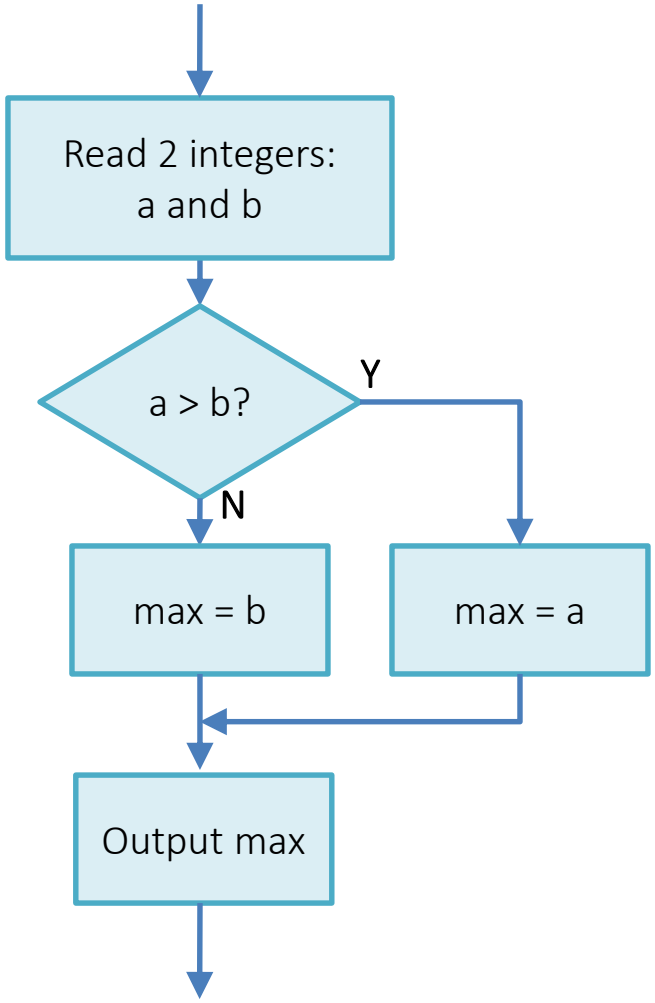
First step: devise a logical flow (i.e., the algorithm) for the solution.

Example 1

- Write a program that reads 2 input integers and outputs the bigger one.

```
#include <iostream>
using namespace std;
int main() {









  return 0;
}
```

Always start with this template for writing a program with standard I/O



Example 1

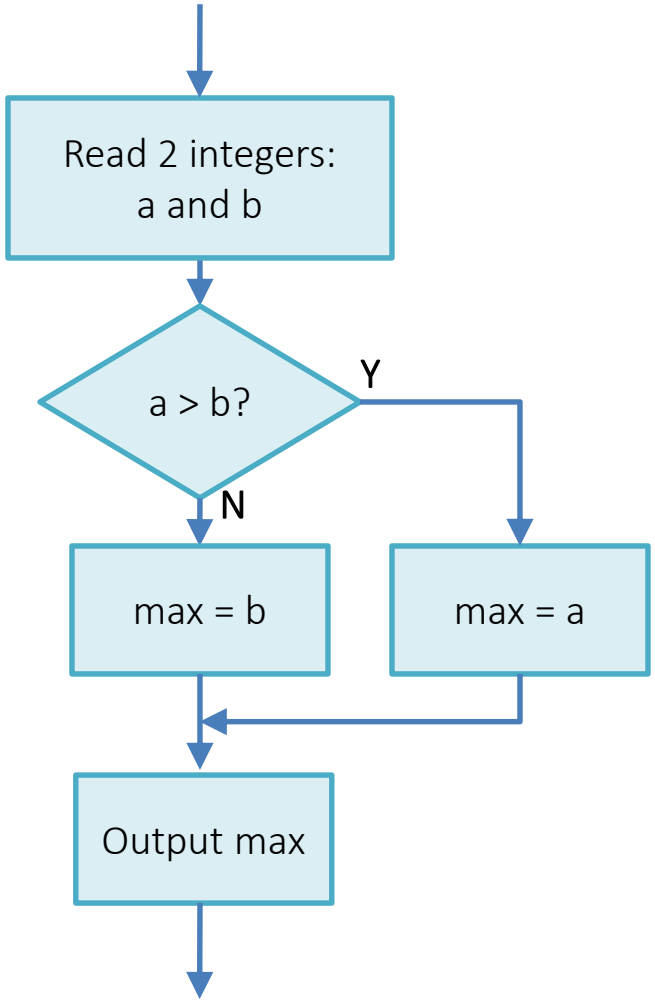
- Write a program that reads 2 input integers and outputs the bigger one.

```
#include <iostream>
using namespace std;
int main() {
```

Now think about it:
How many variables do you need?
What are their data types?

Remember to declare and initialize the variables before using them.

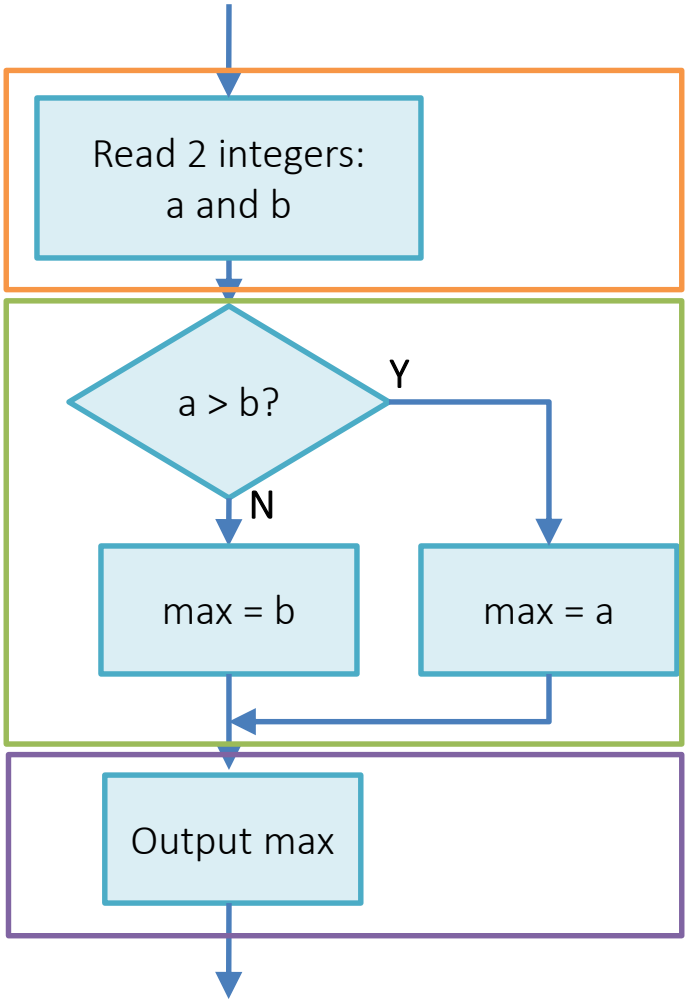
```
  return 0;
}
```



Example 1

- Write a program that reads 2 input integers and outputs the bigger one.

```
#include <iostream>
using namespace std;
int main() {
  int a, b, max;
  cin >> a >> b;

  if (a > b)
     max = a;
  else
     max = b;

  cout << max;

  return 0;
}
```

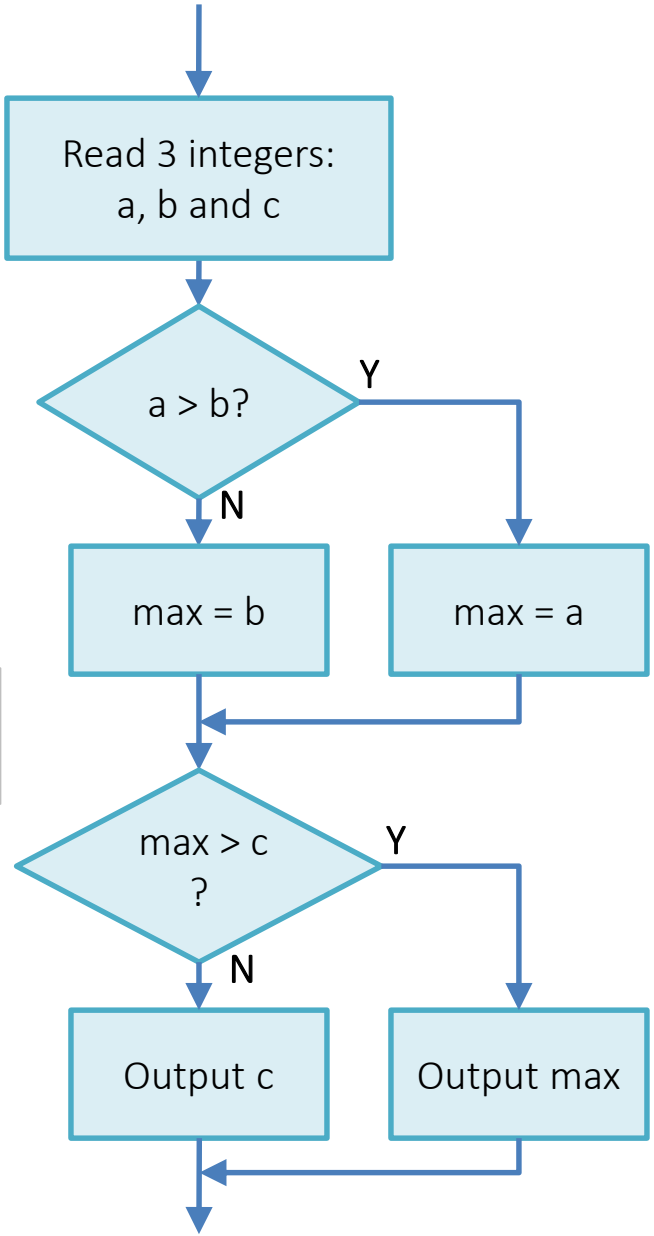


Example 2

- Write a program that reads 3 input integers and outputs the maximum one.

```
#include <iostream>
using namespace std;
int main() {




  return 0;
}
```

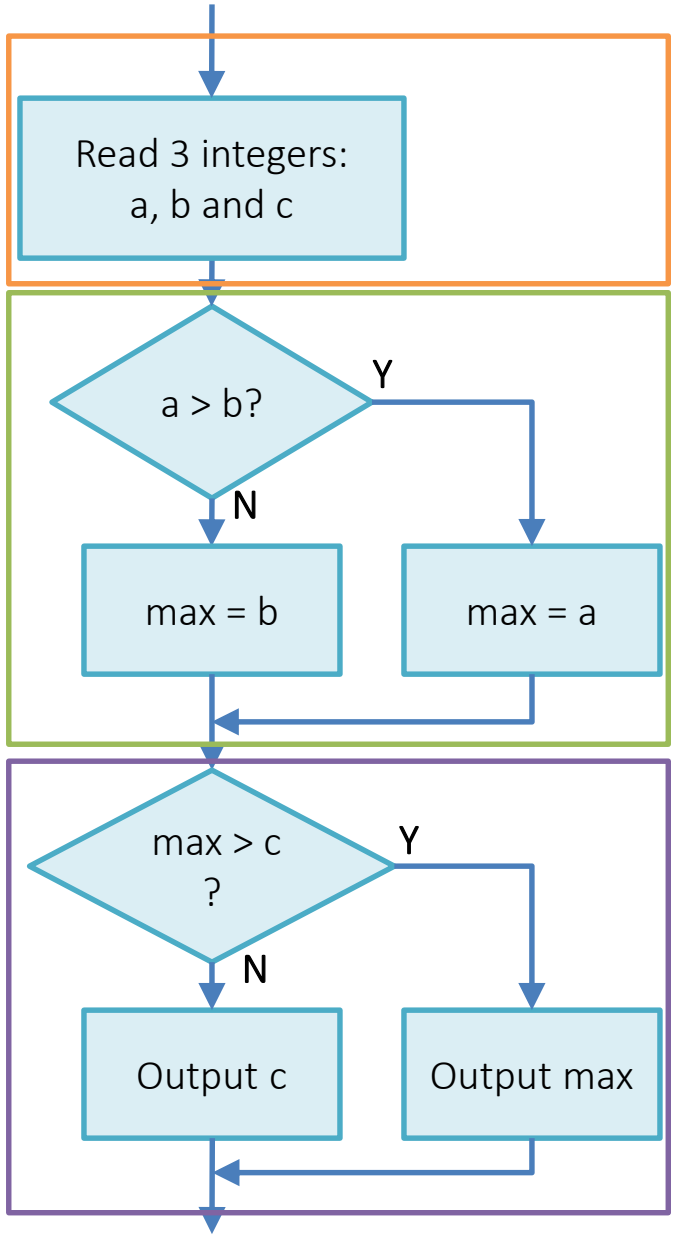
Let's first come up with an algorithm to solve the problem.



Example 2

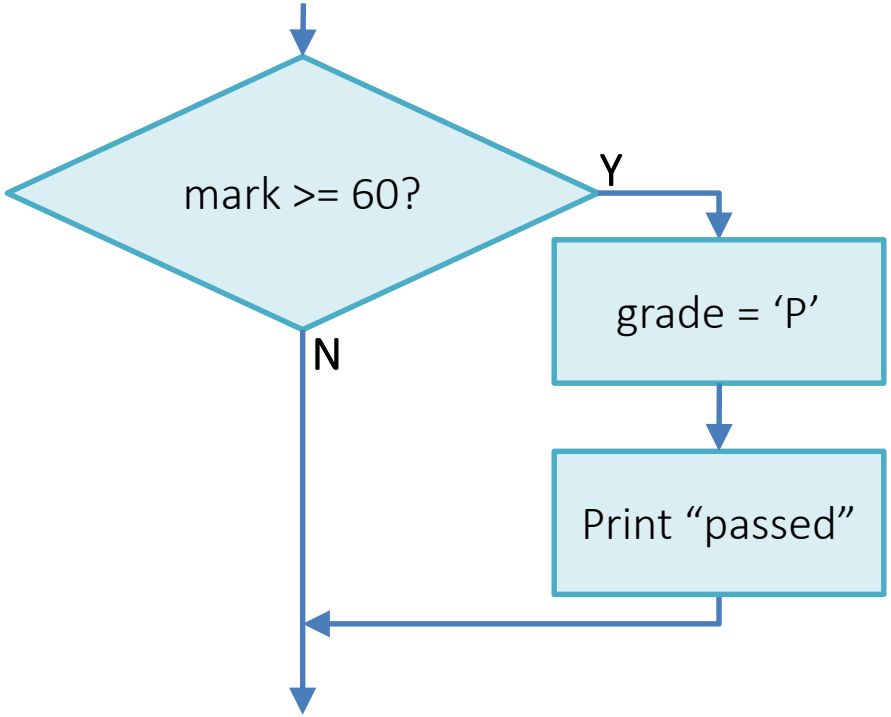
- Write a program that reads 3 input integers and outputs the maximum one.

```
#include <iostream>
using namespace std;
int main() {
  int a, b, c, max;
  cin >> a >> b >> c;
  if (a > b)
    max = a;
  else
    max = b;
  if (max > c)
    cout << max << endl;
  else
    cout << c << endl;
  return 0;
}
```



Compound Statements

- What if an action involves more than one statement?



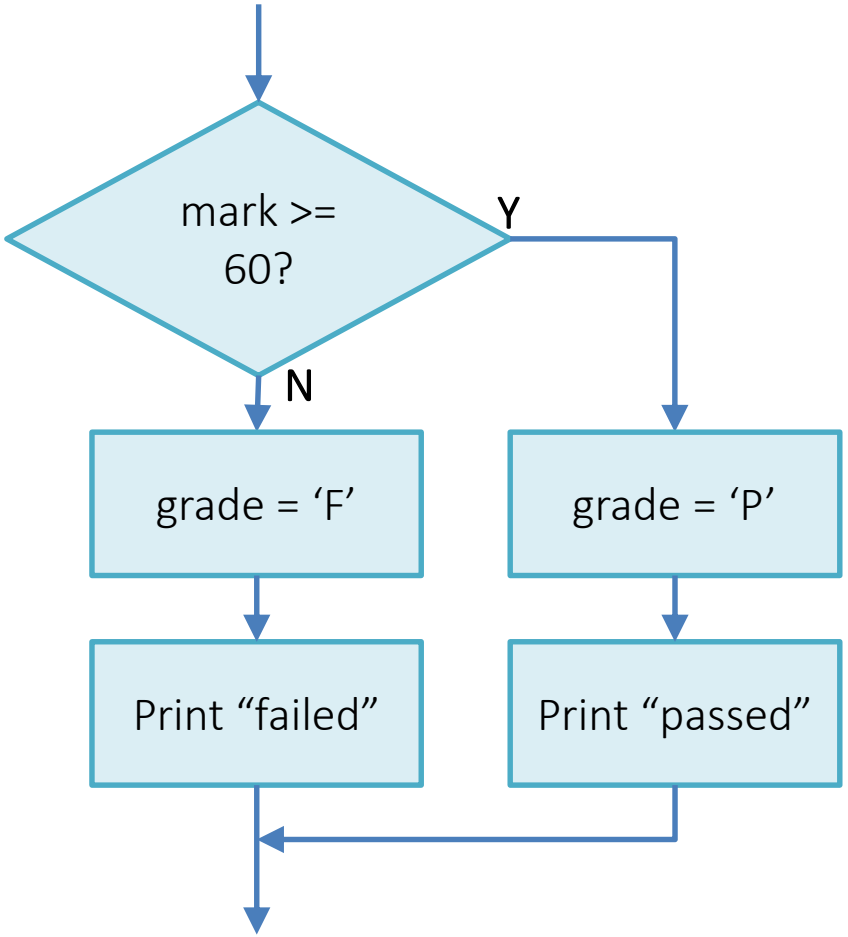
Syntax

```
if (condition)   statement;
```

a statement can also be a **compound statement** or a **block of statements** enclosed in { and }

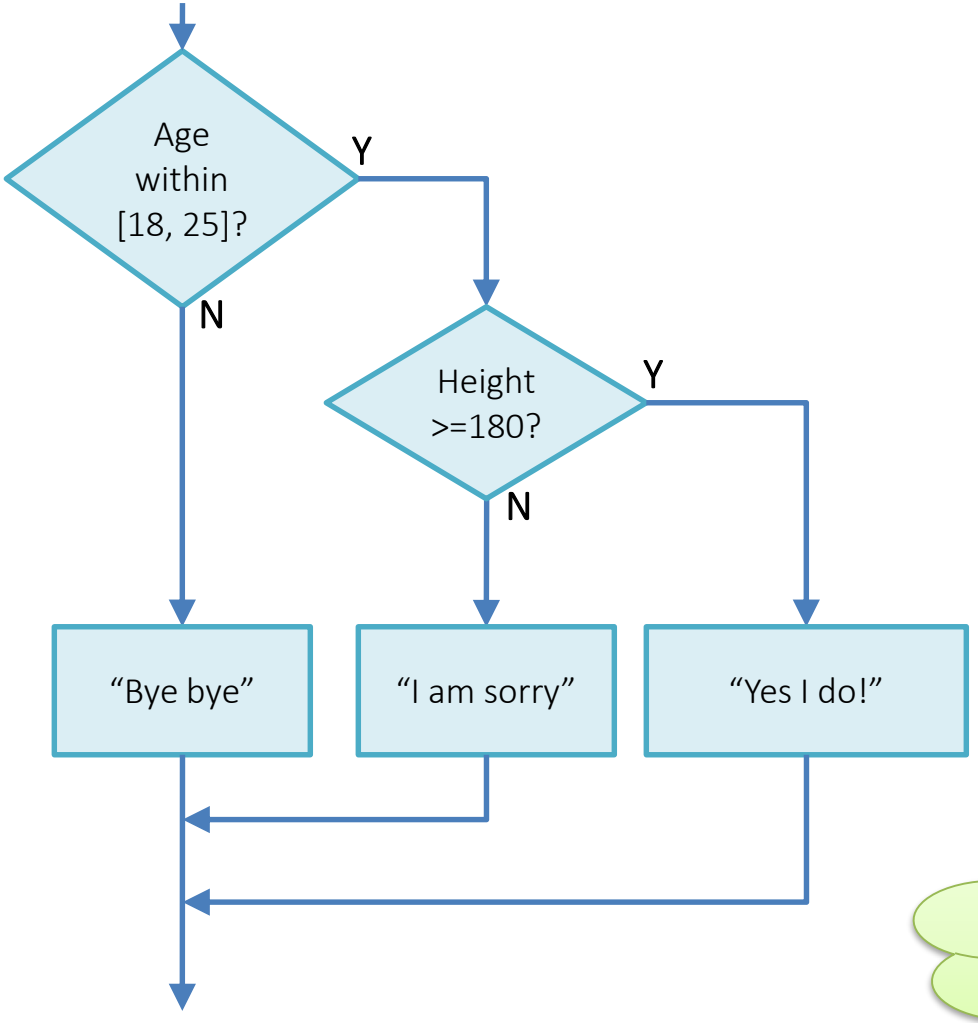
```
if (mark >= 60) {
  grade = 'P';
  cout << "passed";
}
```

Compound Statements




```
if (mark >= 60) {
  grade = 'P';
  cout << "passed";
}
else {
  grade = 'F';
  cout << "failed";
}
```

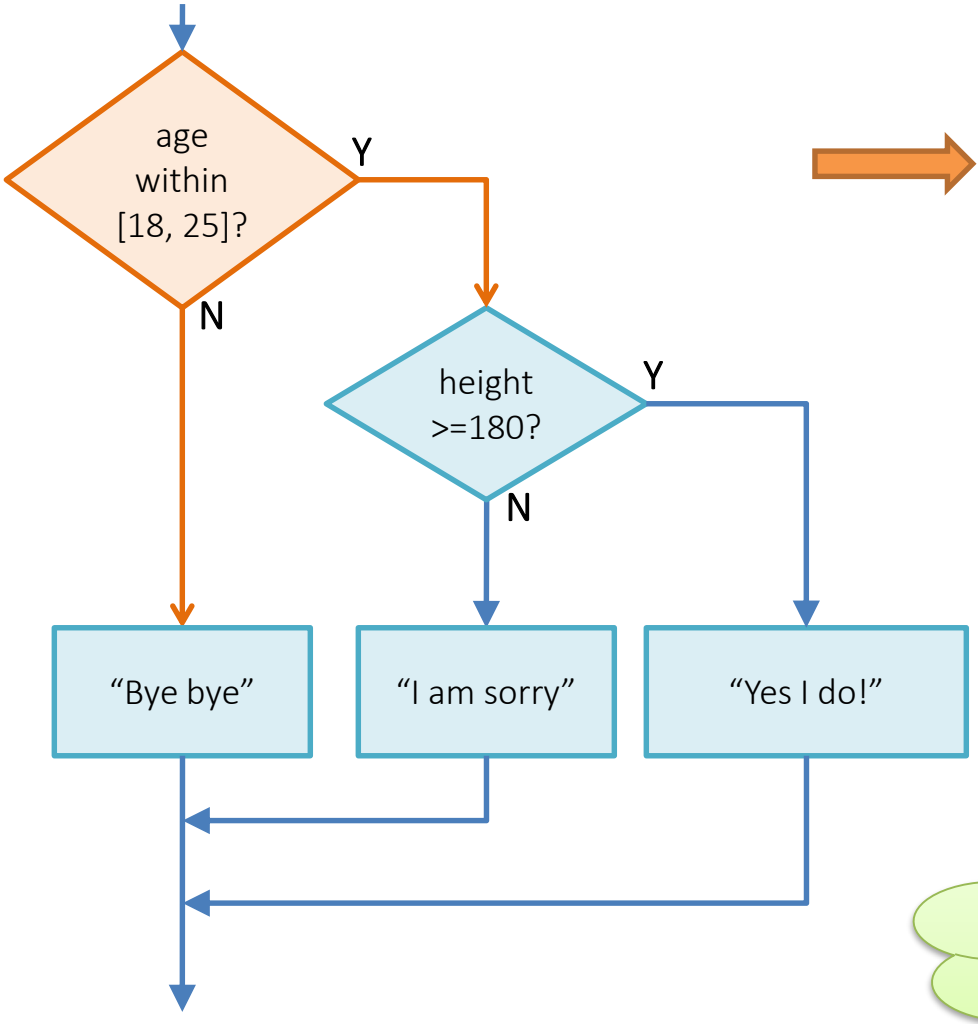
Nested **if…else** Statements



- An **if-else** statement can be nested within another **if-else** statement

My Mr. Right…
1. 18 to 25 years old, **AND**
2. Height: 180 cm or above

Nested **if…else** Statements

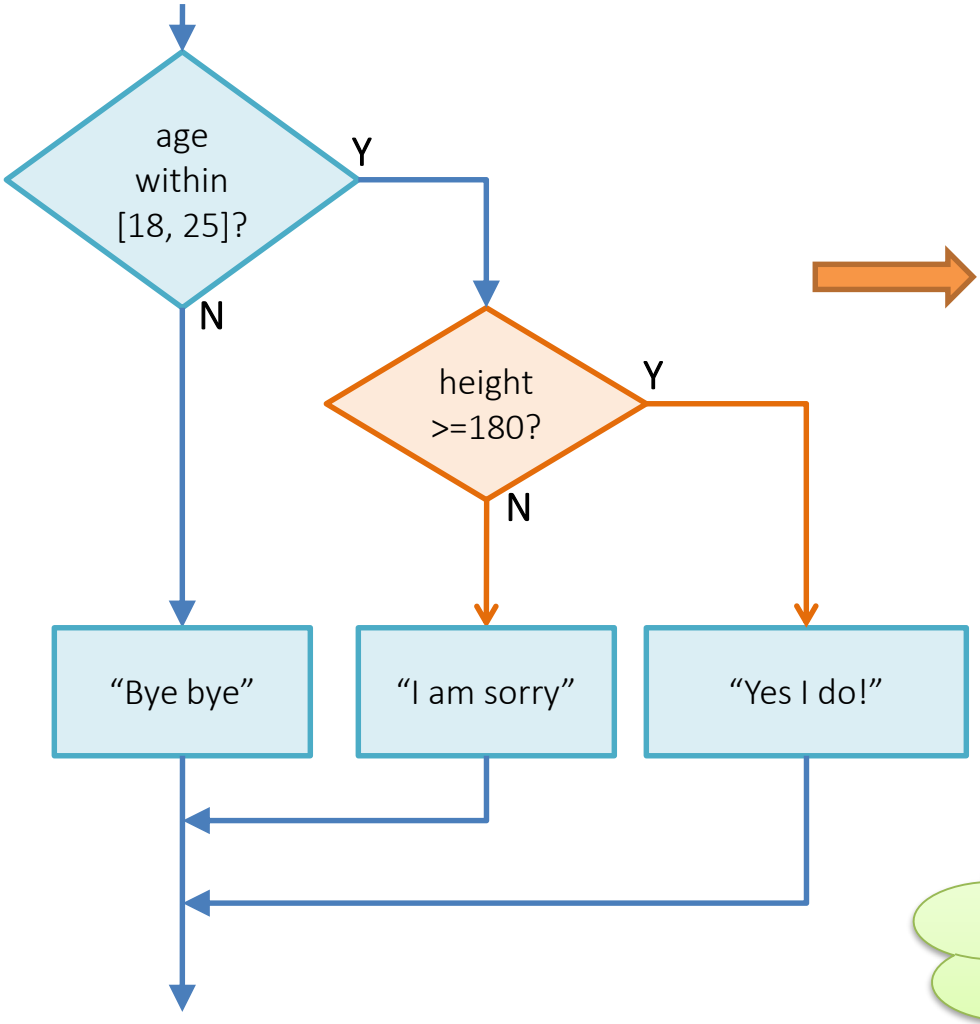



```
…
if (age >= 18 && age <=25)
{
   // the Yes part to
   // be dealt with here



}
else {
     cout << "Bye bye.";
}
…
```

My Mr. Right…

1. 18 to 25 years old, **AND**
2. height: 180 cm or above

Nested **if…else** Statements




```
…
if (age >= 18 && age <=25)
{
  if (height >= 180)
    cout << "Yes I do!";
  else
    cout << "I am sorry.";
}
else {
    cout << "Bye bye.";
}
…
```

My Mr. Right…

1. 18 to 25 years old, **AND**
2. height: 180 cm or above

A Note on Block Statement in C/C++ vs. Python

- How do you specify a block statement in Python?

```
if mark >= 60:
  grade = "P"
  print("passed")
else:
  grade = "P"
  print("failed")
```

Python: Indentation

```
if (mark >= 60) {
  grade = 'P';
  cout << "passed";
}
else {
  grade = 'P';
  cout << "failed";
}
```

C/C++: { }

- In C/C++ , we use {} to specify a block instead
- Indeed, the C/C++ compiler does not care about indentation which means that you can write in as few lines as possible, as long as the compiler can parse it.

Try this: a C++ program in 2 lines

```
#include<iostream>
int main() {if (2>3) std::cout << "no way"; else std::cout << "Why
do you desperately need to make life harder?\n"; return 0;}
```

Coding Hints

- Visualize the logic of the program before writing the code.
- When writing the code, follow the logic in the diagram, implement the processes in the diagram one at a time.
- Use proper indentation (spacing) to make your program more human readable (even when C/C++ does not require this). Always remember that you or others will need to maintain your codes later.

Dangling-Else Problem

Unlike Python, indentation does NOT determine blocks of statements in C/C++!

The following program segments are treated the same by the C/C++ compiler, although they have different indentations as appear to us. So how would the C/C++ treat it? Should the **else** be paired with the 1st **if** or the 2nd **if**?

```
if ( x > 5 )
  if ( y > 5 )
    cout << "x and y are > 5";
else
  cout << "x is <= 5";
```

Looks as if:
1st **cout** is executed when x > 5 and y > 5,
2nd **cout** is executed when x <= 5

```
if ( x > 5 )
  if ( y > 5 )
    cout << "x and y are > 5";
  else
    cout << "x is <= 5";
```

Looks as if:
1st **cout** is executed when x > 5 and y > 5,
2nd **cout** is executed when x > 5 and y <= 5

✓ **this is what the compiler treats as**

Dangling-Else Problem

- Recall that C++ is a free formatting language
  - The compiler will ignore any whitespaces, including indentations
- The compiler always pairs an **else** with the nearest previous **if** that is not already paired with some **else**
- To avoid the dangling else problem, use braces **{ }** to tell the compiler how to group the statements

Dangling-Else Problem

```
if ( x > 5 )
  if ( y > 5 )
    cout << "x and y are > 5";
  else
    cout << "x is <= 5";
```

is equivalent to

```
if ( x > 5 ) {
   if ( y > 5 )
     cout << "x and y are > 5";
   else
     cout << "x is <= 5";
 }
```

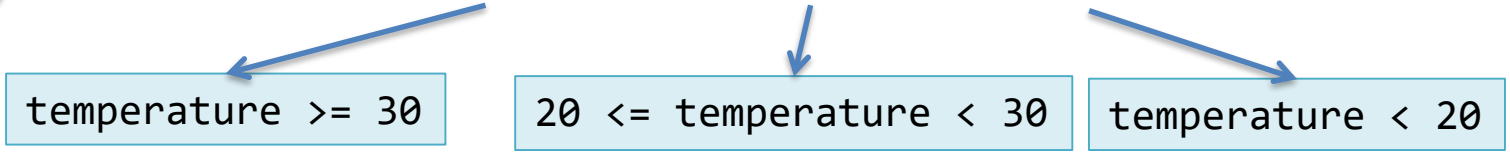
- If you want the 2nd **cout** to be executed when x <= 5:

```
if ( x > 5 ) {
  if ( y > 5 )
    cout << "x and y are > 5";
}
else
  cout << "x is <= 5";
```

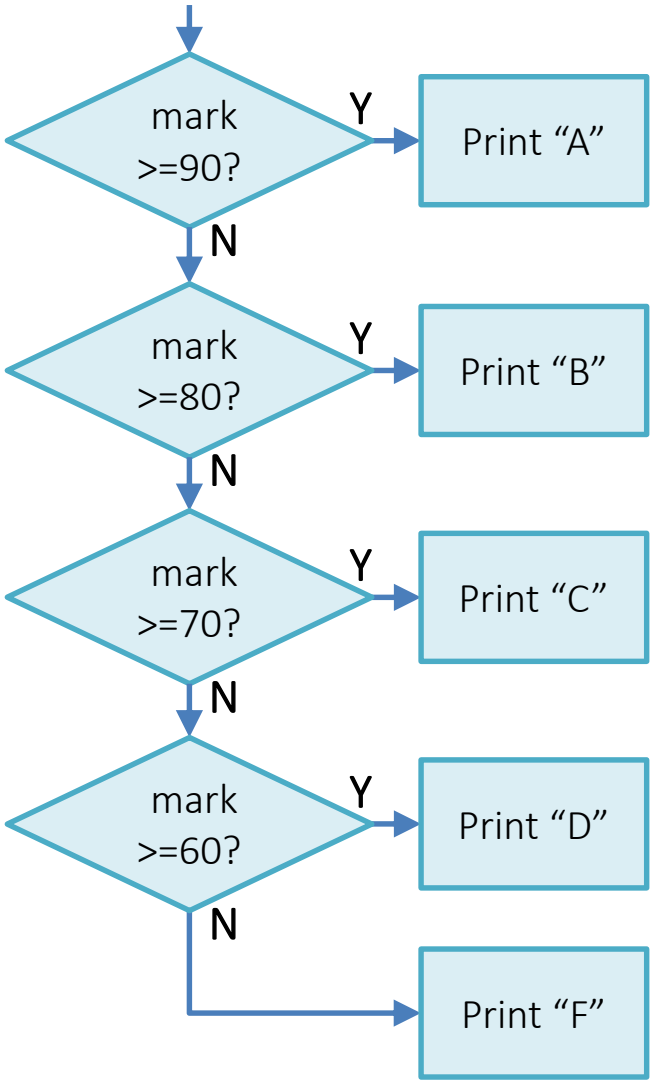
A Dangling-Else Example

```
if ( temperature >= 20 )
  if ( temperature >= 30 )
    cout << "good day for swimming" << endl;
  else
    cout << "good day for golfing" << endl;
else
  cout << "good day to play tennis";
```

1. How to pair up the **if**'s and **else**'s?
2. Conditions for swimming, golfing & tennis?

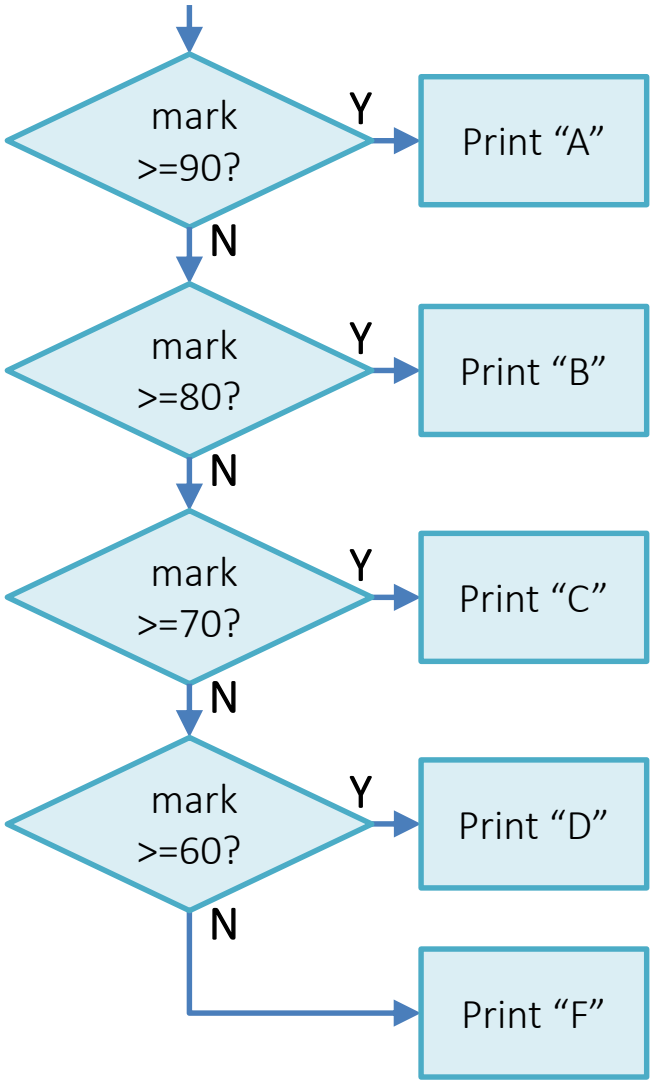


Multi-way **if-else** Statement




```
if ( mark >= 90 )  // 90 and above gets "A"
  cout << "A";
else
  if ( mark >= 80 )   // 80-89 gets "B"
    cout << "B";
  else
    if (mark >= 70 )   // 70-79 gets "C"
      cout << "C";
    else
      if (mark >= 60 )  // 60-69 gets "D"
        cout << "D";
      else          // less than 60 gets "F"
        cout << "F";
```

Multi-way **if-else** Statement



A more compact style is preferred

```
if ( mark >= 90 )   // 90 and above gets "A"
  cout << "A";
else if (mark >= 80 )   // 80-89 gets "B"
  cout << "B";
else if (mark >= 70 )   // 70-79 gets "C"
  cout << "C";
else if (mark >= 60 )   // 60-69 gets "D"
  cout << "D";
else     // less than 60 gets "F"
  cout << "F";
```

Series of **if** vs. Multi-way **if-else**

- What's the difference between the following two program segments?

```
if (mark >= 90 )
  cout << "A";
else if (mark >= 80 )
  cout << "B";
else if (mark >= 70 )
  cout << "C";
else if (mark >= 60 )
  cout << "D";
else
  cout << "F";
```

Faster, skip remaining if testing once hitting a true condition

```
if ( mark >= 90 )
  cout << "A";
if ( mark < 90 && mark >= 80 )
  cout << "B";
if ( mark < 80 && mark >= 70 )
  cout << "C";
if ( mark < 70 && mark >= 60 )
  cout << "D";
if ( mark < 60 )
  cout << "F";
```

Slower, needs to test all conditions even though only one of them can be true

Same program outcome but different performance!

switch Statement

Syntax

```
switch (controlling_expression) {
        case constant_1:
                statement_1;
                break;
        case constant_2:
                statement_2;
                break;
        …
        case constant_n:
                statement_n;
                break;
        default:
                default_statement;
}
```

- A multi-way branching action can also be achieved using a **switch** statement

The controlling expression in a switch statement must return either a Boolean value, an integer or a character

optional

switch Statement

```
char grade;
cin >> grade;
switch ( grade )
{
    case 'A':
    cout << "grade point is 4.0";
    break;
    case 'B':
    cout << "grade point is 3.0";
    break;
    case 'C':
    cout << "grade point is 2.0";
    break;
    case 'D':
    cout << "grade point is 1.0";
    break;
    case 'F':
    cout << "grade point is 0.0";
    break;
    default:
    cout << "grade is invalid";
}
```

1 When a **switch** statement is executed, the controlling_expression is evaluated,
the value of which must be one of Boolean, integer or character types

2 The constants given after the **case** keywords are checked in order until the first that equals the value of the controlling_expression is found, and then the following statements are executed

3 The **switch** statement ends when a **break** statement is encountered

4 If none of the constants matches the value of the controlling_expression, then the default_statement is executed

switch Statement

```
char grade;
cin >> grade;
switch ( grade )
{
     case 'A':
       cout << "grade point is 4.0";
       break;
     case 'B':
       cout << "grade point is 3.0";
       break;
     case 'C':
       cout << "grade point is 2.0";
       break;
     case 'D':
       cout << "grade point is 1.0";
       break;
     case 'F':
       cout << "grade point is 0.0";
       break;
     default:
       cout << "grade is invalid";
}
```

is equivalent to

```
char grade;
cin >> grade;
if (grade == 'A')
  cout << "grade point is 4.0";
else if (grade == 'B')
  cout << "grade point is 3.0";
else if (grade == 'C')
  cout << "grade point is 2.0";
else if (grade == 'D')
  cout << "grade point is 1.0";
else if (grade == 'F')
  cout << "grade point is 0.0";
else
  cout << "grade is invalid";
```

The switch statement is sometimes preferably especially when it can show clearly the flow of control depends on the value of `grade` only.

switch Statement more examples

```
switch ( mark / 10 ) {
    case 0:   case 1:
    case 2:   case 3:
    case 4:   case 5:
        grade = 'F';
        break;
    case 6:
        grade = 'D';
        break;
    case 7:
        grade = 'C';
        break;
    case 8:
        grade = 'B';
        break;
    case 9:
    case 10:
        grade = 'A';
        break;
    default:
        cout << "invalid mark";
}
```

Assuming that `mark` is of type `int` with range 0 to 100. Note that this is an integer division which results in an integer value.

What is the range of mark for grade to be assigned 'A'? 90-100

for grade to be assigned 'B'? 80-89

for grade to be assigned 'C'? 70-79

for grade to be assigned 'D'? 60-69

for grade to be assigned 'F'? 0-59

What if mark is out of the range 0 to 100?

The program will output "invalid mark" on screen

switch Statement more examples

```
switch ( age >= 18 ) {
    case 1:
        cout << “Old enough to vote";
        break;
    case 0:
        cout << “Not old enough to
vote";
        break;
}
```

What is the program output?

If age >= 18 is true, then output “Old enough to vote” to screen;
Otherwise output “Not old enough to vote” to screen

switch Statement

more examples

```
int main()
{
  int mark;
  cout << "Enter the mark: ";
  cin >> mark;

  switch ( mark / 10 ) {
    case 0:   case 1:
    case 2:   case 3:
    case 4:   case 5:
      cout << "The grade is F." << endl;
      break;
    case 6:
      cout << "The grade is D." << endl;
      break;
    case 7:
      cout << "The grade is C." << endl;
      break;
    case 8:
      cout << "The grade is B." << endl;
      break;
    case 9:
    case 10:
      cout << "The grade is A." << endl;
      break;
    default:
      cout << "Invalid mark." << endl;
  }

  return 0;
}
```

A recap

What is the output of the program segment if the input mark is 75?

```
Enter the mark:   75
The grade is C.
```

switch Statement more examples

```
int main()
{
  int mark;
  cout << "Enter the mark: ";
  cin >> mark;

  switch ( mark / 10 ) {
    case 0:   case 1:
    case 2:   case 3:
    case 4:   case 5:
      cout << "The grade is F." << endl;
    case 6:
      cout << "The grade is D." << endl;
    case 7:
      cout << "The grade is C." << endl;
    case 8:
      cout << "The grade is B." << endl;
    case 9:
    case 10:
      cout << "The grade is A." << endl;
    default:
      cout << "Invalid mark." << endl;
  }

  return 0;
}
```

Pay ATTENTION!
The break; statements are missing!

What is the output of the program segment if the input mark is 75?

```
Enter the mark:   75
The grade is C.
The grade is B.
The grade is A.
Invalid mark.
```

Common Mistakes

- Below are some common mistakes in the Boolean condition of an **if** or **if…else** statement:
  - Using an assignment instead of the equality operator, e.g.,

```
if (a = 10)
```

✗

```
if (a == 10)
```

✓

  - Using bitwise AND/OR instead of logical AND/OR operator, e.g.,

```
if (a != 0 & b > 0)
if (a != 0 | b > 0)
```

✗

```
if (a != 0 && b > 0)
if (a != 0 || b > 0)
```

✓

  - Using strings of inequalities, e.g.,

```
if (a < b < c)
```

✗

```
if (a < b && b > c)
```

✓

- These are all legal expressions in C++ and hence the compiler will not report any syntax error

?: — A Shorthand for If-Else

- A ternary operator that takes three operands:

condition ? expr1 : expr2

- A conditional expression that evaluates to a value:
  - if condition is **true**, **expr1** is the value of the expression
  - if condition is **false**, **expr2** is the value of the expression

```
if (mark >= 60)
    cout << "passed";
else
    cout << "failed";
```

is equivalent to

```
cout << ((mark >= 60)? "passed" : "failed");
```

Note: **if…else** is a statement,
?: is an operator that forms an expression

Doing something repeatedly

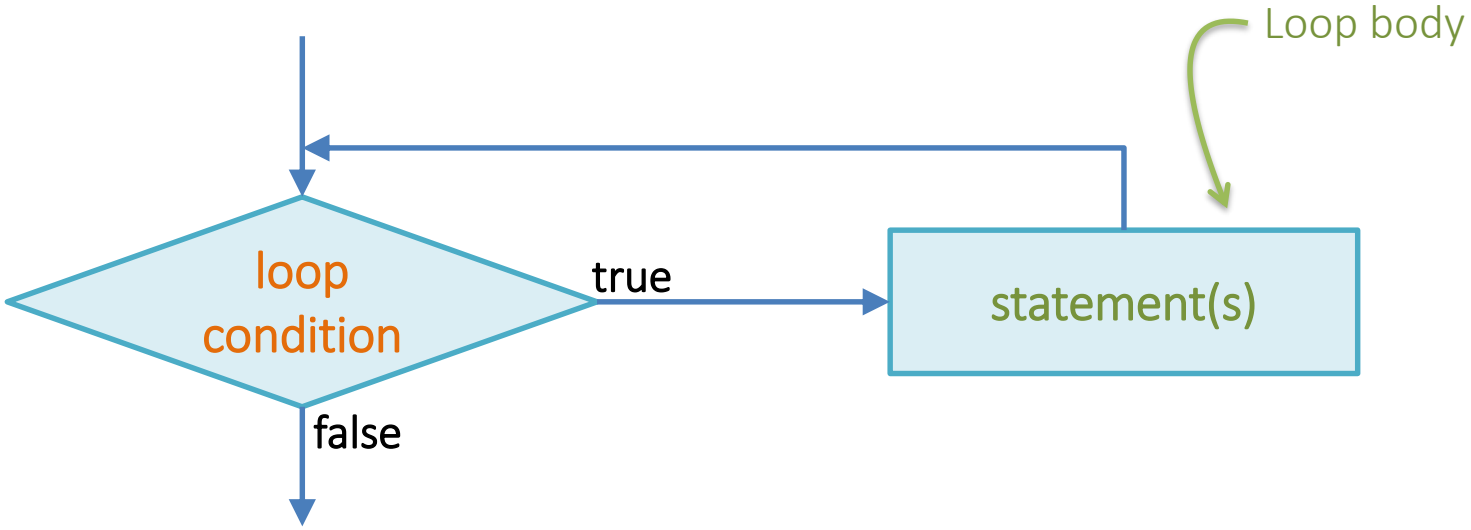
LOOPING

Loop

- A loop is any program construction that repeats a statement (or a compound statement) a number of times.
- The statement to be repeated in a loop is called the body of the loop.
- Each repetition of the loop body is called an iteration.
- In C++, looping can be achieved using either a **while** statement or a **for** statement.

Note: There is also the **do…while** statement, but we will leave it for you interest only.

while Statement



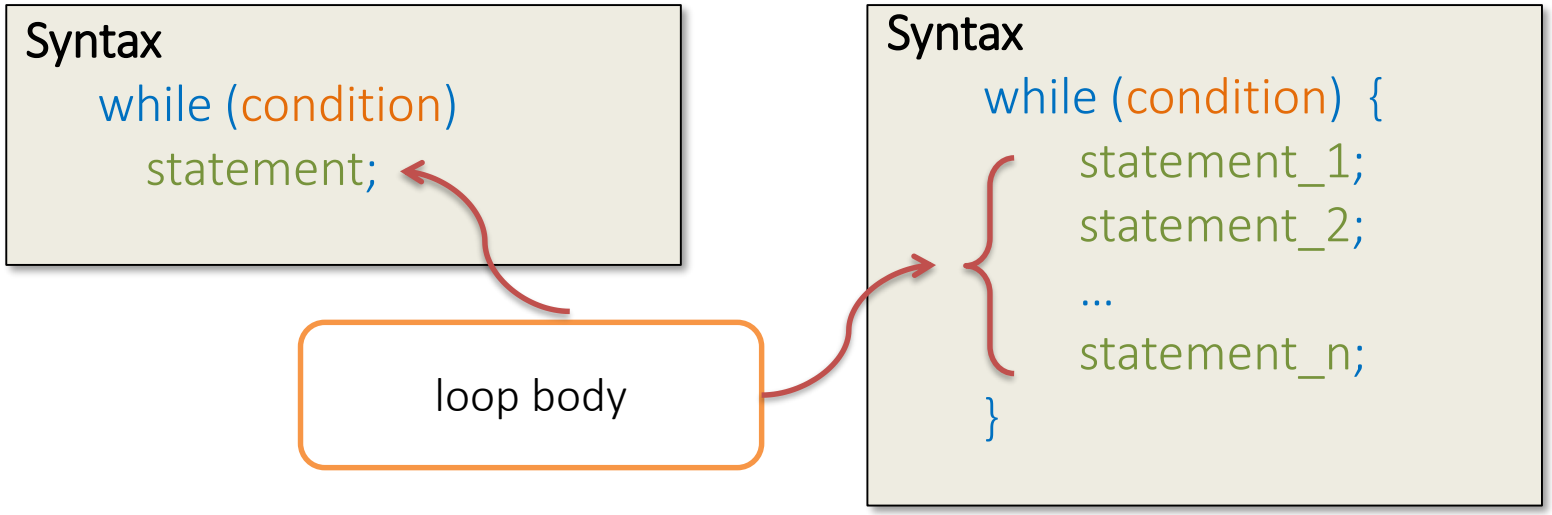

```
while (condition) {
        statement_1;
        statement_2;
        ...
        statement_n;
}
```

1 execution path when condition is true

2 execution path when condition is false

The while statement controls whether to repeat a loop body depending on a condition. Essentially, the loop body is executed repeatedly as long as condition is true

while Statement



- When a while statement (a.k.a. **while loop**) is executed, the condition is evaluated.
  - If it returns true, the loop body is executed once (i.e., one iteration).
  - If it returns false, the loop ends without executing its body.
- After each iteration, condition will be evaluated again and the process repeats.

while Statement

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

int main()
{
  int answer = 0;

  while (answer != 4) {
    cout << "2 * 2 = ";
    cin >> answer;
  }

  cout << "Correct!" << endl;

  return 0;
}
```

What does this program do?

Asks the user to answer 2 * 2 repeatedly until the user inputs the correct answer

What if the user keeps giving a wrong answer?

The program will keep asking again.

while Statement

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

int main()
{
  int answer = 0;
  int trials = 0;

  while (answer != 4) {
    cout << "2 * 2 = ";
    cin >> answer;
    trials++;
  }

  cout << "Correct!" << endl;
  cout << "You've tried " << trials << " times." << endl;

  return 0;
}
```

We may use a loop variable (or counter), which is of integer type, to count the number of iterations (i.e., how many times the loop body is executed).

What is the loop variable in this example?

`trials`

while Statement

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

int main()
{
  int x = 0, total = 0;
  cout << "Enter a negative num to end." << endl;

  while (x >= 0) {
    total += x;
    cout << "Total = " << total << endl;
    cout << "next number? ";
    cin >> x;
  }

  cout << "Program ends." << endl;

  return 0;
}
```

Sentinel-controlled while loops

to use a **special value** to indicate end of loop

In this example, the special value is any negative number. Also, the number of times the loop body is executed is determined at run time only (loops until user inputs a negative number).

Screen output?

```
Enter a negative number to end.
Total = 0
next number? 4 ↵
Total = 4
next number? 3 ↵
Total = 7
next number? 2↵
Total = 9
next number? 1↵
Total = 10
next number? -1 ↵
Program ends.
```

Note that the loop condition depends on the value of x, and hence **it is important** to make sure that the value of x will be updated within the loop body (as in cin >> x) in order for the condition (x >= 0) to change to false to exit the loop.

while Statement

Counter-controlled while loops
by decrementing a counter

How many times will the loop body be executed?

n

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

int main()
{
  int x = 0, total = 0, n;
  cout << "Enter the number of values to be added: ";
  cin >> n;

  while (n > 0) {
    cout << "next number? ";
    cin >> x;
    total += x;
    cout << "Total = " << total << endl;
    n--;
  }

  return 0;
}
```

```
Enter the number of values to be added: 3↵
next number? 4↵
Total = 4
next number? 3↵
Total = 7
next number? 2↵
Total = 9
```

Screen output?

Again, note that how the value of n is updated within the loop body to control loop repetition

while Statement

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

int main()
{
  int x = 0, total = 0, i, n;
  cout << "Enter the number of values to be added: ";
  cin >> n;

  i = 0;
  while (i < n) {
    cout << "next number? ";
    cin >> x;
    total += x;
    cout << "Total = " << total << endl;
    i++;
  }

  return 0;
}
```

Counter-controlled while loops
by incrementing a counter

How many times will the loop body be executed?

n

```
Enter the number of values to be added: 3↵
next number? 4↵
Total = 4
next number? 3↵
Total = 7
next number? 2↵
Total = 9
```

Screen output?

Typical Structure of a Counter-Controlled Loop

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

int main()
{
  int x = 0, total = 0, i, n;
  cout << "Enter # of values to add: ";
  cin >> n;

  i = 0;
  while (i < n) {
    cout << "next number? ";
    cin >> x;
    total += x;
    cout << "Total = " << total << endl;
    i++;
  }

  return 0;
}
```

loop variable — to count the no. of iterations

initialization of loop variable

condition for continuation

updating of loop variable inside the loop body

What if you forgot to update the loop variable?

while Statement

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

int main()
{
  int num = 23;
  int guess;
  bool isGuessed;

  isGuessed = false;

  while (!isGuessed) {
    cout << "Make a guess (0-99)? ";
    cin >> guess;

    if (guess == num) {
      cout << "Correct!" << endl;
      isGuessed = true;
    }
    else if (guess < num)
      cout << "Too small.  Guess again!" <<
    else
      cout << "Too large.  Guess again!" <<
  }
  return 0;
}
```

Flag-controlled while loops

use a **bool** variable to control the iterations

What is the flag in this example?

`isGuessed`

Screen output?

~~Make a guess (0-99)? 48 ↲~~
~~Too large. Guess again? 20 ↲~~
~~Too small. Guess again? 35 ↲~~
~~Too large. Guess again? 23 ↲~~
~~Correct!~~

```
Make a guess (0-99)? 48 ↲
Too large. Guess again!
Make a guess (0-99)? 20 ↲
Too small. Guess again!
Make a guess (0-99)? 35 ↲
Too large. Guess again!
Make a guess (0-99)? 23 ↲
Correct!
```

while Statement

What's wrong here?

```
int i = 0, n = 10;

while (i < n);
{
  cout << "next number? ";
  cin >> x;
  total += x;
  cout << "Total = " << total << endl;
  i++;
}
```

Never put a semicolon after the parenthesis as it is equivalent to introducing an empty statement (a.k.a. null statement) as the loop body. Essentially, this while statement contains an empty loop body

Will the loop counter be updated? So what will happen? Try it!

Quick Exercise 1

Write a complete C++ program that outputs the numbers 1 to 20, one per line, using a **while loop**.

(A sample program can be found at the end of this set of slides.)

for Statement

- The for statement (a.k.a. for loop) in C++ provides a compact way of expressing a loop structure

Output 1 to 20, one number of a line, using a for loop (i.e., same program outcome as quick exercise 1).

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

int main()
{
  int i;

  for (i = 1; i <= 20; ++i)
    cout << i << endl;

  return 0;
}
```

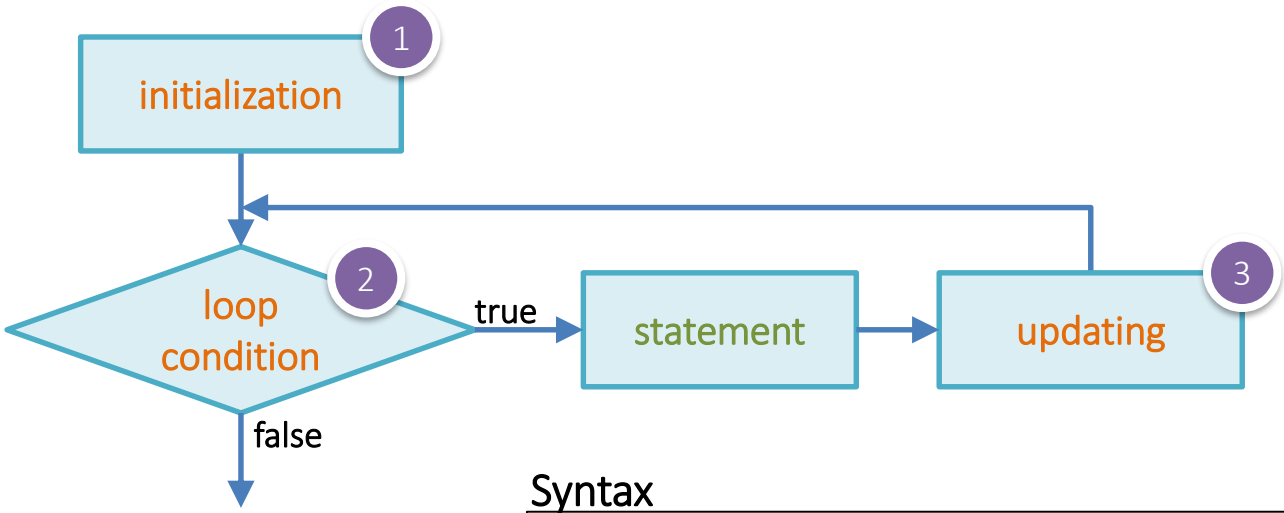
Now, take a close look at the three statement inside the round brackets () after the for keyword:

`i = 1;`
this statement is for initialization, i.e., it will only be executed once before the loop begins for the first time

`i <= 20;`
this statement is the loop condition for deciding whether to continue to loop. The loop body will be executed only if it is true.

`++i`
this statement is the updating statement which will be executed after each iteration of the loop. It usually updates the loop control variable (in this case `i`).

for Statement



Syntax

```
for (initialization; condition; updating) {
        statement_1;
        ...;
        statement_n;
}
```

execution path if condition is false, i.e., to exit the loop

execution path if condition is true, i.e., to execute the loop body

for Statement

- When a **for** statement is executed

  1. The initialization is performed.
     - Generally, it sets the initial value of the loop variable.
     - The initialization is executed only once.
  2. The condition is evaluated.
     - If it is true, the loop body is executed once (i.e., one iteration).
     - If it is false, the loop ends without executing its body.
  3. After each iteration, the updating of loop variable is performed, and the loop continues at Step 2.

for Statement

- Most while loops can be implemented as a for loop

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

int main()
{
  int answer = 0;
  int trials;

  for (trials = 0; answer != 4; trials++) {
    cout << "2 * 2 = ";
    cin >> answer;
  }

  cout << "Correct!" << endl;
  cout << "You've tried " << trials << " times." << endl;

  return 0;
}
```

Compare this program to this previous while loop example.

for vs. while

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

int main()
{
  int x = 0, total = 0, i, n;
  cout << "How many numbers to add? ";
  cin >> n;

  // for loop
  for (i = 0; i < n; i++) {
    cout << "next number? ";
    cin >> x;
    total += x;
    cout << "Total = " << total << endl;
  }

  return 0;
}
```

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

int main()
{
  int x = 0, total = 0, i, n;
  cout << "How many numbers to add? ";
  cin >> n;

  // while loop
  i = 0;
  while (i < n) {
    cout << "next number? ";
    cin >> x;
    total += x;
    cout << "Total = " << total << endl;
    i++;
  }

  return 0;
}
```

Compare the above two programs which have the same program behavior.

Quick Exercise 2

Write a program that outputs 9 8 7 6 5 4 3 2 1 0 in a single line using a **for** loop.

(A sample program can be found at the end of this set of slides.)

Quick Exercise 3

Write a program that calculates the sum of odd numbers between 1 and 20 using a **for** loop.

(A sample program can be found at the end of this set of slides.)

break Statement

- The break statement can be used to exit a loop from inside a loop body.
- When a break statement is executed,
  - the loop ends immediately.
  - the execution continues with the statement following the loop.
- The break statement may be used in both **while** loop and **for** loop.
- Note: Avoid using a break statement to end a loop unless absolutely necessary because it might make it hard to understand your code.
  - A proper way to end a loop is using the condition for continuation.

break Statement

Yes, you may declare and initialize the counter variable at the same time in the initialize statement in the for loop

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

int main()
{
  for (int i = 0; i >= 0; i++) {
    if (i == 15) break;
    cout << i << " ";
  }

  return 0;
}
```

As the condition is always true, this will be an infinite loop

The break statement is used here to exit the infinite loop when i == 15

Screen output?

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Can you rewrite the program so that it produces the same output without using the break statement?

continue Statement

- The continue statement is used to terminate the current iteration of a loop.
- When a **continue** statement is executed,
  - any loop body statements after it will be skipped.
  - the loop continues by starting the next iteration.
- Like the break statement, the **continue** statement may be used in both **while** loop and **for** loop.

continue Statement

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

int main()
{
    for (int i = 0; i < 20; ++i) {
        if (i % 2 == 0) continue;
        cout << i << " ";
    }

    cout << endl;

    return 0;
}
```

The continue statement is used here to skip those i's which are even

When the continue statement is executed, the succeeding cout statement are skipped. The next iteration begins by updating the loop variable and checking the condition.

Screen output?

```
1 3 5 7 9 11 13 15 17 19
```

Can you rewrite the program so that it produces the same output without using the continue statement?

Examples on **break** and **continue**

```
int count;
for ( count = 1; count <= 10; ++count) {
  if (count == 5) break;

  cout << count << " ";
}
cout << endl << "Broke out of loop at count = " << count << endl;
```

Screen output?

```
1 2 3 4
Broke out of loop at count = 5
```

```
for ( int count = 1; count <= 10; ++count) {
  if (count == 5) continue;
  cout << count << " ";
}
```

Screen output?

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

Answer to Quick Exercise 1

Write a complete C++ program that outputs the numbers 1 to 20, one per line, using a **while loop**

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

int main()
{
  int i = 1, n = 20;

  while (i <= n) {
    cout << i << end;
    i++;
  }

  return 0;
}
```

A shorter version

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

int main()
{
  int i = 1, n = 20;

  while (i <= n)
    cout << i++ << end;

  return 0;
}
```

We can't use ++i here. Using ++i will output 2 to 21 instead. Why? Review how the prefix and postfix operators work here.

Answer to Quick Exercise 2

Write a program that outputs 9 8 7 6 5 4 3 2 1 0 in a single line using a **for** loop.

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

int main()
{
  int i;

  for (i = 9; i >= 0; --i)
    cout << i << ' ';

  return 0;
}
```

Try to repeat this exercise with a while loop.

Answer to Quick Exercise 3

Write a program that calculates and outputs the sum of odd numbers between 1 and 20 using a **for** loop.

```
#include <iostream>
using namespace std;
int main()
{
    int i, sum = 0;
    for (i = 1; i <= 20;
++i) {
        if (i % 2 == 1) {
            sum += i;
        }
    }
    cout << sum << endl;
    return 0;
}
```

We are happy to help you!

"If you face any problems in understanding the materials,
please feel free to contact me, our TAs or student TAs.
We are very happy to help you!
We wish you enjoy learning C++ programming in this class ☺."