Hochiminh City University of Technology
Computer Science and Engineering
OOP in C++

Basic C++

Lecturer: Tran Huy

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.”

– Bjarne Stroustrup

Today's outline

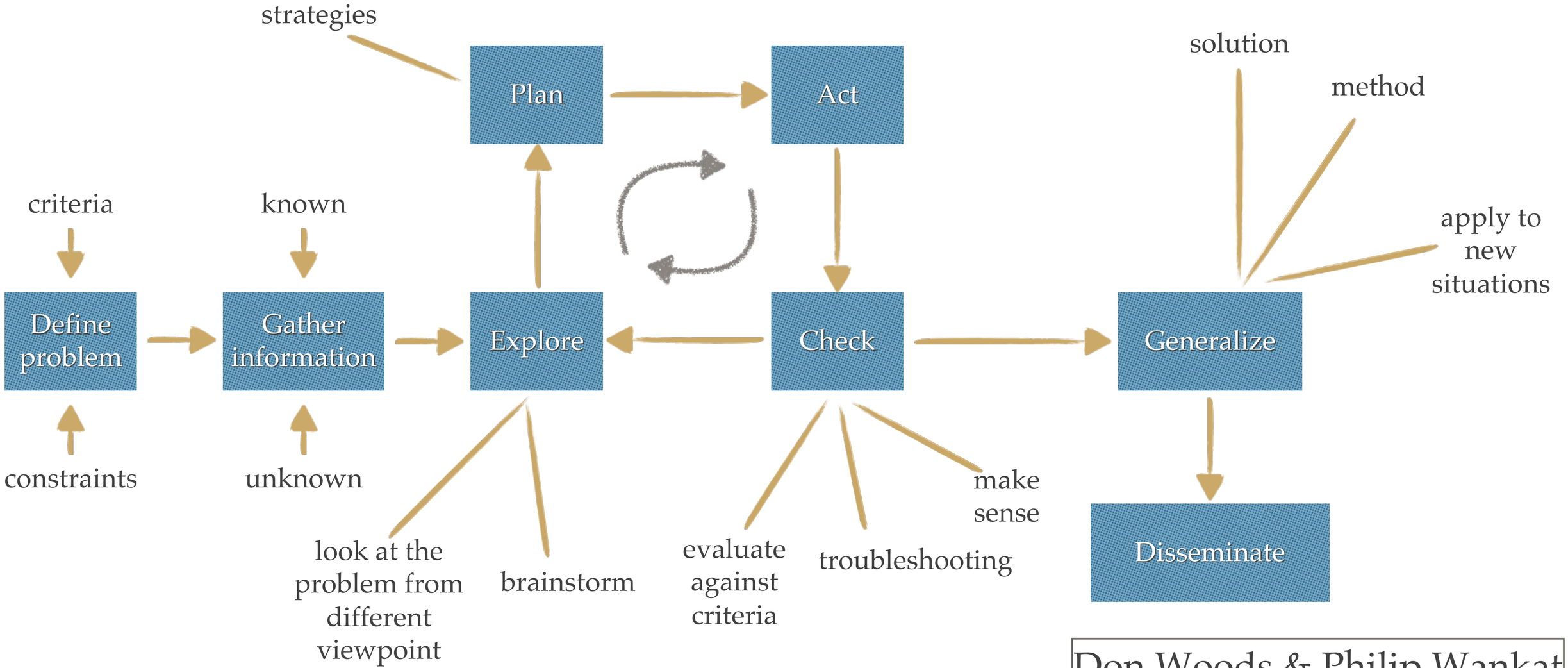
- Program structure
- Variable and Data types
- Operations
- C++ Libraries
- Macro
- Conditional Statements
- Loop Statements
- Array
- String
- Function and parameter passing

Problem solving

Problem solving

- Think about how you solved your problems before
  - Math, Physic, Chemistry, etc.
- How did you solved them?
  - Systematic
  - Random idea
  - Memorising solutions
  - Other approaches

Problem solving



Don Woods & Philip Wankat

Problem solving

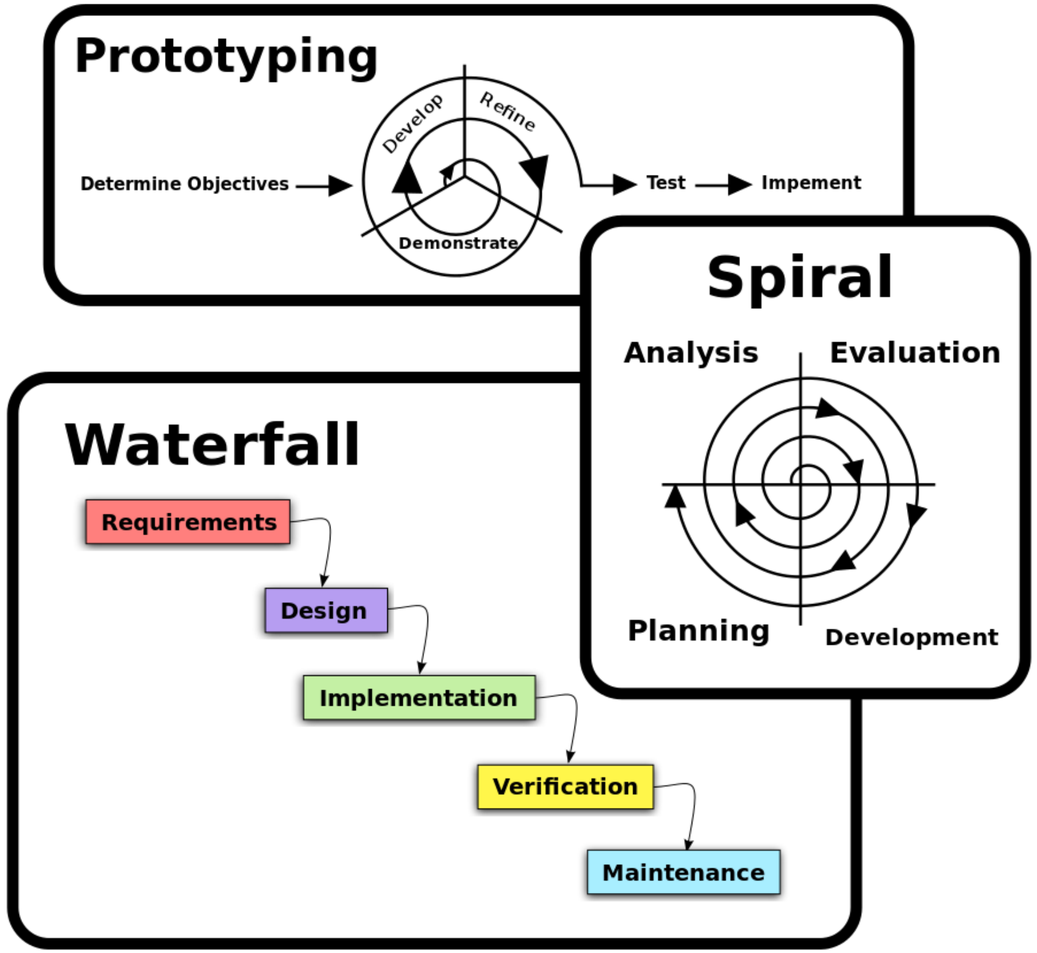
- How will you solve a problem on computer?
  - Computer deals best with performing easy tasks over and over again.
- Principle: break the big problem into smaller pieces
  - We utilize the computer's ability by implementing repetitive techniques to incrementally solve our complex problems.
- Think about a problem that you can solve using this method.

Problem solving

- Two basic approaches
  - Iterative: solve a part of the problem, repeat the process on the remaining problem until the original problem is solved
  - Recursive: define how to solve simplest problems. Then we break the problem into simpler and simpler pieces until they reach the level that computer know how to solve (by our definition).

Software development

- The software development process follows the principles of problem solving.
- Various software development models existed
- What do you need to develop a software at the basic level?
  - Compiler, Debugger, Editor
  - Integrated Development Environment (IDE)
    - Visual Studio, Eclipse, Xcode, etc.



[source: Wikipedia]

Software development

- Address your problem, collect requirements
- Analyse feasible approaches, find solution
- Design algorithm
- Implement: write code
- Compile, debug
- Evaluate the program
- Deploy software

Steps in development of algorithm

- Problem definition
- Development of a model
- Specification of Algorithm
- Designing an Algorithm
- Checking the correctness of Algorithm
- Analysis of Algorithm
- Implementation of Algorithm
- Program testing
- Documentation Preparation

Algorithm

- Algorithm is a self-contained list of operations to be performed.
- An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function.
- Describe algorithm
  - Text: details of data flow and processing steps
  - Pseudo code
  - Flowchart

Algorithm

- Pseudo code
  - Independent from programming language.
  - An informal high-level description of the operating principle of a computer program or algorithm.
  - No standard for pseudo code syntax
  - Algorithm is often designed with pseudo code description

```
function randomSound()
Loop: from i = 1 to 100
    print_number = True
    If i is even Then
       If print_number Then
         Print (i + random())
       Else
         PlaySound( rand() )
       End If
    Else
       Print “Odd…”
    End If
End (loop)
End (function)
```

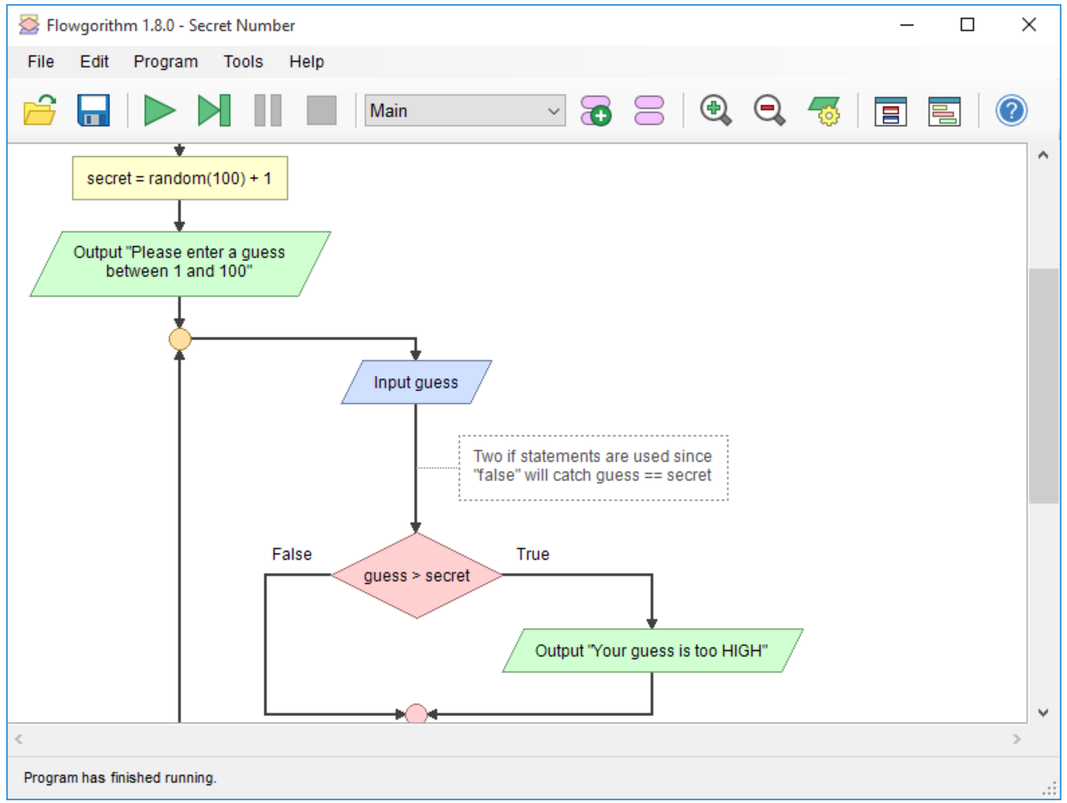
Algorithm

Pseudo code - guideline

- Mimic good code and natural language
- Ignore unnecessary details
- Don’t belabour the obvious
- Take advantage of programming shorthands
- Consider context
- Don’t lose sign of the underlying model
- Check for balance

Algorithm

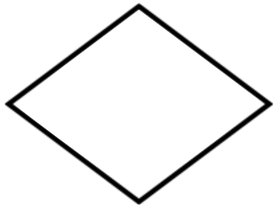
- Flow chart
  - A type of diagram that represents the algorithm (in our context). The flow chart illustrates a solution model to a given problem.
  - A flow chart is constructed from basic shapes that have specific meanings.



Algorithm

- Flow chart - Building blocks


Flow line

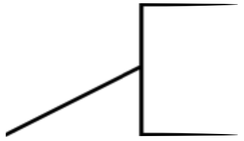

Decision


Process


On-page connector


Input/Output


Preparation


Annotation


Predefined process


Off-page connector


Terminal

[source: Wikipedia]

Algorithm

- Flowchart
  - Flow line: represents control pass from one symbol to another.
  - On-page connector: has more than one arrow coming into it but only one going out. It is useful to represent iterative processes.
  - Annotation: represents comments or remarks about the flowchart.
  - Terminal: usually has word/phrase to indicate the start/end of a process.
  - Decision: where the decision must be made (usually Y/N).

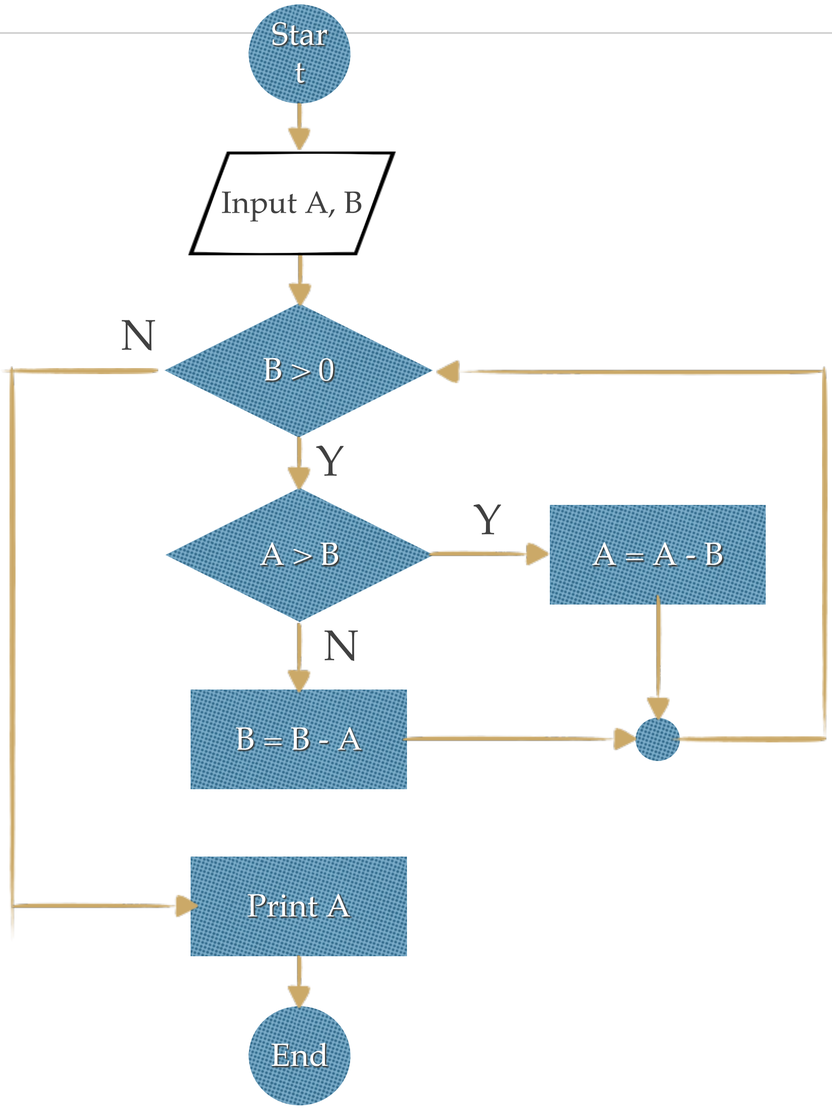
Algorithm

- Flowchart
  - Input/Output: involves receiving data and displaying processed data.
  - Predefined process: represents complex processing steps which maybe detailed in a separate flowchart.
  - Process: shows that something is performed.
  - Preparation: prepares a value for a subsequent conditional or decision step (replace decision symbol in case of conditional loop).
  - Off-page connector: connect to another page.

Algorithm

- Flowchart - example

```
INPUT A,B
Loop while B > 0
    If A > B then
        A = A - B
    Else
        B = B - A
    End
End loop
Print A
```



Program structure

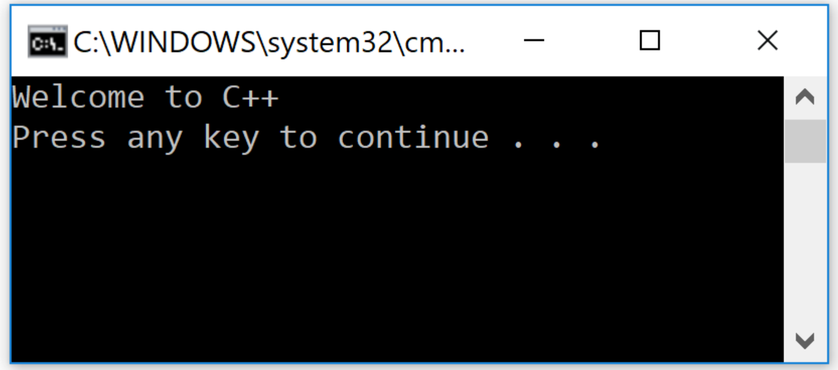
Program structure

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

int main()
{
   cout << "Welcome to C++!\n";
   return 0;
}
```

preprocessor directives

main() function



Program structure

- Preprocessing directive: anything begin with #
  - include: source inclusion (use libraries or additional code)
  - define, undef, etc.: constant, macros
  - pragma: specify diverse options for compiler
- main(): the entry point of our program. This is where everything start.

Program structure

- Global variables definition: these variable are visible to all classes and functions in the program
- Structure, Class or Function definition and implementation
- *Namespace*: use to group components of a module, library, or a pack of smaller libraries.

Using namespace

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

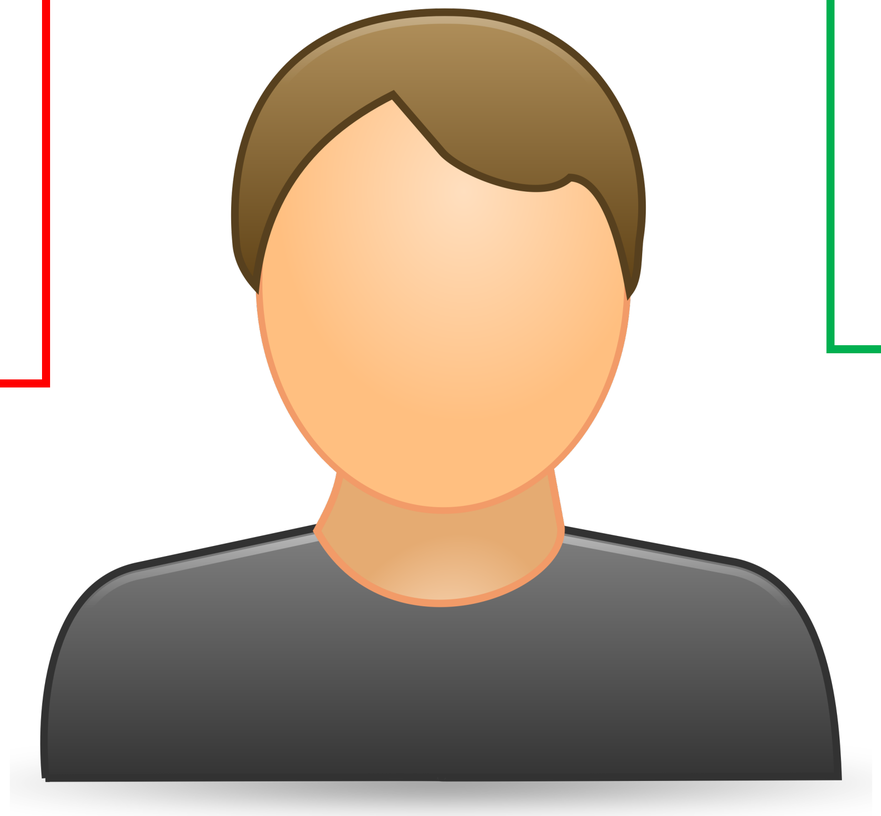

int main()
{
   cout << "Welcome to C++!\n";
   return 0;
}
```

Program structure

```
#include <something>
// define data structure
// define functions
// declare global variables
// using namespace
int a = 5;
int main()
{
   // local variables
   /* Your code should be put here.
   TO DO
   */
   int a = 3;
   return 0;
}
```

User IO

std::cout for writing to the console.



std::cin
for reading from the console

User IO

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

int main()
{
    int age;
    cout << "How old are you?\n";
    cin >> age;
    cout << "Your age: " << age << endl;
    return 0;
}
```

Comments

- Two types
  - Single line comments: anything after //
    - `a = 0; // set variable a to 0`
  - Block comments: anything between /* and */
    - ```
      b = 1; /* set b to 1.
      remember that the value variable b
      can be changed later */
      ```
      ```

Style guide

- There are a number of style guides available, **the best one is the one used by the people who are paying you**.
- A straightforward style guide is:
  C++ Coding Standards
- For a more detailed guideline:
  Google C++ Style Guideline

Compile the program

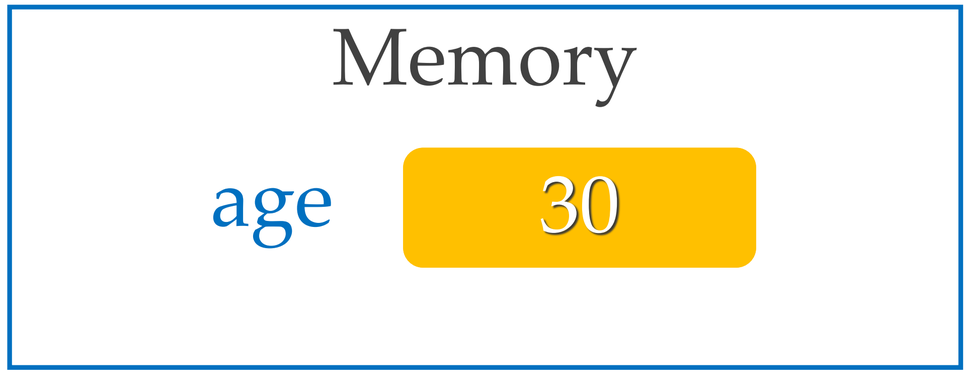
- Simplest way: using IDE
  - Visual Studio, Xcode, KDE IDE, Eclipse, etc.
- Manually: gcc, g++
  - gcc example.c
  - g++ example.cpp
- Compile and link separately
- Use other build system: e.g. CMake

Variables and Data types

Variable

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

int main()
{
    int age;
    cout << "How old are you?\n";
    cin >> age;
    cout << "Your age: " << age << endl;
    return 0;
}
```



Data types

- Data types in C++ is mainly divided into two types:
  - **Primitive Data Types:** built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char , float, bool etc.
  - **Abstract or user defined data type:** defined by user itself. Like, defining a class in C++ or a structure.

Primitive Built-in Types

Type	Keyword
Boolean	Bool (1 bit)
Character	Char (1 byte)
Integer	Int (4 byte)
Floating point	Float (4 byte)
Double floating point	Double (8 byte)
Valueless	void
Wide character	wchar_t

Memory size

Type	Typical Size (in bytes)	Typical Range
char	1	-127 to 127 or 0 to 255
unsigned char	1	0 to 255
signed char	1	-127 to 127
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
signed int	4	-2147483648 to 2147483647
short int	2	-32768 to 32767
unsigned short int	2	0 to 65,535
signed short int	4	-32768 to 32767
long int	4	-2,147,483,648 to 2,147,483,647
signed long int	4	same as long int
unsigned long int	4	0 to 4,294,967,295
float	4	+/- 3.4e +/- 38 (~7 digits)
double	8	+/- 1.7e +/- 308 (~15 digits)
long double	8 or 12	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4	1 wide character

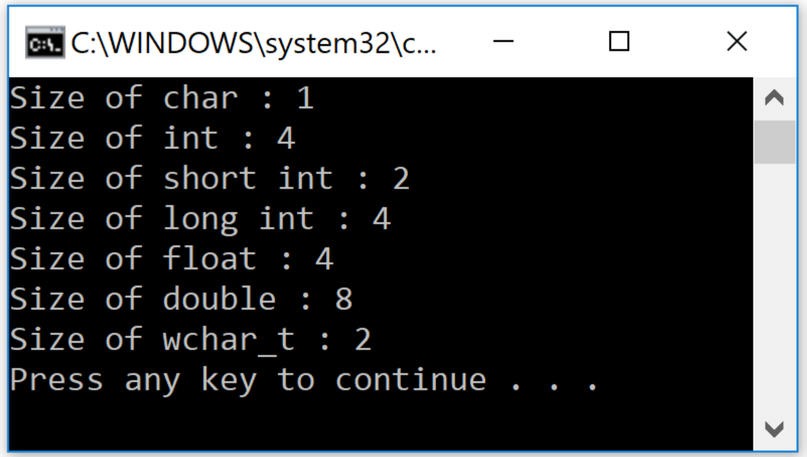
https://www.tutorialspoint.com/cplusplus/cpp_data_types.htm

Memory size

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

int main() {
  cout << "Size of char : " << sizeof(char) << endl;
  cout << "Size of int : " << sizeof(int) << endl;
  cout << "Size of short int : " << sizeof(short int) << endl;
  cout << "Size of long int : " << sizeof(long int) << endl;
  cout << "Size of float : " << sizeof(float) << endl;
  cout << "Size of double : " << sizeof(double) << endl;
  cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

  return 0;
}
```



Variable declaration

- [optional] <type> <variable name>[<array declaration>][=assigned values]
  - int a = 5;
  - float x = 0.5f;
  - const char* depName = “BK-CSE”;
  - volatile int noOpt = 100;
  - etc.
  - const int a = 5, b = 5, c = 5;

Variable declaration

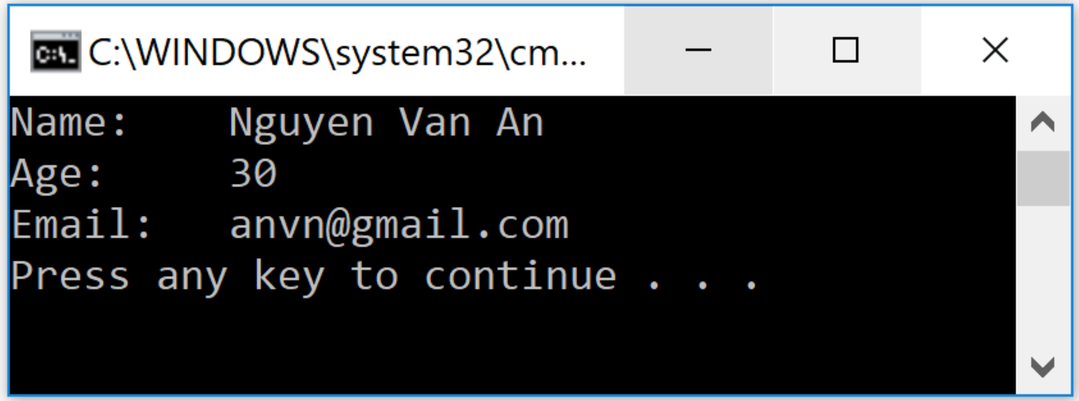
- Rules for variable name (identifiers, in general):
  - A valid identifier is a sequence of one or more letters, digits, or underscore characters (_).
  - Spaces, punctuation marks, and symbols cannot be part of an identifier.
  - Identifiers shall always begin with a letter/_.
- Case sensitive
- Reserved keywords:
  - `alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq`

Output format

Output format

- Use escape sequences
  - \n for new line
  - \t for tab

```
cout << "Name:\t Nguyen Van An\n";
cout << "Age:\t 30\n";
cout << "Email:\t anvn@gmail.com\n";
```



Output format

- Use <iomanip> library (http://www.cplusplus.com/reference/iomanip/)
  - `setw(n)`: set the *field width n* to be used on output operations. By default, `setw(n)` will **justify** everything to the right.
  - Use `left, right, internal` to adjust to the side you want

```
#include <iomanip>

cout << left << setw(8) << "Name:" << "Nguyen Van An\n";
cout << left << setw(8) << "Age:" << "30\n";
cout << left << setw(8) << "Email:" << "anvn@gmail.com\n";
```

Output format

- Using cout: require “iomanip” for formatting
  - `cout.width(<output width>)`: set width of the output result, both text and number
    - `cout << setw(<output width>)`
  - `cout.precision(<number>)`: set maximum number of significant digits (set to 0 to reset this setting)
    - `cout << setprecision(<number>)`
- `cout << “*” << setw(4) << 8 << “*” << endl;`
- `cout << “*” << setprecision(4) << 12356.4 << “*” << endl;`

Output format

- Save old settings:
  - `ios::fmtflags old_settings = cout.flags();`
  - `int old_precision = cout.precision();`
- Load settings:
  - `cout.flags(new_settings);`
  - `cout.precision(new_precision);`

Output format

- Fixed format for floating point number
  - `cout.setf(ios::fixed, ios::floatfield);`
  - `cout << fixed;`
  - Reset to default: `cout.setf(0, ios::floatfield);`
- E.g.:
  - ```
    cout.setf(ios::fixed, ios::floatfield);
    cout.precision(2);
    cout << 3.14159 << “, ” << 0.8642e-3;
    ```
    ```

Basic Operations

Basic Operations

- Arithmetic operators: `+, -, *, /, %`
- Bitwise operators: `^, ~, &, |, >>, <<`
- Logic operators: `!, &&, ||, >, <, ==`
- Assignment: `=`

Arithmetic operations

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Example

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

int main()
{
    cout << 15 / 4 << endl; -> 3
    cout << 15 / 4.0 << endl; -> 3.75
    cout << 15 % 4 << endl;  -> 3
    return 0;
}
```

Assignment operation

Assignment operation

- <left operand> = <expression>
- return <left operand>
- <left operand> can’t be constant
- Example:
  - `Const pi = 3.1415;`
  - `PI = 4;`
  - `keyPressed = ‘q’;`
  - `(a = (b = 5)); => (a = 5);`

Assignment operation

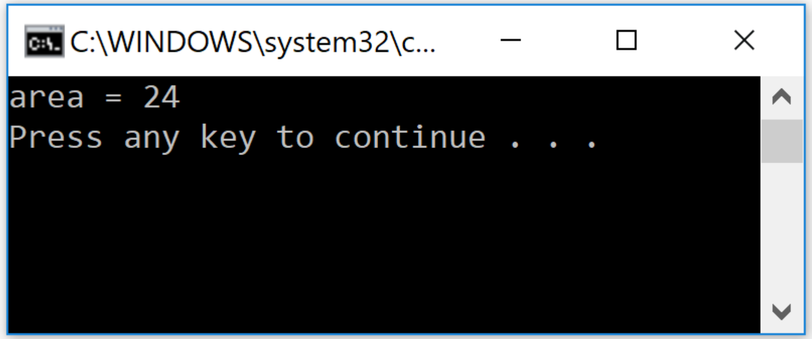
- Assign at the declaration instruction:
  - `int x = 10;`
  - `int y{8};`
  - `float z(10.01f);`

Example

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

int main()
{
   float width = 4.5;
   float height = 5.5;

   int area = width * height;
   cout << "area = " << area << endl;
   return 0;
}
```



Exercise

```
/*Goal: write a program that calculates the volumes of a cube,
** Write the values to the console.
*/
#include<iostream>
using namespace std;

int main()
{
    //Dimension of the cube
    float cubeSide;
    cout << "Enter the size of cube: ";
    cin >> cubeSide;

    //TODO

    return 0;
}
```

Assignment operation

- What is the default type of constants?
- Can we assign different types of value to a variable?

Assignment operation

- Default type of constants depend on how you declare it
  - 10: decimal value, default type depends on context
    - 012: octal value
    - 0x64: hexadecimal value
  - 3.1415: default type is double

Cast operator

- Implicit convert the value from one type to another type
- (<target type>)<expression>
- E.g.:
  - `int a = 3.9583;`
  - `float x = (float)a + 0.5f;`
  - `double y = (double)x * (double)a;// y = x * a; is fine`

Implicit conversion

```
int a = 65, integer = 80;
char charA = 65, charB = ‘B’, charC = 67;
float answer = 0, floatNumber = 0.0;

//we can assign an integer to a float
floatNumber = integer;

//we can assign a char to a float
floatNumber = charB;

answer = floatNumber / 4;
//But assigning a float to a char doesn't quite work
charC = answer;

//assigning a float to an interger, results in the float being truncated
integer = answer;
```

Auto type

- *auto type appears from C++11 standard*.
- Should we use auto?
- Where can we use auto?
- auto type: good or bad?

Compound assignments

Operator	Example	Equivalent expression
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

PreFix and PostFix

Incrementing	Decrementing
prefix: ++a	prefix: --a
postfix: a++	postfix: a--

Example

```
#include<iostream>
using namespace std;
int main() {
    int a = 10, b = 10;
    int post, pre = 0;
    cout << "Inital values: \t\t\tpost = " << post << " pre= " << pre << "\n";
    post = a++;
    pre = ++b;
    cout << "After one postfix and prefix: \tpost = " << post << " pre= " << pre << "\n";
    post = a++;
    pre = ++b;
    cout << "After two postfix and prefix: \tpost = " << post << " pre= " << pre << "\n";
    return 0;
}
```

C++ Standard Library

C++ Standard Library

- Library is the place where you implement functions, classes to serve some specific tasks.
- Library contains:
  - Definitions: constants, macro, structure, class
  - Functions: implement specific algorithms, a unit of reusable code
  - Class implementations

Library functions

- Function: a named sequence of code that performs a specific task
- Definition
  - `<return type> <function name>(<in/out parameters>);// prototype`
  - ```
    <return type> <function name>(<in/out parameters>)
    {
        // your implementation
    }
    ```
    ```

C++ Standard Library

- Common standard libraries:
  - <stdio.h>, <cstdio>
  - <math.h>, <cmath>
  - <string.h>, <cstring>
  - <assert.h>, <cassert>
  - <errno.h>, <cerrno>
  - <time.h>, <ctime>
- For more detail refer to http://en.cppreference.com/w/cpp/header

<cstdio> library

- `<cstdio>` perform Input/Output operations:
- For more detail refer to http://www.cplusplus.com/reference/cstdio/

Output format

- Text output format
  - `printf("i = %d\n", i);`
  - `cout << "i = " << i << endl;`
- Using function is a convenient way to format output.
- Using I/O streams require a bit modification in the sequence.

Output format

- printf(<format string>, arguments)
  - Format string can contain format specifiers with the following syntax:
    - `%[flags][width][.precision][length]specifier`
    - specifier: d/i, u, o, x/X(uppercase), f/F, e/E, g/G, a/A, c, s, p, n, %(escape character)
    - flags: +, -, space, #, 0
    - .precision: .number, .*
    - width: number, *

Output format

specifier	output	example
d/i	signed decimal integer	-2354
u	unsigned decimal integer	3056
o	unsigned octal	342
x/X	unsigned hexadecimal integer	6f0c
f/F	decimal floating point	3.14159
e/E	scientific notation	3.14159e-05
g/G	use shortest representation	3.14159
a/A	hexadecimal floating point	-0xc.90dep-3
c	character	a
s	string	damn it
p	pointer address	b8000000
n	nothing will be printed, argument must be a pointer to a signed int. The number of printed characters are stored location pointed by the pointer.	
%	print '%' character	%

Output format

```c
#include <stdio.h>

int main()
{
    printf("Characters: %c %c \n", 'a', 65);
    printf("Decimals: %d %ld\n", 1977, 650000L);
    printf("Preceding with blanks: %10d \n", 1977);
    printf("Preceding with zeros: %010d \n", 1977);
    printf("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
    printf("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
    printf("Width trick: %*d \n", 5, 10);
    printf("%s \n", "A string");
    return 0;
}
```

<cstdio> library

```c
/* gets example */
#include <cstdio>

int main()
{
    char string[256];
    printf("Insert your full address: ");
    gets_s(string);
    printf("Your address is: %s\n", string);
    return 0;
}
```

http://www.cplusplus.com/reference/cstdio/gets/

<cmath> library

- `<cmath>` declares a set of functions to compute common mathematical operations :
  - Trigonometric functions (`sin, cos, tan`, etc)
  - Hyperbolic functions (`sinh, cosh, tanh`, etc)
  - Exponential and logarithmic functions (`exp, log`, etc)
  - Power functions (`pow, sqrt`, etc)
  - Rounding and remainder functions (`ceil, floor`, etc)
- For more detail refer to http://www.cplusplus.com/reference/cmath/

<cmath> library

```
/* sin example */
#include <cstdio>       /* printf */
#include <cmath>        /* sin */

#define PI 3.14159265

int main()
{
    double param, result;
    param = 30.0;
    result = sin(param*PI / 180);
    printf("The sine of %f degrees is %f.\n", param, result);
    return 0;
}
```

http://www.cplusplus.com/reference/cstdio/gets/

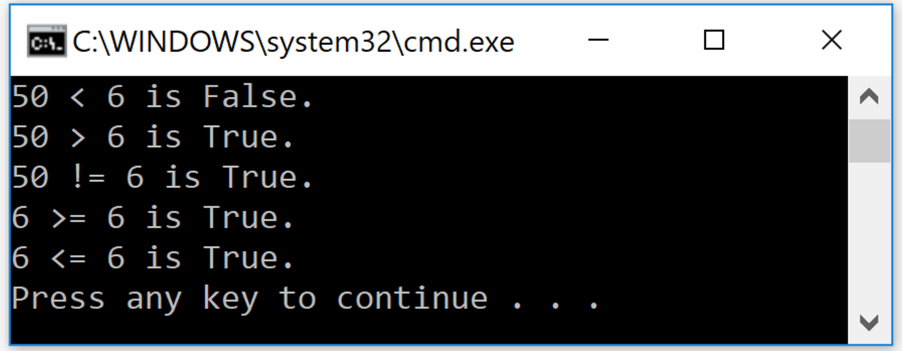
Relational and logical operators

Relational operators

Operator	Meaning
“=="	Equal to
“<"	Less than
“>"	Greater than
“<="	Less than or equal to
“>="	Greater than or equal to
“ !=”	Not equal to

Relational operators

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

int main() {
    //instead of printing 0 and 1, create an array where
    //0 = False, 1 = True
    string TorF[] = { "False", "True" };
    int a = 50, b = 6, c = 6;

    //Print out the string values of each relational operation
    printf("%d < %d is %s.\n", a, b, TorF[a < b].c_str());
    printf("%d > %d is %s.\n", a, b, TorF[a > b].c_str());
    printf("%d != %d is %s.\n", a, b, TorF[a != b].c_str());
    printf("%d >= %d is %s.\n", b, c, TorF[b >= c].c_str());
    printf("%d <= %d is %s.\n", b, c, TorF[b <= c].c_str());
    return 0;
}
```

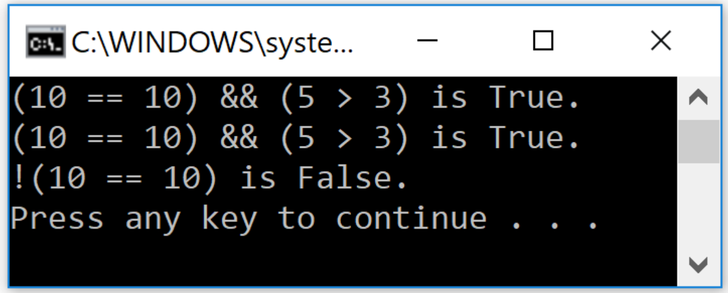


Logical operators

Operator	Meaning	Behavior
&&	and	If both inputs are true the outcome of the operation is true; otherwise false
||	or	If both inputs are false the outcome of the operation is false; otherwise true
!	not	Negates the logical condition

Logical operators

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

int main() {
  int a = 10, b = 5, c = 10, d = 3;
  std::string TorF[] = { "False", "True" };

  //The && operator
  printf("(%d == %d) && (%d > %d) is %s.\n", a, c, b, d, TorF[(a == c) && (b > d)].c_str());
  //The || operator
  printf("(%d == %d) && (%d > %d) is %s.\n", a, c, b, d, TorF[(a == c) || (b == d)].c_str());
  //The 'Not' operator
  printf("!(%d == %d) is %s.\n", a, c, TorF[!(a == c)].c_str());

  return 0;
}
```



Conditional execution

- Boolean expression: evaluate to true/false
  - What is true? What is false?
  - bool type
  - Type conversion
  - Assignment
  - Common expressions

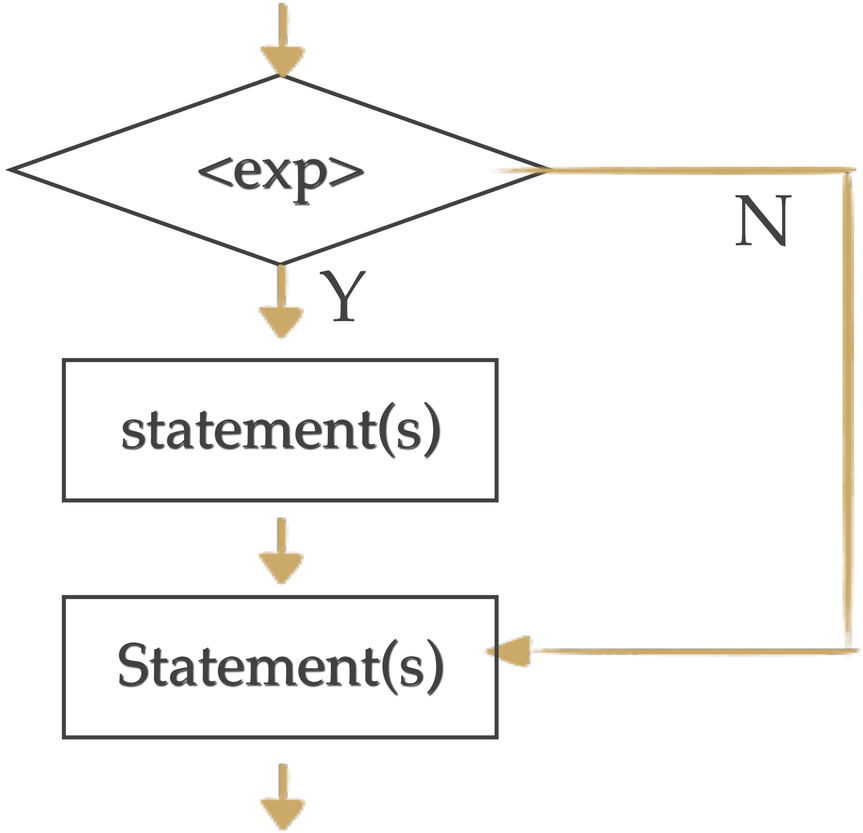
Conditional execution

- Type bool: true/false
  - Size: 1 byte (basic unit of storage)
  - Be represented as integer: true = 1, false = 0
- What happens when you assign a value to boolean type:
  - False: 0 value (for integer, floating point number, character '\0')
  - True: anything else (except structures, unless a casting operator is defined)

Conditional execution

- Examples:
  - ```
    bool b = true, b1 = false;
    int a = -1, c = 0;
    float x = 0.5f, y = 1.2f;
    b = a > c;
    b1 = a;
    b = c;
    b1 = x < y && a > c;
    b = x;
    c = y + b1;
    b1 = 50 != ‘a’;
    b = x + 4.9 < y / 0.5f;
    ```
    ```

If-else statement

If statement

- Simple if statement:
  - Execute a statement or a list of statements if the given condition is satisfied
  - `if (<conditional expression>) <statement>;`
  - ```
    if (<conditional expression>) {
        <statements>
    }
    ```
    ```

Flowchart



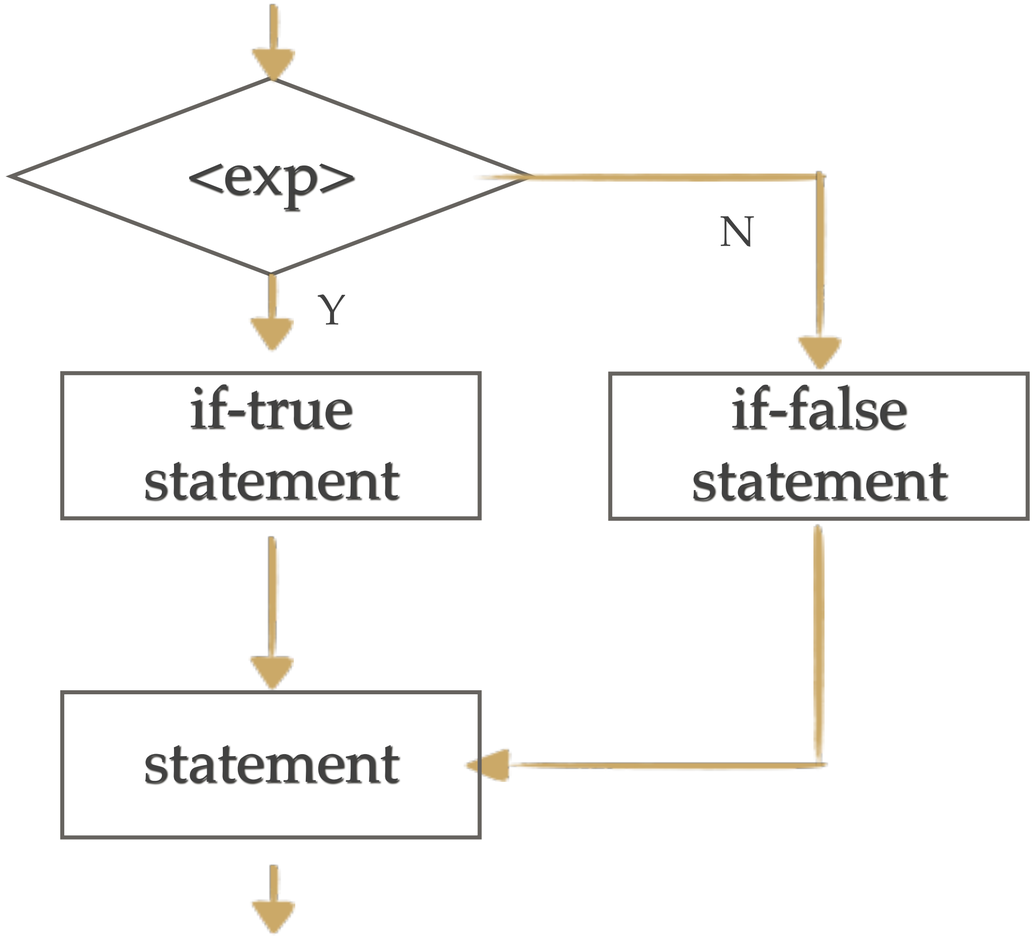
Example

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

int main() {
   int number;
   cout << "Enter an integer: ";
   cin >> number;

   if (number > 0)
       cout << "You entered a positive integer: " << number << endl;

   cout << "This statement is always executed.";
   return 0;
}
```

If-else statement

- Full if-else statement:
  - ```
    if (<conditional expression>) <if-true statement>;
    else <if-false statement>;
    ```
  - ```
    if (<conditional expression>) {
        <if-true statements>
    }
    else {
        <if-false statements>
    }
    ```

Flowchart



Example

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

int main() {
   int a, b, max;
   cout << "Enter two integer numbers: ";
   cin >> a >> b;
   if (a > b) {
       max = a;
   }
   else {
       max = b;
   }
   printf("The maximum value between %d and %d is %d\n", a, b, max);
   return 0;
}
```

Nested conditionals

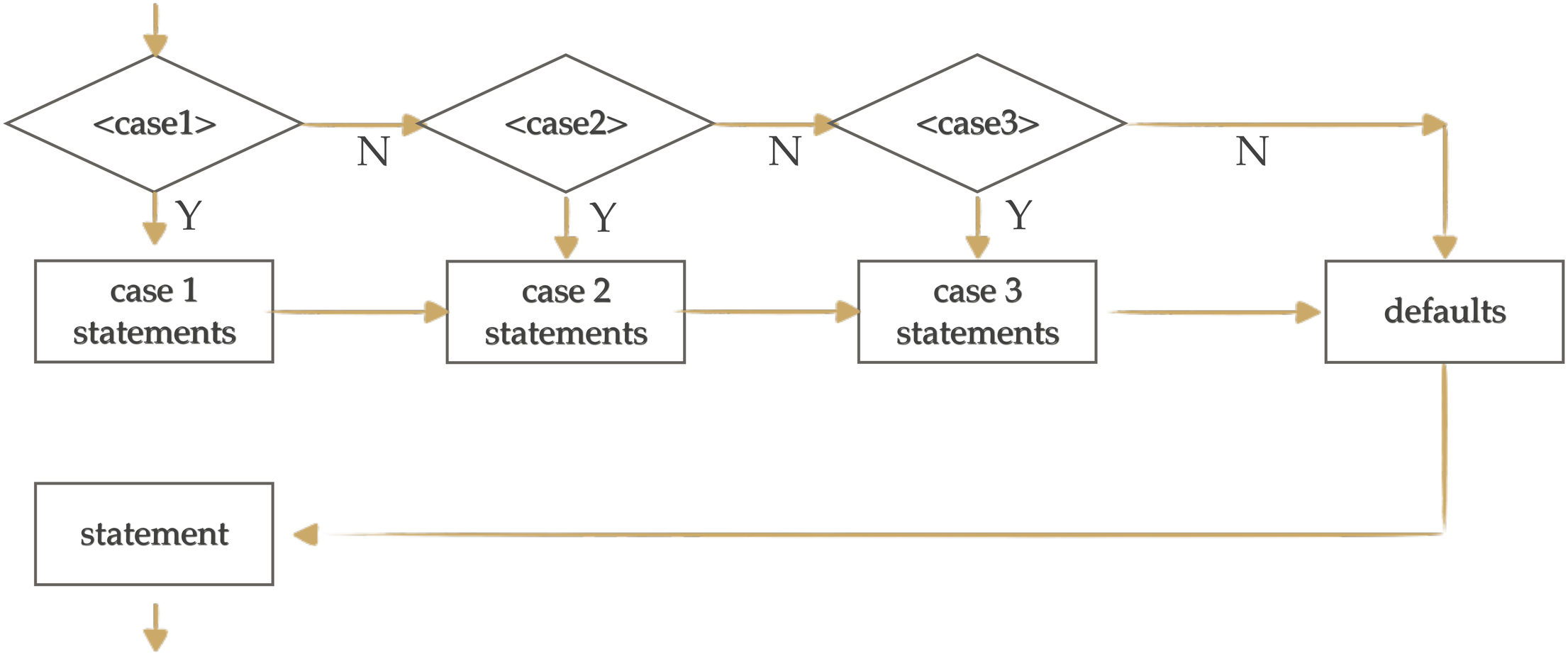
- Nested if-else statements
  - ```
    if (<exp>)                          // first check
          if (<exp>)                    // second check
              if (<exp>)                // third check
                  <statement>
              else <statement>
          else <statement>
    else if (<exp>) <statement>
          else if (<exp>) <statement>
               else <statement>
    ```
    ```

Nested conditionals

- Nested if-else statements: multi-way
  - ```
    if (<exp 1>) <statement 1>
    else if (<exp 2>) <statement 2>
    else if (<exp 3>) <statement 3>
    else <statement 4>
    ```
  - ```
    if (<exp 1>) <statement 1>
    else if (<exp 2>) <statement 2>
         else if (<exp 3>) <statement 3>
              else <statement 4>
    ```

Example

```
#include<iostream>
using namespace std;
int main() {
    float score;
    char grade;
    cout << "Enter a score [0 - 10]: ";
    cin >> score;
    if (score >= 8)
        grade = 'A';
    else if (score >= 6.5)
        grade = 'B';
    else if (score >= 5)
        grade = 'C';
    else if (score >= 4)
        grade = 'D';
    else
        grade = 'F';

    cout << "Your grade is " << grade << endl;
    return 0;
}
```

Conditional operator

- Syntax:
  - `<expression> ? <if-true expression> : <if-false expression>`
  - Equivalent to if-else statement but apply for expressions
- Example:
  - ```
    char outChar;
    outChar = a == 'c' ? 'C' : 'c';
    ```
  - ```
    float diff;
    diff = x > y ? x - y : y - x;
    ```

Example

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

int main()
{
   int a, b, max;
   cout << "Enter two integer numbers: ";
   cin >> a >> b;
   max = (a > b) ? a : b;

   printf("The maximum value between %d and %d is %d\n", a, b, max);
   return 0;
}
```

Switch statement

Switch statement

- A convenient way to write multi-way statement
- ```
  switch(<exp>) {
      case <value 1>: <statements>; break;
      case <value 2>: <statements>; break;
      …
      case <value N>: <statements>; break;
      default: <statements>;
  }
  ```
  ```

Flowchart


<case1>
N
<case2>
N
<case3>
N
Y
Y
Y
case 1 statements
case 2 statements
case 3 statements
defaults
statement

Example

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

int main() {
   char o;
   float num1, num2;

   cout << "Enter an operator (+, -, *, /): ";
   cin >> o;
   cout << "Enter two operands: ";
   cin >> num1 >> num2;

   switch (o) {
   case '+':
      cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;
      break;
   case '-':
      cout << num1 << " - " << num2 << " = " << num1 - num2 << endl;
      break;
   case '*':
      cout << num1 << " * " << num2 << " = " << num1 * num2 << endl;
      break;
   case '/':
      cout << num1 << " / " << num2 << " = " << num1 / num2 << endl;
      break;
   default:
      cout << "Error! operator is not correct";
      break;
  }

return 0;
}
```

What happens if no break statement?

```
switch(<exp>) {
     case <value 1>: <statements>;
     case <value 2>: <statements>;
     …
     case <value N>: <statements>;
     default: <statements>;
}
```

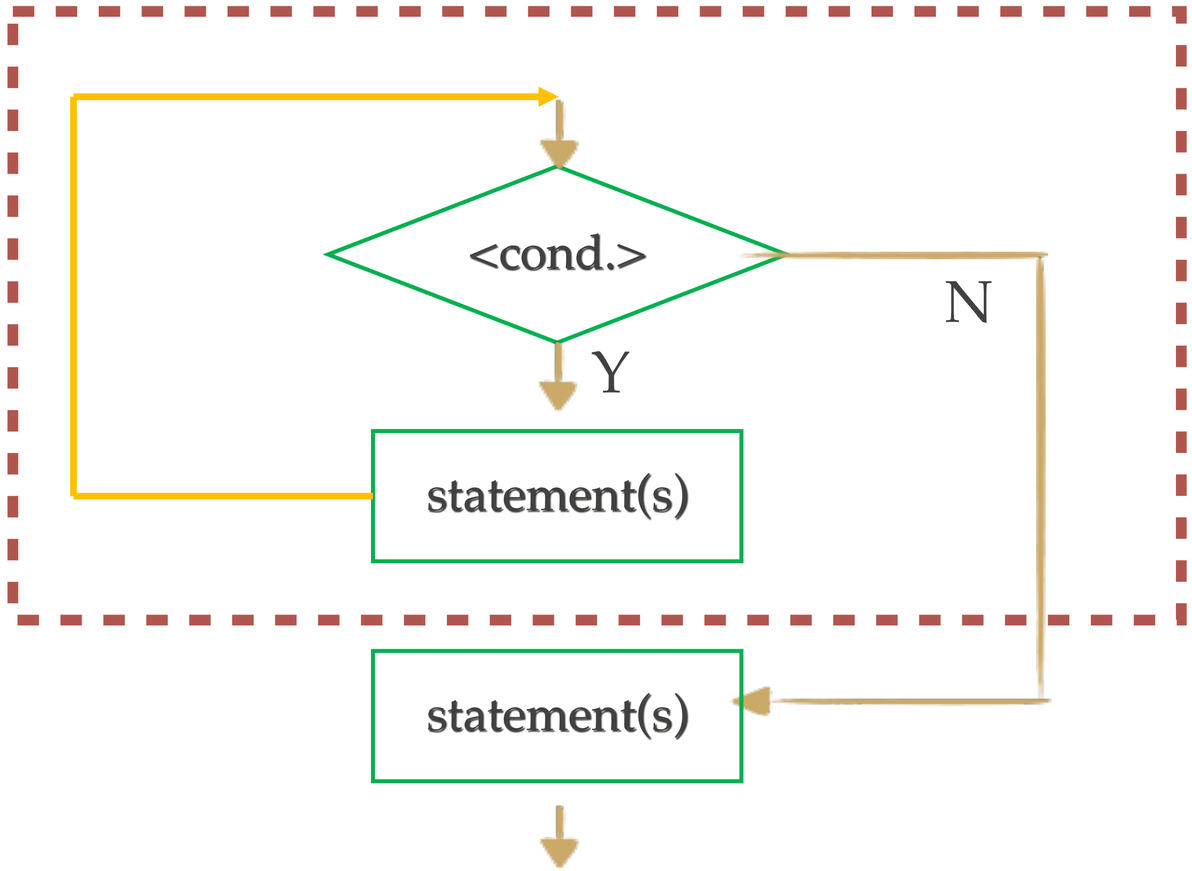
If break statement is not used, all cases after the correct case is executed.

Loop statements

while statement

- Why do we need iterations?
  - Waiting for something to happen
  - Operate on several objects
    - List, array of objects
    - String

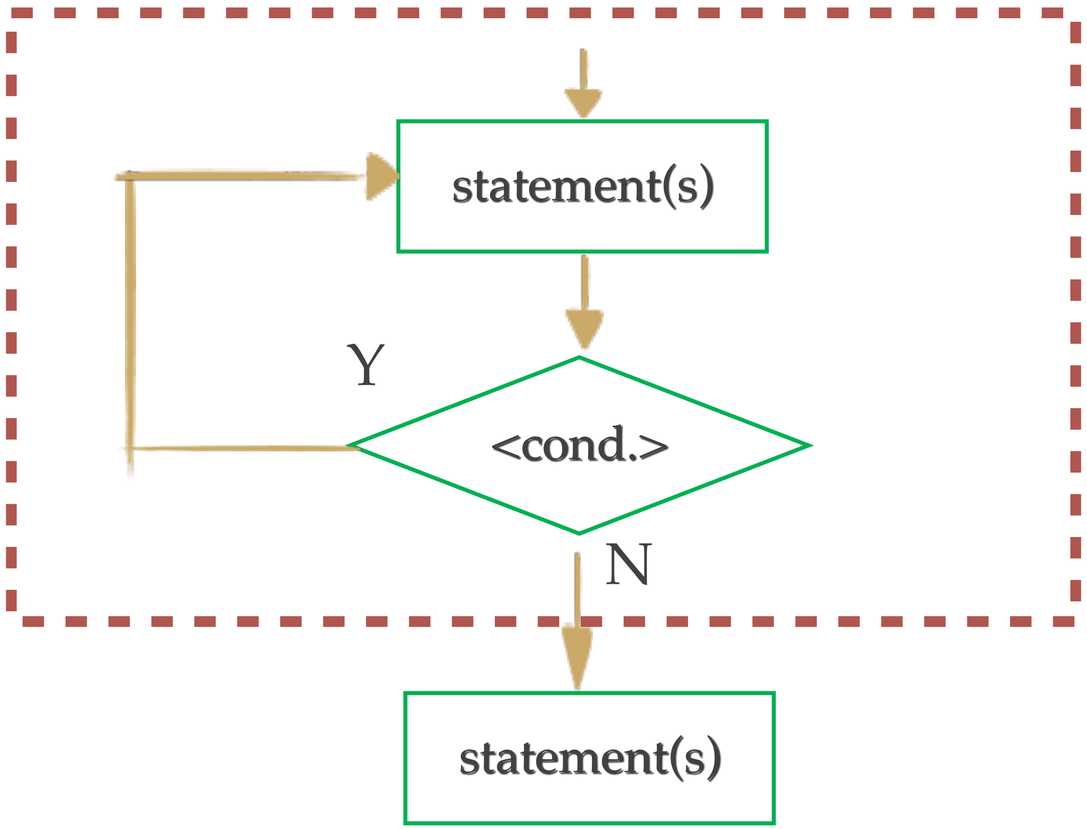
while loop

- Syntax:
  - `while (<condition>) <statement>;`
  - ```
    while (<condition>) {
        <statements>;
    }
    ```
    ```

while loop

- Flowchart



Example

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

int main() {
   int counter = 0;
   while (counter < 10) {
      cout << counter << " ";
      counter++;
   }
   cout << endl;
   return 0;
}
```

Do-while loop

- Syntax:
  - `do <statement> while (<condition>);`
  - ```
    do {
        <statements>;
    } while (<condition>);
    ```
    ```

Do-while loop

- Flowchart



Example

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

int main() {
   int counter = 0;
   do {
      cout << counter << " ";
      counter++;
   } while (counter < 10);
   cout << endl;
   return 0;
}
```

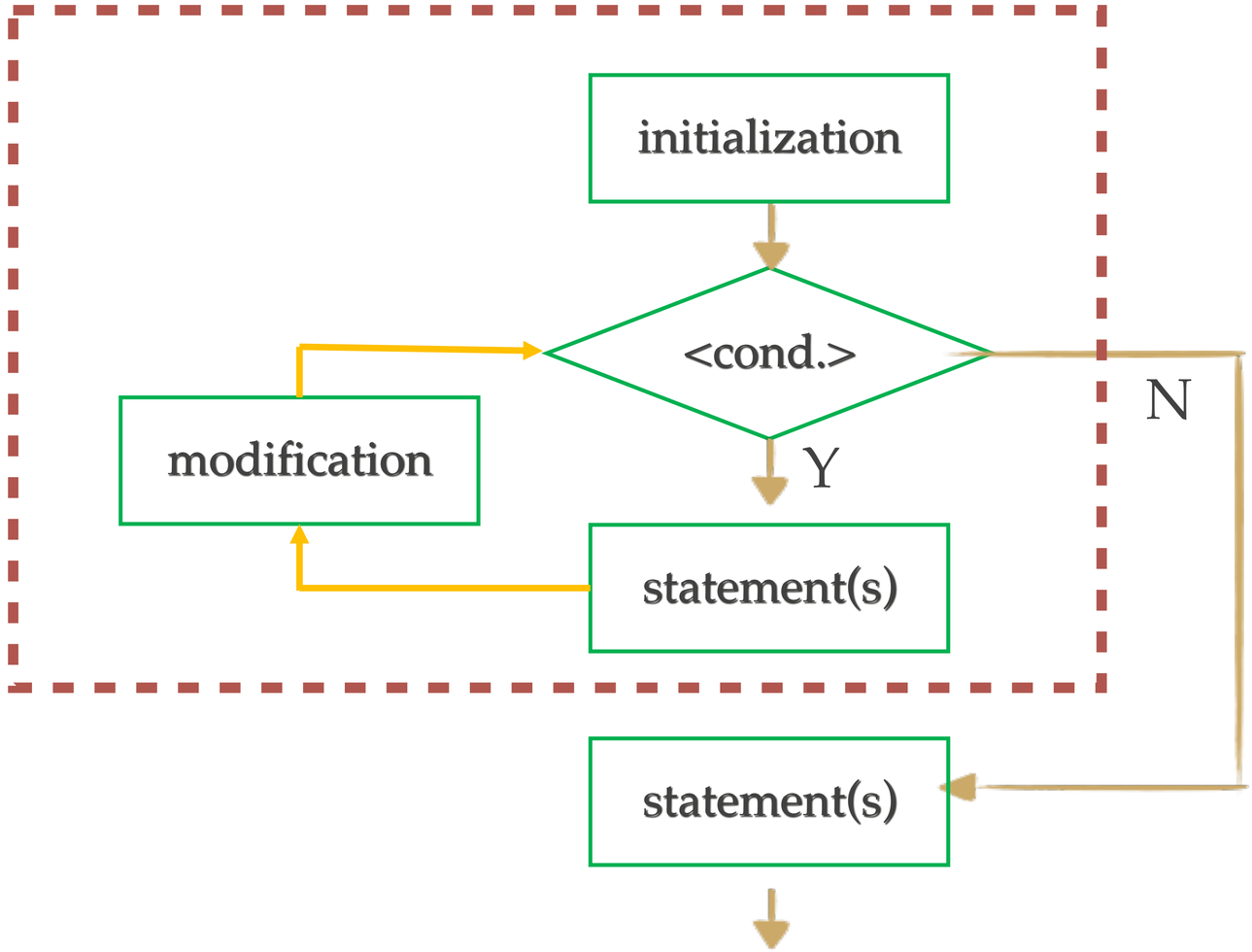
while statement

- Note:
  - Remember to initialize variables in the condition expression before entering the while statement (at least you know what will happen when you check the condition).
  - Do not forget stopping condition.
  - Take care of counters.

for statement

- Why do you need for statement?
  - Just another way to write iteration/loop structure!
  - Counting is a frequent activity
  - for: a specialised loop that package the following tasks in a statement
    - Initialise a counter variable
    - Modify the counter
    - Check complete condition

for statement

- for loop:
  - `for (<initialization>; <condition>; <modification>) <statement>;`
  - ```
    for (<initialization>; <condition>; <modification>) {
        <statements>;
    }
    ```
    ```

for statement

- Flowchart



for statement

- Initialization: set value for the counter
  - Declare one or many counters (same type) and init them at once
  - Initialize many counters if needed
- Condition: a boolean expression that must be evaluated at each loop
- Modification: change value of the counter at each loop

Example

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

int main() {
  for (int i = 0; i < 10; i++)
     cout << i << " ";
  i--;
  cout << endl;
  return 0;
}
```

for statement

- Note that initialization and modification can contain multiple statements separated by commas.

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

int main() {
   int i, j;
   for (i = 5, j = 10; i + j < 20; i++, j++) {
       cout << "i + j = " << (i + j) << '\n';
   }
   return 0;
}
```

Infinite loops

```
while (100) {
}
while (true) {
}
do {

} while (-20);

for (;;) {
}
```

Exit loops

- The two most commonly used are:
  - break: will end the loop and begin executing the first statement that comes AFTER the end of the loop.
  - continue: force the next iteration to be executed.

Example

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

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

   cout << "\nBroke out of loop at count = " << count << endl;
   return 0;
}
```

Example

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

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

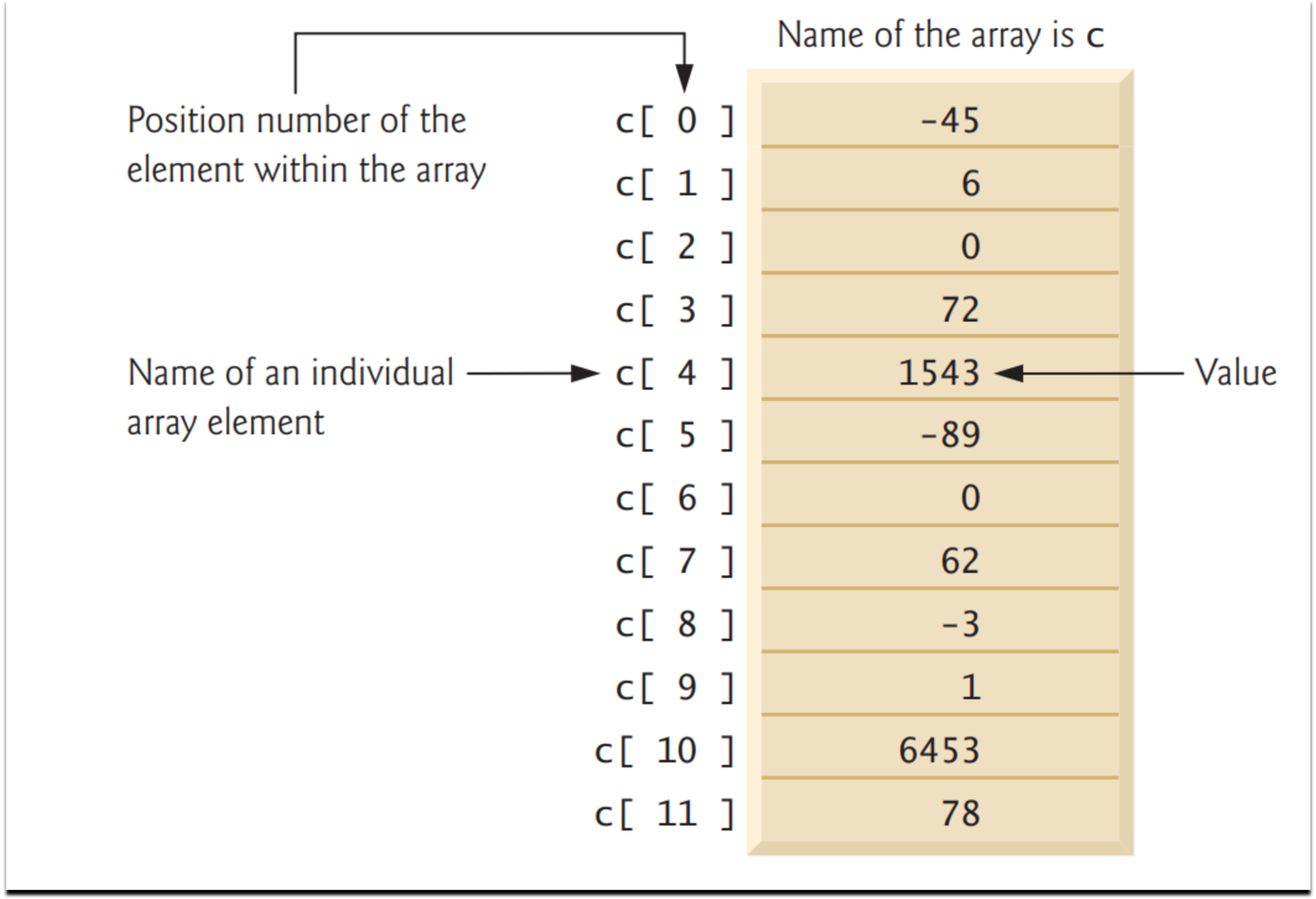
   cout << "\nUsed continue to skip printing 5" << endl;
   return 0;
}
```

Nested loop

- A loop can be nested inside a loop.
- ```
  while (<condition 1>) {
      <statements>;
      while (<condition 2>) {
          <statements>;
          while (<condition 3>);
      }
      <statements>;
  }
  ```
  ```

Example

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

int main() {
   int i, j;

   for (i = 2; i<100; i++) {
      for (j = 2; j <= (i / j); j++)
          if (!(i%j)) break; // if factor found, not prime
      if (j >(i / j)) cout << i << " is prime\n";
   }

   return 0;
}
```

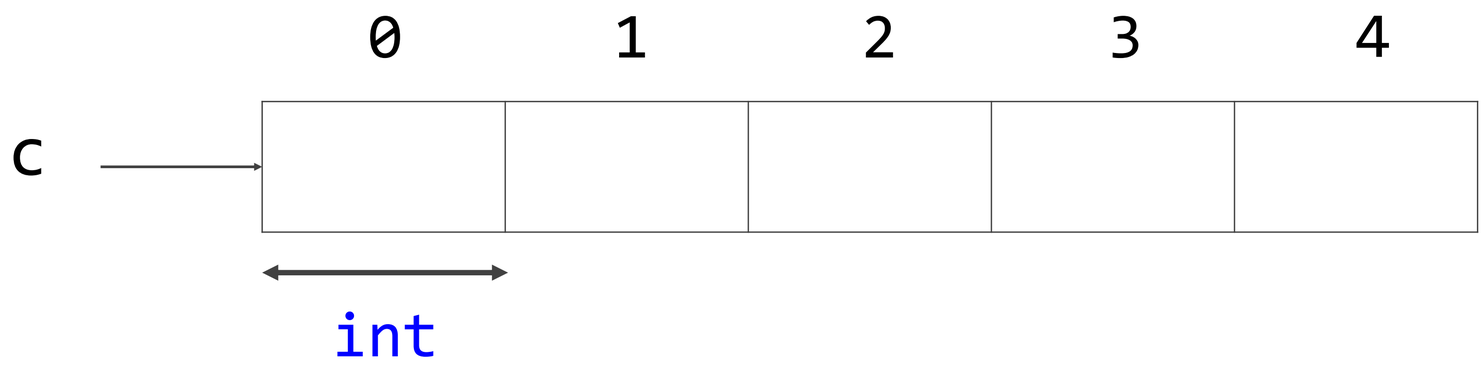
Array

Array


Name of the array is c
Position number of the element within the array
Name of an individual array element
Value
c[0]
c[1]
c[2]
c[3]
c[4]
c[5]
c[6]
c[7]
c[8]
c[9]
c[10]
c[11]
-45
6
0
72
1543
-89
0
62
-3
1
6453
78

Declaring Arrays

- <type> *arrayName* [*arraySize*];
  - E.g: `int c[5];`



Initializing Arrays

- One by one
  - E.g: `c[4] = 10;`
- Using a loop
  - E.g: `for (int i = 0; i < 5; i++) c[i] = 0;`
- Declaring with an initializer list
  - E.g: `int c[5] = { 10, 20, 30, 40, 50 };`

Accessing the values of an array

- The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:
  - name[index]
- Example:

```
int a = 2;
c[0] = a;
c[a] = 75;
b = c[a + 2];
c[c[a]] = c[2] + 5;
```

Example

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

int main() {
   int c[10];
   for (int i = 0; i < 10; i++) c[i] = 0;

   cout << "Element" << setw(13) << "Value" << endl;
   for (int i = 0; i < 10; i++)
       cout << setw(7) << i << setw(13) << c[i] << endl;

   return 0;
}
```

Example

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

int main() {
   int c[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

   cout << "Element" << setw(13) << "Value" << endl;
   for (int i = 0; i < 10; i++)
       cout << setw(7) << i << setw(13) << c[i] << endl;

   return 0;
}
```

2D Array

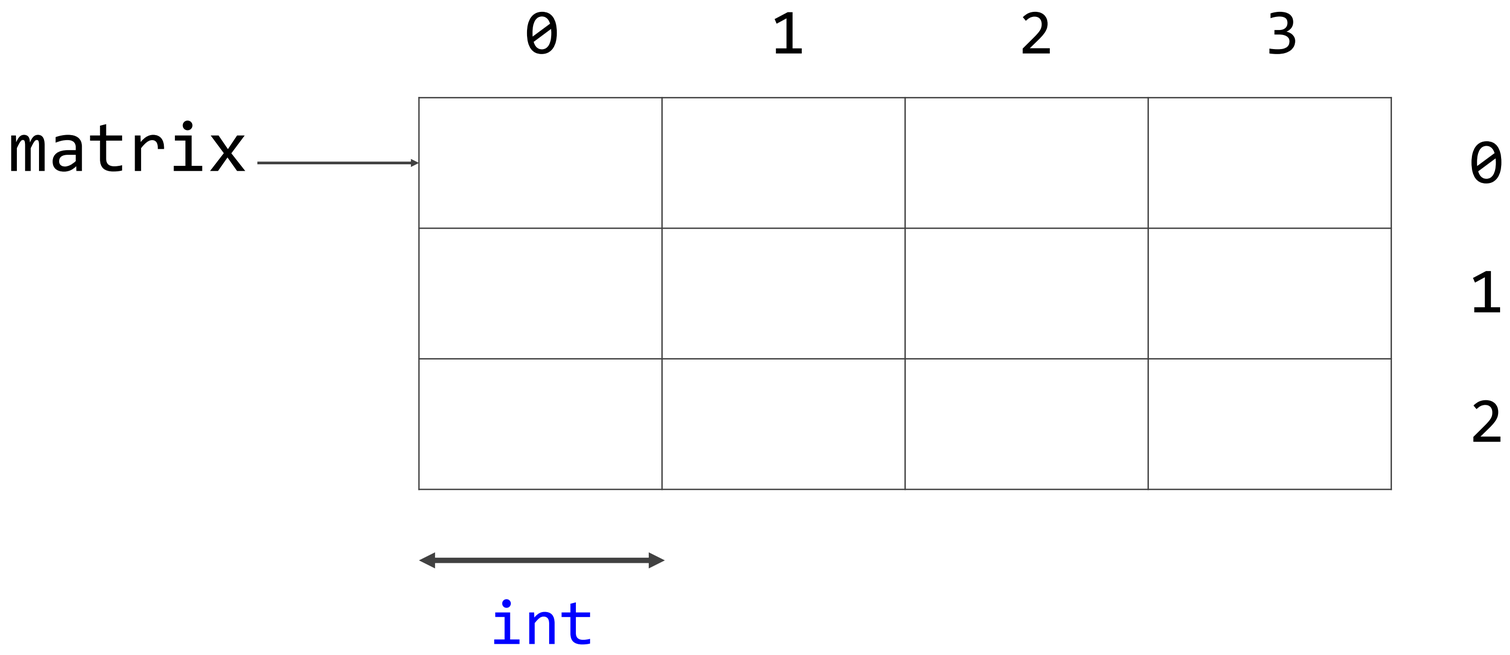
2D Array

- The elements in a 2-dimensional array are indexed for access, using two indexes:
  - *Row* index starts from 0 to (*Number of rows* - 1)
  - *Column* index from 0 to (*Number of columns* - 1)

	0	1	2	3
0	10	20	30	40
1	50	60	70	80
2	90	100	110	120

Declaring 2D Arrays

- <type> *arrayName* [*rowSize*][*columnSize*];
  - E.g: `int matrix[3][4];`



Initializing Arrays

- One by one
  - E.g: `c[2][3] = 4;`
- Declaring with an initializer list
  - E.g:

```
int a[3][4] = {{ 10, 20, 30, 40},
               {50, 60, 70, 80},
               {90, 100, 110, 120}};
```

- Using a loop
  - E.g:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++)
        matrix[i][j] = 0;
    }
}
```

Accessing the values of an 2D array

- The syntax is the same as accessing an array, except that you have to locate the *row index* and *column index* of the element:
  - name[*rowIndex*][*columnIndex*]
- Example:

```
int r = 0;
int c = 2;
a[r][c] = 1;
a[a[0][2]][c - 1] = 3;
```

Example

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

int main() {
   int a[3][4];
   for (int i = 0; i < 3; i++)
       for (int j = 0; j < 4; j++)
          a[i][j] = 0;

   cout << “rowIdx" << setw(13) << “colIdx" << setw(13) << "Value" << endl;
   for (int i = 0; i < 3; i++)
       for (int j = 0; j < 4; j++)
          cout << setw(7) << i << setw(13) << j << setw(13) << a[i][j] << endl;

   return 0;
}
```

String <cstring>

String definition

- In C, a string is an array of characters in a string and ends with a special character namely `'\0'`.
- Given the string **"Hello World":**
  - Length: 12 characters
  - Number of elements in array:

0	1	2	3	4	5	6	7	8	9	10	11
'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'

String declarations

- With given size and no initialization:
  - `char s1[MAX_LEN];`
- With given size and character-by-character initialized:
  - `char s1[MAX_LEN] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\’};`
- With given size and string literal initialized:
  - `char s1[MAX_LEN] = “Hello”;`
- Or even without given size (initialization required):
  - `char s1[] = “Hello”;`

String input & output

To retrieve a string, we can use:

- `gets_s()` in `<cstdio>`
- Or `getline()` in `<iostream>`

To output a string, just use `cout` for the array like other variables.

Example

```
#include<iostream>
#include<cstring>
#define MAX_LENGTH 50
using namespace std;

int main()
{
    char userName[MAX_LENGTH];
    cout << "Tell me your name? ";
    getline(cin, userName);
    cout << "Hello " << userName << "!\n";
    return 0;
}
```

String functions

- **strlen()** return the length of the string
- **strcat_s()** concatenates a copy of str2 to str1
- **strcmp()** compares two strings
- **strcpy_s()** copys contents of str2 to str1

Example

```
/* strcat example */
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char str[80];
    strcpy_s(str, "these ");
    strcat_s(str, "strings ");
    strcat_s(str, "are ");
    strcat_s(str, "concatenated.");
    cout << str << endl;
    cout << "The length of the string: " << strlen(str)<<endl;
    return 0;
}
```

String <string>

Introduction

- C++ also provides an additional class called `std::string` that allows manipulation of strings more easily and safely.
- The whole class `std::string` is packaged in the library `<string>` so we have to add `#include <string>` beforehand.
- Note: `<string>` versus `<cstring>`/`<string.h>`

String declarations (<string>)

- No initialization:
  - `string s1;`
- String literal initialized:
  - `string s2 = “Hello”;`

String input & output (<string>)

The same as `<cstring>`

Example

```
#include<iostream>
#include<string>
#define MAX_LENGTH 50
using namespace std;

int main()
{
    string userName;
    cout << "Tell me your name? ";
    getline(cin, userName);
    cout << "Hello " << userName << "!\n";
    return 0;
}
```

String methods, operators (<string>)

- **length()** return the length of the string
- **substr()** return a substring of the string
- **+ operator** concatenates a copy of str2 to str1
- **>, <, ==, != operator** compares two strings
- **= operator** copys contents of str2 to str1

Example

```
/* string concatenation example */
#include <iostream>
#include <string>
using namespace std;

int main() {
    string str;
    str += "these ";
    str += "strings ";
    str += "are ";
    str += "concatenated.";
    cout << str << endl;
    cout << "The length of the string: " << str.length() <<endl;
    return 0;
}
```

Function

Function

- You should never write monolithic code
  - Difficult to write correctly.
  - Difficult to debug.
  - Difficult to extend.
- Hard to maintenance
- Non-reusable
- Nonsense!

Function

- Definition: a group of statements that is given a name, and which can be called from some point of the program.

- All C++ functions (except the special case of the main function) should have:
  - A declaration (khai báo): this is a statement of how the function is to be called.
  - A definition (định nghĩa): this is the statement(s) of the task the function performs when called

Function Declaration

- A function is declared with the syntax:

```
returnVariableType functionName(parameter1, parameter2,
...,parameterN);
```

- Note the semi-colon at the end of the statement.

Function Definition

- A function is defined with the syntax:

```
returnVariableType  functionName(parameter1, parameter2,
...,parameterN)
{
    statement A;
    statement B;
}
```

Function

- C++ functions can:
  - Accept parameters, but they are not required
  - Return values, but a return value is not required
  - Can modify parameters, if given explicit direction to do so

Function: No Return, No Parameters

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

void displayMessage();

int main() {
   displayMessage();
   return 0;
}

void displayMessage() {
   cout << "Welcome to CO1011!\n";
}
```

Functions with Parameters

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

void displaySum(int, int);

int main() {
   displaySum(5, 10);
   return 0;
}

void displaySum(int num1, int num2) {
   cout << num1 << “+” << num2 << “=“ << num1 + num2;
}
```

Functions with Return

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

int computeSum(int, int);

int main() {
   int sum = computeSum(5, 10);
   printf("%d + %d = %d\n", 5, 10, sum);
   return 0;
}

int computeSum(int num1, int num2) {
   return num1 + num2;
}
```

Inline Function

- Similar to function, except that the compiled code will be inserted where we call inline functions.
- Purpose: improve performance
- ```
  inline <return type> <function name>(<parameters>) {
      <function body>
  }
  ```
  ```

Pass Parameters

- Parameters: there are two ways to pass parameters to a function
  - Value: the value will be copied to local variable (parameter) of the function
  - Reference (only in C++): the parameter is associated with passed variable
    - Passing by reference refers to passing the address of the variable
    - Any change in the parameter affects the variable

Example

```
#include<iostream>
using namespace std;
void increment(int &input);

int main() {
   int a = 34;
   cout << "Before the function call a = " << a << "\n";
   increment(a);
   cout << "After the function call a = " << a << "\n";
   return 0;
}

void increment(int &input){
   input++;
}
```

Arrays as Parameters

- There are three methods for passing an array by reference to a function:
  - `returnType functionName(variableType *arrayName)`
  - `returnType functionName(variableType arrayName[arraySize])`
  - `returnType functionName(variableType arrayName[])`

Example

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

void arrayAsPointer(int *array, int size);

int main() {
    const int size = 3;
    int array[size] = { 33,66,99 };
    arrayAsPointer(array, size);
    return 0;
}

void arrayAsPointer(int *array, int size) {
    cout << setw(5);
    for (int i = 0; i<size; i++)
        cout << array[i] << " ";
    cout << "\n";
}
```

Default Parameters

- Default arguments are used in place of the missing trailing arguments in a function call.
- Default arguments must be the **rightmost** (trailing) arguments in a function's parameter list

Example

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

int computeBoxVolume(int width, int length = 1, int height = 1);

int main() {
   cout << "The default box volume : " << computeBoxVolume(10) <<endl;
   cout << "The second box volume : " << computeBoxVolume(10,20) <<
endl;
}

int computeBoxVolume(int length, int width, int height) {
   return length * width * height;
}
```

Function Overloading

- Several functions can have the same name: overloaded functions.
- Functions with the same name have different **signatures** (prototypes).
- Function signature: **name** + **parameter list**
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call

Example

```
double div(double a, int b);
int div(int a, int b);
double div(int a, double b);

float add(float a, float b) {
   return a + b;
}

int add(int a, int b) {
   return a + b;
}

double add(int a, double b) {
   return (double)a + b;
}
```

Enumerated type

- Define a list of possible values of a type
  - `enum <type name> {<name of possible values>};`
  - `enum [<type name>] {<name of possible values>} <variables>;`
- Example:
  - ```
    enum Color {Red, Orange, Yellow, Green, Blue, Violet};
    Color c = Yellow;
    cout << “Yellow color has value: ” << c << endl;
    ```
    ```

Enumerated type

- Define a list of possible values of a type
  - `enum <type name> {<name0 = value0>, <name1 = value1>, …};`
  - `enum [<type name>] {<name0>} <variables>;`
- Example:
  - ```
    enum Color {Red = -1, Orange = 2, Yellow = 8, Green = 3, Blue,
    Violet};
    Color c = Blue;
    cout << “Blue color has value: ” << c << endl;
    ```
    ```

Why enums are used in C++ programming?

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

enum suit {club = 0, diamonds = 10, hearts = 20, spades = 3} card;

int main()
{
   card = club;
   cout << "Size of enum variable " << sizeof(card) << " bytes.\n";
   return 0;
}
```

How to use enums for flags?

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

enum designFlags {
    BOLD = 1,
    ITALICS = 2,
    UNDERLINE = 4
};

int main()
{
   int myDesign = BOLD | UNDERLINE;
   cout << myDesign;
   return 0;
}
```

Macro definition

Macro definition

- #define/#undef: preprocessor directives
- Extend across single line of code
- No semicolon “;” at the end
- Use “\” to write the define instruction with multiple lines

Macro definition

- Define constants: #define <identifier> <replacement>
  - `#define MAX_LENGTH 50`
  - `#define MY_STRING  “This is a constant string”`
  - `#define pi_2       3.14159/2`
  - `#define pi_2       1.570785`
- Constants variables:
- Const pi = 3.1415928
  - `const float x_2 = x / 2;// x_2 cannot be changed`

Macro definition

- More examples:
  - ```
    #define NORMALIZE_FACTOR  50
    …
    float fx = sum / NORMALIZE_FACTOR;
    ```
  - ```
    #define sub(a, b) …
    …
    float x = 0.5 * sub(z + 3.9, y + f(t));
    ```

Macro definition

- Macros:
  - `#define sub(a, b) a - b`
  - `#define sub(a, b) (a - b)`
  - `#define sub(a, b) ((a) - (b))`
  - ```
    #define swap(a, b, c) {\
                    a = b;\
                    b = c;\
                    c = a;\
    }
    ```
    ```

Example

```c
#include <stdio.h>

#define swap(t, x, y) {t tmp = x; x = y; y = tmp;}

int main() {
  int x = 10, y = 2;
  swap(int, x, y);

  printf("%d %d\n", x, y);
  return 0;
}
```

Macro definition

- Special operators:
  - #: create a string literal that contains the argument
    - ```
      #define text(a) #a
      …
      cout << text(Be careful) << endl;// print “Be careful”
      ```
  - ##: concatenate two arguments
    - ```
      #define glue(a, b) a ## b
      …
      glue(c, out) << “Weird way to write code\n”;// but acceptable
      ```

Preprocessor directives

Preprocessor directives

- Conditional Pre-processor directives:
  - `#define, #undef, #ifdef, #ifndef, #else, #elif, #endif`
  - E.g.:
    - ```
      int foo(float a, double b) {
      #ifdef __MSC_VER
          return a * 3.14159 - sqrt(b * a);
      #else
          return a * 3.14159 + b * b;
      #endif
      }
      ```
      ```

Preprocessor directives

- Conditional Pre-processor directives
  - Library headers (*.h, *.hpp):
    - ```
      #pragma once
      // library definition
      ```
    - ```
      #ifndef __MY_LIBRARY_H__
      #define __MY_LIBRARY_H__
      // library definition
      #endif
      ```

Preprocessor directives

- Power of macros and preprocessor directive
  - One definition fit all
  - Flexible, portable
  - Open source community

Indents, coding style

- Use indents to enhance your code
  - Easy to manage flow of code
  - Easy to read code
- Coding requires skills and the programmer, in most of cases need to follow rules of their community.