2. C++ basic syntax

Today:

Using some of C++ basic constructs:

- Namespace, `std`, `using`
- Variables, types, type safety
- Strings
- Performing input/output (IO)
- Program organization

C++ comments and const

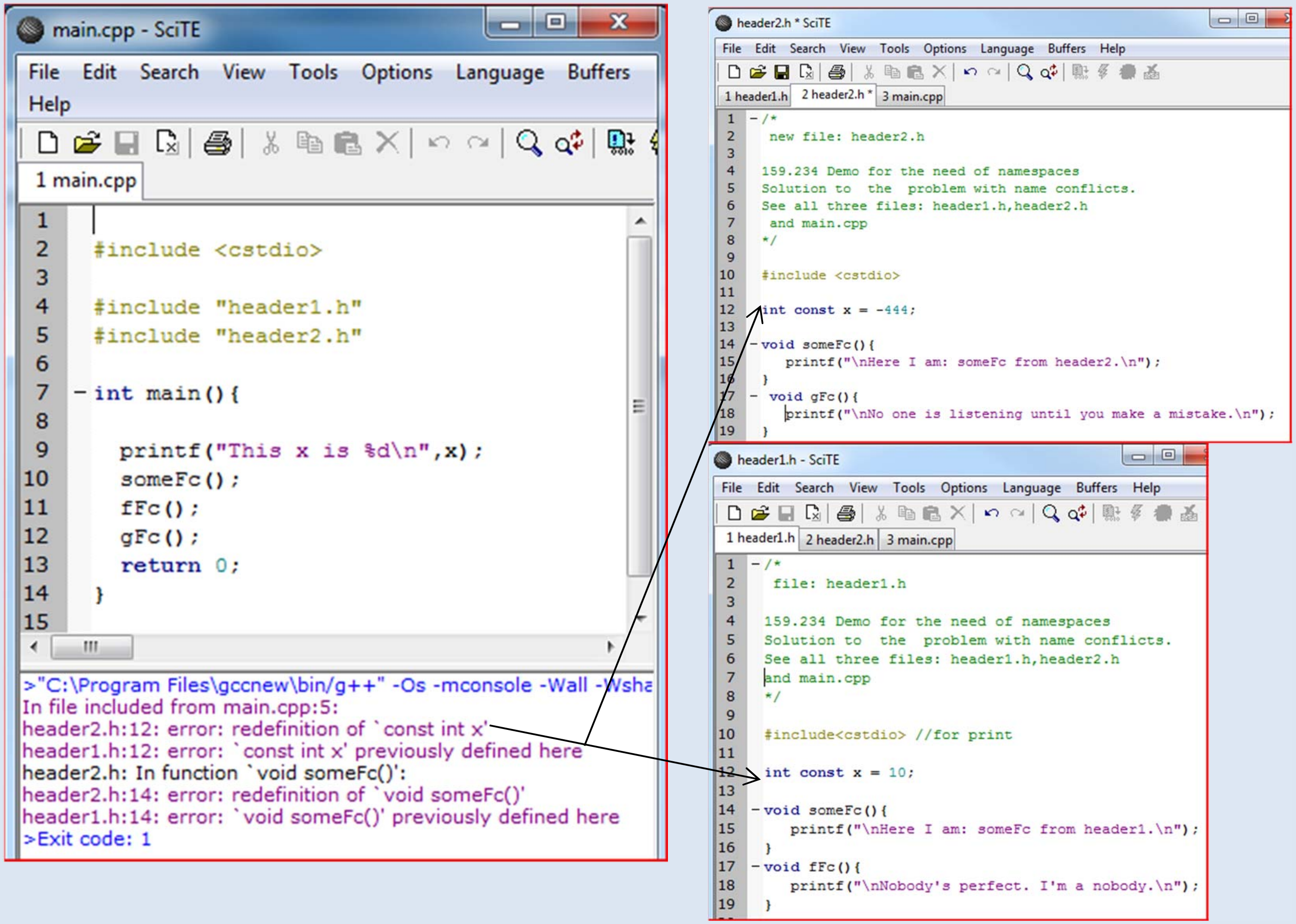
- Traditional C comments still supported
- C++ adds rest of line comment

Example:

```
/*  File name: a1.cpp
159.234 program demo for types
Author Calude E.
modified by  Mickey Mouse.
*/
const int SIZE;  //number of elements in array
float total;     //total cost per customer
int customerID;  ~~// id of customer~~
```

Problem: name conflict in global space

E
X
A
M
P
L
E



```

#include <cstdio>

#include "header1.h"
#include "header2.h"

int main(){

   printf("This x is %d\n",x);
   someFc();
   fFc();
   gFc();
   return 0;
}

```

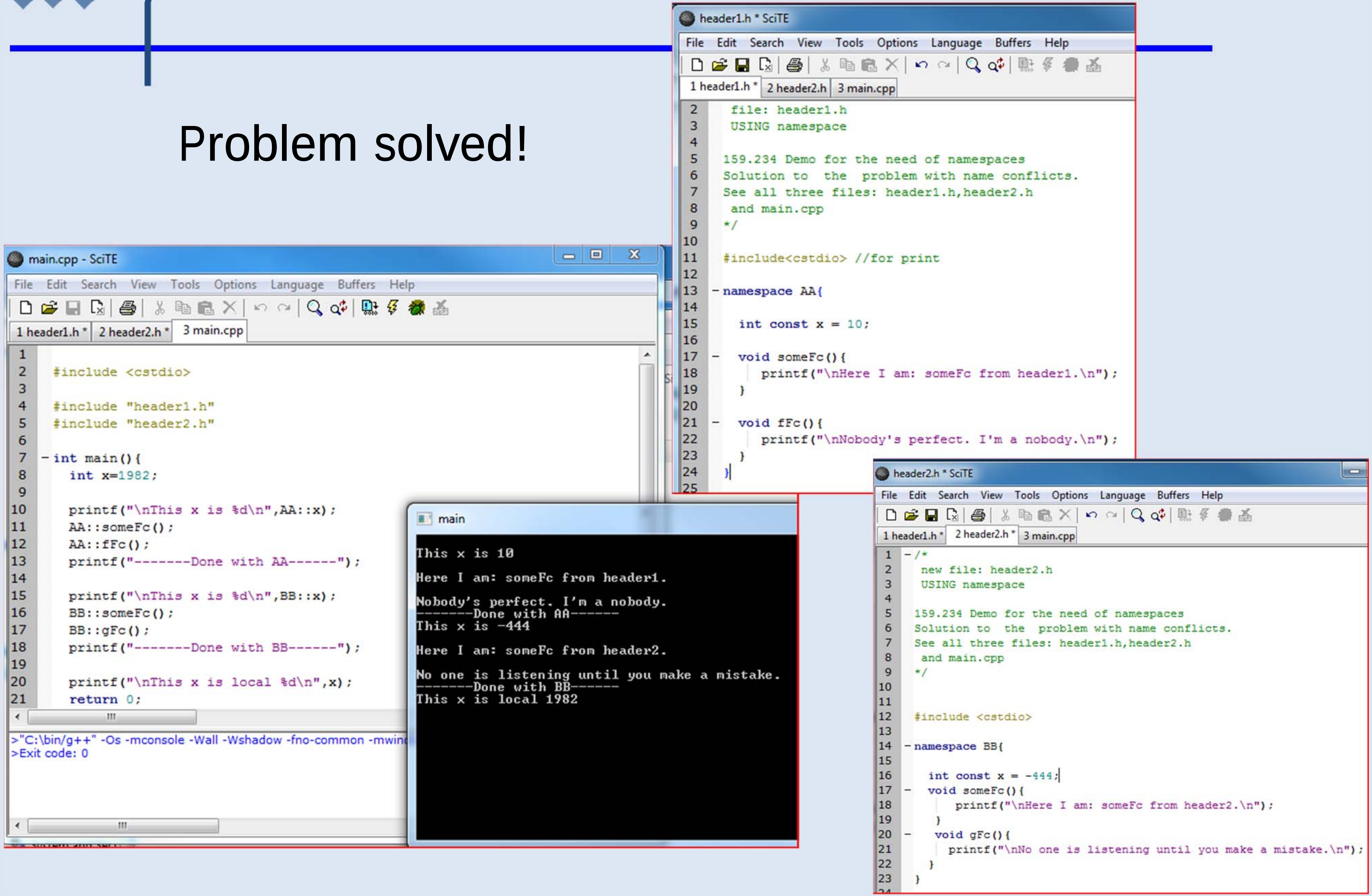
```
>"C:\Program Files\gccnew\bin/g++" -Os -mconsole -Wall -Wsha
In file included from main.cpp:5:
header2.h:12: error: redefinition of `const int x'
header1.h:12: error: `const int x' previously defined here
header2.h: In function `void someFc()':
header2.h:14: error: redefinition of `void someFc()'
header1.h:14: error: `void someFc()' previously defined here
>Exit code: 1
```

```
/*
 new file: header2.h

159.234 Demo for the need of namespaces
Solution to  the  problem with name conflicts.
See all three files: header1.h,header2.h
 and main.cpp
*/

#include <cstdio>

int const x = -444;

void someFc(){
   printf("\nHere I am: someFc from header2.\n");
}
 void gFc(){
   printf("\nNo one is listening until you make a mistake.\n");
}
```

```
/*
 file: header1.h

159.234 Demo for the need of namespaces
Solution to  the  problem with name conflicts.
See all three files: header1.h,header2.h
and main.cpp
*/

#include<cstdio> //for print

int const x = 10;

void someFc(){
   printf("\nHere I am: someFc from header1.\n");
}
void fFc(){
   printf("\nNobody's perfect. I'm a nobody.\n");
}
```

Solution: Namespace

Problem solved!



header1.h:

```
 file: header1.h
 USING namespace

159.234 Demo for the need of namespaces
Solution to  the  problem with name conflicts.
See all three files: header1.h,header2.h
 and main.cpp
*/

#include<cstdio> //for print

namespace AA{

  int const x = 10;

  void someFc(){
     printf("\nHere I am: someFc from header1.\n");
  }

  void fFc(){
     printf("\nNobody's perfect. I'm a nobody.\n");
  }
}
```

header2.h:

```
/*
 new file: header2.h
 USING namespace

159.234 Demo for the need of namespaces
Solution to  the  problem with name conflicts.
See all three files: header1.h,header2.h
 and main.cpp
*/


#include <cstdio>

namespace BB{

  int const x = -444;
  void someFc(){
     printf("\nHere I am: someFc from header2.\n");
   }
   void gFc(){
    printf("\nNo one is listening until you make a mistake.\n");
  }
}
```

main.cpp:

```

#include <cstdio>

#include "header1.h"
#include "header2.h"

int main(){
  int x=1982;

  printf("\nThis x is %d\n",AA::x);
  AA::someFc();
  AA::fFc();
  printf("-------Done with AA------");

  printf("\nThis x is %d\n",BB::x);
  BB::someFc();
  BB::gFc();
  printf("-------Done with BB------");

  printf("\nThis x is local %d\n",x);
  return 0;
```

```
>"C:\bin/g++" -Os -mconsole -Wall -Wshadow -fno-common -mwin
>Exit code: 0
```

```
This x is 10

Here I am: someFc from header1.

Nobody's perfect. I'm a nobody.
-------Done with AA------
This x is -444

Here I am: someFc from header2.

No one is listening until you make a mistake.
-------Done with BB------
This x is local 1982
```

Scope resolution operator

- Namespaces are used to avoid name conflicts
- Extends C’s single, global namespace to allow program elements to be items of various namespaces
- Names must only be unique within a namespace
- Provides for programmer-defined scope

- `AA::someFc()`

SCOPE resolution **operator**

using keyword

Items in a namespace may be accessed

- *explicitly* with scope-resolution operator (this is `::`)

  example: **`AA::x`**
- *implicitly* with `using` statement
  - Providing access to a single item:

    `using AA::x;`
  - Providing access to all items:

    `using namespace AA;`

In 159.234 we will use

```
        using namespace std;
```

C++ header files

- C++ headers provide all standard components in namespace std, hence
  - `using namespace std;`
- **do not** have “.h”
  - `#includes<iostream>`
- the C headers renamed drop *.h* suffix and add `c` prefix
  - Instead of: `#include<stdlib.h>`
  - Use: `#include<cstdlib>`

Variables

Local variables can be declared **anywhere** within a function: the only condition is to **declared it before using it**.

When the enclosing { } ends, the variable is meaningless (out of scope).

```
if (x > 0){
   y = x;
   int z = x*x + x;
   w = y + z*z;
 }
```

Variables

- Defining local variables when they're needed produce more
  - readable,
  - maintainable and
  - more efficient code

than defining variables at the beginning of compound statements.

- Some rules for defining local variables:
  - Local variables should be created at `intuitively right' places
  - In general, variables should be defined in such a way that their scope is **as limited** and **localized** as possible.
  - Global variables should be avoided..

Initialization, assignment and incremenation

```
int a = 7;          // initialization
int a{7};           // initialization
a = 9;              // assignment
a = a+a;            // assignment
a += 2;             // increment a's value by 2
++a;                // increment a's value (by 1)
```

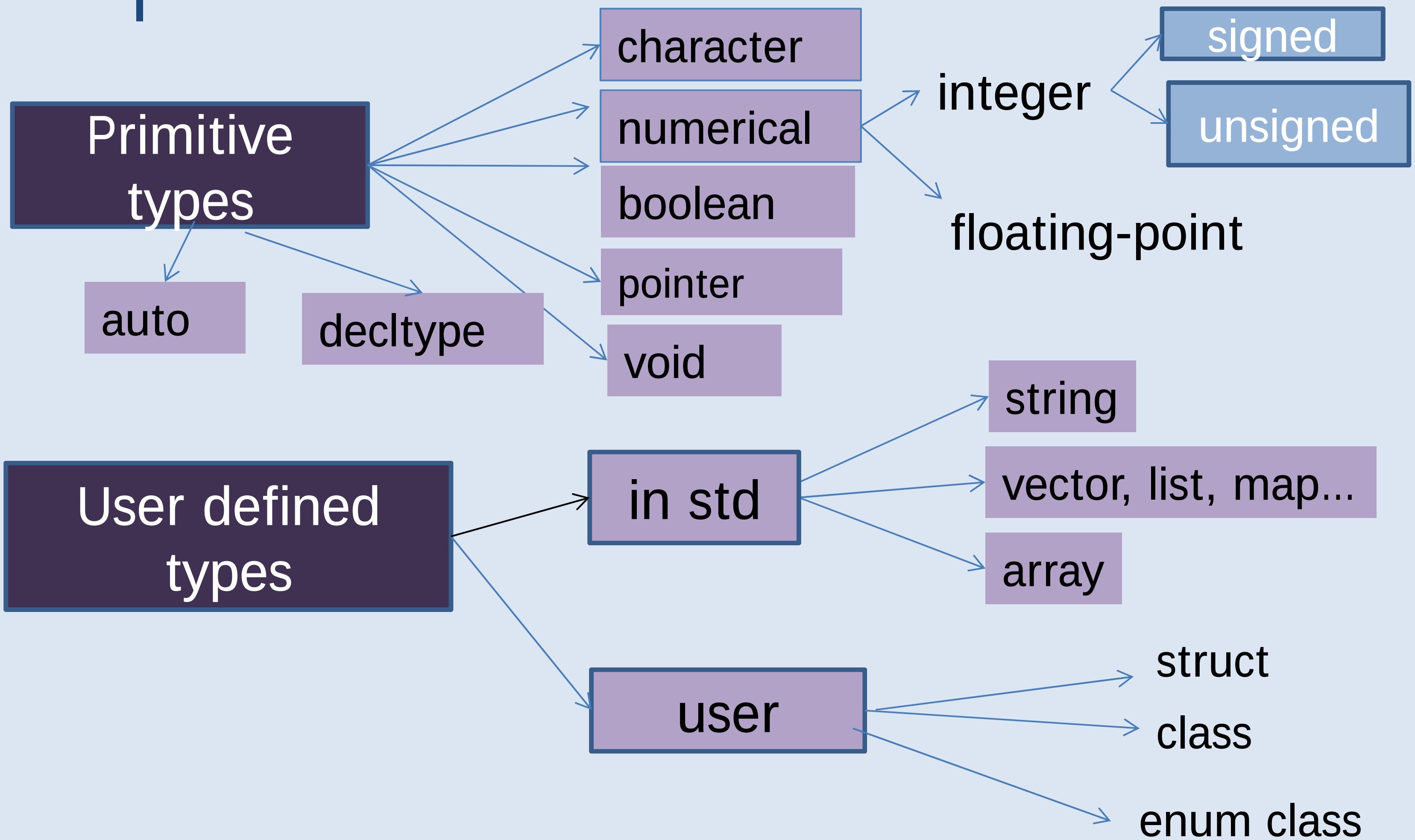
C++ types

Every variable has a type.
The type of a variable determines what operations we can do on it.

- all C’s types: `char, int, float, double`, `bool` and `void`
- C++ adds new types
  - **`wchar_t`**: wide character type, used to

    represent “large” character sets (UNICODE)
  - `auto`, `decltype`... (C++11)

Overview of C++ types



Primitive types

character

numerical

integer

signed

unsigned

floating-point

boolean

pointer

void

auto

decltype

User defined types

in std

string

vector, list, map…

array

user

struct

class

enum class

C++ is strongly typed.

Type safety

- **Language rule: type safety**
  - Every object will be used **only according to its type**
    - A variable will be used only after it has been initialized
    - Only operations defined for the variable's declared type will be applied
    - Every operation defined for a variable leaves the variable with a valid value
- Ideal: static type safety
  - A program that violates type safety will not compile
    - The compiler reports every violation (in an ideal system)
- Ideal: dynamic type safety
  - If you write a program that violates type safety it will be detected at run time
    - Some code (typically "the run-time system") detects every violation not found by the compiler (in an ideal system)

Type safety

- **Type safety is a very big deal**
  - Try very hard not to violate it
  - “when you program, the compiler is your best friend”
    - But it won’t feel like that when it rejects code you’re sure is correct
- C++ is not (completely) statically type safe
  - No widely-used language is (completely) statically type safe
  - Being completely statically type safe may interfere with your ability to express ideas
- C++ is not (completely) dynamically type safe
  - Many languages are dynamically type safe
  - Being completely dynamically type safe may interfere with the ability to express ideas and often makes generated code bigger and/or slower

C++ strings

- **Strings** are specific constructs that are geared towards processing sequences of characters.
- Use: `#include<string>`
- string declaration:

```
    string myStr;
```

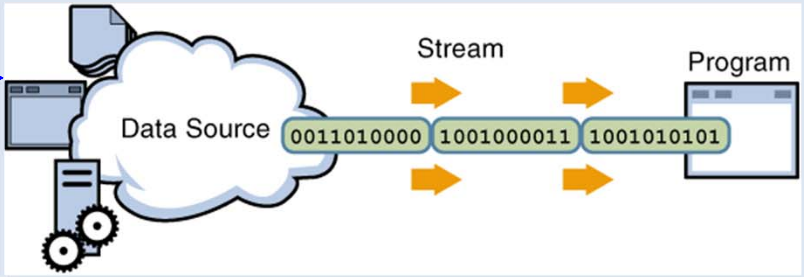
- string declaration and initialization:

```
  string myStr1 {"Hello there"};
  string myStr2 = "Hello again";
```

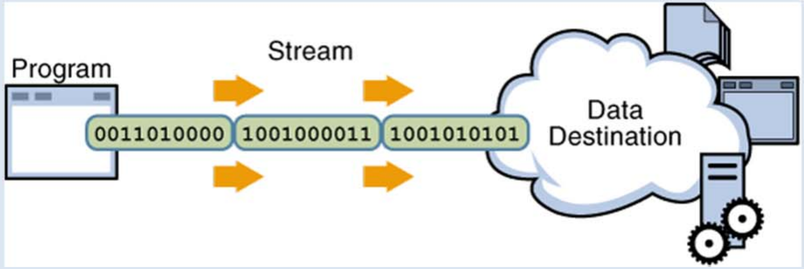
Input/Output streams

C++ uses the “stream” notion to communicate with the outside world.
A **stream** can be thought of as a sequence of bytes of infinite length that is used as a buffer to hold data that is waiting to be processed.

Data source/ destination can be any of: input devices-keyboard, disk file, memory array,..



Input stream



Output stream

Output

Remember C way:

```
 #include<stdio.h>
..
  printf(“%s %d\n",”Hello World”,i);
```

C++ way:

```
#include<iostream>
using namespace std;
..
cout << "Hello World" << i << endl;
```

< < is insertion operator

Input

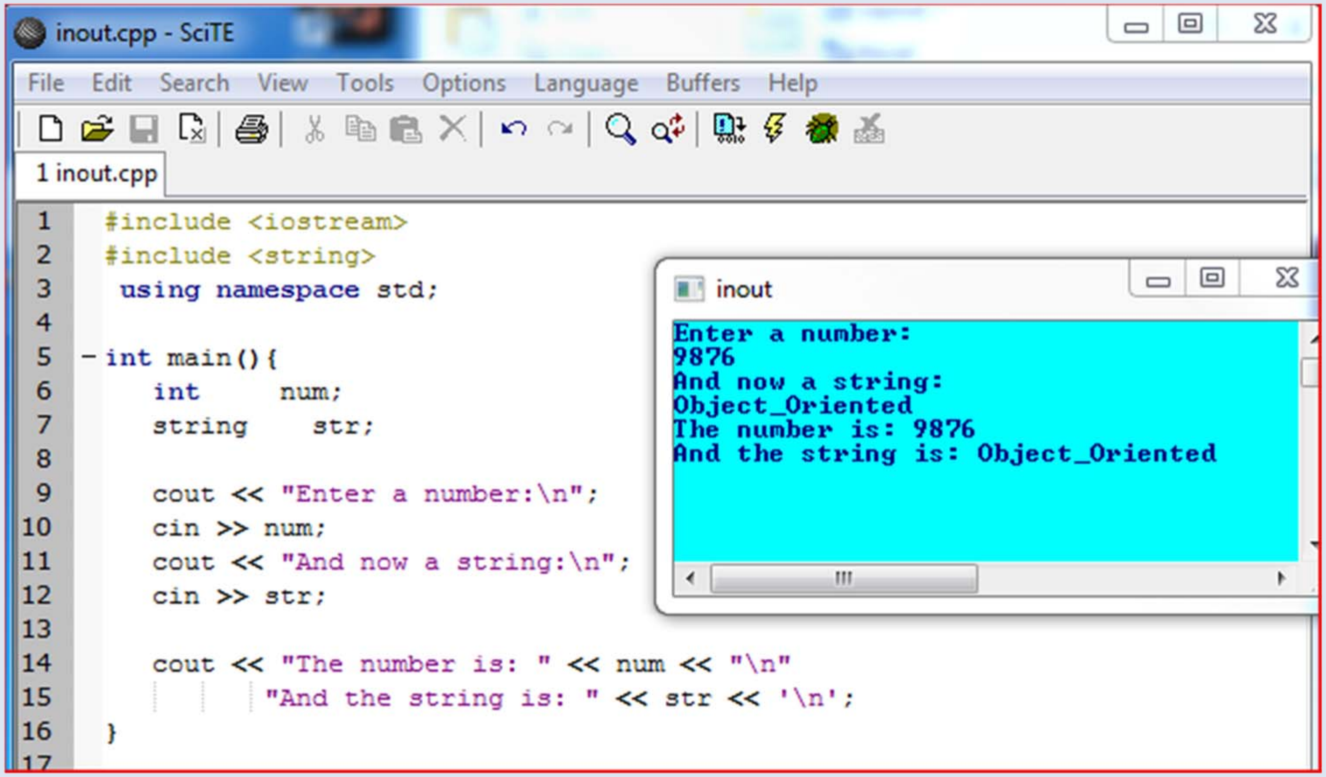
```
cin >> i;   //extraction operator
```

Ignores leading whitespace characters (blank, tab, newline, etc)

Separates one input value from another by the occurrence of

- ✓At least one whitespace character or,
- ✓A character that cannot be part of the variable being formed.

cout, cin



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

int main(){
    int      num;
    string    str;

    cout << "Enter a number:\n";
    cin >> num;
    cout << "And now a string:\n";
    cin >> str;

    cout << "The number is: " << num << "\n"
           "And the string is: " << str << '\n';
}
```

```
Enter a number:
9876
And now a string:
Object_Oriented
The number is: 9876
And the string is: Object_Oriented
```

cout, cin

Some **advantages** of using streams are:

•Using insertion and extraction operators is type-safe.

• The insertion and extraction operators may be extended . This cannot be done with printf for example.

• Streams are independent of the media they operate upon

The *iostream library* has a lot more to offer than just cin, cout.

C++ strings

- String can be output as any other type:

```
string s = "hello world";
cout << s;
```

- two ways to input strings:
  - using extraction operator - strips white space and assigns **the first** “word” to the string variable
  - using **`getline`** function –`getline(cin,str)`: assigns all characters to `str`, up to newline (not included)
  - do not mix `cin` and `getline` in the same program!

	user types:	What is in s?
`cin >> s ;`	Nice try	
`getline(cin, s);`	Nice try	

Program organization

```
//comments: Authors, program’s task, date,
//anything you want to let the user of your //program know
#include <file1>
using namespace std;

//------const global variables here------
constexpr int AGE = 5

//------function prototypes---------------
void func1(int, int);
int func2();

//-------------------------------
int main(){
   //code here. . .
   return 0;//optional in gcc
}
//--------function definitions here----------
void func1(int n1, int n2){
  //function body here. . .
}
//----------------------------------
int funct2(){
   return 5*20;
}
```

Summary

- The standard streams are declared in the header file `iostream`.
- The streams `cout`, `cin` and `cerr` are **objects**.
- The stream `cin` extracts data from a stream and copies the extracted information to variables (e.g., `num` in the above example) .
- The operators which manipulate `cin`, `cout` and `cerr` (i.e., `>>` and `<<`) manipulate variables of different types.

  `cout << num` results in the printing of an integer value,

  `cout << "Enter a number"` results in the printing of a string.
- The extraction operator (`>>`)
  - performs a **type** safe assignment to a variable.
  - **skips all white space** characters preceding the values to be extracted.

Next: Functions