High Level Programming

C++ Basics

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Project organization

- C++ inherits most of C's syntax
- It is almost always implemented as a compiled language
- Filename have extensions
  - .cpp for source files
    - Sometimes .cc, .cxx
  - .h for header files
    - Sometimes .H, .hh, .hpp, .hxx, .h++

"Hello world": Version 1

- A C++ version of the "Hello world" program that uses the standard library stream

#include <iostream> is a preprocessor directive in C++ that tells the compiler to include the contents of the <iostream> header file in your program.

```
#include <iostream>

int main(int argc, char ** argv) {
  int v1, v2;

  std::cout << "Enter two Values: ";
  std::cin >> v1 >> v2;
  std::cout << "Sum: " << v1+v2 << std::endl;

  return 0;
}
```

C++ libraries do not need ".h"

std:: is a namespace (see ahead)

- Enter two Values: 26 12
- Sum: 38

I/O Streams

- The I/O library define four objects
  - cin
    - An istream to read from standard input (normally the keyboard)
  - cout
    - An ostream for standard output, usual output (normally the screen)
  - cerr
    - An ostream for standard error , i.e., errors and warnings (not buffered)
  - clog
    - An ostream for standard logs, i.e., general information about the execution (buffered)

C style:
scanf (…) - fscanf (stdin, …)

C style:
printf (…) - fprintf (stdout, …);

C style:
fprintf (stderr, …);

Console Output

Correspond to the C printf

```
std::cout << expression;
```

- Sends expression to the standard output
  - We do not need the type of the objects
    - It is done automatically by the compiler
    - The output operator can handle different types
  - Multiple "<<" can be chained together
  - The
    - **std::endl** is the pre-defined "end of line " ('\n')
      - Each new line flushes the buffer
    - **std::flush** flushes the buffer
    - **std::ends** write a null and then flushes the buffer

Examples

```
std::cout << "We have " << classNum << " classes!";

std::cout << "I am " << age << " years old!" << std::endl;


std::cout << "hi!" << std::endl;

std::cout << "hi!" << std::flush;

std::cout << "hi!" << std::ends;
```

No newline

Newline

Write "hi!" then a new line, then flushes the buffer

Write "hi!" then flushes the buffer

Write "hi!", then a null, then flushes the buffer

Console Input

Corresponding to the C scanf

```
std::cin >> expression;
```

- Reads input from console & stores in variable
  - We do not need the type of the objects
    - It is done automatically by the compiler
    - The input operator can handle different types
  - Multiple "**>>**" can be chained together
  - Usually preceded by a prompt message to the user
  - It is difficult to detect invalid input
  - Read single strings not entire lines

Examples

```
int age, weight;

std::cout << "Enter age and weight: ";
std::cin >> age >> weight;
```

The first input goes into age and the second one into weight

```
std::cout << "Enter age and weight: ";
cin >> age;
cin >> weight;
```

Equivalent to the previous one

```
string s;
cin >> s;
```

Work for all types and for files (strings will be introduced in unit 4)

```
int sum=0, value=0;
while (std::cin >> value)
  sum *= value;
```

The input operator returns std::cin which is invallied (i.e., false) when we hit end-of-file or an invalid input (i.e., wrong type)

Namespace

The concept is common to the C language

- A namespace is region that provides a scope to the identifiers
  - Namespaces are used to organized code into logical groups
  - They prevent name collisions when the code includes multiple libraries
- The prefix **std::** indicates that the names are defined inside the namespace **std**
  - Libraries separate their symbols (function and variables) into namespaces
  - We need to indicate where symbol comes from by using **namespace::** before the symbol

Using declaration

- The "using" declaration

```
using std::cin;
```

  - Says that we want to use the name **cin** from the namespace **std**
    - Brings symbols from the library's namespace into the global scope of program
  - Each using declaration introduces a single namespace member at a time
  - Do not use **using** inside header files
    - To avoid automatic naming everytime the header is included

Using Directive

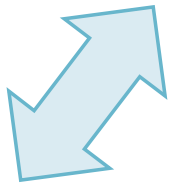
- The "using" directive

```
using namespace std;
```

  - Says that we want to use the namespace **std**, i.e., it takes all the names from a specific namespace visible without quantification
    - We retain no control over which names are made visible
    - We do not control name collision problems inherent in using multiple libraries

Example

```
std::cout << ... << std::endl;
std::cin >> …
std::cerr <<
```



Need to be placed in all cpp files (even for multi-file projects)

```
using std::cin;
using std::cout;
using std::endl;

cout << ... << endl;
cin >> ...
cerr << ...
```

```
using namespace std;

cout << ... << endl;
cin >> ...
cerr << ...
```

"Hello world": Version 2

- A C++ version of the "Hello world" program that uses the standard library stream and namespaces

```
#include <iostream>

using namespace std;

int main(int argc, char ** argv) {
  int v1, v2;

  cout << "Enter two Values: ";
  cin >> v1 >> v2;
  cout << "Sum: " << v1+v2 << endl;

  return 0;
}
```

Arithmetic types

Notice: Minimum size versus sizeof

Type	Meaning	Minimum Size
bool	Boolean	NA
char	Character	8
wchar_t	Wide Character	16
short	Short integer	16
int	Integer	16
long	Long integer	32
long long	Long integer	64
float	Single-precision floating point	6 significant digits
double	Double-precision floating point	10 significant digits
long double	Extended-precision floating point	10 significant digits

For other types (e.g., char16_t, char32_t, etc.) see documentation

Arithmetic types

- The type of an object defines the data contained and the operations available
- The type bool assumes values true and false
- To support internationalization, there are several character types
  - A char is big enough to hold all numeric values corresponding to a character in the set available in the hardware architecture used
  - The type wchar_t (char16_t, char32_t, etc.) are uwed for extended character set (i.e., the Unicode representation)

Arithmetic types

- For some type signed and unsigned versions are possible
  - Pay attention when signed and unsigned types are mixed

```
unsigned u = 19;
int i = -42;

std.cout << i + i << std::endl;
std.cout << u + i << std::endl;
```

i+i → -84

u+i → 4294967264
on 32-bit

Signed integer types can represent both positive and negative numbers, including zero. The range of values that a signed integer type can represent is divided evenly between positive and negative numbers, with one bit reserved for the sign.

Unsigned integer types can only represent non-negative numbers, including zero. All bits in an unsigned integer are used to represent the magnitude of the number, with no bit reserved for the sign. As a result they can represent a larger range of positive values compared to signed int types of same size

Literals

- A value, e.g., 10, is a literal because its values is self-evident
  - Every literal has a type
  - The form and value of the literal determine its value

Type	Meaning	Example
Integer	Decimal, Octal, Hexadecimal	20, 024, 0x23
Float	Include a decimal point and an exponent	3.1415, 1.2e3
Character	A character within single quotes	'a'
String	Zero or more character enclosed in quotation marks	"this is a string"
Escape sequence	Newline, carriage return, vertical tab, horizontal tab, etc.	\n, \r, \v, \t

Literals

- Literals may have prefix and suffix specifiers
  - Prefixes and suffixes can override the default type

Prefix	Meaning	Example
u	Unicode 16 character (char16_t)	u"this"
U	Unicode 32 character (char32_t)	U"this"
L	Hexadecimal (wchar_t)	L23

Postfix	Meaning	Example
U or u	Unsigned	42u
L or l	Long	3.1415l
LL or ll	Long long	46116860184273879O4LL

Variables

- Variables are defined in C++ as in C
  - Each variable has a type
  - Definitions can optionally provide an initial value
    - In C++ variables can be initialized in different ways
    - Variables defined outside a function are default-initialized to zero

```
int i = 0;
int i = {0};
int i{0};
int i(0);

int j{i}, k(i), l=i;
```

C++11
Curly braces, list initialization

Variables

- As in C variables, we may have
  - A declaration and a definition
    - A declaration makes a name known to the program
    - A definition creates the entity
    - Variables must be defined exactly once but can be declared many times
  - The keyword **extern** is used an in C
    - It defines a declaration that is not a definition
    - If the variable is initialized the declaration is also a definition

Definitions

```
int i;
extern j = 0;
```

Declarations

```
extern int i;
extern j;
```

Pointers

- Pointers are used as in C

```
int* ptr;
int *ptr;
int * ptr:
```

The size does not depend on the type because pointers are **memory addresses**

- A pointer is an object "pointing" to another object
- Pointers can be used to access objects
  - Pointers support pointer-arithmetic

```
int i = 10;
int *ptr = &i;

cout << i;
cout << *ptr;
```

Equivalent

When x is an array, the behavior of int* ptr = x; and int* ptr = &x; is slightly different due to the difference in how arrays and pointers are handled in C++.

```
int v[10];
int *ptr;

ptr = &v[0];
ptr++;
ptr=ptr+5;
```

Now ptr points to v[6]

int x[5] = {1, 2, 3, 4, 5};
int (*ptr)[5] = &x; // Pointer to an array of 5 integers. This is how we can create a pointer to an array of all 5 elements.

Pointers

- Pointers of type **void *** are special pointers that can hold the address of any object
- The main difference from C is that **NULL** (or 0) is **nullptr**
  - The use of **NULL** is deprecated (but accepted)
  - A pointer should always be set to **nullptr** before and after usage

introduced in C++11

  - **nullptr** is a literal with a type that can be converted to all other types

References

- A reference defines an alternative name for an object

```
int i = 0;
int &r = i;
int &wrong_r;
```

The variable r is an alias of variable i

Error: A reference must be initialized

  - A reference exists only if the object exists
    - Instead of copying the object we bind the reference to its initializer
    - A reference **cannot** be assigned to **nullptr** as it **must** be assigned to a **variable**
  - There is no standard implementation
    - Most of the times the compiler translate a reference into a pointer with de-referencing operations

References

C++, a reference is an alias or alternative name for an existing object. It allows you to access and manipulate the same object using different names. References provide a convenient way to work with variables without copying their values, and they are often used as function parameters and return types, as well as in variable declarations.

- We can define multiple references in a single definition
- References may be bound only to objects not to literals

If ref value changes the original val will also change so if int a = 3; int &ref = a then if ref = 44 so does a become 44.

```
int i = 0;
int &r1 = i, &r2 = i;

int &wrong_r = 10;

float f;
int &rf = f;
```

Error: Initializer must be an object

Once initialized it is bound to that variable for the lifetime. Can't be changed.

Error: The initializer must be an integer

Function Parameters: References are commonly used in function parameters to avoid the overhead of copying large objects. When a function parameter is declared as a reference, changes made to the parameter inside the function affect the original object passed as an argument.

Differences from Pointers: While references are similar to pointers in many ways, there are some key differences:
References cannot be null and must be initialized when declared.
Once initialized, references cannot be made to refer to a different object.
References do not have their own memory address; they simply provide an alias for an existing object.

Qualifiers

In C++ qualifiers are keywords that modify the behavior of variables, functions, and classes.There are several types of qualifiers in C++, including const, volatile, mutable, and constexpr.

- Any type (except functions and reference types) can be cv-qualified
  - Const-qualified
    - Used when we want to define an object we know cannot be changed
  - Volatile-qualified

modified unexpectedly by external factors, such as hardware or other parts of the program.

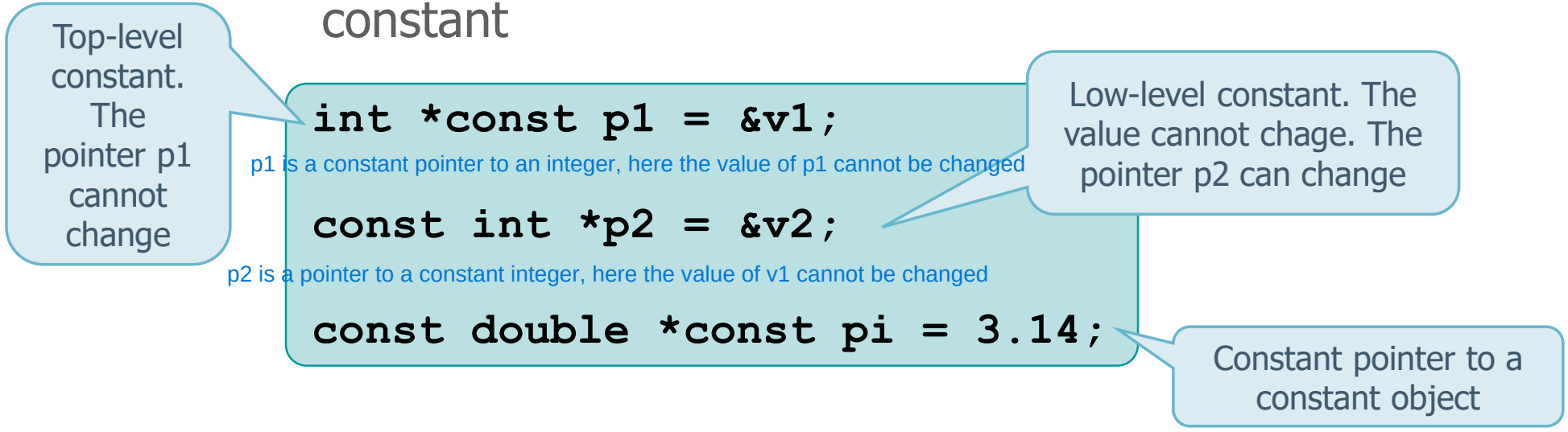
    - Used for objects that can be changed in ways outside the control of the program (e.g., a variable updated by the system clock)

```
const int c = 0;
volatile int v;
const volatile int cv;
volative const int vc;
```

cv-qualifiers can appear in any order, before or after the type

Qualifiers

- Constant objects cannot be modified once defined and initialized
  - Constant types can be used similarly to their non-constant version
  - When we have const with pointers, we can have
    - Top-level constants, i.e., the pointer is constant
    - Low-level constants, i.e., the referenced object is constant

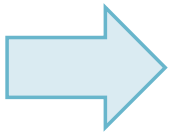


Qualifiers

- Volatile objects prevent the compiler to optimize the code segments involving the object
  - The compiler must suppose that the value of the object can be changed
  - Volatile should be avoided in most cases
    - **Deprecated** from C++20

```
int i = 100;
while (i == 100) {
   ...
}
```

If i is not modified, the compiler may be tempted to change the while construct into while(true)



```
volatile int i = 100;
while (i == 100) {
   ...
}
```

To avoid the compiler optimization we can define the object as volatile

The auto specifier

- From C+11 we can let the compiler figure out the type of an object by using the auto type specifier
  - A variable defined with auto must have an initializer
  - As with any other specifier we can define multiple objects using auto but all declaratios must havev ther same type

Although auto deduces the type of a variable at compile time, it still provides compile-time type checking. This means that the type inferred by auto must be compatible with the initializer expression, preventing type mismatch errors.

```
auto i = 0;
auto pi = 3.1415;
int &r = i;
auto j = 10, k = 20;
```

Correct

Error:
Inconsistent types

```
auto i = 0, pi = 3.1415;
```

The decltype specifier

- We can let the compiler deduces the type of an object from an expression using the decltype type specifier
  - A variable defined with decltype must have an initializer

```
decltype (f()) v = 0;

int i;
decltype (i + 3.1415) f;
```

Function f is not called, it is just used to define the type of v

```
int x = 10;
    decltype(x) y = 20; // declares y with the same type as x (int)
```

Reported for the sake of completeness. Not often used. You may ignore it !

Expressions

- There are no major differences between C and C++ operators

Type	Syntax	Example
Precedence and associativity		i+j*k = (i+(j*k))
Arithmetic operators	+, -, *, /, %	quotient = i/2; remainder = n%2;
Logical operators	!, &&, ||	
Relational operators	<, <=, ==, >=, >, !=	i==0 && j>0
Assignment		i=3; j=k=5;
Bitwise		i>>2
Etc.		

Type conversions

- In C++ some types are related to each other
  - We can use an object or value of one type when another one is expected
  - We say that we **convert** one type into the other
- In C++ we have
  - Implicit conversions
    - The ones carried out automatically without the intervention of the programmer
    - For example, in most expressions smaller types are promoted to larger type
  - Explicit conversions
    - The ones explicitly forced by the programmer

Type conversions

- C++ extends C in terms of explicit conversions
- C++ allows
  - C (old-type) conversions
  - C++ (new) named-cast conversions
    - static_cast
    - const_cast
    - reinterpreted_cast
    - dynamic_cast

Not used.

Run-time type identification:
It converts a pointer or a reference to a base type into a pointer or reference to a derived type

- The target of these operators is to provide a safer syntax to dangerous constructs

C (old-style) cast

- The C language resorts to cast operations when a type conversion is required

```
(<type>) <expression>
<type> (<expression>)
```

  - The C compiler
    - Does not perform any check about the accuracy or meaning of the cast
    - It only checks its syntactic feasibility

```
int i, j;
float f;

i = (int) f;
f = (float) j;
```

Depending on the types involved an old-style cast has the same behavior as a static_cast, const_cast, reinterpret_cast

C++ static cast

Safe Conversions:
static_cast is used for conversions that are safe and well-defined at compile time, such as converting between fundamental data types (like int to double) or performing upcasting and downcasting in inheritance hierarchies.

```
static_cast <new_type> (expression)
```

We are not only specifying the type but also the fact we wanna cast it

- It is used to cast related types
  - It performs the conversion and checks whether the conversion is acceptable
  - Converts the **expression** to the **new_type**
    - It is often used to convert a **larger** type into a **smaller** one
    - It is used to perform conversions the compiler will not generate automatically
    - It can be used to convert **void** pointers to pointers of another type

Example

```
int i = 42;
void *vp = &i;
int *ip = static_cast<int*>(vp);
```

From void * to int *

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

int main() {
   int i = 42, j = 31;
   double x = 3.14, y = 23.34;

   x = sum (i, j);
   y = sum (i, static_cast<int>(y));
   z = sum (static_cast<int>(x),static_cast<int>(y));
}
```

From double to int*

C++ const cast

```
const_cast <new_type> (expression)
```

- It converts a constant into a non-constant type
  - It casts away the **const**
  - Once we have cast away the const, the compiler will no longer prevent us from writing to the object
    - However, using a const_cast to write to a const object is undefined
- Most useful in the context of overloaded functions

Reported for the sake of completeness. Not often used. You may ignore it !

Example

```
int f1 (int *ptr) {
  return (*ptr+10);
}
...
int f2 (int *ptr) {
  *ptr = *ptr + 10;
  return (*ptr);
}
...
const int val = 10;
const int *ptr = &val;
int *ptrc = const_cast <int *> (ptr);


cout << f1 (ptrc);

cout << f2 (ptrc);
```

It can be used to pass constant data to a functioon that does not receive const.
Output: 20

It is undefined behavior to modifying a value which is initially declared as const.
Output: Undefined

C++ reinterpret cast

```
reinterpret_cast <new_type> (expression)
```

- It is used to convert unrelated types
  - It converts the underlying bit sequence representing **expression** as a value of **new_type** without any logic check
    - It forces a low-level reinterpretation of the bit pattern of its operand
    - It is used when we want to work with bits
      - It is a compile-time directive which instructs the compiler to treat the **expression** as if it had the type **new_type**
    - Usually, it does not generate any CPU instructions

IN c we can define a structure (struct) or a union which is very much like struct. Once we define the union the compiler reserves value for space

C++ reinterpret cast

- It is similar to the use of unions in C language

reinterpret_cast in C++ allows converting between unrelated pointer types or between pointer types and integral types, but it doesn't change the underlying memory layout.
union in C allows multiple members of different types to share the same memory location, enabling storage of different types of data in the same space.
Both can manipulate memory in unconventional ways, but reinterpret_cast is more flexible with pointer conversions, while union is specifically designed for shared memory storage of different types.

- It is inherently **machine dependent**
  - Not to use it unless strictly required
- Requires a complete understanding of the types involved in the operation
- Only a very restricted set of conversions is allowed
  - Invalid conversions usually lead to undefined behavior
  - It is undefined to access an object using an expression of a different type

Example

From pointer to integer
(a static casts would be an error)

```
int i = 10;

uintptr_t v1 = reinterpret_cast<uintptr_t>(&i);

cout << "Value of &i: 0x" << v1 << '\n';
```

For now consider this as a malloc (dynamic memory allocation); pi points to 65

```
int *pi = new(65);
char *pc = reinterpret_cast<char*> (p);

cout << *pi << endl;
cout << *pc << endl;

cout << pi << endl;
cout << pc << endl;
```

65

A

0x1609c20

A
Hug ?

Statements

- There are no major differences between C and C++ statements

Type	Syntax	Example
Null	;	
Composed (block)	{}	{i=0;j=1; ...}
Conditional	if, if-else	if (i==10) {...} else {...}
The switch	switch, case, default	
Iterative	while	while (i<=N) { ... }
	for	for (int i=0; i<N; i++) {...}
	do-while	do { ... } while (i<=N);
Jump statements	Break, continue, goto	

The range for statement

- C++11 introduced a simpler for statement
  - It is used to iterate through the elements of a container or other sequences

```
for (declaration : expression)
  statement
```

  - The expression must represent a sequence
    - Braced initializer
    - Array
    - An object, such as a vector or string

Please, see the unit 04 for containers

  - The declaration defines a variable
    - It must be possible to convert each element of the sequence to the variable type

Example

The types **string** and **vector** are introduced in section u04s02

```
vector<int> v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto r : v)
  cout << r;

for (auto &r : v)
  r =* 2;
```

vector is the type, and also the type of the element which is an integer. In a way the vector is a classs which can be used with all different types. We can have a vector of integers or a vector of strings, etc. A vector which is an array of integers.

r is automatically defined such that compatible with values in v. We are just printing them out with no space.

Double the value of each element

r is not a copy of the element but a synonym of v with i varying from 0 to 9

First Loop: In this loop, auto i creates a copy of each element in the vector v2.
Inside the loop, i is modified, but the modifications do not affect the original elements in the vector v2.
Therefore, the loop prints the values of i after adding 2 to each copied element, but the elements of v2 remain unchanged.

Second Loop: In this loop, auto &i creates a reference to each element in the vector v2.
Inside the loop, i is modified directly, affecting the original elements in the vector v2.
Therefore, the loop prints the values of i after adding 2 to each original element in the vector v2.

```
string s("this is a string");

for (auto c : s)
  cout << c;

for (auto &c : s)
  if (ispunct())
    cnt++:;
```

String initialization

If it is a punctuation character count it up

Exceptions

- For exception in C++
  - See documentation
  - Recall notions of Java and/or software engineering courses