Contemporary C++: Learning Modern C++ in a Modern Way

آلفا
ALPHA

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 3/24

Computation (Part II), Expressions and Statements

- Initialization vs. Assignment
- Initialization and Uniform initialization
- Writing simple arithmetic programs
- Expressions and Statements
- Autos: Automatic type deduction
- Structured programming: An introduction
- The elements of Structured programming
- Selection statements & Iterative statements
- Q & A

150 min (incl. Q & A)

Arithmetic operators cont.

- The operands of % shall have integral types. It can't be applied to floating-point types.
- An example for % operator calculation: In Gregorian calendar, a year is *leap* year if it is divisible by 4 but not by 100 except that years divisible by 400.

```
if (( year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
  std::cout << year << " is a leap year." << std::endl;
```

- If the second operand of / or % is zero the behavior is undefined.

a = (a / b) * b + a % b

- The if statement and && and || will be discussed later.

- Unary plus and Unary minus

Assignment operator

Operator	Function	Use
=	simple assignment	lvalue = expr
+=	add and assign	lvalue += expr
-=	subtract and assign	lvalue -= expr
*=	multiply and assign	lvalue *= expr
/=	divide and assign	lvalue /= expr
%=	modulo and assign	lvalue %= expr

Compound assignment operators

- Notational compactness

L-value op= R-value

L-value = L-value op R-value

“Longhand” simple assignment

- Association: All operators are right-to-left associative.

```
i = j = k = 0;
i++; j--; k++;
i += j -= k;
```

Increment and decrement operators cont.

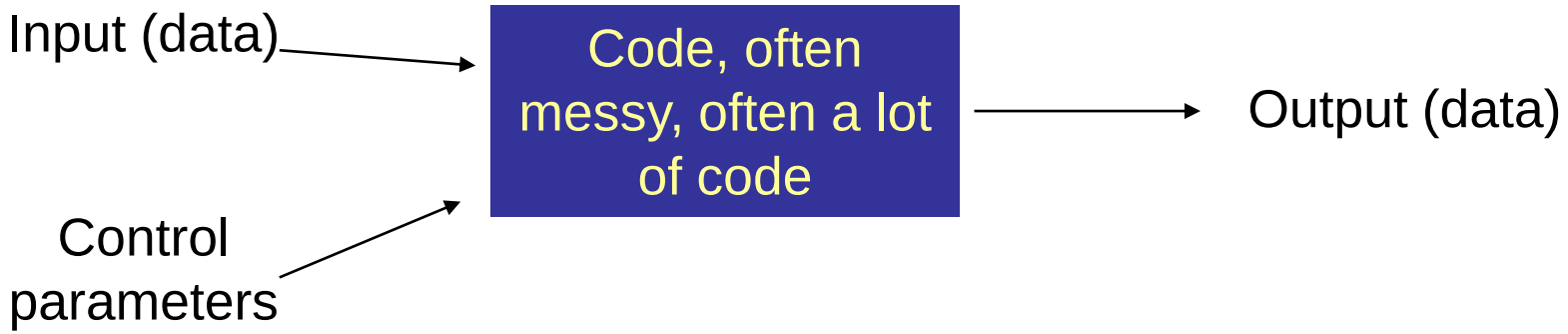
- Increment and decrement operators are assignment operators:
- x++/++x means x += 1 which again means x = x + 1
- x--/--x means x -= 1 which again means x = x - 1
- The value of ++x, is the new (that is, incremented) value of x.
- The value of x++, is the old value of x.

```
y = ++x; // y = (x += 1) or ++x; y = x;

y = x++; // y = (t = x, x += 1; t) or t = x; x++; y = t;
```

```
char c = 'A';
c++; // ok, c is 'B'
double f = 0.0;
f++; // ok, f is 1.0
--3; // error: 3 is not l-value
(1+5)++; // error: (1+5) is not l-value
```

☞ Use prefix ++ and -- unless you have a good reason not to.

Simple computations, simple programs



• Input: from screen, from files, from other input devices, from other programs, from other parts of a program

• Output: to screen, to files, to other output devices, to other programs, to other parts of a program

• Our job is to express computations
  - Correctly
  - Simply
  - Efficiently
• Our main tool is to break up a big computation into many little ones
  - Abstraction
    . Provide higher-level concept that hides detail
  - “Divide and conquer”

Simple computations, simple programs

- Write a program that read a length in inches and convert it to centimeter.

```
// inch to centimeter conversion
#include <iostream>
int main()
{
  double cm_per_inch = 2.54; // number of centimeters in an inch
  std::cout << "Please enter a length in inches: (0 for exit)\n";
  int length; // length in inches
  std::cin >> length;
  if (length != 0) {
    std::cout << length << " in == " << cm_per_inch * length << std::endl;
  }
}
```

- Write a program that read the radius of a circle and computes the area and its circumference.

```
// area and circumference
#include <iostream>
int main()
{
  double Pi = 3.14; // The pi number
  std::cout << "Please enter radius: (0 for exit)";
  int r = 0; // radius
  std::cin >> r;
// to be continued on next page
```

Simple computations, simple programs

```
//  continued from last page
   double Area = Pi * r * r;
   double Circ = 2 * Pi * r;
   if (r != 0) {
     std::cout << "Area = " << Area << std::endl;
     std::cout << "Circ = " << Circ << std::endl;
   }

   return 0;
}
```

Declaration and Definition

- Before a name (identifier) can be used in a C++ program, it must be declared. A *declaration* introduces names into a scope. A declaration specifies the interpretation and attributes of these names.
- A declaration is a *definition* if it specifies the entity to which the declared name refers.

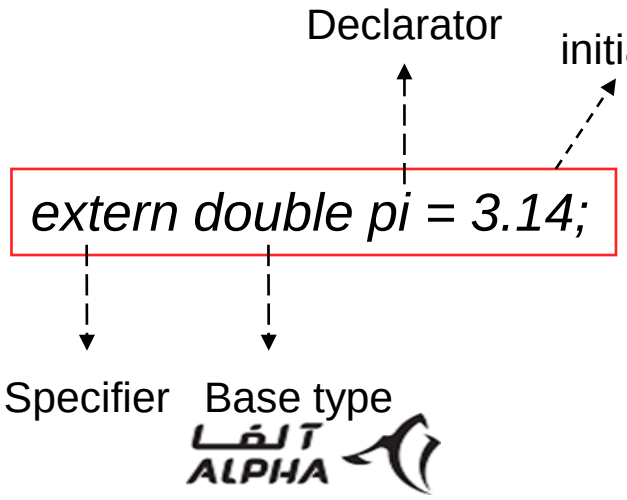
```
char ch; // decl. and def.
string s; // decl. and def.
int count = 1; // decl., def. and init.
struct Date { int d, m, y; }; // decl., def.
double sqrt(double); // just decl.
struct user; // just decl.
int Incr(int a); // just decl.
int Day(Date* p) { return p->d; } // def.
```

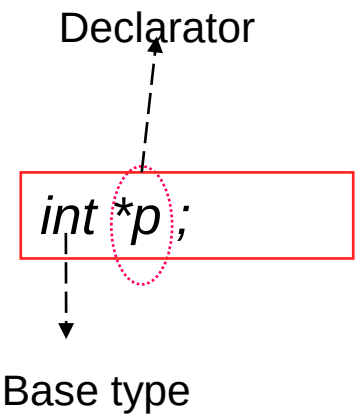
```
int Incr(int a) // defines Incr
{
   return ++a;
}
```

- **;** is necessary, because it is a statement *terminator*.

•A declaration consists of four parts:

- an optional "specifier"
- a base type
- a declarator
- optional initializer





nitialization vs. assignment

Initialization vs. assignment

- Initialization differs from assignment.

Type object = value

Type object = old_object

vs.

object = value

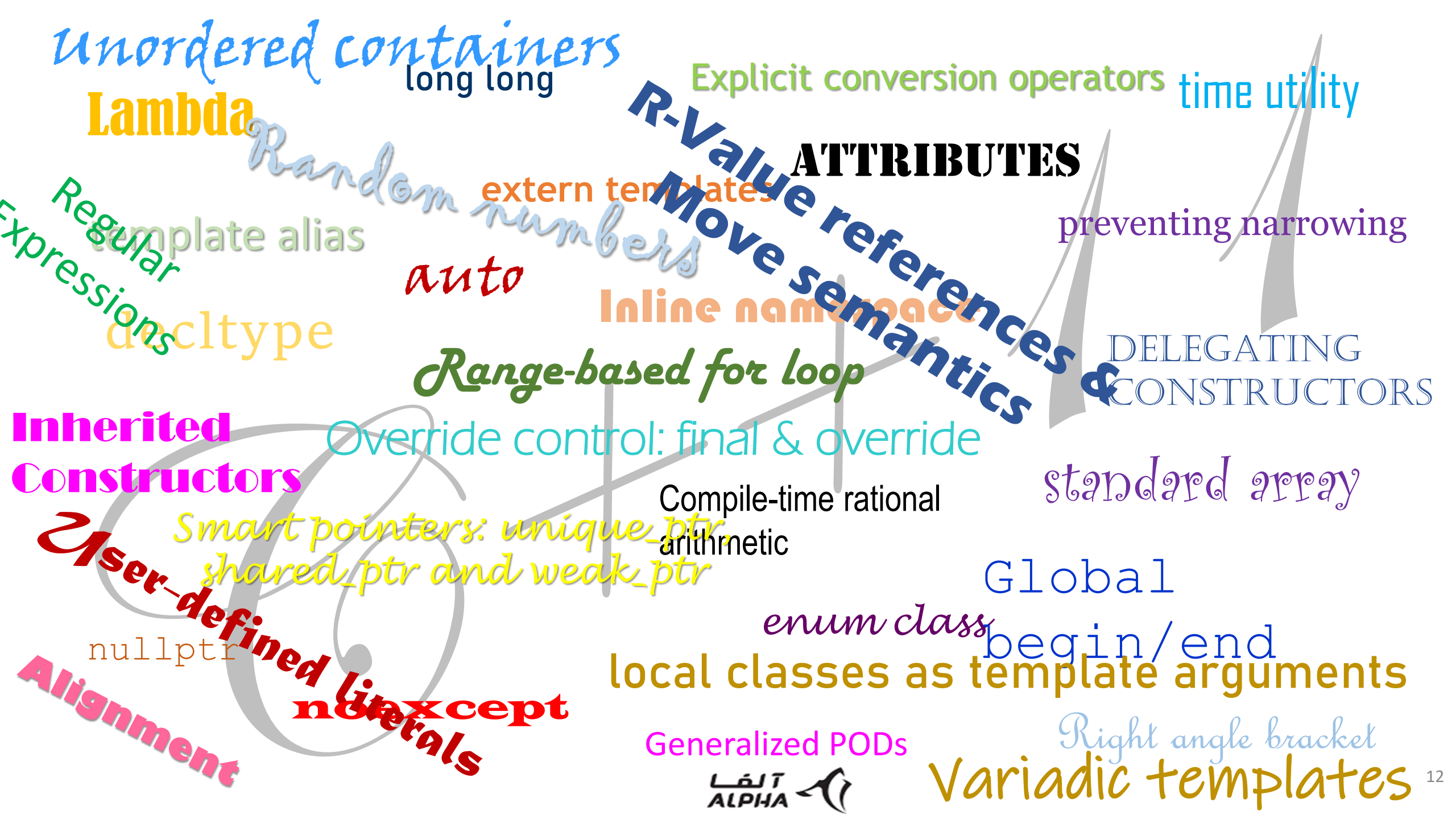
object = old_object

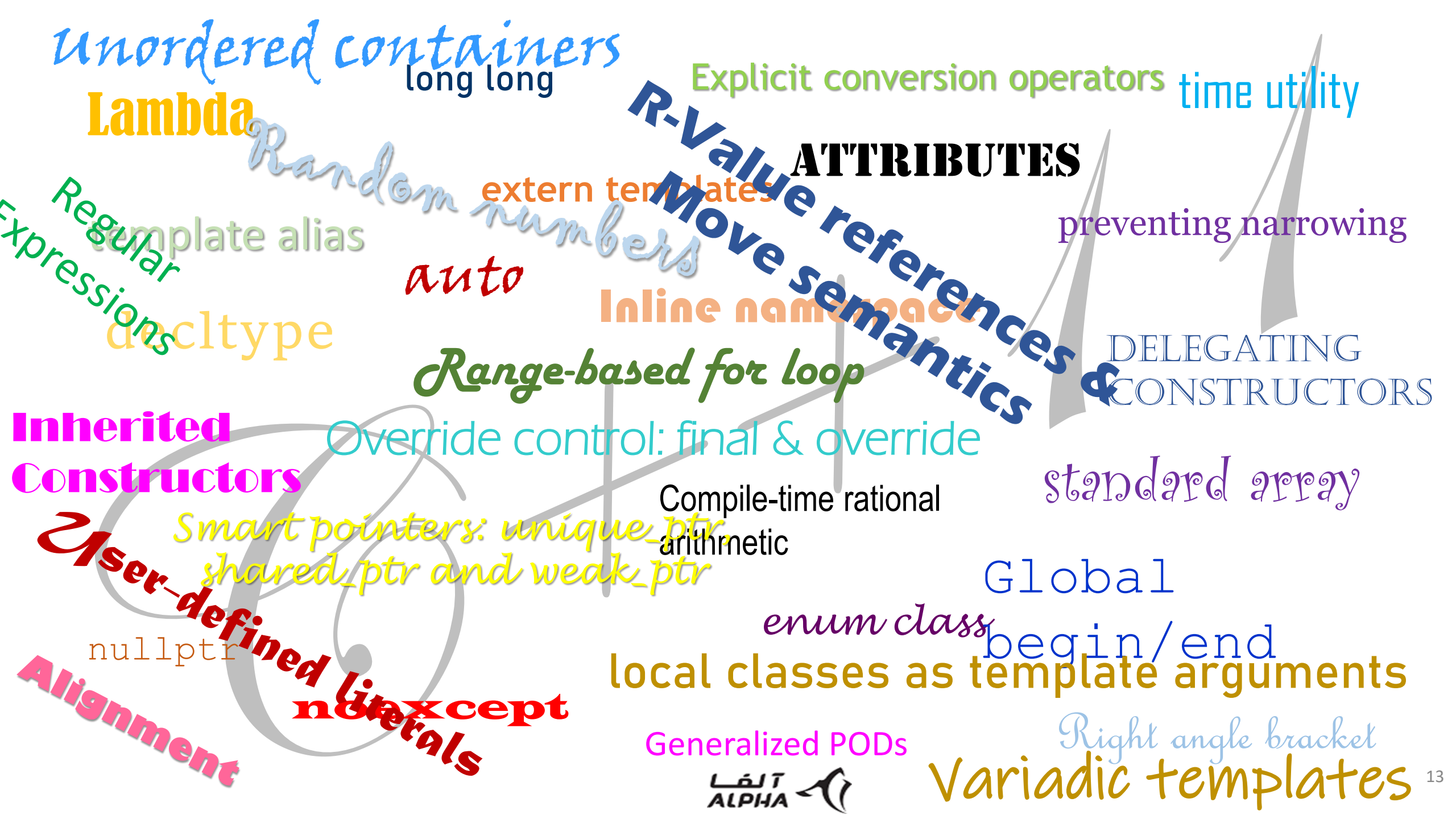
- Note that = is the assignment operator and == tests equality.
- the task of initialization is to make an uninitialized piece of memory into a valid object.

- Before an object can be used, it must be given a value. C++ offers a variety of notations for expressing initialization, such as the = used above, and a universal form based on curly-brace delimited initializer lists.

```
double d1 = 2.3; // initialize d1 to 2.3
double d2 {2.3}; // initialize d2 to 2.3
double d3 = {2.3}; // initialize d3 to 2.3 (the = is optional with { ... })
d3 = d2; // assignment
d3{d1}; // error
```

- Initialization: Constructors → Default, Copy and Move
- Assignment: copy assignment and move assignment operators

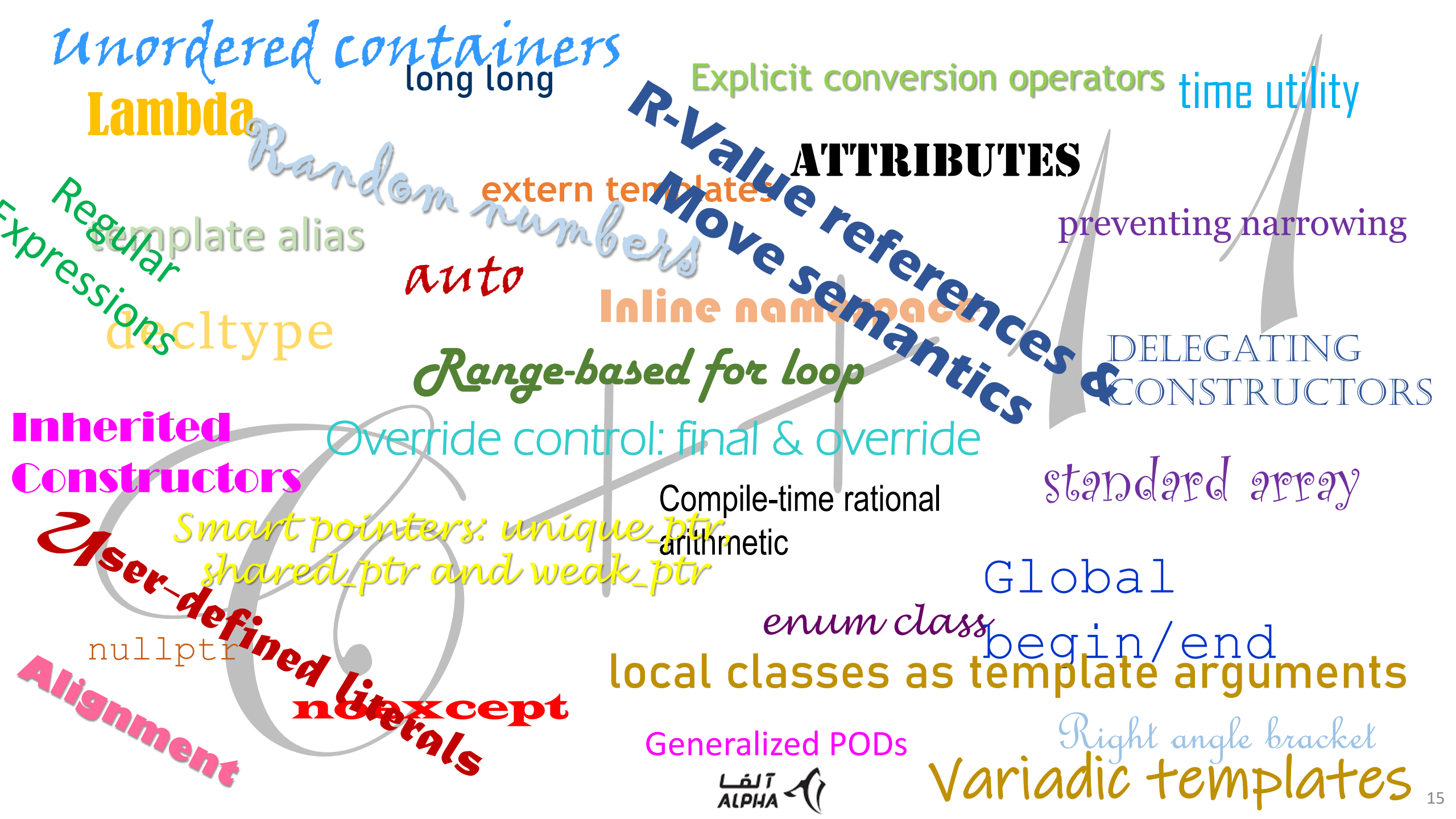

Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates

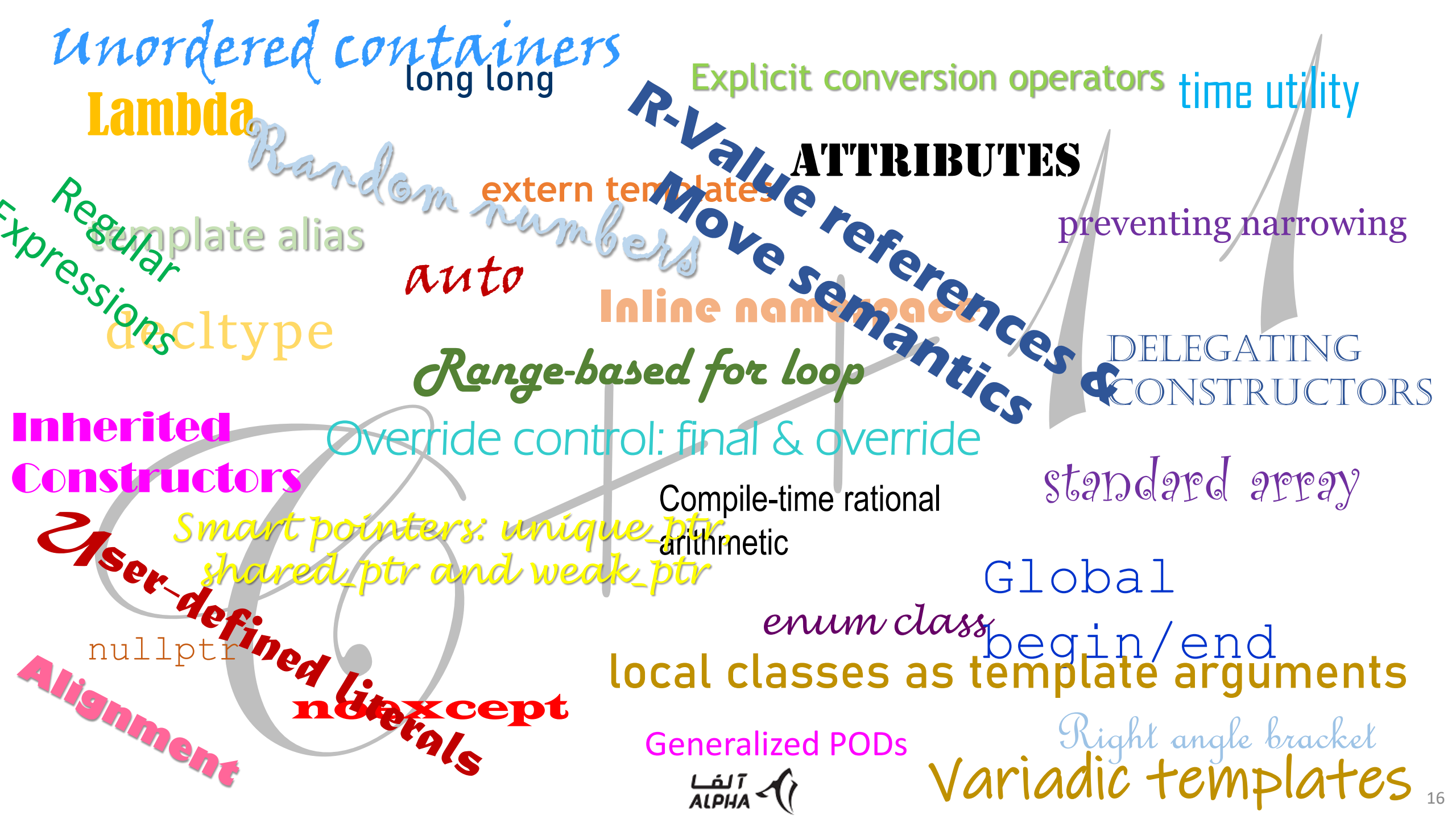

Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates

preventing Narrowing

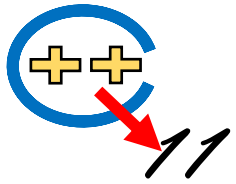
- {} initialization doesn't narrow:

```
int i = 7.2; // i becomes 7
int i2{7.2}; // error: double to integer conversion
int i3 = {7.2}; // error: double to integer conversion
double d = 7;
int x2{d};         // error: narrowing (double to int)
char x3{7};        // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };       // error: double to int narrowing
```

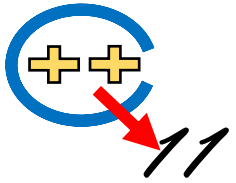

Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates


Unordered containers
long long
Explicit conversion operators
time utility
Lambda
Random numbers
R-Value references &
Move semantics
ATTRIBUTES
extern templates
Regular Expressions
template alias
preventing narrowing
auto
Inline namespace
decltype
Range-based for loop
DELEGATING CONSTRUCTORS
Override control: final & override
Inherited Constructors
standard array
Compile-time rational arithmetic
Smart pointers: unique_ptr
shared_ptr and weak_ptr
User-defined literals
Global
enum class
begin/end
nullptr
local classes as template arguments
noexcept
Alignment
Generalized PODs
Right angle bracket
Variadic templates

Auto: Type deduction from initializer



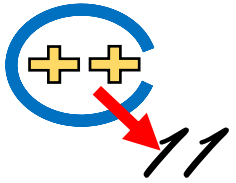
Auto: Type deduction from initializer



1.

- Static languages: Fortran, Algol, C, C++, Java, … → State the type of variables clearly
- Dynamic languages: Lisp, Python, … → Deduce the type of variables with the assigning values

Auto: Type deduction from initializer



1.

Static languages: Fortran, Algol, C, C++, Java, … → State the type of variables clearly

Dynamic languages: Lisp, Python, … → Deduce the type of variables with the assigning values

- In C++98, the programmer must state the type of variables.
- Compiler prefers type to initializer.
- Python

```
answer = 42    # answer is now an int
answer = 42.1 # answer is now a float
```

<type-specifier> variable-name = initializer $_{opt}$;

```
double g = 9.81f; // g is double
int answer = 42.1; // answer = 42
```

2.

Global variable

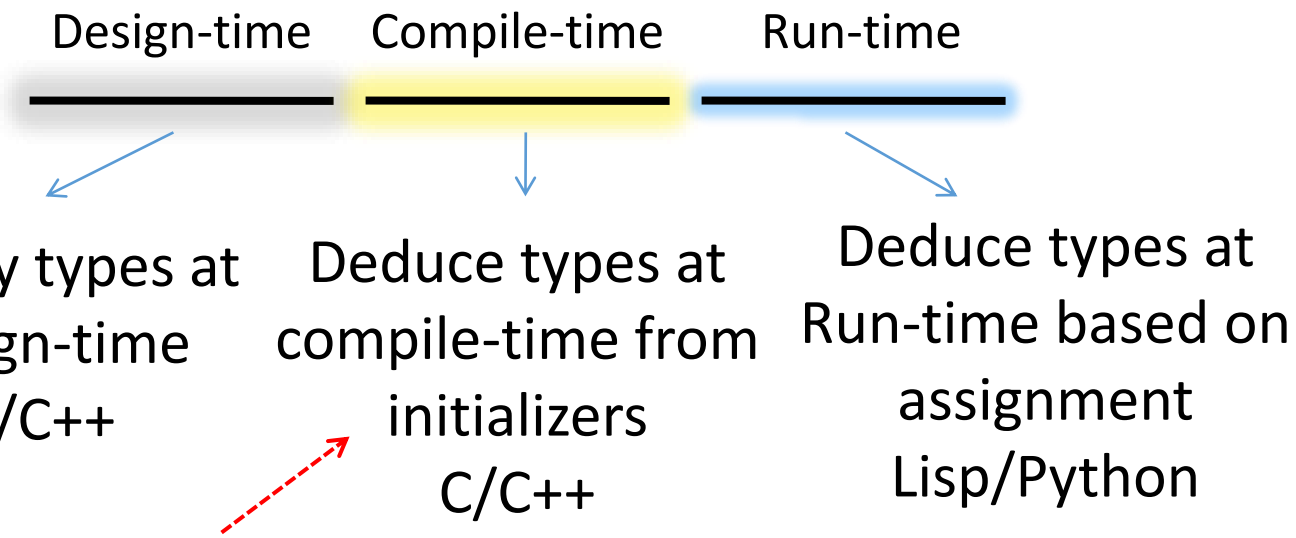
Local variables

Automatic storage

```
bool b; // non-local variable: b = false
int main()
{
   long long LL; // uninitialized
   auto long long LL; // redundant auto. rather verbose
   auto std::string s; // calling default ctor
   return 0;
} //
```

New mission of auto

- Third approach: Compiler can figure out the type of expressions at compile-time.



- The original auto is generally useless keyword.
- Recycle the keyword auto:

  1) To remove the old redundant usage

  2) New responsibility: Determine or deduce the type of expressions from initializer

```
int x = 5; // verbose: the int is redundant
auto x = 5; // OK: x is int because 5 is int
```

auto <variable> = expression

Statements

- Statements form the smallest executable unit within a C++ program.
- Statement are terminated with semicolon.
- Null statement: just a single statement

```
; // null statement
int val;; // additional null statement
```

- Declaration statements
- Expression statements

```
// one of the shortest C++ program
int main()
{
   ;
}
```

```
float f = 1.0; // declaration statement
double a = f + 1; // expression statement
```

Selection statements, comparison operators

- Selection statement

  - if statement

```
if (condition)
  statement
```

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

  - switch statement

- The condition is an expression that yields a truth value.

Operator	Function	Use
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equality	expr == expr
!=	inequality	expr != expr

Relop

- Relop returns bool (true or false)

```
if (x) // ...
```

```
if (x != 0) // ...
```

Simple computations, simple programs

• Write a program that asks the user to enter two numbers and tells the user which number is larger that the other.

• Declaring multiple names: comma-separated declarators.

• Divide two floating-point numbers. read from input two long double numbers and divide first by second. Note: Division by zero is *mathematically undefined*.

```
// Maximum of two numbers
#include <iostream>
int main()
{
   std::cout << "Enter two numbers: ";
   int a, b; // declaring multiple names
   std::cin >> a >> b;
   int max;
   if (a > b)
      max = a;
   else
      max = b;
   std::cout << "Max = " << max << std::endl;

   return 0;
}
```

```
// floating-point division
#include <iostream>

int main()
{
   std::cout << "Enter two numbers: ";
   long double a, b;
   std::cin >> a >> b;
// to be continued on next page
```

Simple computations, simple programs cont.

```
// continued from last page
  if (b == 0.0) { // handle division by zero
    std::cout << "Division by zero!" << '\n';
    return 0;
  }
  std::cout << a << '/' << b << " = " << a / b << '\n';

  return 0;
}
```

- Common mistake:

```
if (a = 7) { // oops!: constant assignment in condition
}
```

Compilers usually issue warning

```
// most likely it should be
if (a == 7) {
}
```

Iteration statements

- Iteration statement
  - while statement

```
while (condition)
  statement
```

  - for statement

```
for (for-init statement condition opt; expression opt) statement
```

  - do-while statement

```
do statement while (expression) ;
```

- Each of these statements executes a statement (called the *controlled statement* or the *body of the loop*) repeatedly until the condition becomes false or the programmer breaks out of the loop some other way.

while statement, increment and decrement operators

• The world's first "real program" running on a stored program computer.
Write a program to calculate the squares of int values up to 100.

Control information

```
// calculate and print a table of squares 0-99
#include <iostream>
int main()
{
  int i = 0;
  while (i < 100) {
    cout << i << '\t' << i * i << std::endl;
    ++i; // increment i (that is, i becomes i+1)
  }
}
// no it wasn't actually written in C++ ☺
```

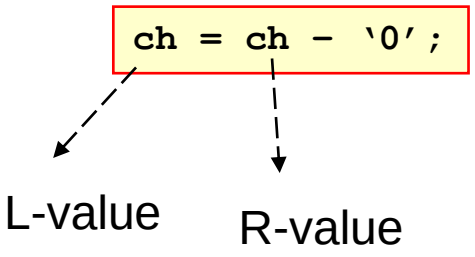
• Increment and decrement operators.

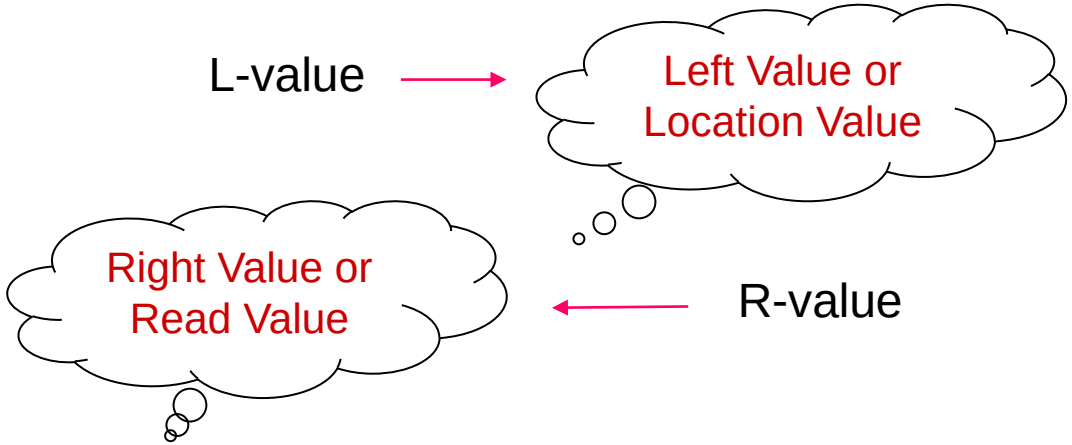
Operator	Function	Use
++ --	Post increment Post decrement	lvalue++ lvalue--
++ --	Pre increment Pre decrement	++lvalue --lvalue

• The operators ++ and can be used as both *prefix* and *postfix* operators.

Variable, Object, L-Value and R-Value

- There are two values associated with a symbolic variable:
  - The data value, stored at some location in memory: *rvalue*
  - Its location value; that is, the address in memory at which its data value is stored: *lvalue*





- lvalue is something that can be on the left hand side of an assignment.
- a *variable* is an object that has a name.
- an *object* is a contiguous region of storage.
- It is possible to have objects that do not have name like temporary objects.
- lvalue is an expression that refers to an object.

```
*p[a+10]= 7;
```

for statement

for (for-init statement condition $_{opt}$; expression $_{opt}$)
statement

→ for header
→ for body

- Re-write the table of squares of with for statement.

```
#include <iostream>

int main()
{
  for (int i = 0; i < 100; i++) {
    std::cout << i << '\t' << i * i << std::endl;
  }

  return 0;
}
```

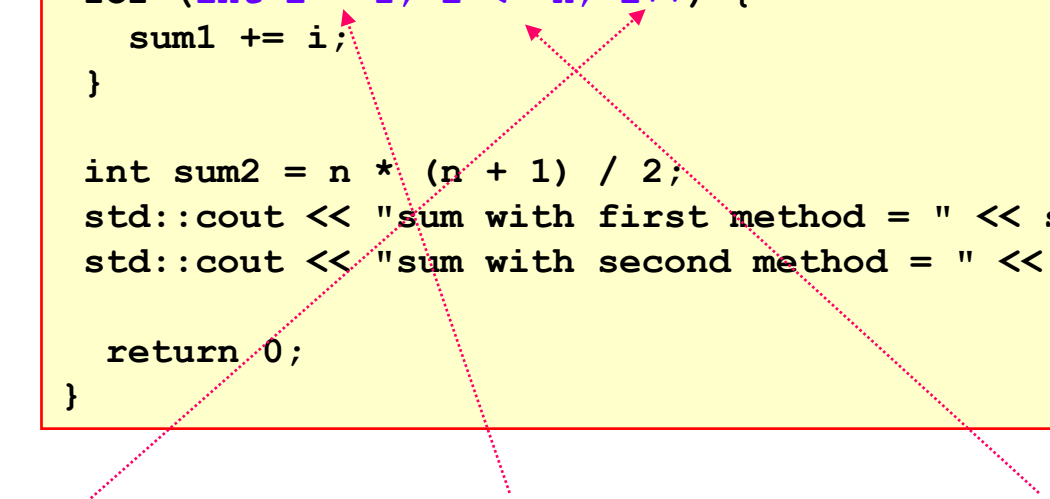
for-init statement
while (condition) {
  statement
  expression
}

- Write a program that read an integer n from input and computes the summation 1, 2, 3, … n. then compute n * (n + 1) / 2. compare two results.


```
#include <iostream>

int main()
{
  std::cout << "Enter an integer number (0 for exit): ";
  int n;
  cin >> n;
// to be continued on next page
```

for statement cont

```
//  continued from last page
if (n == 0) // if (!n)
  return 0;
int sum1 = 0; // must be initialized
for (int i = 1; i <= n; i++) {
  sum1 += i;
}

int sum2 = n * (n + 1) / 2;
std::cout << "sum with first method = " << sum1 << std::endl;
std::cout << "sum with second method = " << sum2 << std::endl;

 return 0;
}
```



- In for statement, the *loop variable*, the *termination condition*, and the *expression* that updates the loop variable can be presented *"up front"* on a single line. This can greatly increase readability and thereby decrease the frequency of errors.
- Declarations in for statements: A variable can be declared in the initializer part of a for statement.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant