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

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 4/24

Session 4 .Constants, Pointers, References, Functions and Introduction to the Standard Containers (Part I)

- Working with unknown number of data
- Constants
- Pointers & References
- Null pointers and nullptr
- C++ standard library: standard array
- Range-based for loop
- More on Autos: Automatic type deduction
- C as a Procedural Programming Language
- Functions
- Writing simple functions
- Inline functions
- Q & A

150 min (incl. Q & A)

Declarations in for statements

```
{
  int i; // decl., def., uninitialized
  for (i = 1; i <= n; ++i) { // assignment
    // …
  }
}
```

just Assignment

decl., def. & init.

```
for (int i = 1; i <= n; ++i) { // good

}
```

• Scope of i: Outside of the for statement. i is not local.

• Scope of i: until the end of for statement. i is local.

• If the final value of an index needs to be known after exit from a for loop, the index variable must be declared outside the for loop.

Language-technical rule:

Locality is good.

Initialization, default initialization

- Initialization: giving an object an initial value.

```
int sum1 = 0;
```

- When we do not specify an initial value for a variable we are implicitly relaying on default initialization.

default initialization → implicit initialization

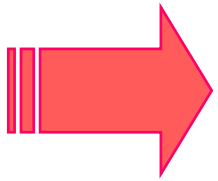
implicit initialization →
- For objects of class type: constructor
- For global variables of fundamental types: Zero initialized.
- For local variables of fundamental types: *undefined*

```
int i; // i = 0;
double d; // d = 0.0
int main()
{
  std::string s; // s is initialized by constructor
  int k; // k does not have well-defined value
  bool b = false; // explicitly initialized
  char c = 'a'; // explicitly initialized
}
```

- Important rule: Every declaration with an initializer is a definition.

Simple computation, simple programs

• Compute n!: Get n from input and compute n * (n – 1) * (n – 2) * …* 1. Note that: 0! = 1 and 1! = 1. Handle input for negative numbers.



```
#include <iostream>
int main()
{
  std::cout << "Enter an integer number: (-1 for exit) ";
  int n;
  std::cin >> n;
  if (n == -1) {
    return 0;
  }
  if (n == 0) {
    std::cout << n << "! = " << 1 << '\n';
    return 0;
  }
  long int Fact = 1; // must be explicitly initialized
  for (int i = n; i > 0; --i) {
    Fact *= i;
  }
  std::cout << n << "! = " << Fact << '\n';

  return 0;
}
```

Selection statements: the Switch statement

Selection statements: the Switch statement

- Selection statement
  - if statement
  - switch statement

```
if (condition)
  statement
```

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

```
switch (condition)
  statement
```

- The switch statement causes control to be transferred to one of several statements depending on the value of a condition.

- A **switch**-statement tests a value against a set of constants. Those constants, called **case**-labels, must be distinct, and if the value tested does not match any of them, the **default** is chosen. If the value doesn't match any case-label and no default is provided, no action is taken.

Selection statements: if vs. Switch statement

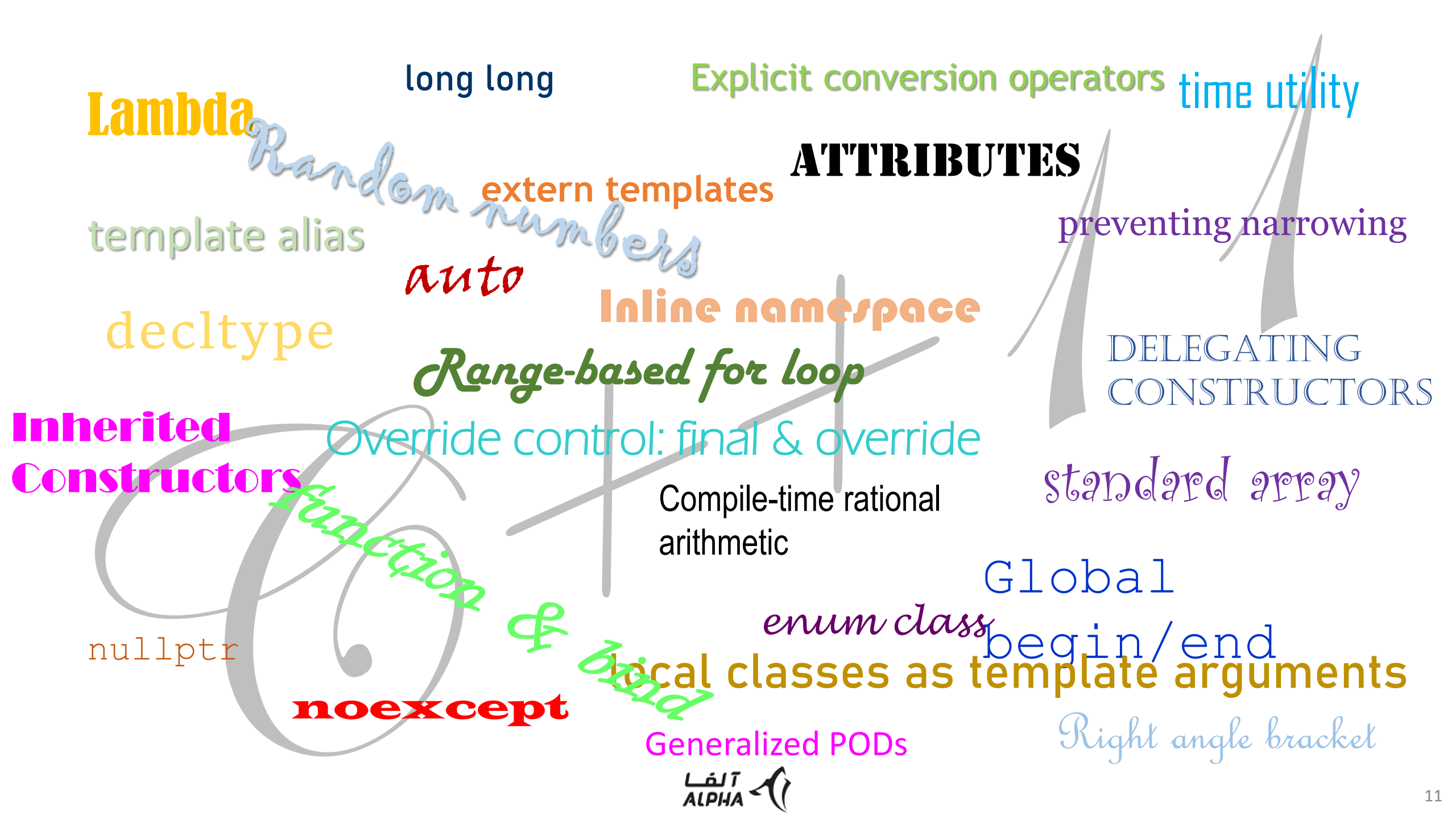
```
bool accept()
{
    cout << "Do you want to proceed (y or n )? "; // write question
    char answer;
    cin >> answer; // read answer
    if (answer == ´y´)
        return true;
    return false;
}
```

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n"; // write question
    char answer = 0; // initialize to a value that will not appear on input
    cin >> answer; // read answer
    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

switch statement and Break

```
bool accept3()
{
    cout << "Do you want to proceed (y or n)?\n"; // write question
    char answer = 0; // initialize to a value that will not appear on input
    cin >> answer; // read answer
    bool result =
    switch (answer) {
    case 'y':
    case 'Y':
        result = true;
        break;
    case 'n':
    case 'N':
        result = false;
        break;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```


long long
Explicit conversion operators
time utility
Lambda
Random numbers
ATTRIBUTES
extern templates
template alias
preventing narrowing
auto
Inline namespace
decltype
DELEGATING CONSTRUCTORS
Range-based for loop
Inherited Constructors
Override control: final & override
standard array
Compile-time rational arithmetic
function & bind
Global begin/end
enum class
nullptr
Local classes as template arguments
noexcept
Generalized PODs
Right angle bracket


long long
Explicit conversion operators
time utility
Lambda
Random numbers
ATTRIBUTES
extern templates
template alias
preventing narrowing
auto
Inline namespace
decltype
DELEGATING CONSTRUCTORS
Range-based for loop
Inherited Constructors
Override control: final & override
standard array
Compile-time rational arithmetic
function
Global
enum class
nullptr
begin/end
& bind
Literal classes as template arguments
noexcept
Right angle bracket
Generalized PODs
ALPHA

Attributes

- Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.
- Attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions __attribute__((...)), Microsoft extension __declspec(), etc.
- A construct [[…]] is called an attribute and can be placed just about anywhere in the C++ syntax.
  - C++11 attributes
    - noreturn
    - carries_dependency: C++ memory model
  - C++14 attributes: deprecated

static_assert

[[fallthrough]]

fold expression

Hexadecimal floating-point literals

Structured Bindings

string_view

template <auto>

Inline variable

Nested namespace

Class template argument deduction

Selection statements with initializer

Byte type

if constexpr

static_assert

[[fallthrough]]

fold expression

Hexadecimal floating-point literals

Structured Bindings

string_view

template <auto>

Inline variable

Nested namespace

Class template argument deduction

Selection statements with initializer

Byte type

if constexpr

Fallthrough

- The use of a fallthrough statement is intended to suppress a warning that an implementation might otherwise issue for a case or default label that is reachable from another case or default label along some path of execution.

```
void f(int n) {
    void g(), h(), i();
    switch (n) {
    case 1:
    case 2:
        g();
        [[fallthrough]];
    case 3: // warning on fallthrough discouraged
        h();
    case 4: // implementation may warn on fallthrough
        i();
        [[fallthrough]]; // ill-formed
    }
}
```

Working with unknown number of data

- Input loop:

```
while (cin >> x) { /* … */ }
```

```
cin >> x; // return cin
```

- attempt to read from `cin` (input stream) until hit the end of file or read invalid object for the type of x.

```
if (cin >> x) { /* … */ }
```

→

```
cin >> x;
if (cin) { /* … */ }
```

- test the condition and read a value as a side effect.
- End-of-file signal:
  - MS Windows operating system : *Control* key + z
  - Unix and Linux operating systems: *Control* key + d
- Read unknown number of characters and write them.

```
#include <iostream>
int main()
{
  char x;
  while (std::cin >> x)
    std::cout << x << '\n';
  return 0;
}
```

Simple computation, simple programs

- Read some “integers” and compute 1. number of inputs 2. sum and 3. average of numbers.


```
#include <iostream>

int main()
{
  using namespace std;
  int n;
  int count = 0;
  int sum = 0;
  int average;
  while (cin >> n) { // standard input loop
    ++count;
    sum += n;
  }

  average = sum / count; // integer division
  cout << count << '\t' << sum << '\t' << average << '\n';

  return 0;
}
```

Simple computation, simple programs

• Prime numbers. Read some “integers” and determine which are prime numbers. The numbers should be greater than equal 1. A positive integer p is called *prime*, if it has just two divisors, namely 1 and p. By convention 1 is not prime.

```
#include <iostream>

int main()
{
  using namespace std;

  for (;;) {
    cout << "X =  (0 for exit) ";
    int x;
    if (cin >> x) {
      if (x <= 0) {
        cout << "Invalid number!";
        break;
      }
      else if (x == 1)
        cout << x << " isn't prime." << endl;
      else { // x >= 2
        bool is_prime = true;
      // inefficient and naïve implementation
      for (int i = 2; (i <= x / 2) && (is_prime = (x % i != 0)); ++i) ;
// to be continued on next page
```

Forever loop, logical operators, …

- Logical operators:
  - &&

    expr1 && expr2: if both expressions are true, it returns true, otherwise it returns false.
  - ||

    expr1 || expr2: if both expressions are false, it returns false, otherwise it returns true.
- *Short-circuit evaluation*
- Logical operators: && (logical and), and || (logical or) guarantee that their left-hand operand is evaluated before their right-hand operand.

```
// continued from last page
        if (is_prime)
            cout << x << " is prime." << endl;
        else
            cout << x << " is composite." << endl;
        }
    }
  }
  return 0;
}
```

```
if (i < 0 || i >= max) // …
```

```
if (i >= 0 && i < max) // …
```

- “infinite” loop

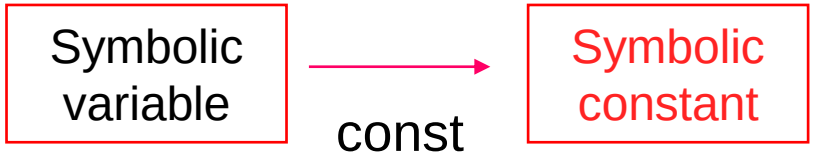
```
for (;;) // …
```

```
while (true) // …
```

Operator	Function	Use
&&	Logical and	expr && expr
||	Logical or	expr || expr

Constants

- The keyword const can be added to the declaration of an object to make the object declared a constant.
- Constants must be initialized.
- Constants can't be assigned.




```
const float PI = 3.14; // PI is r-value
const int light_speed = 3 * 100000000; // m/s
const double cm_per_inch; // error: no initializer
PI++; // ++ needs lvalue
const char vowel[5] = { 'A', 'E', 'I', 'O', 'U' };
```

- (Obviously) you can assign a const to a non-const but not a non-const to a const.

- Constants → maintainable code
- Constants → more localized information
- Use symbolic constants to avoid *magic numbers*.

```
const int max = 128;
int v[max];
```

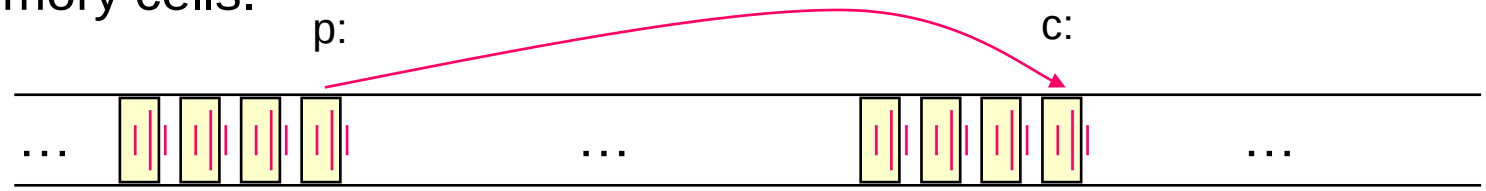
```
for (int i = 0; i < BUF_SIZE; i++)
  // …
```

CentimeterInchConversionProg.

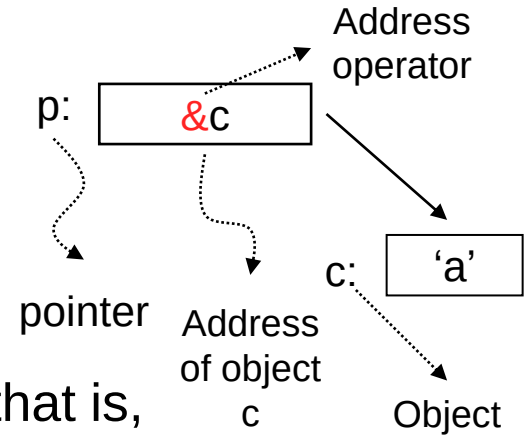
CentimeterInchConversion2Prog.

Pointers

• A typical machine has an array of consecutively numbered or addressed memory cells.



- For a type *T* , *T* * is the type "pointer to *T* ." That is, a variable of type *T* * can hold the address of an object of type *T*.
- *Pointer* is an object which its value represents the address of another object or 0.
- If you can access an object, you can obtain its address, and vice versa.




```
char c = 'a';
char *p = &c; // p holds the address of c
```

- The fundamental operation on pointer is *dereferencing*, that is, referring to the object pointed to by the pointer. This operation is also called *indirection*.

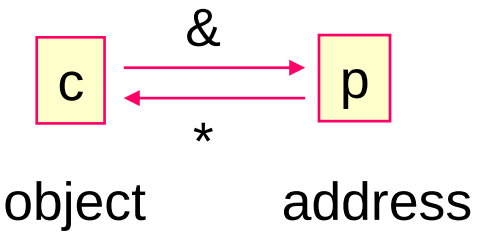
```
char c2 = *p; // c2 == 'a'
```

Dereference operator

Pointers ... cont.

- Indirection operators:

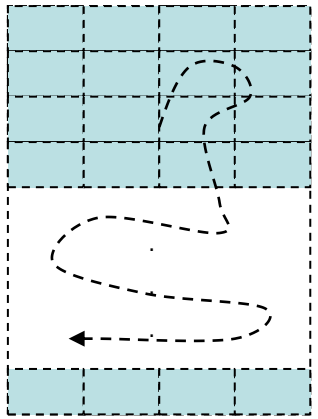
Operator	Function	Use
&	address of	&lvalue
*	dereference	*expr




```
int     x; // x is an object of type int
int     *p; // *p has type int
int*    p; // p has type int*
```

declarator

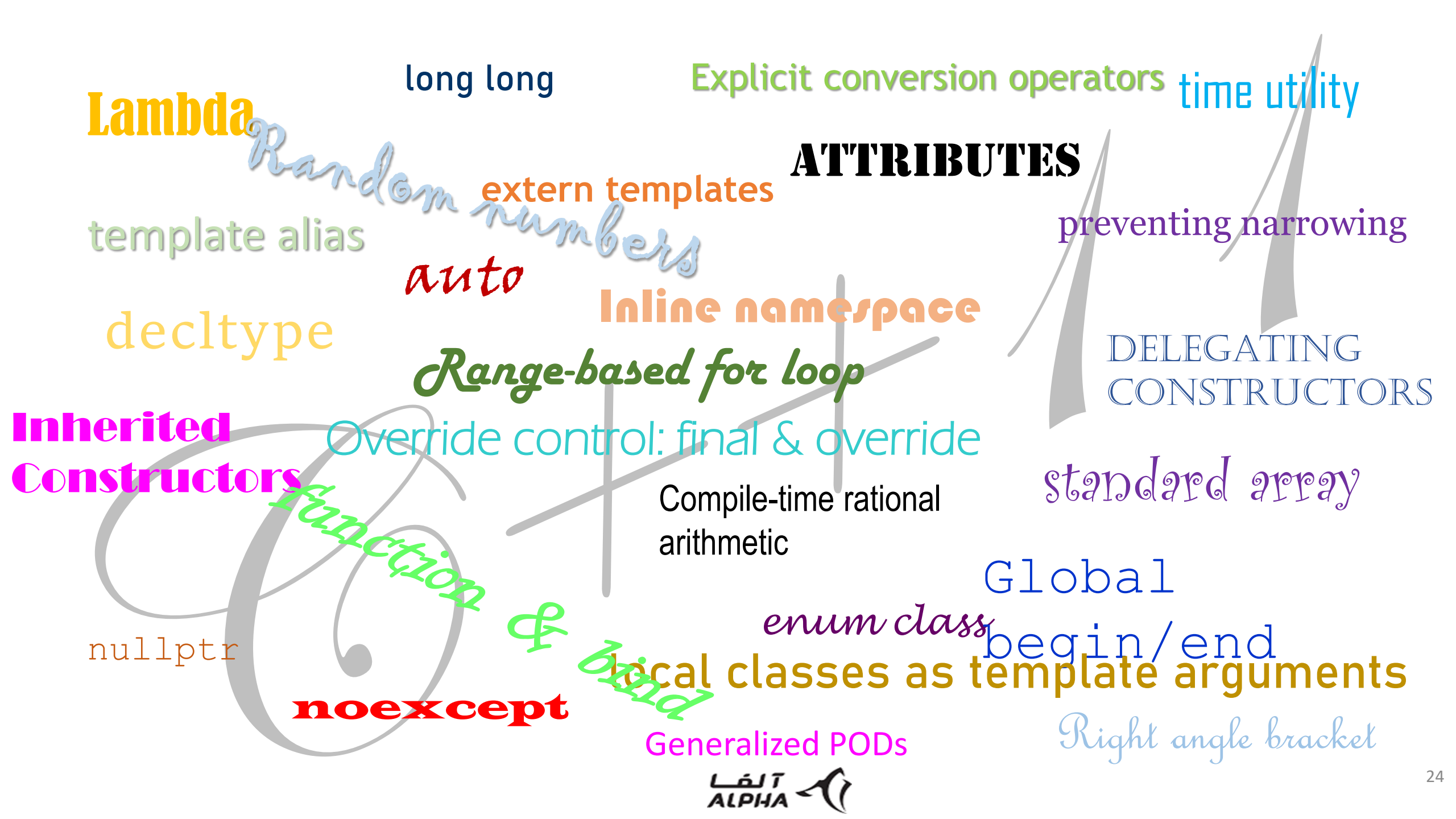
Type specifier

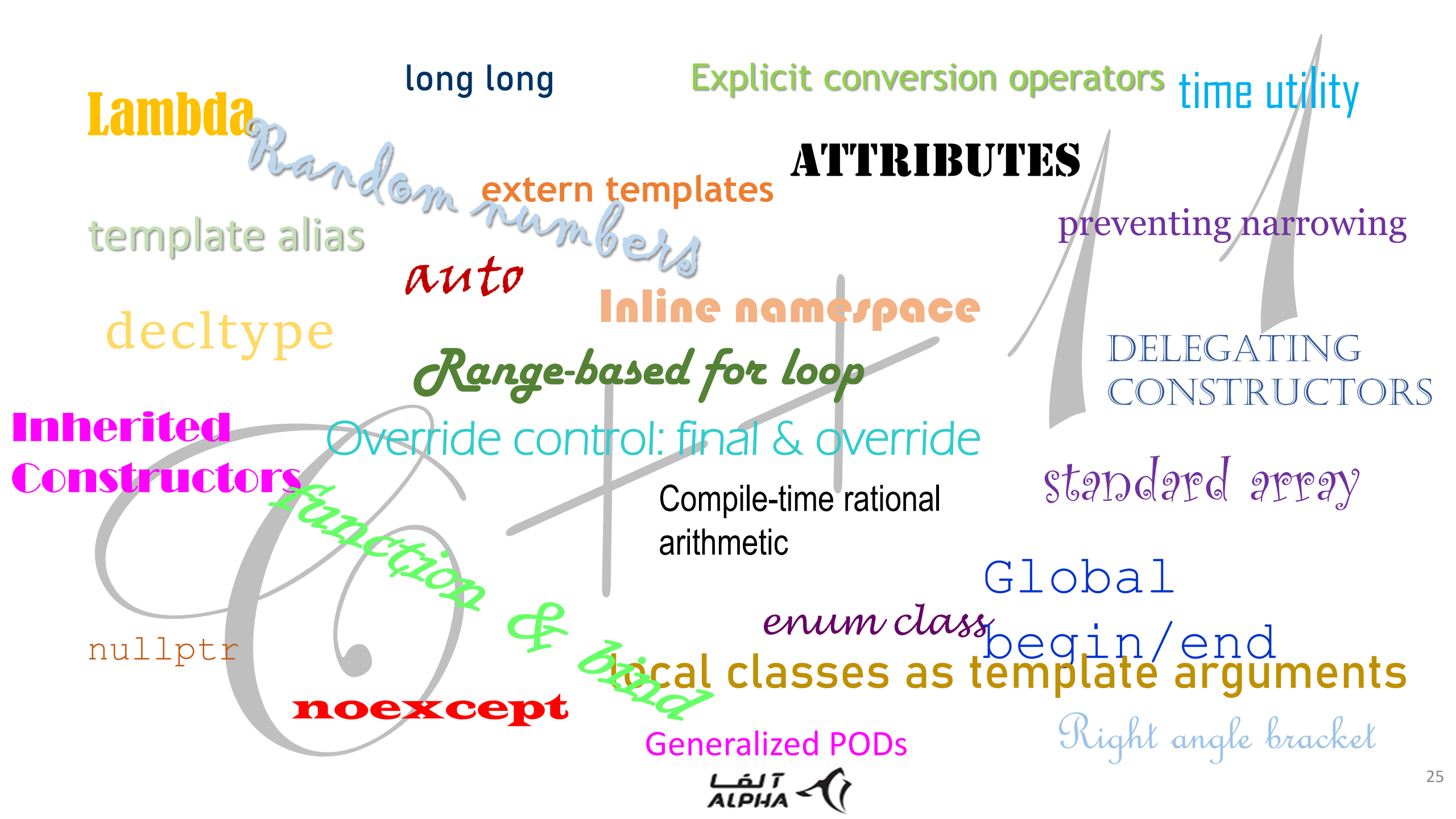


- Zero (0) is an `int`.
  - Standard conversion
  - No object is allocated with the address 0.
- Every pointer has an associated type.

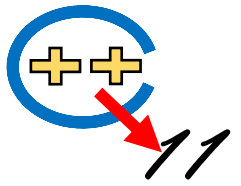
```
void g()
{
  int i = 1024;
  int* pi = i; // error: initialize with rvalue
  double* pd = pi; // error: type mismatch
}
```

```
void f()
{
  int* p; // points to nowhere
  p = 0; // pointer literal
  int x = 1;
  p = &x; // p points to x
  (*p)++; // () is necessary!
}
```


long long
Explicit conversion operators
time utility
Lambda
Random numbers
ATTRIBUTES
extern templates
template alias
preventing narrowing
auto
Inline namespace
decltype
DELEGATING CONSTRUCTORS
Range-based for loop
Inherited Constructors
Override control: final & override
standard array
Compile-time rational arithmetic
function & bind
Global begin/end
enum class
nullptr
Literal classes as template arguments
noexcept
Generalized PODs
Right angle bracket


long long
Explicit conversion operators
time utility
Lambda
Random numbers
ATTRIBUTES
extern templates
template alias
preventing narrowing
auto
Inline namespace
decltype
DELEGATING CONSTRUCTORS
Range-based for loop
Inherited Constructors
Override control: final & override
standard array
Compile-time rational arithmetic
function & bind
Global begin/end
enum class
nullptr
Local classes as template arguments
noexcept
Generalized PODs
Right angle bracket

Null pointers in C and C++



- Pointers and null pointers

```
char c = 'a';
char* p = &c; // '*' means 'pointer to' p points to c, '&' is address operator
```

- C99 → Null pointer constant

✓An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.

- C++03 → Null pointer constant

```
// macros
#define NULL 0
#define NULL ((void *)0)
char* p = NULL;
```

✓A null pointer constant is an integral constant expression rvalue of integer type that evaluates to zero.

- Zero (0) is an int.

```
char* p = 0; // points to nowhere
             // no memory allocation yet
```

- No object is allocated with the address *0. 0 acts as a pointer literal, indicating* that a pointer doesn't refer to an object.

Nullptr: a literal for null pointers

- Problems with NULL and 0:

```
void f(int) { /* … */ }
void f(char*) { /* … */ }
f(0); // f(int) called
f(NULL); // f(int) called
```

1. *Distinguishing between null and zero.*

- There is no way to write a call to f(char*) with a null pointer value without writing an explicit cast (i.e., f((char*)0)) or using a named variable.

2. *Naming null*

```
std::string s1(false); // compiles, calls char* constructor with null
std::string s2(true); // error
```

- The pointer literal is the keyword *nullptr*.

```
// macros
#define NULL 0
#define NULL ((void *)0)
char* p = NULL;
```

- C++11 → Null pointer constants

```
int x = nullptr; // error
char* pc = 0; // still good
char* pc2 = nullptr; // best
bool b = (pc == pc2); // OK: b = true
f(nullptr); // called f(long long*)
g(nullptr); // error
class Point { /* … */ } *pp = nullptr;
```

Nullptr cont.

- A null pointer constant is an integral constant expression rvalue of integer type that evaluates to *zero* or a *rvalue of type std::nullptr_t*.

```
typedef decltype(nullptr) nullptr_t;
```

- A complete example

```
#include <cstddef>
#include <iostream>
template<class F, class A> void forward(F f, A a)
{
   f(a);
}

void g(int* i)
{
   std::cout << "Function g called\n";
}

int main()
{
   g(NULL); // fine
   g(0); // fine

   forward(g, nullptr);   // fine
   forward(g, NULL);  // ERROR: No function g(int)
}
```

Reference Types

- The notation T& means *reference to T.*
- A reference is an alternative name for an object.
- Like constants, references must be initialized.

Type Specifier& Reference = Referent;

```
void f()
{
  int i = 0;
  int& ir; // error: reference should be initialized
  int& iref = i; // OK
  int& iref2 = 1; // error: should be initialized with l-value of an object
}
```

•The main use of references is for specifying arguments and return types for functions.

```
const int& ir = 5; //ok
```

```
void swap(int& x, int& y)
{
  int tmp = x; // decl. statement
  x = y;
  y = tmp;
}
```

```
int a = 0, b = 1;
swap(a, b);
```

Pointer vs. Reference

- A Reference must always refers to some object. There is no *Null Reference.*

```
int& ir = 0; // error: null reference?
int* ip = 0; // ok: null pointer
```

- A Reference should be *initialized*. Pointer doesn't have such restrictions.

```
bool& ir; // error: reference should be initialized?
int* ip; // ok: uninitialized pointer: valid but risky
```

```
void UseDouble(const double& rd)
{
  // …
}
```

```
void UseDouble(const double* pd)
  if (pd)
    // …
}
```

- You can't separate reference from referent. You can't reseat reference.

```
string s1("Hello");
string s2("Good bye");
string& rs = s1; // rs refers to s1
string* ps = &s1; // ps points to s1
rs = s2; // rs still refers to s1, but s1's value is now "Good Bye"
ps = &s2; // ps not points to s2; s1 is unchanged
```

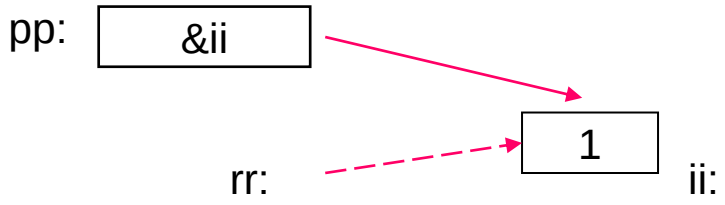
Q When should I use references, and when should I use pointers?

A Use references when you can, and pointers when you have to.

Reference implementation: An application of constant pointer

•The obvious implementation of a reference is as a (constant) pointer that is dereferenced each time it is used.

```
void g()
{
  int ii = 0;
  int& rr = ii;
  rr++; // ii is incremented to 1
  int* pp = &rr; // pp points to ii
}
```



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