HIGH-LEVEL PROGRAMMING 2

Lvalue References

by Prasanna Ghali

Pointers and constness

Top-level const indicates that object itself is const

```
int i{4}, j{5};
int const ci{6}; // ci is top-level const
int *p1{&i};      // ok
++*p1;     // modify i thro' p1
p1 = &j;   // make p1 point to another object
*p1 *= 2; // j is now 10
p1 = &ci; // error!!!
```

Pointers and constness

When a pointer can point to a const object, we refer to that const as low-level const

```
int i{4}, j{5};
int const ci{6};    // ci is top-level const
int const *p1{&i};  // p1 is low-level const
++*p1;      // error: cannot modify i thro' p1
++i;        // ok: can modify i directly
p1 = &j;    // ok: make p1 point to another object
*p1 *= 2;   // error: cannot modify j thro' p1
p1 = &ci;   // ok: can read ci thro' p1
```

Pointers and constness

Top-level const indicates that object itself is const

```
int i{4}, j{5};
int const ci{6};      // ci is top-level const
int * const p1{&i};   // p1 is top-level const
++*p1;      // ok: can modify i thro' p1
p1 = &j;   // error: p1 is top-level const
p1 = &ci; // error: p1 is top-level const
int * const p2{&ci}; // error: p2 is not low-level
                      // const
```

Even though p2 is top-level const object, it is *not* low-level const and can only point to non-const objects

Pointers and `const`ness

const to right of * indicates pointer itself is const while
const to left of * indicates object pointed to is const

```
int i{4}, j{5};
int const ci{6}; // ci is top-level const
int const * const p1{&i};  // ok: p1 is both top-level and
                           // low-level const
++*p1;    // error: p1 is low-level const
p1 = &j;  // error: p1 is top-level const
p1 = &ci; // error: p1 is top-level const
int const * const p2{&ci}; // ok: p2 low-level const
```

Distinction Between Top-Level and Low-Level `const`ness

- Distinction matters when we copy an object
- When we copy an object, top-level `const`s are ignored because copying an object doesn't change the copied object!!!

```
int i{4};
int * const pi1{&i}; // pi is top-level const
int const ci{6};  // ci is top-level const
int const *pi2 = &ci; // pi2 is low-level const

i   = ci;  // ok: top-level const in ci is ignored
pi2 = pi1; // ok: pointed-to type matches;
           // top-level const in pi1 is ignored
```

Distinction Between Top-Level and Low-Level `const`ness

- Low-level `const` is never ignored: when copying both objects must have same low-level `const` qualification or there must be conversion between types of two objects
- In general, we can convert non`const` to `const` but not vice-versa

```
int i{4};
int const ci{6};       // ci is top-level const
int const *pi2 = &ci; // pi2 is low-level const
int const * const pi3 = pi2; // ok: pi3 is both top-level
                             // and low-level const
int *pi = pi3; // error: pi3 has low-level const
               // but pi doesn't
pi2 = pi3; // ok: pi2 and pi3 have same
           // low-level const qualification
pi2 = &i;  // ok: we can convert int* to int const*
```

Top-Level `const` Pointers Are Useful!!!

- They remove certain drawbacks of ordinary pointers!!!
  - Pointer must be initialized
  - Once initialized, pointer must always point to same object
  - Thro' pointer, we can change pointed-to object
- Anywhere you can use pointed-to object, you can use top-level `const` pointer that points to object
- Only if something can be done about nasty syntax!!!

```
int i{4}, j{5};
int * const p1{&i};  // p1 is top-level const
++*p1;      // ok: can modify i thro' p1
p1 = &j;    // error: p1 is top-level const
```

lvalues and *rvalues*

- Every expression is an *lvalue* or an *rvalue*
- ***lvalue*** (short for *locater value*) is expression that refers to identifiable memory location
- By exclusion, any non-*lvalue* expression is an ***rvalue*** - think of ***rvalue*** as "value resulting from expression"
- Useful to visualize a variable as name associated with certain memory locations

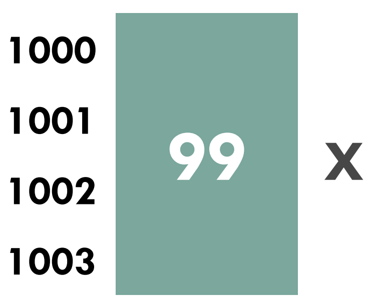
```
int x = 99;
```

- Sometimes (as an *lvalue*) x means its memory locations and sometimes x (as an *rvalue*) means value stored in those memory locations

```
int x = 99;
x = 100;
x = x + 2;
```

here, x means *lvalue*

while here, x means *rvalue*



Lvalue References

- C++ repurposes top-level `const` pointers as *lvalue references* without the ugly syntax!!!

```
int i{4}, j{5};
int const ci{6};  // ci is top-level const
int * const p1{&i};
++*p1;     // ok: can modify i thro' p1
p1 = &j;   // error: p1 is read-only object
*p1 = j;   // ok: copy j to i thro' p1
*p1 = ci; // ok: top-level const is ignored when copying

int &ri{i}; // ri is reference to i [similar to p1]
ri = j; // ok: assign j to i thro' alias ri
ri = ci; // ok: assign ci to i thro' alias ri
```

`ri` is *alias* for variable `i` i.e., name `ri` is convenient *shorthand* for name `i`

What Can We Do With References?

- Why pointers?
  - Pointers provide ability for functions to bypass pass-by-value semantics and change objects
  - Pointers provide ability to efficiently pass data structures between functions with minimum overhead
- We can use references to provide same benefits as pointers without the ugly syntax!!!

Functions: Pass-by-Value Convention

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

client calls function `myabs` using function call operator `()`

```
int num = 10;
num = myabs(-num)
```

this expression is called *function argument*

1) At runtime, expression (or argument) `-num` is evaluated
2) Result of evaluation is used to initialize parameter `number`
3) Changes made to parameter `number` are localized to function `myabs`
4) Function `myabs` terminates by returning value of type `int`
5) When function `myabs` terminates, variable `number` ceases to exist

Functions: Pass-by-Value Convention And Pointers

- Top-level `const` pointers provide ability for functions to modify objects

```
void swap(int * const lhs, int * const rhs) {
  int tmp{*lhs};
  *lhs = *rhs;
  *rhs = tmp;
}

int main() {
  int i{5}, j{6};
  std::cout << "i: " << i << " | j: " << j << "\n";
  swap(&i, &j);
  std::cout << "i: " << i << " | j: " << j << "\n";
}
```

Functions: Pass-by-Reference Convention

- Idea that a reference can be convenient shorthand for some object provides *pass-by-reference* semantics

```
void swap(int& lhs, int& rhs) {
  int tmp{lhs};
  lhs = rhs;
  rhs = tmp;
}

int main() {
  int i{5}, j{6};
  std::cout << "i: " << i << " | j: " << j << "\n";
  swap(i, j);
  std::cout << "i: " << i << " | j: " << j << "\n";
}
```

Ugly pointer syntax is gone!!!

Top-Level and Low-Level const Pointers Are Useful!!!

- They remove certain drawbacks of ordinary pointers!!!
  - Pointer must be initialized
  - Once initialized, pointer must always point to same object
  - Thro' pointer, we can ~~change~~ read from but not write to pointed-to object
- Anywhere you can use pointed-to object, you can use top-level & low-level const pointer that points to object

```
int i{4}, j{5};
int const * const p1{&i};  // p1 is top-level & low-level const
++*p1;     // error: p1 is low-level const
p1 = &j;   // error: p1 is top-level const
j = *p1;   // ok: ignore top-level const when copying
int const ci{6};
int const * const p2{&ci}; // ok
++*p2;     // error: p2 is low-level const
j = *p2;   // ok: ignore top-level const when copying
```

Top-Level and Low-Level const Pointers Are Useful!!!

- If only something can be done about ugly syntax!!!

```
int i{4}, j{5};
int const * const p1{&i};  // p1 is top-level & low-level const
++*p1;     // error: p1 is low-level const
p1 = &j;   // error: p1 is top-level const
j = *p1;   // ok: ignore top-level const when copying
int const ci{6};
int const * const p2{&ci}; // ok
++*p2;     // error: p2 is low-level const
j = *p2;   // ok: ignore top-level const when copying

int const &rci1{i}; // rci1 is const-reference to i aka p1
++rci1;     // error: rci1 is reference to const object
j = rci1;   // ok: ignore top-level const when copying

int const &rci2{ci}; // rci2 is const-reference to ci aka p2
++rci2;     // error: rci2 is const-reference to ci
j = rci2;   // ok: ignore top-level const when copying
```

Top-Level and Low-Level `const` Pointers Are Useful!!!

- Top-level and low-level `const` pointers provide ability for functions to move large values without copying these values while guaranteeing values are not modified

```
int largest(int const * const ptr, int size) {
  int large_val{*ptr};
  for (int i{1}; i < size; ++i) {
    large_val = (ptr[i] > large_val) ? ptr[i] : large_val;
  }
  return large_val;
}

int main() {
  int big_array[1'000'000];
  // fill big_array with values ...
  int largest_value = largest(big_array, 1'000'000);
  // use largest_value ...
}
```

Functions: Pass-by-const-Reference Convention

- Modern C++ provides C++ standard library type `std::array<T,N>` as replacement for static C-style arrays!!!

```
#include <limits>
#include <array>

int largest(std::array<int, 1'000'000> const& value) {
  int large_val {std::numeric_limits<int>::min()};
  for (int x : value) {
    large_val = (x > large_val) ? x : large_val;
  }
  return large_val;
}

int main() {
  std::array<int, 1'000'000> big_array;
  // fill big_array with values ...
  int largest_value = largest(big_array);
}
```

Pass-by-Value vs. Pass-by-Reference

- When should you use pass-by-value, pass-by-reference, and pass-by-`const`-reference?
  - Pass-by-value gives you copy
  - If you want to change value of object passed, you must use non-`const` reference
  - Pass-by-`const`-reference prevents you from changing value of object passed

```
void f(int a, int& r, int const& rci) {
  ++a;   // change the local a
  ++r;   // change object aliased by r
  ++rci; // error: rci is alias for read-only object
}
```

Pass-by-Value vs. Pass-by-Reference

- When should you use pass-by-value, pass-by-reference, and pass-by-`const`-reference?

```
void g(int a, int& r, int const& rci) {
  ++a;           // change the local a
  ++r;           // change object aliased by r
  int x = rci; // ok: read object aliased by rci
  // use x ...
}

int main() {
  int x{}, y{}, z{};
  g(x, y, z); // ok: x==0; y==1; z==0
  g(1, 2, 3); // error: r needs a variable to refer to
  g(1, y, 3); // ok: since rci is const, we can pass literal
}
```

Pass-by-Value vs. Pass-by-Reference: Rules of Thumb

1) Use pass-by-value to pass very small objects [one or two `int`s, one or two `double`s, or something like that]
2) Use pass-by-`const`-reference to pass large objects that you don't need to modify
3) Return a result rather than modifying an object thro' reference argument
4) Use pass-by-reference only when you've to [for manipulating containers and other large objects and for functions that change several objects]

Rule #3

3) Return a result rather than modifying an object thro' reference argument

```
// return new value as the result
int incr1(int a) {
  return a+1;
}

// modify object passed as reference
void incr2(int& a) {
  ++a;
}

int x{7};
x = incr1(x); // pretty obvious
incr2(x);     // pretty obscure
```

Initializer for const-Reference


```
void g(int a, int& r, int const& rci) {
  ++a;          // change the local a
  ++r;          // change object aliased by r
  int x = rci; // ok: read object aliased by rci
  // use x ...
}

int main() {
  int x{}, y{}, z{};
  g(x, y, z); // ok: x==0; y==1; z==0
  g(1, 2, 3); // error: r needs a variable to refer to
  g(1, y, 3); // ok: since rci is const, we can pass literal
}
```

Notice that const-reference doesn't need an lvalue initializer

Initializer for `const`-Reference


```
int i{5}, j{6};
int &ri1{i}; // ok: ri1 is another name for lvalue expression
int &ri2{10}; // error: 10 is rvalue expression!!!

int const &ri3{i};     // ok: ri3 is const-reference to i
int const &ri4{12.1};  // error: narrowing conversion!!!
int const &ri5 = 12.1; // ok: ri5 is const-reference to
                       // unnamed temporary variable
                       // initialized to 12
```

Return By Lvalue Reference

- Useful if you want function to be used on both left- and right-hand sides of assignment operator
- More on this later …

Pointers vs. References

- No such thing as null reference but there is such a thing as null pointer
- Possibly more efficient – no need to check if reference is valid but need to check if pointer is not null pointer
- Can reassign pointer but cannot reassign reference
- Better syntax when returning reference compared to returning pointer

Review

- Range-`for` iteration
- Pointers and `const`ness: top-level and low-level
- Four types of pointers
- Review of lvalue and rvalue expressions
- References: C++ mechanism that implement top-level `const` pointers without ugly syntax
- `std::array<T,N>`: C++ standard library type as modern C++ replacement for static C-style array
- Pass-by-value and pass-by-reference: when to choose pass-by-value and when to choose pass-by-reference
- Pointers vs references: when to choose pointer and when to choose reference