References

References:

The material in this handout is collected from the following references:

- Sections $2.3.1$ and $6.2.2$ of the text book C++ Primer.
- Section 7.7 of C++ Programming Language.
- Section 8.5 of Programming: Principles and Practice Using C++.

Types, objects, and variables

Every name and every expression has a type that determines the operations that may be performed on it.

- A *type* defines a set of possible values and a set of operations that can be applied on these values.
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.
- A *declaration* is a statement that introduces a name to the compiler. It specifies a type associated with that name. The following statements are declarations (but not definitions):

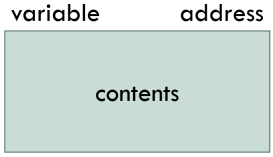
```
extern int mile;   // declaration of an external (or global) variable
int add(int, int); // declaratinon (or prototype)
```

- A *definition* is a special kind of declaration that causes memory to be reserved for a variable and initializes the memory with an optional set of values. The following statements are definitions (and also declarations):

```
int mile;
int add(int left, int right) {
  return left + right;
}
```

Review of pointers and memory [from C]

Each variable in a program stores its contents in the computer's memory, and each byte-sized chunk of the memory has an address number. The size of the memory chunk or the number of bytes associated with a variable is determined by the variable's type. For example, an `int` variable requires 4 bytes of memory while a `double` variable requires 8 bytes of memory. This is the notation that will be used when talking about variables in memory:



In the picture above,

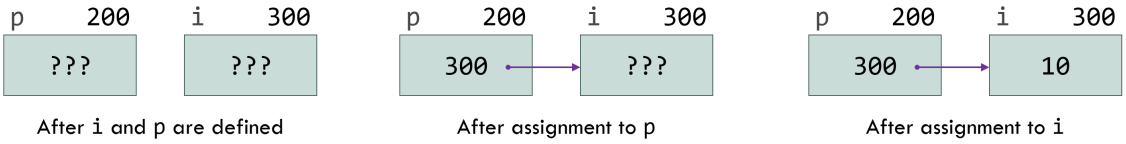
- **variable** refers to a named object.
- **address** refers to the arbitrary address number associated with the starting byte of the memory chunk where the variable is given storage. Note that the actual addresses are determined at program startup but using specific values would be useful for discussion purposes.
- **contents** refers to the value stored at this location; `???` means the value is unspecified.

Conceptually, a pointer variable itself does not directly contain a value like an `int` or a `double`, but it contains the address in memory of another variable or function. When the `int` or `double` value is accessed through the pointer variable, then that value is accessed indirectly.

We can declare pointer variables easily:

```
void foo() {
  int i;  // i can only store integer values
          // the value of i is unspecified at this point
  int *p; // p can only store the address of an integer
          // the value of p is unspecified at this point
  p = &i; // the value of p is now the address of variable i
  i = 10; // the value of i is now 10
}
```

Visualizing the code in function `foo`:



After `i` and `p` are defined | After assignment to `p` | After assignment to `i`

From the pictures, you can see that it is possible to modify variable `i`'s value in two different ways:

- directly through the variable `i`:

  ```
  i = 20; // value of i is now 20
  ```

- indirectly through the pointer variable `p`. The `*` operator is called the *dereference* or *indirection* operator and it indirectly allows us to access the `int` value stored at the variable that the pointer `p` is pointing to. The compiler will evaluate expression `*p` in the following manner:

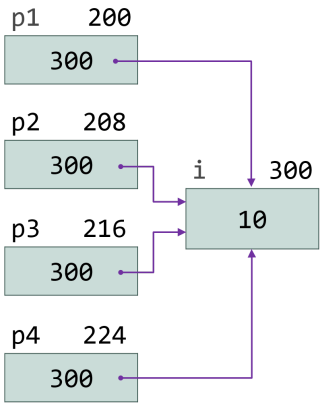
  1. The compiler will read the value stored in variable `p` and treat it as a memory address where an `int` value is stored.
  2. Next, the compiler will reference the four-byte sized memory chunk starting at this address to retrieve the integer stored there. In the following statement, we're indirectly assigning value 30 to variable `x`:

     ```
     *p = 30; // value of i is now 30
     ```

     Expression `*p = 30` takes integer value 30 and stores it at memory address stored in variable `p`. Since `p` contains address 300, value 30 will then be stored at memory address 300, thus indirectly changing the value of variable `i` from 10 to 30.

The pointer and the variable it is pointing to are distinct and independent objects. Assigning a new value to `i` doesn't affect point `p` - it will continue to point to `i`; assigning `p` the address of a different `int` variable doesn't affect variable `i`. In fact, we can have any number of pointers pointing to `i`:

```
int i {10};
int *p1 {&i}, *p2 {p1}, *p3 {p2}, *p4 {p3};
```



Each of these pointers can be used to modify `i`. In addition, each pointer variable can be modified to point at some other `int` variable.

An important thing that you learnt in C is that every program object (variable or function) can only be associated with one address. That is, once you name a memory location, that name cannot be used for another memory location in the same scope. In other words, once you *bind* a name to a memory location, you can't unbind it.

```
int i {1}; // i is the name of a four byte-sized memory chunk in the
           // .data region of program memory
void f() {
  int i;    // OK: i is the name of a four byte-sized memory chunk in the
            // .stack region of program memory
            // This local (internal) variable i has a different memory chunk
            // then the global (external) variable i
  double d; // OK: d is name of 8 byte-sized memory chunk
  float  d; // ERROR: trying to rename another memory chunk to d
  // other code here ...
}
```

C++ introduces *references* which conceptually allow multiple names to be associated with one address.

Motivation for references

Pointers are useful in two contexts: to *efficiently pass large amounts of data* to a function and to pass an address to a function so that the function *modifies the original* and not the copy.

The simplest way of passing an argument to a function is to give the function a copy of the value you use as the argument. A parameter of a function `foo()` is a local variable in `foo()` that's initialized each time `foo()` is called with the value that the argument evaluates to. For example:

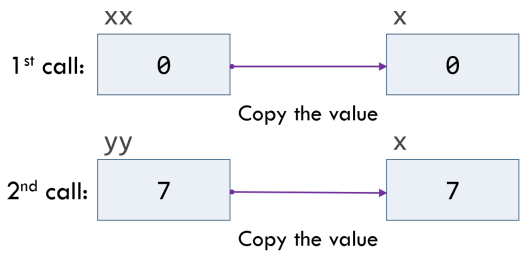
```
// pass-by-value (give the function a copy of the value passed)
int foo(int x) { // x is initialized with argument's value
  x = x + 1;     // give local x a new value
```

```
  return x;
}

int main() {
  int xx {0};
  std::cout << foo(xx) << '\n'; // write: 1
  std::cout << xx << '\n'; // write 0; foo doesn't change xx

  int yy {7};
  std::cout << foo(yy) << '\n'; // write: 8
  std::cout << yy << '\n'; // write 7: foo doesn't change yy
}
```

Since a copy is passed, expression `x = x + 1` in `foo()` only increments local copies of `xx` and `yy` passed in the two calls. This kind of communication between functions is called *pass-by-value* or *call-by-value*. We can illustrate a pass-by-value communication like this:



Pass-by-value is a pretty straightforward and its cost is the cost of copying the value. Pass-by-value can be used to pass around pointers. For example, a function to swap or exchange the values of two variables can be authored like this:

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

A pointer allows us to pass potentially large amounts of data around between functions at a low cost – instead of copying the data we simply copy a pointer to the data. That is, we pass a function the address where the data is stored as a pointer value. The type of the pointer determines what can be done to the data through the pointer. For example:

```
// compute the sum of the elements of a potentially large array
int accumulate(int const *p, int size) {
  int sum {*p};
  for (int i {1}; i < size; ++i) {
    sum += *(p + i);
  }
  return sum;
}
```

Using a pointer differs from using the name of an object in a few ways:

- We use a different syntax, for example, `*p` instead of `obj` and `(*p).m` or `p->m` rather than `obj.m`.
- We can make a pointer point to different objects at different times.

- We must be more careful when using pointers than when using an object directly: a pointer may be a `nullptr` or point to an object that wasn't the one we expected.

These differences can be annoying.

- For example, some programmers find `foo(&x)` ugly compared to `foo(x)`.
- Worse, managing pointer variables with varying values and protecting code against the possibility of `nullptr` can be a significant burden.
- Finally, when we want to overload an operator, say `+`, we want to write `x+y` rather than `&x+&y`.

The mechanism in C++ addressing these concerns is called a *lvalue reference*. Recall that lvalue is an expression representing an object whose address you can take.

- Like a pointer, a lvalue reference is an *alias* for an object. That is, a lvalue reference is an *alternative name* for an object.
- A lvalue reference is usually implemented to hold the machine address of an object.
- A lvalue reference does not impose performance overhead compared to pointers.

But a lvalue reference differs from a pointer in that:

- You access a lvalue reference with exactly the same syntax as the name of an object.
- A lvalue reference always refers to the object to which it was initialized.
- There is no *null lvalue reference* and we may assume that a lvalue reference will always refer to an object.

The main use of lvalue references is to do what pointers do:

- efficiently pass large amounts of data to a function, and
- pass a lvalue reference to a function that can be used by the function to modify the variable being referenced

while avoiding the annoying syntax of pointers and without the possibility of dealing with null pointers.

Syntax for defining and using lvalue references

A lvalue reference is a construct that allows a user to declare a new name for an object, *an alias*. In a type name, the notation `T&` means "lvalue reference to `T`." Since `int&` means a lvalue reference to an `int` object, we can write:

```
int var {1}; // var is name of 4 byte-sized memory chunk with value 1
int &r {var}; // r is lvalue reference or alias to var
int x = r;    // x is another int variable with a different 4 byte-sized
              // memory that is initialized with the value of var
```

The symbol `r` doesn't represent a new variable nor a new storage location; it is just simply another name for `var`. What we've done is bind the name `r` to the memory location known as `var`. Conceptually, think of `r` and `var` as just aliases for the same memory location. That is, any use of `r` is really a use of `var`. `r` can be used anywhere that `var` is used and in the exact same way. Anywhere variable `var` can be used in an expression, it can be replaced with its alias:

```
var = 2; // var is (directly) assigned 2
r   = 4; // var becomes through an alias r 4
std::cout << r << " | " << var << '\n'; // write: 4 | 4
```

> ***A lvalue reference is just an alternative name or alias for a variable.***

To ensure that a lvalue reference is a name for something (that is, that it is bound to an object), we must *initialize the lvalue reference*. For example:

```
int var {1};
int &r1 {var};  // OK: r1 initialized
int &r2;        // ERROR: initializer missing
extern int &r3; // OK: r3 initialized elsewhere
```

Initialization of a lvalue reference is trivial when the initializer is an lvalue (an object whose address you can take):

```
int var  {10};
int &r   {var};  // OK: r is initialized as alias to var
int &rr  {10};   // ERROR: initializer to reference rr is not an lvalue!!!
int &rrr {var+10}; // ERROR: initializer to reference rrr not an lvalue!!!
```

The definition of `r` as a reference to `var` is legal in line 1 since `var` is addressable. Expression `10` represents an rvalue expression and thus the definition in line 2 is illegal. The expression `var+10` on line 4 is an rvalue expression and therefore the definition on line 3 is also illegal.

> ***An lvalue reference must be initialized with the initializer for a "plain" `T&` being an lvalue of type `T`.***

Once initialized, a lvalue reference cannot refer an other variable. This means that the initialization of a lvalue reference is something quite different from assignment to it:

```
int var1 {10}, var2 {20};
int &r {var1}; // OK: r initialized to alias var1
r = var2;      // OK: var1 is assigned value of var2
```

On line 2, `r` is initialized to reference `int` variable `var1`. The assignment expression `r = var2` on line 3 is not making `r` be an alias to a different `int` variable `var2`. Instead, the value stored in variable `var2` is being assigned to the variable `var1` that `r` is aliasing. Consequently, the value of a lvalue reference cannot be changed after initialization; it always refers to the object it was initialized to denote. Thus, the obvious implementation of a lvalue reference is as a *read-only* pointer initialized to point to an object that is dereferenced each time it is used. Since the pointer is read-only, it can never point to another object. It is ok to think about lvalue references that way, as long as one remembers that a lvalue reference isn't an object that can be manipulated the same way a pointer is.

> ***Once initialized, a lvalue reference cannot alias another variable.***

Despite appearances, no operator operates on a lvalue reference:

```
int var {10};
int &rr {var}; // OK: rr can never alias anything else other than var
++rr;          // var is incremented to 11
```

Here, `++rr` doesn't increment lvalue reference `rr`; rather, `++` is applied to the `int` to which `rr` refers, that is, to `var`.

> ***Operators don't operate on a lvalue reference; they operate on the referenced variable.***

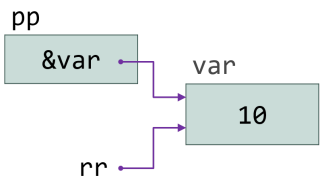
Applying the `&` operator on a lvalue reference doesn't give a pointer to a lvalue reference but gives a pointer to the object aliased by the lvalue reference:

```
int var {10};
int &r {var};
int *p {&r}; // p points to object aliased by r, that is, var
```

Both lvalue references and pointers provide an alternative way to access an existing variable: pointers through the variable's address, and lvalue references through another name for that variable. Since a lvalue reference can be thought of as an *alias* for another variable, a lvalue reference, unlike a pointer, does not take up any additional memory.

```
int var {10}; // var is 4 byte-sized memory chunk with value 10
int *pp {&var}; // pp is 8 byte-sized memory chunk having address of var
int &rr {var};  // rr is lvlaue reference or alias to var
```

The following picture makes it clear that a pointer and lvalue reference are different entities:



The picture shows that `rr` is not a new variable [like a pointer] with memory storage reserved for it; it is just another name for `var`. In some cases, the compiler will optimize away a lvalue reference so that there is no object representing that lvalue reference at run time.

```
std::cout << "var: " << var << " | rr: " << rr << " | *pp: " << *pp << '\n';
std::cout << "&var: " << &var
          << " | &rr: " << &rr << " | pp: " << pp << '\n';
```

You can see that `var` and `rr` do, in fact, represent the same memory chunk with pointer `pp` pointing to that memory chunk:

```
var: 10 | rr: 10 | *pp: 10
&var: 0x7ffd5b702c34 | &rr: 0x7ffd5b702c34 | pp: 0x7ffd5b702c34
```

> ***An lvalue reference does not have an address and is therefore not an object. Therefore, there can be no lvalue references to lvalue references, no pointers to lvalue references, nor an array of lvalue references.***

Lvalue references to constants

Initialization of a lvalue reference is trivial when the initializer is an lvalue [an object whose address you can take]. The initializer for a "plain" `T&` must be an lvalue of type `T`.

```
int var {10};
int &r1 {var}; // OK: var is an lvalue
int &r2 {20};  // ERROR: 20 is not an lvalue
```

The initializer for a `T const&` need not be an lvalue or even of type `T`. In such cases:

- First, implicit type conversion to `T` is applied if necessary.
- Then, the resulting value is placed in a temporary variable of type `T`.
- Finally, this temporary variable is used as the value of the initializer.

Consider the following code fragment:

```
double &dr {1};        // ERROR: lvalue needed as initializer to reference
double const &rcd {1}; // OK
```

The interpretation of the initialization might be:

```
double temp = double {1.0}; // first, create a temporary with right value
double const &rcd {temp};   // then use the temporary as cdr's initializer
```

The temporary created to hold a lvalue reference initializer persists until the end of its lvalue reference's scope.

Main use of references

The main uses of lvalue references is to do what pointers do:

- efficiently pass large amounts of data to a function,
- pass a lvalue reference to a function that can be used by the function to modify the variable being referenced, and
- return values by lvalue reference

while avoiding the annoying syntax of pointers and without the possibility of dealing with null pointers.

Pass-by-`const`-lvalue-reference

Pass-by-value is simple, straightforward, and efficient when we pass small values such as an `int` or a `double`. But what if the value is large, such as an image consisting of several million bytes, a large table of values consisting of thousands of integers, or a lengthy string consisting of thousands of characters. Then, copying can be costly. For example, we could write a function to print out a `vector` of floating-point numbers like this:

```
void print(std::vector<double> v) { // is pass-by-value appropriate?
  std::cout << "{";
  for (int i {0}, len {v.size()}; i < len; ++i) {
    std::cout << v[i] << (i != len-1 ? ", " : "}\n");
  }
}
```

We could use function `print()` for `vector`s of all sizes. For example:

```
void f(int x) {
  std::vector<double> vd1(10);         // small vector
  std::vector<double> vd2(1'000'000); // large vector
  std::vector<double> vd3(x);          // vector of unknown size
  // fill vd1, vd2, and vd3 will values
  print(vd1);
  print(vd2);
  print(vd3);
}
```

This code works, but the first call to `print()` has to copy ten `double`s, the second has to copy a million `double`s, and we don't know how much the third call has to copy. The intent of `print()` is to just print the `vector`s, not to make copies of their elements. One way to pass a variable to a function without copying it is to provide `print()` a pointer to the `vector` variable:

```
void print(std::vector<double> const *pv) { // pass-by-const-pointer
  std::cout << "{";
  for (int i {0}, len {pv->size()}, i < len; ++i) {
    std::cout << (*pv).[i] << (i != len-1 ? ", " : "}\n");
  }
}

print(&vd2);
```

The parameter declaration `vector<double> const *pv` means `pv` is a pointer to *read-only* `vector` of `double`s with the `const` enabling the compiler to prevent changes to the container. The major disadvantage of pointers is that there is no test in `print()` for a `nullptr`. The programmer probably decided to reduce the overhead of a test to make the function more efficient. A second albeit minor disadvantage is the awkward and cumbersome syntax on lines 3, 4, and 8.

We've learnt that another way to pass a variable to a function without copying is to provide function `print()` an *lvalue reference* to the `vector` variable:

```
void print(std::vector<double> const &v) { // pass-by-const-lvalue-reference
  std::cout << "{";
  for (int i {0}; i < v.size(); ++i) {
    std::cout << v[i] << (i != v.size()-1 ? ", " : "}\n");
  }
}
```

The `&` means "reference" and the `const` is there to stop `print()` from inadvertently modifying its argument. Apart from the change to the parameter declaration, all is the same as before; the only change is that instead of operating on a copy, `print()`'s parameter now *refers* back to the argument through the reference. Note the phrase "refer back"; such parameters are called references because they "refer" to objects defined elsewhere. We can call this `print` exactly as before:

```
void f(int x) {
  std::vector<double> vd1(10); // small vector
  std::vector<double> vd2(1'000'000); // large vector
  std::vector<double> vd3(x); // vector of unknown size
  // fill vd1, vd2, and vd3 will values
  print(vd1);
  print(vd2);
  print(vd3);
}
```

Pass-by-`const`-reference is an useful and popular mechanism because a test for *null reference* is not required and unlike pointers, the syntax is similar to pass-by-value semantics. More importantly, a `const` reference has the useful property that we can't accidentally modify the object passed. For example, if we inadvertently tried to assign to an element from within `print()`, the compiler would catch it:

```
void print(std::vector<double> const &v) { // pass-by-const reference
  // ...
  v[i] = 7; // ERROR: v is a const and therefore is not mutable
  // ...
}
```

As a second example, consider a function `my_find()` that searches for a `string` in a `vector` of `string`s:

```
int my_find(std::vector<std::string> vs, std::string s);
```

Pass-by-value is unnecessarily costly. If the container contains thousands of `string`s, the time spent even on a fast computer would be noticeable. So, we could improve `my_find()` by making it take its parameters by `const` reference:

```
int my_find(std::vector<std::string> const &vs, std::string const &s) {
  for (int i {0}, len {vs.size()}; i < len; ++i) {
    if (vs[i] == s) {
      return i;
    }
  }
  return -1;
}
```

We can also specify *references to arrays*:

```
int accumulate(int const (&ra)[N]) {
  int sum{}; // zero out sum
  for (int elem : ra) { // this is range-for statement
    sum += elem;
  }
  return sum;
}
```

Pass-by-lvalue-reference

But what if we did want a function to modify its parameters? Sometimes, that's a perfectly reasonable thing to wish for. For example, we might want a function `init()` that assigns values to `vector` elements:

```
void init(std::vector<double> &v) { // pass-by-reference
  for (int i {0}, len {v.size()}; i < len; ++i) v[i] = i;
}

void g(int x) {
  std::vector<double> vd1(10);         // small vector
  std::vector<double> vd2(1'000'000); // large vector
  std::vector<double> vd3(x);          // vector of unknown size
  // fill vd1, vd2, and vd3 will values
  init(vd1);
  init(vd2);
  init(vd3);
}
```

Here, we wanted `init()` to modify the `vector` argument `vd1` (and `vd2` and `vd3`), so we did not copy [did not use pass-by-value] or declare the lvalue reference `const` [did not use pass-by-`const` lvalue reference] but simply passed a *plain lvalue reference* to each of the `vector`s.

The key property of lvalue references, that a lvalue reference can be a convenient shorthand for some object, is what makes them useful as parameters. Useful because a lvalue reference can be used to specify a function parameter so that the function can change the value of an object passed to it. For example:

```
int increment(int &aa) {
  ++aa;
  return aa;
}

int main() {
  int xx {0};
  std::cout << increment(xx) << "\n"; // write: 1
  std::cout << xx << "\n";            // write: 1

  int yy {7};
  std::cout << increment(yy) << "\n"; // write: 8
  std::cout << yy << "\n";            // write: 8
}
```

The semantics of argument passing are defined to be those of initialization, so when called, `increment()`'s parameter `aa` becomes another name for `xx` in the first call to `increment()` and `aa` becomes another name for `yy` in the second call to `increment()`.

Pass-by-lvalue-reference is clearly a very powerful mechanism: we can have a function operate directly on any object to which we pass a reference. For example, swapping two values is an important operation in many algorithms, such as sorting. Using lvalue references, we can write a function that swaps `double`s like this:

```
void my_swap(double &lhs, double &rhs) {
  double tmp {lhs};
  lhs = rhs;
  rhs = tmp;
}

int main() {
  double x {10}, y {20};
  std::cout << "Before swap: x = " << x << " | y == " << y << "\n";
  my_swap(x, y);
  std::cout << "After  swap: x = " << x << " | y == " << y << "\n";
}
```

`my_swap()`'s parameters `lhs` and `rhs` - both of type lvalue reference to `double` - *refer* back to arguments `x` and `y`, respectively.

> ***Note: When you pass a parameter by reference, you are actually passing an address to the function. Roughly speaking, you get pass-by-address semantics with pass-by-value syntax. The compiler is doing all of the necessary dereferencing for you behind the scenes.***

The standard library provides function `swap()` for every type that you can copy that possibly looks like this:

```
template <typename T>
void swap(T& lhs, T& rhs) {
  T tmp = lhs;
  lhs   = rhs;
  rhs   = tmp;
}
```

This is called a *template function*. We'll discuss this topic later in the semester and is provided here in case you were wondering how the standard library implements the `swap` function for a variety of types.

Output-only lvalue reference parameters

Suppose we wish to author a function to compute and "return" the two roots of a quadratic equation:

$$ax^2 + bx + c = 0$$

$$\implies \begin{cases} x_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a} \end{cases}$$

It is well known that functions can return a single object. However, the quadratic solution function must output two values. One way for the function to output two values to the function caller is to use "output-only" lvalue reference parameters. In the following code fragment, lvalue reference parameter `r1` and `r2` are examples of parameters specified only to output values from the function to the function caller.

```
double discriminant(double a, double b, double c) { // helper function
  return b * b - 4 * a * c;
}
```

```
// r1 and r2 are output-only lvalue reference parameters
void quadratic(double a, double b, double c, double &r1, double &r2) {
  double sq_disc {std::sqrt(discriminant(a, b, c))};
  double den     {2.0 * a};
  r1 = (-b + sq_disc) / den; // r1 and r2 were passed in as references
  r2 = (-b - sq_disc) / den;
}

int main() {
  double a {1.0}, b {4.0}, c {2.0}, root1, root2;
  quadratic(a, b, c, root1, root2);

  std::cout << "a = " << a << ", b = " << b << ", c = " << c << "\n";
  std::cout << "root1 = " << root1 << " | root2 = " << root2 << "\n";
}
```

Pass-by-value vs. pass-by-lvalue-reference

When should you use pass-by-value, pass-by-reference, and pass-by-`const`-lvalue-reference? Consider first a technical example:

```
void f(int a, int &r, int const &cr) {
  ++a;  // change the local a
  ++r;  // change the object referred to by r
  ++cr; // ERROR: cr is read-only!!!
}
```

If you want to change the value of the object passed, you must use a non-`const` lvalue reference: pass-by-value gives you a copy and pass-by-`const`-lvalue-reference prevents you from changing the value of the object passed.

For the simplest, least error-prone, and most efficient code, the rules of thumb are:

1. Use pass-by-value to pass *very small* objects. By very small, we mean one or two `int`s, one or two `double`s, or something similar.
2. Use pass-by-`const`-lvalue-reference to pass large objects that you don't need to modify.
3. Return a result rather than modifying an object through a lvalue reference parameter.
4. Use pass-by-lvalue-reference only when you have to. This happens when you want to manipulate containers such as `vector`s and other large objects and for functions that change several objects.

The third rule reflects that you've a choice when you want to use a function to change the value of a variable. Consider:

```
int  incr1(int a)  { return a+1; } // return new value as the result
void incr2(int &a) { return ++a; } // modify object passed as reference

int x = 7;
x = incr1(x); // pretty obvious what is going on here
incr2(x);     // pretty obscure: could be pass-by-value or pass-by-reference
```

`incr2(x)` notation doesn't give a clue to the reader that `x`'s value is being modified, the way `x = incr1(x)` does. Consequently, "plain" lvalue references should be used only where the name of the function gives a strong hint that the lvalue reference argument is modified.

Return-by-lvalue-reference

Lvalue references can also be used as return types. This is mostly used to define functions that can be used on both the left-hand and right-hand sides of an assignment. Returning lvalue references is especially useful for overloaded operators that can be used on both the left-hand and right-hand sides of an assignment. The most common examples are member functions `std::vector::operator[]()` and `std::string::operator[]()`.

```
std::vector<int> v(10); // create an int vector of size 10
v[5] = 10;    // v[5] evaluates to int&
int x = v[5]; // v[5] evaluates to int&

std::string s("tomorrow");
s[0] = 'T';     // s[0] evalutes to char&
char ch = s[2]; // evaluates to char&
```