Table of content

Variables, Address Types and Reference Types in C++

by Ulrich Grude, Beuth University of Applied Sciences

Abstract: This is an attempt to explain the operators `*` and `&` and the terms *address type* and *reference type* as simple as possible (avoiding multiple words for the same thing), using *buoys* (from Algol-68) as a graphical representation for variables.

There are various kinds of types in C++, among them *address types* (or: pointer types, e.g. `int *`, `string *` and `int * *`) and *reference types* (e.g. `int &` and `string &`). Theses types and two operations connecting them (`*` and `&`) shall be described here in detail, after some even more basic terms (variable, unmodifiable variable, constant etc.) have been introduced.

In the paragraphs to come the following entities are assumed to be defined:

```
// Variables of various types:
int    iv=171,      iva=251,      ivb=-31;
short  sv=17,       sva=25,       svb=-3;
double dv=17.5,     dva=25.5,     dvb=-3.5;
bool   bv=true,     bva=false,    bvb=true;
string tv="Hello ", tva="Sonja!", tvb="How are you?";

// Arrays of various types:
int    ir[] = {111,   222,      333,    444,    555};
short  sr[] = {11,    22,       33,     44,     55};
double dr[] = {11.1,  22.2,     33.3,   44.4,   55.5};
bool   br[] = {false, true,     true,   false,  true};
string tr[] = {"Hi ", "Sunny!", " How", " are", " you?"};
```

1. Variables, unmodifiable variables and constants

A *variable* consists essentially of two parts: an *address* and a *value*. Some variables have additionally a *name* (or several names) or/and a *target value*.

Names of variables are chosen by the *programmer*. Addresses are chosen by the *exer* (the entity which executes the program written by the programmer). In the following examples, the addresses chosen by a fictitious but realistic exer are shown. These addresses are 4-digit hex numbers. Different exers may choose different addresses.

Example-01: Variables and their parts

As buoys the variables defined above may look as follows:

```
1  Name   Address  Value
2 |iv  |--<3000>--[171]
3 |sv  |--<300c>--[17]
4 |dv  |--<3018>--[17.5]
5 |bv  |--<3030>--[true]
6 |tv  |--<6014>--["Hello "]
```

The first buoy (in line 2) represents a variable with name `|iv|`, address `<3000>` and value `[171]`.

In a buoy, *names* will always be enclosed in bars `|...|`, *addresses* in angle brackets `<...>` and *values* in square brackets `[...]`. Spaces after names are not significant.

The value of a variable, e.g. `iv`, can be changed with an assignment. This is not allowed if the variable is defined with the type-specifier `const`. Here, such variables are called *unmodifiable variables* (saving the term *constant* for another kind of things, see below).

Example-02: Unmodifiable variables of various types

```
int    const uiv = 171;
short  const usv = 17;
double const udv = 17.5;
bool   const ubv = false;
string const utv = "Sonja!";
```

Instead of `int const` one can write `const int` too.

A *constant* consists of exactly two parts: A name and a value. Thus, a constant does not have an address.

The connection between the *name* of a constant and its *value* has to be established by the *programmer* with a definition. Since programmers may make errors, the value of a constant may be erroneous and not what the name suggests.

Note: *Literals* like e.g. `123`, `0.5`, `'A'`, `"Hello"` etc. are close relatives of *constants*, in that they too are some kind of "names for values". The difference is: The connection between a *literal* and its *value* is defined by (the language and) the exer. Therefore, in a certain sense, it can not be erroneous. The C/C++-standard does not define the values of many literals (e.g. `'A'`, `'B'`, ..., `0.1f`, `0.1` etc.) and different exers use different values, but those differences do not count as errors. Fortunately, there are a few counterexamples: The literals `10`, `0xA`, `0Xa` and `012` seem to have the same value, i.e. *ten*, whatever exer you use :-).

In C/C++ constants can be defined e.g. with the preprocesser command `#define` like that:

```
#define VAT_PERCENTAGE = 2.5
```

The value of this constant may be wrong.

Besides `#define`-constants there are other kinds of constants C/C++: *Arrays* and *functions*. The name of an array is (technically speaking) the name of a *constant*, the value of which is an *address*.

Example-03: The array (or: address constant) `ir` (defined above) as a buoy

```
13 |ir  |--[<4034>]
14          <4034>--[111]
15          <4038>--[222]
16          <403c>--[333]
17          <4040>--[444]
18          <4044>--[555]
```

In line 13, the angle brackets `<...>` specify, that `4034` is an address. The additional square brackets `[<...>]` express, that the address is a value, i.e. the value of the address constant `ir`. Only in the buoy of a constant is the name (e..g. `ir`) directly connected to the value (e.g. `[<4034> ]`).

The value `4034` is the address of the 0th component `ir[0]` of the array `ir`. In the example this component is a variable, consisting of the address `<4034>` and the value `[111]`. The next component hast the address `4038` and the last component `ir[4]` starts at the address `4044`.

That `ir` is an *address constant* has two consequences (which can be checked easily):

1. If you output `ir` to the screen (e.g. with `cout << ir << endl;`) an address-like number (e.g. `0x4034`) will appear, not the number `111` stored at that address.

2. If you try to assign a value to `ir` (e.g. with `ir=ir;`) your exer will reject your program.

The name of a function also is (technically speaking) the name of an address constant.

2. Address types, address variables, address values and addresses

AT-rule (basic version) : In C/C++ for (nearly) every type T there is an *address type* (or: pointer type) T* (pronounced: "Address of a T variable" or shorter: "Address of T").

Here the * after the T is part of the type name, not an operator. You may insert as much whitespace between the T and the * as you like (0 spaces or 1 space are recommended).

Example-01: Address types

Type	**Address type**	**pronounced**
int	inT*	Address of int
double	double *	Address of double
string	string *	Address of string

The AT-rule may also be applied to *address types* (i.e. it may be applied recursively)

Example-02: Multi star address types

Type	**Address type**	**pronounced**
int *	int * *	Address of address of int
double * *	double * * *	Address of address of address of double
string * * *	string * * * *	Address of address of address of address of string

The **AT-rule** may **not** be applied to so-called *reference types*, which will be discussed below in chapter 6. For a reference type int & there is no address type int & *. Instead you probably want to use the type int*.

Def.: An *address variable* is a variable of an address type (no surprises here :-).

Example-03: An address variable and its buoy

```
int* aiv = new int(171);
```

The following buoy represents, what the exer generates when executing the above definition:

```
|aiv |--<f284>--[<b3c>]--[171]
```

This buoy can be interpreted in two ways:

Interpretation 1: There are two "overlapping" variables:
a variable of type int* with name aiv, address <f284> and value [<b3c>] and
a variable of type int without a name and with address <b3c> and value [171].

bc3 is the "overlap": it is the *value* of the first variable and at the same time the *reference* of the second. The second variable (of type int) is generated by the expression new int(171).

Interpretation 2: There is only one variable
with name aiv, reference <f284>, value [<b3c>] and *target value* [171].

Interpretation 1 is more fundamental and in some sense "cleaner", but a bit harder to describe and use. **Interpretation 2** introduces a new basic term (target value) in a sort of "ad hoc" manner, but in many cases is easier to describe and use.

The following ASCII-diagram depicts both interpretations of the buoy:

```
Interpretation 1: Name1  Address1 Value1
                   |       |     Address2 Value2
                   ↓       ↓        ↓       ↓
Buoy:             |aiv |--<f284>--[<b3c>]--[171]
                   ↑       ↑        ↑       ↑
                   |       |        |       |
Interpretation 2: Name   Address    Value   Target-value
```

End of Example-03.

In what follows, both interpretations will be used.

Remember: There are two kinds of `float` values: Those, which represent *numbers*, and those, which don't. The latter are called NaN values or **NaNs** (NaN: Not a Number).

Similarly: At least one value, which may be assigned to an address variable, is not an address and we will call this value **NanA** (Not an Address).

In an appropriate context this value is denoted by the literal 0, and a constant `NULL` with this value is defined in the header file `<cstddef>`. Here we will assume that a constant `NanA` with this value has been defined, like in the following example.

Example-04: Address variables with the address value `NanA` (alias `NULL`, alias `0`)

```
#define NanA 0
#define NULL 0

int* aiva = NanA;
int* aivb = NULL;
int* aivc = 0;
```

Now each of the three variables has a a value which is not an address.

In buoys we will always use the `NanA` constant (and not 0 or NULL):

```
|aiva|--<ff00>--[NanA]
|aivb|--<ff04>--[NanA]
|aivc|--<ff08>--[NanA]
```

Example-05: Addresses of addresses of ... represented as buoys:

The following address variables

```
int*   a1 =                      new int(171);
int**  a2 =           new int* (new int(172));
int*** a3 = new int** (new int* (new int(173)));
```

represented as buoys may look as follows:

```
|a1|--<1000>--[<A000>]--[171]
|a2|--<1004>--[<A004>]--[<B000>]--[172]
|a2|--<1008>--[<A008>]--[<B004>]--[<C000>]--[173]
```

Summary: An address *type* is a type with one or more stars `*` in its name (e.g. `int*`, `short**`, `string***`). Variables and values of such a type are called *address variables* and *address values*, respectively. An address value is either an address or equal to `NanA`.

Address types are often called *pointer types*.

3. R-values and L-values

Remember: *Expressions* are syntactic entities (the programmer may write them into his program). *Values* are semantic entities (only the exer will compute and handle them during the execution of a program). An expression is a command (given by the programmer to the exer) to compute a value.

A variable has at least two parts: An *address* and a *value*. Sometimes the address is called the *L-value* of the variable, and the value is called the *R-value* of the variable.

Example-01: L-value and R-value as alternatives for address and value

```
 1  Name   Address  Value
 2  Name   L-value  R-Value
 3 |iv  |--<3000>---[171]
 4 |iva |--<3200>---[251]
 5 |ivb |--<3204>---[-31]
```

The terms L-value and R-value arose to describe assignment statements like e.g.

```
 6 iva = ivb;
```

This statement can be translated into English as follows: "Write the value of `ivb` to the address of `iva`", or, with the alternative pair of terms: "Write the L-value of `ivb` to the R-value of `iva`". Depending on whether a variable is located on the left or on the right of the assignment operator, only its L-value or its R-value is used by the exer.

In a source program, *values* are denoted by *expressions*. In this paper expressions, which denote L-values (or R-values), are called L-expressions (or R-expressions, respectively). The name of a variable is an L-expression, a literal is an R-expression. A constant like `NanA` is an R-expression.

In a certain sense an L-expression "is more" than an R-expression. An L-expression denotes directly an address and indirectly the value stored at this address. Thus an L-expression denotes directly an L-expression and indirectly an R-expression. An R-expression, on the other hand, only denotes an R-value. In other words: An L-expression denotes a variable (which consist of an L-expression and an R-expression), whereas an R-expression "only" denotes a value (i.e. an R-value).

Note: An L-expression has to denote an *address*, and not a `NanA` value.

L-R-rule: At every location within a source program, where an R-value is expected, an L-value may be used instead. In such a case the exer will automatically take the R-value of the L-value.

Example-02: An L-expression used instead of an R-expression

```
 7  iva = ivb;
```

On the right side of the assignment operator, an R-value is expected. In line 7, the programmer has written the name of a variable (`ivb`) there, which is an L-value. The exer will automatically take the value of `ivb` which is an R-value.

In the following examples (of L- and R-expressions), L-expressions will always be written *before* and R-expressions *after* an assignment operator, to emphasize "their left/right character".

Example-03: L-expressions

Every simple name of a variable is an L-expression:

```
 8  iv=...; sv=...; dv=...; aiv=...;
```

Every (expression denoting a) component of an array of variables is an L-expression:

```
 9  ir[0]=...; ir[1]=...; ir[iv]=...; ir[2*iv-3]=...; dr[ir[2*iv-3]]=...;
```

The ternary (or: three-place-) operator `...?...:...` forms an L-expression, if its second and third argument are L-expressions:

```
10  (0<iv && iv<=4 ? ir[iv] : ir[0]) = ...;
```

This command will assign some value either to the variable `ir[iv]` or to `ir[0]`, depending on whether the expression `0<iv && iv<=4` evaluates to `true` or `false`.

Example-04: R-expressions

Every literal is an R-expression:

```
11  ...=17;   ...=3.5;   ...="Hello!";
```

Most operators form R-expressions:

```
12  ...=iv+1;    ...=2*sv+321;   ...=ir[0]+1;   ...=5*ir[2*iv-3]-1;
13  ...=iv++;    ...=++iv;       ...=iv--;      ...=--iv;
```

A first exception to this rule is shown in the previous example, line 10.

If `f` is a non-void function with a "normal" return type (e.g. `int` or `double` or `string` or `string*` or `int**` etc.), then every call of `f` is an R-expression:

```
14  ...=sin(3.141);  ...=sqrt(2.0*cos(3.5));
```

But on the other hand: Their are non-void functions with a so-called *reference type* as return type. Every call of them is an L-expression. Reference types will be discussed in chapter 6.

The name of a constant is an R-expression:

```
15 #define VAT 22.5     // Another erroneous constant?
16 #define PI   3.141   // A pretty bad approximation for π.
17 ...=MWST;   ...=PI;
18 ...=ir;     ...=dr;  // ir and dr are address constants!
```

End of Example-04.

Note: There are L-expressions, which are not allowed on the left side of an assignment, e.g. the name of an *unmodifiable variable*. All names of variables are L-expressions, but in

```
19 int const uiv = 171;
20 uiv = 172;
```

line 20 is not allowed, because `uiv` is defined as `const`.

4. The variable operator * and the address operator &

As their names suggest, the variable operator `*` and the address operator & have to do with variables and addresses.

The address operator `&` maps a variable to its address.
The variable operator `*` maps an address to its variable.

In other words:

The address operator `&` maps an L-value (a variable) to a specific kind of R-value (its address).
The variable operator `*` maps certain R-values (the address of a variable) to an L-value (its variable).

The following ASCII-diagram illustrates these explanations with buoys:

```
  Variable      Operator    Address
<abc>--[def]    --- & -->    <abc>
<abc>--[def]    <-- * ---    <abc>
```

Example-01: Applying the address operator `&` to variables

The expression `&iv` denotes the address of the variable `iv`.
The expression `&ir[0]` denotes the address of the variable `ir[0]`.
The expression `&dr[ir[2*iv-3]]` denotes the address of the variable `dr[ir[2*iv-3]]`.

"Normal expressions" denote "normal values" (e.g. `int`-values, `double`-values, `string`-values etc.). Address expressions denote address values (i.e. addresses or the value `NanA`).

Example-02: One normal expression and three address expressions

```
int  iv   = 171;    // The literal 171 denotes a normal value
int* aiva = &iv;
int* aivb = new int(-31);
int* aivc = NanA;   // NanA denotes an address value, but not an address
```

As buoys the four variables of this example may look as follows:

```
 Name   Address Value
|iv  |--<5000>---[171]
                   ↑
                   |
|aiva|--<ef04>--[<5000>]
|aivb|--<ef00>--[< a68>]--[-31]
|aivc|--<eefc>--[<NanA>]
```

Here, the address expression `&iv` denotes the address `<5000>`,
the expression `new int(-31)` returns the address `<a68>`, (don't ask why `a68`!) and
the constant `NanA` denotes an address value, but not an address.

Here comes a (nearly) complete description of the variable operator `*`:

Let RE be an R-expression, which denotes (not the value `NanA` but) the address of a variable V.
Then `*RE` is an L-expression, denoting ("all parts of") the variable V.

Example-03: Applying the variable operator `*` to addresses

The address variable `aiv` (see previous example) has the value `[<5000>]`.
The expression `*aiv` denotes the variable `<5000>--[171]` (which also has the name `iv`).

The expression `*aivb` denotes the variable `<a68>--[-31]` (which happens to have no name).

The address variable `aivc` has the value `[<NanA>]`, which is not an address. Therefore
the expression `*aivc` does not denote a variable, but throws an exception.

The address expression `&iv` denotes the address `<5000>`.
The expression `*&iv` denotes the variable `iv`. So does the expression `*&*&*&iv`.

Example-04: Swapping values of variables using *address types*

Assume we have two variables

```
double d1 = 1.7;
double d2 = 2.5;
```

the buoys of which look like

```
|d1|--<A010>---[1.7]
[d2|--<A014>---[2.5]
```

By calling a void-function `swap01` and passing it our variables as parameters, we want to swap the values of `d1` and `d2`. One way to do that works as follows:

We equip the function `swap01` with parameters of the address type `double*`:

```
void swap01(double* a1, double* a2) {
   double tmp = *a1;
   *a1        = *a2;
   *a2        = tmp;
}
```

When calling this function, we have to pass it the addresses of our variables `d1` and `d2`:

```
swap01(&d1, &d2);
```

During the execution of this call, the buoys of parameters `a1` and `a2` and the corresponding arguments `d1` and `d2` may look as follows:

```
|d1|--<A010>---[1.7]
                 ↑
|a1|--<B000>--[<A010>]

|d2|--<A014>---[2.5]
                 ↑
|a2|--<B004>--[<A014>]
```

During the execution of the function, the L-expressions `*a1` and `*a2` denote the variable `d1` and `d2`, respectively, and the function can change their values.

Example-05: L-expressions and R-expressions are different, even if they have equal values.

The buyos of the variables

```
int  n = 171;
int* a = &n;
```

may look as follows:

```
 Name   L-value  R-value
|n   |--<B000>---[171]
                   ↑
|a   |--<C000>--[<B000>]
```

Now the following holds:

The L-expression `n` has the L-value `<B000>` (and the R-value `[171]`)
The L-expression `a` has the R-value `<B000>` (and the L-value `<C000>`)
The R-expression `&n` has the R-value `<B000>`

On the right side of an assignment statement, the L-expression `n` is allowed, but the R-expression `&n` is not allowed there, although both expressions have the same value `<B000>`.

Example-06: Addresses of addresses of ...

```
int       v0 = 171;
int *     v1 = &v0;
int * *   v2 = &v1;
int * * * v3 = &v2;
```

As buoys these variables may look as follows:

```
|v0|--<4000>---[171]
                 ↑
|v1|--<5000>--[<4000>]
                 ↑
|v2|--<6000>--[<5000>]
                 ↑
|v3|--<7000>--[<6000>]
```

The variable `v3` is of type *address of address of address of* `int` and in this example has the value `[<6000>]`. Similar statements hold for `v2`, `v1` and `v0`.

Remark: The term **address operator** for the operator & is widely used in the C/C++-literature. The term **variable operator** for the operator `*` is not used in the literature. Instead, the terms **dereferencing operator** and **indirection operator** are common. But the common terms obfuscate the simple fact, that e.g. `v1*` is a variable (and besides they sound rather intimidating).

5. Computations with addresses

In C/C++ it is possible, to program simple computations involving address values, e.g. an address and an integral number may be added, or the addresses of two components of the same array may be subtracted from each other.

Let AVA and AVB be two expressions of an address type `T*`, and let `III` be an expression of an integer type (e.g. `int`, `short`, `long`, `unsigned int`, ... etc.). Then the following expressions are allowed and have the indicated type:

```
 1 Expression   Type
 2 AVA + III    T *         // address  plus  number  allowed
 3 AVA - AVB    ptrdiff_t // address  minus address allowed
 4 AVA - III    T *         // address  minus number  allowed
 5 III + AVA    T *         // number   plus  address allowed
```

The following expressions are not allowed:

```
 6 AVA + AVB                // address  plus  address  not allowed
 7 III - AVA                // number   minus address  not allowed
```

The type name `ptrdiff_t` is defined in the header file `<cstddef>` and denotes a signed integral type (depending on the exer), e.g. the type `int`.

The operations multiplication, division and modulo are not allowed for address values.

Example-01: The types of expressions, which do computations with address values

The operator `typeid` may be applied to any expression. It returns an object of class `type_info` (which is defined in the header file `<typeinfo>`). This object contains a function named `name`. An expression like `typeid(A).name()` returns the name of the type of the expression `A`, like in the following examples:

```
 8 typeid(adva).name():      double *
 9 typeid(advb).name():      double *
10 typeid(adva+1234).name(): double *
11 typeid(adva-advb).name(): int          // i.e. ptrdiff_t
12 typeid(adva-1234).name(): double *
13 typeid(1234+adva).name(): double *
```

Lines 8 and 9 show, that `adva` and `advb` are variables of type `double*`.

Note: The type names returned by `typeid(A).name()` are not standardized. The Gnu C++ compiler gcc makes the function `name` return `Pd` (like "Pointer to double") instead of `double *` and `i` instead of `int` etc.

Question: What is the value of `(adr + 1) - adr` (provided that it is allowed, i.e. if `adr` is the address of an array component). This question is harder than it may appear at first sight, but do not run for your calculator, it will not help :-).

The answer depends on the type of the variable which `adr` is addressing.

If `T` is a type, then `size(T)` will return the size of a variable of type `T`, measured in bytes. If `V` is a variable, the `size(V)` similarly returns the the number of bytes occupied by `V`.
Many C/C++-exers use 4 bytes for each `int`-variable, thus `size(int)` will be 4 (some exers use different sizes). And if `adr` is an address of type `T*`, than `adr + 1` will be larger than `adr` by `size(T)`.

Example-02: An addition table for address values

The numbers in the following table depend on the exer, the mileage of your exer may differ:

```
14 typeid(aiv).name():       int *
15 typeid(asv).name():       short *
16 typeid(adv).name():       double *
17 typeid(abv).name():       bool *
18 typeid(atv).name():       string *
19
20 aiv: 1000, aiv+1: 1004, aiv+2: 1008, aiv+3: 100c
21 asv: 2000, asv+1: 2002, asv+2: 2004, asv+3: 2006
22 adv: 3000, adv+1: 3008, adv+2: 3010, adv+3: 3018
23 abv: 4000, abv+1: 4001, abv+2: 4002, abv+3: 4003
24 atv: 5000, atv+1: 5010, atv+2: 5020, atv+3: 5030
```

The lines 14 to 18 document the *types* of 5 address variables. In lines 20 to 24 you can see, how much the values of those variables are increased by applying the operations `+1`, `+2` and `+3` to them. All numbers are hexadecimals.

To be honest: It took some bribing to make the exer choose round values like `1000`, `2000`, ... for the address variables :-).

To access the components of an array named `dra`, usually *index variables* and (L-) expressions like `dra[3]`, `dra[i]`, etc. are used. If the keys for the square brackets `[` and `]` on your keyboard are broken, you can use an *address variable* instead.

Example-03: How to access array components with an address variable

```
25 double    dra[] = {1.11, 2.22, 3.33, 4.44, 5.55}; // An array and
26 int const LEN   = sizeof(dra)/sizeof(dra[0]);     // its length
27
28 for (double * a=dra; a<dra+LEN; a++) {
29    cout << "a: <" << hex << a << ">, *a: " <<  *a << endl;
30 }
```

Here an address variable `a` is used instead of the usual index variable `i` of type `int`, and `dra[i]` has been replaced by `*a` (no need to buy a new keyboard). Note, that `a++` will increase `a` (not by 1 but) by `sizeof(double)`, because `a` is of type `double *`.

As a buoy the array `dra` and its components may look as follows:

```
31 |dra |--[<eec8>]
32          <eec8>--[1.11]
33          <eed0>--[2.22]
34          <eed8>--[3.33]
35          <eee0>--[4.44]
36          <eee8>--[5.55]
```

And the the loop starting in line 28 will output the following lines:

```
a: eec8, *a: 1.11
a: eed0, *a: 2.22
a: eed8, *a: 3.33
a: eee0, *a: 4.44
a: eee8, *a: 5.55
```

Even if you do not plan to access array components with address variables yourself, you should be able to read such accesses, in order to understand the code of other programmers (which may use broken keyboards or take pride in typing as few characters as possible).

Note: The C-Standard (BS ISO/IEC 9899:1999, paragraph 6.5.2.1 Array sub-scripting) defines the meaning of square brackets as follows:

"The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`."

The C++-Standard (BS ISO/IEC 14882:2003, paragraph 5.2.1 Subscripting) contains a very similar definition. The upshot of those definitions: The four L-expressions `r[i]`, `i[r]`, `*(r+i)` and `*(i+r)` have the same meaning (provided that `r` is an array and `i` an `int`-variable). If you want to demonstrate bad taste, always use `i[r]` instead of `r[i]`.

Independent from each other an address *variable* and the variable it addresses (its *target variable*) may be modifiable or unmodifiable. Therefore, the following rule holds:

AT-rule (full version): In C/C++ for (nearly) every type `T` there are 4 *address types* (or: pointer types)

Address type	**pronounced:**
`T       *`	(variable) address of T (variable)
`T const *`	(variable) address of const T
`T       * const`	const address of T (variable)
`T const * const`	const address of const T

To pronounce the names of address types, just read them from right to left (like a normal Arabic or Hebrew text), reading a "*" as "address of" and "const" as "const". Add "variable" to your taste.

Example-05: (un)modifiable address variables and (un)modifiable target variables

```
// A quick way to declare 4 int variables:
int ir[4] = {17, 27, 37, 47};

// 4 address variables of
// different types:
                                  // address var.:   target var.:
int       *       ai01 = &ir[0]; // modifiable      modifiable
int const *       ai02 = &ir[2]; // modifiable      unmodifiable
int       * const ai03 = &ir[1]; // unmodifiable    modifiable
int const * const ai04 = &ir[3]; // unmodifiable    unmodifiable

 *(ai01 ++);  //   modifiable address variable, allowed
(* ai01)++ ;  //   modifiable  target variable, allowed
 *(ai02 ++);  //   modifiable address variable, allowed
(* ai02)++ ;  // unmodifiable  target variable, not allowed
 *(ai03 ++);  // unmodifiable address variable, not allowed
(* ai03)++ ;  //   modifiable  target variable, allowed
 *(ai04 ++);  // unmodifiable address variable, not allowed
(* ai04)++ ;  // unmodifiable  target variable, not allowed
```

Const-rule: The target of an address variable of type `T const *` ("address of const T") may be a modifiable (or an unmodifiable) variable.

At first sight that may sound like a contradiction. Take a second look after the following example:

Example-06: Address variables with and without a license to modify

```
56    float         m     = 1.2; //   modifiable float variable
57    float const   u     = 3.4; // unmodifiable float variable
58
59    float const * amNL = &m;   // address var. No   license to modify
60    float const * auNL = &u;   // address var. No   license to modify
61
62    float       * amWL = &m;   // address var. With license to modify
63 // float       * auWL = &u;   // address var. With license to modify
```

The L-expressions `*amNL` and `*amWL` denote the same (modifiable) variable `m`. But the expression `*amNL` (or rather: the programmer using it) does not have "a license to modify its target variable" (because of the "`const`" in line 59). The expression `*amWL` (or rather: the programmer using it) does have such a license (because there is no "`const`" in line 62 revoking it). Of the following three statements only one is "properly licensed":

```
   *amNL = *amNL + 0.1; // not allowed
   *auNL = *auNL + 0.1; // not allowed
   *amWL = *amWL + 0.1; // allowed
```

Problem-01: Explain, why line 63 is erroneous (and therefore out-commented).

6. Reference types

RT-rule (basic version): In C/C++ for (nearly) every type T there is a *reference type* `T &` (pronounced: reference to T).

Example-01: Reference types

Type	Reference type	pronounced
`int`	`int &`	reference to int
`string`	`string &`	reference to string
`int *`	`int * &`	reference to address of int
`string * *`	`string * * &`	reference to address of address of string

The ampersand & in e.g. `int &` is not an operator, but just a part of the type name (similar to a star in a typename like e.g. `int *`).

Important fact: A *reference type* is **not a type** (i.e. it is neither a blueprint for the construction of variables nor does it consist of a set of values and a set of operations applicable to those values).

A so-called reference type may be used (similar to a real type) in variable declarations, as the type of function parameters or as the return type of functions. But that is not enough to make it a real type. There are no *values* and no *variables* of a reference type.

The least useful (but easiest to explain) use of a so-called reference type is, to give one or more names to a variable (which may already have a name).

Example-02: How to declare a variable with three names

```
 1                // Translation into English:
 2 int    n1 = 17; // Generate an int-variable named n1 with initial value 17
 3 int & n2 = n1; // Let "n2" be another name for n1
 4 int & n3 = n1; // Let "n3" be another name for n1
```

Lines 3 and 4 look deceptively like *variable declarations*. Do not be deceived. Those declarations do not generate variables (least variables of the reference type `int &`), but require that the variable `n1` already has been generated. They only supply that variable with additional names. As a buoy the variable `n1` (alias `n2`, alias `n3`) may look as follows:

```
 5 |n1|-+-<A000>--[17]
 6      ↑
 7 |n2|-+
 8      ↑
 9 |n3|-+
```

For the exer it makes no difference, which of the names you use. After the following statements

```
10 n3 = n1 + 3;
11 n2 = n2 + 2;
12 n1 = n3 + 1;
```

the variable (singular!) has the value `23` (and `23` is not a value of the so-called reference type `int &`, but a plain `int`-value).

The following example shows a more useful use of a reference type.

Example-04: Swapping values of variables using *reference types*

Assume we have two variables

```
13 double d1 = 1.7;
14 double d2 = 2.5;
```

the buoys of which look like

```
15 |d1|--<A010>---[1.7]
16 [d2|--<A014>---[2.5]
```

By calling a void-function `swap02` and passing it our variables as parameters, we want to swap the values of `d1` and `d2`. A second way to do that works as follows (a first way has been shown above in **Example-04** of chapter 4):

We equip the function `swap02` with parameters of the reference type `double*`:

```
17 void swap02(double& a1, double& a2) {
18    double tmp = a1;
19    a1         = a2;
20    a2         = tmp;
21 }
```

When calling this function, we only have to write the plain names of our variables as arguments (no fancy address operator is needed or allowed):

```
22 swap02(d1, d2);
```

The reference type `double&` of the parameter `a1` causes the *address* of `d1` (instead of the the *value* of `d1`) to be passed to the function (and similarly for the parameter/argument `a2/d2`).

During the execution of the call in line 22, the buoys of parameters `a1` and `a2` and the arguments `d1` and `d2` may look as follows:

```
23 |d1|-+-<A010>---[1.7]
24      ↑
25 |a1|-+
26
27 |d2|-+-<A014>---[2.5]
28      ↑
29 |a2|-+
```

The variable `d1` will have `a1` as an additional name (and `d1` will have the additional name `a2`), and since `a1` and `a2` are well known inside the function `swap02`, the values of the variables can be swapped.

Formally, the the first parameter in the definition of of `swap02` is of the reference type `double&`. But "in reality" that means:

1. The first argument in any call of `swap02` has to be of type `double` (which is a real *type*, not a so-called *reference type*)
2. This first argument has to be a *variable* (actually it may be any L-expression).
3. The function `swap02` will supply that variable with an additional name and using that name can change the value of the variable.

The same holds for der the *second* argument in each call of `swap02`.

Thus reference types enable the programmer, to pass a variable `x` to a function in order to have the value of `x` modified. A non-void-function returns exactly 1 (in words: one) result. But if you write a (void- or non-void) function with 10 parameters of (possibly different) *reference types*, (and call it with 10 of "your" variables as arguments) the function can write 10 results into your variables.

Aside: Reference parameters are a very useful invention. But in C/C++ one detail has been criticized: When only reading a call like the one in line 22 (`swap02(d1, d2);`), you can not see, whether the *values* or the *addresses* of the arguments `d1` and `d2` will be passed to the function. In order to understand the call, you must know at least the declaration of the function (you must know whether the parameters are of the real type `double` or of the "reference type" `double&`). The designers of the language C# have found a simple way to avoid such criticism, by introducing two different calls:

```
swap02(d1, d2);          // will pass the values    of the arguments
swap02(ref d1, ref d2);  // will pass the addresses of the arguments
```

Only types are called types in C#.

A second useful use of reference types is shown in the following example.

Example-05: A function with a reference type as its return type

```
30 double & dv(int index) {
31    static double      default = 9.0;
32    static double      dr[5]   = {0.5, 1.5, 2.5, 3.5, 4.5};
33    static const  int LEN     = sizeof(dr)/sizeof(dr[0]);
34
35    if (0 <= index && index < LEN) {
36       return dr[index];
37    } else {
38       return default;
39    }
40 } // dv
```

This function guards an array `dr` (defined inside the function, but surviving any numbers of calls because it is `static`) against accesses with an out-of-bounds index. Whenever it is called, it will return a variable (an L-value, not only an R-value), either one of the elements of the array `dr` (if the `index` is OK) or the variable `default`. The caller can read and modify the value of the returned variable. Calls of `dv` may look as follows:

```
41    double & d1 = dv(2);
42    double & d2 = dv(7);
43
44    cout << setprecision(2) << showpoint;
45    cout << "A d1: " << d1 << ", d2: " << d2 << endl;
46    d1 = d1 + 0.3;
47    d2 = d2 + 0.4;
48    cout << "B d1: " << d1 << ", d2: " << d2 << endl;
```

Theses commands will output the following lines (without the line numbers):

```
49  A d1: 2.5, d2: 9.0
50  B d1: 2.8, d2: 9.4
```

End of Example-05.

Like a normal name of an address variable, an alias-name may or may not have a license to modify its target variable, as the following example shows.

Example-06: Alias names with and without a license to modify (see also Example-06 in chapter 5)

```
51    float         m     = 1.2; //   modifiable float variable
52    float const   u     = 3.4; // unmodifiable float variable
53
54    float const & maNL = m;    // alias for m, No   license to modify
55    float const & uaNL = u;    // alias for u, No   license to modify
56
57    float       & maWL = m;    // alias for m, With license to modify
58 // float       & uaWL = u;    // alias for u, With license to modify
```

Of the following three statements only one is "properly licensed":

```
59    *amNL = *amNL + 0.1; // not allowed
60    *uaNL = *uaNL + 0.1; // not allowed
61    *amWL = *amWL + 0.1; // allowed
```

Problem-01: Explain, why line 58 is erroneous (and therefore out-commented).

RT-rule (full version): In C/C++ for (nearly) every type T there are 2 *reference types*

Reference type	**pronounced**
`T       &`	reference of T (variable)
`T const &`	reference of const T

Often reference types are used to *pass* big objects *by reference* (instead of *by value*) to a function.

Example-07: Big `string`-objects are passed by reference to a function

```
62 void process(string const & str) {
63    // Prints the first and the last char of str:
64    cout << "str.at(0):            " << str.at(0)            << endl;
65    cout << "str.at(str.size()-1): " << str.at(str.size()-1) << endl;
66 // str.at(17) = 'A'; // Not allowed!
67 } // process
68
69 #define MILLION 1000*1000
70
71 string textA(2*MILLION, '?'); // A big string object
72 string textB(3*MILLION, 'X'); // A big string object
73
74 int main() {
75    process(textA);
76    process(textB);
77    ...
78 } // main
```

During the execution of the function call in line 75, the buoy of the argument `textA` and the parameter `str` (of function `process`) may look as follows:

```
79 |textA|-+-<AC00>--["????? ... ?????"]
80         ↑
81 |str|-- +
```

Only a few machine instructions have to be executed to supply the variable `textA` with the additional name `str`. The 2 million question marks in `textA` do not have to be copied. That is a big advantage of *pass by reference* over *pass by value*. At the same time, `textA` can not be modified (inadvertently or maliciously) by the function `process`. Thus, *pass by reference* is no less secure than *pass by value*.

If you want the function `process` to somehow *modify* the strings passed to it, just erase the word `const` in line 62.