2

Objects, Types, and Values

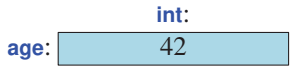
Fortune favors the prepared mind.
– Louis Pasteur

This chapter introduces the basics of storing and using data in a program. To do so, we first concentrate on reading in data from the keyboard. After establishing the fundamental notions of objects, types, values, and variables, we introduce several operators and give many examples of use of variables of types **char**, **int**, **double**, and **string**.

2.1 Input

The "Hello, World!" program just writes to the screen. It produces output. It does not read anything; it does not get input from its user. That's rather a bore. Real programs tend to produce results based on some input we give them, rather than just doing exactly the same thing each time we execute them.

CC To read something, we need somewhere to read into; that is, we need somewhere in the computer's memory to place what we read. We call such a "place" an object. An *object* is a region of memory with a *type* that specifies what kind of information can be placed in it. A named object is called a *variable*. For example, character strings are put into **string** variables and integers are put into **int** variables. You can think of an object as a "box" into which you can put a value of the object's type:



This would represent an object of type **int** named **age** containing the integer value **42**. Using a string variable, we can read a string from input and write it out again like this:

```
// read and write a first name
#include "PPP.h"

int main()
{
        cout << "Please enter your first name (followed by "enter"):\n";
        string first_name;              // first_name is a variable of type string
        cin >> first_name;              // read characters into first_name
        cout << "Hello, " << first_name << "!\n";
}
```

The **#include** and the **main()** are familiar from Chapter 1. Since the **#include** or the equivalent direct use of **import** is needed for all our programs, we'll leave it out of our presentation to avoid distraction. Similarly, we'll sometimes present code that will work only if it is placed in **main()** or some other function, like this:

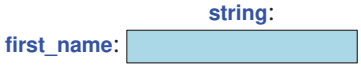
```
cout << "Please enter your first name (followed by 'enter'):\n";
```

We assume that you can figure out how to put such code into a complete program for testing.

The first line of **main()** simply writes out a message encouraging the user to enter a first name. Such a message is typically called a *prompt* because it prompts the user to take an action. The next lines define a variable of type **string** called **first_name**, read input from the keyboard into that variable and write out a greeting. Let's look at those three lines in turn:

```
string first_name;    // first_name is a variable of type string
```

This sets aside an area of memory for holding a string of characters and gives it the name **first_name**:

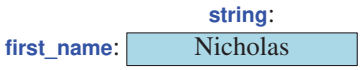


A statement that introduces a new name into a program and sets aside memory for a variable is called a *definition*.

The next line reads characters from input (the keyboard) into that variable:

```
cin >> first_name;   // read characters into first_name
```

The name **cin** refers to the standard input stream (pronounced "see-in," for "character input") defined in the standard library. The second operand of the **>>** operator ("get from") specifies where that input goes. So, if we type some first name, say **Nicholas**, followed by a newline, the string **"Nicholas"** becomes the value of **first_name**:



The newline is necessary to get the machine's attention. Until a newline is entered (the Enter key is hit), the computer simply collects characters. That "delay" gives you the chance to change your mind, erase some characters and replace them with others before hitting Enter. The newline will not be part of the string stored in memory.

AA

Having gotten the input string into **first_name**, we can use it:

```
cout << "Hello, " << first_name << "!\n";
```

This prints **Hello,** followed by **Nicholas** (the value of **first_name**) followed by **!** and a newline (**'\n'**) on the screen:

```
Hello, Nicholas!
```

If we had liked repetition and extra typing, we could have written three separate statements instead:

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

However, we are indifferent typists, and – more importantly – strongly dislike needless repetition (because repetition provides opportunity for errors), so we combined those three output operations into a single statement.

Note the way we use quotes around the characters in **"Hello, "** but not for **first_name**. We use quotes when we want a literal string. When we don't quote, we refer to the value of something with a name. Consider:

```
cout << "first_name" << " is " << first_name;
```

Here, **"first_name"** gives us the ten characters **first_name** and plain **first_name** gives us the value of the variable **first_name**, in this case, **Nicholas**. So, we get

```
first_name is Nicholas
```

2.2 Variables

CC

Basically, we can do nothing of interest with a computer without storing data in memory, the way we did it with the input string in the example above. The "places" in which we store data are called *objects*. To access an object, we need a *name*. A named object is called a *variable* and has a specific *type* (such as **int** or **string**) that determines what can be put into the object (e.g., **123** can go into an **int** and **"Hello, World!\n"** can go into a **string**) and which operations can be applied (e.g., we can multiply **int**s using the * operator and compare **string**s using the **<=** operator). The data items we put into variables are called *values*. A statement that defines a variable is (unsurprisingly) called a *definition*, and a definition can (and usually should) provide an initial value. Consider:

```
string name = "Annemarie";
int number_of_steps = 39;
```

The value after the {=} is called an *initializer*.

You can visualize these variables like this:

You cannot put values of the wrong type into a variable:

```
string name2 = 39;                    // error: 39 isn't a string
int number_of_steps = "Annemarie";    // error: "Annemarie" is not an int
```

The compiler remembers the type of each variable and makes sure that you use it according to its type, as specified in its definition.

C++ provides a rather large number of types. You can find complete lists on the Web (e.g., **cppreference.com**). However, you can write perfectly good programs using only five of those:

```
int number_of_steps = 39;          // int for integers
double flying_time = 3.5;          // double for floating-point numbers
char decimal_point = '.';          // char for individual characters
string name = "Annemarie";         // string for character strings
bool tap_on = true;                // bool for logical variables
```

The reason for the name **double** is historical: **double** is short for "double-precision floating point." Floating point is the computer's approximation to the mathematical concept of a real number.

Note that each of these types has its own characteristic style of literals:

```
39              // int: an integer
3.5             // double: a floating-point number
'.'             // char: an individual character enclosed in single quotes
"Annemarie"     // string: a sequence of characters delimited by double quotes
true            // bool: either true or false
```

That is, a sequence of digits (such as **1234**, **2**, or **976**) denotes an integer, a single character in single quotes (such as **'1'**, **'@'**, or **'x'**) denotes a character, a sequence of digits with a decimal point (such as **1.234**, **0.12**, or **.98**) denotes a floating-point value, and a sequence of characters enclosed in double quotes (such as **"1234"**, **"Howdy!"**, or **"Annemarie"**) denotes a string.

2.3 Input and type

CC

The input operation >> (“get from”) is sensitive to type; that is, it reads according to the type of variable you read into. For example:

```
int main()       // read name and age
{
        cout << "Please enter your first name and age\n";
        string first_name = "???";          // string variable  ("???" indicates "don't know the name")
        int age = –1;                       // integer variable (-1 means "don't know the age")
        cin >> first_name >> age;           // read a string followed by an integer
        cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

So, if you type in Carlos 22 the >> operator will read Carlos into first_name, 22 into age, and produce this output:

```
Hello, Carlos (age 22)
```

Why won't it read (all of) Carlos 22 into first_name? Because, by convention, reading of strings is terminated by what is called *whitespace*, that is, space, newline, and tab characters. Otherwise, whitespace by default is ignored by >>. For example, you can add as many spaces as you like before a number to be read; >> will just skip past them and read the number.

Just as we can write several values in a single output statement, we can read several values in a single input statement. Note that << is sensitive to type, just as >> is, so we can output the int variable age as well as the string variable first_name and the string literals "Hello, ", " (age ", and ")\n".

If you type in 22 Carlos, you'll see something that might be surprising until you think about it. The (misguided) input 22 Carlos will output

```
Hello, 22 (age –1)
```

The 22 will be read into first_name because, after all, 22 is a sequence of characters, and they are terminated by whitespace. On the other hand, Carlos isn't an integer, so it will not be read. The output will be 22 and age's initial value –1. Why? You didn't succeed in reading a value into it, so it kept its initial value.

AA

A string read using >> is (by default) terminated by whitespace; that is, it reads a single word. But sometimes, we want to read more than one word. There are of course many ways of doing this. For example, we can read a name consisting of two words like this:

```
int main()
{
        cout << "Please enter your first and second names\n";
        string first;
        string second;
        cin >> first >> second;                 // read two strings
        cout << "Hello, " << first << " " << second << '\n';
}
```

We simply used >> twice, once for each name. When we want to write the names to output, we must insert a space between them.

Note the absence of initializers for the two **string**s used as targets for input (**first** and **second**). By default, a **string** is initialized to the empty string, that is **""**.

> TRY THIS
>
> Get the "name and age" example to run. Then, modify it to write out the age in number of months: read the input in years and multiply (using the * operator) by 12. Read the age into a **double** to allow for children who can be very proud of being five and a half years old rather than just five.

2.4 Operations and operators

In addition to specifying what values can be stored in a variable, the type of a variable determines what operations we can apply to it and what they mean. For example:

```
int age = –1;
cin >> age;                    // >> reads an integer into age

string name;
cin >> name;                   // >> reads a string into name

int a2 = age+2;                // + adds integers
string n2 = name + " Jr. ";    // + concatenates strings

int a3 = age–2;                // - subtracts integers
string n3 = name – " Jr. ";    // error: - isn't defined for strings
```

XX By "error" we mean that the compiler will reject a program trying to subtract strings. The compiler knows exactly which operations can be applied to each variable and can therefore prevent many mistakes. However, the compiler doesn't know which operations make sense to you for which values, so it will happily accept legal operations that yield results that may look absurd to you. For example:

```
age = –100;
```

It may be obvious to you that you can't have a negative age (why not?) but nobody told the compiler, so it'll produce code for that definition.

Here is a table of useful operators for some common and useful types:

operation	bool	char	int	double	string
assignment	=	=	=	=	=
addition			+	+	
concatenation					+
subtraction			–	–	
multiplication			*	*	
division			/	/	
remainder (modulo)			%		

operation	bool	char	int	double	string
increment by 1			++	++	
decrement by 1			−−	−−	
increment by n			+= n	+= n	
add to end					+=
decrement by n			−= n	−= n	
multiply and assign			*=	*=	
divide and assign			/=	/=	
remainder and assign			%=		
read from s into x	s >> x	s >> x	s >> x	s >> x	s >> x
write x to s	s << x	s << x	s << x	s << x	s << x
equals	==	==	==	==	==
not equal	!=	!=	!=	!=	!=
greater than	>	>	>	>	>
greater than or equal	>=	>=	>=	>=	>=
less than	<	<	<	<	<
less than or equal	<=	<=	<=	<=	<=

A blank square indicates that an operation is not directly available for a type (though there may be indirect ways of using that operation; see §2.9).

We'll explain these operations, and more, as we go along. The key points here are that there are a lot of useful operators and that their meaning tends to be the same for similar types.

Let's try an example involving floating-point numbers:

```
int main()      // simple program to exercise operators
{
    cout << "Please enter a floating−point value: ";
    double n = 0;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
        << '\n';
}
```

Obviously, the usual arithmetic operations have their usual notation and meaning as we know them from primary school. The exception is that the notation for equal is `==`, rather than just `=`. Plain `=` is used for assignment. Naturally, not everything we might want to do to a floating-point number, such as taking its square root, is available as an operator. Many operations are represented as named functions. In this case, we use `sqrt()` from the standard library to get the square root of `n`: `sqrt(n)`. The notation is familiar from math. We'll use functions along the way and discuss them in some detail in §3.5 and §7.4.

> TRY THIS
>
> Get this little program to run. Then, modify it to read an **int** rather than a **double**. Also, ''exercise'' some other operations, such as the modulo operator, **%**. Note that for **int**s **/** is integer division and **%** is remainder (modulo), so that **5/2** is **2** (and not **2.5** or **3**) and **5%2** is **1**. The definitions of integer *****, **/**, and **%** guarantee that for two positive **int**s **a** and **b** we have **a/b * b + a%b == a**.

Strings have fewer operators, but they have plenty of named operations (PPP2.Ch23). However, the operators they do have can be used conventionally. For example:

```
int main()      // read first and second name
{
      cout << "Please enter your first and second names\n";
      string first;
      string second;
      cin >> first >> second;                  // read two strings

      string name = first + ' ' + second;      // concatenate strings
      cout << "Hello, " << name << '\n';
}
```

For strings, **+** means concatenation; that is, when **s1** and **s2** are strings, **s1+s2** is a string where the characters from **s1** are followed by the characters from **s2**. For example, if **s1** has the value **"Hello"** and **s2** the value **"World"**, then **s1+s2** will have the value **"HelloWorld"**. Comparison of **string**s is particularly useful:

```
int main()      // read and compare names
{
      cout << "Please enter two names\n";
      string first;
      string second;
      cin >> first >> second;   // read two strings

      if (first == second)
            cout << "that's the same name twice\n";
      if (first < second)
            cout << first << " is alphabetically before " << second <<'\n';
      if (first > second)
            cout << first << " is alphabetically after " << second <<'\n';
}
```

Here, we used an **if**-statement, which will be explained in detail in §3.4.1.1, to select actions based on conditions.

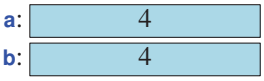
2.5 Assignment and initialization

CC In many ways, the most interesting operator is assignment, represented as **=**. It gives a variable a new value. For example:

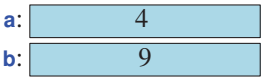
```
int a = 3; // a starts out with the value 3
```


```
a = 4;          // a gets the value 3 (becomes 4)
```

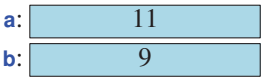

```
int b = a;      // b starts out with a copy of a's value (that is, 4)
```




```
b = a+5;        // b gets the value a+5 (that is, 9)
```




```
a = a+7;        // a gets the value a+7 (that is, 11)
```



That last assignment deserves notice. First of all it clearly shows that `=` does not mean equals – clearly, `a` doesn't equal `a+7`. It means assignment, that is, to place a new value in a variable. What is done for `a=a+7` is the following:

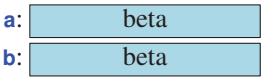
[1] First, get the value of `a`; that's the integer 4.
[2] Next, add 7 to that 4, yielding the integer 11.
[3] Finally, put that 11 into `a`.

We can also illustrate assignments using strings:

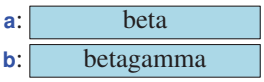
```
sting a = "alpha";      // a starts out with the value "alpha"
```


```
a = "beta";             // a gets the value "beta" (becomes "beta")
```

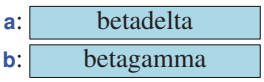

```
string b = a;           // b starts out with a copy of a's value (that is, "beta")
```




```
b = a+"gamma";          // b gets the value a+"gamma" (that is, "betagamma")
```




```
a = a+"delta";          // a gets the value a+"delta" (that is, "betadelta")
```



We use "starts out with" and "gets" to distinguish two similar, but logically distinct, operations:

- *Initialization*: giving a variable its initial value.
- *Assignment*: giving a variable a new value.

Logically assignment and initialization are different. In principle, an initialization always finds the variable empty. On the other hand, an assignment (in principle) must clear out the old value from the variable before putting in the new value. You can think of the variable as a kind of small box and the value as a concrete thing, such as a coin, that you put into it. Before initialization, the box is empty, but after initialization it always holds a coin so that to put a new coin in, you (i.e., the assignment operator) first have to remove the old one ("destroy the old value"). Things are not quite this literal in the computer's memory, but it's not a bad way of thinking of what's going on.

2.5.1 An example: detect repeated words

Assignment is needed when we want to put a new value into an object. When you think of it, it is obvious that assignment is most useful when you do things many times. We need an assignment when we want to do something again with a different value. Let's have a look at a little program that detects adjacent repeated words in a sequence of words. Such code is part of most grammar checkers:

```
int main()
{
        string previous;                    // previous word; initialized to ""
        string current;                     // current word
        while (cin>>current) {              // read a stream of words
                if (previous == current)    // check if the word is the same as last
                        cout << "repeated word: " << current << '\n';
                previous = current;
        }
}
```

This program is not the most helpful since it doesn't tell where the repeated word occurred in the text, but it'll do for now. We will look at this program line by line starting with

```
string current;        // current word
```

By default, a **string** is initialized to the empty string, so we don't have to explicitly initialize it. We read a word into **current** using

```
while (cin>>current)
```

AA This construct, called a **while**-statement, is interesting in its own right, and we'll examine it further in §3.4.2.1. The **while** says that the statement after **(cin>>current)** is to be repeated as long as the input operation **cin>>current** succeeds, and **cin>>current** will succeed as long as there are characters to read on the standard input. Remember that for a **string**, **>>** reads whitespace-separated words. You terminate this loop by giving the program an end-of-input character (usually referred to as *end of file*). On a Windows machine, that's Ctrl+Z (Control and Z pressed together) followed by an Enter (return). On a Linux machine, that's Ctrl+D (Control and D pressed together).

So, what we do is to read a word into **current** and then compare it to the previous word (stored in **previous**). If they are the same, we say so:

```
if (previous == current)          // check if the word is the same as last
    cout << "repeated word: " << current << '\n';
```

Then we have to get ready to do this again for the next word. We do that by copying the **current** word into **previous**:

```
previous = current;
```

This handles all cases provided that we can get started. What should we do for the first word where we have no previous word to compare? This problem is dealt with by the definition of **previous**:

```
string previous;          // previous word; initialized to ""
```

The empty string is not a word. Therefore, the first time through the **while**-statement, the test

```
if (previous == current)
```

fails (as we want it to).

One way of understanding program flow is to "play computer," that is, to follow the program line for line, doing what it specifies. Just draw boxes on a piece of paper and write their values into them. Change the values stored as specified by the program. AA

> TRY THIS
>
> Execute this program yourself using a piece of paper. Use the input **The cat cat jumped**. Even experienced programmers use this technique to visualize the actions of small sections of code that somehow don't seem completely obvious.

> TRY THIS
>
> Get the "repeated word detection program" to run. Test it with the sentence **She she laughed "he he he!" because what he did did not look very very good good**. How many repeated words were there? Why? What is the definition of *word* used here? What is the definition of *repeated word*? (For example, is **She she** a repetition?)

2.5.2 Composite assignment operators

Incrementing a variable (that is, adding 1 to it) is so common in programs that C++ provides a special syntax for it. For example:

```
++counter
```

means

```
counter = counter + 1
```

There are many other common ways of changing the value of a variable based on its current value. For example, we might like to add 7 to it, to subtract 9, or to multiply it by 2. Such operations are also supported directly by C++. For example:

```
a += 7;     // means a = a+7
b -= 9;     // means b = b-9
c *= 2;     // means c = c*2
```

In general, for any binary operator `oper`, `a oper= b` means `a = a oper b`. For starters, that rule gives us operators `+=`, `-=`, `*=`, `/=`, and `%=`. This provides a pleasantly compact notation that directly reflects our ideas. For example, in many application domains `*=` and `/=` are referred to as "scaling."

2.5.3 An example: find repeated words

Consider the example of detecting repeated adjacent words above. We could improve that by giving an idea of where the repeated word was in the sequence. A simple variation of that idea simply counts the words and outputs the count for the repeated word:

```
int main()
{
    int number_of_words = 0;
    string previous;                    // previous word; initialized to ""
    string current;
    while (cin>>current) {
        ++number_of_words;              // increase word count
        if (previous == current)
            cout << "word number " << number_of_words << " repeated: " << current << '\n';
        previous = current;
    }
}
```

We start our word counter at `0`. Each time we see a word, we increment that counter:

```
++number_of_words;
```

That way, the first word becomes number `1`, the next number `2`, and so on. We could have accomplished the same by saying

```
number_of_words += 1;
```

or even

```
number_of_words = number_of_words+1;
```

but `++number_of_words` is shorter and expresses the idea of incrementing directly.

AA Note how similar this program is to the one from §2.5.1. Obviously, we just took the program from §2.5.1 and modified it a bit to serve our new purpose. That's a very common technique: when we need to solve a problem, we look for a similar problem and use our solution for that with suitable modification. Don't start from scratch unless you really have to. Using a previous version of a program as a base for modification often saves a lot of time, and we benefit from much of the effort that went into the original program.

2.6 Names

We name our variables so that we can remember them and refer to them from other parts of a program. What can be a name in C++? In a C++ program, a name starts with a letter and contains only letters, digits, and underscores. For example:

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

The following are not names:

```
2x                  // a name must start with a letter
time@to@market      // @ is not a letter, digit, or underscore
Start menu          // space is not a letter, digit, or underscore
```

When we say "not names," we mean that a C++ compiler will not accept them as names.

If you read system code or machine-generated code, you might see names starting with underscores, such as `_foo`. Never write those yourself; such names are reserved for implementation and system entities. By avoiding leading underscores, you will never find your names clashing with some name that the implementation generated.

Names are case sensitive; that is, uppercase and lowercase letters are distinct, so `x` and `X` are different names. This little program has at least four errors:

```
import std;

int Main()
{
    STRING s = "Goodbye, cruel world! ";
    cOut << S << '\n';
}
```

It is usually not a good idea to define names that differ only in the case of a character, such as `one` and `One`; that will not confuse a compiler, but it can easily confuse a programmer.

> TRY THIS
>
> Compile the "Goodbye, cruel world!" program and examine the error messages. Did the compiler find all the errors? What did it suggest as the problems? Did the compiler get confused and diagnose more than four errors? Remove the errors one by one, starting with the lexically first, and see how the error messages change (and improve).

The C++ language reserves many names as *keywords*, such as `if`, `else`, `class`, `int`, and `module`. You can find complete lists on the Web (e.g., `cppreference.com`). You can't use those to name your variables, types, functions, etc. For example:

```
int if = 7;     // error: if is a keyword
```

You can use names of facilities in the standard library, such as `string` for your own variables, but you shouldn't. Reuse of such a common name will cause trouble if you should ever want to use the standard library:

```
int string = 7;  // this will lead to trouble
```

AA When you choose names for your variables, functions, types, etc., choose meaningful names; that is, choose names that will help people understand your program. Even you will have problems understanding what your program is supposed to do if you have littered it with variables with "easy to type" names like **x1**, **x2**, **s3**, and **p7**. Abbreviations and acronyms can confuse people, so use them sparingly. These acronyms were obvious to us when we wrote them, but we expect you'll have trouble with at least one:

```
mtbf    TLA    myw    NBV
```

We expect that in a few months, we'll also have trouble with at least one.

Short names, such as **x** and **i**, are meaningful when used conventionally; that is, **x** should be a local variable or a parameter (see §3.5 and §7.4) and **i** should be a loop index (§3.4.2.3).

Don't use overly long names; they are hard to type, make lines so long that they don't fit on a screen, and are hard to read quickly. These are probably OK:

```
partial_sum    element_count    stable_partition
```

These are probably too long:

```
the_number_of_elements    remaining_free_slots_in_symbol_table
```

Our "house style" is to use underscores to separate words in an identifier, such as **element_count**, rather than alternatives, such as **elementCount** and **ElementCount**. We never use names with all capital letters, such as **ALL_CAPITAL_LETTERS**, because that's conventionally reserved for macros (PPP2.§27.8), which we avoid. We use an initial capital letter for types we define, such as **Square** and **Graph**. The C++ language and standard library don't use the initial-capital-letter style, so it's **int** rather than **Int** and **string** rather than **String**. Thus, our convention helps to minimize confusion between our types and the standard ones.

XX Avoid names that are easy to mistype, misread, or confuse. For example:

```
Name    names    nameS    foo    f00    fl    f1    fl    fi
```

The characters **0** (zero), **o** (lowercase **O**), **O** (uppercase **o**), **1** (one), **l** (uppercase **i**), and **l** (lowercase **L**) are particularly prone to cause trouble.

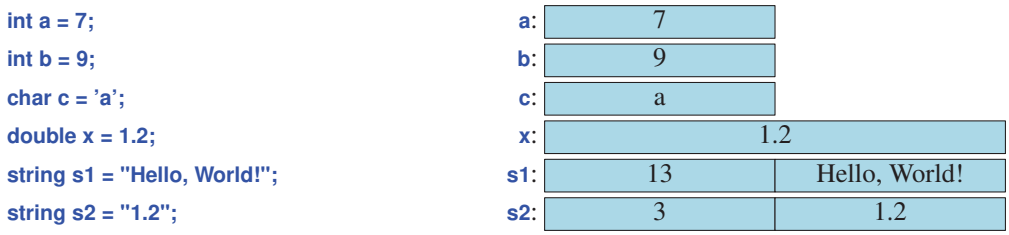
2.7 Types and objects

CC The notion of type is central to C++ and most other programming languages. Let's take a closer and slightly more technical look at types:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of a given type.
- A *value* is a set of bits in memory interpreted according to a type.
- A *variable* is a named object.
- A *declaration* is a statement that gives a name and a type to an object.
- A *definition* is a declaration that sets aside memory for an object.

Informally, we think of an object as a box into which we can put values of a given type. An **int** box can hold integers, such as **7**, **42**, and **–399**. A **string** box can hold character string values, such as

"Interoperability", "operators: +-*/%", and "Old MacDonald had a farm". Graphically, we can think of it like this:



The representation of a string is a bit more complicated than that of an int because a string keeps track of the number of characters it holds. Note that a double stores a number whereas a string stores characters. For example, x stores the number 1.2, whereas s2 stores the three characters '1', '.', and '2'. The quotes for character and string literals are not stored.

Every int is of the same size; that is, the compiler sets aside the same fixed amount of memory for each int. On a typical computer or phone, that amount is 4 bytes (32 bits). Similarly, bools, chars, and doubles are fixed size. You'll typically find that a computer uses a byte (8 bits) for a bool or a char and 8 bytes for a double. Note that different types of objects take up different amounts of space. In particular, a char takes up less space than an int, and string differs from double, int, and char in that different strings can take up different amounts of space.

CC

The meaning of bits in memory is completely dependent on the type used to access it. Think of it this way: computer memory doesn't know about our types; it's just memory. The bits of memory get meaning only when we decide how that memory is to be interpreted. This is similar to what we do every day when we use numbers. What does 12.5 mean? We don't know. It could be $12.5, 12.5cm, or 12.5 gallons. Only when we supply the unit does the notation 12.5 mean anything.

For example, the very same bits of memory that represent the integer value 120 when looked upon as an int would be the character 'x' when looked upon as a char. If looked at as a string, it wouldn't make sense at all and would become a run-time error if we tried to use it. We can illustrate this graphically like this, using 1 and 0 to indicate the value of bits in memory:

This is the setting of the bits of an area of memory (a word) that could be read as an int (120) or as a char ('x', looking at the rightmost 8 bits only). A *bit* is a unit of computer memory that can hold the value 0 or 1.

2.8 Type safety

AA

Every object is given a type when it is defined, and that type never changes. A program – or a part of a program – is type-safe when all objects are used only according to the rules for their type. Complete type safety is the ideal and the general rule for the language. Unfortunately, a C++

compiler cannot by itself guarantee complete type safety for arbitrary code, so we must avoid unsafe techniques. That is, we must obey some coding rules to achieve type safety. There are ways of enforcing such rules, but historically such rules have been considered overly restrictive and have not been consistently enforced. Given older versions of C++ (earlier than recent ISO C++ standards) and techniques adopted from the C language, this was unavoidable and therefore not unreasonable. However, when using modern C++ and modern analysis tools, type safety can be verified for most uses of C++. The ideal is never to use language features that cannot be proven type-safe before the program starts executing: *static type safety*. With the obvious exception of code used to illustrate unsafe techniques (e.g., §16.1.1), the code in this book follows the C++ Core Guidelines [CG] and has been verified to be type safe.

XX The ideal of type safety is incredibly important when writing reliable code. That's why we spend time on it this early in the book. Please note the pitfalls and avoid them. If you don't, you will face much frustration and your code will contain many obscure errors.

AA For example, using an uninitialized variable is not type-safe:

```
int main()
{
    double x;          // we "forgot" to initialize: the value of x is undefined
    double y = x;      // the value of y is undefined
    double z = 2.0+x;  // the meaning of + and the value of z are undefined
}
```

AA Always initialize your variables! Implementations can easily enforce this rule, but unfortunately they typically don't do so by default. Except, fortunately, for types such as **string** and **vector** where default initialization is guaranteed (§2.5, §7.2.3, §8.4.2). Figure out how to enable warnings (often a **–Wall** compiler option) and adhere to them. Doing so will save you a lot of grief.

CC Modern C++ implementations also come with significant static-analysis tools that allow us to prevent more subtle problems. Professionals use such tools extensively, and so will you if you become or aim to become a professional, but at this initial stage of learning just follow the rules and styles used in this book.

2.9 Conversions

In §2.4, we saw that we can't directly add **char**s or compare a **double** to an **int**. However, C++ provides indirect ways to do both. When needed in an expression, a **char** is converted to an **int** and an **int** is converted to a **double**. For example:

```
char c = 'x';
int i1 = c;          // i1 gets the integer value of c
int i2 = c+1000;     // i2 gets the integer value of c added to 1000
double d = i2+7.3;   // d gets the floating-point value of i2 plus 7.3
```

Here, **i1** gets the value **120**, which is the integer value of the character **'x'** in the popular 8-bit character set, ASCII. This is a simple way of getting the numeric representation of a character. To get the value of **i2**, the addition is done using integer arithmetic and gives the value **1120**. The **char** is said to be *promoted* to **int** before the addition.

Similarly, when having a mixture of floating-point values and integer values, the integers are promoted to floating point to give unsurprising results. Here, **d** gets the value **1127.3**.

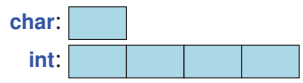
Conversions are of two kinds

- *widening*: Conversions that preserve information, such as **char** to **int**.
- *narrowing*: Conversions that may lose information, such as **int** to **char**.

A widening conversion converts a value to an equal value or to the best approximation of an equal value. Widening conversions are usually a boon to the programmer and simplify writing code.

Unfortunately, C++ also allows for implicit narrowing conversions. By narrowing, we mean that a value is turned into a value of another type that does not equal the original value. XX

Consider **int** and **char**. Conversions from **char** to **int** don't have problems with narrowing. However, a **char** can hold only very small integer values. Often, a **char** is an 8-bit byte whereas an **int** is 4 bytes:



We can't put a large number, such as **1000**, into a **char**. Such conversions are called *narrowing* because they put a value into an object that may be too small (''narrow'') to hold all of it. Unfortunately, conversions such as **double** to **int** and **int** to **char** are by default accepted by most compilers even though they are narrowing. Why can this be a problem? Because often we don't suspect that a narrowing – information destroying – conversion is taking place. Consider: XX

```
double x = 2.7;
// ... lots of code ...
int y = x;              // y becomes 2
```

By the time we assign **x** to **y** we may have forgotten that **x** was a **double**, or that a **double**-to-**int** conversion *truncate*s (always rounds down, toward zero) rather than using the conventional 4/5 rounding (rounding towards the nearest integer). What happens is well-defined, but there is nothing in the **y = x;** to remind us that information (the **.7**) is thrown away.

To get a feel for conversions and an understanding why narrowing conversions must be avoided, experiment. Consider this program that shows how conversions from **double** to **int** and conversions from **int** to **char** are done on your machine:

```
int main()
{
    double d = 0;
    while (cin>>d) {          // repeat the statements below as long as we type in numbers
        int i = d;            // try to squeeze a floating-point value into an integer value
        char c = i;           // try to squeeze an integer into a char
        cout << "d==" << d                // the original double
             << " i=="<< i                // double converted to int
             << " c==" << c               // int value of char
             << " char(" << c << ")\n";   // the char
    }
}
```

TRY THIS

Run this program with a variety of inputs:
- Small values (e.g., 2 and 3).
- Large values (larger than 127, larger than 1000).
- Negative values.
- 56, 89, and 128.
- Non-integer values (e.g., 56.9 and 56.2).

You'll find that many inputs produce "unreasonable" results when converted. Basically, we are trying to put a gallon into a pint pot (about 4 liters into a 500ml glass).

CC Why do people accept the problem of narrowing conversions? The major reason is history: C++ inherited narrowing conversions from its ancestor language, C, so from day one of C++, there existed much code that depended on narrowing conversions. Also, many such conversions don't actually cause problems because the values involved happen to be in range, and many programmers object to "compilers telling them what to do." In particular, the problems with narrowing conversions are often manageable in small programs and for experienced programmers. They can be a source of errors in larger programs, though, and a significant cause of problems for novice programmers. Fortunately, compilers can warn about narrowing conversions – and many do. Adhere to those warnings.

When we really need narrowing, we can use `narrow<T>(x)` to check that `x` can be narrowed to a `T` without loss of information (§7.4.7). When we want rounding, we can use `round_to<int>(x)`. Both are supplied by `PPP_support`.

CC For historical and practical reasons, C++ offers four notations for initialization: For example:

```
int x0 = 7.8;          // narrows, some compilers warn
int x1 {7.8};          // error: {} doesn't narrow
int x2 = {7.8};        // error: ={} doesn't narrow (the redundant = is allowed)
int x3 (7.8);          // narrows, some compilers warn
```

The `=` and `={}` notations go back to the early days of C. We use the `=` notation when an initialization simply copies its initializer and the `{}` and `={}` notations for more complex initializations and when we want compile-time protection against narrowing.

```
int x = 7;
double d = 7.7;
string s = "Hello, World\n";

vector v = {1, 2, 3, 5, 8 };      // see §17.3
pair p {"Hello",17};              // see §20.2.2
```

We reserve `()` initialization to a few very special cases (§17.3).

2.10 Type deduction: auto

You may have noticed a bit of repetitiveness in definitions. Consider:

```
int x = 7;
double d = 7.7;
```

We know that 7 is an integer and that 7.7 is a floating-point number, and so does the compiler. Why then do we have to say int and double? Well, we don't have to unless we want to; we can let the compiler deduce the type from the type of the initializer:

```
auto x = 7;        // x is an int (because 7 is)
auto d = 7.7;      // d is a double (because 7.7 is)
```

This version using auto means *exactly* the same as the one with explicit types. We use auto when, and *only* when, the type is obvious from the initializer and we don't want any conversion. AA

When we use longer type names (§18.5.2, §20.2.1) and in generic programming (§18.1.2) the notational convenience of auto becomes significant. For example:

```
auto z = complex<double>{1.3,3.4};
auto p = make_unique<Pair<string,int>>{"Harlem",10027};   // a unique_ptr<Pair<string,int>> (§18.5.2)
auto b = lst.begin();                                     // lst.begin is a vector<int>::iterator (§19.3.2)
```

Until then, resist the temptation to overuse auto. Overuse of auto can make code obscure so that we get unpleasant surprises.

Drill

After each step of this drill, run your program to make sure it is really doing what you expect it to. Keep a list of what mistakes you make so that you can try to avoid those in the future.

[1] Write a program that produces a simple form letter based on user input. Begin by typing the code from §2.1 prompting a user to enter his or her first name and writing "Hello, first_name" where first_name is the name entered by the user. Then modify your code as follows: change the prompt to "Enter the name of the person you want to write to" and change the output to "Dear first_name,". Don't forget the comma.

[2] Add an introductory line or two, like "How are you? I am fine. I miss you." Be sure to indent the first line. Add a few more lines of your choosing – it's your letter.

[3] Now prompt the user for the name of another friend and store it in friend_name. Add a line to your letter: "Have you seen friend_name lately?"

[4] Prompt the user to enter the age of the recipient and assign it to an int variable age. Have your program write "I hear you just had a birthday and you are age years old." If age is 0 or less or 110 or more, call simple_error("you're kidding!") using simple_error() from PPP_support.

[5] Add this to your letter:

> If your friend is under 12, write "Next year you will be age+1." If your friend is 17, write "Next year you will be able to vote." If your friend is over 70, write "Are you retired?"

Check your program to make sure it responds appropriately to each kind of value.

[6] Add "Yours sincerely," followed by two blank lines for a signature, followed by your name.

Review

[1] What is meant by the term *prompt*?
[2] Which operator do you use to read into a variable?
[3] What notations can you use to initialize an object?
[4] If you want the user to input an integer value into your program for a variable named `number`, what are two lines of code you could write to ask the user to do it and to input the value into your program?
[5] What is `\n` called and what purpose does it serve?
[6] What terminates input into a string?
[7] What terminates input into an integer?
[8] How would you write the following as a single line of code:

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

[9] What is an object?
[10] What is a literal?
[11] What kinds of literals are there?
[12] What is a variable?
[13] What are typical sizes for a `char`, an `int`, and a `double`?
[14] What measures do we use for the size of small entities in memory, such as `int`s and `string`s?
[15] What is the difference between `=` and `==`?
[16] What is a definition?
[17] What is an initialization and how does it differ from an assignment?
[18] What is string concatenation and how do you make it work in C++?
[19] What operators can you apply to an `int`?
[20] Which of the following are legal names in C++? If a name is not legal, why not?

This_little_pig	This_1_is fine	2_For_1_special	latest thing
George@home	_this_is_ok	MineMineMine	number
correct?	stroustrup.com	$PATH	

[21] Give five examples of legal names that you shouldn't use because they are likely to cause confusion.
[22] What are some good rules for choosing names?
[23] What is type safety and why is it important?
[24] Why can conversion from `double` to `int` be a bad thing?
[25] Define a rule to help decide if a conversion from one type to another is safe or unsafe.
[26] How can we avoid undesirable conversions?
[27] What are the uses of `auto`?

Terms

assignment	definition	operation	**cin**
increment	operator	concatenation	initialization
type	conversion	name	type safety
declaration	narrowing	value	decrement
object	variable	widening	truncation
int	**double**	**string**	**auto**
==	**!=**	**=**	**++**
<	**<=**	**>**	**>=**

Exercises

[1] If you haven't done so already, do the **TRY THIS** exercises from this chapter.

[2] Write a program in C++ that converts from miles to kilometers. Your program should have a reasonable prompt for the user to enter a number of miles. Hint: A mile is1.609 kilometers.

[3] Write a program that doesn't do anything, but declares a number of variables with legal and illegal names (such as **int double = 0;**), so that you can see how the compiler reacts.

[4] Write a program that prompts the user to enter two integer values. Store these values in **int** variables named **val1** and **val2**. Write your program to determine the smaller, larger, sum, difference, product, and ratio of these values and report them to the user.

[5] Modify the program above to ask the user to enter floating-point values and store them in **double** variables. Compare the outputs of the two programs for some inputs of your choice. Are the results the same? Should they be? What's the difference?

[6] Write a program that prompts the user to enter three integer values, and then outputs the values in numerical sequence separated by commas. So, if the user enters the values **10 4 6**, the output should be **4, 6, 10**. If two values are the same, they should just be ordered together. So, the input **4 5 4** should give **4, 4, 5**.

[7] Do exercise 6, but with three string values. So, if the user enters the values **Steinbeck**, **Hemingway**, **Fitzgerald**, the output should be **Fitzgerald, Hemingway, Steinbeck**.

[8] Write a program to test an integer value to determine if it is odd or even. As always, make sure your output is clear and complete. In other words, don't just output **yes** or **no**. Your output should stand alone, like **The value 4 is an even number**. Hint: See the remainder (modulo) operator in §2.4.

[9] Write a program that converts spelled-out numbers such as "zero" and "two" into digits, such as 0 and 2. When the user inputs a number, the program should print out the corresponding digit. Do it for the values 0, 1, 2, 3, and 4 and write out **not a number I know** if the user enters something that doesn't correspond, such as **stupid computer!**.

[10] Write a program that takes an operation followed by two operands and outputs the result. For example:

```
+ 100 3.14
* 4 5
```

Read the operation into a string called **operation** and use an **if**-statement to figure out which operation the user wants, for example, **if (operation=="+")**. Read the operands into variables of type **double**. Implement this for operations called **+**, **–**, *****, **/**, **plus**, **minus**, **mul**, and **div** with their obvious meanings.

[11] Write a program that prompts the user to enter some number of pennies (1-cent coins), nickels (5-cent coins), dimes (10-cent coins), quarters (25-cent coins), half dollars (50-cent coins), and one-dollar coins (100-cent coins). Query the user separately for the number of each size coin, e.g., "How many pennies do you have?" Then your program should print out something like this:

```
You have 23 pennies.
You have 17 nickels.
You have 14 dimes.
You have 7 quarters.
You have 3 half dollars.
The value of all of your coins is 573 cents.
```

Make some improvements: if only one of a coin is reported, make the output grammatically correct, e.g., **14 dimes** and **1 dime** (not **1 dimes**). Also, report the sum in dollars and cents, i.e., **.73** instead of **573 cents**.

Postscript

Please don't underestimate the importance of the notion of type safety. Types are at the center of most notions of correct programs, and some of the most effective techniques for constructing programs rely on the design and use of types – as you'll see in Chapter 5 and Chapter 8, and indeed in most of the rest of the book.