C++

Brief Notes on a Vast Subject

BCS230: Foundations of Computer Programming II

Farmingdale — Spring 2026

Alexander Kasiukov

February 14, 2026

Contents

Chapter 1

Foreword

Ἐν ἀρχῇ ἦν ὁ λόγος...
[In the beginning was the *Word*...]
John 1:1

The aim of these notes is to introduce you to C++ *programming language*, and I want to say a couple of general words about *programming* and *languages* first.

Any language can be seen as a way to *describe* a world. A large leap forward comes with realization that the world *described* is the world *created*. Allowing one to ***execute*** the ***source code***, computer programming makes that crucial idea deceptively obvious, to the extent that we don't pause in awe when we say that we "*build*" our programs, instead of merely writing their source code...

However, the world sprung up inside of a physical computer by the code you write must come alive not when you ***run*** it on a machine, but when you just *imagine* it in your head. The fact that correct C++ code can be modeled by an electrical process in a silicon-based physical device is just a lucky coincidence. If you don't imagine a different — more immediate, visceral and intuitive — ***model*** of the world you are creating as you write, read and edit your code[1], you are robbing yourself of the meaning and pleasure that comes with it, making it all but impossible to learn the subject.

The concept of a model is made precise in mathematical ***logic***. A single logical ***theory*** can have many different models that could reveal different aspects of the theory itself. This is studied in the area of logic called ***model theory***.

1.1. If you want to get serious...

Although not required for this class, there are several things you should do if you want to get serious about programming. I list them below in the order of urgency, starting with the most pressing one.

[1] When a variable is introduced, think of a box that can be filled; when a constant is used, imagine a box with a lid that once sealed cannot be reopened, etc. ...

1.1.1. Get C++ on *Your* Computer

If you have a Windows computer, you can follow the Guide for Windows. If you have a Mac, you can use the Guide for Mac. If you have a Unix, e.g. Linux, you probably already know what to do...

If you take the next suggestion (about command line) to heart, you don't need to torture yourself with all the installation steps for the Visual Studio Code, outlined in the Guide for Windows. To use C++ on a command line, you only need a good text editor, a compiler and a debugger.

1.1.2. Use Command Line

This recommendation is as much — if not more so — about *not doing* than it is about *doing* something. **Do not learn to rely on any integrated development environment (IDE)**, such as Visual Studio, or any other. The computers we have in the classroom have Visual Studio preinstalled, configured and ready to use. While it gives some degree of comfort and reassurance, an integrated development environment hides many important steps behind its "Run" button and thus impedes full understanding of the craft, fostering bad habits that don't translate well to large and complicated projects.

Command line interface is the best professional way to communicate with a computer. It is done via ***console***, also know as the "terminal" or the "command prompt". On Windows, you can get the command prompt by pressing the ⊞ + R; then typing "`cmd`" and pressing Enter. On a Mac, press ⌘ + Space; type "`Terminal`" and press Enter. On Linux, you probably already live on the command prompt...

In these notes, I will show all the commands using the syntax of a Bash command prompt. Bash is the default on Linux and Mac, and you can get it on Windows by installing Cygwin. Most of the Bash commands related to this course can be used verbatim on Windows and Mac command prompts, at least as long as you install the GNU toolchain (`g++` compiler and `gdb` debugger) as suggested in the above mentioned Windows Guide. (Sometimes you may need to use `/Option:Value` on Windows in place of the `-Option Value` syntax used in Bash.)

Command line is the tool that can help you understand the life cycle of a C++ ***program***. You should learn how to do — one by one — all the steps that take the source code to an executable[2]:

- use a text editor to write and edit the ***source code*** files[3];
- use the ***preprocessor*** to see the ***expanded source code***, obtained by applying all the macro directives and `#include`'s to the original source[4];

[2] I am just *mentioning* these things here, we will go over these steps in more detail later.

[3] Making as many source files as needed; as an example, assume we made a single file named `main.cpp`

[4] For example, for the single source file `main.cpp` and for the GNU compiler `g++`, this can be achieved by issuing the following command on the command prompt:

```
g++ -E main.cpp > main.i
```

- apply the ***compiler*** to the source code files (expanded or not) to make the individual ***assembly*** files[5];
- apply the ***assembler*** to the assembly files to make the individual (relocatable) ***object*** files[6];
- apply the ***linker*** to all the object files in your project to form a single ***executable***[7];

```
ld \
    -dynamic-linker \
        /lib64/ld-linux-x86-64.so.2  \
        /usr/lib/x86_64-linux-gnu/crt1.o \
    -L /usr/lib/gcc/x86_64-linux-gnu/14 \
        main.o \
            -lm  -lc -lstdc++  \
            -o main
```

Figure 1. For your amusement: full linker command on author's computer.

- use the ***debugger*** to step through the executable to diagnose and correct runtime errors[8];
- once the program is ready, run the program as its user would.

1.1.3. Automate the Build Process

In our class, we will stay in the shallow waters of C++ development with projects rarely involving more than a couple of separate files. However, any serious production has many moving parts, making it necessary to automate and document the build process with a single build script. I would like to suggest that you start using building scripts early — as soon as your projects involve more than one file. I will probably show you how to do automated builds with the help of GNU Make[9].

[5] Continuing from where we left it in the previous footnote, compilation to assembly can be done with

```
g++ -S main.i
```

— or, if starting directly from unexpanded source code, with `g++ -S main.cpp` — resulting in the assembly file `main.s`

[6] Continuing again from where we left it in the previous footnote, assembly into object file can be done with

```
g++ -c main.s
```

— or, if combined with compilation step, via `g++ -c main.cpp` or `g++ -c main.i` — resulting in file `main.o`

[7] Again, starting at the place where the previous step ended, we can do the linking with

```
g++ main.o -o main
```

However, that would involve a bit of cheating, since the `g++` command itself is doing quite a bit of heavy lifting. To reveal what is really happening, one should use the linker `ld` command with all its options listed explicitly. See the Figure 1 for all the gory details.

[8] To enable debugging, the executable must be compiled with the `-g` flag, as in

```
g++ -g main.cpp -o main
```

Once it is done, the debugging can be started with

```
gdb ./main
```

[9] But I don't promise we get to it...

1.1.4. Use Version Control

Version control is another "must have" of software development. By retaining the full history of project evolution, version control systems allow

- to write new code without the risk of breaking the old one;
- implement, track and revert changes;
- collaborate across large teams of developers by permitting individual developers to work simultaneously without interfering with each other;
- maintain institutional history and culture of code development by providing mechanism for code attribution and review.

Learning to use a version control system is certainly outside of the scope of this class, but I highly recommend that you look into it, specifically using Git for the task...

Chapter 2

Review

2.1. The Roots of C++

2.1.1. Unix and C

C++ is rooted in the earlier language called C [2]. It is impossible to understand C++ without getting some sense of C first. C was invented by Dennis Ritchie in 1972 at Bell Labs, to support the efforts of Ken Thompson in porting the UNIX operating system to PDP-11. The first incarnation of UNIX was written in the assembly language specific to the PDP-7 computer[10].

Figure 2. Ken Thompson (left) and Dennis Ritchie (right).

When Thompson decided to move UNIX to PDP-11, he was looking for a language that would be close enough to the physical computer to make it both easy to implement and suitable for writing an operating system in it, yet far enough from it to make porting programs from one computer to another feasible.

2.1.2. Von Neumann Machine

C was written to run on ***von Neumann machine***. The von Neumann machine is an abstraction of a real computer characterized by ***von Neumann architecture***, introduced by John von Neumann in 1945 [1]. This architecture is defined by having single addressable memory shared by both the programs and the data. Von Neumann machine provides the stage for C (and C++) programs to live and run. We will explore it in more detail later. Von Neumann machine

[10]PDP-7 was already old by that time, which probably was one of the reasons it was available as a playground for the new system's development.

Figure 3. John von Neumann.

has proven[11] to be the happy middle ground between a physical computer and the abstract environments of higher-level languages.

2.1.3. C++ and Its Place

C++ language was invented by Bjarne Stroustrup in early 1980s as an extension of C. At that time, C was already a very popular language proven to be

Figure 4. Bjarne Stroustrup.

useful far outside of its original domain. The use of C in the far broader realm than operating system development exposed limitations of its expressive power in modeling ***state*** and ***behavior*** of real life objects. The aim of C++ was to mitigate just that with ***object-oriented*** paradigm. As a result, C++ extends C in the direction of the "abstract" side of the programming languages continuum.

While C++ is trying to keep the connection with the underlying basics of the von Neumann architecture, it is a trying task, as every new iteration of the C++ standard is pushing it farther away from the physical level, almost succeeding by now in making it C++ a ***high-level*** language. Yet at the same time the ability of C++ to open up and reconfigure its own intestines leaves open the possibility of reconnecting it back to that almost-lost basic C level, making C++ unique in the world of programming languages. Even though it takes more and more work as the C++ Standards Committee is producing new specifications, C++ can be still viewed as an *arbitrary* level language.

[11] In no small part — thanks to C and C++.

2.2. Expressions

The most basic unit[12] of C++ code is an ***expression***. Expressions may have ***values*** and ***side effects***. Every time you write an expression in your code, make sure you understand what value and effect, if any, your expression has.

2.2.1. The `main()` function

All *standalone*[13] C and C++ programs must include a `main()` ***function***. Thus the simplest C++ program is this[14]:

```
int main()
{
}
```

What a function is in general will be addressed later in these notes. For now it suffices to say that functions may be either built-in — like `sizeof`[15] or ***defined*** in the code — like the `main()` function above.

Regardless of whether a function is built-in or defined in the code, it can be ***called*** in the code. The `main()` function is special not only because it must be there, but also because it is implicitly called by the operating system when the program's executable is loaded.

2.2.2. Objects and Expressions

We live in the world made of objects, and those objects can sometimes be modeled inside of computer code. Even though the word ***object*** means something more specific in the context of ***object-oriented*** programming languages, we will use it for now in its more colloquial sense to refer to any entity that can be represented in code.

An ***expression*** is a string of characters in the code describing a particular object. When an expression describes a specific object, we say that the expression ***evaluates*** to that object, or, put it differently, that the object in question is the ***value*** of the expression.

The above is a first pass at the definition of an expression. We will later revisit it to add additional possibilities for what an expression may be. Some of those yet-to-be discussed expressions will qualify as expressions without evaluating to an object. Those expressions are called ***void expressions***.

[12] I don't use *building block* here only because the word *block* means something specific in the context of C and C++.

[13] We are assuming for now that the source file in question is not a part of a bigger project.

[14] Depending on compiler and compilation flags used, this may or may not produce warnings as you compile it. Regardless of those, what you see is a formally correct — even though completely useless — program.

[15] The `sizeof` is a bit special, so it is formally not a function but a *pseudo-function*, but that distinction is not important at the moment.

2.2.3. Literals

For example, two expressions `8` and `5+3` both evaluate to the integer number 8. Let's focus on the class of expressions exemplified by the first one of these, namely the `8`.

A single token of code directly representing a value stored in the source of the program is called a ***literal expression***, or *literal* for short. A literal occupies space in the source code, but does not have any other presence anywhere else in the computer memory[16]. It cannot be recalled elsewhere in the code without repeating the same literal representation.

The `8` is an example of a literal. If you want to use number 8 in some other place of your program, you must write `8` in your code again.

2.2.4. Functions, Operations, Operators and Call Expressions

It is hard to talk about ***expressions*** and ***functions*** one at a time because they are so tightly intertwined. These notes are a review and not an introduction of these concepts; I will use this fact as an excuse for using functions before we formally introduce them. To simplify this brief foray into the functions territory, I will use only ***operations*** to illustrate my point.

Let's depart from C++ and talk about mathematical functions for a moment. A ***function*** is a way or method for converting inputs into outputs. Conceptually speaking, functions and operations are the same thing, just written in a different notation. When a function is written in ***prefix notation***, it looks like the familiar $\sin(45°)$. But the arithmetic operations, like $3+5$, are functions too! We are just more used to writing $3+5$ instead of $+(3,5)$ — but both of these expressions mean 8 and can be used interchangeably. The way of writing $3+5$ is called the ***infix notation***. So, one can say that an ***operation*** is simply a function written in the infix notation.

The prefix notation is more general. In mathematics and C++ alike, it can be used with functions having more than two parameters[17]. In C++ context, there is one more aspect that makes prefix notation more general than the infix one: (prefix-denoted) functions can be *defined*, while (infix-denoted) operations cannot, so that we are limited to those operations that are built-in. Those pre-defined C++ operations are called ***operators***.

Jumping ahead, standard operators of C++, such as `+`, `-`, `*`, `=` and even the parentheses `()` — can be *overloaded*.

Consider the expression `5 + 3` that evaluates to `8`. The first expression is an example of a ***function call expression***. Those expressions evaluate to the result of the function when that function is applied to the objects specified by the

[16] There is an exception to that general statement. String literals, being constant arrays of characters, actually occupy specific place in computer memory. We will talk about it in more detail later.

[17] While the expressions like $3+5+7$ in the infix notation seem to allow multiple parameters as well, formally speaking they represent repeated use of the corresponding binary function. So, in reality, $3+5+7=(3+5)+7$, even if it is usually written without parentheses. Thanks to associativity of addition, the omission of parentheses does not create the ambiguity in this case, unlike other situations like $(a/b)/c \neq a/(b/c)$.

constituent expressions "plugged into" the function at hand. In our example, the constituent expressions are the literals `5` and `3`, and the function is the addition operation. We now have an important principle: **function call expressions construct (compound) expressions from other (constituent) expressions**.

2.2.5. PRValues

We can now try to define what an expression means in general, using as a production rule the following definition: **an expression is either a literal, or a function call expression constructed from other expressions**[18]. Limiting our "seed" expressions to literals is too restrictive: there are other types of things we can use as well, but before discussing those more general things, let's take the simple case first.

The expressions we just defined are a subclass of a slightly more general (but still narrow) class of expressions, called ***prvalues***. Prvalues represent objects whose identity is transient and fully subsumed by their ***state***.

The term "prvalue" stands for *pure right-hand value*. There is a long history associated with this terminology. Originally — in the early versions of C — there were two terms: "lvalue" and "rvalue". Back then, life was simple and "lvalue" was a shorthand for "left-hand value", meaning an expression that could appear on the left and on the right side of the assignment operator (like a variable name `x`). Similarly, "rvalue" stood for "right-hand value" and described an expression that could appear only on the right side of an assignment operator, (like a literal `8` or a function call made from literals, such as `5 + 3`). C++ and the later versions of C piled up a lot of extra complexity on top of those simple concepts, shifting the meaning of the terms *lvalue* and *rvalue* quite a bit. Now the term "lvalue" is reinterpreted as "locator value", but the "rvalue" (and its derived narrower variant "prvalue") were not reinterpreted in a similar way. So, perhaps a better way of describing what "prvalue" stands for would be something like: *an expression of the kind that resembles right-hand value from the old days*.

2.2.6. Object Declarations and LValues

In addition to prvalues, expressions may represent objects that are stored in the computer memory. Stored objects have their own identity independent from their current state, and may be recalled and used somewhere else in the program. Expressions describing such objects are called ***lvalues***, meaning *locator* values[19].

The simplest kind of an lvalue expression is an ***identifier***. An identifier is a name of a specific object. Any sequence of letters, digits and underscore symbols starting with a letter or underscore symbol, which is not a reserved word of the

[18] This is an example of a ***recursive*** production rule. As in the case of any recursion, it must have a *base case* that serves as the "seed", or the starting point of the recursive build process. Here, such a seed is the literal expression.

[19] As mentioned in the remark at the end of the previous section, it used to mean *left-hand value*.

language, can serve as an identifier[20]. C++ is case-sensitive, so the identifier `user_input` is not the same as `USER_INPUT`.

Every identifier must be ***declared*** before it can be used. A declaration ***statement*** creates an object in the world described by the code and gives it a unique name — that very identifier — by which that object will be recalled in the subsequent code.

We will not define the meaning of the word ***statement*** here. Instead, we will build the concept of a statement incrementally throughout the course of studying C++.

2.2.7. Type and Type Declarations

A declaration statement in C++ must state the ***type*** of the identifier being introduced. For example, a declaration statement for an integer identifier can look like this:

```
int i;
```

In the above snippet of code, the `int` is the ***type declaration***, and the `i` is the ***identifier***.

Broadly speaking, computer programming languages are classified as ***dynamically typed*** and ***statically typed***. In a dynamically typed language, such as Javascript, *values* have types (e. g. `5` is an integer and `3.14` is a float) but the *identifiers* themselves are typeless:

```
// Javascript:
let x = 5;       // x holds an integer 5
x = 3.14;        // now the same x contains a float
x = "Hello";     // finally the x contains a string
```

On the other hand, in statically typed languages — such as C and C++ — *identifiers* themselves have types.

If one thinks of an identifier as a box with a label, a dynamically typed language has boxes of standard size, large enough to contain any object of that language. On the other hand, a statically typed language has different types of boxes varying according to their size and shape.

The simplest objects in C++ (as well as in other programming languages) are integer and floating point numbers. There are many different built-in types for both of those categories[21]. They all share one fundamental property: **the amount of space occupied by a single number does not depend on that number's value and is determined by that number's *type* alone**. So, for instance, an integer number `5` takes the same space as the integer number `100`. Each type allocates some fixed number of binary digits for every number of that type. The difference between various types of numbers boils down to three characteristics:

1. whether it is an integer or a float;

[20] The identifiers starting with underscore, e.g. `_start`, are often used by the compiler and the libraries for their internal identifiers and should be avoided when *naming* new identifiers in the code. However, one can *use* those built-in compiler-provided identifiers. For instance, most compilers will implicitly declare the identifier `__func__` and store the name of the current enclosing function in it.

[21] described, for instance, at https://en.cppreference.com/w/cpp/language/types.html

2. the specific number of bits each number of that type occupies;

3. whether (for integers) it is a ***signed*** or an ***unsigned*** number.

How many bits are occupied by any number (or, more generally, object) of a particular type may depend on the architecture used[22]. That architecture-specific information is known at compile time and can be determined within code using the `sizeof` ***pseudo-function***. In contrast with a regular functions, `sizeof` can be used with an argument being either a *type*, as in `sizeof( int )`, or an *object* of a particular type, as in `sizeof( 5 )`[23]. The `sizeof` returns the size in *bytes*, and the `sizeof( char )` is always one.

2.2.8. Value Categories

When an object is created by its identifier declaration, that object's state can be explicitly specified. Such specification is called ***initialization***. The code

```
int i = 8;
```

creates an integer number, called `i`, and makes that number equal to `8`. This particular code ***constructs*** an object in a particular state.

Whether a given expression is an lvalue or a prvalue is a characteristic called the ***value category***. In addition to those two, there is another one, called ***xvalue*** which we will discuss later. There are also two more categories (***glvalue*** and ***rvalue***) which are constructed by combining the three mentioned earlier in various ways.

To summarize, a prvalue is a state of an object without an identity, and an lvalue is an object that has both a state and an identity.

2.2.9. Side Effects of Expressions

Besides evaluating to a particular object, an expression may also have a ***side effect***. Side effect of an expression is the full set of changes in the ***states*** of objects directly or indirectly involved in that expression resulting from evaluation of that expression. In the next section we describe the most important and familiar example of an operator with a side effect. But while we are discussing side effects, let's also describe what an ***expression statement with side effect*** is.

An object, or — more formally — **an expression cannot appear in a C++ program by itself**. Rather, it has to be included as a part of a larger *statement*. The simplest way to form a statement from an expression is to put the semicolon "`;`" after the expression. In this case, the object the expression evaluates to is discarded, and the only point of doing it at all is the side effect of the expression at hand. In other words, expression statements are made solely for the sake

[22] Many classic C types, like `int`, are architecture-dependent, but beginning with C99 (with `#include <stdint.h>`) and C++11 (with `#include <cstdint>`), compiler understands the so-called "fixed width" types, like `int32_t`, which are the same size on all platforms where they are available. Note, however, that the `char` type is always 1 byte long.

[23] There is another way in which `sizeof` differs from a usual function: an array does not decay to a pointer when given as an argument to `sizeof`, so that `sizeof( a )` returns the combined size of all elements of the *array* `a`, rather than the size of the *pointer* `a`.

of their side effect. The following is a program with an expression statement which actually does not have any side effect (and is thus completely useless, while correct):

```
int main(){
    5;
}
```

In the above, the `5` is an expression that evaluates to the number 5, and the "`5;`" is the resulting *expression statement*.

2.2.10. The Assignment Operator

The ***assignment operator*** looks like `a = b`. Unlike the mathematical concept described by the same notation, the assignment is not a claim of `a` being equal to `b`. In addition, `a = b` is categorically not the same as `b = a`, as it is crucially important which expression is on the left and which one is on the right side of the "=" sign. The *expression* `a = b`

1. makes the state of the object `a` the same as that of the object `b`, and
2. evaluates to that assigned state.

The first is the ***side effect*** and the second is the ***value*** of the assignment expression. In the code

```
int i = 5;  // declare an integer i, initializing it to 5
i = 10;     // assign value 10 to that i
```

the assignment operator is called entirely for its side effect. Also note that the assignment and initialization look deceptively similar. It is extremely important to realize that initialization *statement* creates a new object with a specified initial state, while an assignment *expression* alters the state of an existing object (and evaluates to that new state).

Jumping ahead to the time when we will build our own classes, we can say that initialization and assignment refer to two distinct aspects of object ***behavior*** — the first being controlled by the ***constructor*** method, and the second — by the ***copy assignment*** operator.

An assignment *expression* evaluates to the expression being assigned, and thus can be used in any function call expression — even including another assignment:

```
int i = 8;
int j = 10;
i = ( j = 7 );
```

It can surely result in unreadable code if used indiscriminately.

An lvalue can be used on the left or on the right side of an assignment, since it has both a location and a value stored at that location[24].

[24] Even though that value may be garbage.

```
int i = 5;
int j = 7;
i = j;     // lvalue i is on the left; lvalue j is on the right
```

Sometimes people pedantically describe the above evolution of `j` as an *lvalue to prvalue conversion*, since strictly speaking the `j` first decays to its prvalue `7` and only then that prvalue is assigned to `i`.

On the other hand, a prvalue cannot appear on the left side of an assignment but can be used on the right:

```
int i;
i = 7;       // okay: a prvalue 7 is on the right
7 = i;       // WRONG: a prvalue 7 cannot be on the left
i + 3 = 7;   // WRONG: a prvalue i + 3 cannot be on the left
```

2.2.11. CV Type Qualifiers

A declaration of an identifier may have a ***constant*** or ***volatile*** type qualifier. These are known collectively as ***cv qualifiers***.

A constant identifier must be explicitly initialized, and once initialized, the object it refers to cannot change its state:

```
int main()
{
    const int i;  // error: uninitialized ‘const i’
    const int i = 5;      // OK
    i = 7;        // error: assignment of read-only variable ‘i’
}
```

It may be worth noting that the value of the constant does not have to be known at *compile time*. In a sense, the constants known to compiler at compile time (such as literals) are even “more constant” than the constants considered above. C++11 provides a way to specify such constants:

```
constexpr int x = 5;
```

The *volatile* qualifier informs the compiler that the object's state can be changed by factors external to the code where it is declared (such as hardware, other threads, or interrupt routines). While unaware of those externalities, with this information the compiler will not optimize access to this object by, say, cashing its value in a register for faster access.

2.2.12. Reference Declaration and Initialization

C++ has a unique concept of a ***reference declaration***. There are two types of such declarations: ***lvalue reference*** and ***rvalue reference***. An lvalue reference is indicated by the ampersand after the type declaration, and an rvalue reference — by a double ampersand. For example, an integer lvalue and rvalue reference declarations can start like this:

```
int & r ...     // lvalue reference declaration
int && rr ...   // rvalue reference declaration
```

The magic of references comes into play only when an identifier of a reference type is declared and initialized. A reference initialization creates an alias of the original identifier, so that the original and the reference equally refer to the same object.

Let's consider an example when an lvalue reference `r` is initialized with an lvalue `i`:

```
int i = 5; // not a reference, just a normal integer
int & r = i; // r is now a reference to i
```

From that point on, the identifier `r` behaves like a stand-in for `i`:

```
int r = 7; // now i is also 7
```

Let's think of identifiers as boxes and the values as the content of those boxes. A regular initialization of an identifier with an expression — like the initialization of `i` with `5` — places the value of that expression into that identifier's box. On the other hand, a reference initialization of an identifier with an *lvalue* expression — like the initialization of `r` with `i` — creates a teleport tunnel connecting the opening of the initializer box `i` with the bottom of the initialized box `r`. Thus, both boxes — the initializer and the initialized — provide access to the same exact content when they are opened.[25] This metaphor should make it clear that taking reference repeatedly, as in

```
int i = 5;
int & r = i;
int & rr = r;
rr = 10; // all three identifiers now refer to the same 10
```

does not produce a "reference to reference" — `rr` is just another reference to the same old `i` and to `r`.

One possible point of confusion is the value category of references. Both lvalue references and rvalue references are *lvalue expressions* themselves. They are classified as lvalue and rvalue not based on what they *are*, but based on what they *refer to*. Another possibility for confusion stems from an unfortunate terminology. A *reference identifier*, once declared and initialized, is no different from a non-reference one. The word "reference" distinguishes not the identifier itself, but its declaration. The true utility of reference declaration will manifest itself later when we argument-to-parameter binding in functions and the ***move semantics***.

For the sake of completeness, let's consider all possible combinations of reference declarations to see what is legal and what is not. References must be initialized at the time of their declaration:

[25] In principle, one can use an even simpler metaphor. Instead of "teleport tunnels", think of objects as boxes and their identifiers as labels slapped onto those boxes. Then a reference declaration of an identifier is like putting the label with that identifier onto an existing box. Even though it is more complicated, I find the tunnel metaphor more effective in illustrating parameter-to-argument binding.

```
int main()
{
    int &           nonconst_l_ref;     //error:
    // 'nonconst_l_ref' declared as reference
    // but not initialized

    const int &     const_l_ref;        //error:
    // 'const_l_ref' declared as reference
    // but not initialized

    int &&          nonconst_r_ref;     //error:
    // 'nonconst_r_ref' declared as reference
    // but not initialized

    const int &&    const_r_ref;        //error:
    // 'const_r_ref' declared as reference
    // but not initialized

}
```

An rvalue reference and a constant lvalue reference can be initialized with an rvalue, but a non-constant lvalue reference cannot:

```
int main()
{
    const int &&    const_r_ref     = 10;   // OK
    int &&          nonconst_r_ref  = 10;   // OK
    const int &     const_l_ref     = 10;   // OK
    int &           nonconst_l_ref  = 10;   // error:
    // cannot bind non-const lvalue reference of type 'int&'
    // to an rvalue of type 'int'
}
```

When initializing references with a constant lvalue, only a constant lvalue reference can get it as the initializing value; all other combinations will result in errors:

```
int main()
{
    const int const_lvalue = 20;

    const int &     const_l_ref     = const_lvalue; // OK
```

```
    int &           nonconst_l_ref  = const_lvalue; // error:
    // binding reference of type ‘int&’ to ‘const int’
    // discards qualifiers

    int &&          nonconst_r_ref  = const_lvalue; // error:
    // cannot bind rvalue reference of type ‘int&&’
    // to lvalue of type ‘const int’

    const int &&    const_r_ref     = const_lvalue; // error:
    // cannot bind rvalue reference of type ‘const int&&’
    // to lvalue of type ‘const int’
}
```

Finally, lvalue references can be initialized with a non-constant lvalue, while rvalue references cannot:

```
int main()
{
    int nonconst_lvalue = 30;

    int &           nonconst_l_ref = nonconst_lvalue; // OK
    const int &     const_l_ref = nonconst_lvalue;    // OK

    int &&          nonconst_r_ref = nonconst_lvalue; // error:
    // cannot bind rvalue reference of type ‘int&&’
    // to lvalue of type ‘int’

    const int &&    const_r_ref = nonconst_lvalue;    // error:
    // cannot bind rvalue reference of type ‘const int&&’
    // to lvalue of type ‘int’
}
```

2.3. Functions

2.3.1. Function Definition

Functions are the most basic way of structuring code into smaller parts. Some functions and operators[26] are built-in (like the arithmetic operations on basic numerical types). New functions can be defined. Let's recall some related terminology. A complete ***function definition*** may look like this:

```
int square( int i )
{
```

[26] Operators may be seen as functions written in a particular format, called the ***infix notation***.

```
    return i * i;
}
```

where the first type declaration — in this case the `int`:

```
int square( int i )
{
    return i * i;
}
```

is the ***return type***; what follows it — in this case the `square`:

```
int square( int i )
{
    return i * i;
}
```

is the ***identifier***, or — more colloquially — the *name* of the function; the parenthesized ordered list of ***parameter*** types — in this example the `( int )`:

```
int square( int i )
{
    return i * i;
}
```

is the ***signature***; the parenthesized ordered list of identifiers — in this case the `i`:

```
int square( int i )
{
    return i * i;
}
```

is the ***parameter*** identifier (list): the part between the curly braces:

```
int square( int i )
{
    return i * i;
}
```

is the ***body***; finally, the `i * i`:

```
int square( int i )
{
    return i * i;
}
```

is the ***return expression***.

The first line, with or without the parameter identifier(s), is the ***declaration*** of the function, also called a *forward declaration* (of the function) or a *function prototype*. It can appear, as a separate *statement*, by itself — without the body of the function; in this case it is terminated by a semicolon:

```
int square ( int i ); // with parameter identifier
int square ( int );   // without parameter identifier
```

2.3.2. Function Call Expressions

While a ***function definition*** is a recipe describing how to do a task (i. e. "this is how to square an integer number `i`"), a ***function call expression*** is an order to go ahead and do that task (i. e. "now please go ahead and square this particular number exactly as I described that process earlier").

Syntactically, a function ***call*** is the identifier of the function followed by the parenthesized list of ***argument expressions***. For example, calling the just defined function `square` with the argument expression `3 + 5` will look like this:

```
square( 3 + 5 )
```

Like any other expression, a function call may

- have a value;
- have a side effect;
- appear in a larger expression.

The value of the function call expression is the value of the return expression of the function:

```
int square( int i )
{
    return i * i;
}
int main()
{
    int x = 5;
    int y = square( x ); // now y is 25
}
```

A function must be *declared* before it may be called. However, functions, once declared in a file, can be used in that file right away:

```
int square( int ); // function declared but undefined
int main()
{
    int x = 5;
    int y = square( x ); // function can be used here
}
int square( int i ) // and defined elsewhere
{
    return i * i;
}
```

Declared and used in a particular source file, a function does not have to be *defined* in that file. All the compilation steps up to but not including linking[27], which process one individual source file at a time, will work just fine as long as the functions being used in them are declared (but not necessarily defined) before use. Only the linking step requires the definitions of all functions used.

All identifiers declared in the body of the function are not visible outside of the function. They are called ***local variables***.

Function parameters are *positional*.

[27] Namely, the preprocessing, compilation and assembly.

```
int add_first_and_squared_second( int n, int m )
{
    return n + m * m;
}
int main()
{
    int a = add_first_and_squared_second( 1, 2 ); // a is 5
    int b = add_first_and_squared_second( 2, 1 ); // b is 3
}
```

2.3.3. `void` Type in Functions

A function may have `void` as their return type. Void functions do not return any value; they are called entirely for their side effect. In void functions, the `return` statements are optional (and are implicitly inserted by the compiler at the end of the body):

```
void greet( std::string given_name )
{
    std::cout << "Hi " << given_name << "!\n";
}
```

If the `return` statements are used at all, they must have empty return expressions:

```
void fussy_greet( std::string given_name )
{
    // I don't talk to mice
    if( given_name == "Mickey Mouse" ) return;
    // the above "return" has an empty return expression

    // implicitly the rest of the function is the "else" part:
    std::cout << "Hi " << given_name << "!\n";
}
```

A function can also have `void` signature, indicating that it does not take any inputs. However, the form `func( void )` — even though legal — is usually replaced by an equivalent shorter form `func()`, both in declarations and calls. Here is an example of a complete program with a void function with void signature:

```
#include <iostream> // for cerr, cout
#include <cstdlib> // for exit
void die() // void function with void signature
{
    std::cerr  << "Usage: ./greet <name>\n";
    exit( 1 );
```

```
}
void greet( std::string given_name ) // void function
{
    std::cout << "Hi " << given_name << "!\n";
}
int main( int argc, char **argv )
{
    if( argc != 2 ) die();
    greet( argv[ 1 ] );
    return 0;
}
```

Note the use of the `std::cerr` in the `die` function. Unlike the usual output with `std::cout` in the `greet` function, the `std::cerr` writes the error message about program's intended usage into the ***standard error*** instead of the ***standard output*** stream. When a program is called on the command prompt, both the standard output and standard error are printed on the terminal by default, but the user can separate them by turning one of them off or redirecting either or both to a separate file.

Assuming that the executable of this program is called `greet`, interacting with it will look like this:

```
user@computer:~$ ./greet
Usage: ./greet <name>
user@computer:~$ ./greet Mickey Mouse
Usage: ./greet <name>
user@computer:~$ ./greet "Mickey Mouse"
Hi Mickey Mouse!
user@computer:~$ ./greet Mickey
Hi Mickey!
```

2.3.4. Argument to Parameter Binding

When a function with parameters is called, implicit declaration and initialization of parameters take place. The arguments passed to the function are the initializers for the parameters in question. This is called ***argument-to-parameter binding***. Let's go back to the code we considered earlier:

```
1  int square( int i )
2  {
3      return i * i;
4  }
5  int main()
6  {
7      int x = 5;
8      int y = square( x );
```

```
9       }
```

When line 8 is executed, implicit initialization `int i = x;` happens before the `square()` function is called into action. Thus, the value of `i` that `square` operates on has nothing to do with the `x` in the `main()` function. The `square` operates on a *copy* of `x`, not on `x` itself. So, for example, if we add line 3 to the above code:

```
1       int square( int i )
2       {
3           i = i + 1;
4           return i * i;
5       }
6       int main()
7       {
8           int x = 5;
9           int y = square( x ); // x is 5, y is 36
10      }
```

then at the end of the `main()` function, the value of `x` will still be `5`. This behavior of the parameter `i` is called ***call-by-value***.

With this in mind, we can better appreciate one of the main uses of lvalue references. Take the above code and add the reference declaration symbol `&` to the `square()` function's signature:

```
1       int square( int & i )
2       {
3           i = i + 1;
4           return i * i;
5       }
6       int main()
7       {
8           int x = 5;
9           int y = square( x ); // x is 6, y is 36
10      }
```

When line 9 is executed, the implicit initialization happens again, now taking the form `int & i = x;` and saying that the `i` inside of the `square` refers to the same entity as the `x` in the `main()`. In this case, any change to `i` done in the `square` affects `x` in the `main()`. This behavior of the parameter `i` is called ***call-by-reference***.

2.3.5. Implicit Argument Coercion

In assignment and initialization, the value of the initializing expression is coerced into the type of the value being assigned or initialized.

Type coercion also happens in the implicit initializations taking place during argument-to-parameter binding and return during a function call. The general

rules of type coercion apply in this case as well: when expressions are formed using functions and operators, the arguments of those functions and operations are coerced into types accepted by those operations. Then — for user-constructed functions and operators — the return value is coerced into the return type of the function or operator being constructed.

For example, since many built-in arithmetic operators take `int` or `double` numbers, all arguments having various shorter integer types (like `char`, `short` and `unsigned short`) are implicitly promoted to `int` when used those operators:

```
#include <iostream>
int main()
{
    std::cout << 'A' + 0; // promotion char{'A'}->int{65}
}
```

will print `65` instead of `A`.

2.3.6. Function Overloading

The same identifier can be used in function definitions with different signatures. Effectively, those definitions create completely distinct and unrelated functions that can, in principle, do completely different things. Thus, **function's identity is tied to the combination of its identifier and its signature** and not to the identifier alone. Since the signature characterizes function's *inputs*, functions cannot be distinguished by their return type, i. e. the *outputs*. The use of the same identifier for different functions is called ***function overloading***. (In the strict sense, overloading happens to the function's *identifier* rather than the *function* itself.)

When several functions with the same identifier exist, the version used for specific arguments is selected based on the number and type of those arguments:

```
#include <iostream> // for cout
void greet()
{
    std::cout << "Hello!\n";
}
void greet( int x )
{
    std::cout << "I have an integer number " << x << "!\n";
}
int main()
{
    greet(); // the first function is called
    greet( 5 ); // the second function is called
}
```

When several choices are present, they are selected in the following order of preference: exact match of the argument types, type promotion, type demotion. The selection process fails when encountering ambiguity. Adding an overloaded function to an existing list of overloaded functions may introduce such an ambiguity and thus result in a compiler error. For example, this version of `main()` will work just fine with the above two `greet` functions:

```
int main()
{
    greet( 5.7 ); // implicit type demotion double{5.7}->int{5}
}
```

but adding another potential candidate for type demotion — in this case `greet( short )` — will result in an error:

```
#include <iostream> // for cout
void greet()
{
    std::cout << "Hello!\n";
}
void greet( int x )
{
    std::cout << "I have an integer number " << x << "!\n";
}
void greet( short x )
{
    std::cout << "I have a short int " << x << "!\n";
}
int main()
{
    greet( 5.7 ); // error:
    // call of overloaded 'greet(double)' is ambiguous
}
```

2.3.7. Function Templates

COME AGAIN LATER

2.4. Expressions: Summary

2.4.1. What is an Expression?

This is an approximate, incomplete and imperfect attempt to summarize how an expression must look like syntactically. The next section will address what expressions mean semantically.

An expression is either a literal (e. g. `5`), or an identifier (e. g. `x`), or a function call[28] applied to other — shorter — expressions (e. g. `f(x)`),

2.4.2. What You Must Know about Expressions

Consider this code snippet:

```
1  int main()
2  {
3      int x = 5;
4      int y = 7;
5      x = ( y++ );
6      // ...
```

I will use the expression on line 5 and the box-with-stuff metaphor to illustrate the points made below. **Every time you form an expression in C++ you must know:**

- the *identity* of the object it evaluates to, if any — i. e. the box the expression selects (in the example, it is `x`) or lack thereof[29];
- the *state* of the object it evaluates to, if any — i. e. the stuff, in the box or by itself, the expression constructs (in the example, it is `7`) or lack thereof[30];
- the *type* of the object the expression evaluates to — i. e. the shape and form of the stuff it describes (in the example, it is `int`)[31];
- the *side effects* it produces, if any — i. e. the change in the content of other boxes involved in its evaluation (in the example, it is the change of the content of `y` from `7` to `8`)[32];
- the *memory location* of the object, if any — i. e. the place of the box in the memory storage (in the example, it `x` is a local variable of the `main()` function, thus residing on the stack in the stack frame of the `main()`) or lack thereof;

[28] The word "function" is understood in the general sense which includes operators as well as proper functions.

[29] What is the term describing the expressions that have no identity?

[30] What is the term describing the expressions that have no state?

[31] Even a prvalue has type, so the type characteristic is an attribute of both the box and the stuff, even if that stuff is not in a box. Since the `void` expressions which neither construct stuff nor select any box still have the `void` type, type is a mandatory characteristic of an expression, so that every expression must have it.

[32] Give an example of expression without side effects.

- the *lifetime* of the object — i. e. the span of time when the object the expression evaluates to will exist and retain its identity (in the example, it is the time from `int x = ...` declaration until `main()` function's return);
- the *visibility scope* of the expression itself — i. e. the context in which this expression has meaning (in the example, it is the section of the body of `main()` from line 5 until the end);
- the *cv type* of the object the expression evaluates to — i. e. whether the box is closed and sealed or open and whether a force external to our code can change its content (in the example, `x` is neither constant nor volatile).

HOMEWORK: Answer all questions and address any requests made in the footnotes to the above list. Which specific combination of the above characteristics determines the *value category* of the expression? Try to come up with examples of expressions illustrating each possible combination of these characteristics.

2.4.3. In-Class Quiz

What will be the output of the following two programs?

Program 1.

```
#include <iostream>
int main()
{
    int x = 5;
    ++(x++);
    std::cout << "x = " << x << "\n";
}
```

Program 2.

```
#include <iostream>
int main()
{
    int x = 5;
    (++x)++;
    std::cout << "x = " << x << "\n";
}
```

References allow us to re-implement the two increment operators[33] from scratch:

[33] There will be a better — a more complete and faithful — way to do it, which we will explore later.

```
// ++x
int & pre_increment( int & x )
{
    x = x + 1;
    return x;
}

// x++
int post_increment( int & x )
{
    int tmp = x;
    x = x + 1;
    return tmp;
}
```

so that the original quiz problem becomes:

Program 1.

```
#include <iostream>
// include the two function definitions from above
int main()
{
    int x = 5;
    pre_increment( post_increment( x ) );
    std::cout << "x = " << x << "\n";
}
```

Program 2.

```
#include <iostream>
// include the two function definitions from above
int main()
{
    int x = 5;
    post_increment( pre_increment( x ) );
    std::cout << "x = " << x << "\n";
}
```

This explication immediately shows that the first program will fail to compile due to incompatibility of the input-output types. Indeed, the evaluation of the inner `post_increment( x )` results in an `int` *prvalue* output `5` (coming from the `tmp` inside of `post_increment` function), which then becomes the initializer value of the `int &` parameter of `pre_increment` function. But as we know from section 2.2.12 (page 17), the implicit initialization `int & x = 5` happening in

`pre_increment(...)` function call will fail to compile since an lreference initialization requires an lvalue as the initializer.

If we were to try fixing it with a reference *return* type:

```
int & post_increment( int & x )
{
    int tmp = x;
    x = x + 1;
    return tmp;
}
```

we would immediately run into an even bigger problem. A function returning a reference to its own local variable creates a teleport tunnel to nowhere since the stack frame where that local variable (in this example `tmp`) resides is discarded immediately after the return. Accessing that return value results in a segmentation fault.

The second program will compile and run with the output `x = 7`. Here is the full program:

```
#include <iostream>
// ++x
int & pre_increment( int & x )
{
    x = x + 1;
    return x;
}
// x++
//int & post_increment( int & x ) // Segmentation fault
int post_increment( int & x )
{
    int tmp = x;
    x = x + 1;
    return tmp;
}
int main()
{
    int x = 5;
    //pre_increment( post_increment( x ) ); // error:
    // cannot bind non-const lvalue reference of type 'int&'
    // to an rvalue of type 'int'
    post_increment( pre_increment( x ) ); // produces "x = 7"
    std::cout << "x = " << x << "\n";
}
```

2.5. Control Flow Statements

2.5.1. The `if` Statement

The `if` statement has the form

```
if( <condition> ) <statement>
```

and executes the `<statement>` if and only if the `<condition>` is true. It can optionally include an alternative branch, as in

```
if( <condition> ) <statement 1>
else <statement 2>
```

and in that case the `<statement 2>` executes if and only if the `<condition>` is false. A nested `if...else` construction may lead to the *dangling `else`* ambiguity, since

```
if( <condition 1> )
    if( <condition 2> ) <statement 1>
    else <statement 2>
```

while meaning the same as

```
if( <condition 1> )
    if( <condition 2> ) <body 1>
else <body 2>
```

seems to suggest a different execution flow. Whenever this ambiguity arises, use curly braces to resolve it, making it clear to both a human reader and computer whether you mean this:

```
if( <condition 1> ) {
    if( <condition 2> ) <statement 1>
    else <statement 2>
}
```

or that:

```
if( <condition 1> ) {
    if( <condition 2> ) <statement 1>
}
else <statement 2>
```

Some style guides suggest consistent use of braces in all `if...else` statements regardless of the number of sub-statements in any sub-branch, disavowing the code like this:

```
std::string letter grade( int score )
{
    if( score >= 93 ) return "A";
    if( score >= 90 ) return "A-";
    if( score >= 87 ) return "B+";
    if( score >= 83 ) return "B";
    if( score >= 80 ) return "B-";
    if( score >= 77 ) return "C+";
    if( score >= 73 ) return "C";
    if( score >= 70 ) return "C-";
    if( score >= 67 ) return "D+";
    if( score >= 60 ) return "D";
    return "F";
}
```

I see it as a bit too extreme, but decide for yourself.

2.5.2. The `switch` Statement and `break` Command

A `switch` statement has the form

```
switch( <expression> ){
case <constant 1>:
    <statement 1 1>
    <statement 1 2>
    ...
case <constant 2>:
    <statement 2 1>
    <statement 2 2>
    ...
}
```

In the above, the `<expression>` must evaluate to some version of *integer* type, while each of the `<constant..>`'s must be a *compile time* constant of the same integer type. Each `<constant i>` is the place where the control flow jumps to when the corresponding the `<expression>` equals the `<constant i>`. From that point on, all the subsequent statements of the `switch` statement are executed. For example, when `daynum` happens to be `5`, the code

```
switch( daynum ){
    case 1:
        std::cout << "Monday\n";
    case 2:
        std::cout << "Tuesday\n";
```

```
        case 3:
            std::cout << "Wednesday\n";
        case 4:
            std::cout << "Thursday\n";
        case 5:
            std::cout << "Friday\n";
        case 6:
            std::cout << "Saturday\n";
        case 7:
            std::cout << "Sunday\n";
    }
```

will result in the console output

```
Friday
Saturday
Sunday
```

If the intended effect is to skip all the subsequent possibilities, a `break` command can be used for exiting the `switch` statement: with the same value `5` for the variable `daynum`, the code

```
    switch( daynum ){
        case 1:
            std::cout << "Monday\n";
            break;
        case 2:
            std::cout << "Tuesday\n";
            break;
        case 3:
            std::cout << "Wednesday\n";
            break;
        case 4:
            std::cout << "Thursday\n";
            break;
        case 5:
            std::cout << "Friday\n";
            break;
        case 6:
            std::cout << "Saturday\n";
            break;
        case 7:
            std::cout << "Sunday\n";
    }
```

will result in the console output

```
Friday
```

If the <constant...>'s don't cover all possible values of the integer type in question, the compiler will[34] give a warning[35]. All possible values can be handled by a `switch` statement with a `default` clause:

```
switch( <value> ){
case <label 1>:
    <statement 1 1>
    <statement 1 2>
    ...
case <label 2>:
    <statement 2 1>
    <statement 2 2>
    ...
default:
    <statement d 1>
    <statement d 2>
    ...
}
```

For example:

```
switch( daynum ){
    case 1:
        std::cout << "Monday\n";
    case 2:
        std::cout << "Tuesday\n";
    case 3:
        std::cout << "Wednesday\n";
    case 4:
        std::cout << "Thursday\n";
    case 5:
        std::cout << "Friday\n";
    case 6:
        std::cout << "Saturday\n";
    case 7:
        std::cout << "Sunday\n";
    default:
```

[34] with compilation key `-Wall`

[35] An integer type can have finitely many possible values when it is an `enum` type. In that case it is possible to cover all cases with appropriate <constant...>'s.

```
            std::cout << "Not sure what you mean\n";
    }
```

At least one sub-statement must be included in the `switch`, but that statement can be just an empty ";". The cases can be combined as in:

```
    switch( daynum ){
        case 1:
            std::cout << "Monday\n";
            break;
        case 2:
            std::cout << "Tuesday\n";
            break;
        case 3:
            std::cout << "Wednesday\n";
            break;
        case 4:
            std::cout << "Thursday\n";
            break;
        case 5:
            std::cout << "Friday\n";
            break;
        case 6: // combining this case with the next
        case 7:
            std::cout << "Weekend\n";
        default:
            ; // empty statement to avoid the compiler warning
    }
```

2.5.3. Loops

All loops can include the `break` command in their <body>. The `break` effect in a loop is the same as in a `switch` statement: it takes the control flow to the place immediately after the loop. In addition, a single iteration of a loop can be skipped with a `continue` command. The `break` and `continue` commands can appear anywhere in the <body> of the loop.

`while` loop

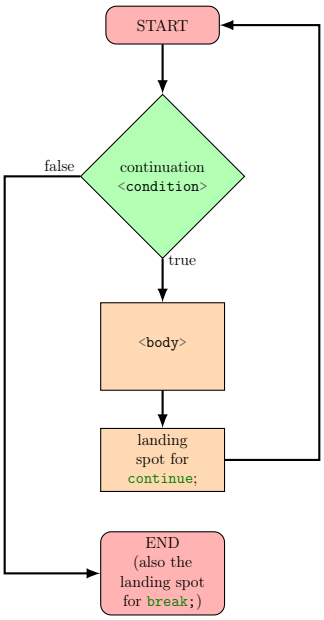

Figure 5. `while( <condition> ){ <body> }` flow chart.

`do...while` loop

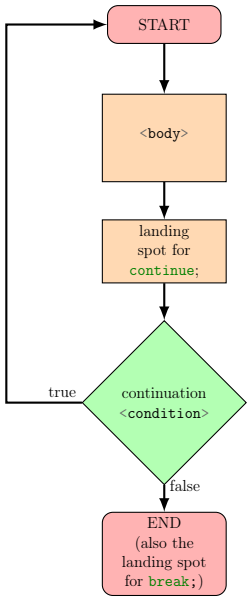

Figure 6. `do{ <body> } while( <condition> );` flow chart.

`for` loop

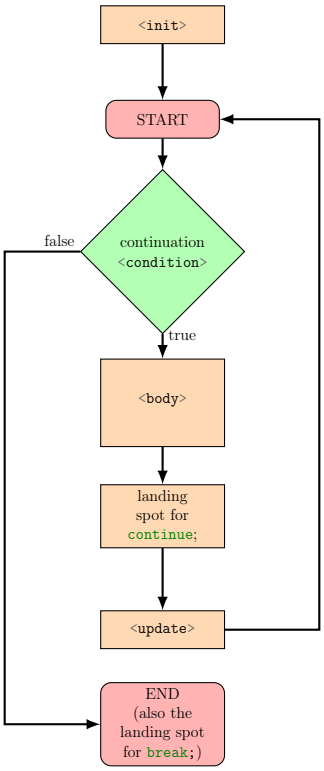

Figure 7. `for( <init>; <condition>; <update> ){ <body> }` flow chart.

2.6. Pointers

2.6.1. Address and Dereference Operators

```
#include <iostream> // for cerr, cout
#include <cstdint> // for uintptr_t
void allocate_and_report_int_on_stack()
{
    // uniform (a.k.a. 'brace') initialization:
    int my_int = int{4} + int{7};
    int * my_int_ptr = & my_int;

    // my_int_ptr == & my_int
    // is equivalent to
    // * my_int_ptr == my_int

    std::cout
        << "*("
        << reinterpret_cast<uintptr_t>( my_int_ptr )
        << ") = " << my_int << " on the STACK"
        << " of the function " << __func__
        << "\nThe pointer size is " << sizeof( my_int_ptr )
        << ", the integer size is " << sizeof( my_int )
        << ".\n\n";
}
void allocate_and_report_int_on_heap()
{
    int * ptr = new int{5}; // allocate on the heap
    std::cout
        << "*("
        << reinterpret_cast<uintptr_t>( ptr )
        << ") = " << *ptr << " on the HEAP."
        << "\nThe pointer size is " << sizeof( ptr )
        << ", the integer size is " << sizeof( *ptr )
        << ".\n\n";
    delete ptr;     // free memory on the heap
    ptr = nullptr;  // just in case ptr is used below
}
void alert()
{
    std::cout
```

```
            << "\nPay attention to"
            << " which of the two pointers is bigger"
            << " and recall the virtual memory structure:\n\n";
}
int main()
{
    alert();
    allocate_and_report_int_on_heap();
    allocate_and_report_int_on_stack();
}
```

2.6.2. Arrays

Multi-Dimensional Arrays

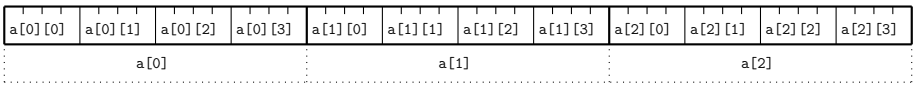

Figure 8. Array `int a[3][4]` in memory.

2.6.3. C-Strings

A ***C-string*** is an array of the type `char` with at least one element being the null character `'\0'`. Semantically, the content of the string is the sequence of characters up to — but not including — the very first null character. This is expressed by saying that C-strings are *null-terminated*.

For example, the string "Hello" will look like this in memory:

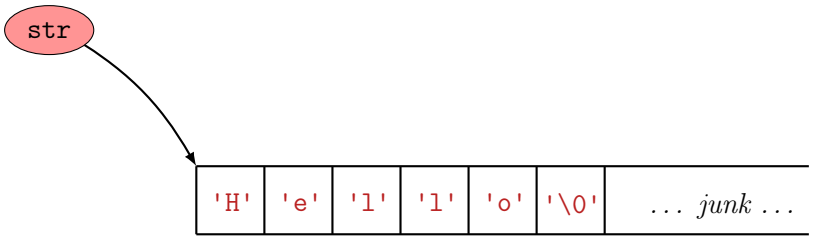

Figure 9. String `"Hello"` in memory, as a `char` array

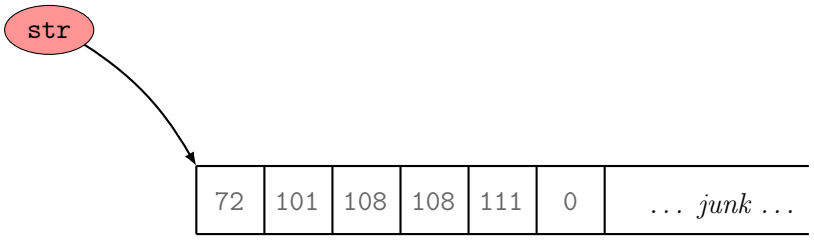

Figure 10. String `"Hello"` in memory, as an ASCII codes `int` array.

String literals are C-strings located in the read-only portion of the static memory. Being addressable, string literals are an exception from the literals of other types:

```
&"Hello"; // OK
&5; // WRONG
```

The address of the string literal (in static memory) is itself a constant pointer to `char` and can be stored to a variable like this:

```
const char * str = "Hello";
```

In the above, the string literal *array* effectively decays to the pointer on the left hand side.

A string literal can also be used for array initialization (which is one of the few cases[36] when an array does not decay into a pointer):

```
const char str[] = "Hello";
```

In the array initialization, the original string literal is *copied* into its new location, and the initialized array is automatically allocated with the length equal to the string length in characters plus one extra space for the null character. In other words, the memory representation depicted above for the example of "Hello" will have no trailing junk, and the following code:

```
#include <iostream>
int main()
{
    const char str[] = "Hello";
    std::cout << sizeof(str)/sizeof(str[0]) << "\n";
}
```

will output `6` on the console. Note that

```
const char * str = "Hello";
```

also makes sense, but means something else. Can you guess the output of the following code?

```
#include <iostream>
int main()
{
    const char * str  = "Hello";
    std::cout << sizeof(str)/sizeof(str[0]) << "\n";
}
```

If we go over the array size:

[36] Besides the `sizeof` and `&`.

```
#include <iostream>
int main()
{
    const char str[] = "Hello";
    for( int i = 0; i < 10; i++ )
        std::cout << "'" << str[i] << "', ";

}
```

we will see something like

```
'H', 'e', 'l', 'l', 'o', '', '...', '...', '...', '...',
```

while

```
#include <iostream>

int main()
{
    const char str[] = "Hello";
    for( int i = 0; i < 10; i++ )
        std::cout << static_cast<int>( str[i] ) << ", ";

}
```

will produce

```
72, 101, 108, 108, 111, 0, ..., ..., ..., ...,
```

In the above console outputs, the ellipsis "..." indicates the particular junk that happens to be stored on your machine after the array in question.

2.6.4. The `main()` Function Revisited

As mentioned earlier, every standalone C and C++ program must contain a *definition* of a function called `main()`. When the program's executable is called, the system (in our case, the console) will *call* the `main()`.

One can think that the `main()` function *declaration* is implicitly built-in. Furthermore, there are two signatures possible, so, in a sense, `main()` is overloaded. The declaration of the first variant is

```
int main();
```

and the second one looks like this:

```
int main( int, char ** );
```

Both versions of the `main()` function have an `int` return type, describing the return value that is (typically) given back to the system[37] and not to some other part of our code. For this reason, the return value of `main()` is often called the ***exit code*** of the program. If the return value of the `main()` function is not explicitly provided in the code, the compiler assumes that there is a `return 0;` statement at the end of `main()` body. So, for example,

```
int main()
{
    std::cout << "Hello world!\n";
}
```

is interpreted by the compiler as

```
int main()
{
    std::cout << "Hello world!\n";
    return 0;
}
```

By tradition, `0` is the "success" exit code, signaling to the outside world that the program executed as intended and encountered no errors.

The variant of the `main()` function that take input parameters will be considered in the next section. Since (in the usual situation) the function `main()` is called by forces external to our code, the parameters themselves must come from outside as well.

2.6.5. Command Line Arguments

An executable run on a console using command line may be given additional ***command line arguments***. Those arguments are strings of characters separated by (one or more) space symbols. If a single command line argument needs to include a space character within, that whole argument must be in (single or double) quotes. For example, an executable named `main` can be called with command line arguments "`Hello!`", "`The Answer is`", and "`42`" like this:

```
user@computer:~$ ./main Hello! "The Answer is"       42
```

If an argument string needs to include a quote of the same type that was used for enclosing that string, it must be escaped using the backslash symbol:

```
user@computer:~$ ./main "I contain the double quote \" symbol."
```

When the system loads the executable and calls its `main()` function, it passes two arguments to it. The first is an integer specifying the total count of the command line arguments. (Traditionally it is denoted `argc`, although you can use any other identifier.) The second argument (traditionally denoted `argv`) is a bit

[37] To be more precise, to the system *process* that called the executable of our program. In most cases in this class it will be the console process.

more complicated. Roughly speaking, it is an array of C-strings, with each string representing a single command line argument. But because of the array-to-pointer decay in function calls, that array of C-strings turns into a pointer to pointer to `char`. In addition, that array of pointers is itself terminated by `NULL` pointer, and for that reason, the `argc` is one less than the length of `argv`. The name of the executable itself is the `0`-th element of the arguments array. To use these two arguments in C++ code, the `main()` function must include them in its signature:

```
#include <iostream>
int main( int argc, char **argv )
{
    std::cout
        << "This program was called with "
        << argc << " arguments:\n";
    for( int i = 0; i < argc; i++ )
        std::cout
            << "\targument " << i
            << " = " << argv[ i ] << "\n";
    return 0;
}
```

If the executable of the above program is named `main` and is called using the command line

```
user@computer:~$ ./main Hello! "The Answer is"       42
```

the call will result in the following `argv` array:

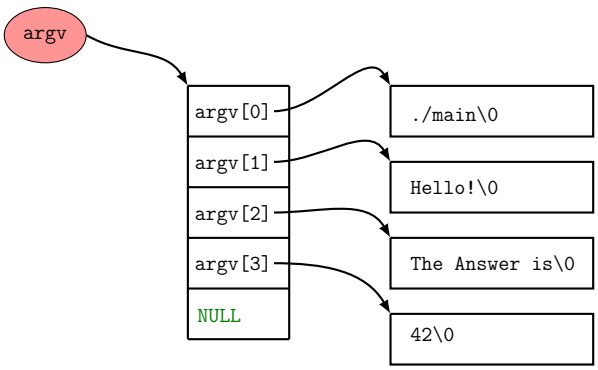

Figure 11. The structure of command line arguments for `./main Hello! "The Answer is" 42`

and, consequently, will produce the following output:

```
user@computer:~$ ./main Hello! "The Answer is"       42
This program was called with 4 arguments:
        argument 0 = ./main
        argument 1 = Hello!
```

```
argument 2 = The Answer is
argument 3 = 42
```

Bibliography

[1] John von Neumann. *First Draft of a Report on the EDVAC*. Technical Report. Contract No. W-670-ORD-4926. Philadelphia, PA: Moore School of Electrical Engineering, University of Pennsylvania, June 1945. URL: https://library.si.edu/digital-library/book/firstdraftofrepo00vonn.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-110362-8.

[3] Bjarne Stroustrup. *The C++ Programming Language*. 2nd Edition. Addison-Wesley, Reading, Massachusetts, 1991.

[4] Bjarne Stroustrup. *The design and evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

Index of Terms

Index of People