C++

Brief Notes on the Vast Subject

BCS230: Foundations of Computer Programming II

Farmingdale — Spring 2026

Alexander Kasiukov

March 19, 2026

Contents

Chapter 1

Larger Context

1.1. 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***.

1.2. 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. ...

1.2.1. 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.

1.2.2. 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 ⊞ + 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.

1.2.3. 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...

1.2.4. 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...

1.3. The Roots of C++

1.3.1. 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.

1.3.2. 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

[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.

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

1.3.3. 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 useful

Figure 4. Bjarne Stroustrup.

far outside of its original domain of operating systems development. The use of C in the far broader realm 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.

Keeping the connection of C++ with the underlying basics of the von Neumann architecture is a trying task, as every new iteration of the C++ standard is pushing the language farther away from the physical level, almost succeeding by now in making C++ a ***high-level*** language. Yet at the same time the ability of C++ to open up and reconfigure its own intestines retains — for now — the possibility of reconnecting it back to that almost-lost basic C level. Even though it takes more and more effort as the C++ Standards Committee is producing new specifications,

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

C++ can be still viewed as an *arbitrary* level language, giving it a unique place in the world of programming languages.

Chapter 2

Expressions

2.1. Basics

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.

2.1.1. 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.

2.1.2. 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

[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.

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***.

2.1.3. 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.

2.1.4. 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

[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.

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 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**.

2.1.5. 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*.

[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 $(a/b)/c \neq a/(b/c)$.

[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.

2.1.6. 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 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++.

2.1.7. 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.

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

[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.

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;
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.

2.1.8. 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.

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

[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`.

2.1.9. 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 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*.

2.1.10. 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. has the ***side effect*** of making the state of object `a` the same as that of `b`, and
2. ***evaluates*** to object `a` in that new state.

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 object in the 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].

```
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
```

2.1.11. 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:

[24] Even though that value may be garbage.

```
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.

2.1.12. 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
```

[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.

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:

```
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'
```

```
}
```

2.2. Functions

2.2.1. 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 )
{
    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***.

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

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
```

2.2.2. 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 that the definitions of all functions used be available.

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*.

```
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
}
```

2.2.3. `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
```

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

```
    // 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!
```

2.2.4. 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***.

2.2.5. 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
}
```

2.2.6. Coercion

In assignment and initialization, the value of the initializer may sometimes be automatically converted by the compiler to the type of the lvalue expression being initialized. This automatic transformation is called ***coercion***. There are several types of coercion, ordered below according to their compiler-perceived distance from the original:

0. exact match
1. ***trivial conversion***

   Examples of those are

   - lvalue-to-rvalue conversion, e. g. `x = y`,
   - array-to-pointer decay, e. g. `int * arr_ptr = array`,
   - function-to-pointer decay, e. g. `void (*func_ptr)() = &func`,
   - addition of cv-modifier to a value, e. g. `const int i = 5`.

From the point of machine realization, these transformations are not changing anything in the memory bits of the original value. For that reason they are considered an nearly equivalent to an exact match.

2. ***promotion***

   Promotion is an in-kind[28] expansion of a type occupying fewer bits of memory into a type occupying more bits of memory. Compiler guarantees that promotion will never result in any change of mathematical value, or any other data loss. Example: `int a = 'A'` turns `char 'A'` into `int 65`.

3. ***demotion***

   Demotion, more formally known as ***standard conversion***, is when the compiler attempts to do something reasonable without pretense of preserving your data. Most of those transformations are intuitively obvious mathematical approximations[29]. Example: `int i = 3.14` turns `double 3.14` into `int 3`.

   Another example of standard conversion is transformation of an integer to a double, like `double x = 5`. It may seem like a promotion. Indeed, from mathematical point of view it is a value-preserving operation, merely converting `5` into `5.0`. However, on some rare platforms an `int` may have longer bit length than the one that can be stored in a `double`, resulting in a data loss when this conversion is used.

   Note also that the transformations between signed and unsigned integer types, say, `int` to `unsigned int`, while an exact match in terms of bit length, constitute a standard conversion and not a promotion since they will change the value for part of the types' range.

   Somewhat surprisingly, `long` to `int` and `double` to `long double` are also considered as *standard conversions* rather than *promotions*, even though these transformations preserve the mathematical value of the original. This exception is the result of treating `int` and `double` as integer and floating point *base* types.[30]

4. user-defined conversion via non-explicit object constructor, to be discussed in more detail later.

Since function calls involve implicit initialization in argument-to-parameter binding and return, **function calls may result in coercion**. Functions and operators, when called, will coerce their arguments into the types of their parameters, and their return values into the types they are supposed to return.

Conversion to `bool` type is an exception in two ways. First, turning any non-zero into `true` and zero into `false`, it is not motivated by the idea of approximating one value with another. Second, a non-boolean expression is converted to a boolean

[28] Meaning integer to integer, like `short` to `int`, and floating point to floating point, like `float` to `double`.

[29] The only exception to the *obvious* rule is the conversion to `bool`, considered in more detail below.

[30] The double to long double conversion also may lead to a loss of diagnostic information in "NaN" bits.

not only when used as an argument in a boolean operation, e. g. `(x + y) && (x / z)`, but also when used in a larger statement in place of a boolean expression, e.g. `if( <boolean expression> )...`

One idiomatic case of boolean conversion is the use of counting as a check of whether whatever is counted ever happens:

```
int count( int lookup, int * arr_ptr, int arr_len )
{
    int count = 0;
    for( int i = 0; i < arr_len; i++ ) {
        if( lookup == arr_ptr[ i ] ) count++;
    }
    return count;
}
bool found( int lookup, int * arr_ptr, int arr_len )
{
    return count( lookup, arr_ptr, arr_len );
}
```

In the above code, the `count()` function finds how many times an element appears in an array, and the second, `found()`, is using counting for answering true-or-false question of whether the looked up integer appears in the array at all. Of course the counting function can be used directly when this check is needed, e. g.

```
if( count( lookup, arr_ptr, arr_len ) ) ...
```

Most built-in arithmetic operators take pairs of `int` or `double` numbers as their parameters. For that reason, when called with arguments having shorter integer types (like `char`, `short` and `unsigned short`) arithmetic operators promote their arguments to `int`, so that

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

will print `65` instead of `A`.

When the function or operator identifier is ***overloaded***, there may be several candidates that could be used in a particular function call after coercion of their arguments. This possibility requires some ranking procedure for the candidates. We will discuss it later when ***function templates*** will be added to the mix of possibilities to further complicate this matter. One important principle to remember is that **the candidate ranking and selection of the best match to be used in the function call happens before coercion**.

Preventing Implicit Coercion with Deleted Overloads

Since overload selection happens before implicit coercion of arguments, function overloading can be used to forbid implicit coercion. The language feature[31] used for this effect is called ***explicitly deleted overloads***. Let's go back to the example:

```
#include <iostream>
void greet( int x )
{
    std::cout << "I have an integer number " << x << "!\n";
}
int main()
{
    greet( 5.7 ); // implicit type demotion double{5.7}->int{5}
}
```

We can prohibit the implicit demotion by creating an explicitly deleted overload for the type whose demotion we want to prevent:

```
#include <iostream>
void greet( int x )
{
    std::cout << "I have an integer number " << x << "!\n";
}
void greet( double x ) = delete; // explicitly deleted overload
int main()
{
    //greet( 5.7 ); // error:
                    // use of deleted function
                    // 'void greet(double)'

    greet( static_cast<int>( 5.7 ) ); // explicit demotion OK
}
```

2.2.7. Function Templates

Suppose we want to write a function that prompts the user for an `int`, receives the user input and returns it to the caller of the function. In its simplest form[32], this function may look like this:

[31] There are other ways of achieving this effect on the level of class constructor definitions. We will explore those later.

[32] Don't use this particular function in any practical applications. It suffers from many problems (for one thing, it leaves all possible errors in the input stream, destroying the chance of receiving any further user inputs) and is chosen for its illustrative simplicity rather than utility.

```
#include <iostream>
int prompt_and_read_int( std::string given_prompt )
{
    int user_input;
    std::cout << given_prompt;
    std::cin >> user_input;
    return user_input;
}
int main()
{
    int i = prompt_and_read_int( "Your int = " );
    std::cout << "int = " << i << ".\n";
}
```

Imagine now that we want to get another number, this time a `double`. Writing another function would do the job:

```
#include <iostream>
int prompt_and_read_int( std::string given_prompt )
{
    int user_input;
    std::cout << given_prompt;
    std::cin >> user_input;
    return user_input;
}
double prompt_and_read_double( std::string given_prompt )
{
    double user_input;
    std::cout << given_prompt;
    std::cin >> user_input;
    return user_input;
}
int main()
{
    int i = prompt_and_read_int( "Your int = " );
    double d = prompt_and_read_double( "Your double = " );
    std::cout << "int = " << i << ", double = " <<  d << ".\n";
}
```

But now we have two functions which do the same thing in the same way and differ only in their parameter and return types. This kind of redundancy can be avoided with templates.

A ***function template***[33] is a recipe for creating functions for different type

[33] There are also *class* templates. We will study them later.

combination of their parameters that otherwise do the same thing. Syntactically, function templates look almost like function definitions, but start with the word `template` followed by the list of ***template parameters*** in angle brackets. Unlike function parameters, template parameters may include *type* names. Those names are introduced with the `typename` keyword and may be used in lieu of various specific types in the template function definition.

Template function *call* looks like a regular function call, but may[34] need to include not only the list of the usual function arguments, but also the ***template arguments*** in angle brackets. The template call instructs the compiler to make an actual function with the specific template arguments provided, and call that function using the regular arguments of the template call. This process is called ***template instantiation***. If the same combination of template arguments appears in several template calls, the first instance of the template function is reused.

Here is how our original problem can be solved with templates and without redundancy:

```
#include <iostream>
template<typename T> // 'T' stands for any type with '>>'
T prompt_and_read( std::string prompt )
{
    T user_input;
    std::cout << prompt;
    std::cin >> user_input;
    return user_input;
}
int main()
{
    // 'int' instance of 'prompt_and_read':
    int i = prompt_and_read<int>( "Your int = " );

    // 'double' instance of 'prompt_and_read':
    double d = prompt_and_read<double>( "Your double = " );

    std::cout << "int = " << i << ", double = " <<  d << ".\n";
}
```

Template Argument Deduction

Sometimes the compiler can instantiate a function template without explicit template argument list. Similar to function overloading mechanism, ***template argument deduction*** selects the right function based on the types of the regular arguments included in the template call. For example, suppose we want to compare the user input with the value provided to the comparison function:

[34] In certain situations we will consider shortly, function templates can be used without explicit template arguments.

```
#include <iostream>
template<typename T>
T prompt_and_read( std::string prompt )
{
    T user_input;
    std::cout << prompt;
    std::cin >> user_input;
    return user_input;
}
template<typename T>
void compare( std::string prompt, T rv )
{
    T user_input = prompt_and_read<T>( prompt );
    if( user_input < rv ) {
        std::cout << "Your number is way too small!\n";
    }
    else {
        std::cout << "Wow!\n";
    }
}
int main()
{
    // 100500 being 'int' implies <int> template:
    compare( "Your int = ", 100500 );
    //compare<int>( "Your int = ", 100500 ); // also OK

    // 7234.43 being 'double' implies <double> template:
    compare( "Your double = ", 7234.43 );
    //compare<double>( "Your double = ", 7234.43 ); // also OK
}
```

In the above code, the `compare` template does not need the explicit template argument list because it can deduce the required type from the type of its regular argument `rv`.

Template Specialization

Our template for comparing user input with given reference value works for any type which has the comparison operation "<". However, comparing C-strings using "<" — while still possible — will have nothing to do with the actual content of those strings[35]. To implement the usual lexicographic[36] order comparison for

[35] Can you see why?

[36] Meaning the "dictionary" order.

the `compare` template, we need special handling for the C-string case. Definition of the template for a special case follows the usual template syntax, but leaves the template parameter list empty, as in `template<>`. The parameter list of the special case with the type arguments specific to that special case may[37] need to be included in the angle brackets after the name of the template, as if it were a template call. **The specialized version of the template must appear after the generic version of the same template.**

```
#include <iostream>
#include <cstring>
template<typename T>
T prompt_and_read( std::string prompt )
{
    T user_input;
    std::cout << prompt;
    std::cin >> user_input;
    return user_input;
}
template<typename T> // generic template
void compare( std::string prompt, T rv )
{
    T user_input = prompt_and_read<T>( prompt );
    if( user_input < rv ) {
        std::cout << "Your number is way too small!\n";
    }
    else {
        std::cout << "Wow!\n";
    }
}
template<> // same template specialized for C-strings
void compare<const char*>( std::string prompt, const char* rv )
{
    const char* user_input =
        prompt_and_read<std::string>( prompt ).c_str();
    if( strcmp( user_input, rv ) < 0 ) {
        std::cout << "Your word is near dictionary start!\n";
    }
    else {
        std::cout << "Wow!\n";
    }
}
```

[37] Again, sometimes the compiler can figure it out without explicit argument list.

```
int main()
{
    // 100500 being 'int' implies <int> template:
    compare( "Your int = ", 100500 );

    // 7234.43 being 'double' implies <double> template:
    compare( "Your double = ", 7234.43 );

    // "zero" being a string LITERAL implies <const char*>:
    compare( "Your C-string = ", "zero" );

    // BUT: "boo" being a string implies <std::string>:
    compare( "Your std::string = ", std::string( "boo" ) );
}
```

Non-type Template Parameters

Since templates are instantiated by the compiler at compile time, the actual arguments binding to non-type template parameters must be compile-time constants. For example[38],

```
#include <iostream>
template <typename T, int length> // non-type 'length'
void print( T * ptr )
{
    std::cout << "[ ";
    std::string separator = " ";
    for( int i = 0; i < length; i++ ) {
        std::cout << separator << ptr[ i ];
        separator = ", ";
    }
    std::cout << " ]\n";
}
int main()
{
    int primes[] = { 2, 3, 5, 7, 11, 13 };

    // 'constexpr' is our promise of compile-time availability
    int constexpr len = sizeof( primes ) / sizeof( primes[0] );
```

[38]Not that it ever makes sense to do it this way: with this approach, the compiler will have to instantiate a new version of the `print` template every time `print` is called with a different length. It is much better to just pass the length as a regular parameter:
`template<typename T> void print( T * ptr, int length ) {...}`

```
    print<int, len>( primes );

}
```

2.2.8. Template Deduction, Overload and Coercion Order

Template type resolution, function overloading and argument coercion create competing possibilities when a function call in the code can correspond to different overloads, templates, or coercions. Compilers use ***overload resolution process*** for determining which exact function or template instance will be called in each case a function call with a specific identifier and signature is used in code.

First, the candidate list is generated in two steps:

1. All templates that allow ***template argument deduction*** appropriate for this call without any type coercion are instantiated and added to the candidates list.

2. All non-template functions that can match this call, possibly after coercion of the call arguments, are added to the candidates list.

All template instances and functions used as candidates must be in visibility scope of the function call.

Second, the candidates are ranked, according to the following ranking rules[39]. The rule appearing earlier in the list trumps any one that follows it:

1. exact match of argument types is better than coercion

2. non-template version is is better than template

3. more specialized template is better than more general

4. exact match is better than promotion

5. promotion is better than demotion

6. built-in coercion is better than user-defined one

If an unresolved ambiguity remains after those rules are applied, or if the selected best candidate is explicitly deleted, compiler fails with a compile-time error.

```
#include <iostream>
#include <cstring>
template<typename T>
T prompt_and_read( std::string prompt )
{
    T user_input;
    std::cout << prompt;
```

[39] This is a simplified list.

```
    std::cin >> user_input;
    return user_input;
}
template<typename T> // generic template
void compare( std::string prompt, T rv )
{
    T user_input = prompt_and_read<T>( prompt );
    if( user_input < rv ) {
        std::cout << "Your number is way too small!\n";
    }
    else {
        std::cout << "Wow!\n";
    }
}
template<> // same template specialized for C-strings
void compare<const char*>( std::string prompt, const char* rv )
{
    const char* user_input =
        prompt_and_read<std::string>( prompt ).c_str();
    if( strcmp( user_input, rv ) < 0 ) {
        std::cout << "Your word is near dictionary start!\n";
    }
    else {
        std::cout << "Wow!\n";
    }
}
int main()
{
    // 100500 being 'int' implies <int> template:
    compare( "Your int = ", 100500 );

    // 7234.43 being 'double' implies <double> template:
    compare( "Your double = ", 7234.43 );

    // "zero" being a string LITERAL implies <const char*>:
    compare( "Your C-string = ", "zero" );

    // BUT: "boo" being a string implies <std::string>:
    compare( "Your std::string = ", std::string( "boo" ) );
}
```

Built-in and user-defined functions and operators differ in how overload resolu-

tion is done for them. For built-in arithmetic operators, promotion of short integer types to int happens before any other candidates are identified and ranked. For user-defined functions and operators, all possible coercion targets are used as candidates, with the distance (in terms of conversion steps) used for ranking of the available candidates. The following code illustrates that difference:

```
#include <iostream>
double divide( double x, double y )
{
    return x / y;
}
int divide( int x, int y )
{
    return x / y;
}
template<typename T1, typename T2>
void test( std::string given_caption, T1 top, T2 bottom )
{
    std::cout
        << std::string( 30, '-' ) << "\n"
        << given_caption;
    std::cout
        << "\tbuilt-in " << top << "/" << bottom << " = "
        << top / bottom << "\n";          // LINE 18: OK
    std::cout
        << "\tuser-defined " << top << "/" << bottom << " = "
        << divide( top, bottom ) << "\n"; // LINE 21: PROBLEM
}
int main()
{

    test( "Both int:\n", 3, 2 ); // OK;
    // built-in and user-defined versions produce same result

    test( "Both double:\n", 3.0, 2.0 ); // OK;
    // built-in and user-defined versions produce same result

    test( "One int, another char:\n", 'A', 3 ); // OK;
    // built-in and user-defined versions produce same result

    //test( "One int, another double:\n", 3, 2.0 ); // error:
    // call of overloaded ‘divide(int&, double&)’ is ambiguous
```

```
37          //    21 |              << divide( top, bottom ) << "\n";
38          // BUT: built-in version in line 17 would be OK by itself
39      }
```

In the above code, line 26 and 29 tests allow the use the divisions which are exact matches for the types of the arguments in these tests. These exact-match versions are selected, resulting in full agreement of the built-in and the user-defined division.

The test on line 32 (with one `int` and one `char` argument) results in `char` to `int` promotion for both the built-in and user-defined divisions. For both, the `int` by `int` and `double` by `double` versions are considered as candidates, but the latter one is ruled out as a more distant match for the argument types provided in the call. Thus, both the selection processes result in a single and unambiguous best match for both built-in and user-defined divisions.

Finally, the test on line 35 (which involves one `int` and one `double` argument) works for the built-in and fails for the user-defined version of division. The built-in version considers as candidates only those possibilities that result in promotion, leaving only one choice — namely the `double` by `double` division. For the user-defined division, both the `double` by `double` division and the `int` by `int` division are considered as candidates, and are ranked as equal (i. e. equidistant from the case at hand, both requiring one coercion). This selection process ends with an ambiguity resulting in the compile-time error shown in the comments above.

2.3. Summary

2.3.1. 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[40] applied to other — shorter — expressions (e. g. `f(x)`),

2.3.2. 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       // ...
```

[40] The word “function” is understood in the general sense which includes operators as well as proper functions.

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[41];
- 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[42];
- 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`)[43];
- 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`)[44];
- the *memory location* of the object, if any — i. e. the place of the box in the memory storage (in the example, `x` is a local variable of the `main()` function, thus residing on the stack, in the `main()`'s stack frame) or lack thereof;
- 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 sealed or remains open after initialization, 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.

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

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

[43] 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.

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

2.3.3. Order of Evaluation in Compound Expressions

Compound expressions constructed from smaller constituent sub-expressions may need some additional rules to clarify their order of evaluation. From mathematics, we already know about *precedence* and *associativity*. C++ introduces another concept, *sequence points*, into this consideration. The aim of this section is to make you aware of the problem, rather than to give a long reference list of rules helping you to "solve" it. The main reason for choosing this approach is my belief that the best way to deal with this problem is to avoid it altogether.

To avoid the issues related to expression evaluation order:

- use parentheses to resolve the ambiguity related to precedence and associativity, writing

  ```
  ( a[i] ) = ( i++ );
  ```

  instead of

  ```
  a[i] = i++;
  ```

- separate sub-expressions with interacting side effects into separate statements when in doubt about order of side effects, writing

  ```
  a[i] = i; // update array element with old value of index
  i++;
  ```

  or

  ```
  a[i+1] = i; // update array element with new value
  i++;
  ```

  to clarify the intended effect in the same example situation[45].

Take the rest of this section as an attempt to convince you to use the technique outlined above.

Precedence Rules

The ***order of precedence*** is an order relation fixed on the set of all operations. When *different* operations are used in constructing a compound expression from smaller constituent sub-expressions, the operation with the highest order of precedence is evaluated first. This rule, called the ***rule of precedence***, resolves the ambiguity of interpreting `1 + 2 * 3` as `1 + ( 2 * 3 )` instead of `( 1 + 2 ) * 3` based on the fact that operation `*` has higher precedence than operation `+`.

C++ standard specifies precedence rules. See standard references for the complete information[46]. Here is a brief synopsis:

- **PE(MD)(AS) > comparisons > logical > ternary > assignments**
  so that `x + y * z <= a + b && a < b` is the same as
  `( ( x + ( y * z ) ) <= ( a + b ) ) && ( a < b )`

[45] More on that example later in this section.
[46] If you have the nerve to read it.

- **unary > binary**
  so that `a && !b` is the same as `a && ( !b )`; note that array element expression `arr[i]` is interpreted as unary operation `[...]`, and so is the function call `func(...)`, regardless of the number of parameters in the `(...)`

- **suffix > prefix**
  so that `++arr[i]` is the same as `++( arr[i] )`

Association Rules

The associative *property* states that the result does not depend on how the arguments of the *same* operation are grouped together when that operation is used repeatedly. For instance, associativity of operation `+` guarantees that the result of `1 + 2 + 3` does not depend on whether we interpret it as `( 1 + 2 ) + 3` or `1 + ( 2 + 3 )` — both of these result in 6.

For a non-associative operation, such as `/`, the ***association rule*** resolves the ambiguity of interpreting `12 / 6 / 2` as `( 12 / 6) / 2` based on "left to right" associativity of `/`, as opposed to `12 / ( 6 / 2 )` which would be the case, if `/` were "right to left" associative.

C++ standard specifies association rules — see any one of the standard references for the complete story. The main thing to remember[47] is this: **most operations are left-to-right associative**:

`12 / 6 / 2` is the same as `( 12 / 6 ) / 2`

but *ternary operator* and all *assignments* are right-to-left associative:

`a += b -= c` is the same as `a += ( b -= c )`

and

`a ? b : c ? d : e` is the same as `a ? b : ( c ? d : e )`

Parentheses

As should be already clear from our examples, precedence and association rules are merely conventions that save us from drowning in the sea of parentheses. These rules are entirely disposable if one is willing to enclose every constituent expression in parentheses, getting a fully parenthesized expression as a result. So, one can view the precedence and associativity rules as a way to restore those parentheses that were omitted for the sake of brevity.

Fully parenthesized expressions in mathematics are computed incrementally from inside out, starting with those sub-expressions that do not contain any sub-expressions. No additional rule is required for their evaluation. So, for instance, in the expression

```
f( a(...), b(...), ... );
```

the sub-expressions `a(...), b(...), ...` will be computed before the `f(...)`.

Sequencing

When one moves from mathematics to C++ (or even C), *side effects* create an entirely new set of issues. Take the fully parenthesized expression

[47] This rule misses some fringe cases but is right in most situations.

```
f( a(...), b(...), ... );
```

In mathematics, since objects were immutable for the whole duration of expression evaluation, the precise order of evaluation of the sub-expressions `a(...)`, `b(...)`, `...` did not matter and was unresolved by the precedence and associativity rules. Now assume that the evaluation of the sub-expression `a(...)` has a side effect on some of the variables involved in evaluation of sub-expression `b(...)`. In this context, the order of evaluation of the sub-expressions `a(...)`, `b(...)`, `...` is significant. Furthermore, the exact moment when the side effect of each constituent expression "takes hold" may affect evaluation of the compound expression. Kernighan and Ritchie illustrate this phenomenon in their book on C [2][48] with the example:

```
a[i] = i++;
```

The issue at stake is whether the state of `i` is updated before or after it is used in `a[i]`.

Side effect ordering is largely left to discretion of the compiler. The meager guarantees that the compiler is required to provide concerning side effect ordering are called[49] ***sequence point*** rules. Sequence points are moments in the evaluation of expression when compiler guarantees to apply all side effects in the (sub-)expressions encountered thus far. The most basic sequence point is the end of the whole compounded expression[50]. The end of expression may be marked by

- semicolon "`;`" — for example in
  - the expression statement with side effect,
  - `for( <expression1>; <expression2>; ... )` and
  - `return <expression>;` or
- closing parenthesis "`)`" — for example in
  - `if( <expression> )`,
  - `while( <expression> )`,
  - `do ... while( <expression> )`,
  - `for( ...; <expression> )`, and
  - `switch( <expression> )`.

Another interesting case is the sequence point right before disjunction `||`, conjunction `&&` and ternary conditional operator `?`. It provides an example of ***lazy evaluation***. Lazy evaluation is the way of computing an expression by doing the least work necessary for determining the result of that expression. Lazy evaluation stands in contrast with the ***eager evaluation*** used by C++ most of the time.

[48] Page 51 in Chapter 2 - Types, Operators and Expressions > Section 2.12 - Precedence and Order of Evaluation starting.

[49] This terminology was changed in the recent standard to *sequenced before*, *sequenced after*, and *unsequenced* or *indeterminately sequenced*.

[50] Meaning the expression which is not a sub-expression in any bigger expression.

Eager evaluation of an expression evaluates all of its constituent sub-expressions, regardless of whether their value is needed.

For the conjunction expression `<expression1> && <expression2>`, the `<expression1>` is guaranteed to be evaluated first, with all its side effects applied. If the value of `<expression1>` is `false`, the `<expression2>` is not evaluated (so that its side effects don't take place) and the whole expression evaluates to `false`. Only if the value of `<expression1>` is `true`, the `<expression2>` is evaluated. This gives rise to the idiom `<check that an object exists> && <do something with it>`, where the second step does not make sense[51] if done after the first step failed. For example in

```
if( answer_ptr != nullptr && *answer_ptr == 42 ){
    ...
}
```

dereferencing the pointer in `*answer_ptr == 42` makes sense only if `answer_ptr` actually points somewhere, which is exactly the condition checked by `answer_ptr != nullptr`, making the whole code safe from the runtime error.

For the disjunction expression `<expression1> || <expression2>`, the `<expression1>` is guaranteed to be evaluated first, with all its side effects applied. If the value of `<expression1>` is `true`, the `<expression2>` is not evaluated (so that its side effects don't take place) and the whole expression evaluates to `true`. Only if the value of `<expression1>` is `false`, the `<expression2>` is evaluated. This gives rise to the idiom `<action that returns true on success> || <handle the error>`, where the second step is needed only when the first step fails.

For the ternary expression `<expression1> ? <expression2> : <expression3>`, the `<expression1>` is guaranteed to be evaluated first, with all its side effects applied. If the value of `<expression1>` is `true`, the `<expression2>` is evaluated but `<expression3>` is not evaluated (so that its side effects don't take place) and the whole expression evaluates to `<expression1>`. If the value of `<expression1>` is `false`, the `<expression3>` is evaluated but `<expression2>` is not evaluated (so that its side effects don't take place) and the whole expression evaluates to `<expression3>`.

HOMEWORK: Can you implement the `if` statement using only the

1. conjunction?
2. disjunction?
3. ternary operator?

Again, to get a full story of sequence points, look up this topic in some standard reference. Here is a brief synopsis of what (I think) is worth remembering:

[51] And could result in an undefined behavior or a runtime error.

- in the expression

  `f( a(...), b(...), ... )`

  the sub-expressions `a(...), b(...), ...` are sequenced before the `f(...)` and are indeterminately sequenced relative to one another;

- whole compound expression is sequenced before anything coming after that expression;

- conjunction, disjunction and ternary operator sequence their first argument before the rest.

2.3.4. 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[52] from scratch:

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

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

```
// 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 2.1.12 (page 15), the implicit initialization `int & x = 5` happening in `pre_increment(...)` function call will fail to compile since an lvalue reference 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";
}
```

Chapter 3

Control Flow Statements

3.1. 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.

3.2. 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[53] give a warning[54]. 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";
```

[53] with compilation key `-Wall`

[54] 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.

```
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:
    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
}
```

3.3. 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.

3.3.1. `while` loop

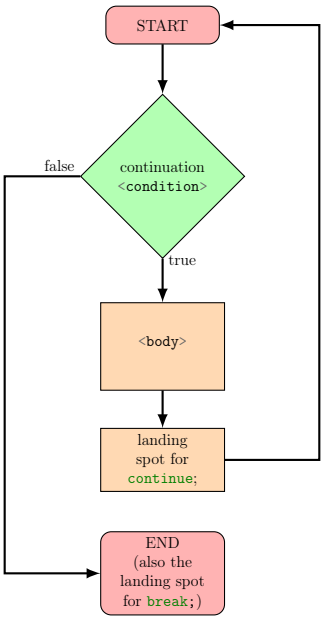

Figure 5. `while( <condition> ){ <body> }` flow chart.

3.3.2. `do...while` loop

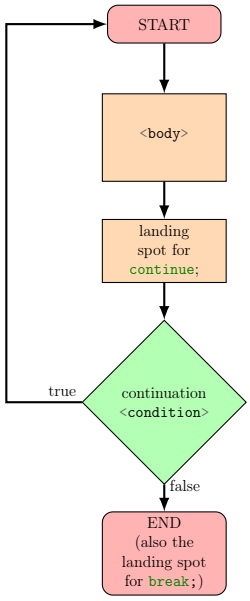

Figure 6. `do{ <body> } while( <condition> );` flow chart.

3.3.3. `for` loop

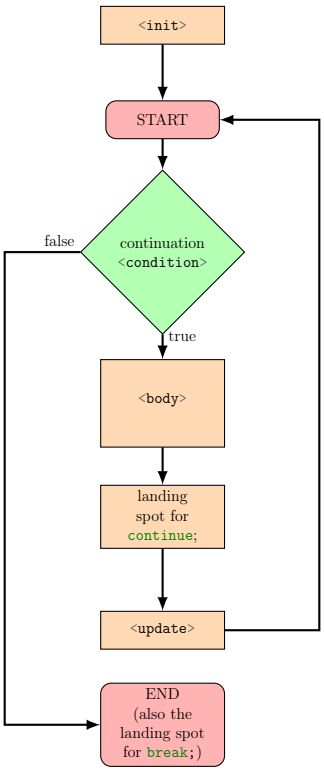

Figure 7. `for( <init>; <condition>; <update> ){ <body> }` flow chart.

Chapter 4

Pointers

4.1. Address Operator and Dereference Operator

The presence of pointers separates low level languages from high level ones, and makes the former suitable for creating the living realms of the latter, as well as for operating systems development.

C and C++ use the von Neumann machine as the realm for their programs' abstract models, representing lvalue expressions[55] as contiguous intervals of blocks in the linear, homogeneous and addressable memory of the von Neumann architecture.

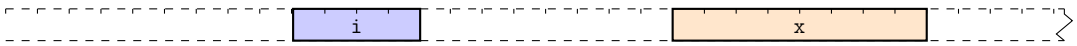

Figure 8. An `int i` and a `double x` in memory.[56].

The fact that these blocks are *individually addressable* enables both languages to reify the concept of a pointer.

A ***pointer*** of an lvalue is the memory address of the first block occupied by that lvalue, with an additional information about the *type* of the lvalue in question. In particular, the pointer "knows" how many subsequent blocks of memory are occupied by the lvalue. A ***void pointer*** is a pure address without any type information.

Pointer of an lvalue is given by the ***address operator*** `&`. If an lvalue `x` has type `T`, the expression `&x` has type of a *pointer to T*, denoted `T *`. Note that the expression `&x` is a *prvalue*[57]. For example, for type `int`:

```
int i = 5;      // i is a regular int
int * p = &i;   // &i and p are of 'pointer to an int' type
```

[55] More precisely, the objects that those expressions evaluate to.

[56] On most machines, integers have 4 byte length, and doubles — 8 byte length, as depicted in the above picture.

[57] In particular, the *expression* `&&x` never makes sense, even though this *notation* is used in rvalue reference declaration, where the symbol `&` has different meaning — not the address operator, but a reference declaration.

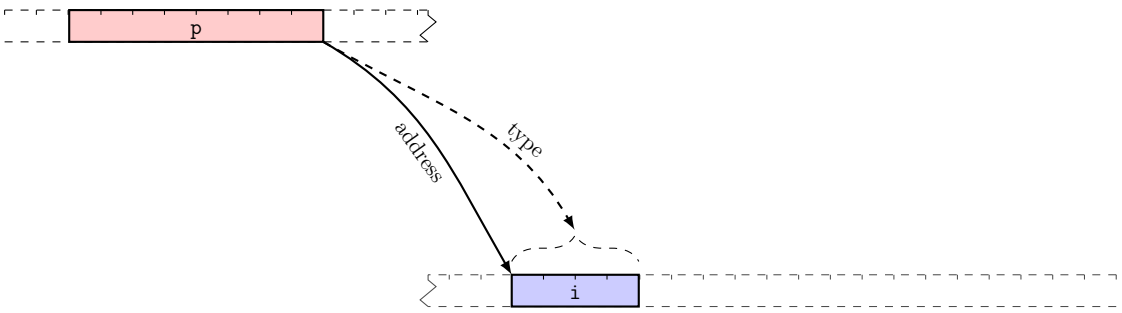

Figure 9. Corresponding `int * p` and `int i` in memory[58].

The inverse ***dereference operator*** `*` takes a pointer and gives back the lvalue that pointer points to, so that going from `i` to `p` as above will make `*p` same as `i`, which can be checked by using either one for reference initialization:

```
int & j = *p;    // same thing as 'int & j = i;'
```

and then verifying that `i` and `j` are the same.

C++ compiler will coerce a non-void pointer into a void one:

```
void * v = &i;  // coercion: type of i is lost
```

but will fail to do coercion in the opposite direction[59]:

```
int * q = v;     // error:
                 // invalid conversion from ‘void*’ to ...
```

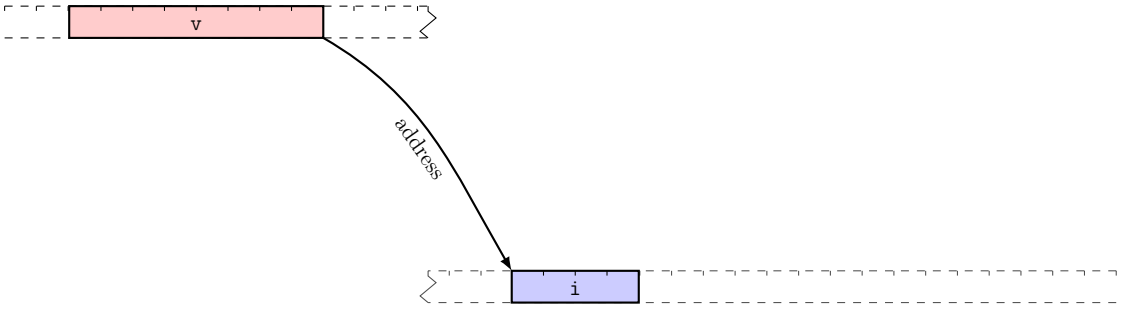

Figure 10. Corresponding `void * v` and `int i` in memory.

A void pointer cannot be dereferenced:

```
*v;              // error:
                 // ‘void*’ is not a pointer-to-object ...
```

because the compiler does not know what object it is to return, even though it knows the address of that object. However, if the type of the object located at void pointer's address is known, C++ allows that void pointer to be explicitly typecast:

[58] On most machines, all pointers (regardless of the type they point to) have 8 byte length, as depicted in the above picture.

[59] In contrast with C where an implicit coercion from void to non-void pointer is valid.

```
    int * cast_ptr = static_cast<int *>( v );
```

Here is the full program:

```
#include <iostream>
int main()
{
    int i = 5;      // i is a regular int
    int * p = &i;   // p is a pointer to an int
    void * v = &i;  // v is a void pointer (type of i is lost)

    std::cout
        << "Number i, defined by 'int i = " << i << ";'"
        << " is an integer, and it occupies "
        << sizeof( int ) << " bytes in memory.\n"
        << "Pointer p defined by 'int * p = &i;'"
        << " contains the address " << p
        << " of the first byte of i.\n"
        << "In addition to the address, p knows"
        << " that its target *p is an integer "
        << " occupying " << sizeof( *p ) << " bytes.\n";

    // check that dereference of p is the same as i
    int & j = *p;   // same thing as 'int & j = i;'
    j = 7;
    std::cout
        << "*p is really the same as the i itself: "
        << "defining j as a reference to *p "
        << "and changing j to 7 makes i = " << i << ".\n";

    //int * q = v;  // error:
                    // invalid conversion from ‘void*’ to ...

    //*v;           // error:
                    // ‘void*’ is not a pointer-to-object ...

    int * cast_ptr = static_cast<int *>( v );
    std::cout
        << "*cast_ptr = " << *cast_ptr
        << " and " << "sizeof( *cast_ptr ) = "
        << sizeof( *cast_ptr ) << ".\n";
}
```

The size of all pointers themselves (as opposed to what they point to) is determined by the system's architecture. On a 32-bit architecture, the address size is 32 bits (or 4 bytes). Thus the addresses can range from 0 to $2^{32} - 1$, allowing for the maximum addressable memory size of 4 GiB. On a 64-bit architecture present on most modern computers, addresses occupy 64 bits or 8 bytes, theoretically allowing 16 exabyte addressable memory. In practice, the maximum size of addressable memory on a 64-bit machine is often set at 128 TiB.

There is a built-in literal value `nullptr` of a pointer that does not point to any address. When used in boolean context, `nullptr` is coerced into the boolean `false`. Any other pointer is coerced into `true` in the same situation. Curiously, while allowing coercion to `bool`, C++ does not coerce pointers into `int`, requiring explicit type casting to achieve that effect.

4.1.1. Pointer Declaration Caveats

There are a few notation caveats related to pointers. In spite of what you may think,

```
int * a, b;
```

does not define two pointers but only one. The above is equivalent to

```
int * a;
int b;
```

To declare two pointers, use

```
int * a, * b;
```

or

```
int * a;
int * b;
```

In view of the above, one may be tempted to interpret the * as a dereference operator in all cases, reading the

```
int * a;
```

as a statement that `a` is a pointer simply because `* a` is an integer (which it is). However, this intuition fails to give a satisfying parsing of

```
int * a = &i;
```

where it is the `a` itself and not the `*a` that equals `&i`. It is better to give the symbol * two different meanings, depending on where it appears:

```
int * a = &i;    // * means that a is a pointer
*a = 7;          // * is the dereference operator yielding i
```

Another caveat is distinguishing constant pointers from pointers to constants. It is easy to determine which one is which with a simple rule: everything before * pertains to the target of the pointer, and everything after the * — to the pointer itself:

```
int main()
{
    int i = 10;
    int j = 100;
    int k = 1000;

    const int * a = &i;       // pointer to constant
    int * const b = &j;       // constant pointer
    const int * const c = &k; // constant pointer to constant

    //*a = 7; // error:
              // assignment of read-only location

    a = b;    // fine: a is not constant and can change

    *b = 7;   // fine: *b is not constant and can change

    //b = a;  // error:
              // assignment of read-only variable

    //*c = 7; // error:
              // assignment of read-only location

    //c = a;  // error:
              // assignment of read-only variable


    // also note:
    const int m = 5;
    //int * d = &m; // error:
            // invalid conversion from ‘const int*’ to ‘int*’
}
```

Pointers to constants are much more common than constant pointers, and still more common than constant pointers to constants!

4.1.2. Example: Three Versions of `swap`

To illustrate the utility of pointers and to differentiate them from references let's consider how either one of them can be used for implementing a `swap()` function. The task at hand is to interchange values between two variables, say `x` and `y`. Before we make functions for that task, perhaps we should recall how to achieve it directly when no argument-to-parameter binding issues complicate the situation. The most obvious "solution"

```
x = y;
y = x;
```

would not work in general because the first assignment would destroy the old value of `x`, making it impossible to assign it to `y` (unless the two were the same to begin with). Thus we need to save a temporary copy of `x` before assigning anything to `x`, so that we can recover the original value of `x` when we need to assign it to `y`. Assuming for simplicity that all involved are integers, we can do the swapping like so:

```
int tmp = x;
x = y;
y = tmp;
```

It is tempting to turn these three lines of code into a function:

```
void swap( int a, int b )
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

so that instead of repeating them every time we want to swap two variables we could just call `swap( x, y )` instead. It should be clear, however, that this way of doing it would not work. Indeed, when arguments of the function call `swap( x, y )` are bound to parameters `a` and `b`, the implicit initialization

```
int a = x; // implicit
int b = y; // implicit
```

is *copying* the values of `x` and `y` into `a` and `b` respectively. The swapping code, while fully functional, is acting on the variables in the stack frame of `swap( int a, int b )` and has no effect on the `x` and `y` outside of that frame:

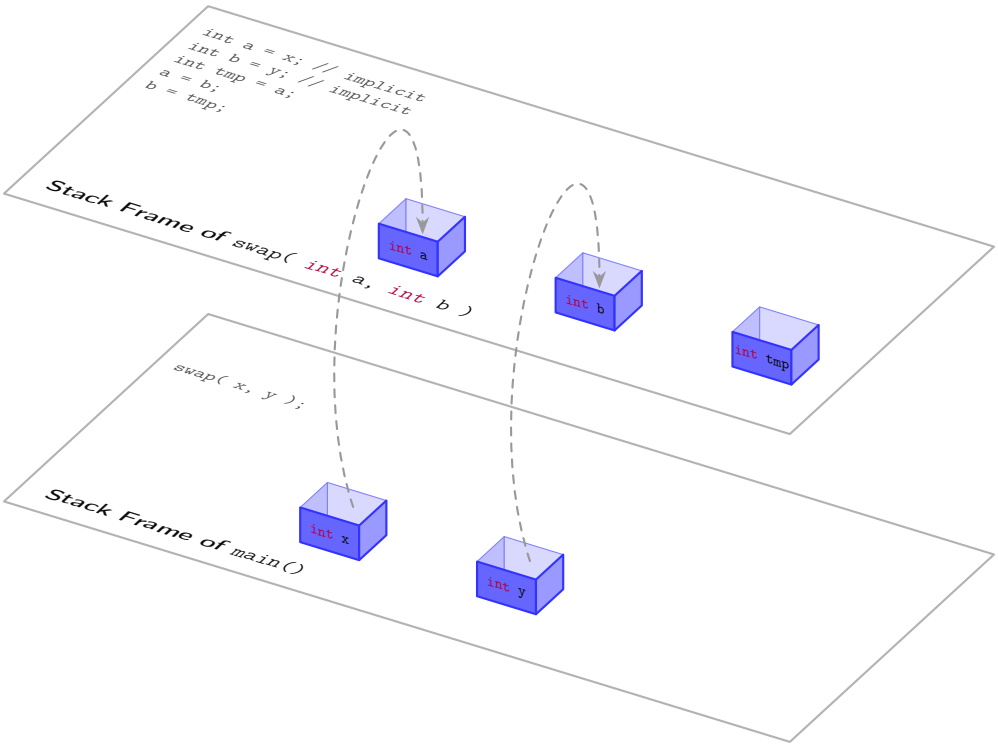

Figure 11. Call-by-value `swap( x, y )` function.

We have already seen in the previous course how reference parameters can address the issue we just encountered. Let's program `swap()` to take reference parameters:

```
void swap( int & a, int & b )
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

When arguments of the function call `swap( x, y )` are bound to parameters `a` and `b`, the implicit *reference* initialization

```
int & a = x; // implicit
int & b = y; // implicit
```

is not copying anything, but rather is opening teleport tunnels from `x` and `y` into `a` and `b` respectively. The swapping code, while still acting on the variables in the stack frame of `swap( int a, int b )`, is in effect acting on the original variables `x` and `y` on the lower frame by virtue of teleporting:

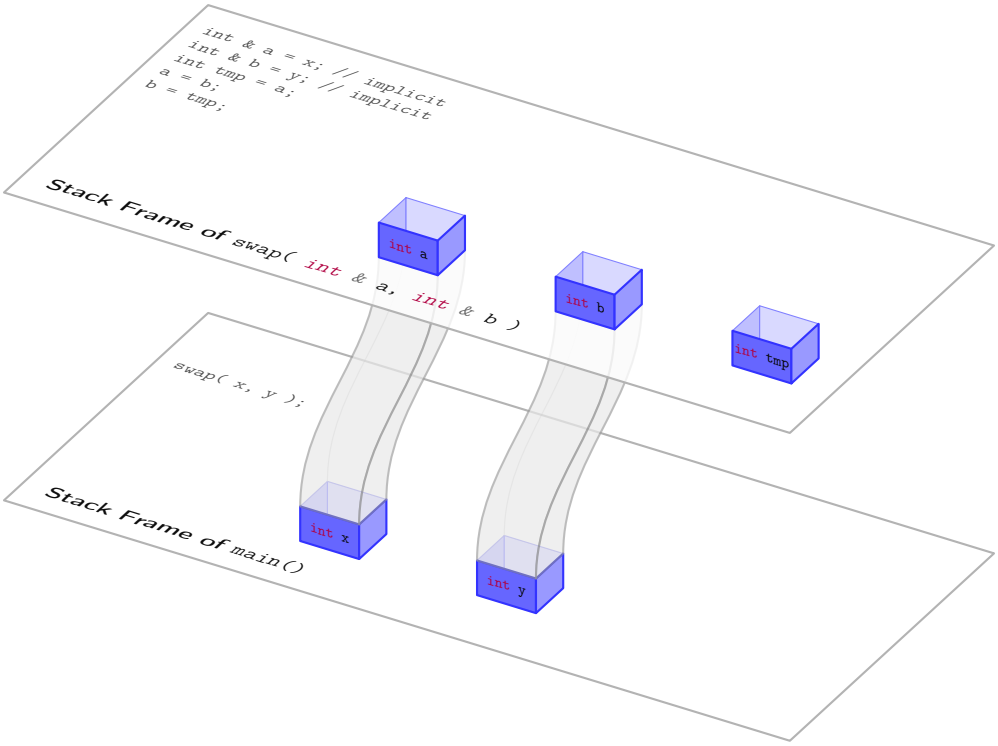

Figure 12. Call-by-reference `swap( x, y )` function.

Finally, the same effect can be achieved with pointers, albeit less elegantly. However, this is the way it is done in C, and that alone makes it worth knowing. We can define the `swap()` function to accept *pointer* arguments:

```
void swap( int * p, int * q )
{
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```

This way, the function must be called not with the two integer variables that need to be swapped, but with their addresses: `swap( &x, &y )`. When arguments of the function call `swap( &x, &y )` are bound to parameters `p` and `q`, the implicit initialization

```
int * p = &x; // implicit
int * q = &y; // implicit
```

copies the *pointers* of `x` and `y` into `p` and `q` respectively. This is still an instance of call-by-value, just like the first (failed) attempt to get a `swap()` function. However, unlike before, the actual variables that need to be swapped are accessible in the stack frame of `swap( int * p, int * q )` through dereferencing pointers. Indeed, `*p` is `x` and `*q` is `y`, so that the above swapping code has clear meaning and effect on the lower stack frame of `x` and `y`:

```
void swap( int * p, int * q )
{
    int tmp = *p;    // i.e. tmp = x
    *p = *q;         // i.e. x = y
    *q = tmp;        // i.e. y = tmp
}
```

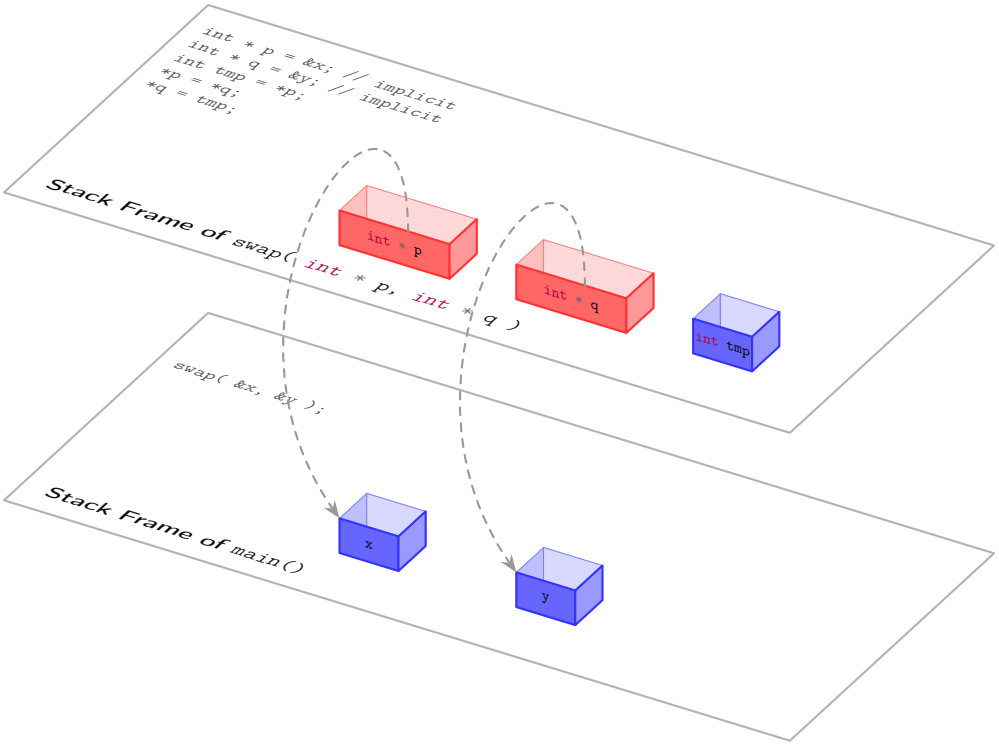

Figure 13. Call-by-pointer `swap( &x, &y )` function.

This particular type of call-by-value is called ***call-by-pointer***.

4.2. Arrays

An ***array*** is a *contiguous* sequence of objects in memory, all having the same type. An array is defined by

- its *address* in memory;
- the common *type* of its member objects; and
- its *length*, defined as the total number of its members.

An array may be declared by specifying its type and length like so:

```
int arr[4]; // array of 4 integers
```

In C and C++ arrays are *pointers with length information*: the name of the array itself is the (typed) pointer to array's first member, combining the address and the member type information of the array into the defining characteristics of the pointer object:

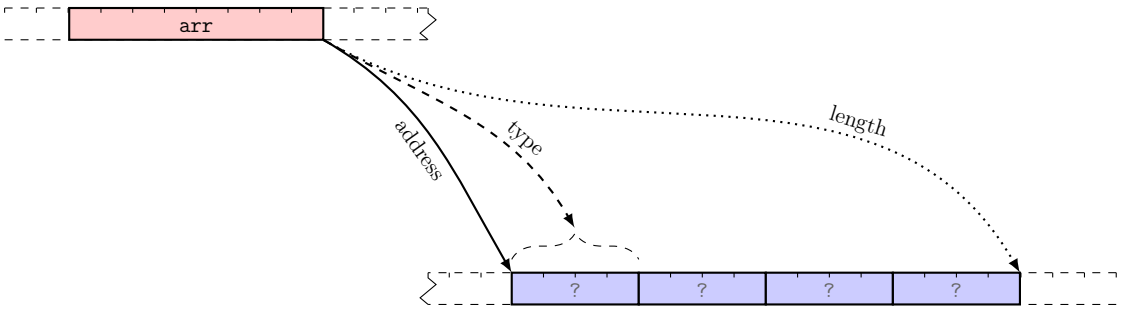

Figure 14. `int arr[4]` in memory.

4.2.1. Subscript Operator []

Any individual member of an array can be accessed using the order number of that member in the sequence. That order number is called the *index*, or — more formally — the ***subscript*** of the member in question. Subscripts start at zero, so that the "first" element is really the element with subscript zero. Consequently, the last member of an array with n elements has index $n - 1$. The ***subscript operator*** `[]` returns an individual member of an array when given the array and the subscript of the desired member:

```
int arr[4];      // arr as an array of 4 contiguous integers
arr[0] = 5;      // 5 is the ``first'' integer
arr[3] = 7;      // 7 is now the last of these integers
```

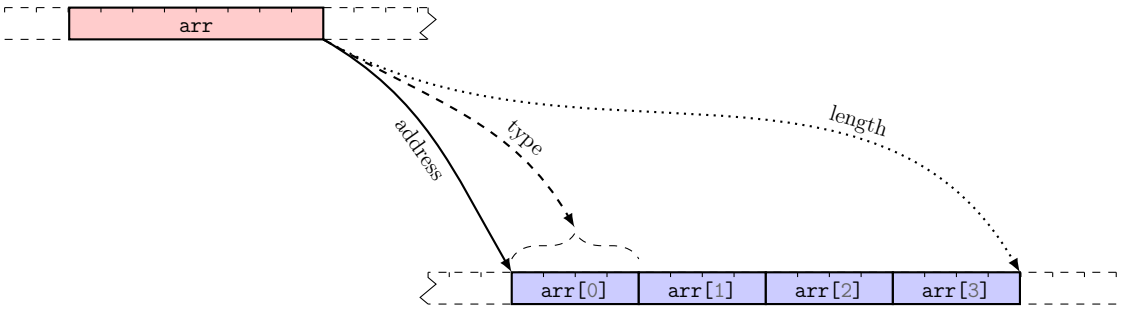

Figure 15. `int arr[4]` members described by subscript.

Note that the subscript operator returns an lvalue, so we can assign to it. The `[]` notation is similar to the $*$ symbol in the sense that it means different things in different contexts:

```
int arr[4];      // [n] means length of arr is n
arr[0];          // [n] means n-th element of arr
arr[4];          // WRONG: there is no 3rd element in arr
```

Arrays in C and C++ do not check that the subscript used in the subscript operator falls within the array bounds. Out-of-bounds array access results in *undefined behavior* and it is the responsibility of the programmer — and not of the compiler — to avoid it.

4.2.2. Array-to-Pointer Decay

In most situations an array "forgets" the length information and behaves as if it were just a constant pointer to its first[60] element:

```
int arr[4]; // array of 4 integers
int * const arr_ptr; // almost the same thing
```

There are very few situations when it acts the part commensurate with its full — i. e. the array — status. The most notable among those is the `sizeof` pseudo-function[61]. We can see it by continuing with our example:

```
sizeof( arr ); // is 16
sizeof( arr_ptr ); // is 8
```

assuming our platform allocates 4 bytes for an `int` and 8 bytes for a pointer (of type `int *` or anything else).

The difference between an array and the pointer which is part of that array's identity is the *length* information. Its loss is called ***array-to-pointer decay***. For example, whenever an array appears on the right hand side of an assignment operator by itself, it decays to pointer.

```
int arr[4];
int * const arr_ptr = arr;
```

That pointer addresses the starting (i. e. zeroth) element of the array and has the type of the array's elements:

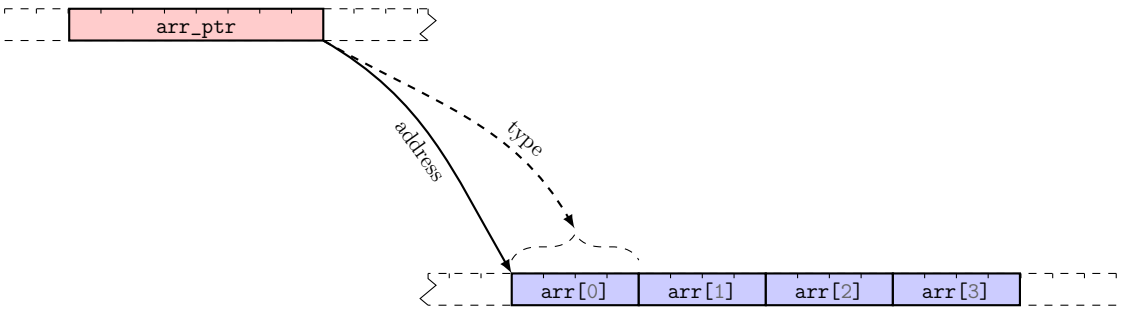

Figure 16. `int arr[3]` decayed to `int * const` pointer.

No actual array data is lost in the decay. The loss is confined to the information "known" to the pointer about what it is pointing to. In a sense this situation resembles coercion of a typed pointer to a void one. We can see all stages of degradation

[60] Meaning the element with subscript *zero*.

[61] Other similar cases include array behavior in `typeid` and `alignof` operators.

```
int arr[4];
int * const arr_ptr = arr;
void * const void_ptr = arr_ptr;
```

when referring to the same 4 integers in memory:

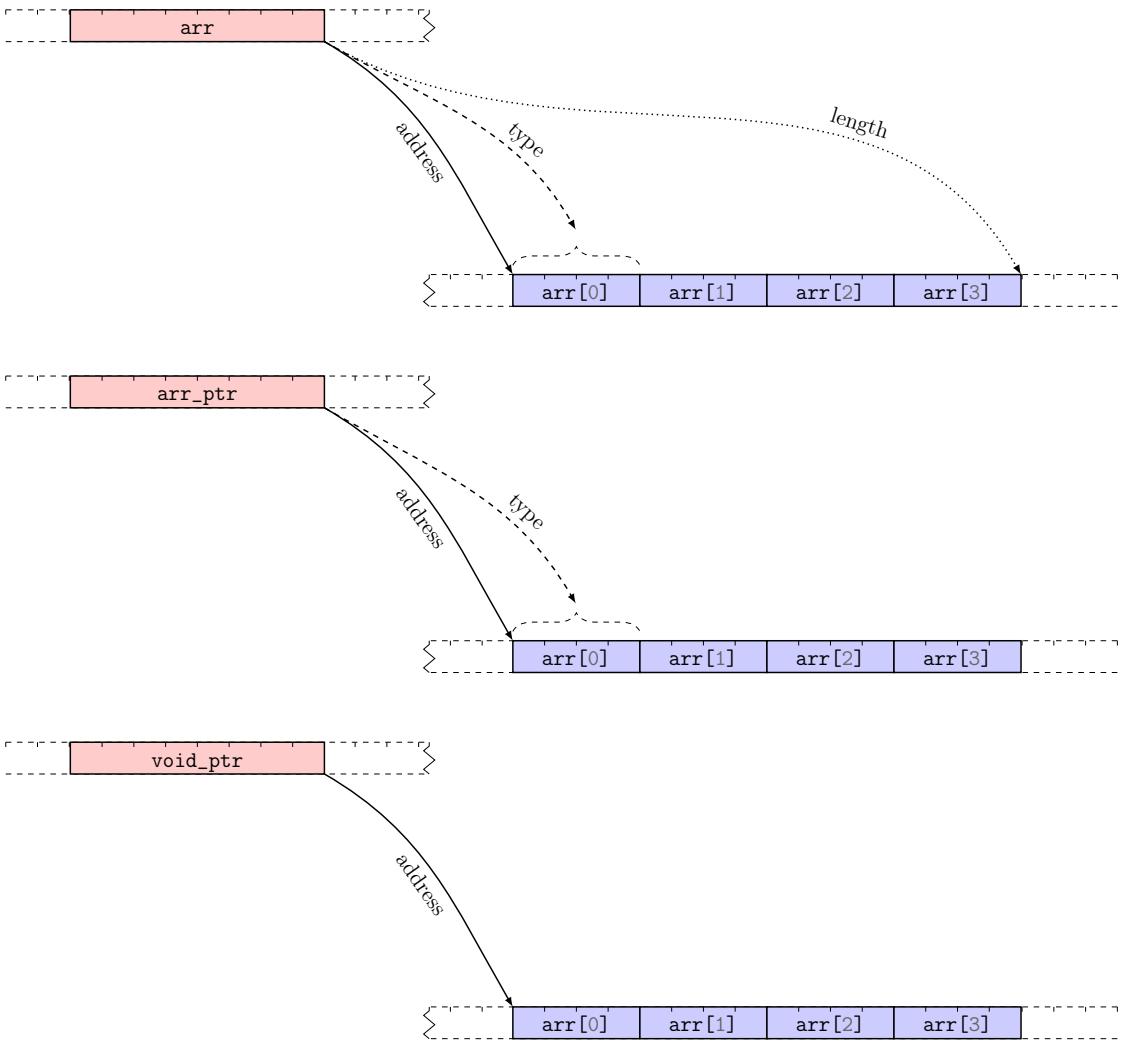

Figure 17. `int arr[4]` and its various stages of degradation.

When the lvalue on the left hand side of the assignment operator is itself a pointer, compiler will do coercion (resulting in the decay):

```
int arr[4];
int * arr_ptr = arr;                // coercion via decay
int * const const_arr_ptr = arr;    // coercion via decay
```

but a true array cannot accept any *assignment*, effectively behaving as if it were a constant pointer[62]. Even more interesting, an array cannot be *initialized* with a

[62] Formally speaking, an array is not considered a constant pointer. Instead, there is a formal rule that an array cannot be assigned to. This explains the somewhat cryptic error message in the above code.

regular expression. A special ***initializer*** syntax must be used[63]. We will consider ***list initialization*** of an array later, but for now let's continue with the previous code snippet:

```
// assignment attempt:
arr = arr_ptr;  // error:
                // incompatible types
                // in assignment of ‘int*’ to ‘int [3]’

// initialization attempt:
int arr2[5] = arr;  // error:
        // array must be initialized
        // with a brace-enclosed initializer

// another initialization attempt:
int arr2[] = arr; // results in two errors:
        // the one immediately above and this one:

        // error:
        // initializer fails to determine size of ‘arr2’
```

Curiously, in argument-to-parameter binding the compiler will accept code that appears to do the implicit initialization `int arg_arr[] = param_arr` — the kind resembling the last command in the failing code above. This is misleading, since the appearance of an array argument is masking array-to-pointer decay taking place during the binding:

```
#include <iostream>
int length( int a[] )
{
    // this will fail since a
    // has decayed to pointer by now:
    return sizeof( a ) / sizeof( int );
}
int main()
{
    int arr[4];

    // direct computation works:
    int len = sizeof( arr ) / sizeof( int ); // is 4 = 16/4
```

[63] An initializer is not considered to be an *expression*.

```
    // but the function fails:
    int computed_length = length( arr ); // is 2 = 8/4

    std::cout
        << "Directly computed length is " << len
        << " and the length function gives " << computed_length
        << ".\n";
}
```

Even though the compiler accepts functions — like the above `length` — which give the impression of accepting and *working with* an array, never declare a function with an array input parameter. To reduce the possibility of confusion, the above function `length()` should have been written with signature

```
int length( int * const a );
```

or

```
int length( int * a );
```

making it obvious that it works with a pointer and not a true array (even though it can accept an array).

4.2.3. Array Initialization (Automatic Memory)

When an array is declared and initialized with

```
int arr[length] ...
```

the array *length* must be

- greater than zero; and
- known at compile time[64].

The latter is a stronger requirement than merely insisting on having a `const`. Indeed, a constant is something that cannot be changed once initialized, but is not necessarily known at the compile time:

```
int user_input;
// ... get user input
const int length = user_input;  // this is fine
int arr[length];                // this is NOT!
```

What we need for an array size is a `constexpr`[65]:

[64] For dynamically allocated arrays the `length` can be a variable. Dynamically allocated arrays are declared and initialized using different syntax `int * arr_ptr = new int[ length ]` and will be considered later.

[65] For compatibility with pre-C++11 standards which did not have `constexpr`, compiler will treat a mere `const` as if it were a `constexpr`, as long as it can determine its value at compile time. So, something like `const int length = 5;` can actually be used in `int arr[length];`

```
constexpr int length = 4;        // this is fine
int arr[length];                 // this is fine too
```

An array can be initialized using ***brace initializer***. Specific type of brace initialization, called ***list initializer***, allows you to provide a list of array member values. When providing a *full* list for all members, you can even omit the length declaration of the array:

```
int arr[] = { 2, 3, 5, 7 };
        // length of arr is determined automatically to be 4,
        // arr[0] = 2, arr[1] = 3, arr[2] = 5, arr[3] = 7
```

When both an explicit length declaration and an initializer list are used, the length of the initializer list must not exceed the declared array length. When the initializer list is shorter than the length, the undefined tail entries of the array get the default value, which is zero for an `int`:

```
int arr[5] = { 2, 3, 5 };
    // is equivalent to
    // int arr[] = { 2, 3, 5, 0, 0 };
```

This is called ***partial initialization***.

In contrast with braced initialization, merely declaring an array with

```
int arr[4];
```

does not guarantee in general that all its entries will be zero. Uninitialized values of a *global* array are guaranteed to be zero. However, ***local* arrays must be explicitly initialized** because they may contain any garbage data otherwise. The reason is simple: local objects are allocated in automatic memory (which — in most cases — means on the stack) and that type of memory is not cleared before use for efficiency sake. For example, the following program:

```
#include <iostream>

int main(){
    for( int i = 0; i < 3; i++ ){
        int arr[10]; // uninitialized local array
        std::cout
            << "Attempt " << i + 1
            << " of allocating the array; "
            << "arr[0] = " << arr[0] << "\n";
        arr[0] = 5;
    }
    return 0;
}
```

is likely to print some garbage during the first loop, and then 5 two times, showing the leftover values from the previous loop executions. Here is what I got on my machine:

```
Attempt 1 of allocating the array; arr[0] = 1600677166
Attempt 2 of allocating the array; arr[0] = 5
Attempt 3 of allocating the array; arr[0] = 5
```

To force zero initialization, an empty initializer list can be used:

```
#include <iostream>

int main(){
    for(int i = 0; i < 3; i++ ){
        int arr[10] = {}; // zero-initialize whole array
        std::cout
            << "Attempt " << i + 1
            << " of allocating the array; "
            << "arr[0] = " << arr[0] << "\n";
        arr[0] = 5;
    }
    return 0;
}
```

producing:

```
Attempt 1 of allocating the array; arr[0] = 0
Attempt 2 of allocating the array; arr[0] = 0
Attempt 3 of allocating the array; arr[0] = 0
```

Some compiler extensions treat the empty initializer list as an equivalent of the one with a single zero:

```
int arr[10] = {0};
```

making it possible to use either variant in declaring an array without length specifier:

```
int arr[] = {0};     // legal in C++
int arr[] = {};      // illegal in C++
                     // but may have the same effect
                     // with default compiler extensions
          // both create an array with a SINGLE zero element
```

It is best to avoid the `{}` and use `{0}` for zero-initializing *all* arrays, regardless of whether those arrays come with a length specifier.

If you want to turn off those pesky compiler extensions, compile your code with `-pedantic-errors` flag, e. g. `g++ -Wall -pedantic-errors main.cpp -o main`

4.2.4. VLA Abomination

The C99 standard introduced the abomination called ***variable length arrays*** (VLA) into the C language. They have been subsequently removed from C, and — luckily — they have never been an official part of C++. However, some compiler implementations and configurations still support them by default, making it easy to cross into the forbidden territory if one is not careful or even aware of the danger. A VLA looks deceptively similar to a regular array:

```
int length =        // ... code that determines length
int arr[length];    // arr is a VLA
```

The main issue with VLAs is the risk of stack overflow. On most platforms, automatic memory size (i. e. stack memory) is relatively small. Leaving the `length` variable contingent on the runtime conditions inserts huge vulnerability into the code. A comparatively minor issue is the undefined behavior in case of zero length. The most fundamental issue is the interference of the VLA concept with `sizeof` being a compile-time expression. VLA support adds enormous additional complexity to the compiler and runtime.

Run the following program once trying different values for the length until you crash your program, and then never do it again in your code:

```
#include <iostream>

int main(){
    for(;;){
        std::cout << "Array length = ";
        int length;
        std::cin >> length;
        int abomination[ length ]; // VLA -- never again!
        // if the array is not used at all,
        // the compiler may recognize that
        // and stop short of actually allocating it,
        // thus creating the illusion of working code ---
        // so, let's use it:
        abomination[ 0 ] = 1;
    }
    return 0;
}
```

To ensure you did not accidentally stumble into creating a VLA, you can compile your code with `-Werror=vla` flag, e. g. `g++ -Wall -Werror=vla main.cpp -o main`

4.2.5. Pointer Arithmetic

We have already seen the subscript operator `[]` when it was applied to an array:

```
int arr[4]; // length declaration, not subscript operator
arr[0] = 7; // subscript operator
```

It turns out that the only thing needed for its use is a typed pointer:

```
int * arr_ptr = arr;     // array decays to a pointer arr_ptr
arr_ptr[0];              // still 7
```

Even more interesting is the possibility of adding or subtracting integer numbers to and from typed pointers, without any array present:

```
int i;
int * p = &i;
int * q = p + 3;
```

The value of a typed pointer can be seen as an interval in computer memory. The starting point (i. e the block with the lowest address) of that interval is the address of the pointer. The length of the interval is determined by the type of the pointer's target. Adding an integer n to a typed pointer shifts the original interval forward by n intervals of the same length, and subtracting an integer does the same but backward:

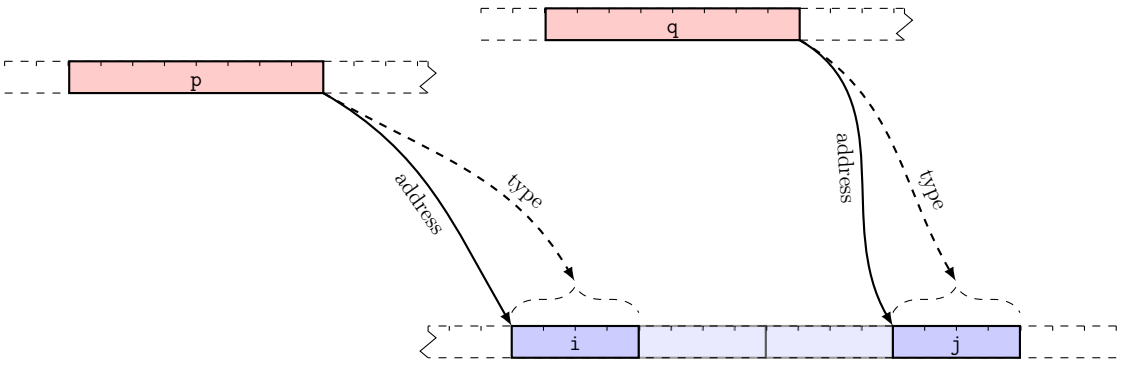

Figure 18. Pointer `q` is `p + 3`.

The array subscript operator turns out to be an exact equivalent of point increment and dereference:

```
int arr[4];
arr[ 2 ];        // member with index 2
*( arr + 2 );    // same thing
```

We need parentheses around pointer and integer sum because — as we know from the Order of Evaluation section — unary operations have higher priority than binary, so without parentheses we would get an equivalent of `(*arr) + 2` which would be just an addition of two integers.

Equality of `a[b]` and `*(a + b)` is preserved even when we switch the order of addition, so that

```
arr[2] = *(arr + 2) = *(2 + arr) = 2[arr]
```

(The above are mathematical equalities, not assignment operators.) The last expression is one of those things which, even though possible, should never be used.

Unless you want to deal with undefined behavior, these operation should be used when it is known that the memory for the target of the resulting pointer has been allocated. In practice it means that the pointer arithmetic should be confined to a contiguous block of memory allocated for an array.

Pointer arithmetic may clarify why array indexing starts at 0. Indeed, the array index is merely the *offset* from the starting member.

Besides combining integers with pointers, one can also *subtract* pointers from each other. Subtracted pointers must have the same type. The result of subtraction is an integer. Subtraction of pointers `q - p` returns the length of the shift from `*p` to `*q`, measured in objects of the type that these two pointers share. This operation is the inverse of pointer and integer addition. Indeed, if we go back to the old picture, we will see that the difference `q - p` in our old example is 3:

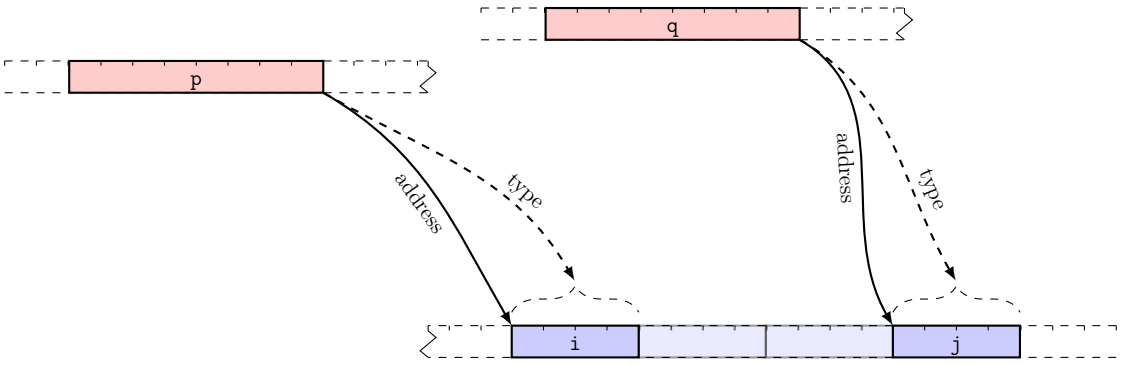

Figure 19. Result of pointer subtraction `q - p` is 3.

Like before, pointer subtraction is meaningful only for pointers of objects that reside within the same array.

If you studied Linear Algebra, you might have noticed that pointers and integers resemble points of affine space and vectors. Points can be subtracted, resulting in a vector; vectors can be added to points, yielding new points. In that sense, an array is like an affine space and the whole pointer arithmetic looks like a bit of linear algebra.

4.2.6. Multi-Dimensional Arrays

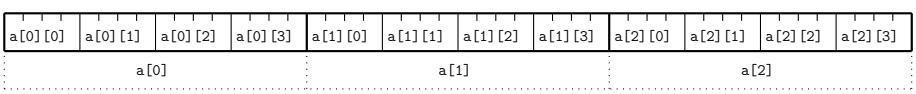

Figure 20. Array `int a[3][4]` in memory.

4.2.7. 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:

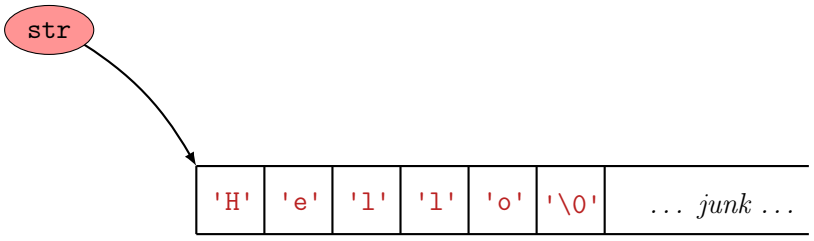

Figure 21. String `"Hello"` in memory, as a `char` array

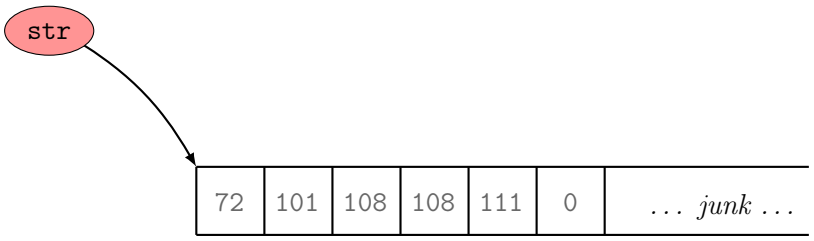

Figure 22. String `"Hello"` in memory, as an ASCII codes `int` array.

String Literals

String literals are C-strings located in the read-only portion of the static memory. Being C-strings, they are terminated by the null character `'\0'`. In addition, they don't have any trailing junk afterwards, so that their length (as that of an array) is their character length (not including the `'\0'`) plus one[66]:

```
sizeof( "Hello" ); // is 6
```

String literals are an exception among the literals of other types: string literals are *addressable*, while other type literals are not:

```
&"Hello";    // OK
&5;          // error: lvalue required as unary ‘&’ operand
```

The address of a string literal in static memory can be stored in a variable as a mutable pointer to an immutable `char` at the start of the literal array:

```
const char * ptr = "Hello"; // *ptr is the const 'H'
```

This results in array-to-pointer decay, so that in the above example, the *array* `"Hello"` decays to the *pointer* `ptr`.

When the same string literal is used repeatedly in the same program, the language standard permits (but does not require) the compiler to reuse the same

[66] Remember that `sizeof(char)` is always one, so to find the length of a string `str` as an array it is enough to compute `sizeof( str )` instead of `sizeof( str )/sizeof( str[0] )`.

location in memory for it. This reuse is called ***duplicate string merging*** or ***string literals pooling***. It is easy to check if your compiler is doing it[67]:

```
#include <iostream>
int main()
{
    const char * str1_ptr = "Hello";
    const char * str2_ptr = "Hello"; // use same literal
    std::string conclusion = ( str1_ptr == str2_ptr ?
        "is" : "is not" );
    std::cout
        << "Compiler " << conclusion
        << " pooling string literals.\n"
        << "The addresses of two duplicates are:\n\t"
        << static_cast<const void*>( str1_ptr ) << "\n\t"
        << static_cast<const void*>( str2_ptr ) << "\n";
}
```

A string literal can also be used for array initialization (which is one of the few cases[68] when an array does not decay into a pointer):

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

In such array initialization, the original string literal is *copied* into its new location, and the array being initialized is automatically allocated with the length equal to the string length in characters plus one extra space for the null terminator, so that the following code:

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

outputs `6` on the console. Note also that thus initiated array `str` may or may not be constant, unlike the string literal pointer `ptr` in the earlier example (which always must be a pointer to `const char`).

Can you guess the output of the following code? Check:

[67] While most pointer types, such as `int *`, will print as *hexadecimal memory addresses*, the stream insertion `operator<<` in `std::ostream` (used by `std::cout`) for `char *` specifically is overloaded to output the *character sequence* from the address in question until the null terminator `'\0'`. For that reason, we need to cast the pointer to `void`.

[68] The other two cases are the `sizeof( some_array )`, which returns the size of the array itself rather than that of the pointer to its start, and `&some_array`, which is the *pointer to the array*, rather than pointer to pointer, so that for `int some_array[ 5 ]` we have `int (*ptr_to_array)[ 5 ] = &some_array;`

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

If we go over the array size:

```
#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.

4.2.8. 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* `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[69] 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.

4.2.9. 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:

[69] 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.

```
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 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. For that reason, the length of the array `argv` is one more than the `argc` (or, to put it differently, `argc` is the index of the `NULL` pointer in `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:

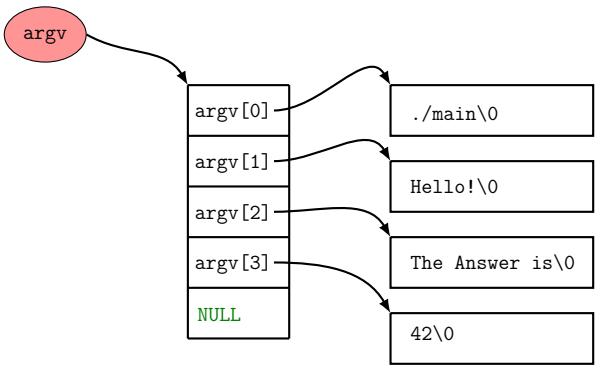

Figure 23. The structure of command line arguments for `./main Hello! "The Answer is" 42`

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
```

4.3. Dynamic Memory Allocation

One of the main uses of pointers is in dynamic memory allocation. While the memory of the von Neumann machine itself is linear and homogeneous, a program running on the top of an operating system gets a much more elaborate setup to work with. The OS allocates what is called a virtual memory to each process it runs. The virtual memory of a C++ program has several parts classified based on the lifetime and management of objects stored there:

- ***static*** (objects are loaded when program starts and stay resident in memory for the whole duration of program's execution);
- ***automatic*** (objects are allocated and — when no longer in use — discarded automatically by the system);
- ***dynamic*** (objects are allocated and deallocated by explicit instructions in code).

The actual implementation of this memory management is platform-specific and the following terminology is, formally speaking, not a part of the C++ proper. Usually the compiled executable contains different *sections* and when that executable is loaded by the system loader, those sections are mapped into the static memory, giving the latter the same overall layout. Thus you may see the following terms used for both: the various sections of an executable, and for the corresponding *segments* of the virtual memory of the process running the executable:

- ***.text*** — the compiled executable code from all object files of the project; read/execute permissions;
- ***.rodata*** — may be included as a separate section holding global `const` variables, lookup tables, perhaps string literals, sometimes `constexpr` that must be stored in memory at all, like variables whose addresses are used, or variables holding complex objects, and `static constexpr` members of a class (often merged with the .text); has read/execute permissions;
- ***.data*** — initialized global and local `static` variables together with their initial values, also `constinit` expressions; has read/write permissions;
- ***.bss*** — may be included to save space; holds *uninitialized* global and static variables[70]; has read/write permissions;

[70]The space saving is in reduction of *executable's* file size. The .bss section of executable does not need any space for holding the values of uninitialized variables. When the system's loader expands this section from executable to RAM, it does allocates space for all the values (and zeroes that space out), so there is no space saving for the virtual memory segment.

Strictly speaking, there is a difference between sections and segments. A single segment is comprised of several of those sections that share the same permissions. So, the .text and .rodata are in the *Text Segment*, while .data and .bss are in the *Data Segment*.

Most platforms implement automatic memory via ***stack*** and dynamic — via ***heap***, so that a typical layout of program's virtual memory may look like this:

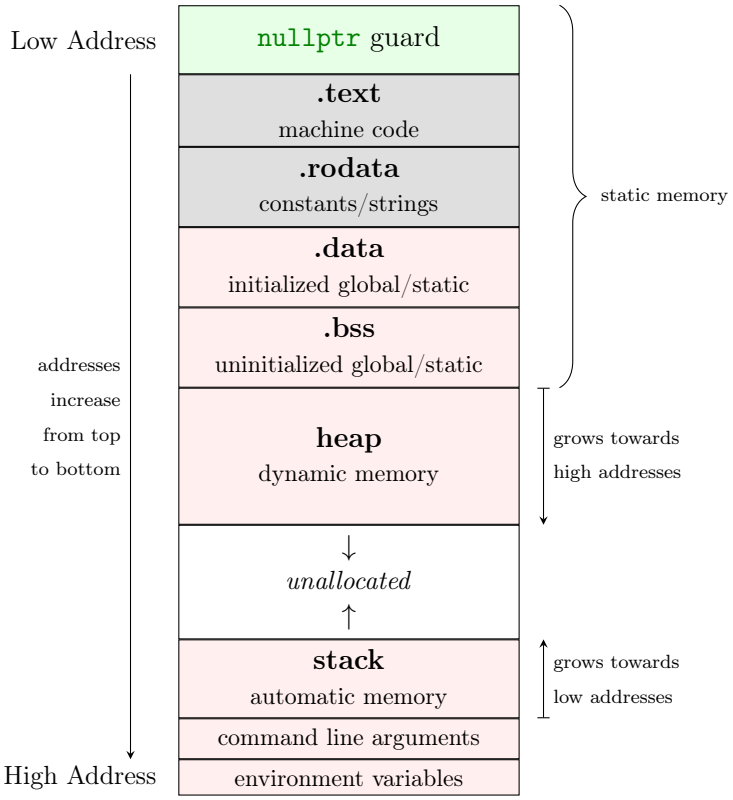

Figure 24. Typical memory layout of a C++ program.

Here is a program illustrating where various things may land in the memory layout depicted above:

```
#include <iostream> // for cerr, cout
#include <cstdint> // for uintptr_t
#include <cstdlib> // for getenv


int initialized_global = 10;         // will go to .data
int uninitialized_global;            // will go to .bss
const int global_const = 100;        // will go to .rodata
const char * string_ptr = "Hello";   // will go to .rodata
```

```
// we will output the results in CSV format:
void header_row() // will go to .text
{
    std::cout
        // the address:
        << "Address,"

        // the value (perhaps decayed) at that address:
        <<  "Value at Address,"

        // description of that value
        << "Description of Value,"

        // where (we expect) it will end up
        << "Expected Location\n";
}


template<typename L, typename D, typename T> // will go where?
void report_row(
    L given_location,
    D given_description,
    T * given_ptr
){
    std::cout

        // the address:
        << reinterpret_cast<uintptr_t>( given_ptr ) << ","

        // the value (perhaps decayed) at that address:
        <<  *given_ptr << ","

        // description of that value
        << given_description << ","

        // where (we expect) it will end up
        << given_location << "\n";
}
```

```
const char * get_user() // will go to .text, like any function
{
    const char * user;
    // Unix:
    if( ( user = std::getenv( "USER" ) ) ) return user;
    // Windows:
    if( ( user = std::getenv( "USERNAME" ) ) ) return user;
    // Giving up (this will mess up this program)
    return "user";
}
void check_and_report_user()
{
    const char * user_ptr = get_user(); // assuming it is there
    report_row(
        "env",
        "(first char of) an environment variable",
        user_ptr
    );
}
void check_and_report_first_argv( int argc, char ** argv )
{
    if( argc > 1 ){
        report_row( "argv",
            "(first char of) a command line argument",
            argv[1] );
    }
}
void stack_allocate()
{
    int local_int = 1000;  // in stack frame
    int * local_int_ptr = &local_int; // address in the stack

    // local_int_ptr == &local_int
    // is equivalent to
    // *local_int_ptr == local_int

    report_row( std::string( "in stack frame of '" )
        + __func__ + "'",
        "local variable", local_int_ptr );
}
```

```
void heap_allocate()
{
    // 'int{5}' is the 'uniform', a.k.a. 'brace' initialization
    int * ptr = new int{5}; // 5 will go to the heap
    report_row( "on the heap",
        "dynamically allocated variable", ptr );
    delete ptr;     // free memory on the heap
    ptr = nullptr;  // just in case ptr is used below, so that
    //*ptr = 5;     // would hit the nullptr guard, causing a
                    // segmentation fault,
                    // which is what we want!
}
int main( int argc, char ** argv )
{
    header_row();
    report_row( ".text", "function (may decay to bool)",
        &header_row );
    report_row( ".rodata", "global constant",
        &global_const );
    report_row( ".rodata", "(first char of) a string literal",
        string_ptr );
    report_row( ".data", "initialized global variable",
        &initialized_global );
    report_row( ".bss", "uninitialized global variable",
        &uninitialized_global );
    heap_allocate();
    stack_allocate();
    check_and_report_first_argv( argc, argv );
    check_and_report_user();
}
```

HOMEWORK: What section of the memory layout will the template in the above code go?

Check: 

You can run the above program[71] with the following command to specify the environment variable `USER` explicitly and to provide a command line argument. (I am also including below the output I got on my machine. Specific addresses in that output will change between different program runs.)

[71] Assuming it was saved as file `main.cpp` and compiled with `g++ -Wall main.cpp -o main`.

```
user@computer:~$ USER=mickey && ./main hi
Address,Value at Address,Description of Value,Expected Location
94177725113017,1,function (may decay to bool),.text
94177725116424,100,global constant,.rodata
94177725116428,H,(first char of) a string literal,.rodata
94177725124824,10,initialized global variable,.data
94177725125140,0,uninitialized global variable,.bss
94177867781824,5,dynamically allocated variable,on the heap
140736195288572,1000,local variable,in stack frame of
↪  'stack_allocate'
140736195294475,h,(first char of) a command line argument,argv
140736195297036,m,(first char of) an environment variable,env
```

This output is in the comma separated values (CSV) format. It can be saved to a CSV file `memory-test.csv` with:

```
user@computer:~$ USER=mickey && ./main hi > memory-test.csv
```

Afterwards, you will be able to open the file `memory-test.csv` in Microsoft Excel or LibreOffice Calc and see something like this:

Address	Value at Address	Description of Value	Expected Location
94177725113017	1	function (may decay to bool)	.text
94177725116424	100	global constant	.rodata
94177725116428	H	(first char of) a string literal	.rodata
94177725124824	10	initialized global variable	.data
94177725125140	0	uninitialized global variable	.bss
94177867781824	5	dynamically allocated variable	on the heap
140736195288572	1000	local variable	in stack frame of 'stack_allocate'
140736195294475	h	(first char of) a command line argument	argv
140736195297036	m	(first char of) an environment variable	env

Table 4.1. Resulting CSV file `memory-test.csv`.

To examine the *order* of the addresses, you can use the Bash `sort` command:

```
user@computer:~$ USER=mickey && ./main hi | sort -t',' -k1,1 -n
```

If all objects end up where we expect, the sorted output will be exactly the same as the unsorted one.

The above command is an example of "piping" the output of one command into the next one. For that reason, the `'|'` is often called the ***pipe symbol*** in Bash. In the above, everything produced by '`USER=mickey && ./main hi`' is fed into `sort`. The '`sort`' command breaks its input into fields along the comma '`,`' delimiter (the '`-t','`' flag), selects the first field (the '`-k1,1`' flag), and sorts *numerically* (the '`-n`' flag) all the lines based on their first field.

Chapter 5

Objects-Oriented Programming

Chapter 6

Project Management

6.1. Build Process

6.1.1. Build Steps

6.1.2. Directory Structure

6.1.3. Automating Build with GNU `Make`

6.2. Libraries

6.2.1. Linking External Libraries

6.2.2. Building Libraries

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