A Tour Of C++

Bjarne Stroustrup

March 12, 2022

Contents

1 The Basics

1.1 Introduction

The operator << (“put to”) writes its second argument onto its first

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call.

Defining multiple functions with the same name is known as function **overloading** and is one of the essential parts of generic programming

1.2 Types, Variables and Arithemtic

A **declaration** is a statement that introduces an entity into the program. It specifies a type for the entity:

- A **type** defines a set of possible values and a set of operations (for an object)
- An **object** is some memory that holds a value of some type.
- A **value** is a set of bits interpreted according to a type.
- A **variable** is a named object.

Unfortunately, conversions that lose information, **narrowing conversions**, such as `double` to `int` and `int` to `char`, are allowed and implicitly applied when you use `=` (but not when you use `{}`)

When defining a variable, you don't need to state its type explicitly when it can be deduced from the initializer:

With `auto`, we tend to use the `=` because there is no potentially troublesome type conversion involved, but if you prefer to use `{}` initialization consistently, you can do that instead.

1.3 Scope and Lifetime

- **Local scope**: A name declared in a function or lambda is called a local name. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A **block** is delimited by a `{ }` pair. Function argument names are considered local names.
- **Class scope**: A name is called a **member name** (or a **class member name**) if it is defined in a class , outside any function , lambda, or enum class. Its scope extends from the opening `{` of its enclosing declaration to the end of that declaration.
- **Namespace scope**: A name is called a **namespace member name** if it is defined in a namespace outside any function, lambda, class, or enum class. Its scope extends from the point of declaration to the end of its namespace.

1.4 Constants

C++ supports two notions of immutability:

- `const`: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces so that data can be passed to functions using pointers and references without fear of it being modified. The compiler enforces the promise made by `const`. The value of a `const` can be calculated at run time.

- `constexpr`: meaning roughly "to be evaluated at compile time." This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance. The value of a `constexpr` must be calculated by the compiler.

For example

For a function to be usable in a **constant expression**, that is, in an expression that will be evaluated by the compiler, it must be defined `constexpr`. For example:

A `constexpr` function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a `constexpr` function to be called with non-constant-expression arguments in contexts that do not require constant expressions. That way, we don't have to define essentially the same function twice: once for constant expressions and once for variables.

To be `constexpr`, a function must be rather simple and cannot have side effects and can only use information passed to it as arguments. In particular, it cannot modify non-local variables, but it can have loops and use its own local variables. For example:

```
constexpr double nth(double x, int n) // assume 0<=n {
{
    double res = 1;
    int i = 0;
    while (i<n) {
        res*=x;
        ++i;
    }
    return res;
}
```

1.5 Pointers, Arrays, and References

```
char* p = &v[3];
char x = *p;
```

in an expression, prefix unary * means "contents of" and prefix unary & means "address of"

If we didn't want to copy the values from v into the variable x, but rather just have x refer to an element, we could write:

```
void increment() {
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto& x : v) // add 1 to each x in v
        ++x;
    // ...
}
```

In a declaration, the unary suffix & means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization.

References are particularly useful for specifying function arguments. For example: By using a reference, we ensure that for a call `sort(vec)`, we do not copy `vec` and that it really is `vec` that is sorted and not a copy of it.

When used in declarations, operators (such as &, *, and []) are called declarator operators:

We try to ensure that a pointer always points to an object so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g., for an end of a list), we give the pointer the value `nullptr` ("the null pointer"). There is only one `nullptr` shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr; // pointer to a Link to a Record
int x = nullptr; // error: nullptr is a pointer not an integer
```

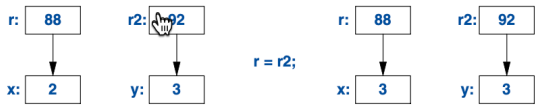
1.6 Tests

1.7 Mapping to Hardware

An assignment of a built-in type is a simple machine copy operation.

A reference and a pointer both refer/point to an object and both are represented in memory as a machine address. However, the language rules for using them differ. Assignment to a reference does not change what the reference refers to but assigns to the referenced object:

```
int x = 2;
int y = 3;
int& r = x; // r refers to x
int& r2 = y; // now r2 refers to y
r = r2; // read through r2, write through r: x becomes 3
```



2 User-Defined Types

2.1 Introduction

Types built out of other types using C++'s abstraction mechanisms are called **user-defined types**. They are referred to as **classes** and **enumerations**.

2.2 Structures

The `new` operator allocates memory from an area called the **free store** (also known as **dynamic memory** and **heap**). Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using the `delete` operator