Poznan University of Technology
C++ basics

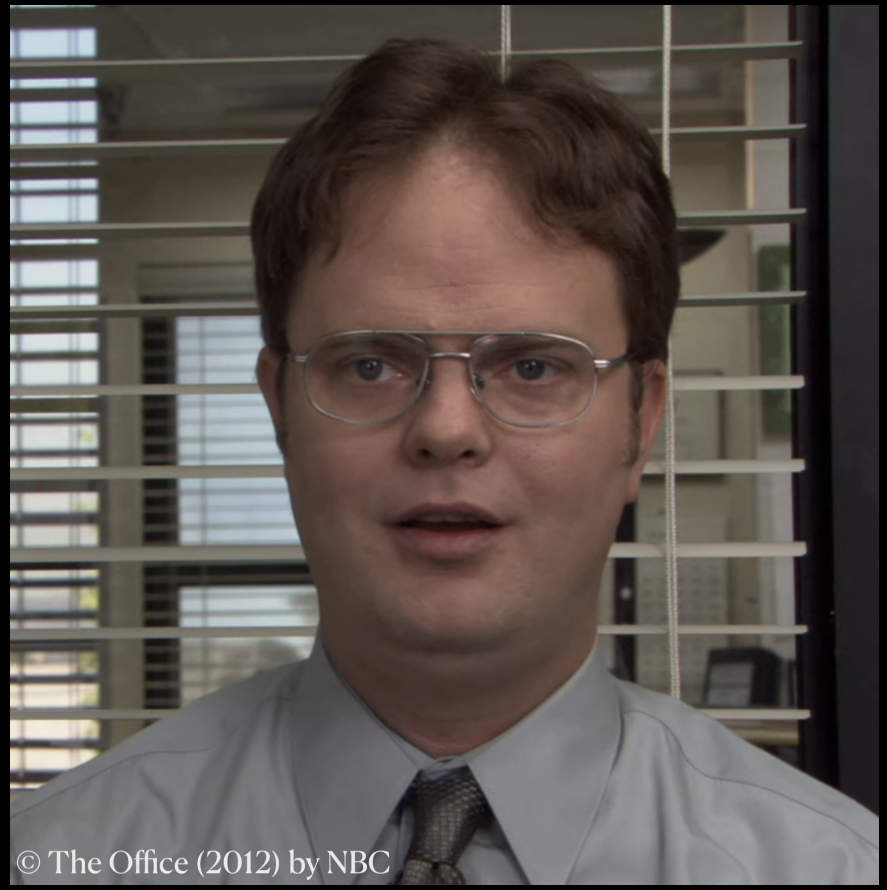
Jakub Piotr Hamerliński

C++ basics

Agenda

01. Type system
02. Scope
03. `main` function
04. Modern C++ elements

> *"Whenever I'm about to do something, I think, 'Would an idiot do that?' And if they would, I do not do that thing."*
> Dwight Schrute



Type system

C++ basics

Type system

To be compiled, each variable, function argument, and function return value has to have a type. Additionally, the compiler implicitly assigns a type to every expression (including literal values) before it evaluates them.

Built-in types like `int` for storing integer data, `double` for storing floating-point values, and types from the Standard Library such class `std::basic_string` for storing text are a few examples of kinds. By establishing a `class`, you may develop a custom type. The type defines how much memory is set aside for the variable (or expression result). In addition, the type defines the types of data that may be stored, how the compiler interprets the bit patterns in those values, and the operations that can be applied to them.

C++ basics

Type system - terminology

A type known as a **scalar** stores a single number from a predetermined range. Arithmetic types (integral or floating-point values), members of enumeration type, pointer types, pointer-to-member types, and `std::nullptr_t` are examples of scalars. Scalar types tend to be fundamental kinds.

A type that isn't a scalar type is a **compound** type. Compound types include reference types, pointers to non-static class members, class (or struct) types, union types, enumerations, and type arrays.

A data quantity's symbolic name is called a **variable**. Within the confines of the code where it is declared, the name can be used to retrieve the data it refers to. Examples of scalar data types are frequently referred to in C++ as variables, but instances of other kinds are frequently referred to as objects.

C++ basics

Type system

Every object in C++ has a type, and that type never changes since it is both a strongly typed and a statically typed language. You must either explicitly define a variable's type when you declare it in your code, or you may use the `auto` keyword to tell the compiler to infer the type from the initializer.

The type of each parameter and return value must be specified when a function is declared in your code. If the function returns nothing, use the return value type `void`.

C++ basics

Type system - variable declaration and assignment

A variable's type cannot be changed later on after it has been declared. The value of the variable or the return value of a function can be copied into a different type of variable, though. These activities, known as type conversions, are occasionally required but can potentially lead to data loss or inaccuracy.

C++ basics

Type system - examples

```
int result = 0;              // Declare and initialize an integer.
double coefficient = 10.8;   // Declare and initialize a floating
                             // point value.
auto name = "Lady G.";       // Declare a variable and let compiler
                             // deduce the type.
auto address;                // Error! Compiler cannot deduce a type
                             // without an intializing value.
age = 12;                    // error. Variable declaration must
                             // specify a type or use auto!
result = "Kenny G.";         // error. Can't assign text to an int.
string result = "zero";      // error. Can't redefine a variable with
                             // new type.
int maxValue;                // Not recommended! maxValue contains
                             // garbage bits until it is initialized.
```

C++ basics

Type system - fundamental (built-in) types

There isn't a single basic type in C++ from which all other types may be derived. The built-in types, also known as basic types, are numerous in the language. These types include the ASCII and UNICODE character types `char` and `wchar_t`, as well as mathematical types like `int`, `double`, `long`, and `bool`. The majority of integral basic types have unsigned counterparts, which alter the range of values that the variable may hold (except for `bool`, `double`, `wchar_t`, and similar types). The range of possible values for an `int`, which stores a 32-bit signed integer, is -2,147,483,648 to 2,147,483,647. The range of possible values for an `unsigned int`, which is likewise stored as 32 bits, is 0 to 4,294,967,295. The total number of possible values in each case is the same; only the range is different.

C++ basics

Type system - void

The `void` type is a peculiar type; you cannot declare a variable of the `void` type, but you can define a variable of the `void *` (reference to `void`) type, which is occasionally required when allocating raw (untyped) memory. Pointers to `void` are not type-safe, hence current C++ discourages the use of them. A `void` return value in a function declaration indicates that the function does not return a value; this is a typical and permitted use of the `void` type.

C++ basics

Type system - pointers

As in the earliest versions of the C language, C++ continues to let you declare a variable of a pointer type by using the special declarator `*` (asterisk). A pointer type stores the address of the location in memory where the actual data value is stored. In modern C++, these pointer types are referred to as raw pointers, and they're accessed in your code through special operators: `*` (asterisk) or `->` (dash with greater-than, often called arrow). This memory access operation is called dereferencing. Which operator you use depends on whether you're dereferencing a pointer to a scalar, or a pointer to a member in an object.

C++ basics

Type system - pointers

Working with pointer types has long been one of the most challenging and confusing aspects of C and C++ program development. This section outlines some facts and practices to help use raw pointers if you want to. However, in modern C++, it's no longer required (or recommended) to use raw pointers for object ownership at all, due to the evolution of the smart pointer (discussed more at the end of this section). It's still useful and safe to use raw pointers for observing objects. However, if you must use them for object ownership, you should do so with caution and with careful consideration of how the objects owned by them are created and destroyed.

C++ basics

Type system - pointers

The first thing that you should know is that a raw pointer variable declaration only allocates enough memory to store an address: the memory location that the pointer refers to when it's dereferenced. The pointer declaration doesn't allocate the memory needed to store the data value. (That memory is also called the backing store.) In other words, by declaring a raw pointer variable, you're creating a memory address variable, not an actual data variable. If you dereference a pointer variable before you've made sure that it contains a valid address for a backing store, it causes undefined behavior (usually a fatal error) in your program. The following example demonstrates this kind of error:

```
int* pNumber;       // Declare a pointer-to-int variable.
*pNumber = 10;      // error. Although this may compile, it is
                    // a serious error. We are dereferencing an
                    // uninitialized pointer variable with no
                    // allocated memory to point to.
```

C++ basics

Type system - pointers

The example dereferences a pointer type without having any memory allocated to store the actual integer data or a valid memory address assigned to it. The following code corrects these errors:

```
int number = 10;          // Declare and initialize a local integer
                          // variable for data backing store.
int* pNumber = &number;   // Declare and initialize a local integer
                          // pointer variable to a valid memory
                          // address to that backing store.
//(...)
*pNumber = 41;            // Dereference and store a new value in
                          // the memory pointed to by
                          // pNumber, the integer variable called
                          // "number". Note "number" was changed, not
                          // "pNumber".
```

C++ basics

Type system - pointers

The corrected code example uses local stack memory to create the backing store that `pNumber` points to. We use a fundamental type for simplicity. In practice, the backing stores for pointers are most often user-defined types that are dynamically allocated in an area of memory called the heap (or free store) by using a `new` keyword expression. Once allocated, these variables are normally referred to as objects, especially if they're based on a class definition. Memory that is allocated with `new` must be deleted by a corresponding `delete` statement.

C++ basics

Type system - pointers

However, it's easy to forget to delete a dynamically allocated object - especially in complex code, which causes a resource bug called a memory leak. For this reason, the use of raw pointers is discouraged in modern C++. It's almost always better to wrap a raw pointer in a smart pointer, which automatically releases the memory when its destructor is invoked. (That is, when the code goes out of scope for the smart pointer.) By using smart pointers, you virtually eliminate a whole class of bugs in your C++ programs. In the following example, assume `MyClass` is a user-defined type that has a public method `DoSomeWork();`

```
void someFunction() {
  unique_ptr<MyClass> pMc(new MyClass);
  pMc->DoSomeWork();
}
// No memory leak. Out-of-scope automatically calls the destructor
// for the unique_ptr, freeing the resource.
```

Scope

C++ basics

Scope

A program element, such as a class, function, or variable, must be declared before it may be used elsewhere in the program. The term "scope" refers to the environment in which a name is visible. For instance, if a variable named `x` is declared inside a function, only that function's body can see it. It is limited locally. Other variables with the same name may exist in your application, provided that they are in distinct scopes and do not violate the one definition rule. In this case, there won't be an issue.

C++ basics

Scope

There are six kinds of scope:

01. **Global scope**: Any name that is declared outside a class, function, or namespace is referred to as a global name. Even these identifiers, though, are present in C++ with an implied global namespace. Global names have a range that starts with their declaration and goes all the way to the end of the file where they are defined.
02. **Namespace scope**: A name that is declared outside a class, enum definition, or function block within a namespace is visible from the moment of declaration all the way to the namespace's end. A namespace can be defined in a number of blocks spread over various files.

C++ basics

Scope

03. **Local scope**: refers to a name's ability to be used within of a function or lambda, including its parameter names. They are frequently called "locals." Only the portion of the function or lambda body after its declaration is displayed.
04. **Class scope**: regardless of the point of declaration, class names have class scope, which encompasses the whole class definition. The `public`, `private`, and `protected` keywords also limit class members' accessibility.
05. **Statement scope**: names declared in a for, if, while, or switch statement are visible until the end of the statement block.
06. **Function scope**: a label has function scope, which means it is visible throughout a function body even before its point of declaration.

`main` function

C++ basics

`main` function

A main function is a requirement for all C++ programs. The compiler reports an error if you attempt to compile a C++ program without a main function. (Static libraries and dynamic-link libraries lack a main function.) Your source code runs in the main function, but before it does, all static class members without explicit initializers are initialized to o. Before entering main in Microsoft C++, global static objects are also initialized. The main function is subject to a number of limitations that do not apply to any other C++ functions.

C++ basics

`main` function - signature

The `main` function doesn't have a declaration, because it's built into the language. If it did, the declaration syntax for `main` would look like this:

```
int main();
int main(int argc, char *argv[]);
```

If no return value is specified in main, the compiler supplies a return value of zero.

C++ basics

`main` function - standard command-line arguments

The arguments for `main` allow convenient command-line parsing of arguments. The types for `argc` and `argv` are defined by the language. The names `argc` and `argv` are traditional, but you can name them whatever you like. The argument definitions are as follows:

01. `argc`: An integer that contains the count of arguments that follow in argv. The argc parameter is always greater than or equal to 1.

02. `argv`: An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, `argv[0]` is the command with which the program is invoked. `argv[1]` is the first command-line argument. The last argument from the command line is `argv[argc - 1]`, and `argv[argc]` is always NULL.

C++ basics

`main` function - standard command-line arguments example

```
#include <iostream>
#include <string.h>

using namespace std;
int main( int argc, char *argv[], char *envp[] ) {
  bool numberLines = false;    // Default is no line numbers.
  // If /n is passed to the .exe, display numbered listing
  // of environment variables.
  if ( (argc == 2) && _stricmp( argv[1], "/n" ) == 0 )
    numberLines = true;
  // Walk through list of strings until a NULL is encountered.
  for ( int i = 0; envp[i] != NULL; ++i ) {
    if ( numberLines )
      cout << i << ": "; // Prefix with numbers if /n specified
    cout << envp[i] << "\n"; // The optional envp parameter is
    // an array of strings representing the variables set in
    // the user's environment. This array is terminated by
    // a NULL entry.
  }
}
```

Modern C++

C++ basics

Modern C++

Since its inception, C++ has grown to rank among the top programming languages used globally. Programs created in C++ are quick and effective. It can function at the highest levels of abstraction as well as at the level of the silicon, making it more adaptable than other languages. Highly optimized standard libraries are provided by C++. It makes it possible to access low-level hardware functions, maximizing speed and reducing memory needs. Games, device drivers, HPC, cloud, desktop, embedded, and mobile apps, among many other types of programs, can all be written in C++. C++ is used to create libraries and compilers for other programming languages.

C++ basics

Modern C++

Backward compatibility with the C language was one of the original specifications for C++. As a result, C++ has always supported C-style programming, including capabilities like raw pointers, arrays, and null-terminated character strings. They might boost performance, but they might also breed complexity and bugs. The emphasis on features in C++'s growth has significantly decreased the necessity for C-style idioms. When you need them, the outdated C-programming resources are still available. However, you should use them less and less in contemporary C++ programs. Modern C++ code is more elegant, safer, and faster while maintaining the same speed.

C++ basics

Modern C++ - Resources and smart pointers

Memory leaks are one of the main categories of errors in C-style programming. Failure to call `delete` for memory that was allocated with `new` frequently results in leaks. The notion of *resource acquisition and initialization* is stressed in modern C++ (RAII). The concept is basic. An object should possess resources (such as heap memory, file handles, sockets, and so on). The freshly allocated resource is created or received by that object in its constructor, and it is destroyed in its destructor. When the owning object exits its scope, the RAII principle ensures that all resources are correctly returned to the operating system.

C++ basics

Modern C++ - Resources and smart pointers

To support easy adoption of RAII principles, the C++ Standard Library provides three smart pointer types: `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. A smart pointer handles the allocation and deletion of the memory it owns. The following example shows a class with an array member that is allocated on the heap in the call to `make_unique()`. The calls to `new` and `delete` are encapsulated by the `unique_ptr` class. When a widget object goes out of scope, the `unique_ptr` destructor will be invoked, and it will release the memory that was allocated for the array.

C++ basics

Modern C++ - Resources and smart pointers example

```cpp
#include <memory>
class widget
{
private:
  std::unique_ptr<int[]> data;
public:
  widget(const int size) { data = std::make_unique<int[]>(size); }
  void do_something() {}
};

void functionUsingWidget() {
  widget w(1000000);  // lifetime automatically tied to enclosing scope
                      // constructs w, including the w.data gadget member
  // ...
  w.do_something();
  // ...
} // automatic destruction and deallocation for w and w.data
```

Whenever possible, use a smart pointer to manage heap memory. If you must use the new and delete operators explicitly, follow the principle of RAII.

C++ basics

Modern C++ - `std::string` and `std::string_view`

Another significant source of issues is C-style strings. Almost all issues related to C-style strings can be avoided by using std::string and std::wstring. The advantages of member function for searching, appending, prepending, and other operations are also yours. Both have advanced speed optimization. In C++17, you can use std::string view to provide strings to functions that only need read-only access for an even bigger speed boost.

C++ basics

Modern C++ - `std::vector` and other Standard Library containers

The standard library containers all follow the principle of RAII. They provide iterators for safe traversal of elements. And, they're highly optimized for performance and have been thoroughly tested for correctness. By using these containers, you eliminate the potential for bugs or inefficiencies that might be introduced in custom data structures. Instead of raw arrays, use vector as a sequential container in C++.

```
vector<string> apples;
apples.push_back("Granny Smith");
```

C++ basics

Modern C++ - `std::vector` and other Standard Library containers

Use map (not unordered_map) as the default associative container. Use set, multimap, and multiset for degenerate and multi cases.

```
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

When performance optimization is needed, consider using: 1. The array type when embedding is important, for example, as a class member. 2. Unordered associative containers such as unordered_map. These have lower per-element overhead and constant-time lookup, but they can be harder to use correctly and efficiently. 3. Sorted `vector`.

Don't use C-style arrays. For older APIs that need direct access to the data, use accessor methods such as `f(vec.data(), vec.size());` instead.

C++ basics

Modern C++ - Exceptions

Modern C++ emphasizes exceptions, not error codes, as the best way to report and handle error conditions.

01. `std::atomic`: Use the C++ Standard Library std::atomic struct and related types for inter-thread communication mechanisms.
02. `std::variant` (C++17): Unions are commonly used in C-style programming to conserve memory by enabling members of different types to occupy the same memory location. However, unions aren't type-safe and are prone to programming errors. C++17 introduces the `std::variant` class as a more robust and safe alternative to unions. The `std::visit` function can be used to access the members of a `variant` type in a type-safe manner.

C++ basics

Sources

01. C++ logo created by Jeremy Kratz
02. C++ type system
03. C++ scope
04. C++ main and CLI args
05. Modern C++

Thank you

Feel free to reach me via LinkedIn

Fin