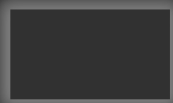
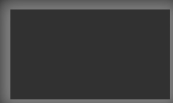


# Types & Variables

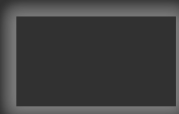


# Data Types



# Data Types

- One of the fundamental strengths of C++ is the way the language deals with data and the associated data types
- On a design level, data types can be considered as representations of a **concept** with a certain kind of **feature set**
- On a hardware level, the data type determines the **piece of memory** that is allocated & reserved during compilation



# Statically Typed

- C++ is a **statically typed language**
- Data types have to be defined (or deduced) at compile time
- Data types have to be associated with variables
- The type of the variable cannot change

# Statically Typed

- Type checking is executed at **compile time**
- To identify type errors early in the development cycle
- To allocate the required memory at compile time
- To ensure faster program execution

# Statically vs Dynamically Typed

- In contrast, **dynamically typed languages** do not associate data types with variables — data types are dynamically defined and checked **at run time** and variable type associations can change
- This is another reason for why interpreted languages are usually slower in execution than statically typed languages

Examples: python, javascript

# Supported Types

- C++ supports **built-in** data types (float, int, bool, ...) and **user-defined** data types (enum, struct, and classes)

# Supported Types

C++ has a set of fundamental types corresponding to the most common basic storage units of a computer and the most common ways of using them to hold data:

§4.2 A Boolean type (*bool*)

§4.3 Character types (such as *char*)

§4.4 Integer types (such as *int*)

§4.5 Floating-point types (such as *double*)

In addition, a user can define

§4.8 Enumeration types for representing specific sets of values (*enum*)

There also is

§4.7 A type, *void*, used to signify the absence of information

From these types, we can construct other types:

§5.1 Pointer types (such as *int\**)

§5.2 Array types (such as *char[ ]*)

§5.5 Reference types (such as *double&*)

§5.7 Data structures and classes (Chapter 10)

... represent mathematical concepts like numbers, logical operations and language / text concepts.

... represent user-defined sets & the concept of nothingness, i.e., mathematical concept of zero.

... represent concepts of memory allocation, memory access & aliasing as well as fully user-defined concepts.

Source credit: Bjarne Stroustrup (1997): **The C++ Programming Language**. Upper Saddle River, NJ: Pearson Education, Inc.





# Built-in Types

The table shows how many bytes in memory will be allocated when using a specific data type

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	May be signed or unsigned Always exactly 1 byte
	wchar_t	1 byte	
	char16_t	2 bytes	C++11 type
	char32_t	4 bytes	C++11 type
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	C99/C++11 type
floating point	float	4 bytes	
	double	8 bytes	
	long double	8 bytes	

Source credit: <http://www.learncpp.com/cpp-tutorial/23-variable-sizes-and-the-sizeof-operator/>

# User-Defined Types

- The possibility to specify user-defined types allows
  - to create completely new data types
  - to specifically match the application and/or user needs
  - to translate complex concepts into software code
  - to improve the readability of complex systems

# User-Defined Types

```
1 // Prof. Dr. Angela Brennecke | Creative Coding II | Filmuniversitaet Babelsberg | 2018
2 // Based on: Ulrich Breymann (2017): Der C++ Programmierer. Carl Hanser Verlag München.
3 #include <iostream>
4
5 enum Color
6 {
7     red,
8     yellow,
9     green
10 };
11
12 struct Point
13 {
14     int x;
15     int y;
16     bool isVisible;
17     Color aColor;
18 };
19
```

- The **enumeration** type is a user-defined data type that allows to group a list of symbolic constant of type integer
- A variable of type Color functions like a constant integer variable

# User-Defined Types

```
1 // Prof. Dr. Angela Brennecke | Creative Coding II | Filmuniversitaet Babelsberg | 2018
2 // Based on: Ulrich Breymann (2017): Der C++ Programmierer. Carl Hanser Verlag München.
3 #include <iostream>
4
5 enum Color
6 {
7     red,
8     yellow,
9     green
10 };
11
12 struct Point
13 {
14     int x;
15     int y;
16     bool isVisible;
17     Color aColor;
18 };
19
```

- The **structure** type allows to combine different kinds of data types inside of one struct
- The keyword **struct** is required to define the data type

# User-Defined Types

- Typical **class** interface of an openFrameworks application class
- The class prototype is usually defined in the header file \*.h ...

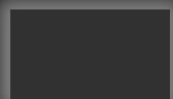
```
1 #pragma once
2 #include "ofMain.h"
3
4 // ofApp derives from ofBaseApp
5 class ofApp : public ofBaseApp
6 {
7     // private section - everything here is private and only
8     // accessible from within the class ofApp
9
10    public:
11        // everything after keyword public can be accessed from
12        // outside of the class, i.e., the public area represents
13        // the interface of the class
14        void setup();
15        void update();
16        void draw();
17
18        void keyPressed(int key);
19        void keyReleased(int key);
20        void mouseMoved(int x, int y );
21        void mouseDragged(int x, int y, int button);
22        void mousePressed(int x, int y, int button);
23        void mouseReleased(int x, int y, int button);
24
25    private:
26        // everything after keyword private can not be accessed from
27        // outside of the class but only from this class alone
28        // the private area represents the "hidden" implementation
29        // details of the class
30        std::string appFirstWords;
31        bool startDrawing;
32};
```

# User-Defined Types

- ... whereas the actual implementation is defined in the definition file \*.cpp

```
1  #include " ofApp.h"
2
3  //-----
4  void ofApp::setup()
5  {
6      ofBackground(0);
7      ofSetBackgroundAuto(false);
8  }
9
10 //-----
11 void ofApp::update()
12 {
13
14 }
15
16 //-----
17 void ofApp::draw()
18 {
19
20 }
21 |
22 //-----
23 void ofApp::keyPressed(int key)
24 {
25
26 }
27
28 //-----
29 void ofApp::keyReleased(int key){
30
```

# Variables



# Variables

- Variables in C++ are **objects** that have a name, can be accessed by their name, and can be changed — unless they are constants
- On a hardware level, an object is a piece of memory that is allocated & reserved by the computer based on the **data type**
- The data type helps to interpret the allocated memory





# Type Association

- Variables are associated with a certain data type in the code

```
float grade {5.0};           // uniform initialization
int years = 4 + 3;           // copy initialization (slower)
bool test;                   // declaration (better do initialization)

auto grade {5.0};
auto years = 4 + 3;
auto test;                   // error

myClass myObjectVar {};      // user defined type calling default constructor
```

- The newly introduced type **auto** allows for automatic type deduction — this only works for initializing variables upon creation (see <https://www.learncpp.com/cpp-tutorial/4-8-the-auto-keyword/>)

# Initialization

- **Initialization** describes the process of immediately specifying the value of the variable once it has been defined

```
1 int numYears {10};    // uniform initialization of a variable since C++11
2 int numMonths {};     // uniform initialization to zero
3
4 int numDays = 55;     // copy initialization (slower)
5
6 int numMinutes(45);   // direct initialization (old version)
```

- Uninitialized variables hold some kind of random value
- **Always initialize variables to avoid undefined behavior**

# Constants

- Variables are used to store a certain value of a certain data type
- Sometimes, these values shall change
- Sometimes, these values shall not change — rather the variable is used as a **constant** or even **symbolic constant**
- C++ provides to options to ensure that variables can only be initialized but not changed — **const** & **constexpr**

# const Keyword

- The const keyword is most of used for function parameters to ensure that the function does not change the argument

```
1 void printInteger(const int myValue)
2 {
3     std::cout << myValue;
4 }
```

Image credit: <http://www.learncpp.com/cpp-tutorial/2-9-symbolic-constants-and-the-const-keyword/>

- The const keyword is also used for variables that are being initialized during run-time — that are not know at compile-time

```
1 std::cout << "Enter your age: ";
2 int age;
3 std::cin >> age;
4
5 const int usersAge (age); // usersAge can not be changed
```

Image credit: <http://www.learncpp.com/cpp-tutorial/2-9-symbolic-constants-and-the-const-keyword/>

# constexpr Keyword

```
1 constexpr double gravity (9.8); // ok, the value of 9.8 can be resolved at compile-time
2 constexpr int sum = 4 + 5; // ok, the value of 4 + 5 can be resolved at compile-time
3
4 std::cout << "Enter your age: ";
5 int age;
6 std::cin >> age;
7 constexpr int myAge = age; // not okay, age can not be resolved at compile-time
```

Image credit: <http://www.learncpp.com/cpp-tutorial/2-9-symbolic-constants-and-the-const-keyword/>

- The **constexpr** keyword is used for variables that are known at compile-time & can be directly initialized with a value then
- constexpr variables can well be used for symbolic constants