**Structures**

Structures allow developers to create their own types ("user-defined" types) to aggregate data relevant to their needs.

For example, a user might define a Rectangle structure to hold data about rectangles used in a program.

**struct** Rectangle {

**float** length;

**float** width;

};

**Types**

Every C++ variable is defined with a [**type**](http://www.cplusplus.com/doc/tutorial/variables/).

**int** value;

**Rectangle** rectangle;

**Sphere** earth;

In this example, the "type" of value is int. Furthermore, rectangle is "of type" Rectangle, and earth has type Sphere.

**Fundamental Types**

C++ includes [**fundamental types**](https://en.cppreference.com/w/cpp/language/types), such as int and float. These fundamental types are sometimes called [**"primitives"**](https://www.geeksforgeeks.org/c-data-types/).

The Standard Library [includes additional types](, such as std::size\_t and std::string.

**User-Defined Types**

Structures are "user-defined" types. Structures are a way for programmers to create types that aggregate and store data in way that makes sense in the context of a program.

For example, C++ does not have a fundamental type for storing a date. (The Standard Library does include types related to [**time**](https://en.cppreference.com/w/cpp/chrono), which can be converted to dates.)

A programmer might desire to create a type to store a date.

Consider the following example:

**struct** Date {

**int** day;

**int** month;

**int** year;

};

The code above creates a structure containing three "member variables" of type int: day, month and year.

If you then create an "instance" of this structure, you can initialize these member variables:

*// Create an instance of the Date structure*

Date date;

*// Initialize the attributes of Date*

date.day = 1;

date.month = 10;

date.year = 2019;

**Member Initialization**

Generally, we want to avoid instantiating an object with undefined members. Ideally, we would like all members of an object to be in a valid state once the object is instantiated. We can change the values of the members later, but we want to avoid any situation in which the members are ever in an invalid state or undefined.

In order to ensure that objects of our Date structure always start in a valid state, we can initialize the members from within the structure definition.

**struct** Date {

**int** day{1};

**int** month{1};

**int** year{0};

};

There are also several other approaches to either initialize or assign member variables when the object is instantiated. For now, however, this approach ensures that every object of Date begins its life in a defined and valid state.

**Access Specifiers**

Members of a structure can be specified as public or private.

By default, all members of a structure are public, unless they are specifically marked private.

Public members can be changed directly, by any user of the object, whereas private members can only be changed by the object itself.

**Private Members**

This is an implementation of the Date structure, with all members marked as private.

**struct** Date {

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

Private members of a class are accessible only from within other member functions of the same class (or from their "friends", which we’ll talk about later).

There is a third access modifier called protected, which implies that members are accessible from other member functions of the same class (or from their "friends"), and also from members of their derived classes. We'll also discuss about derived classes later, when we learn about inheritance.

**Accessors And Mutators**

To access private members, we typically define public "accessor" and "mutator" member functions (sometimes called "getter" and "setter" functions).

**struct** Date {

**public**:

**int** **Day**() { **return** day; }

**void** **Day**(**int** day) { **this**.day = day; }

**int** **Month**() { **return** month; }

**void** **Month**(**int** month) { **this**.month = month; }

**int** **Year**() { **return** year; }

**void** **Year**(**int** year) { **this**.year = year; }

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

In the last example, you saw how to create a setter function for class member attributes. Check out the code in the Notebook below to play around a bit with access modifiers as well as setter and getter functions!

**Avoid Trivial Getters And Setters**

Sometimes accessors are not necessary, or even advisable. The [**C++ Core Guidelines**](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rh-get) recommend, "A trivial getter or setter adds no semantic value; the data item could just as well be public."

Here is the example from the Core Guidelines:

**class** **Point** {

**int** x;

**int** y;

**public**:

Point(**int** xx, **int** yy) : x{xx}, y{yy} { }

**int** **get\_x**() **const** { **return** x; } *// const here promises not to modify the object*

**void** **set\_x**(**int** xx) { x = xx; }

**int** **get\_y**() **const** { **return** y; } *// const here promises not to modify the object*

**void** **set\_y**(**int** yy) { y = yy; }

*// no behavioral member functions*

};

This class could be made into a struct, with no logic or "invariants", just passive data. The member variables could both be public, with no accessor functions:

**struct** Point { *// Good: concise*

**int** x {0}; *// public member variable with a default initializer of 0*

**int** y {0}; *// public member variable with a default initializer of 0*

};

다음

# Classes

Classes, like structures, provide a way for C++ programmers to aggregate data together in a way that makes sense in the context of a specific program. By convention, programmers use structures when member variables are independent of each other, and [**use classes when member variables are related by an "invariant"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c2-use-class-if-the-class-has-an-invariant-use-struct-if-the-data-members-can-vary-independently).

## Invariants

An "invariant" is a rule that limits the values of member variables.

For example, in a Date class, an invariant would specify that the member variable day cannot be less than 0. Another invariant would specify that the value of day cannot exceed 28, 29, 30, or 31, depending on the month and year. Yet another invariant would limit the value of month to the range of 1 to 12.

## Date Class

Let's define a Date class:

*// Use the keyword “class” to define a Date class:*

**class** Date {

**int** day{1};

**int** month{1};

**int** year{0};

};

So far, this class definition provides no invariants. The data members can vary independently of each other.

There is one subtle but important change that takes place when we change struct Date to class Date. By default, all members of a struct default to public, whereas all members of a class default to private. Since we have not specified access for the members of class Date, all of the members are private. In fact, we are not able to assign value to them at all!

### Date Accessors And Mutators

As the first step to adding the appropriate invariants, let's specify that the member variable day is private. In order to access this member, we'll provide accessor and mutatot functions. Then we can add the appropriate invariants to the mutators.

**class** Date {

**public**:

**int** **Day**() { **return** day\_; }

**void** **Day**(**int** d) { day\_ = d; }

**private**:

**int** day\_{1};

**int** month\_{1};

**int** year\_{0};

};

### Date Invariants

Now we can add the invariants within the mutators.

**class** Date {

**public**:

**int** **Day**() { **return** day; }

**void** **Day**(**int** d) {

**if** (d >= 1 && d <= 31) day\_ = d;

}

**private**:

**int** day\_{1};

**int** month\_{1};

**int** year\_{0};

};

Now we have a set of invariants for the the class members!

As a general rule, member data subject to an invariant should be specified private, in order to enforce the invariant before updating the member's value.

# Constructors

Constructors are member functions of a class or struct that initialize an object. The Core Guidelines [**define a constructor**](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/0a4b606e-7aa2-4e1c-8d67-cb5ddd41a6b5/lessons/287ef495-0297-459f-b339-cfe087c7b11a/concepts/(http:/isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary)) as:

constructor: an operation that initializes (“constructs”) an object. Typically a constructor establishes an invariant and often acquires resources needed for an object to be used (which are then typically released by a destructor).

A constructor can take arguments, which can be used to assign values to member variables.

**class** Date {

**public**:

Date(**int** d, **int** m, **int** y) { *// This is a constructor.*

Day(d);

}

**int** **Day**() { **return** day; }

**void** **Day**(**int** d) {

**if** (d >= 1 && d <= 31) day = d;

}

**int** **Month**() { **return** month; }

**void** **Month**(**int** m) {

**if** (m >= 1 && m <= 12) month = m;

}

**int** **Year**() { **return** year\_; }

**void** **Year**(**int** y) { year = y; }

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

As you can see, a constructor is also able to call other member functions of the object it is constructing. In the example above, Date(int d, int m, int y) assigns a member variable by calling Day(int d).

## Default Constructor

A class object is always initialized by calling a constructor. That might lead you to wonder how it is possible to initialize a class or structure that does not define any constructor at all.

For example:

**class** Date {

**int** day{1};

**int** month{1};

**int** year{0};

};

We can initialize an object of this class, even though this class does not explicitly define a constructor.

This is possible because of the [**default constructor**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#cctor-constructors-assignments-and-destructors). [**The compiler will define a default constructor**](https://en.cppreference.com/w/cpp/language/default_constructor), which accepts no arguments, for any class or structure that does not contain an explicitly-defined constructor.

# Scope Resolution

C++ allows different [**identifiers**](https://en.cppreference.com/w/cpp/language/identifiers) (variable and function names) to have the same name, as long as they have different scope. For example, two different functions can each declare the variable int i, because each variable only exists within the scope of its parent function.

In some cases, scopes can overlap, in which case the compiler may need assistance in determining which identifier the programmer means to use. The process of determining which identifier to use is called [**"scope resolution"**](https://docs.microsoft.com/en-us/cpp/cpp/scope-resolution-operator?view=vs-2019).

## Scope Resultion Operator

:: is the [**scope resolution operator**](https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_74/rzarg/cplr175.htm). We can use this operator to specify which namespace or class to search in order to resolve an identifier.

Person::move(); \\ **Call** the **move** the **function** that **is** a **member** **of** the Person **class**.

**std**::**map** **m**; \\ Initialize the map container from the C++ Standard Library.

## Class

Each class provides its own scope. We can use the scope resolution operator to specify identifiers from a class.

This becomes particularly useful if we want to separate class declaration from class definition.

**class** Date {

**public**:

**int** **Day**() **const** { **return** day; }

**void** **Day**(**int** day); *// Declare member function Date::Day().*

**int** **Month**() **const** { **return** month; }

**void** **Month**(**int** month) {

**if** (month >= 1 && month <= 12) Date::month = month;

}

**int** **Year**() **const** { **return** year; }

**void** **Year**(**int** year) { Date::year = year; }

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

*// Define member function Date::Day().*

**void** Date::Day(**int** day) {

**if** (day >= 1 && day <= 31) Date::day = day;

}

## Namespaces

[**Namespaces**](https://en.cppreference.com/w/cpp/language/namespace) allow programmers to group logically related variables and functions together. Namespaces also help to avoid conflicts between to variables that have the same name in different parts of a program.

**namespace** English {

**void** **Hello**() { std::cout << "Hello, World!\n"; }

} *// namespace English*

**namespace** Spanish {

**void** **Hello**() { std::cout << "Hola, Mundo!\n"; }

} *// namespace Spanish*

**int** **main**() {

English::Hello();

Spanish::Hello();

}

In this example, we have two different void Hello() functions. If we put both of these functions in the same namespace, they would conflict and the program would not compile. However, by declaring each of these functions in a separate namespace, they are able to co-exist. Furthermore, we can specify which function to call by prefixing Hello() with the appropriate namespace, followed by the :: operator.

### std Namespace

You are already familiar with the std namespace, even if you didn't realize quite what it was. std is the namespace used by the [**C++ Standard Library**](https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library).

Classes like [**std::vector**](https://en.cppreference.com/w/cpp/container/vector) and functions like [**std::sort**](https://en.cppreference.com/w/cpp/algorithm/sort) are defined within the std namespace.

# Initializer Lists

[**Initializer lists**](https://en.cppreference.com/w/cpp/language/initializer_list) initialize member variables to specific values, just before the class constructor runs. This initialization ensures that class members are automatically initialized when an instance of the class is created.

Date::Date(**int** day, **int** month, **int** year) : year\_(y) {

Day(day);

Month(month);

}

In this example, the member value year is initialized through the initializer list, while day and month are assigned from within the constructor. Assigning day and month allows us to apply the invariants set in the mutator.

In general, [**prefer initialization to assignment**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c49-prefer-initialization-to-assignment-in-constructors). Initialization sets the value as soon as the object exists, whereas assignment sets the value only after the object comes into being. This means that assignment creates and opportunity to accidentally use a variable before its value is set.

In fact, initialization lists ensure that member variables are initialized before the object is created. This is why class member variables can be declared const, but only if the member variable is initialized through an initialization list. Trying to initialize a const class member within the body of the constructor will not work.

## Instructions

1. Declare class Person.
2. Add std::string name to class Person.
3. Create a constructor for class Person.
4. Add an initializer list to the constructor.
5. Create class object.

#include <assert.h>

#include <string>

// TODO: Define class Person

struct Person {

// TODO: Define a public constructor with an initialization list

Person(std::string name) : name(name) {}

// TODO: Define a public member variable: name

std::string name;

};

// Test

int main() {

Person alice("Alice");

Person bob("Bob");

assert(alice.name != bob.name);

}

## Exercise: Constructor Syntax

Initializer lists exist for a number of reasons. First, the compiler can optimize initialization faster from an initialization list than from within the constructor.

A second reason is a bit of a technical paradox. If you have a const class attribute, you can only initialize it using an initialization list. Otherwise, you would violate the const keyword simply by initializing the member in the constructor!

The third reason is that attributes defined as [**references**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) must use initialization lists.

This exercise showcases several advantages of initializer lists.

## Instructions

1. Modify the exist code to use an initialization list.
2. Verify that the test passes.

#include <assert.h>

#include <string>

struct Person {

public:

// TODO: Add an initialization list

Person(std::string const & n) : name(n) {}

std::string const name;

};

// Test

int main() {

Person alice("Alice");

Person bob("Bob");

assert(alice.name != bob.name);

}

**Encapsulation**

[**Encapsulation**](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming%29) is the grouping together of data and logic into a single unit. In object-oriented programming, classes encapsulate data and functions that operate on that data.

This can be a delicate balance, because on the one hand we want to group together relevant data and functions, but on the hand we want to [**limit member functions to only those functions that need direct access to the representation of a class**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c4-make-a-function-a-member-only-if-it-needs-direct-access-to-the-representation-of-a-class).

In the context of a Date class, a function Date Tomorrow(Date const & date) probably does not need to be encapsulated as a class member. It can exist outside the Date class.

However, a function that calculates the number of days in a month probably should be encapsulated with the class, because the class needs this function in order to operate correctly.

#include <cassert>

class Date {

public:

Date(int day, int month, int year);

int Day() const { return day\_; }

void Day(int day);

int Month() const { return month\_; }

void Month(int month);

int Year() const { return year\_; }

void Year(int year);

private:

int day\_{1};

int month\_{1};

int year\_{0};

};

Date::Date(int day, int month, int year) {

Year(year);

Month(month);

Day(day);

}

void Date::Day(int day) {

if (day >= 1 && day <= 31)

day\_ = day;

}

void Date::Month(int month) {

if (month >= 1 && month <= 12)

month\_ = month;

}

void Date::Year(int year) { year\_ = year; }

// Test

int main() {

Date date(29, 8, 1981);

assert(date.Day() == 29);

assert(date.Month() == 8);

assert(date.Year() == 1981);

}

# Accessor Functions

Accessor functions are public member functions that allow users to access an object's data, albeit indirectly.

## const

Accessors should only retrieve data. They should not change the data stored in the object.

The main role of the const specifier in accessor methods is to protect member data. When you specify a member function as const, the compiler will prohibit that function from changing any of the object's member data.

## Exercise: Bank Account Class

Your task is to design and implement class called BankAccount. This will be a generic account defined by its account number, the name of the owner and the funds available.

Complete the following steps:

1. Create class called BankAccount
2. Use typical info about bank accounts to design attributes, such as the account number, the owner name, and the available funds.
3. Specify access so that member data are protected from other parts of the program.
4. Create accessor and mutator functions for member data.

#include <iostream>

#include <string>

class BankAccount

{

public:

int number;

std::string owner;

double funds;

};

int main(){

// TODO: instantiate and output a bank account

BankAccount account;

account.number = 123456789;

account.owner = "David Silver";

account.funds = 1000000.01;

std::cout << "Account Information\n";

std::cout << "-------------------\n";

std::cout << "ID: " << account.number << "\n";

std::cout << "Owner: " << account.owner << "\n";

std::cout << "Funds: $" << account.funds << "\n";

}

# Mutator Functions

A mutator ("setter") function can apply logic ("invariants") when updating member data.

## Exercise: Car Class

In this lab you will create a setter method that receives data as an argument an converts it to a different type. Specifically, you will receive a string as input and convert it to a character array.

1. Create a class called Car.
2. Create 3 member variables: horsepower, weight and brand. The brand attribute must be a character array.
3. Create accessor and mutator functions for all member data. The mutator function for brand must accept a C++ string as a parameter and convert that string into a [**C-style string**](https://www.learncpp.com/cpp-tutorial/66-c-style-strings/) (a character array ending in null character) to set the value of brand.
4. The accessor function for the brand must return a string, so in this function you first will need to convert brand to std::string, and then return it.

#include <string>

#include <cstring>

#include <iostream>

class Car {

// TODO: Declare private attributes

private:

std::string \_brand;

// TODO: Declare getter and setter for brand

public:

void brand(char\*);

std::string brand() const;

};

// Define setters

void Car::brand(char\* brand)

{

Car::\_brand = brand;

}

// Define getters

std::string Car::brand() const

{

return \_brand;

}

// Test in main()

int main()

{

Car car;

char brand[] = "Peugeot";

car.brand(brand);

std::cout << car.brand() << "\n";

}

# Abstraction

Abstraction refers to the separation of a class's interface from the details of its implementation. The interface provides a way to interact with an object, while hiding the details and implementation of how the class works.

## Example

The String() function within this Date class is an example of abstraction.

**class** Date {

**public**:

...

std::string **String**() **const**;

...

};

The user is able to interact with the Date class through the String() function, but the user does not need to know about the implementation of either Date or String().

For example, the user does not know, or need to know, that this object internally contains three int member variables. The user can just call the String() method to get data.

If the designer of this class ever decides to change how the data is stored internally -- using a vector of ints instead of three separate ints, for example -- the user of the Date class will not need to know.

#include <cassert>

#include <string>

#include <vector>

class Date {

public:

Date(int day, int month, int year);

int Day() const { return day\_; }

void Day(int day);

int Month() const { return month\_; }

void Month(int month);

int Year() const { return year\_; }

void Year(int year);

std::string String() const;

private:

bool LeapYear(int year) const;

int DaysInMonth(int month, int year) const;

int day\_{1};

int month\_{1};

int year\_{0};

};

Date::Date(int day, int month, int year) {

Year(year);

Month(month);

Day(day);

}

bool Date::LeapYear(int year) const {

if (year % 4 != 0)

return false;

else if (year % 100 != 0)

return true;

else if (year % 400 != 0)

return false;

else

return true;

}

int Date::DaysInMonth(int month, int year) const {

if (month == 2)

return LeapYear(year) ? 29 : 28;

else if (month == 4 || month == 6 || month == 9 || month == 11)

return 30;

else

return 31;

}

void Date::Day(int day) {

if (day >= 1 && day <= DaysInMonth(Month(), Year())) day\_ = day;

}

void Date::Month(int month) {

if (month >= 1 && month <= 12) month\_ = month;

}

void Date::Year(int year) {

year\_ = year;

}

std::string Date::String() const {

std::vector<std::string> months{"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"};

return months[Month()-1] + " " + std::to\_string(Day()) + ", " + std::to\_string(Year());

}

// Test

int main() {

Date date(29, 8, 1981);

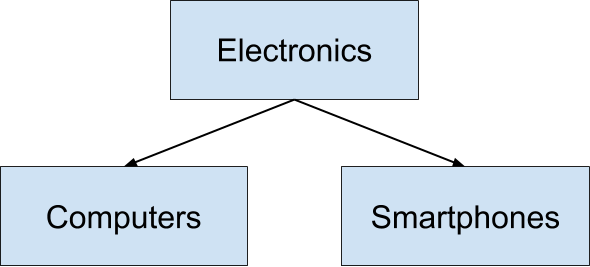
assert(date.String() == "August 29, 1981");

}

## Inheritence

In our everyday life, we tend to divide things into groups, based on their shared characteristics. Here are some groups that you have probably used yourself: electronics, tools, vehicles, or plants.

Sometimes these groups have hierarchies. For example, computers and smartphones are both types of electronics, but computers and smartphones are also groups in and of themselves. You can imagine a tree with "electronics" at the top, and "computers" and "smartphones" each as children of the "electronics" node.



Object-oriented programming uses the same principles! For instance, imagine a Vehicle class:

**class** Vehicle {

**public**:

**int** wheels = 0;

string color = "blue";

**void** **Print**() **const**

{

std::cout << "This " << color << " vehicle has " << wheels << " wheels!\n";

}

};

We can derive other classes from Vehicle, such as Car or Bicycle. One advantage is that this saves us from having to re-define all of the common member variables - in this case, wheels and color - in each derived class.

Another benefit is that derived classes, for example Car and Bicycle, can have distinct member variables, such as sunroof or kickstand. Different derived classes will have different member variables:

**class** Car : **public** Vehicle {

**public**:

**bool** sunroof = false;

};

**class** Bicycle : **public** Vehicle {

**public**:

**bool** kickstand = true;

};

## Instructions

1. Add a new member variable to class Vehicle.
2. Output that new member in main().
3. Derive a new class from Vehicle, alongside Car and Bicycle.
4. Instantiate an object of that new class.
5. Print the object.

#include <iostream>

#include <string>

using std::string;

class Vehicle {

public:

int wheels = 0;

string color = "blue";

string make = "generic";

void Print() const

{

std::cout << "This " << color << " " << make << " vehicle has " << wheels << " wheels!\n";

}

};

class Car : public Vehicle {

public:

bool sunroof = false;

};

class Bicycle : public Vehicle {

public:

bool kickstand = true;

};

class Scooter : public Vehicle {

public:

bool electric = false;

};

int main()

{

Scooter scooter;

scooter.wheels = 2;

scooter.Print();

};

## Inherited Access Specifiers

Just as access specifiers (i.e. public, protected, and private) define which class members users can access, the same access modifiers also define which class members users of a derived classes can access.

[**Public inheritance:**](https://en.cppreference.com/w/cpp/language/derived_class#Public_inheritance) the public and protected members of the base class listed after the specifier keep their member access in the derived class

[**Protected inheritance:**](https://en.cppreference.com/w/cpp/language/derived_class#Protected_inheritance) the public and protected members of the base class listed after the specifier are protected members of the derived class

[**Private inheritance:**](https://en.cppreference.com/w/cpp/language/derived_class#Private_inheritance) the public and protected members of the base class listed after the specifier are private members of the derived class

Source: [**C++ reference**](https://en.cppreference.com/w/cpp/language/access)

In the exercise below, you'll experiment with access modifiers.

## Instructions

1. Update the derived classes so that one has protected inheritance and one has private inheritance.
2. Try to access a protected member from main(). Is it possible?
3. Try to access a private member from main(). Is it possible?
4. Try to access a member of the base class from within the derived class that has protected inheritance. Is it possible?
5. Try to access a member of the base class from within the derived class that has private inheritance. Is it possible?

// This example demonstrates the privacy levels

// between parent and child classes

#include <iostream>

#include <string>

using std::string;

class Vehicle {

public:

int wheels = 0;

string color = "blue";

void Print() const

{

std::cout << "This " << color << " vehicle has " << wheels << " wheels!\n";

}

};

class Car : public Vehicle {

public:

bool sunroof = false;

};

class Bicycle : protected Vehicle {

public:

bool kickstand = true;

void Wheels(int w)

{

wheels = w;

}

};

class Scooter : private Vehicle {

public:

bool electric = false;

void Wheels(int w)

{

wheels = w;

}

};

int main()

{

Car car;

car.wheels = 4;

Bicycle bicycle;

bicycle.Wheels(2);

Scooter scooter;

scooter.Wheels(1);

};

## Composition

[**Composition**](https://en.wikipedia.org/wiki/Composition_over_inheritance) is a closely related alternative to inheritance. Composition involves constructing ("composing") classes from other classes, instead of inheriting traits from a parent class.

A common way to distinguish "composition" from "inheritance" is to think about what an object can do, rather than what it is. This is often expressed as [**"has a"**](https://en.wikipedia.org/wiki/Has-a) versus [**"is a"**](https://en.wikipedia.org/wiki/Is-a).

From the standpoint of composition, a cat "has a" head and "has a" set of paws and "has a" tail.

From the standpoint of inheritance, a cat "is a" mammal.

There is [**no hard and fast rule**](https://www.google.com/search?q=when+to+use+composition+and+when+to+use+inheritance&oq=when+to+use+composition+and+when+to+use+inheritance) about when to prefer composition over inheritance. In general, if a class needs only extend a small amount of functionality beyond what is already offered by another class, it makes sense to **inherit** from that other class. However, if a class needs to contain functionality from a variety of otherwise unrelated classes, it makes sense to **compose** the class from those other classes.

In this example, you'll practice working with composition in C++.

### Instructions

In this exercise, you will start with a LineSegment class and create a Circle class.

Note that you will compose Circle from LineSegment, instead of inheriting Circle from LineSegment. Specifically, the length attribute from LineSegment will become the circle's radius.

1. Create a class LineSegment.
2. Declare an attribute length in class LineSegment.
3. Define pi (3.14159) with a [**macro**](http://www.cplusplus.com/doc/tutorial/preprocessor/).
4. Create a class Circle, composed of a LineSegment that represent's the circle's radius. Use this radius to calculate the area of the circle (area of a circle = \pi r^2*πr*2).
5. Verify the behavior of Circle in main().

// Example solution for Circle class

#include <iostream>

#include <cmath>

#include <assert.h>

// Define PI

#define PI 3.14159;

// Define LineSegment struct

struct LineSegment {

// Define protected attribute length

public:

double length;

};

// Define Circle class

class Circle {

public:

Circle(LineSegment& radius);

double Area();

private:

LineSegment& radius\_;

};

// Declare Circle class

Circle::Circle(LineSegment& radius) : radius\_(radius) {}

double Circle::Area()

{

return pow(Circle::radius\_.length, 2) \* PI;

}

// Test in main()

int main()

{

LineSegment radius {3};

Circle circle(radius);

assert(int(circle.Area()) == 28);

}

## Polymorphism

[**Polymorphism**](https://www.merriam-webster.com/dictionary/polymorphism) is means "assuming many forms".

In the context of object-oriented programming, [**polymorphism**](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) describes a paradigm in which a function may behave differently depending on how it is called. In particular, the function will perform differently based on its inputs.

Polymorphism can be achieved in two ways in C++: overloading and overriding. In this exercise we will focus on overloading.

## Overloading

In C++, you can write two (or more) versions of a function with the same name. This is called [**"overloading"**](https://en.wikipedia.org/wiki/Function_overloading). Overloading requires that we leave the function name the same, but we modify the function signature. For example, we might define the same function name with multiple different configurations of input arguments.

This example of class Date overloads:

**#include <ctime>**

**class** Date {

**public**:

Date(**int** day, **int** month, **int** year) : day\_(day), month\_(month), year\_(year) {}

Date(**int** day, **int** month) : day\_(day), month\_(month) *// automatically sets the Date to the current year*

{

**time\_t** t = time(NULL);

tm\* timePtr = localtime(&t);

year\_ = timePtr->tm\_year;

}

**private**:

**int** day\_;

**int** month\_;

**int** year\_;

};

## Instructions

Overloading can happen outside of an object-oriented context, too. In this exercise, you will practice overloading a normal function that is not a class member.

1. Create a function hello() that outputs, "Hello, World!"
2. Create a class Human.
3. Overload hello() by creating a function hello(Human human). This function should output, "Hello, Human!"
4. Create 2 more classes and use those classes to further overload the hello() function.

#include <iostream>

class Human {};

class Dog {};

class Cat {};

// TODO: Write hello() function

void hello() { std::cout << "Hello, World!\n"; }

// TODO: Overload hello() three times

void hello(Human human) { std::cout << "Hello, Human!\n"; }

void hello(Dog dog) { std::cout << "Hello, Dog!\n"; }

void hello(Cat cat) { std::cout << "Hello, Cat!\n"; }

// TODO: Call hello() from main()

int main()

{

hello();

hello(Human());

hello(Dog());

hello(Cat());

}

## Polymorphism

[**Polymorphism**](https://www.merriam-webster.com/dictionary/polymorphism) is means "assuming many forms".

In the context of object-oriented programming, [**polymorphism**](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) describes a paradigm in which a function may behave differently depending on how it is called. In particular, the function will perform differently based on its inputs.

Polymorphism can be achieved in two ways in C++: overloading and overriding. In this exercise we will focus on overloading.

## Overloading

In C++, you can write two (or more) versions of a function with the same name. This is called [**"overloading"**](https://en.wikipedia.org/wiki/Function_overloading). Overloading requires that we leave the function name the same, but we modify the function signature. For example, we might define the same function name with multiple different configurations of input arguments.

This example of class Date overloads:

**#include <ctime>**

**class** Date {

**public**:

Date(**int** day, **int** month, **int** year) : day\_(day), month\_(month), year\_(year) {}

Date(**int** day, **int** month) : day\_(day), month\_(month) *// automatically sets the Date to the current year*

{

**time\_t** t = time(NULL);

tm\* timePtr = localtime(&t);

year\_ = timePtr->tm\_year;

}

**private**:

**int** day\_;

**int** month\_;

**int** year\_;

};

## Instructions

Overloading can happen outside of an object-oriented context, too. In this exercise, you will practice overloading a normal function that is not a class member.

1. Create a function hello() that outputs, "Hello, World!"
2. Create a class Human.
3. Overload hello() by creating a function hello(Human human). This function should output, "Hello, Human!"
4. Create 2 more classes and use those classes to further overload the hello() function.

#include <iostream>

class Human {};

class Dog {};

class Cat {};

// TODO: Write hello() function

void hello() { std::cout << "Hello, World!\n"; }

// TODO: Overload hello() three times

void hello(Human human) { std::cout << "Hello, Human!\n"; }

void hello(Dog dog) { std::cout << "Hello, Dog!\n"; }

void hello(Cat cat) { std::cout << "Hello, Cat!\n"; }

// TODO: Call hello() from main()

int main()

{

hello();

hello(Human());

hello(Dog());

hello(Cat());

}

# Operator Overloading

. In this exercise you'll see how to achieve polymorphism with [**operator overloading**](https://en.cppreference.com/w/cpp/language/operators). You can choose any operator from the ASCII table and give it your own set of rules!

Operator overloading can be useful for many things. Consider the + operator. We can use it to add ints, doubles, floats, or even std::strings.

In order to overload an operator, use the operator keyword in the function signature:

Complex **operator**+(**const** Complex& addend) {

*//...logic to add complex numbers*

}

Imagine vector addition. You might want to perform vector addition on a pair of points to add their x and y components. The compiler won't recognize this type of operation on its own, because this data is user defined. However, you can overload the + operator so it performs the action that you want to implement.

## Instructions

1. Define class Point.
2. Declare a prototype of overload method for + operator.
3. Confirm the tests pass.

#include <assert.h>

// TODO: Define Point class

class Point {

public:

// TODO: Define public constructor

Point(int x = 0, int y = 0) : x(x), y(y) {}

// TODO: Define + operator overload

Point operator+(const Point& addend) {

Point sum;

sum.x = x + addend.x;

sum.y = y + addend.y;

return sum;

}

// TODO: Declare attributes x and y

int x, y;

};

// Test in main()

int main() {

Point p1(10, 5), p2(2, 4);

Point p3 = p1 + p2; // An example call to "operator +";

assert(p3.x == p1.x + p2.x);

assert(p3.y == p1.y + p2.y);

}

# Virtual Functions

Virtual functions are a polymorphic feature. These functions are declared (and possibly defined) in a base class, and can be overridden by derived classes.

버츄얼 펑션 쓰면은 무조건 implementaion을 해줘야 한다. 예를들어 Animail에게 speak가 있다고하면, 사람이 어떻게 speak하는지 따로 코딩 해야하는게 맞다.

This approach declares an [**interface**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) at the base level, but delegates the implementation of the interface to the derived classes.

In this exercise, class Shape is the base class. Geometrical shapes possess both an area and a perimeter. Area() and Perimeter() should be virtual functions of the base class interface. Append = 0 to each of these functions in order to declare them to be "pure" virtual functions.

A [**pure virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) is a [**virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) that the base class [**declares**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) but does not [**define**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary).

A pure virtual function has the side effect of making its class [**abstract**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary). This means that the class cannot be instantiated. Instead, only classes that derive from the abstract class and override the pure virtual function can be instantiated.

**class** Shape {

**public**:

Shape() {}

**virtual** **double** **Area**() **const** = 0;

**virtual** **double** **Perimeter**() **const** = 0;

};

Virtual functions can be defined by derived classes, but this is not required. However, if we mark the virtual function with = 0 in the base class, then we are declaring the function to be a pure virtual function. This means that the base class does not define this function. A derived class must define this function, or else the derived class will be abstract.

### Instructions

1. Create base class called Shape.
2. Define pure virtual functions (= 0) for the base class.
3. Write the derived classes.
   * Inherit from class Shape.
   * Override the pure virtual functions from the base class.
4. Test in main()

// Example solution for Shape inheritance

#include <assert.h>

#include <cmath>

// TODO: Define pi

#define PI 3.14159;

// TODO: Define the abstract class Shape

class Shape {

public:

// TODO: Define public virtual functions Area() and Perimeter()

// TODO: Append the declarations with = 0 to specify pure virtual functions

virtual double Area() const = 0;

virtual double Perimeter() const = 0;

};

// TODO: Define Rectangle to inherit publicly from Shape

class Rectangle : public Shape {

public:

// TODO: Declare public constructor

Rectangle(double width, double height) : width\_(width), height\_(height) {}

// TODO: Override virtual base class functions Area() and Perimeter()

double Area() const override { return width\_ \* height\_; }

double Perimeter() const override { return 2 \* (width\_ + height\_); }

private:

// TODO: Declare private attributes width and height

double width\_;

double height\_;

};

// TODO: Define Circle to inherit from Shape

class Circle : public Shape {

public:

// TODO: Declare public constructor

Circle(double radius) : radius\_(radius) {}

// TODO: Override virtual base class functions Area() and Perimeter()

double Area() const override { return pow(radius\_, 2) \* PI; }

double Perimeter() const override { return 2 \* radius\_ \* PI; }

private:

// TODO: Declare private member variable radius

double radius\_;

};

// Test in main()

int main() {

double epsilon = 0.1; // useful for floating point equality

// Test circle

Circle circle(12.31);

assert(abs(circle.Perimeter() - 77.35) < epsilon);

assert(abs(circle.Area() - 476.06) < epsilon);

// Test rectangle

Rectangle rectangle(10, 6);

assert(rectangle.Perimeter() == 32);

assert(rectangle.Area() == 60);

}

## Polymorphism: Overriding

[**"Overriding"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary) a function occurs when:

1. A base class declares a [**virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary%20function).
2. A derived class overrides that virtual function by defining its own implementation with an identical function signature (i.e. the same function name and argument types).

**class** Animal {

**public**:

**virtual** std::string **Talk**() **const** = 0;

};

**class** Cat {

**public**:

std::string **Talk**() **const** { **return** std::string("Meow"); }

};

In this example, Animal exposes a virtual function: Talk(), but does not define it. Because Animal::Talk() is undefined, it is called a [**pure virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary), as opposed to an ordinary (impure? 😉) [**virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary).

Furthermore, because Animal contains a pure virtual function, the user cannot instantiate an object of type Animal. This makes Animal an [**abstract class**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary).

Cat, however, inherits from Animal and overrides Animal::Talk() with Cat::Talk(), which is defined. Therefore, it is possible to instantiate an object of type Cat.

### Instructions

1. Create a class Dog to inherit from Animal.
2. Define Dog::Talk() to override the virtual function Animal::Talk().
3. Confirm that the tests pass.

#include <assert.h>

#include <string>

class Animal {

public:

virtual std::string Talk() const = 0;

};

// TODO: Declare a class Dog that inherits from Animal

class Dog : Animal {

public:

std::string Talk() const;

};

std::string Dog::Talk() const {

return "Woof";

}

int main() {

Dog dog;

assert(dog.Talk() == "Woof");

}

# Multiple Inheritance

In this exercise, you'll get some practical experience with multiple inheritance. If you have class Animal and another class Pet, then you can construct a class Dog, which inherits from both of these base classes. In doing this, you are able to incorporate attributes of multiple base classes.

The Core Guidelines have some worthwhile recommendations about how and when to use multiple inheritance:

* [**"Use multiple inheritance to represent multiple distinct interfaces"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c135-use-multiple-inheritance-to-represent-multiple-distinct-interfaces)
* [**"Use multiple inheritance to represent the union of implementation attributes"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c136-use-multiple-inheritance-to-represent-the-union-of-implementation-attributes)

## Instructions

1. Review class Dog, which inherits from both Animal and Pet.
2. Declare a class Cat, with a member attribute color, that also inherits from both Animal and Pet.
3. Instantiate an object of class Cat.
4. Configure that object to pass the tests in main().

#include <iostream>

#include <string>

#include <assert.h>

class Animal {

public:

double age;

};

class Pet {

public:

std::string name;

};

// Dog derives from \*both\* Animal and Pet

class Dog : public Animal, public Pet {

public:

std::string breed;

};

class Cat : public Animal, public Pet {

public:

std::string color;

};

int main()

{

/\*

Cat cat;

cat.color = "black";

cat.age = 10;

cat.name = "Max";

\*/

assert(cat.color == "black");

assert(cat.age == 10);

assert(cat.name == "Max");

}

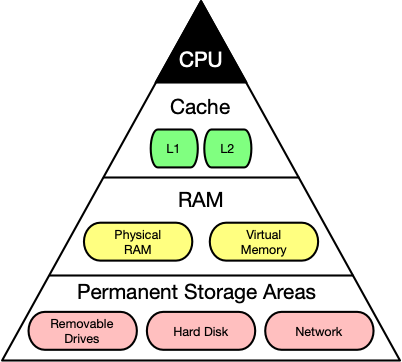
Types of Computer Memory

In a course on memory management we obviously need to take a look at the available memory types in computer systems. Below you will find a small list of some common memory types that you will surely have heard of:

* RAM / ROM
* Cache (L1, L2)
* Registers
* Virtual Memory
* Hard Disks, USB drives

Let us look into these types more deeply: When the CPU of a computer needs to access memory, it wants to do this with minimal latency. Also, as large amounts of information need to be processed, the available memory should be sufficiently large with regard to the tasks we want to accomplish.

Regrettably though, low latency and large memory are not compatible with each other (at least not at a reasonable price). In practice, the decision for low latency usually results in a reduction of the available storage capacity (and vice versa). This is the reason why a computer has multiple memory types that are arranged hierarchically. The following pyramid illustrates the principle:



Computer memory latency and size hierarchy.

As you can see, the CPU and its ultra-fast (but small) registers used for short-term data storage reside at the top of the pyramid. Below are Cache and RAM, which belong to the category of temporary memory which quickly looses its content once power is cut off. Finally, there are permanent storage devices such as the ROM, hard drives as well as removable drives such as USB sticks.

Let us take a look at a typical computer usage scenario to see how the different types of memory are used:

1. After switching on the computer, it loads data from its read-only memory (ROM) and performs a power-on self-test (POST) to ensure that all major components are working properly. Additionally, the computer memory controller checks all of the memory addresses with a simple read/write operation to ensure that memory is functioning correctly.
2. After performing the self-test, the computer loads the basic input/output system (BIOS) from ROM. The major task of the BIOS is to make the computer functional by providing basic information about such things as storage devices, boot sequence, security or auto device recognition capability.
3. The process of activating a more complex system on a simple system is called "bootstrapping": It is a solution for the chicken-egg-problem of starting a software-driven system by itself using software. During bootstrapping, the computer loads the operating system (OS) from the hard drive into random access memory (RAM). RAM is considered "random access" because any memory cell can be accessed directly by intersecting the respective row and column in the matrix-like memory layout. For performance reasons, many parts of the OS are kept in RAM as long as the computer is powered on.
4. When an application is started, it is loaded into RAM. However, several application components are only loaded into RAM on demand to preserve memory. Files that are opened during runtime are also loaded into RAM. When a file is saved, it is written to the specified storage device. After closing the application, it is deleted from RAM.

This simple usage scenario shows the central importance of the RAM. Every time data is loaded or a file is opened, it is placed into this temporary storage area - but what about the other memory types above the RAM layer in the pyramid?

To maximize CPU performance, fast access to large amounts of data is critical. If the CPU cannot get the data it needs, it stops and waits for data availability. Thus, when designing new memory chips, engineers must adapt to the speed of the available CPUs. The problem they are facing is that memory which is able to keep up with modern CPUs running at several GHz is extremely expensive. To combat this, computer designers have created the memory tier system which has already been shown in the pyramid diagram above. The solution is to use expensive memory in small quantities and then back it up using larger quantities of less expensive memory.

The cheapest form of memory available today is the hard disk. It provides large quantities of inexpensive and permanent storage. The problem of a hard disk is its comparatively low speed - even though access times with modern solid state disks (SSD) have decreased significantly compared to older magnetic-disc models.

The next hierarchical level above hard disks or other external storage devices is the RAM. We will not discuss in detail how it works but only take a look at some key performance metrics of the CPU at this point, which place certain performance expectations on the RAM and its designers:

1. The **bit size** of the CPU decides how many bytes of data it can access in RAM memory at the same time. A 16-bit CPU can access 2 bytes (with each byte consisting of 8 bit) while a 64-bit CPU can access 8 bytes at a time.
2. The **processing speed** of the CPU is measured in Gigahertz or Megahertz and denotes the number of operations it can perform in one second.

From processing speed and bit size, the data rate required to keep the CPU busy can easily be computed by multiplying bit size with processing speed. With modern CPUs and ever-increasing speeds, the available RAM in the market will not be fast enough to match the CPU data rate requirements.

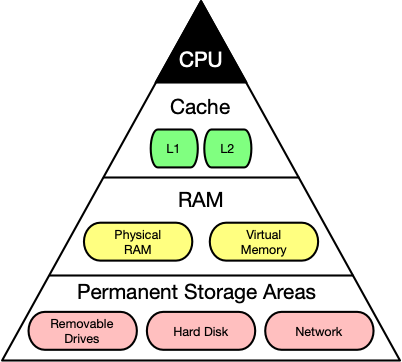
Cache Memory

In a course on memory management we obviously need to take a look at the available memory types in computer systems. Below you will find a small list of some common memory types that you will surely have heard of:

* RAM / ROM
* Cache (L1, L2)
* Registers
* Virtual Memory
* Hard Disks, USB drives

Let us look into these types more deeply: When the CPU of a computer needs to access memory, it wants to do this with minimal latency. Also, as large amounts of information need to be processed, the available memory should be sufficiently large with regard to the tasks we want to accomplish.

Regrettably though, low latency and large memory are not compatible with each other (at least not at a reasonable price). In practice, the decision for low latency usually results in a reduction of the available storage capacity (and vice versa). This is the reason why a computer has multiple memory types that are arranged hierarchically. The following pyramid illustrates the principle:



Computer memory latency and size hierarchy.

As you can see, the CPU and its ultra-fast (but small) registers used for short-term data storage reside at the top of the pyramid. Below are Cache and RAM, which belong to the category of temporary memory which quickly looses its content once power is cut off. Finally, there are permanent storage devices such as the ROM, hard drives as well as removable drives such as USB sticks.

Let us take a look at a typical computer usage scenario to see how the different types of memory are used:

1. After switching on the computer, it loads data from its read-only memory (ROM) and performs a power-on self-test (POST) to ensure that all major components are working properly. Additionally, the computer memory controller checks all of the memory addresses with a simple read/write operation to ensure that memory is functioning correctly.
2. After performing the self-test, the computer loads the basic input/output system (BIOS) from ROM. The major task of the BIOS is to make the computer functional by providing basic information about such things as storage devices, boot sequence, security or auto device recognition capability.
3. The process of activating a more complex system on a simple system is called "bootstrapping": It is a solution for the chicken-egg-problem of starting a software-driven system by itself using software. During bootstrapping, the computer loads the operating system (OS) from the hard drive into random access memory (RAM). RAM is considered "random access" because any memory cell can be accessed directly by intersecting the respective row and column in the matrix-like memory layout. For performance reasons, many parts of the OS are kept in RAM as long as the computer is powered on.
4. When an application is started, it is loaded into RAM. However, several application components are only loaded into RAM on demand to preserve memory. Files that are opened during runtime are also loaded into RAM. When a file is saved, it is written to the specified storage device. After closing the application, it is deleted from RAM.

This simple usage scenario shows the central importance of the RAM. Every time data is loaded or a file is opened, it is placed into this temporary storage area - but what about the other memory types above the RAM layer in the pyramid?

To maximize CPU performance, fast access to large amounts of data is critical. If the CPU cannot get the data it needs, it stops and waits for data availability. Thus, when designing new memory chips, engineers must adapt to the speed of the available CPUs. The problem they are facing is that memory which is able to keep up with modern CPUs running at several GHz is extremely expensive. To combat this, computer designers have created the memory tier system which has already been shown in the pyramid diagram above. The solution is to use expensive memory in small quantities and then back it up using larger quantities of less expensive memory.

The cheapest form of memory available today is the hard disk. It provides large quantities of inexpensive and permanent storage. The problem of a hard disk is its comparatively low speed - even though access times with modern solid state disks (SSD) have decreased significantly compared to older magnetic-disc models.

The next hierarchical level above hard disks or other external storage devices is the RAM. We will not discuss in detail how it works but only take a look at some key performance metrics of the CPU at this point, which place certain performance expectations on the RAM and its designers:

1. The **bit size** of the CPU decides how many bytes of data it can access in RAM memory at the same time. A 16-bit CPU can access 2 bytes (with each byte consisting of 8 bit) while a 64-bit CPU can access 8 bytes at a time.
2. The **processing speed** of the CPU is measured in Gigahertz or Megahertz and denotes the number of operations it can perform in one second.

From processing speed and bit size, the data rate required to keep the CPU busy can easily be computed by multiplying bit size with processing speed. With modern CPUs and ever-increasing speeds, the available RAM in the market will not be fast enough to match the CPU data rate requirements.

Virtual Memory

### Problems with physical memory

Virtual memory is a very useful concept in computer architecture because it helps with making your software work well given the configuration of the respective hardware on the computer it is running on.

The idea of virtual memory stems back from a (not so long ago) time, when the random access memory (RAM) of most computers was severely limited. Programers needed to treat memory as a precious resource and use it most efficiently. Also, they wanted to be able to run programs even if there was not enough RAM available. At the time of writing (August 2019), the amount of RAM is no longer a large concern for most computers and programs usually have enough memory available to them. But in some cases, for example when trying to do video editing or when running multiple large programs at the same time, the RAM memory can be exhausted. In such a case, the computer can slow down drastically.

There are several other memory-related problems, that programmers need to know about:

1. **Holes in address space** : If several programs are started one after the other and then shortly afterwards some of these are terminated again, it must be ensured that the freed-up space in between the remaining programs does not remain unused. If memory becomes too fragmented, it might not be possible to allocate a large block of memory due to a large-enough free contiguous block not being available any more.
2. **Programs writing over each other** : If several programs are allowed to access the same memory address, they will overwrite each others' data at this location. In some cases, this might even lead to one program reading sensitive information (e.g. bank account info) that was written by another program. This problem is of particular concern when writing concurrent programs which run several threads at the same time.

The basic idea of virtual memory is to separate the addresses a program may use from the addresses in physical computer memory. By using a mapping function, an access to (virtual) program memory can be redirected to a real address which is guaranteed to be protected from other programs.

In the following, you will see, how virtual memory solves the problems mentioned above and you will also learn about the concepts of memory pages, frames and mapping. A sound knowledge on virtual memory will help you understand the C++ memory model, which will be introduced in the next lesson of this course.

### Quiz

On a 32-bit machine, each program has its own 32-bit address space. When a program wants to access a memory location, it must specify a 32-bit address, which directs it to the byte stored at this location. On a hardware level, this address is transported to the physical memory via a parallel bus with 32 cables, i.e. each cable can either have the information 'high voltage', and 'low voltage' (or '1' and '0').

### 퀴즈 질문

How large is the address space on a 32-bit system? What is the upper limit for program memory in GB?

* 1 GB
* 2 GB
* 4 GB
* 8 GB

제출

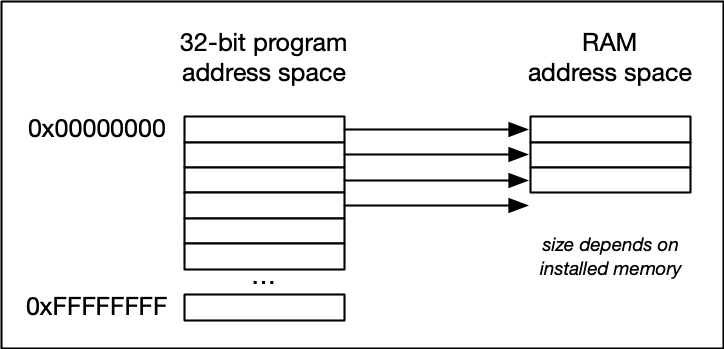
Correct! 2^32 bytes = 4GB; a 32-bit address space gives a program a (theoretical) total of 4 GB of memory it can address. In practice, the operating systems reserves some of this space however.

**julia>** 2^32 / (2^10)^3

4.0

### Expanding the available memory

As you have just learned in the quiz, the total amount of addressable memory is limited and depends on the architecture of the system (e.g. 32-bit). But what would happen if the available physical memory was below the upper bound imposed by the architecture? The following figure illustrates the problem for such a case:

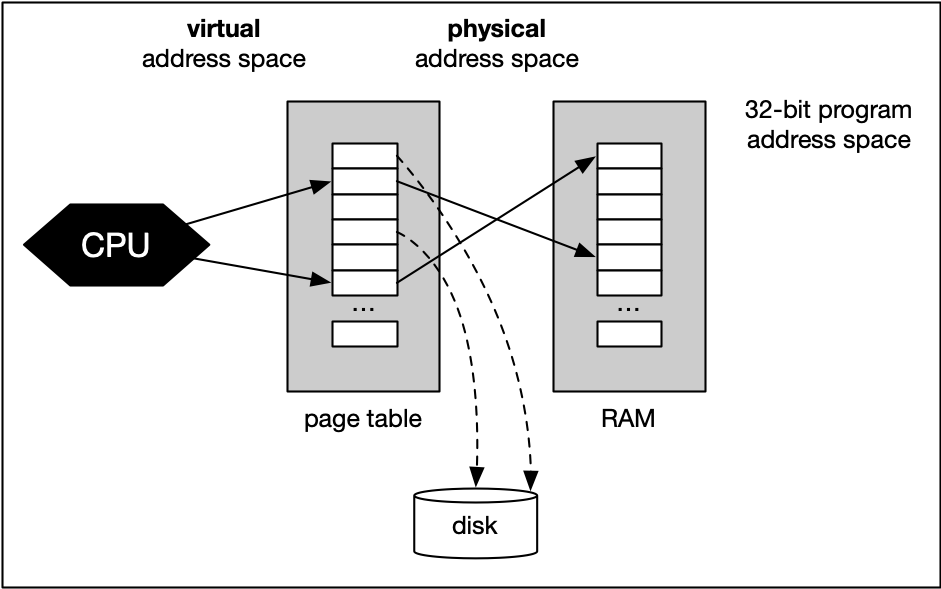


In the image above, the available physical memory is less than the upper bound provided by the 32-bit address space.

On a typical architecture such as MIPS ("Microprocessor without interlocked pipeline stages"), each program is promised to have access to an address space ranging from 0x00000000 up to 0xFFFFFFFF. If however, the available physical memory is only 1GB in size, a 1-on-1 mapping would lead to undefined behavior as soon as the 30-bit RAM address space were exceeded.

With virtual memory however, a mapping is performed between the virtual address space a program sees and the physical addresses of various storage devices such as the RAM but also the hard disk. Mapping makes it possible for the operating system to use physical memory for the parts of a process that are currently being used and back up the rest of the virtual memory to a secondary storage location such as the hard disk. With virtual memory, the size of RAM is not the limit anymore as the system hard disk can be used to store information as well.

The following figure illustrates the principle:



With virtual memory, the RAM acts as a cache for the virtual memory space which resides on secondary storage devices. On Windows systems, the file pagefile.sys is such a virtual memory container of varying size. To speed up your system, it makes sense to adjust the system settings in a way that this file is stored on an SSD instead of a slow magnetic hard drive, thus reducing the latency. On a Mac, swap files are stored in/private/var/vm/.

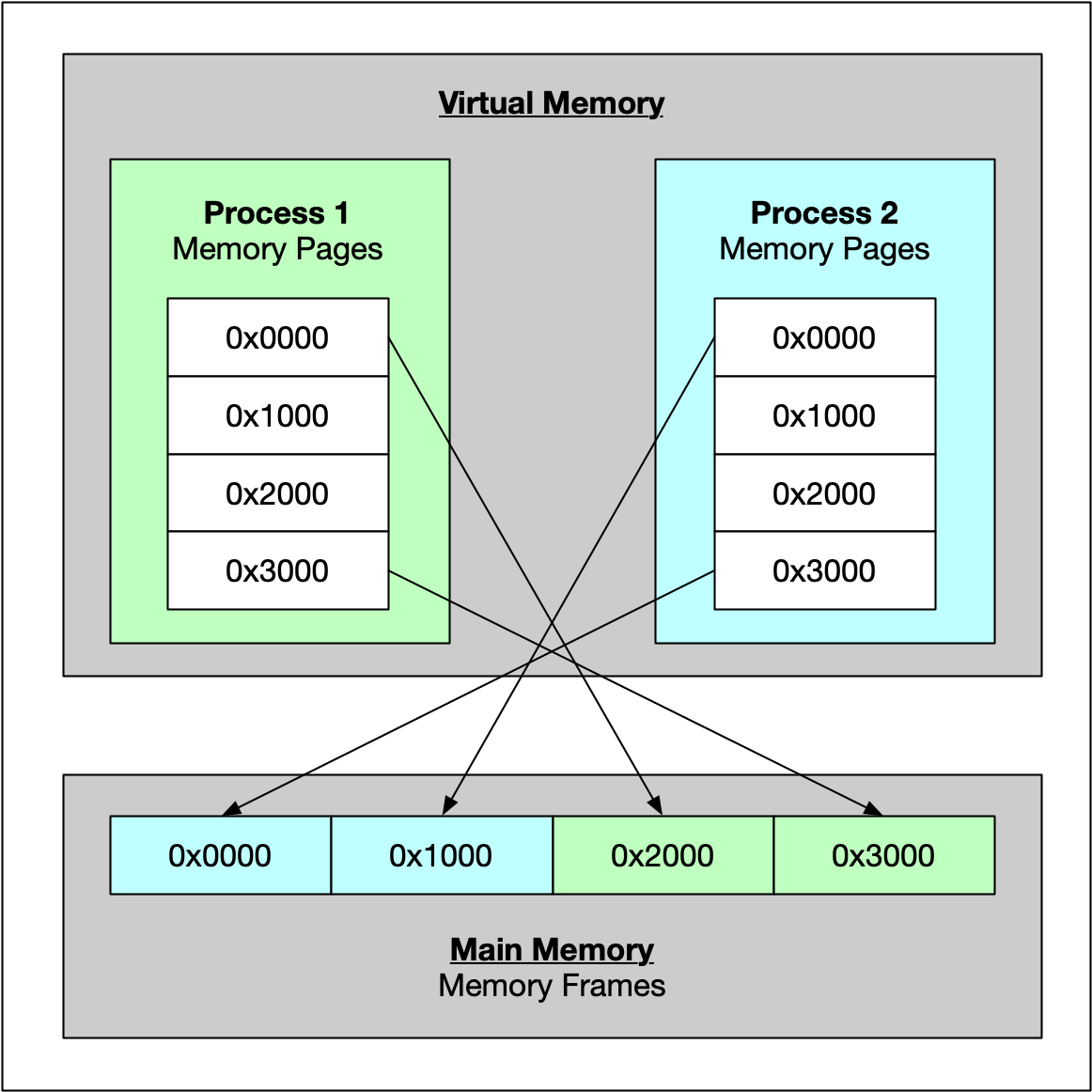
In a nutshell, virtual memory guarantees us a fixed-size address space which is largely independent of the system configuration. Also, the OS guarantees that the virtual address spaces of different programs do not interfere with each other.

The task of mapping addresses and of providing each program with its own virtual address space is performed entirely by the operating system, so from a programmer’s perspective, we usually don’t have to bother much about memory that is being used by other processes.

Before we take a closer look at an example though, let us define two important terms which are often used in the context of caches and virtual memory:

* A **memory page** is a number of directly successive memory locations in virtual memory defined by the computer architecture and by the operating system. The computer memory is divided into memory pages of equal size. The use of memory pages enables the operating system to perform virtual memory management. The entire working memory is divided into tiles and each address in this computer architecture is interpreted by the Memory Management Unit (MMU) as a logical address and converted into a physical address.
* A **memory frame** is mostly identical to the concept of a memory page with the key difference being its location in the physical main memory instead of the virtual memory.

The following diagram shows two running processes and a collection of memory pages and frames:

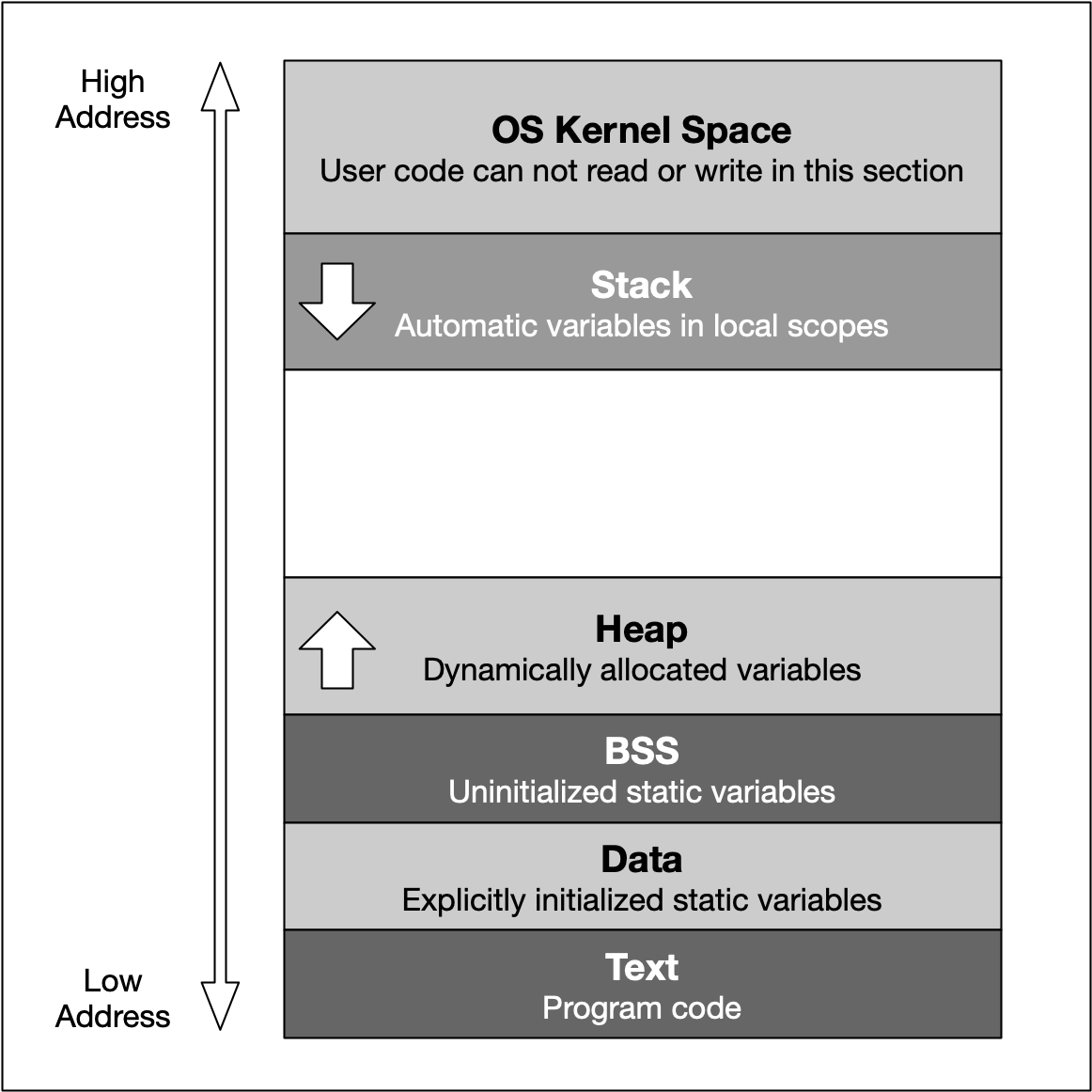


As can be seen, both processes have their own virtual memory space. Some of the pages are mapped to frames in the physical memory and some are not. If process 1 needs to use memory in the memory page that starts at address 0x1000, a page fault will occur if the required data is not there. The memory page will then be mapped to a vacant memory frame in physical memory. Also, note that the virtual memory addresses are not the same as the physical addresses. The first memory page of process 1, which starts at the virtual address 0x0000, is mapped to a memory frame that starts at the physical address 0x2000.

In summary, virtual memory management is performed by the operating system and programmers do usually not interfere with this process. The major benefit is a unique perspective on a chunk of memory for each program that is only limited in its size by the architecture of the system (32 bit, 64 bit) and by the available physical memory, including the hard disk.

## The Process Memory Model

As we have seen in the previous lesson, each program is assigned its own virtual memory by the operating system. This address space is arranged in a linear fashion with one block of data being stored at each address. It is also divided into several distinct areas as illustrated by the figure below:



The last address 0cFFFFFFFF converts to the decimal 4.294.967.295 , which is the total amount of memory blocks that can theoretically addressed in a 32 bit operating system - hence the well-known limit of 4GB of memory. On a 64 bit system, the available space is significantly (!) larger. Also, the addresses are stored with 8 bytes instead of 4 bytes.

From a programming perspective though, we are not able to use the entire address space. Instead, the blocks "OS Kernel Space" and "Text" are reserved for the operating system. In kernel space, only the most trusted code is executed - it is fully maintained by the operating system and serves as an interface between the user code and the system kernel. In this course, we will not be directly concerned with this part of memory. The section called 'text' holds the program code generated by the compiler and linker. As with the kernel space, we will not be using this block directly in this course. Let us now take a look at the remaining blocks, starting from the top:

1. The **stack** is a contiguous memory block with a fixed maximum size. If a program exceeds this size, it will crash. The stack is used for storing automatically allocated variables such as local variables or function parameters. If there are multiple threads in a program, then each thread has its own stack memory. New memory on the stack is allocated when the path of execution enters a scope and freed again once the scope is left. It is important to know that the stack is managed "automatically" by the compiler, which means we do not have to concern ourselves with allocation and deallocation.
2. The **heap** (also called "free store" in C++) is where data with dynamic storage lives. It is shared among multiple threads in a program, which means that memory management for the heap needs to take concurrency into account. This makes memory allocations in the heap more complicated than stack allocations. In general, managing memory on the heap is more (computationally) expensive for the operating system, which makes it slower than stack memory. Contrary to the stack, the heap is not managed automatically by the system, but by the programmer. If memory is allocated on the heap, it is the programmer’s responsibility to free it again when it is no longer needed. If the programmer manages the heap poorly or not at all, there will be trouble.
3. The **BSS** (Block Started by Symbol) segment is used in many compilers and linkers for a segment that contains global and static variables that are initialized with zero values. This memory area is suitable, for example, for arrays that are not initialized with predefined values.
4. The **Data** segment serves the same purpose as the BSS segment with the major difference being that variables in the Data segment have been initialized with a value other than zero. Memory for variables in the Data segment (and in BSS) is allocated once when a program is run and persists throughout its lifetime.

## Memory Allocation in C++

Now that we have an understanding of the available process memory, let us take a look at memory allocation in C++.

Not every variable in a program has a permanently assigned area of memory. The term **allocate** refers to the process of assigning an area of memory to a variable to store its value. A variable is **deallocated** when the system reclaims the memory from the variable, so it no longer has an area to store its value.

Generally, three basic types of memory allocation are supported:

1. **Static memory allocation** is performed for static and global variables, which are stored in the BSS and Data segment. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. **Automatic memory allocation** is performed for function parameters as well as local variables, which are stored on the stack. Memory for these types of variables is allocated when the path of execution enters a scope and freed again once the scope is left.
3. **Dynamic memory allocation** is a possibility for programs to request memory from the operating system at runtime when needed. This is the major difference between automatic and static allocation, where the size of the variable must be known at compile time. Dynamic memory allocation is not performed on the limited stack but on the heap and is thus (almost) only limited by the size of the address space.

From a programmer’s perspective, stack and heap are the most important areas of program memory. Hence, in the following lessons, let us look at these two in turn.

## Properties of Stack Memory

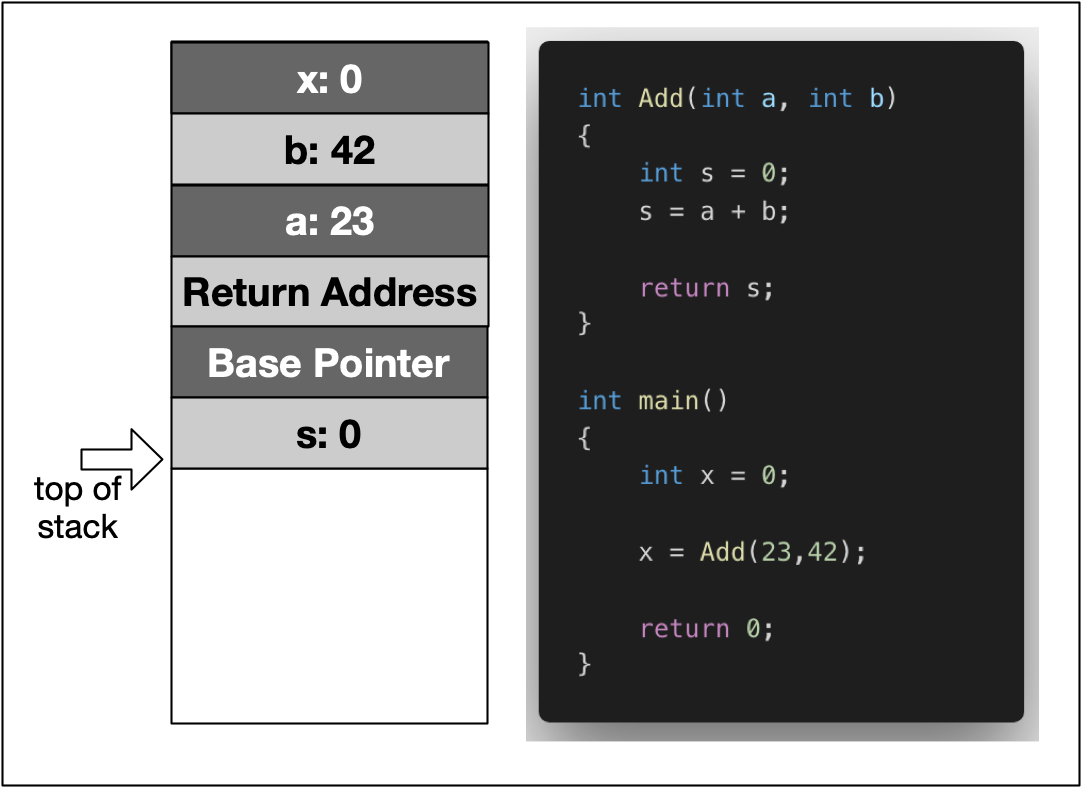
In the available literature on C++, the terms stack and heap are used regularly, even though this is not formally correct: C++ has the free space, storage classes and the storage duration of objects. However, since stack and heap are widely used in the C++ community, we will also use it throughout this course. Should you come across the above-mentioned terms in a book or tutorial on the subject, you now know that they refer to the same concepts as stack and heap do.

As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

In the following, a short list of key properties of the stack is listed:

1. The stack is a **contiguous block of memory**. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
2. When the **maximum size of the stack** memory is exceeded, a program will crash.
3. Allocating and deallocating **memory is fast** on the stack. It only involves moving the stack pointer to a new position.

The following diagram shows the stack memory during a function call:



In the example, the variable x is created on the stack within the scope of main. Then, a stack frame which represents the function Add and its variables is pushed to the stack, moving the stack pointer further downwards. It can be seen that this includes the local variables a and b, as well as the return address, a base pointer and finally the return value s.

In the following, let us dig a little more deeply and conduct some experiments with variables on the stack.

### Quiz: Why does CallByValue require more memory?

In this section, we have argued at length that passing a parameter by reference avoids a costly copy and should - in many situations - be preferred over passing a parameter by value. Yet, in the experiment above, we have witnessed the exact opposite.

Can you explain why?

HIDE SOLUTION

Let us take a look at the size of the various parameter types using the sizeof command:

printf("size of int: %lu\n", sizeof(int));

printf("size of \*int: %lu\n", sizeof(int \*));

The output here is

size of int: 4

size of \*int: 8

Obviously, the size of the pointer variable is larger than the actual data type. As my machine has a 64 bit architecture, an address requires 8 byte.

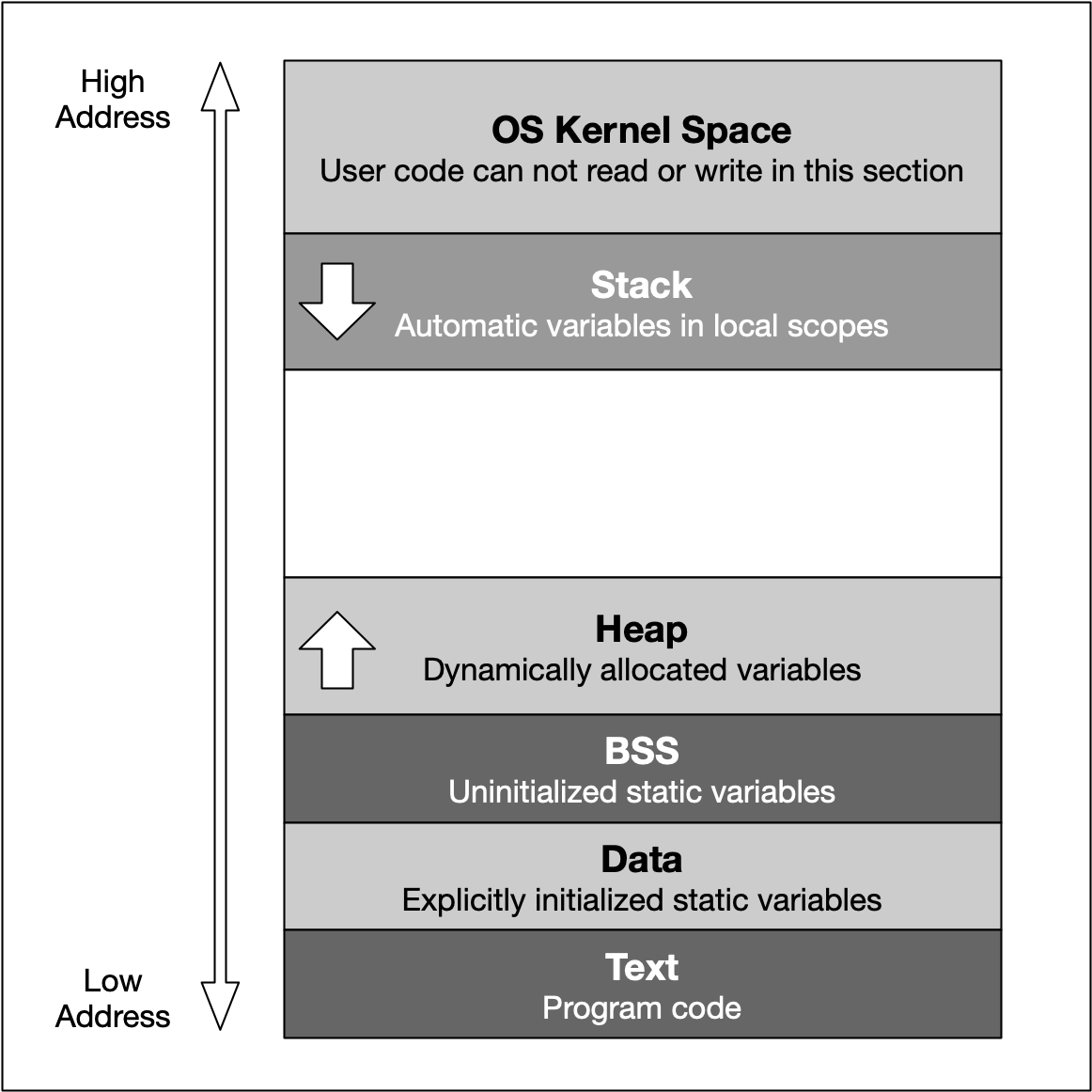
As an experiment, you could use the -m32 compiler flag to build a 32 bit version of the program. This yields the following output:

size of int: 4

size of \*int: 4

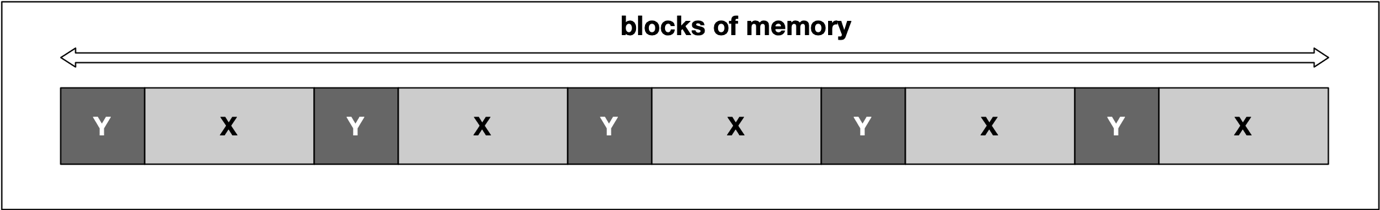
In order to benefit from call-by-reference, the size of the data type passed to the function has to surpass the size of the pointer on the respective architecture (i.e. 32 bit or 64 bit).

**Properties of Heap Memory**

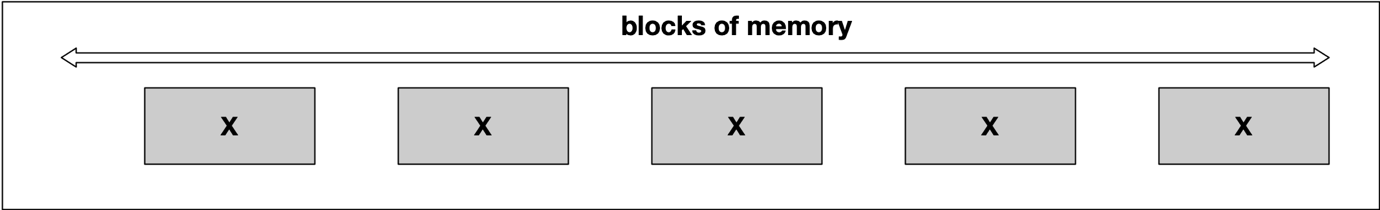


**Memory Fragmentation**

Let us construct a theoretic example of how memory on the heap can become fragmented: Suppose we are interleaving the allocation of two data types X and Y in the following fashion: First, we allocate a block of memory for a variable of type X, then another block for Y and so on in a repeated manner until some upper bound is reached. At the end of this operation, the heap might look like the following:



At some point, we might then decide to deallocate all variables of type Y, leading to empty spaces in between the remaining variables of type X. In between two blocks of type "X", no memory for an additional "X" could now be squeezed in this example.



A classic symptom of memory fragmentation is that you try to allocate a large block and you can’t, even though you appear to have enough memory free. On systems with virtual memory however, this is less of a problem, because large allocations only need to be contiguous in virtual address space, not in physical address space.

When memory is heavily fragmented however, memory allocations will likely take longer because the memory allocator has to do more work to find a suitable space for the new object.

Until now, our examples have been only theoretical. It is time to gain some practical experience in the next section using malloc and free as C-style methods for dynamic memory management.

## Overview of memory management problems

One of the primary advantages of C++ is the flexibility and control of resources such as memory it gives to the programmer. This advantage is further amplified by a significant increase in the performance of C++ programs compared to other languages such as Python or Java.

However, these advantages come at a price as they demand a high level of experience from the programer. As Bjarne Stroustrup put it so elegantly:

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".

In this chapter, we will look at a collection of typical errors in memory management that you need to watch out for.

1. **Memory Leaks** Memory leaks occur when data is allocated on the heap at runtime, but not properly deallocated. A program that forgets to clear a memory block is said to have a memory leak - this may be a serious problem or not, depending on the circumstances and on the nature of the program. For a program that runs, computes something, and quits immediately, memory leaks are usually not a big concern. Memory leaks are mostly problematic for programs that run for a long time and/or use large data structures. In such a case, memory leaks can gradually fill the heap until allocation requests can no longer be properly met and the program stops responding or crashes completely. We will look at an example further down in this section.
2. **Buffer Overruns** Buffer overruns occur when memory outside the allocated limits is overwritten and thus corrupted. One of the resulting problems is that this effect may not become immediately visible. When a problem finally does occur, cause and effect are often hard to discern. It is also sometimes possible to inject malicious code into programs in this way, but this shall not be discussed here.

In this example, the allocated stack memory is too small to hold the entire string, which results in a segmentation fault:

**char** str[5];

strcpy(str,"BufferOverrun");

printf("%s",str);

1. **Uninitialized Memory** Depending on the C++ compiler, data structures are sometimes initialized (most often to zero) and sometimes not. So when allocating memory on the heap without proper initialization, it might sometimes contain garbage that can cause problems.

Generally, a variable will be automatically initialized in these cases:

* + it is a class instance where the default constructor initializes all primitive types
  + array initializer syntax is used, such as int a[10] = {}
  + it is a global or extern variable
  + it is defined static

The behavior of the following code is potentially undefined:

**int** a;

**int** b=a\*42;

printf("%d",b);

1. **Incorrect pairing of allocation and deallocation** Freeing a block of memory more than once will cause a program to crash. This can happen when a block of memory is freed that has never been allocated or has been freed before. Such behavior could also occur when improper pairings of allocation and deallocation are used such as using malloc() with delete or new with free().

In this first example, the wrong new and delete are paired

**double** \*pDbl=**new** **double**[5];

**delete** pDbl;

In this second example, the pairing is correct but a double deletion is performed:

**char** \*pChr=**new** **char**[5];

**delete**[] pChr;

**delete**[] pChr;

1. **Invalid memory access** This error occurs then trying to access a block of heap memory that has not yet or has already been deallocated.

In this example, the heap memory has already been deallocated at the time when strcpy() tries to access it:

**char** \*pStr=**new** **char**[25];

**delete**[] pStr;

strcpy(pStr, "Invalid Access");

Copy Semantics

## The Rule of Three

In the previous examples we have taken a first look at several copying policies:

1. Default copying
2. No copying
3. Exclusive ownership
4. Deep copying
5. Shared ownership

In the first example we have seen that the default implementation of the copy constructor does not consider the "special" needs of a class which allocates and deallocates a shared resource on the heap. The problem with implicitly using the default copy constructor or assignment operator is that programmers are not forced to consider the implications for the memory management policy of their program. In the case of the first example, this leads to a segmentation fault and thus a program crash.

In order to properly manage memory allocation, deallocation and copying behavior, we have seen that there is an intricate relationship between destructor, copy constructor and copy assignment operator. To this end, the **Rule of Three** states that if a class needs to have an overloaded copy constructor, copy assignment operator, ~or~ destructor, then it must also implement the other two as well to ensure that memory is managed consistently. As we have seen, the copy constructor and copy assignment operator (which are often almost identical) control how the resource gets copied between objects while the destructor manages the resource deletion.

You may have noted that in the previous code example, the class SharedCopy does not implement the assignment operator. This is a violation of the **Rule of Three** and thus, if we were to use something like destination3 = source instead of SharedCopy destination3(source), the counter variable would not be properly decremented.

The copying policies discussed in this chapter are the basis for a powerful concept in C++11 - smart pointers. But before we discuss these, we need to go into further detail on move semantics, which is a prerequisite you need to learn more about so you can properly understand the exclusive ownership policy as well as the Rule of Five, both of which we will discuss very soon. But before we discuss move semantics, we need to look into the concept of lvalues and rvalues in the next section.

Lvalues and Rvalues

## What are lvalues and rvalues?

A good grasp of lvalues and rvalues in C++ is essential for understanding the more advanced concepts of rvalue references and motion semantics.

Let us start by stating that every expression in C++ has a type and belongs to a value category. When objects are created, copied or moved during the evaluation of an expression, the compiler uses these value expressions to decide which method to call or which operator to use.

Prior to C++11, there were only two value categories, now there are as many as five of them:

To keep it short, we do not want to go into all categories, but limit ourselves to lvalues and prvalues:

* **Lvalues** have an address that can be accessed. They are expressions whose evaluation by the compiler determines the identity of objects or functions.
* **Prvalues** do not have an address that is accessible directly. They are temporary expressions used to initialize objects or compute the value of the operand of an operator.

For the sake of simplicity and for compliance with many tutorials, videos and books about the topic, let us refer to prvalues as rvalues from here on.

The two characters l and r are originally derived from the perspective of the assignment operator =, which always expects a rvalue on the right, and which it assigns to a lvalue on the left. In this case, the l stands for left and r for right:

int i = 42; // lvalue = rvalue;

With many other operators, however, this right-left view is not entirely correct. In more general terms, an lvalue is an entity that points to a specific memory location. An rvalue is usually a short-lived object, which is only needed in a narrow local scope. To simplify things a little, one could think of lvalues as named containers for rvalues.

In the example above, the value 42 is an rvalue. It does not have a specific memory address which we know about. The rvalue is assigned to a variable i with a specific memory location known to us, which is what makes it an lvalue in this example.

Using the address operator & we can generate an lvalue from an rvalue and assign it to another lvalue:

int \*j = &i;

In this small example, the expression &i generates the address of i as an rvalue and assigns it to j, which is an lvalue now holding the memory location of i.

The code on the right illustrates several examples of lvalues and rvalues:

## Lvalue references

An lvalue reference can be considered as an alternative name for an object. It is a reference that binds to an lvalue and is declared using an optional list of specifiers (which we will not further discuss here) followed by the reference declarator &. The short code sample on the right declares an integer i and a reference j which can be used as an alias for the existing object.

The output of the program is  
i = 3, j = 3

We can see that the lvalue reference j can be used just as i can. A change to either i or j will affect the same memory location on the stack.

#include <iostream>

int main()

{

int i = 1;

int &j = i;

++i;

++j;

std::cout << "i = " << i << ", j = " << j << std::endl;

return 0;

}

## Rvalue references

You already know that an rvalue is a temporary expression which is - among other use-cases, a means of initializing objects. In the call int i = 42, 42 is the rvalue.

Let us consider an example similar to the last one, shown on the right.

As before, the function myFunction takes an lvalue reference as its argument. In main, the call myFunction(j) works just fine while myFunction(42) as well as myFunction(j+k) produces the following compiler error on Mac:

candidate function not viable: expects an l-value for 1st argument

and the following error in the workspace with g++:

error: cannot bind non-const lvalue reference of type ‘int&’ to an rvalue of type ‘int’

While the number 42 is obviously an rvalue, with j+k things might not be so obvious, as j and k are variables and thus lvalues. To compute the result of the addition, the compiler has to create a temporary object to place it in - and this object is an rvalue.

에러가 발생한다.

#include <iostream>

void myFunction(int &val)

{

std::cout << "val = " << val << std::endl;

}

int main()

{

int j = 42;

myFunction(j);

// myFunction(42);

int k = 23;

myFunction(j+k);

return 0;

}

Since C++11, there is a new type available called rvalue reference, which can be identified from the double ampersand && after a type name. With this operator, it is possible to store and even modify an rvalue, i.e. a temporary object which would otherwise be lost quickly.

But what do we need this for? Before we look into the answer to this question, let us consider the example on the right.

Rvalue를 access & modify 다 된다.

#include <iostream>

int main()

{

int i = 1;

int j = 2;

int k = i + j;

int &&l = i + j;

std::cout << "k = " << k << ", l = " << l << std::endl;

return 0;

}

After creating the integers i and j on the stack, the sum of both is added to a third integer k. Let us examine this simple example a little more closely. In the first and second assignment, i and j are created as lvalues, while 1 and 2 are rvalues, whose value is copied into the memory location of i and j. Then, a third lvalue, k, is created. The sum i+j is created as an rvalue, which holds the result of the addition before being copied into the memory location of k. This is quite a lot of copying and holding of temporary values in memory. With an rvalue reference, this can be done more efficiently.

The expression int &&l creates an rvalue reference, to which the address of the temporary object is assigned, that holds the result of the addition. So instead of first creating the rvalue i+j , then copying it and finally deleting it, we can now hold the temporary object in memory. This is much more efficient than the first approach, even though saving a few bytes of storage in the example might not seem like much at first glance. One of the most important aspects of rvalue references is that they pave the way for move semantics, which is a mighty technique in modern C++ to optimize memory usage and processing speed. Move semantics and rvalue references make it possible to write code that transfers resources such as dynamically allocated memory from one object to another in a very efficient manner and also supports the concept of exclusive ownership, as we will shortly see when discussing smart pointers. In the next section we will take a close look at move semantics and its benefits for memory management.

Move Semantics

int i **=** 23;

myFunction(i)

would result in a compiler error. There is a solution to this problem though: The function std::move converts an lvalue into an rvalue (actually, to be exact, into an xvalue, which we will not discuss here for the sake of clarity), which makes it possible to use the lvalue as an argument for the function:

int i **=** 23;

myFunction(std::move(i));

In doing this, we state that in the scope of main we will not use i anymore, which now exists only in the scope of myFunction. Using std::move in this way is one of the components of move semantics, which we will look into shortly. But first let us consider an example of the **Rule of Three**.

#include <iostream>

void myFunction(int &&val)

{

std::cout << "val = " << val << std::endl;

}

int main()

{

myFunction(42);

return 0;

}

## Moving lvalues

There is a solution to this problem in C++, which is std::move. This function accepts an lvalue argument and returns it as an rvalue without triggering copy construction. So by passing an object to std::move we can force the compiler to use move semantics, either in the form of move constructor or the move assignment operator:

int main()

{

MyMovableClass obj1(100); *// constructor*

useObject(std::move(obj1));

**return** 0;

}

**Rvalue references and std::move**

In order to fully understand the concept of smart pointers in the next lesson, we first need to take a look at a powerful concept introduced with C++11 called *move semantics*.

The last section on lvalues, rvalues and especially rvalue references is an important prerequisite for understanding the concept of moving data structures.

Let us consider the function on the right which takes an rvalue reference as its parameter.

The important message of the function argument of myFunction to the programmer is : The object that binds to the rvalue reference &&val is yours, it is not needed anymore within the scope of the caller (which is main). As discussed in the previous section on rvalue references, this is interesting from two perspectives:

1. Passing values like this **improves performance** as no temporary copy needs to be made anymore and
2. **ownership changes**, since the object the reference binds to has been abandoned by the caller and now binds to a handle which is available only to the receiver. This could not have been achieved with lvalue references as any change to the object that binds to the lvalue reference would also be visible on the caller side.

#include <iostream>

void myFunction(int &&val)

{

std::cout << "val = " << val << std::endl;

}

int main()

{

myFunction(42);

return 0;

}

There is one more important aspect we need to consider: rvalue references are themselves lvalues. While this might seem confusing at first glance, it really is the mechanism that enables move semantics: A reference is always defined in a certain context (such as in the above example the variable val) . Even though the object it refers to (the number 42) may be disposable in the context it has been created (the main function), it is not disposable in the context of the reference . So within the scope of myFunction, val is an lvalue as it gives access to the memory location where the number 42 is stored.

Note however that in the above code example we cannot pass an lvalue to myFunction, because an rvalue reference cannot bind to an lvalue. The code

int i **=** 23;

myFunction(i)

would result in a compiler error. There is a solution to this problem though: The function std::move converts an lvalue into an rvalue (actually, to be exact, into an xvalue, which we will not discuss here for the sake of clarity), which makes it possible to use the lvalue as an argument for the function:

int i **=** 23;

myFunction(std::move(i));

In doing this, we state that in the scope of main we will not use i anymore, which now exists only in the scope of myFunction. Using std::move in this way is one of the components of move semantics, which we will look into shortly. But first let us consider an example of the **Rule of Three**.

#include <stdlib.h>

#include <iostream>

class MyMovableClass

{

private:

int \_size;

int \*\_data;

public:

MyMovableClass(size\_t size) // constructor

{

\_size = size;

\_data = new int[\_size];

std::cout << "CREATING instance of MyMovableClass at " << this << " allocated with size = " << \_size\*sizeof(int) << " bytes" << std::endl;

}

~MyMovableClass() // 1 : destructor

{

std::cout << "DELETING instance of MyMovableClass at " << this << std::endl;

delete[] \_data;

}

MyMovableClass &operator=(const MyMovableClass &source) // 3 : copy assignment operator

{

std::cout << "ASSIGNING content of instance " << &source << " to instance " << this << std::endl;

if (this == &source)

return \*this;

delete[] \_data;

\_data = new int[source.\_size];

\*\_data = \*source.\_data;

\_size = source.\_size;

return \*this;

}

};

We can now use our class to copy objects as shown in the following implementation of main:

int main()

{

MyMovableClass obj1(10); *// regular constructor*

MyMovableClass obj2(obj1); *// copy constructor*

obj2 **=** obj1; *// copy assignment operator*

**return** 0;

}

Add this code to the rule\_of\_three.cpp file on the right.

In the main above, the object obj1 is created using the regular constructor of MyMovableClass. Then, both the copy constructor as well as the assignment operator are used with the latter one not creating a new object but instead assigning the content of obj1 to obj2 as defined by our copying policy.

The output of this textbook implementation of the **Rule of Three** looks like this:

CREATING instance of MyMovableClass at 0x7ffeefbff618 allocated with size = 40 bytes

COPYING content of instance 0x7ffeefbff618 to instance 0x7ffeefbff608

ASSIGNING content of instance 0x7ffeefbff618 to instance 0x7ffeefbff608

DELETING instance of MyMovableClass at 0x7ffeefbff608

DELETING instance of MyMovableClass at 0x7ffeefbff618

### Limitations of Our Current Class Design

Let us now consider one more way to instantiate MyMovableClass object by using createObject() function. Add the following function definition to the rule\_of\_three.cpp, outside the scope of the class MyMovableClass:

MyMovableClass createObject(int size){

MyMovableClass obj(size); *// regular constructor*

**return** obj; *// return MyMovableClass object by value*

}

**Note that when a function returns an object by value, the compiler creates a temporary object as an rvalue.** Let's call this function inside main to create an obj4 instance, as follows:

int main(){

*// call to copy constructor, (alternate syntax)*

MyMovableClass obj3 **=** obj1;

*// Here, we are instantiating obj3 in the same statement; hence the copy assignment operator would not be called.*

MyMovableClass obj4 **=** createObject(10);

*// createObject(10) returns a temporary copy of the object as an rvalue, which is passed to the copy constructor.*

*/\**

*\* You can try executing the statement below as well*

*\* MyMovableClass obj4(createObject(10));*

*\*/*

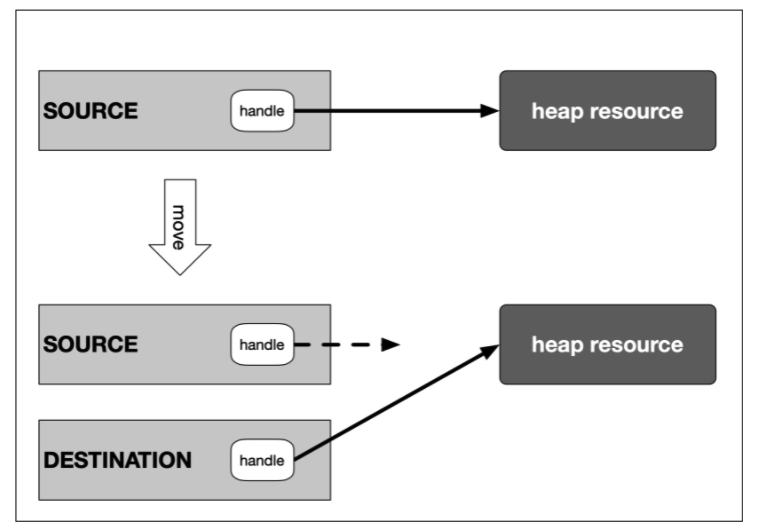
**return** 0;

}

## The move constructor

The basic idea to optimize the code from the last example is to "steal" the rvalue generated by the compiler during the return-by-value operation and move the expensive data in the source object to the target object - not by copying it but by redirecting the data handles. Moving data in such a way is always cheaper than making copies, which is why programmers are highly encouraged to make use of this powerful tool.

The following diagram illustrates the basic principle of moving a resource from a source object to a destination object:



Just like the copy constructor, the move constructor builds an instance of a class using a source instance. The key difference between the two is that with the move constructor, the source instance will no longer be usable afterwards. Let us take a look at an implementation of the move constructor for our MyMovableClass:

MyMovableClass(MyMovableClass **&&**source) *// 4 : move constructor*

{

std::cout **<<** "MOVING (c’tor) instance " **<<** **&**source **<<** " to instance " **<<** **this** **<<** std::endl;

\_data **=** source.\_data;

\_size **=** source.\_size;

source.\_data **=** nullptr;

source.\_size **=** 0;

}

The move assignment operator works in a similar way:

MyMovableClass **&operator=**(MyMovableClass **&&**source) *// 5 : move assignment operator*

{

std::cout **<<** "MOVING (assign) instance " **<<** **&**source **<<** " to instance " **<<** **this** **<<** std::endl;

**if** (**this** **==** **&**source)

**return** **\*this**;

**delete**[] \_data;

\_data **=** source.\_data;

\_size **=** source.\_size;

source.\_data **=** nullptr;

source.\_size **=** 0;

**return** **\*this**;

}

**The Rule of Five**

The five functions are:

1. The **destructor**: Responsible for freeing the resource once the object it belongs to goes out of scope.
2. The **assignment operator**: The default assignment operation performs a member-wise shallow copy, which does not copy the content behind the resource handle. If a deep copy is needed, it has be implemented by the programmer.
3. The **copy constructor**: As with the assignment operator, the default copy constructor performs a shallow copy of the data members. If something else is needed, the programmer has to implement it accordingly.
4. The **move constructor**: Because copying objects can be an expensive operation which involves creating, copying and destroying temporary objects, rvalue references are used to bind to an rvalue. Using this mechanism, the move constructor transfers the ownership of a resource from a (temporary) rvalue object to a permanent lvalue object.
5. The **move assignment operator**: With this operator, ownership of a resource can be transferred from one object to another. The internal behavior is very similar to the move constructor.

## Moving lvalues

There is one final aspect we need to look at: In some cases, it can make sense to treat lvalues like rvalues. At some point in your code, you might want to transfer ownership of a resource to another part of your program as it is not needed anymore in the current scope. But instead of copying it, you want to just move it as we have seen before. The "problem" with our implementation of MyMovableClass is that the call useObject(obj1) will trigger the copy constructor as we have seen in one of the last examples. But in order to move it, we would have to pretend to the compiler that obj1 was an rvalue instead of an lvalue so that we can make an efficient move operation instead of an expensive copy.

There is a solution to this problem in C++, which is std::move. This function accepts an lvalue argument and returns it as an rvalue without triggering copy construction. So by passing an object to std::move we can force the compiler to use move semantics, either in the form of move constructor or the move assignment operator:

int main()

{

MyMovableClass obj1(100); *// constructor*

useObject(std::move(obj1));

**return** 0;

}

Nothing much has changed, apart from obj1 being passed to the std::move function. The output would look like the following:

CREATING instance of MyMovableClass at 0x7ffeefbff718 allocated with size = 400 bytes

MOVING (c'tor) instance 0x7ffeefbff718 to instance 0x7ffeefbff708

using object 0x7ffeefbff708

DELETING instance of MyMovableClass at 0x7ffeefbff708

DELETING instance of MyMovableClass at 0x7ffeefbff718

By using std::move, we were able to pass the ownership of the resources within obj1 to the function useObject. The local copy obj1 in the argument list was created with the move constructor and thus accepted the ownership transfer from obj1 to obj . Note that after the call to useObject, the instance obj1 has been invalidated by setting its internal handle to null and thus may not be used anymore within the scope of main (even though you could theoretically try to access it, but this would be a really bad idea).

## The benefits of smart pointers

To put it briefly: Smart pointers were introduced in C++ to solve the above mentioned problems by providing a degree of automatic memory management: When a smart pointer is no longer needed (which is the case as soon as it goes out of scope), the memory to which it points is automatically deallocated. When contrasted with smart pointers, the conventional pointers we have seen so far are often termed "raw pointers".

In essence, smart pointers are classes that are wrapped around raw pointers. By overloading the -> and \* operators, smart pointer objects make sure that the memory to which their internal raw pointer refers to is properly deallocated. This makes it possible to use smart pointers with the same syntax as raw pointers. As soon as a smart pointer goes out of scope, its destructor is called and the block of memory to which the internal raw pointer refers is properly deallocated. This technique of wrapping a management class around a resource has been conceived by Bjarne Stroustroup and is called **Resource Acquisition Is Initialization (RAII)**. Before we continue with smart pointers and their usage let us take a close look at this powerful concept.

**Resource Acquisition Is Initialization**

테이블이(가) 표시된 사진

자동 생성된 설명

## Smart pointer overview

Since C++11, the standard library includes smart pointers, which help to ensure that programs are free of memory leaks while also remaining exception-safe. With smart pointers, resource acquisition occurs at the same time that the object is initialized (when instantiated with make\_shared or make\_unique), so that all resources for the object are created and initialized in a single line of code.

In modern C++, raw pointers managed with new and delete should only be used in small blocks of code with limited scope, where performance is critical (such as with placement new) and ownership rights of the memory resource are clear. We will look at some guidelines on where to use which pointer later.

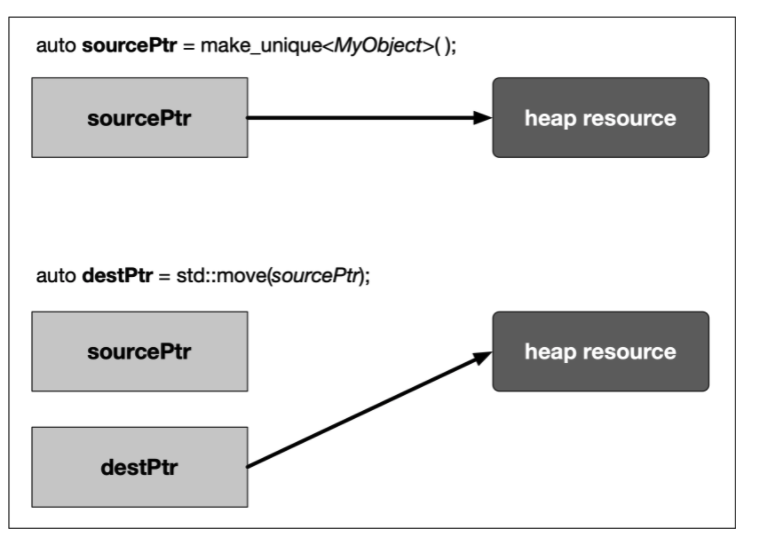
C++11 has introduced three types of smart pointers, which are defined in the header of the standard library:

1. The **unique pointer** std::unique\_ptr is a smart pointer which exclusively owns a dynamically allocated resource on the heap. There must not be a second unique pointer to the same resource.
2. The **shared pointer** std::shared\_ptr points to a heap resource but does not explicitly own it. There may even be several shared pointers to the same resource, each of which will increase an internal reference count. As soon as this count reaches zero, the resource will automatically be deallocated.
3. The **weak pointer** std::weak\_ptr behaves similar to the shared pointer but does not increase the reference counter.

## The Unique pointer

A unique pointer is the exclusive owner of the memory resource it represents. There must not be a second unique pointer to the same memory resource, otherwise there will be a compiler error. As soon as the unique pointer goes out of scope, the memory resource is deallocated again. Unique pointers are useful when working with a temporary heap resource that is no longer needed once it goes out of scope.

The following diagram illustrates the basic idea of a unique pointer:



In the example, a resource in memory is referenced by a unique pointer instance sourcePtr. Then, the resource is reassigned to another unique pointer instance destPtr using std::move. The resource is now owned by destPtr while sourcePtr can still be used but does not manage a resource anymore.

A unique pointer is constructed using the following syntax:

#include <memory>

void RawPointer()

{

int \*raw = new int; // create a raw pointer on the heap

\*raw = 1; // assign a value

delete raw; // delete the resource again

}

void UniquePointer()

{

std::unique\_ptr<int> unique(new int); // create a unique pointer on the stack

\*unique = 2; // assign a value

// delete is not neccessary

}

#include <iostream>

#include <memory>

#include <string>

class MyClass

{

public:

std::string \_text;

MyClass() {}

MyClass(std::string text) { \_text = text; }

~MyClass() { std::cout << \_text << " destroyed" << std::endl; }

void setText(std::string text) { \_text = text; }

};

int main()

{

// create unique pointer to proprietary class

std::unique\_ptr<MyClass> myClass1(new MyClass());

std::unique\_ptr<MyClass> myClass2(new MyClass("String 2"));

// call member function using ->

myClass1->setText("String 1");

// use the dereference operator \*

\*myClass1 = \*myClass2;

// use the .get() function to retrieve a raw pointer to the object

std::cout << "Objects have stack addresses " << myClass1.get() << " and " << myClass2.get() << std::endl;

std::cout << myClass1->\_text << std::endl;

std::cout << myClass2->\_text << std::endl;

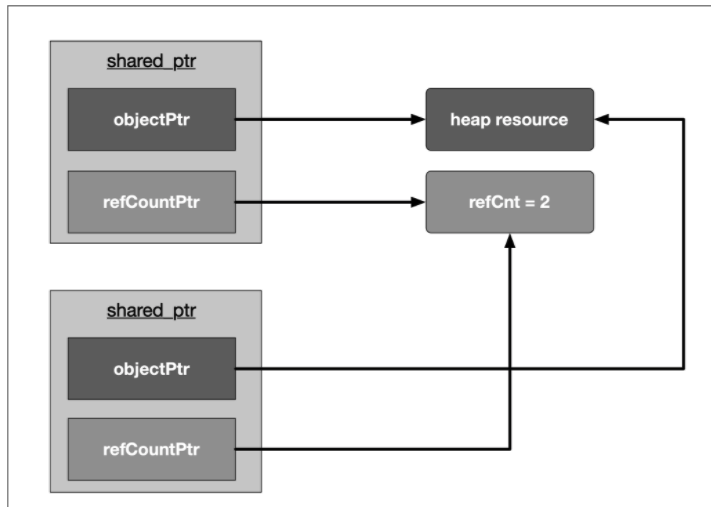
return 0;

}

## The Shared Pointer

Just as the unique pointer, a shared pointer owns the resource it points to. The main difference between the two smart pointers is that shared pointers keep a reference counter on how many of them point to the same memory resource. Each time a shared pointer goes out of scope, the counter is decreased. When it reaches zero (i.e. when the last shared pointer to the resource is about to vanish). the memory is properly deallocated. This smart pointer type is useful for cases where you require access to a memory location on the heap in multiple parts of your program and you want to make sure that whoever owns a shared pointer to the memory can rely on the fact that it will be accessible throughout the lifetime of that pointer.

The following diagram illustrates the basic idea of a shared pointer:



#include <iostream>

#include <memory>

class MyClass

{

public:

~MyClass() { std::cout << "Destructor of MyClass called" << std::endl; }

};

int main()

{

std::shared\_ptr<MyClass> shared(new MyClass);

std::cout << "shared pointer count = " << shared.use\_count() << std::endl;

shared.reset(new MyClass);

std::cout << "shared pointer count = " << shared.use\_count() << std::endl;

return 0;

}

**The Weak Pointer**

Similar to shared pointers, there can be multiple weak pointers to the same resource. The main difference though is that weak pointers do not increase the reference count. Weak pointers hold a non-owning reference to an object that is managed by another shared pointer.

The following rule applies to weak pointers: You can only create weak pointers out of shared pointers or out of another weak pointer. The code on the right shows a few examples of how to use and how not to use weak pointers.

The output looks as follows:

shared pointer count = 1

shared pointer count = 1

#include <iostream>

#include <memory>

int main()

{

std::shared\_ptr<int> mySharedPtr(new int);

std::cout << "shared pointer count = " << mySharedPtr.use\_count() << std::endl;

std::weak\_ptr<int> myWeakPtr1(mySharedPtr);

std::weak\_ptr<int> myWeakPtr2(myWeakPtr1);

std::cout << "shared pointer count = " << mySharedPtr.use\_count() << std::endl;

// std::weak\_ptr<int> myWeakPtr3(new int); // COMPILE ERROR

return 0;

}

#include <iostream>

#include <memory>

int main()

{

std::shared\_ptr<int> mySharedPtr(new int);

std::weak\_ptr<int> myWeakPtr(mySharedPtr);

mySharedPtr.reset(new int);

if (myWeakPtr.expired() == true)

{

std::cout << "Weak pointer expired!" << std::endl;

}

return 0;

}

## Passing smart pointers to functions

Let us consider the following recommendation of the C++ guidelines on smart pointers:

The following examples are **pass-by-value types that lend the ownership** of the underlying object:

1. void f(std::unique\_ptr<MyObject> ptr)
2. void f(std::shared\_ptr<MyObject> ptr)
3. void f(std::weak\_ptr<MyObject> ptr)

#include <iostream>

#include <memory>

class MyClass

{

private:

int \_member;

public:

MyClass(int val) : \_member{val} {}

void printVal() { std::cout << ", managed object " << this << " with val = " << \_member << std::endl; }

};

void f(std::unique\_ptr<MyClass> ptr)

{

std::cout << "unique\_ptr " << &ptr;

ptr->printVal();

}

int main()

{

std::unique\_ptr<MyClass> uniquePtr = std::make\_unique<MyClass>(23);

std::cout << "unique\_ptr " << &uniquePtr;

uniquePtr->printVal();

f(std::move(uniquePtr));

if (uniquePtr)

uniquePtr->printVal();

return 0;

}

Practice

/\*

Smart pointer exercises: Handling unique, shared and smart pointers

// If all tasks are solved properly, the following text should appear in the terminal

Learn Coding with Udacity!

weak pointer is expired

Note: Compile with C++17

\*/

#include <string>

#include <iostream>

#include <memory>

void f1(std::unique\_ptr<std::string> unique\_ptr)

{

// TASK 3: Print the content of unique\_ptr to the terminal

// SOLUTION 3:

std::cout << \*unique\_ptr;

}

void f2(std::shared\_ptr<std::string> shared\_ptr)

{

// TASK 4: Print the use count property of shared\_ptr to the terminal to see how many pointers refer to its resource

// If the use count is 2, print the content of shared\_ptr to the terminal

// SOLUTION 4:

if(shared\_ptr.use\_count()==2)

std::cout << \*shared\_ptr;

}

void f3(std::weak\_ptr<std::string> weak\_ptr)

{

// TASK 5: Lock the weak pointer by assigning it to a shared pointer. Then, print its content to the terminal.

// If the weak ptr can not be locked because the resource it refers to has expired, print the string "weak pointer is expired" to the terminal.

// SOLUTION 5:

if (auto shared\_ptr = weak\_ptr.lock()) // // Copy into a shared\_ptr to use it

{

std::cout << \*shared\_ptr << "\n";

}

else

{

std::cout << "weak pointer is expired\n";

}

}

int main()

{

// create resources to move around

auto unique\_str = std::make\_unique<std::string>("Learn ");

auto shared\_str\_1 = std::make\_shared<std::string>("Coding ");

auto shared\_str\_2 = std::make\_shared<std::string>("with Udacity!");

// Moving a unique pointer to transfer ownership

// TASK 1 : pass the pointer 'unique\_str' into the function f1

// SOLUTION 1:

f1(std::move(unique\_str));

// Pass a shared pointer by value

// TASK 2 : pass the pointer 'shared\_str\_1' into the function f2

// SOLUTION 2:

f2(shared\_str\_1);

// Pass a weak ptr by value and create a shared ptr from it to use it

std::weak\_ptr<std::string> weak\_ptr\_1;

weak\_ptr\_1 = shared\_str\_2;

f3(weak\_ptr\_1);

// Pass a weak ptr by value after the shared ptr has expired

std::weak\_ptr<std::string> weak\_ptr\_2;

{

auto shared\_str\_3 = std::make\_shared<std::string>("without Udacity");

weak\_ptr\_2 = shared\_str\_3;

}

f3(weak\_ptr\_2);

}