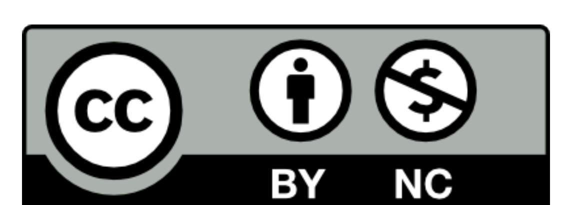
Fundamentals of C++.

From **beginner** to **beyond.**



This work is licensed under CC BY-NC 4.0 and GPL version 3.

oct. 2021

Indirection

http://ginhac.com/teaching/ITC313/2021-2022/05-indirection.pdf

Photo by Nick Tiemeyer on Unsplash

Introduce Indirection used to access or manipulate data contained in memory location pointed to by its address.

Today

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types

Lecture #02
Inheritance

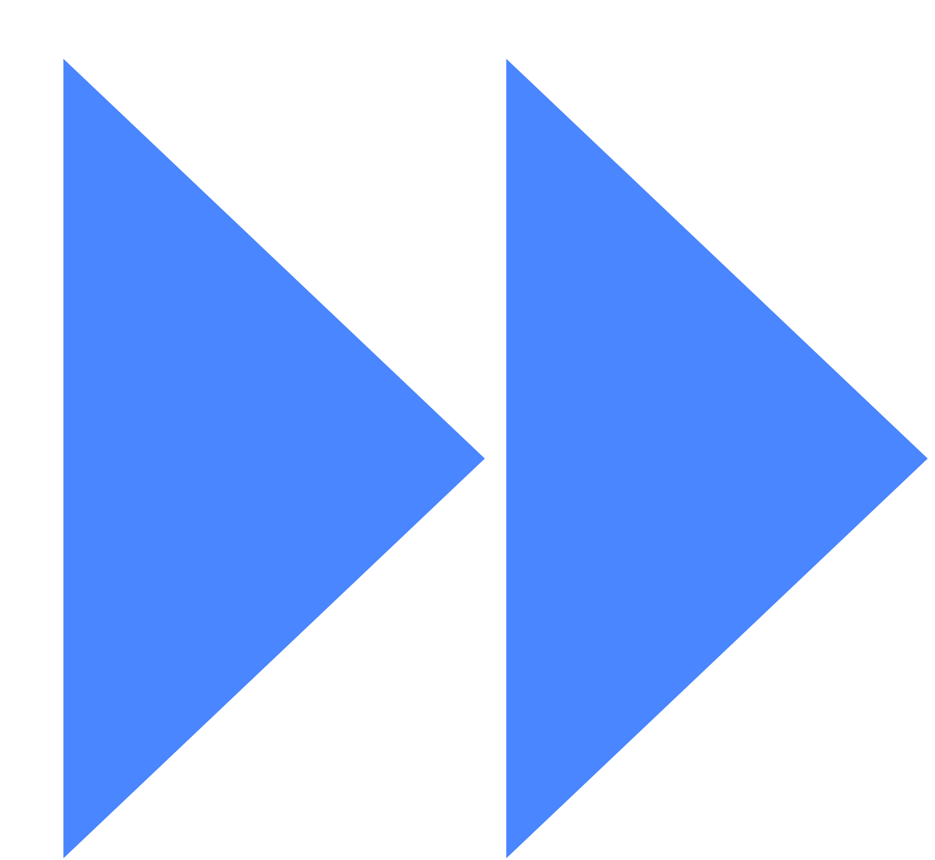
Lecture #03
Polymorphism

Lecture #04
STL Containers

Lecture #05
Indirection

Lecture #06
Templates

Lecture #07
Exceptions



A definition of indirection

In computer programming, **indirection** (also called **dereferencing**) is the ability to reference something using a name, reference, or container instead of the value itself.

The most common form of indirection is the act of manipulating a value through its **memory address**.

AGENDA

01 – A real example to introduce indirection

02 – Passing by values

03 – Passing by references

04 – Passing by pointers

A first real-life case study

Suppose you want to update a todo from the todos list.

Title: Buy ~~Beer~~ Water

Category: Personal

Due Date: 05/11

Priority: Normal

So, you need to write some methods to get the todo “Buy Beer” from the todos list and then update its title with “Buy Water”

Updating a todo

todos.h

```
class Todos {
    public:
        Todo find(std::string title) const;
        void update(Todo todo, std::string title);
   };
```

todos.cpp

```
Todo Todos::find(std::string title) const {
   auto it = std::find_if(_todos.begin(),_todos.end(),[title](const Todo& obj) {
      return obj.title() == title;
   });
   return *it;
}
void Todos::update(Todo todo, std::string title) {
    todo.setTitle(title);
}
```

```
void GenericTodo::setTitle(std::string title) {
   _title = title;
}
```

Updating a todo

main.cpp

```cpp
int main() {
    todo::Todos todos;
    todo::Todo todo1("Play piano", Category::Personal, HIGH, date::Date(11,10));
    todos.add(todo1);
    todo::Todo todo2("Write report", Category::Work, NORMAL, date::Date(11,1));
    todos.add(todo2);
    todo::Todo todo3("Buy beer", Category::Personal, NORMAL, date::Date(11,5));
    todos.add(todo3);
    todo::Todo todo4("Prepare keynote",
            Category::Work, LOW, date::Date(12,20));
    todos.add(todo4);
    std::cout << todos;
    todo::Todo to_update = todos.find("Buy beer");
    std::cout << to_update << "\n";
    todos.update(to_update, "Buy water");
    std::cout << todos;
    return 0;
}
```

```
-zsh                                                        ⌥⌘3
➜  example make
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17  -o app date.o todos.o main.o todo.o
➜  example ./app
TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11

TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12
➜  example
```

Updating a todo does not work! But why ?

Updating a todo

todos.cpp

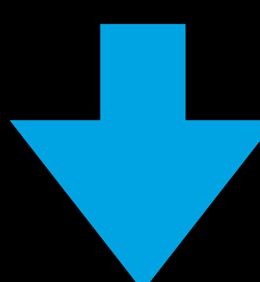
```
void Todos::update(Todo todo, std::string title) {
    std::cout << "   Inside and before update : " << todo.title() << std::endl;
    todo.setTitle(title);
    std::cout << "   Inside and after update: " << todo.title() << std::endl;
}
```

```
-zsh                                                    ⌥⌘3
➜  example make
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  app.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17  -o app date.o todos.o app.o todo.o
➜  example ./app
TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11
   Inside and before update : Buy beer
   Inside and after update: Buy champagne

TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12
➜  example
```

Objects in Memory

_todos

Todo 0
Todo 1
Todo 2
Todo3



Copy of Todo 2

todos.cpp

```
Todo Todos::find(std::string title) const {
    auto it = std::find_if(_todos.begin(),_todos.end(),
        [title](const Todo& obj) {return obj.title()==title;});
    return *it;
}
void Todos::update(Todo todo, std::string title) {
        todo.setTitle(title);
    }
```

main.cpp

```
todos.update(todos.find("Buy beer"), "Buy water");
```

todos.find(“Buy beer”) does not return the todo (“Buy beer”) but a copy of the original todo and that copy is stored in a different memory cell

This copy is then copied and passed to the todos.update() function.
The original data todo (“Buy beer”) in the _todos vector is never modified.

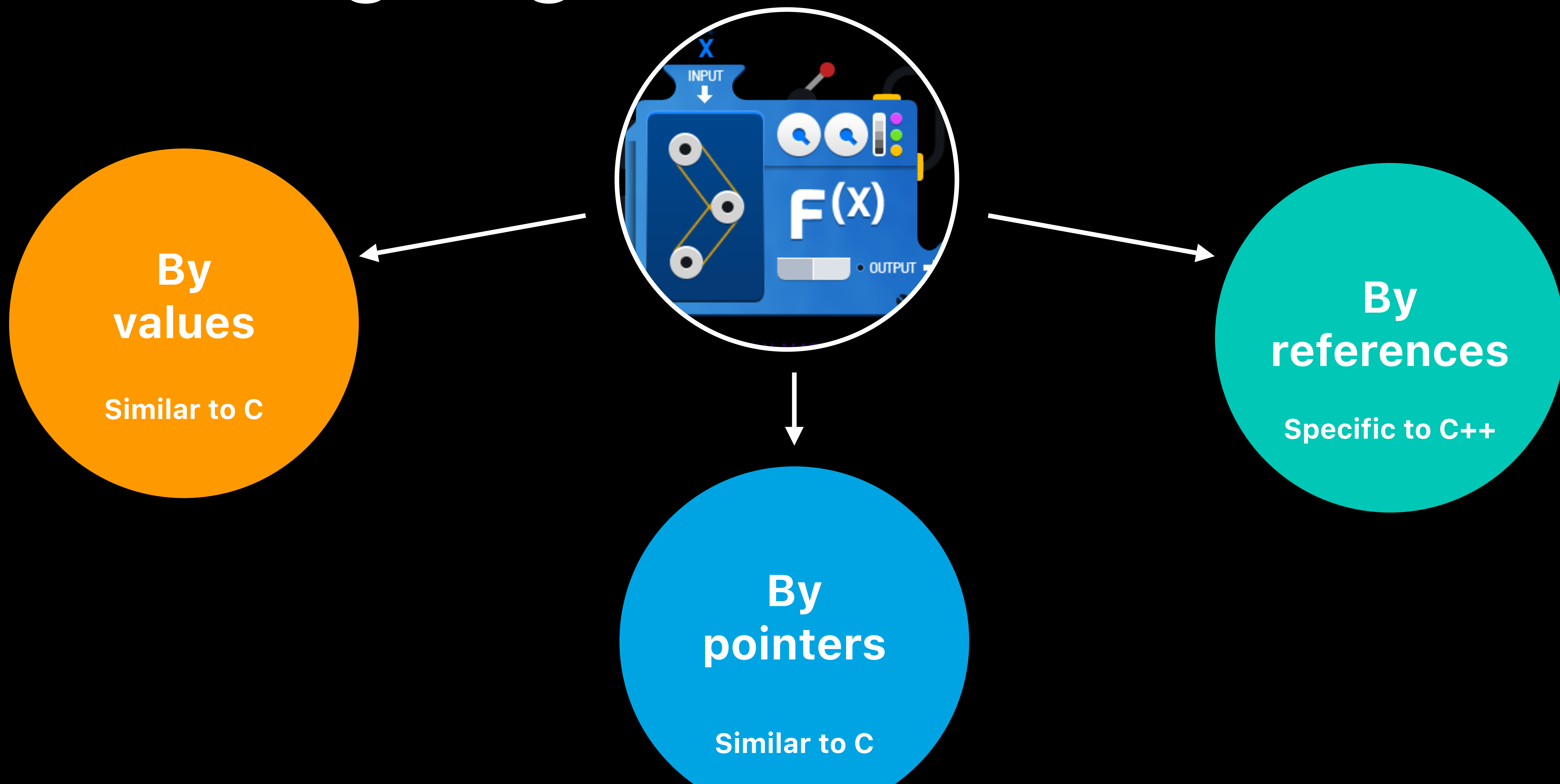
```
→  lesson06  /todos
0x7fc391405830 TODO: Play piano (HIGH) - Nov/10
0x7fc391405868 TODO: Write report (NORMAL) - Nov/1
0x7fc3914058a0 TODO: Buy beer (NORMAL) - Nov/5
0x7fc3914058d8 TODO: Prepare keynote (LOW) - Dec/20
```

```
In find function : 0x7fc3914058a0
After return: 0x7ffee0972738
In update function : 0x7ffee09726e8
```

How can we pass / return objects to / from functions?

MacBook Pro

Passing arguments to functions



AGENDA

01 – A real example to introduce indirection

02 – Passing by values

03 – Passing by references

04 – Passing by pointers

Passing by values

passing-by-values.cpp

```
void foo(int x) {
    std::cout << "   Inside foo, x = " << x << '\n';
    std::cout << "   Inside foo, @x = " << &x << '\n';
    x = 6;
    std::cout << "   After update inside foo, x = " << x << '\n';
} // x is destroyed here

int main() {
    int x = 5;
    std::cout << "Inside main,
        before foo, x = " << x << '\n';
    std::cout << "Inside main,
        before foo, &x = " << &x << '\n';
    foo(x);
    std::cout << "Inside main,
        after foo, x = " << x << '\n';
    return 0;
}
```

```
-zsh                                                    ⌥⌘3
→  values make
clang++ -std=c++17 -MMD -c  passing-by-values.cpp
clang++ -std=c++17  -o app passing-by-values.o
→  values ./app
Inside main, before foo, x = 5
Inside main, before foo, &x = 0x7ffee2a52758
   Inside foo, x = 5
   Inside foo, @x = 0x7ffee2a5272c
   After update inside foo, x = 6
Inside main, after foo, x = 5
→  values
```

PROs CONs

Arguments passed by value can be any type (values, var, expr, objects, ...)

Arguments are never changed, which prevents side effects

So use “Pass by values” when there is no need to modify parameters

“Pass by Values” makes always a copy of all the arguments into the function parameters

Copying complex objects or large arrays can lead to a significant performance penalty

So do not use “Pass by values” with complex data. Prefer other solutions

Rule: When the function does not need to change the arguments, "PASS BY VALUE” is FLEXIBLE, SAFE, and EFFICIENT in the case of fundamental types.

Questions ?

AGENDA

01 – A real example to introduce indirection

02 – Passing by values

03 – Passing by references

04 – Passing by pointers

What is a reference?

A REFERENCE is a type of C++ variable that acts as an ALIAS to another object or value. A reference acts identically to the value it's referencing.

```
int value = 5; // value is 5
int& ref = value; // reference to variable value
value = 6; // value is now 6
ref = 7; // value is now 7
```

You can access to the value with the variable or the reference.

A reference must ALWAYS BE INITIALIZED when created.

Once initialized, it CAN'T BE CHANGED to reference another variable.

References and functions

References are most often used as FUNCTION PARAMETERS.

The reference parameter acts as an alias for the argument, and NO COPY of the argument is made.

A function that uses a reference parameter is able to MODIFY THE ARGUMENT passed in.

```
void foo(int& y) {
    std::cout << "   Inside foo, y = " << y << '\n';
    y = 6;
    std::cout << "   After update inside foo, y = " << y << '\n';
} // y is destroyed here
```

Passing by reference

```
void foo(int& y) {
    std::cout << "Inside foo, y = " << y << '\n';
    y++;
    std::cout << "After update inside foo,
                 y = " << y << '\n';
} // y is destroyed here
int main() {
    int x = 5;
    int& z = x; // create a reference
    std::cout << "Inside main, before foo,
                   x = " << x << '\n';
    std::cout << "Inside main, before foo,
                   z = " << z << '\n';
    foo(z); // Call foo with the reference z
    std::cout << "Inside main, after foo, z = " << z << '\n';
    std::cout << "Inside main, after foo, x = " << x << '\n';
    foo(x); // Compiler automatically creates a reference on x
    std::cout << "Inside main, after foo, x = " << x << '\n';
    return 0;
}
```

passing-by-reference.cpp

```
-zsh — d0m                                        ⌥⌘3
➜  01-passing-by-reference make
clang++ -std=c++17 -MMD -c  passing-by-reference.cpp
clang++ -std=c++17  -o app passing-by-reference.o
➜  01-passing-by-reference ./app
Inside main, before foo, x = 5
Inside main, before foo, z = 5
   Inside foo, y = 5
   After update inside foo, y = 6
Inside main, after foo, z = 6
Inside main, after foo, x = 6
   Inside foo, y = 6
   After update inside foo, y = 7
Inside main, after foo, x = 7
➜  01-passing-by-reference
```

Value vs Reference

```cpp
void fooRef(int& y) {
   std::cout << "fooRef: ad(y)=" << &y << '\n';
    y++;
}
void fooVal(int y) {
    std::cout << "fooVal: ad(y)=" << &y << '\n';
    y++;
}
int main() {
    int x = 10;
    std::cout << "main: ad(x)=" << &x << '\n';
    fooVal(x);
    std::cout << "main: after fooVal, x=" << x << '\n';
    fooRef(x);
    std::cout << "main: after fooRef, x=" << x << '\n';
    return 0;
}
```

value-reference.cpp

-zsh — d0m ⌥⌘3

```
➜  02-value-ref make
clang++ -std=c++17 -MMD -c  value-reference.cpp
clang++ -std=c++17  -o app value-reference.o
➜  02-value-ref ./app
main: ad(x)=0x16f6cb4f8
fooVal: ad(y)=0x16f6cb4cc
main: after fooVal, x=10
fooRef: ad(y)=0x16f6cb4f8
main: after fooRef, x=11
➜  02-value-ref
```

Returning multiple results

```cpp
void swap (int &x, int &y) {
    int tmp = x;
    x=y;
    y=tmp;
}

int main() {
     int a = 2, b = -3;
     std::cout << "Before, a=" << a
               << " - b=" << b << '\n';
     swap(a,b);
     std::cout << "After, a=" << a
               << " - b=" << b << '\n';
     return 0;
}
```

swap.cpp

-zsh — d0m ⌥⌘3

```
➜  03-swap make
clang++ -std=c++17 -MMD -c  swap.cpp
clang++ -std=c++17  -o app swap.o
➜  03-swap ./app
Before, a=2 - b=-3
After, a=-3 - b=2
➜  03-swap
```

PROs CONs

References allow a function to change the value of an argument

References can be used to return multiple values from a function

“Pass by reference” is fast

So use “Pass by reference” when there is a need to modify parameters or when passing complex objects.

It can be hard to tell whether an argument passed as a reference is an input, an output or both

In the body of a function, it’s not possible to make the difference between a value and a reference

So do not use “Pass by reference” when passing fundamental types that don’t need to be modified.

Rule: Use PASS BY REFERENCE instead of pass by value for CLASSES and other expensive-to-copy types.

Reference to const value

References used as function parameters can also be CONST.

You can access the argument without making a copy, while guaranteeing that the function will NOT CHANGE THE VALUE being referenced.

References to const values are particularly USEFUL as function parameters to efficiently pass arrays or complex objects.

It combines the ADVANTAGES of “Pass by value” and “Pass by reference”.

Using reference to const value

```
int sum1(std::vector<int> data) {
    int s=0; auto it=data.begin();
    while (it != data.end()) {
        s+= *it;
        ++it;
    }
    return s;
}
```

```
int sum2(std::vector<int>& data) {
    int s=0; auto it=data.begin();
    while (it != data.end()) {
        s+= *it;
        ++it;
    }
    return s;
}
```

```
int sum3(const std::vector<int>& data) {
    int s=0; auto it=data.begin();
    while (it != data.end()) {
        s+= *it;
        ++it;
    }
    return s;
}
```

3 different ways to pass a vector to a function

Prefer reference to const if you do not have to modify the values inside the function

Updating a todo using references

todos.h

```
class Todos {
    public:
        Todo find(std::string title) const;
        void update(Todo todo, std::string title);
   };
```

todos.cpp

```
Todo Todos::find(std::string title) const {
   auto it = std::find_if(_todos.begin(),_todos.end(),[title](const Todo& obj) {
      return obj.title() == title;
   });
   return *it;
}
void Todos::update(Todo todo, std::string title) {
    todo.setTitle(title);
}
```

What code needs to be updated?

Updating a todo using references

todos.h

```
class Todos {
    public:
        Todo& find(std::string title) const;
        void update(Todo& todo, std::string title);
  };
```

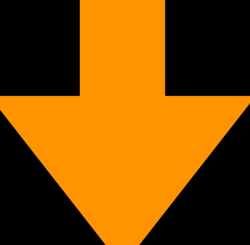
We have to use A REFERENCE each time we call a Todo object as an argument of a function

_todos

Todo 0
Todo 1
Todo 2
Todo3

Ref on Todo

```
Todo& to_update = todos.find("Buy beer");
todos.update(to_update, "Buy water");
```



```
todos.update(todos.find("Buy beer"), "Buy water");
```

Updating a todo using references

todos.h

```cpp
class Todos {
    public:
        Todo& find(std::string title) const;
        void update(Todo& todo, std::string title);
  };
```

Use A REFERENCE for each Todo object as an argument of a function

todos.cpp

```cpp
Todo& Todos::find(std::string title) const {
   auto it = std::find_if(_todos.begin(),_todos.end(),[title](const Todo& obj) {
      return obj.title() == title;
   });
   return *it;
}
void Todos::update(Todo& todo, std::string title) {
    todo.setTitle(title);
}
```

Updating a todo using references

main.cpp

```
int main() {
    todo::Todos todos;
    todo::Todo todo1("Play piano", Category::Personal, HIGH,
    todos.add(todo1);
    todo::Todo todo2("Write report", Category::Work, NORMAL,
    todos.add(todo2);
    todo::Todo todo3("Buy beer", Category::Personal, NORMAL,
    todos.add(todo3);
    todo::Todo todo4("Prepare keynote", Category::Work, LOW,
    todos.add(todo4);
    std::cout << todos << "\n";
    todo::Todo&  to_update = todos.find("Buy beer");
    std::cout << to_update << "\n";
    todos.update(to_update, "Buy champagne”);
    std::cout << todos;
    return 0;
}
```

```
-zsh — d0m                                        ⌥⌘3
➜  04-todos make
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17  -o app date.o todos.o
 main.o todo.o
➜  04-todos ./app
TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11

TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy champagne (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12
➜  04-todos
```

It works now with the same code for main.cpp

Questions

AGENDA

01 – A real example to introduce indirection

02 – Passing by values

03 – Passing by references

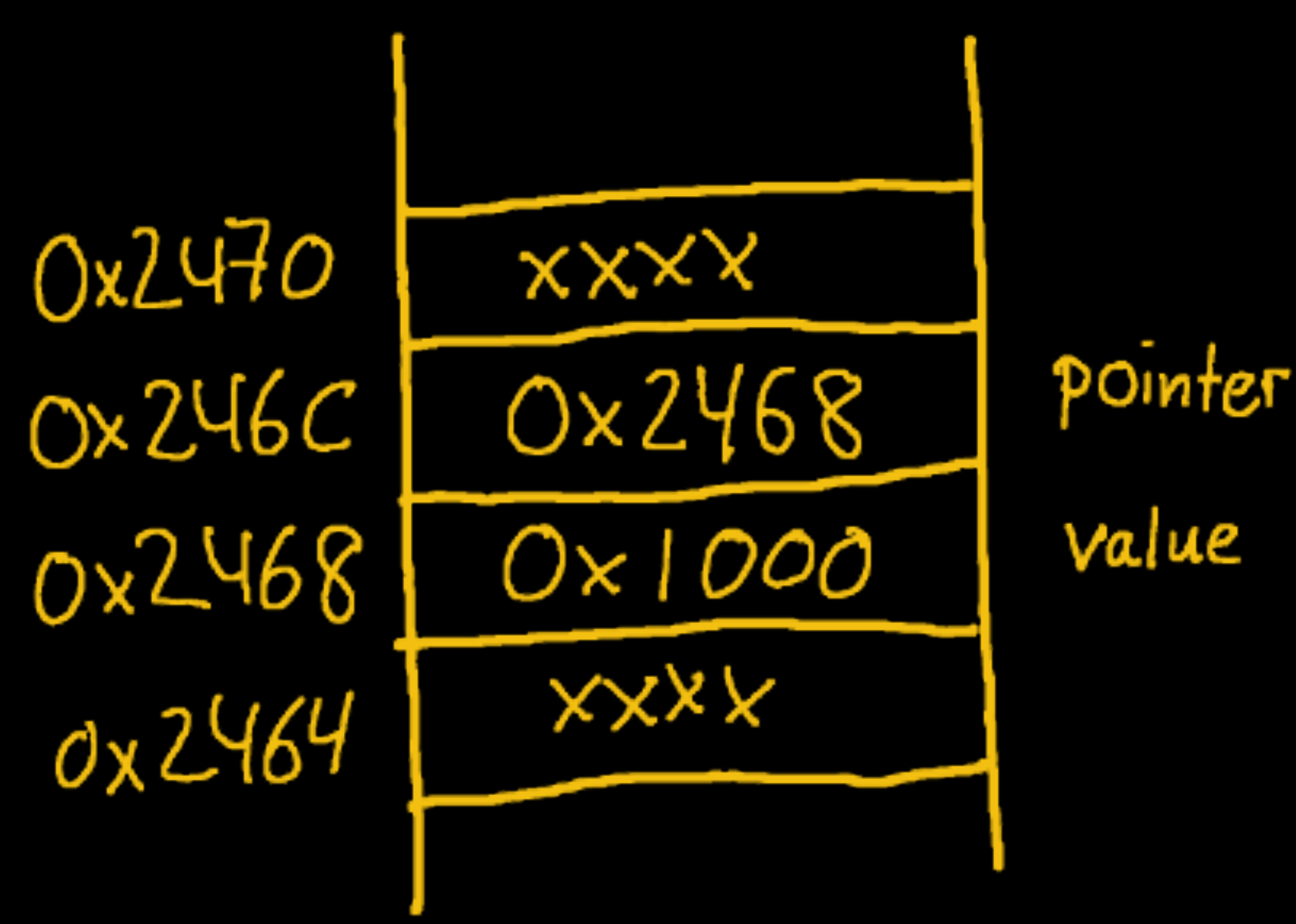
04 – Passing by pointers

The concept of pointers

Pointers are one of the most distinct and exciting features of C/C++ languages

Pointers are used to ACCESS AND MANIPULATE the memory using ADDRESSES

Pointers are variables that store adresses



pointers.cpp

DECLARING a pointer (Type*)

```cpp
// Declares a pointer on an int and init the pointer to null
int* ptr = nullptr; // int *ptr = nullptr;

// Makes ptr point to the integer variable
// using the address-of operator (&)
int value = 42; ptr = &value; // int* ptr = &value;

// prints the address of value and ptr
std::cout << "&v=" << &value << std::endl;
std::cout << "ptr=" << ptr << std::endl;
```

ptr
0x16fdeb52c
~~42~~ 1337
value
*ptr

DEREFERENCING a pointer (operator *)

```cpp
// prints the variable value
std::cout << "v=" << value << std::endl;
// updates the value using ptr
*ptr = 1337; // value = 1337;
// prints the value that ptr is holding
std::cout << "*ptr=" << *ptr << std::endl;
```

-zsh — d0m ⌥⌘3

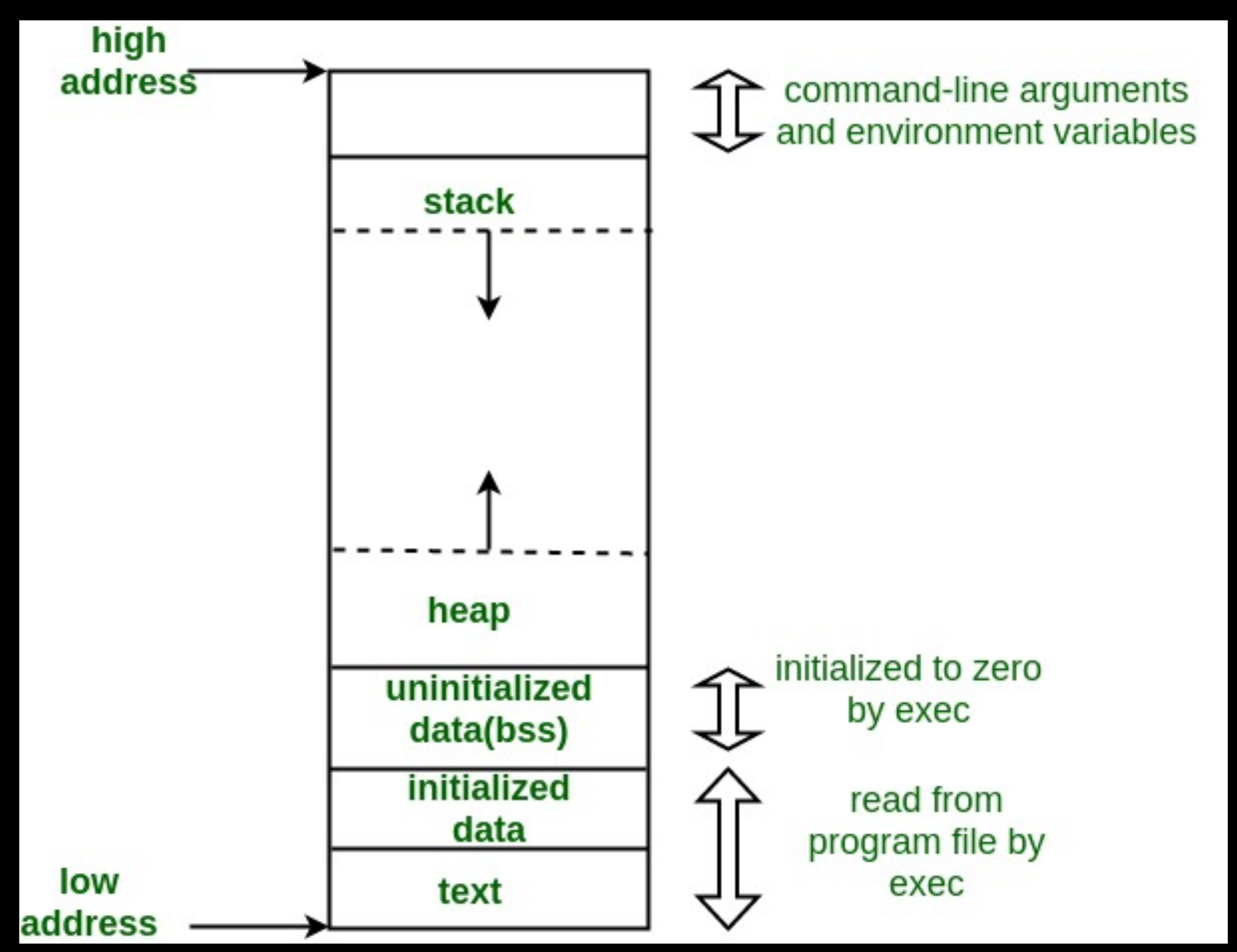
```
→  01-basic make
clang++ -std=c++17 -MMD -c  pointers.cpp
clang++ -std=c++17  -o app pointers.o
→  01-basic ./app
&v=0x16dd7f51c
ptr=0x16dd7f51c
v=42
*ptr=1337
→  01-basic
```

Allocating memory

DYNAMIC MEMORY ALLOCATION refers to performing memory allocation during runtime

Dynamically allocated memory is allocated on the HEAP whereas static variables get memory allocated on the STACK

C++ has two operators NEW and DELETE that perform the task of allocating and freeing the memory



Low-level memory management

alloc.cpp

```cpp
int main() {
  int* pa = nullptr; pa = new int; *pa = 42;
  std::cout << "pa: " << pa << " *pa: " << *pa << std::endl;
  int* pb = new int(666);
  std::cout << "pb: " << pb << " *pb: " << *pb << std::endl;
  int* pc = pb; // No allocation of memory
  std::cout << "pc: " << pc << " *pc: " << *pc << std::endl;

  int* pd = new int[666];
  for (auto i=0;i<666;i++)
      *(pd+i) = i;
  for (auto i=0;i<666;i++)
      std::cout << "pd+“ << i << “:" << pd+i << " *pd+"
                    << i << ": " << *(pd+i) << std::endl;
  delete pa; delete pb; // no delete pc;
  delete [] pd;
  return 0;
}
```

-zsh — d0m ⌥⌘3

```
➜  02-allocation make
clang++ -std=c++17 -MMD -c  alloc.cpp
clang++ -std=c++17  -o app alloc.o
➜  02-allocation ./app
pa: 0x600002bd0030 *pa: 42
pb: 0x600002bd0040 *pb: 666
pc: 0x600002bd0040 *pc: 666
pd+0: 0x148809800 *pd+0: 0
pd+1: 0x148809804 *pd+1: 1
pd+2: 0x148809808 *pd+2: 2
pd+3: 0x14880980c *pd+3: 3
pd+4: 0x148809810 *pd+4: 4
pd+5: 0x148809814 *pd+5: 5
pd+6: 0x148809818 *pd+6: 6
pd+7: 0x14880981c *pd+7: 7
pd+8: 0x148809820 *pd+8: 8
pd+9: 0x148809824 *pd+9: 9
pd+10: 0x148809828 *pd+10: 10
pd+11: 0x14880982c *pd+11: 11
pd+12: 0x148809830 *pd+12: 12
pd+13: 0x148809834 *pd+13: 13
pd+14: 0x148809838 *pd+14: 14
pd+15: 0x14880983c *pd+15: 15
pd+16: 0x148809840 *pd+16: 16
pd+17: 0x148809844 *pd+17: 17
pd+18: 0x148809848 *pd+18: 18
```

Low-level memory management

main.cpp

```cpp
#include "todo.h"

int main()
{
    Todo todo1("Play piano”, todo::Category::Personal, HIGH, date::Date(11,10));
    Todo* todo2 = new Todo("Buy beer",todo::Category::Personal, NORMAL,
                           date::Date(11,5));

    std::cout << todo1.title() << std::endl;
    std::cout << todo2->title() << std::endl;

    delete todo2;
    return 0;
}
```

Use '->' instead '.' to access to members functions

```
-zsh — d0m                                                    ⌥⌘3
➜  03-objects make
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17  -o app date.o main.o todo.o todos.o
➜  03-objects ./app
Play piano
Buy beer
➜  03-objects
```

Memory leaks

Memory leaks occur when new memory is allocated dynamically on heap but NEVER DEALLOCATED

Memory leaks are particularly SERIOUS ISSUES

The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even CRASH a program

Rule: For every "new," you should use a "delete" so that you free the memory you allocated

Memory leak

memory-leak.cpp

```
int main() {
    // No memory leak
    float* marks1 = new float [30]; // Allocate 30 float to store marks.
    delete [] marks1; // Clear those 30 float and make ptr to null.
    marks1 = nullptr;
    // Memory leak 1: Forget to delete
    float* marks2 = new float [30];
    Marks2 = nullptr;
    // Memory leak 2: Never reallocate a ptr that is not free
    float* marks3 = new float [30];
    marks3 = new float [30]; // Give marks another memory address
    delete [] marks3; // Delete only the last allocation
    marks3 = nullptr;
    // Memory leak 3: Never move a ptr that is not free
    float* marks4 = new float [30];
    float* marks5 = new float [30];
    marks5 = marks4; // The first 30 float are impossible to free
    delete [] marks4;
    marks4 = nullptr;
    //delete [] marks5; // Error: access violation
    marks5 = nullptr;
    return 0;
}
```

No error, no warning

```
-zsh — d0m                                        ⌥⌘2
➜  04-leaks clang++ -Wall memory-leak.cpp -o app
➜  04-leaks ./app
➜  04-leaks
```

Dangling pointers

Dangling pointers are pointers that POINT TO A MEMORY LOCATION that has been FREED.

Dangling pointers arise when an object is deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

The system may reallocate the previously freed memory and UNPREDICTABLE BEHAVIOR may result as the memory may now contain completely different data.

Dangling pointers

dangling-pointer.cpp

```
#include <iostream>

int main() {
    int* pa = new int;
    *pa = 2;
    std::cout << "pa=" << pa << " - *pa=" << *pa << std::endl;

    delete pa;
    std::cout << "pa=" << pa << " - *pa=" << *pa << std::endl;

    pa = nullptr;
    if (pa!=nullptr)
        std::cout << "pa=" << pa
        << " - *pa=" << *pa << std::endl;
    return 0;
}
```

-zsh — d0m ⌥⌘2

```
➜  05-dangling clang++ dangling-pointer.cpp -o app
➜  05-dangling ./app
pa=0x600001440030 - *pa=2
pa=0x600001440030 - *pa=-675545040
➜  05-dangling
```

High-level memory management

'Raw' C++ pointers are extremely powerful but can be very dangerous (memory leaks, dangling pointers, unpredictable behavior, ...).

The problem: Every dynamically allocated object must be followed by a manual deallocation. This forces you to mentally keep track of what you have allocated.

```
MyObject* object = new MyObject;
object->doSomething();
delete object; // Just do it once

MyObject* array = new MyObject[10];
for (auto i=0; i<10; i++ )
   (array+i)->doSomething();
delete [] array; // Just do it once
```

High-level memory management

Smart pointers provide automatic memory management: when a smart pointer is no longer in use, the memory is deallocated automatically.

C++11 has introduced three types of smart pointers:

- std::unique_ptr owns a dynamically allocated resource
- std::shared_ptr owns a shared dynamically allocated resource. Several std::shared_ptrs may own the same resource and an internal counter keeps track of them
- std::weak_ptr is like a std::shared_ptr, but it doesn't increment the counter

Smart pointers always should be preferred over ‘raw’ pointers

See https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c

High-level memory management

A std::unique_ptr owns of the object it points to and no other smart pointer can point to it.

When the std::unique_ptr goes out of scope, the object is deleted. This is useful when you are working with a temporary, dynamically-allocated resource that can get destroyed once out of scope.

```
std::unique_ptr<MyObject>     smart_ptr(new MyObject);
std::unique_ptr<MyObject[]>   smart_array(new MyObject[10]);

// std::make_unique requires c++14
std::unique_ptr<MyObject> smart_ptr2 = std::make_unique<MyObject>();
std::unique_ptr<MyObject[]> smart_array2 = std::make_unique<MyObject[]>(10);
```

High-level memory management

```cpp
class Data {
private:
    int _data;
public:
    Data() : _data(0) {}
    int data() const { return _data;}
    void data(int d) {_data = d;}
    ~Data() {std::cout << "Destroy Data " << std::endl;}
};
int main() {
    std::unique_ptr<Data>    smart_ptr(new Data);
    std::unique_ptr<Data[]>  smart_array(new Data[5]);
    smart_ptr->data(666);
    std::cout << smart_ptr->data() << std::endl;
    for (auto i=0; i<5; i++ )
        smart_array[i].data(3*i+1);
    for (auto i=0; i<5; i++ )
        std::cout << smart_array[i].data() << " ";
    std::cout << std::endl;
    std::cout << "That's all folks!" << std::endl;
}
```

unique-ptr.cpp

```
-zsh — d0m                                   ⌥⌘2
➜  06-smart-pointers clang++ -std=c++17
 unique-ptr.cpp -o app
➜  06-smart-pointers ./app
666
1 4 7 10 13
That's all folks!
Destroy Data
Destroy Data
Destroy Data
Destroy Data
Destroy Data
Destroy Data
➜  06-smart-pointers
```

High-level memory management

A std::unique_ptr can’t have multiple references to its dynamic data

```
std::unique_ptr<MyObject>    smart_ptr(new MyObject);
// Error because unique_ptr does not have a copy constructor
std::unique_ptr<MyObject> smart_ptr2 = smart_ptr;
```

A std::shared_ptr owns the object it points to and cave have multiple references to its dynamic data. A special internal counter is decreased each time a std::shared_ptr goes out of scope (reference counting). When the last one is destroyed (counter goes to zero), the data is deallocated

```
std::shared_ptr<MyObject> smart_ptr3(new MyObject);
std::shared_ptr<MyObject> smart_ptr4 = smart_ptr3;
```

High-level memory management

shared-ptr.cpp

```
class Data {
private:
    int _data;
public:
    Data() : _data(0) {}
    int data() const{ return _data;}
    void data(int d) {_data = d;}
    ~Data() { std::cout << "Destroy Data " << std::endl;}
};
int main() {
    std::cout << smart_ptr2->data() << std::endl;
    std::shared_ptr<Data> smart_ptr3(new Data);
    std::shared_ptr<Data> smart_ptr4 = smart_ptr3;
    smart_ptr3->data(666);
    std::cout << smart_ptr3->data() << std::endl;
    std::cout << smart_ptr4->data() << std::endl;
    smart_ptr4->data(1337);
    std::cout << smart_ptr3->data() << std::endl;
    std::cout << smart_ptr4->data() << std::endl;
}
```

```
-zsh — d0m                                              ⌥⌘2
➜  06-smart-pointers clang++ -std=c++17
 shared-ptr.cpp -o app
➜  06-smart-pointers ./app
666
666
1337
1337
Destroy Data
➜  06-smart-pointers
```

Passing by pointer

Passing an argument by pointer involves PASSING THE RAW POINTER or PASSING THE SMART POINTER on the argument variable rather than the argument variable itself.

The function have to DEREFERENCE the pointer to access or change the value being pointed to.

```
void foo(int* y)
{
    *y = 10;
}
```

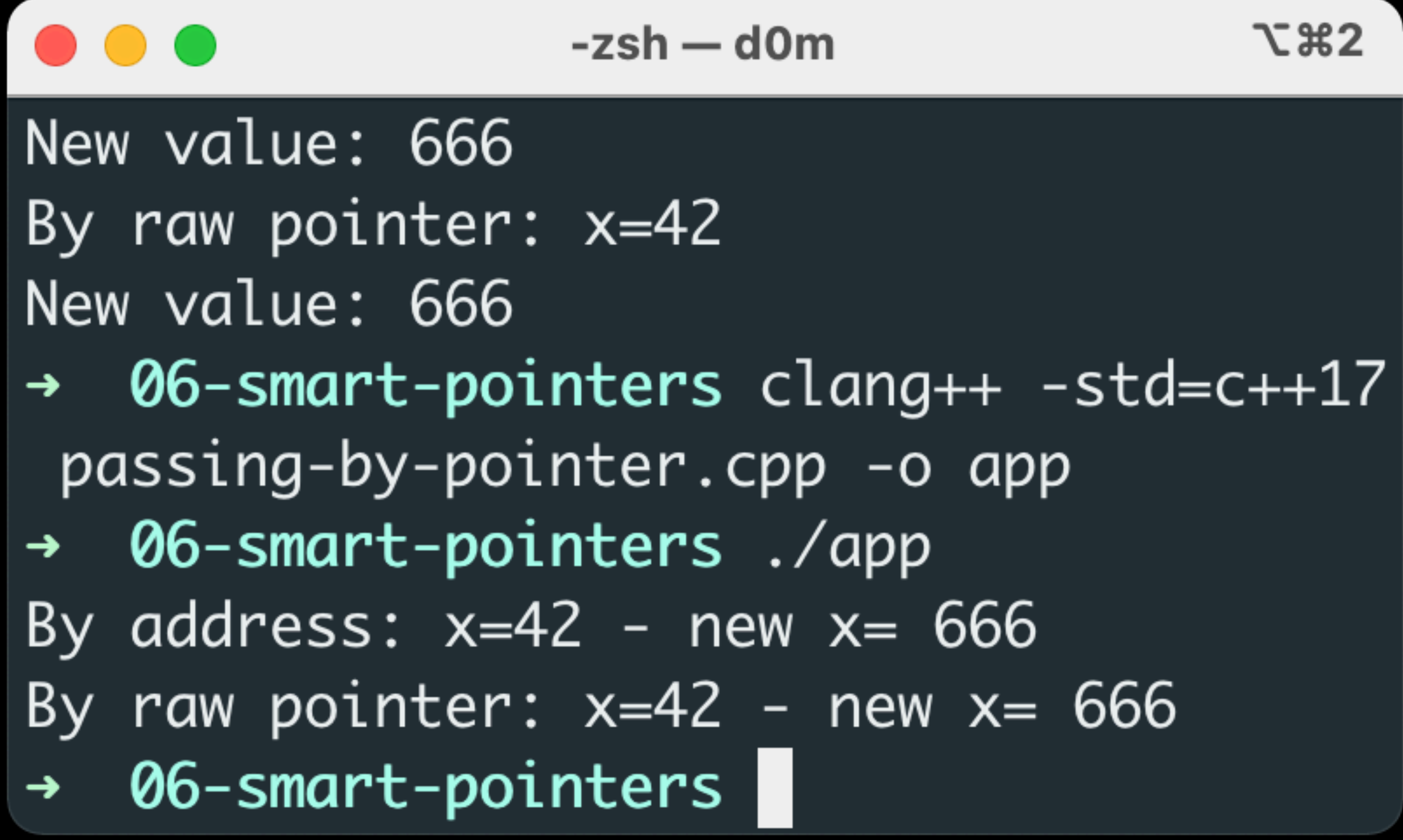
Argument explicitly declared as an address

Using dereferencing operator to modify the argument

Passing by pointer

passing-by-pointer.cpp

```
void foo (int* data, int new_value) {
    *data = new_value;
}
int main() {
    int x = 42;
    int y = 666;
    int* b_ptr = new int (42);

    std::cout << “By Address: x=" << x;
    foo (&x, y);
    std::cout << “- new x=" << x << std::endl;

    std::cout << “By Raw pointer: x=” << *b_ptr;
    foo (b_ptr, y);
    std::cout << “- new x=" << *b_ptr << std::endl;
    return 0;
}
```



Passing by smart pointer

```
void foo2 (std::unique_ptr<int> data, int new_value) { *data = new_value; }
void foo3 (std::unique_ptr<int>& data, int new_value) { *data = new_value; }
void foo4 (std::shared_ptr<int> data, int new_value) { *data = new_value; }
void foo5 (std::shared_ptr<int>& data, int new_value) { *data = new_value; }
int main() {
    std::cout << "Unique smart pointer" << std::endl;
    std::unique_ptr<int> u_ptr(new int(42));
    std::cout << *u_ptr << std::endl;
    foo2(std::move(u_ptr), 666);
    // up is now a hollow object
    // std::cout << *u_ptr << std::endl;
    std::cout << "Unique smart pointer with reference" << std::endl;
    std::unique_ptr<int> ur_ptr(new int(42));
    std::cout << *ur_ptr << std::endl;
    foo3(ur_ptr, 666);
    std::cout << *ur_ptr << std::endl;
    std::cout << "Shared smart pointer" << std::endl;
    std::shared_ptr<int> s_ptr(new int(42));
    std::cout << *s_ptr << std::endl;
    foo4(s_ptr, 666);
    std::cout << *s_ptr << std::endl;
    std::cout << "Shared smart pointer with reference" << std::endl;
    std::shared_ptr<int> sr_ptr(new int(42));
    std::cout << *sr_ptr << std::endl;
    foo5(sr_ptr, 666);
    std::cout << *sr_ptr << std::endl;
    return 0;
}
```

passing-by-smart-ptr.cpp

```
-zsh — d0m                                          ⌥⌘2
➜  06-smart-pointers clang++ -std=c++17
 passing-by-smart-ptr.cpp -o app
➜  06-smart-pointers ./app
Unique smart pointer
42
Unique smart pointer with reference
42
666
Shared smart pointer
42
666
Shared smart pointer with reference
42
666
➜  06-smart-pointers
```

See https://www.internalpointers.com/post/move-smart-pointers-and-out-functions-modern-c
https://www.modernescpp.com/index.php/c-core-guidelines-passing-smart-pointer

References vs Pointers

Reference must be initialized when it is created

Once initialized, we cannot reinitialize a reference

References can never be NULL

Reference is automatically dereferenced

Pointers can be created and initialized any time

Pointers can be reinitialized any number of times

Pointers can be nullptr

* is used to dereference a pointer

When should I use references, and when should I use pointers?

THE RULE: Use references when you can, and pointers when you have to

References are usually preferred over pointers in C++, whenever you do not need “reseating”

Use pointers only if there is no other choice

Prefer smart pointers rather than raw pointers

Updating a todo using pointers

todos.h

```
class Todos {
    public:
        Todo& find(std::string title) const;
        void update(Todo& todo, std::string title);
  };
```

todos.cpp

```
Todo& Todos::find(std::string title) const {
   auto it = std::find_if(_todos.begin(),_todos.end(),[title](const Todo& obj) {
      return obj.title() == title;
   });
   return *it;
}
void Todos::update(Todo& todo, std::string title) {
    todo.setTitle(title);
}
```

What code needs to be updated?

Updating a todo using pointers

We have to replace reference by pointer each time we call a Book object as an argument of a function.

The Rule: Use smart pointers if you want to manipulate the smart pointer itself, use raw pointers if you just need a handle to the underlying object.

todos.h

```
class Todos {
    public:
        Todo* find(std::string title) const;
        void update(Todo* todo, std::string title);
   };
```

main.cpp

```
Todo* to_update = todos.find("Buy beer");
todos.update(to_update, "Buy water");
```

Updating a todo using pointers

todos.h

```
class Todos {
    public:
        Todo* find(std::string title) const;
        void update(Todo* todo, std::string title);
  };
```

Use A POINTER for each Todo object as an argument of a function

todos.cpp

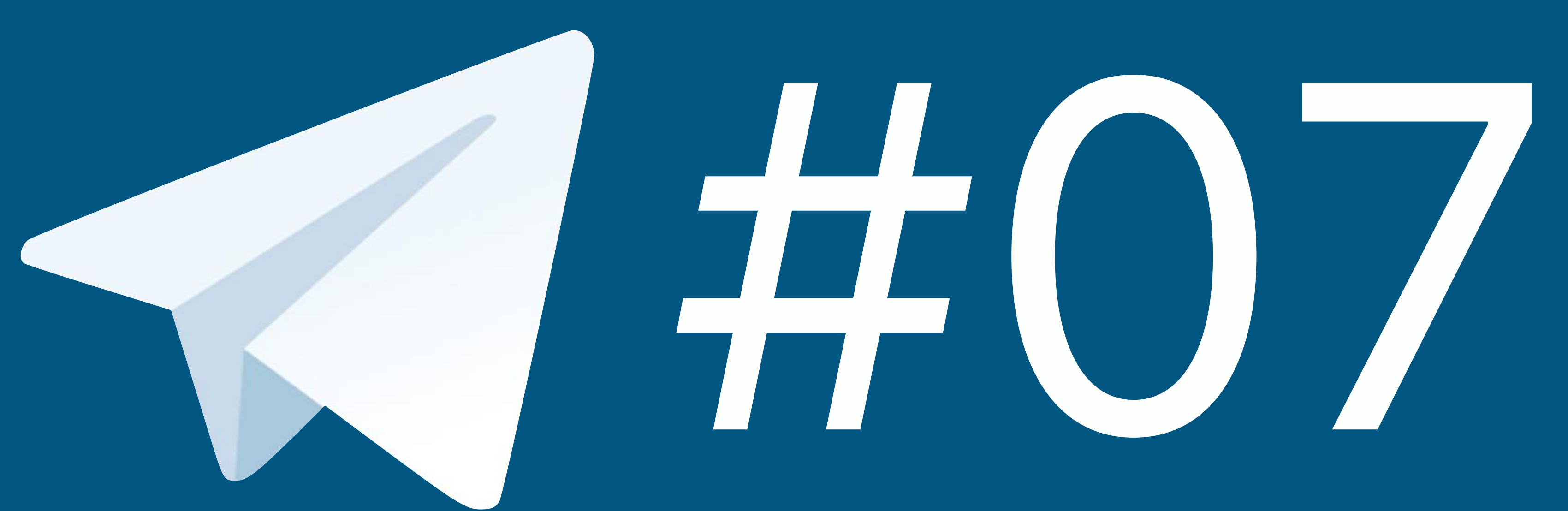
```
Todo* Todos::find2(std::string title) {
    auto it = std::find_if(_todos.begin(), _todos.end(),
                           [title](const Todo& obj) {
                              return obj.title() == title;
                           });
    int index = std::distance(_todos.begin(), it) ;
    return &_todos.at(index);
}
void Todos::update2(Todo* todo, std::string title) {
    todo->setTitle(title);
}
```

Updating a todo using pointers

```cpp
int main() {
    todo::Todos todos;
    todo::Todo todo1("Play piano", Category::Personal,
    todos.add(todo1);
    todo::Todo todo2("Write report", Category::Work, N
    todos.add(todo2);
    todo::Todo todo3("Buy beer", Category::Personal, N
    todos.add(todo3);
    todo::Todo todo4("Prepare keynote", Category::Work
    todos.add(todo4);
    std::cout << todos;
    todo::Todo*  to_update = todos.find("Buy beer”);
    std::cout << *to_update << "\n";
    todos.update(to_update, "Buy champagne");
    std::cout << todos;
    return 0;
}
```

-zsh — d0m ⌥⌘2

```
➜  07-todos make
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17  -o app todos.o todo.o date.o main.o
➜  07-todos ./app
TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11

TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Buy champagne (NORMAL) - 5/11
TODO: Prepare keynote (LOW) - 20/12
➜  07-todos
```



Take Home Message

C++11 and beyond provide different ways to pass arguments to a function

Choosing call-by-value, call-by-reference or call-by-pointer depends on what you want to do.

- If you want to change the object passed, call by reference or use a pointer
- If you don’t want to change the object passed and if this object is big, call by const reference
- Otherwise, call by value

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Come visit us at

https://github.com/dginhac/esirem-itc313

This work is **licensed** under a
Creative Commons Attribution-NonCommercial 4.0 International License.

https://creativecommons.org/licenses/by-nc/4.0/