CS4414 Recitation 2

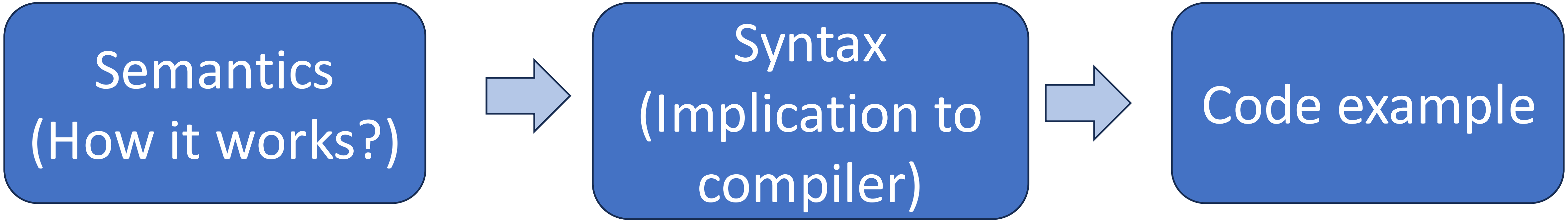
C++ Derived Types and Class

09/2024

Alicia Yang

Overview

- C++ derived types
  - Pointers, Reference, Array, Functions
- C++ class



Semantics (How it works?)

Syntax (Implication to compiler)

Code example

What is C++?

A federation of related languages, with four primary sublanguages

- **C**: C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

C++ types

- Primitive data types
  - bool
  - char
  - int
  - float
  - double
  - void
  - ……
- Derived data types
  - pointer
  - reference
  - array
  - function
- User-defined data types
  - class
  - struct

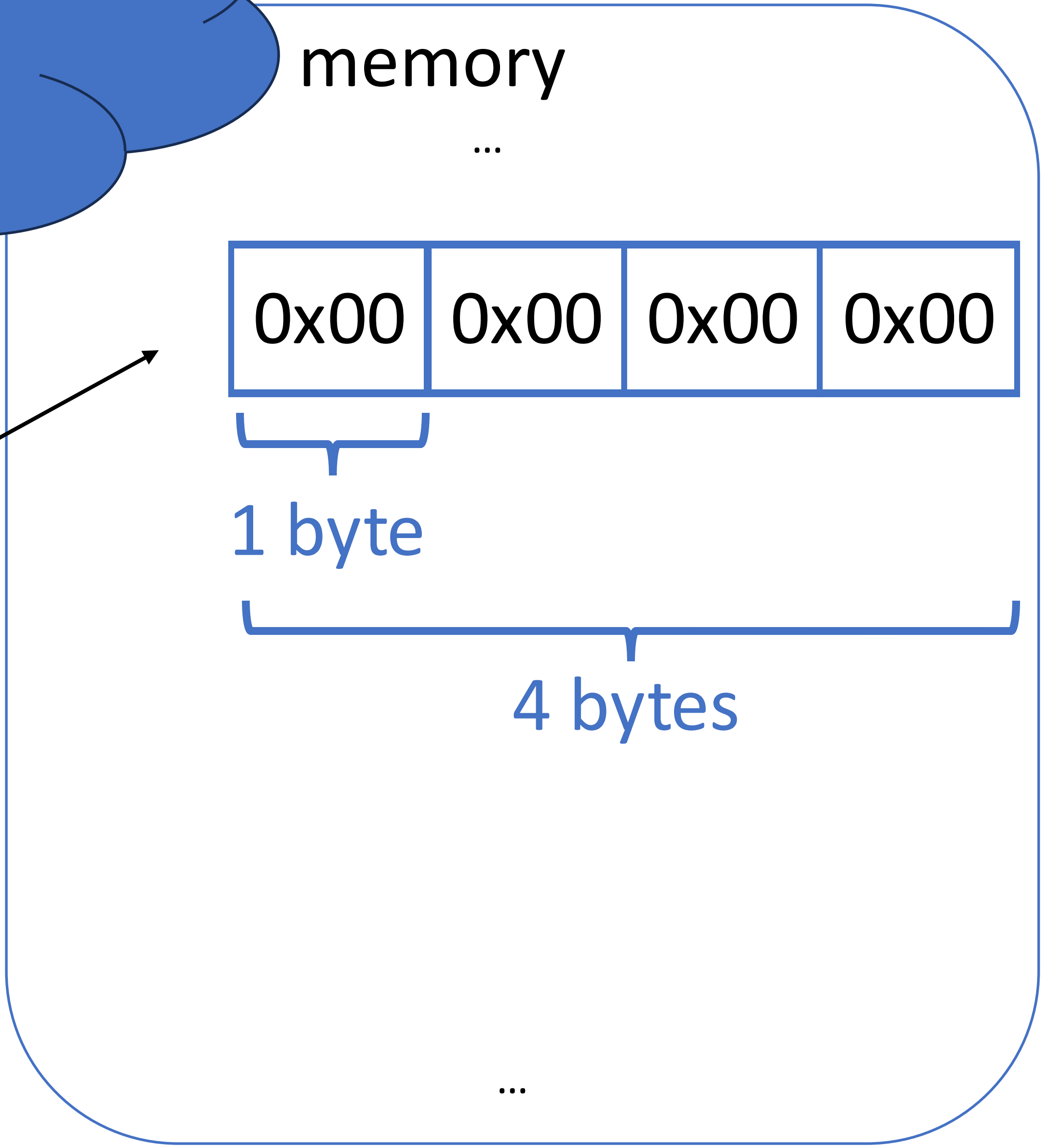
& | Address

Can I store &x in a variable to use in the future?

```
int32_t x = 0;
```

&x is the memory address of x

// e.g. 0x7ffd39809084



memory

…

0x00	0x00	0x00	0x00

1 byte

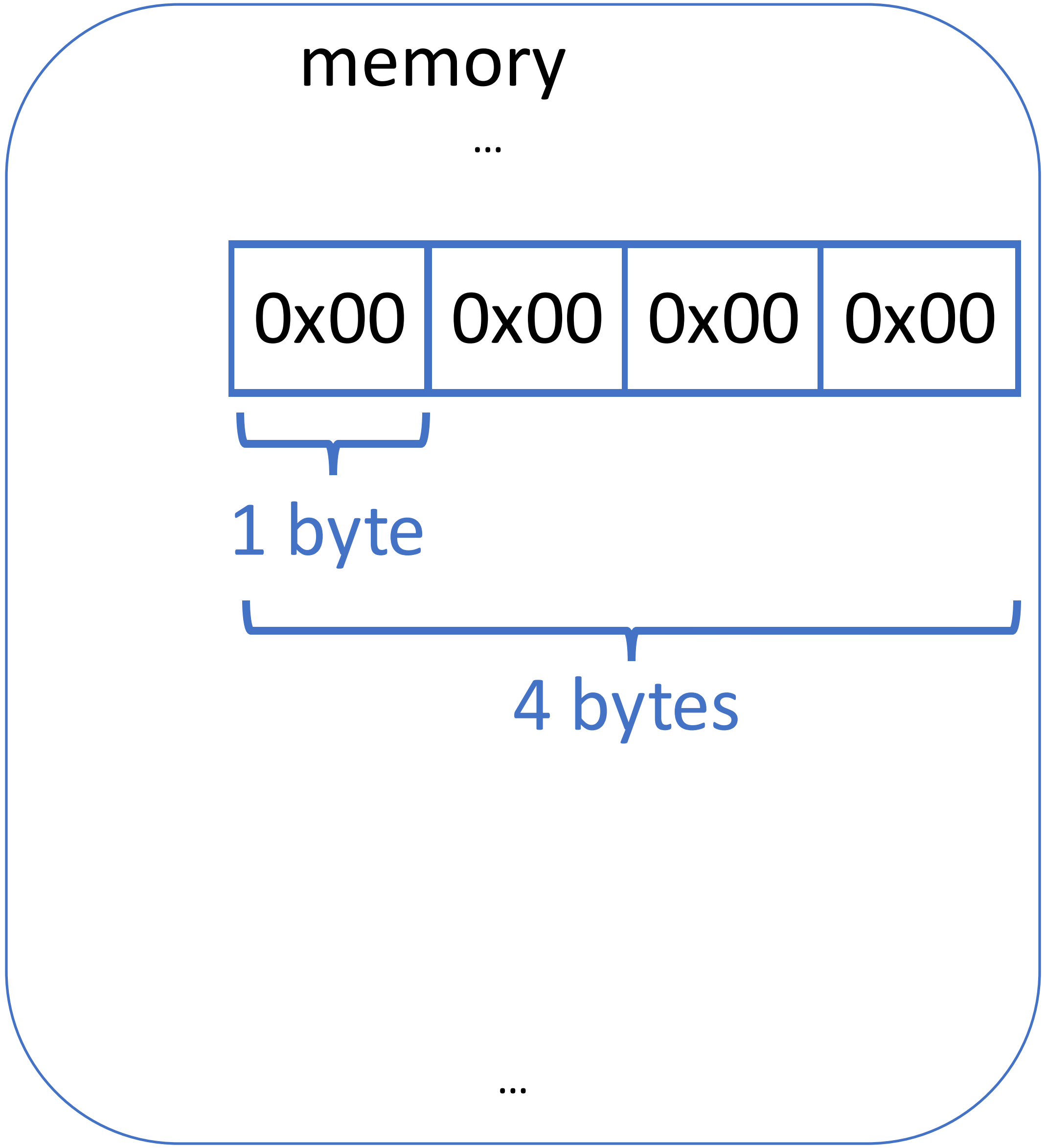
4 bytes

…

* Pointers

A pointer is a variable that stores a memory address.

```
int32_t  x  = 0;

int32_t* px;

px = &x;

// e.g. 0x7ffd39809084
```



memory

…

0x00	0x00	0x00	0x00

1 byte

4 bytes

…

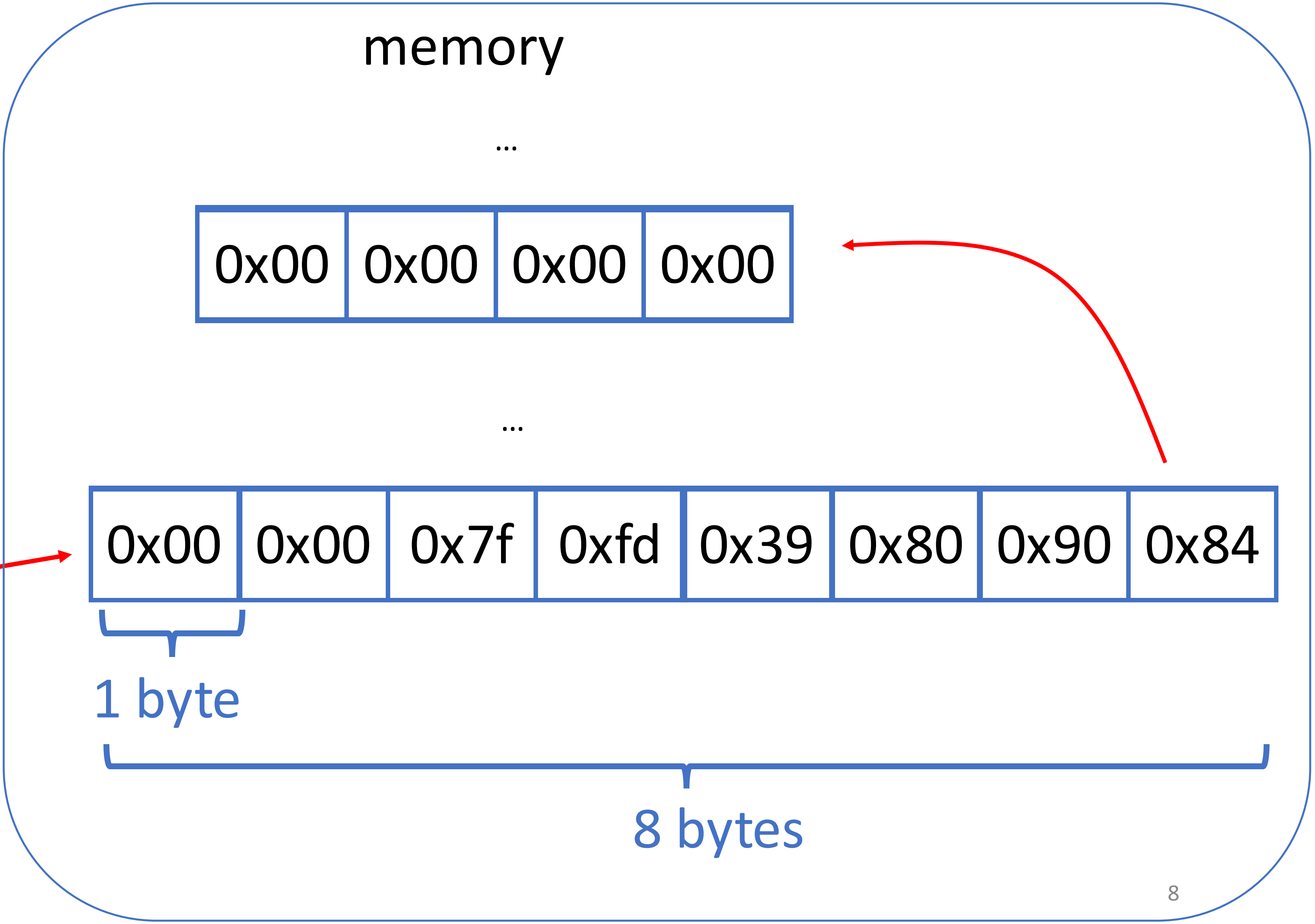
Pointers

- On **the same machine**, all pointers have the same size
  - e.g. sizes of float*, int32_t*, char*, void*, … are the same on the same machine.
- Across **different machine architectures**, pointers' sizes may differ
  - 4 bytes on 32-bit machine
  - 8 bytes on 64-bit machine

← Use this in our demonstration

* Pointers

```
int32_t  x  = 0;

int32_t* px;

px = &x;

// e.g. 0x7ffd39809084
```



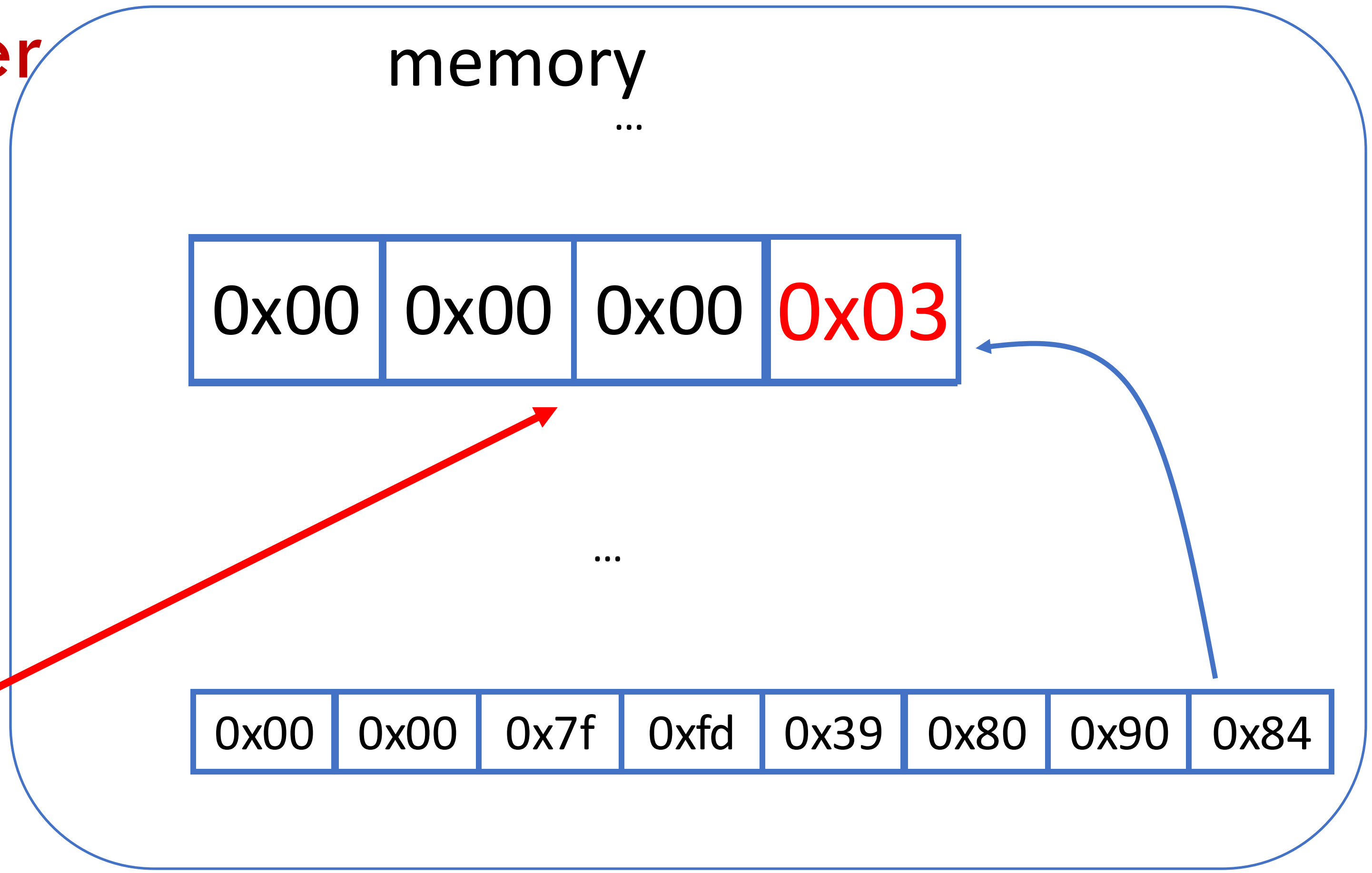
0x00 0x00 0x00 0x00

0x00 0x00 0x7f 0xfd 0x39 0x80 0x90 0x84

*

Dereference a pointer

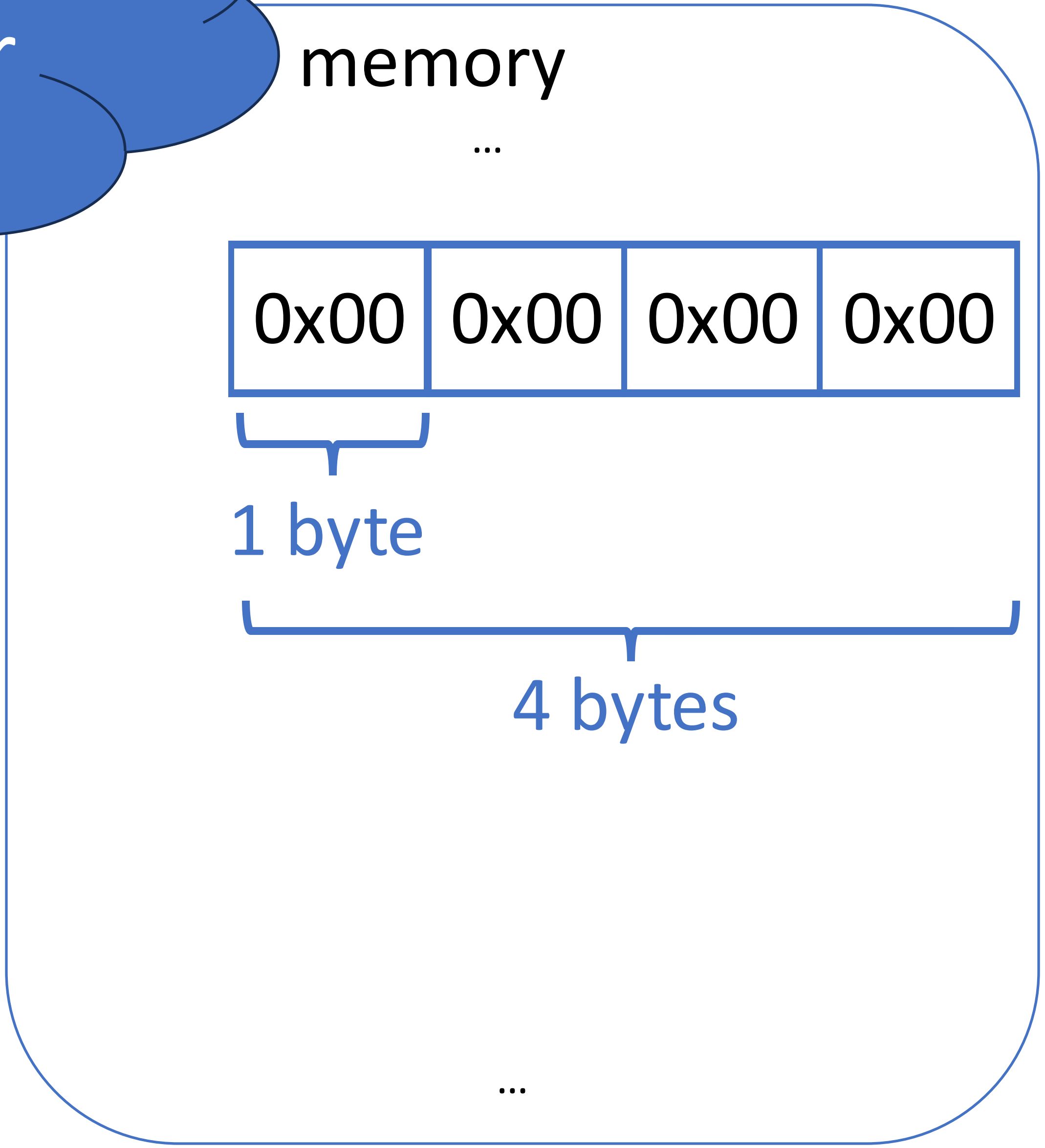
```
int32_t  x  = 0;

int32_t* px;

px = &x;

*px = 3;
```



& Reference

Can I use a different name for object x?

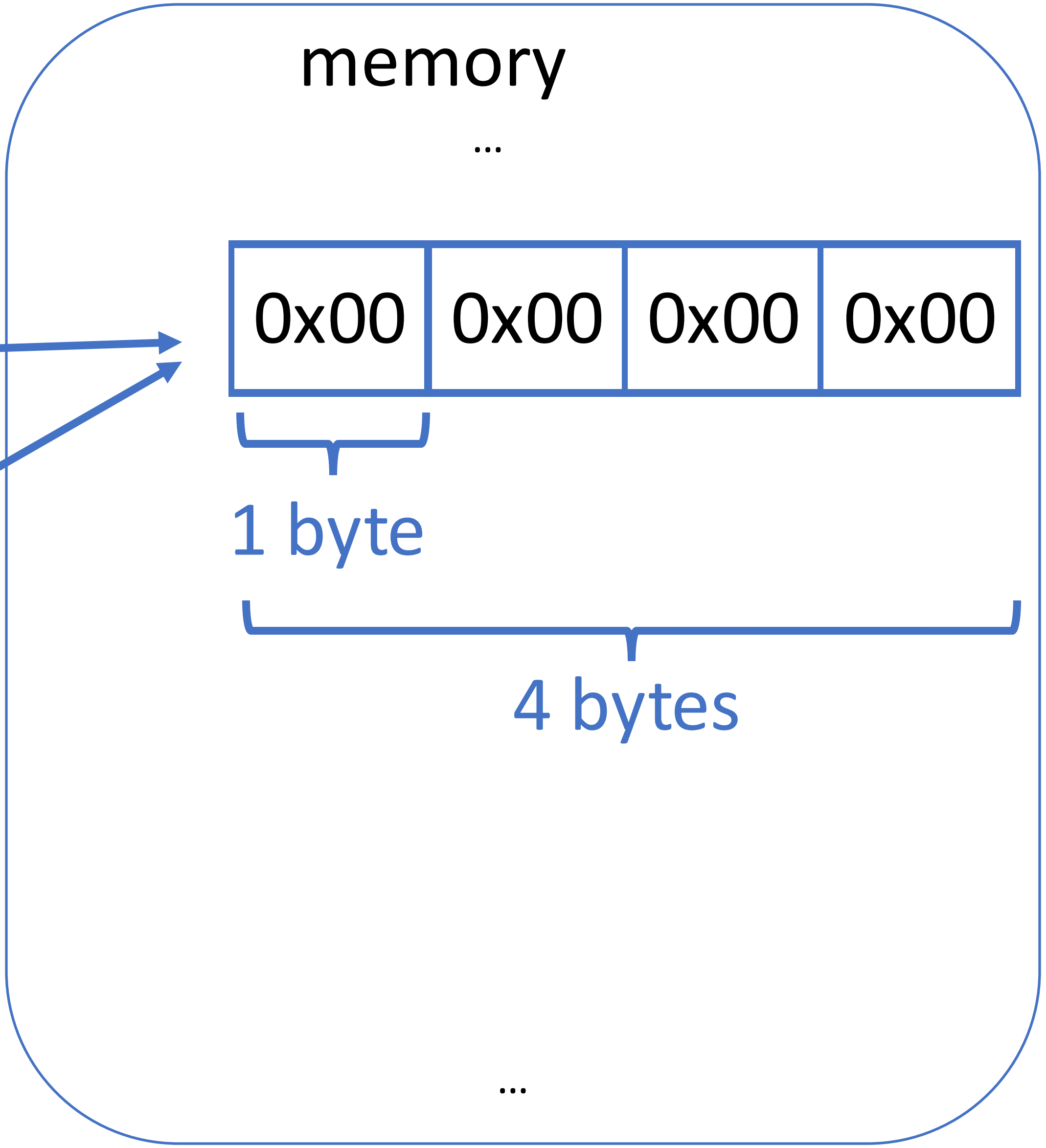
```
int32_t  x  = 0;

int32_t& ref_x = x;
```



& Reference

an **alias** to an **existing** variable

```
int32_t  x  = 0;

int32_t& ref_x = x;
```

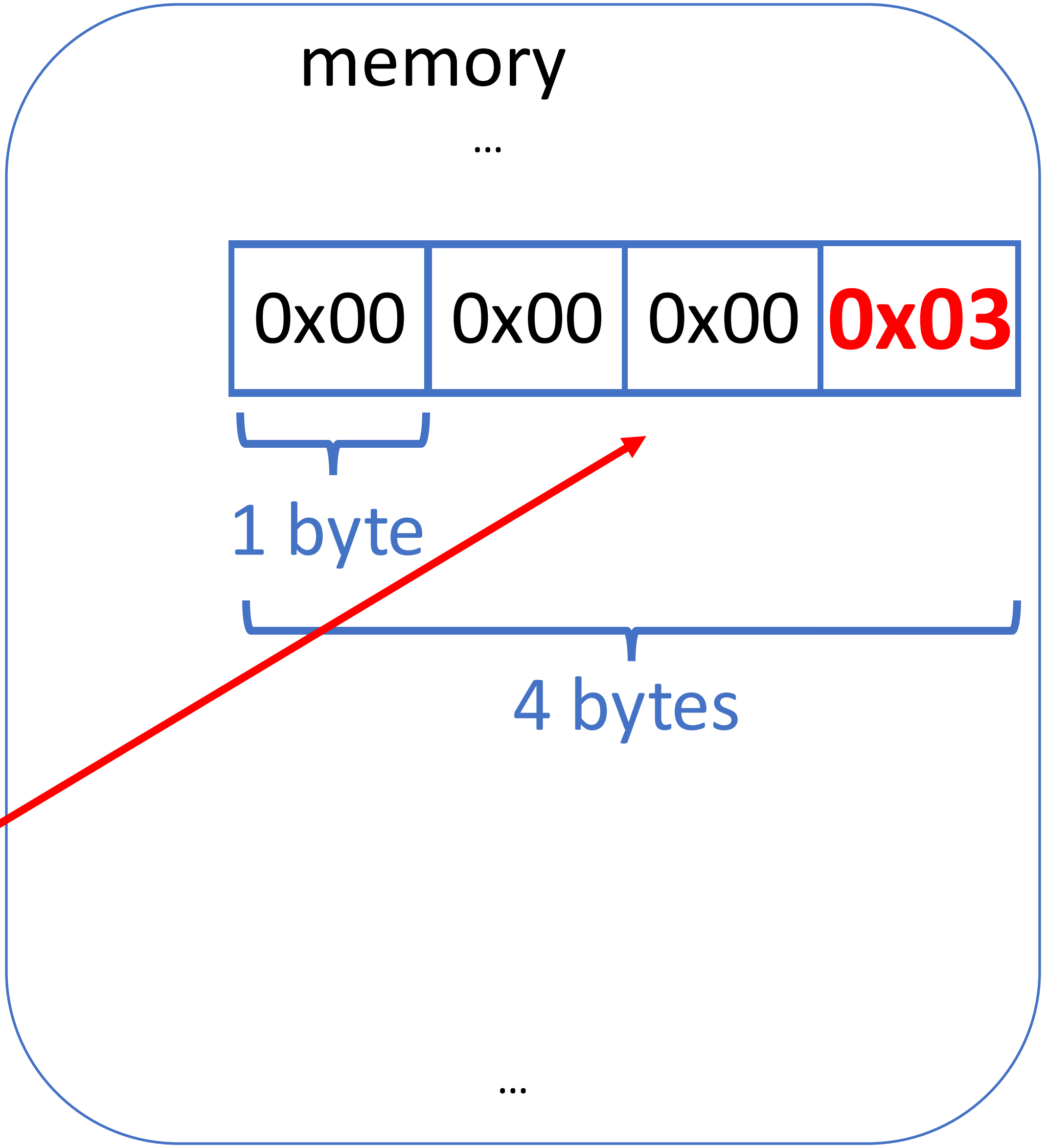


MAYBE IF I WEAR THIS SHIRT
NO ONE WILL BE ABLE TO TELL I'M SPIDERMAN

& Reference

an **alias** to an **existing** variable

```
int32_t  x  = 0;

int32_t& ref_x = x;

ref_x = 3;
```



memory

…

0x00	0x00	0x00	**0x03**

1 byte

4 bytes

…

&

Reference

an **alias** to an **existing** variable

- Cannot be NULL
- Must be initialized at time of creation

```
int32_t  x  = 0;
int32_t& ref_x;
ref_x = x;
```

Compile error! ✗

```
int32_t  x  = 0;
int32_t& ref_x = x;
```

✓

Demo

& Reference

https://pollev.com/cs4414c552

```
int  x  = 0;

int  y = 8;

int& ref = x;

ref = y;

ref = 3;
```

Now, what is x?
What is y?

& Reference

A reference is an **alias**(alternative name) to an **existing** variable

- Permanently bound to a single storage location, and cannot later be rebound

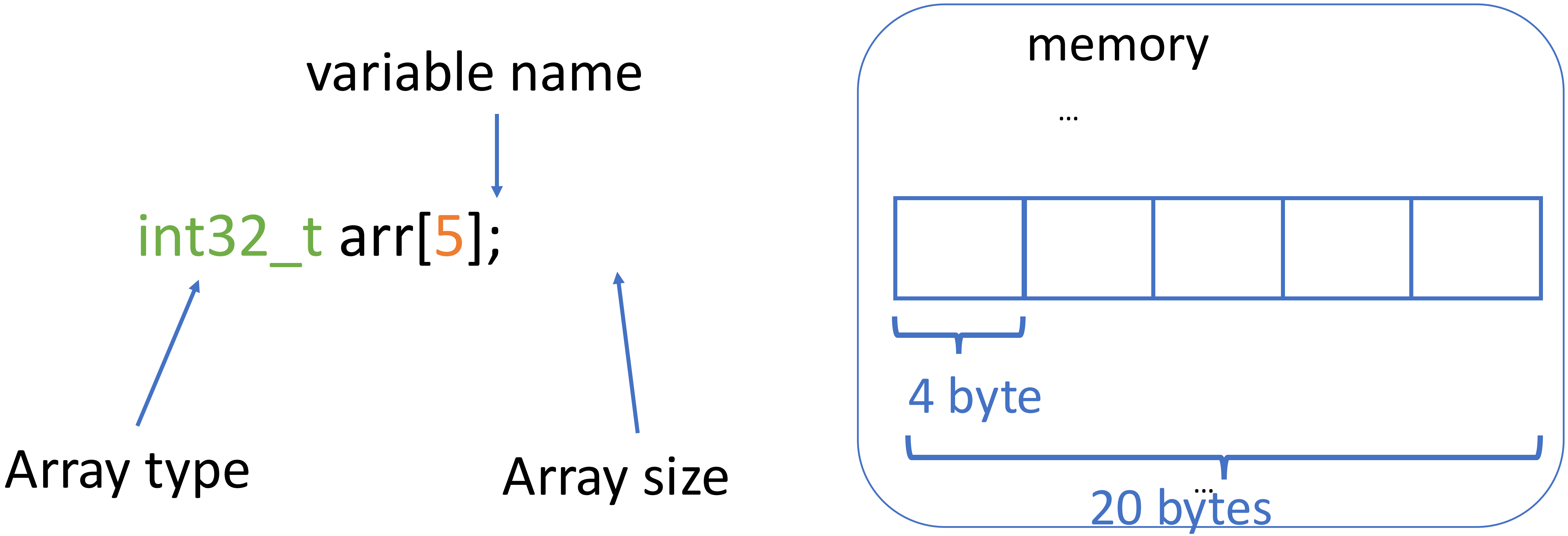
```
int  x  = 0;
int  y = 8;
int& ref = x;          // initialize ref to reference variable x
ref = y;               // assign the value in y to ref
```

Some easily confused notations

In a declaration, prefix with		In an expression, prefix with
	int a = 3;	
* = “pointer to”	int* b = &a;	& = “address of”
& = “reference to”	int& c = a;	
	int d = *b;	* = “contents of”

Fixed-size Array

- **Contiguously** allocated sequence of objects with the **same type**
- The array **size never changes** during the array lifetime.



Fixed-size Array

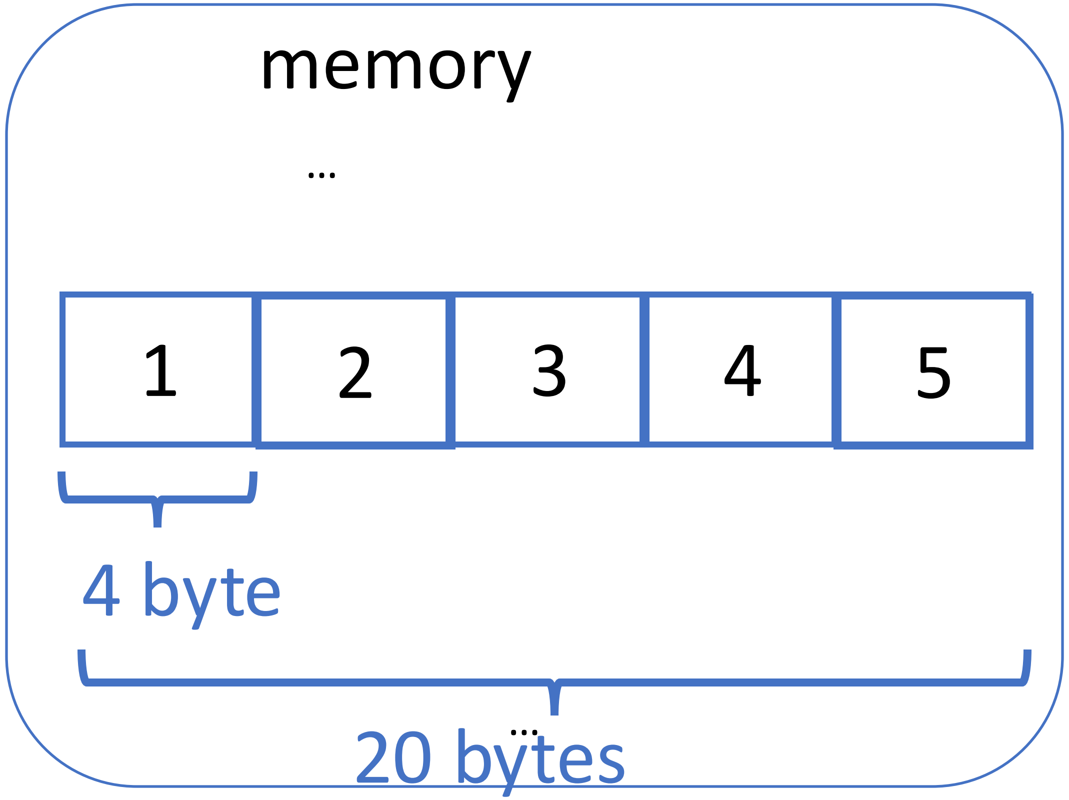
- **Initialize array**

```
int32_t arr[5]={1,2,3,4,5};
                    // declares int[5] initialized to {1,2,3,4,5};

int32_t arr[]={1,2,3,4,5};
                // compiler could deduce the size of array is 5,
                and initialized to {1,2,3,4,5};
```

Fixed-size Array

- **Indexing** array with []

```
int32_t arr[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;
```



memory

…

1	2	3	4	5

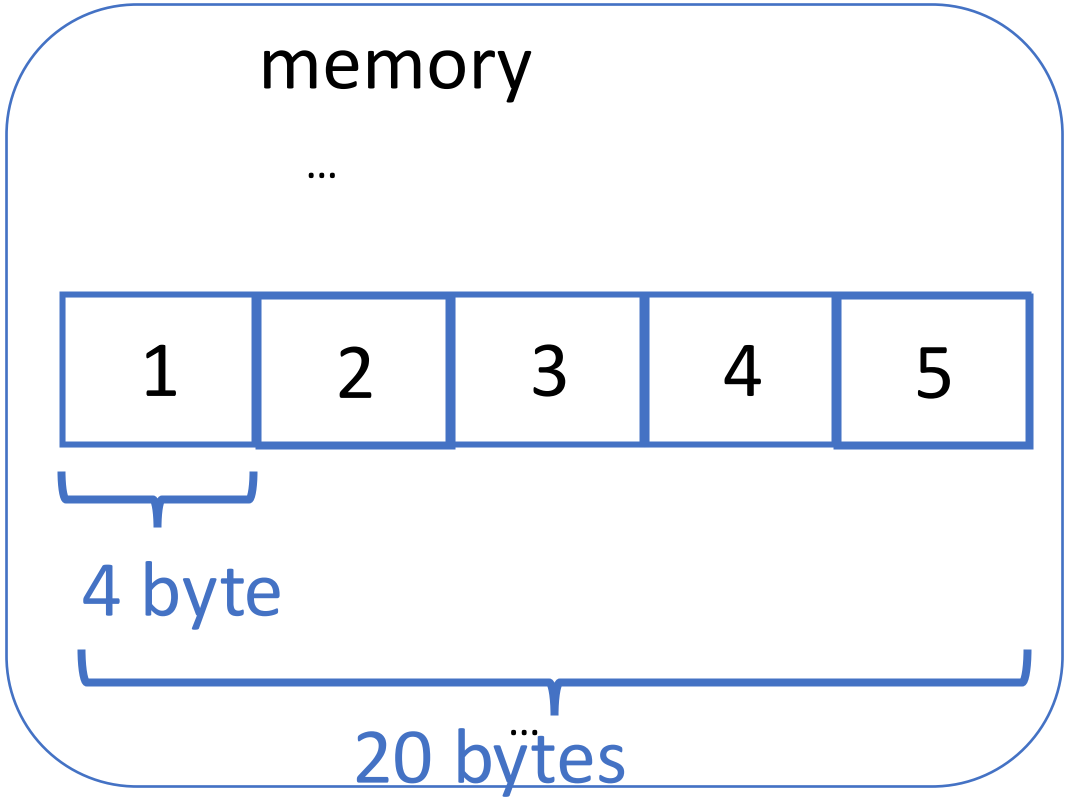
4 byte

…

20 bytes

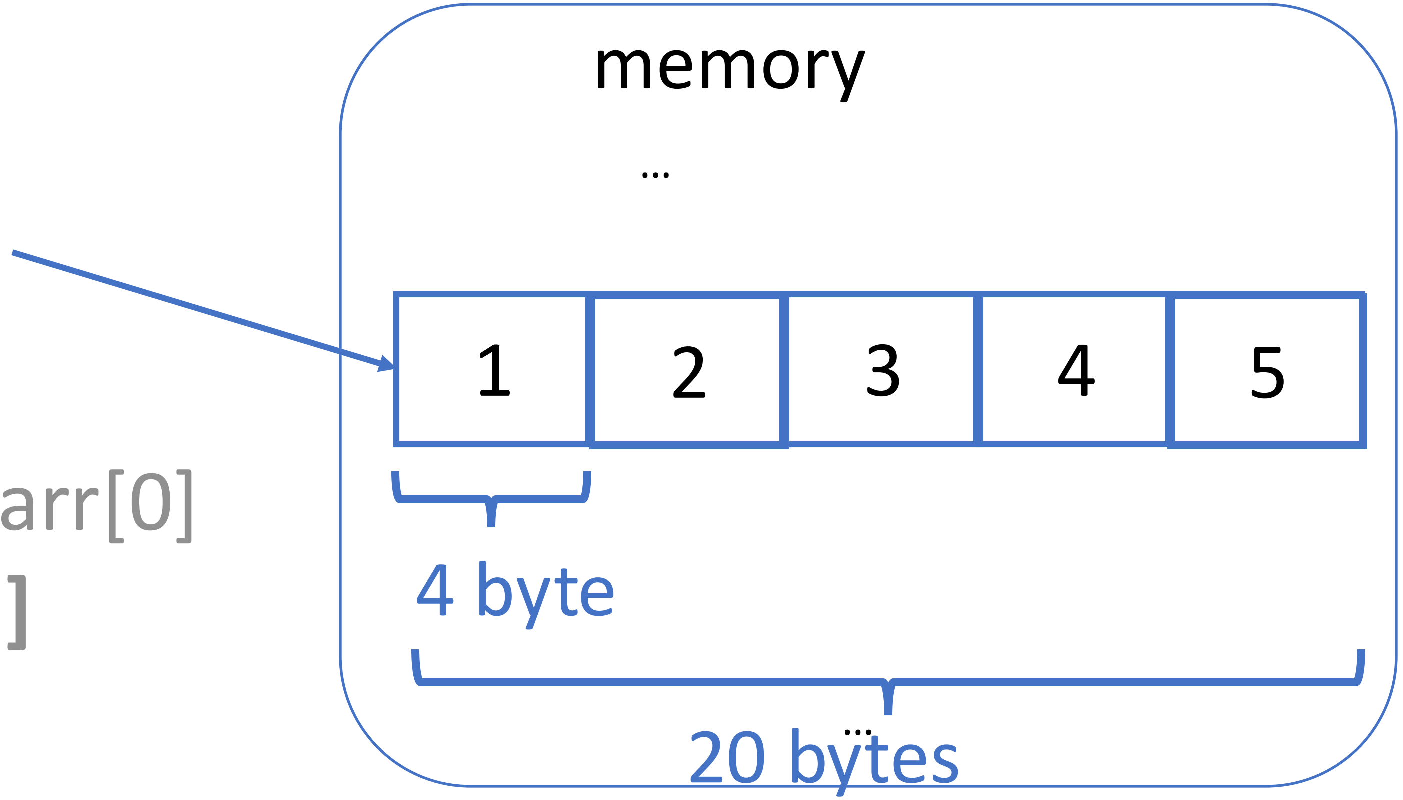
Fixed-size Array size

```
int32_t arr[5]={1,2,3,4,5};

sizeof(arr);
              ≈
sizeof(int32_t[5]);

// 20; size of array of 5 int32_t in bytes


sizeof(arr)/sizeof(int32_t);

// 5;  number of elements in array
```



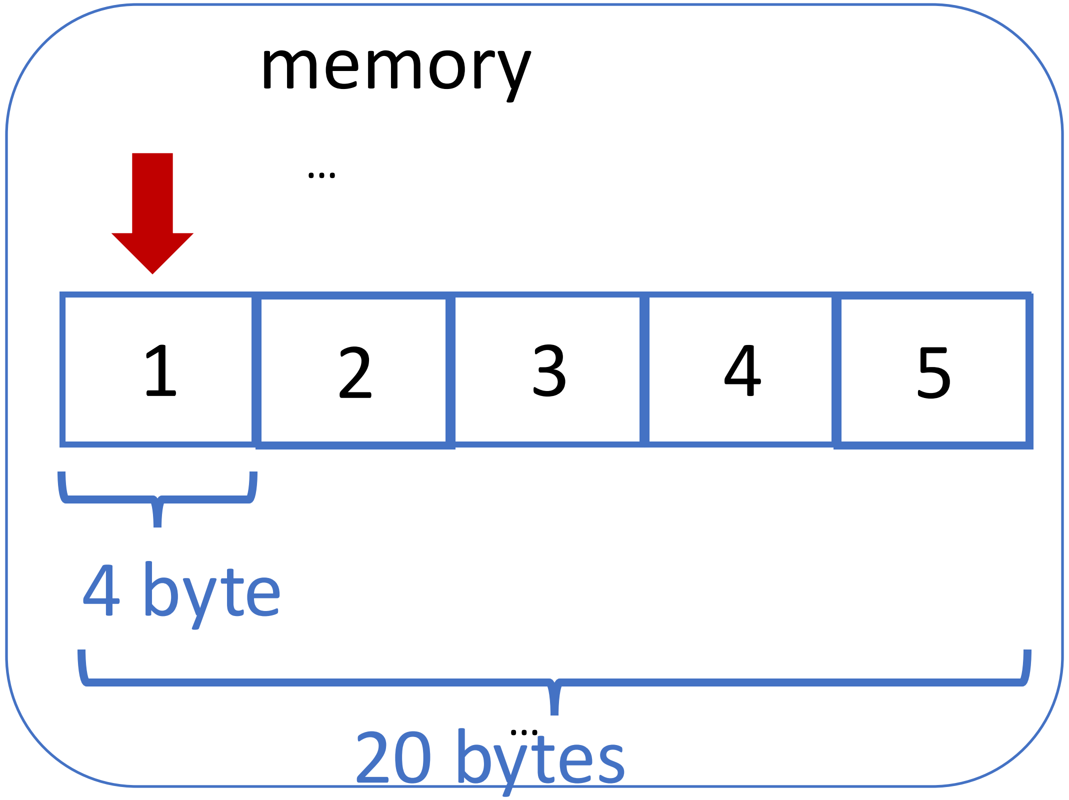
Array to pointer conversion

```
int32_t arr[5]={1,2,3,4,5};

int32_t* ptr = arr;

// ptr points to the address of arr[0]
                        i.e. &arr[0]
```



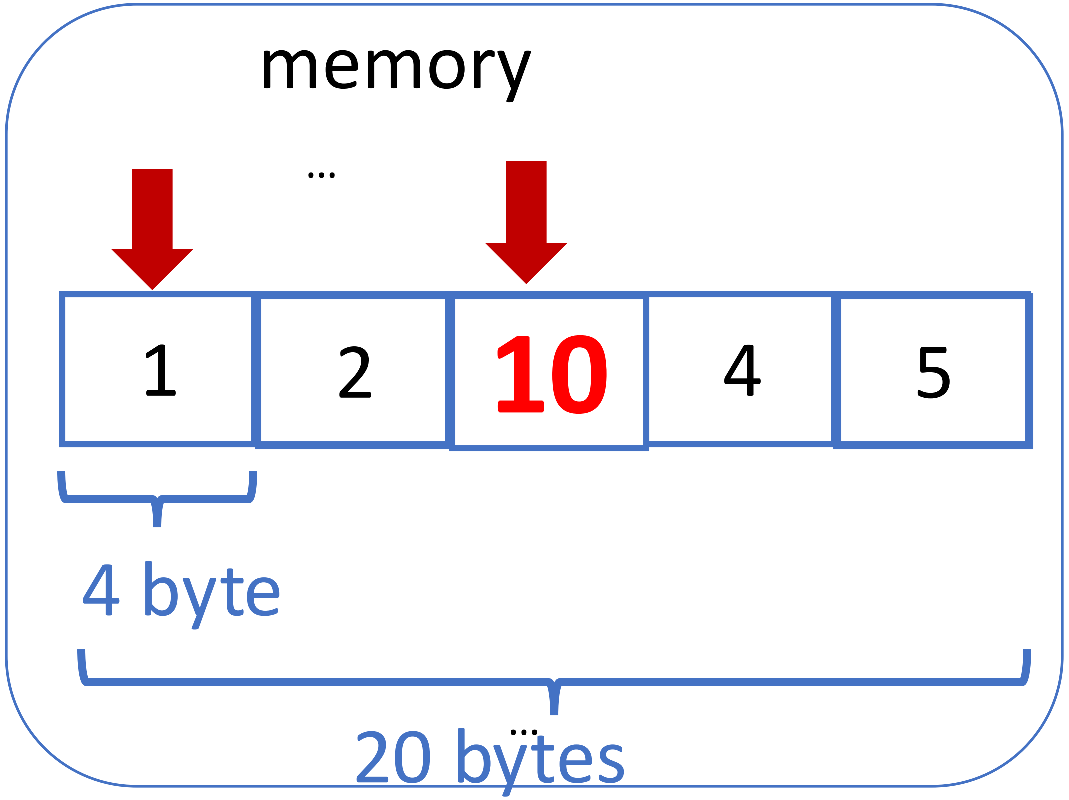
Pointer arithmetic

```
int32_t arr[5]={1,2,3,4,5};

int32_t* ptr = arr;


for (int i=0; i<5; i++){
   std::cout << *ptr << “,”;
   ptr++;
}
  // uint32_t pointer incremented by its type size
```



Pointer arithmetic

```
int32_t arr[5]={1,2,3,4,5};

int32_t* ptr = arr;

*(ptr + 2) = 10;
```



Demo

Pointer arithmetic

```
int32_t arr[5]={1,2,3,4,5};

int32_t* ptr = arr;


std::cout<< sizeof(ptr)<<std::endl;

     // Does it print out the size of array in bytes?
```

Const

- specifies that a variable's value is constant
- tells the compiler to prevent the programmer from modifying it

```
const int a;

a = 10;
```

```
const int a=0;

a = 10;

a ++;
```

Compile error!

Const with pointers

```
int x = 0;

int y = 20;
```

// points to **an object that is const**,
// the **object** can't be changed via ptr.

```
const int* ptr = &x;
```

```
*ptr = 3;
```

✗ Compile error!

```
ptr = &y;
```

✓

Const with pointers

```
int x = 0;

int y = 20;

// a const pointer
// the pointer itself cannot be changed,
but the object that ptr points to could.

int* const ptr = &x;

*ptr = 3;          ✓

ptr = &y;          ✗   Compile error!
```

Const with pointers

```
int x = 0;

int y = 20;
```

```
// a const pointer to an object that is const

const int* const ptr = &x;
```

```
*ptr = 3;
```

✗ Compile error!

```
ptr = &y;
```

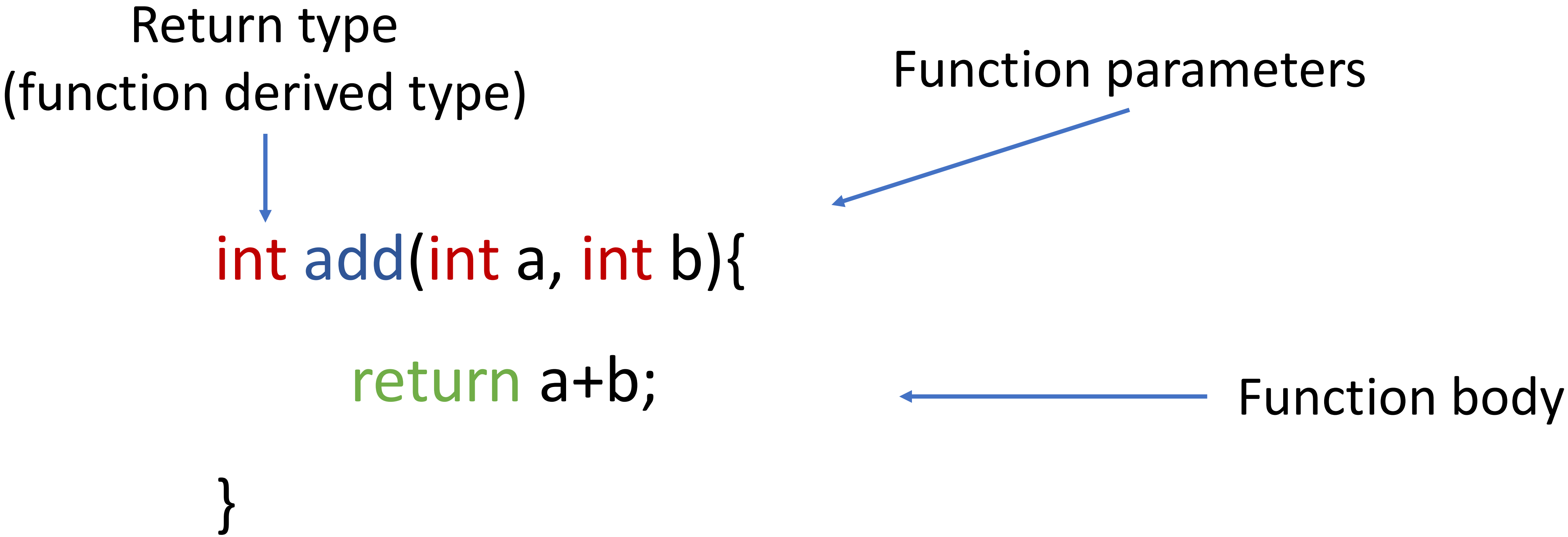
✗ Compile error!

Functions

- Function parameters
- Function returns

C++ Function

A sequence of statements with a name and a list of parameters



Return type
(function derived type)

Function parameters

```
int add(int a, int b){
    return a+b;
}
```

Function body

Function Parameter

- Pass by value : passing the copy of the value

- Pass by pointer : passing the copy of the value’s pointer

- Pass by reference : passing a reference

Function Parameter

--- Passing value

When a value is passed to a function, a copy of the value is created.

```
void increment(int value){
    value ++;
}
int main(){
        int a = 0;
        increment(a);
        …
}
```

a = ?

Function Parameter

--- Passing value

When a value is passed to a function, a copy of the value is created.

```
void increment(int value){
    value ++;
}
int main(){
        int a = 0;
        increment(a);
        std::cout << a << std::endl;
}                          // print 0
```

1. changes inside the function are **NOT** reflected after the function call.

Function Parameter

--- Passing value

When a value is passed to a function, a copy of the value is created.

```
void print_str(std::string value){
        std::cout << value << std::endl;
}
int main(){
        std::string paragraph;
        … …
        print_str(a);
}
```

2. Copying is time-consuming for large objects

Function Parameter

- Pass by value : p

What if I want to change external variable?
Can I avoid copying of parameters?

- Pass by pointer : passing the copy of the value's pointer
- Pass by reference : passing a reference

Function Parameter --- Passing pointer

Semantics:

providing the function with the **address** of the variable rather than its **value**.

- Function can **modify** the original value through dereferencing
- **Direct access** to original variable
- **Memory efficiency**

Function Parameter --- Passing pointer

```
void increment(int* a){
   (*a)++;
}

int main(){
        int a = 0;
        increment(&a);
        …
}
```

a = ?

& Function Parameter --- Passing reference

Semantics: allowing the function to operate directly on the original variable, rather than on a copy

- Function can modify the argument
- Direct access to original variable
- No copy is made

& Function Parameter

--- Passing reference

```
void increment(int& a){
   a++;
}

int main(){
      int a = 0;
      increment(a);
      …
}
```

a = ?

Function Parameter --- const

- **Const keyword** in parameter of **pointers**, depends on its location
  - **const** X* ptr : a promise that the value ptr points to cannot be changed in the function

```
void foo(const int* ptr){
        *ptr = 5;
        …
}
```

// compiler complain: here illegal to have a const pointer’s content change

Function Parameter --- const

- **Const keyword** in parameter of **pointers**, depends on its location
  - X* **const** ptr : a promise that the ptr itself cannot be changed inside the function

```
void foo(int* const ptr, int* second_ptr){
        *ptr = 5;
    ptr = second_ptr;

}
```

// compiler complain: here illegal to have a const pointer changed

& Function Parameter

--- const

- **Const keyword** in parameter of **reference**: a promise that the variable being referenced **cannot** be changed through the reference.

```
// x is a const reference

void print_str(const std::string& x)
{
        std::cout << x << std::endl;
        x = "hello";
}
```

// compile error:

a const reference **cannot** have its value changed!

Functions

- Function parameters
- Function returns

Function Returns --- value

- Return by value : returning a copy of the value

```
int value( int a ) {
        int b = a * a;
        return b;     // return a copy of b
}
```

Note: Return by value could avoid copying under compiler's C++ Return Value Optimization (RVO)

Function Returns --- pointer

Why return pointers?

- Allow direct access to memory

Demo

* Function Returns --- pointer

Correct ways of returning a pointer

- Returning a pointer to a global or static variable
- Returning a pointer to a non-local array element
- Returning a pointer from a class member function
- Returning a pointer to memory on heap

Demo

Function Returns --- pointer

Incorrect way of returning a pointer

- return a pointer to a **local variable**

Function Returns --- pointer

What can go wrong?

Dangling pointers

```
int* dangerousFunc() {
        int localVar = 100;
        return &localVar;
}

int main() {
        int* res = dangerousFunc();
        std::cout << *res << std::endl;
}
```

Undefined behavior!

✗

Function Returns --- pointer

```
int* safeFunc() {
        static int localVar = 100;
        return &localVar;
}

int main() {
        int* res = safeFunc();
        std::cout << *res << std::endl;
}
```

✓

Demo

* Function Returns --- pointer

Correct ways of returning a pointer

- **Returning a pointer to a global or static variable**
- Returning a pointer to a non-local array element
- Returning a pointer from a class member function
- Returning a pointer to memory on heap

Function Returns --- pointer

Dangling pointers

```
int* dangerousArrFunc (int index) {
    int arr[5];
    for (int i = 0; i < 5; i++)
        arr[i] = i;
    return &arr[index];
}
int main(){
    int * result = dangerousArrFunc (2);
    std::cout << " 2nd element is " << (* result )<< std::endl;
... }
```

✗

Function Returns --- pointer

```
int* dangerousArrFunc (int arr[], int index) {
    return &arr[index];
}
int main(){
     int arr[5];
     for (int i = 0; i < 5; i++)
          arr[i] = i;
     int * result = dangerousArrFunc (arr, 2);
     std::cout << " 2nd element is " << (* result )<< std::endl;
... }
```

✓

Demo

* Function Returns --- pointer

Correct ways of returning a pointer

- Returning a pointer to a global or static variable
- **Returning a pointer to a non-local array element**
- Returning a pointer from a class member function
- Returning a pointer to memory on heap

& Function Returns --- reference

Why return reference?

1. avoid copying large objects
2. Allow modification of the original object

Demo

& Function Returns --- reference

Correct ways of returning a reference

- Returning a reference to an array element
- Returning a reference to a member variable
- Returning a reference from an operator overload

Demo

& Function Returns --- reference

Incorrect way of returning a reference

- return a reference to a **local variable**

& Function Returns --- reference

```
int& dangerousFunc() {
        int localVar = 100;
        return localVar;
}

int main() {
        int& res = dangerousFunc();
        std::cout << res << std::endl;
}
```

Undefined behavior!

✗

What is C++?

A federation of related languages, with four primary sublanguages

- **C**: C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

C++ Class

C++ Class

- A class is a user-defined type
- Usually, defined by header file (.hpp) and implementation file (.cpp)
- A class can have following members
    - Data members
    - Constructor, destructor
    - Member functions
    - Copy constructor, move constructors

C++ Class

Why do we need header file?

- The **names** of program elements must **be declared before** they can be used.
- The declaration **tells the compiler** the type of an element.
- C++ use header files to contain declarations, use the **#include** in other files that requires the declarations.

C++ Class --- header file

rectangle.hpp

```
#pragma once
class Rectangle{
        float width;
        float length;
        float area;
public:
        Rectangle();
        Rectangle(float w, float l);
        ~Rectangle();
        float& getArea();
        ... };
```

Data members

For compiler, indicate that it only be parsed once.

C++ Class rectangle.cpp --- implementation file

```
#include "rectangle.hpp”


Rectangle::Rectangle(){
        … …
}
Rectangle::Rectangle(float w, float l){
        … …
}
float& Rectangle::getArea(){
… }
```

Scope resolution operator

Demo

C++ Class

- A class is a user-defined type
- Usually, defined by header file (.hpp) and implementation file (.cpp)
- A class can have following members
  - Data members
  - **Constructor, destructor**
  - Member functions
  - Copy constructor, move constructors

Constructor: construct and initialize objects of that class

- Default constructor: a constructor can be called with no argument

```
Rectangle::Rectangle():
            width(0),
            length(0),
            area(0)
{
        // Constructor body (can be empty or contain additional logic)


}
```

Initializer list

Constructor: construct and initialize objects of that class

- Implicit default constructor:
  - If there is no user-declared constructor for a class type, **the compiler** will implicitly declare a default constructor as an inline public class member.

Constructor: construct and initialize objects of that class

Parameterized constructor: a constructor that accepts argument

```
Rectangle::Rectangle(int w, int l):
                  width(w),
                  length(l)
{
        area = _width * _length;
}
```

Note: When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly create the default constructor

Constructor: construct and initialize objects of that class

Parameterized constructor: a constructor that accepts argument

Note:

- When the **parameterized constructor is defined** and no default constructor is defined explicitly, the compiler will **NOT** implicitly create the default constructor

Constructor: construct and initialize objects of that class

```
#include "rectangle.hpp”



int main(){

        Rectangle rec;

        Rectangle explicit_rec = Rectangle(10.0,12.0);

        Rectangle implicit_rec(30.0, 28.1);

}
```

Constructor: const data member

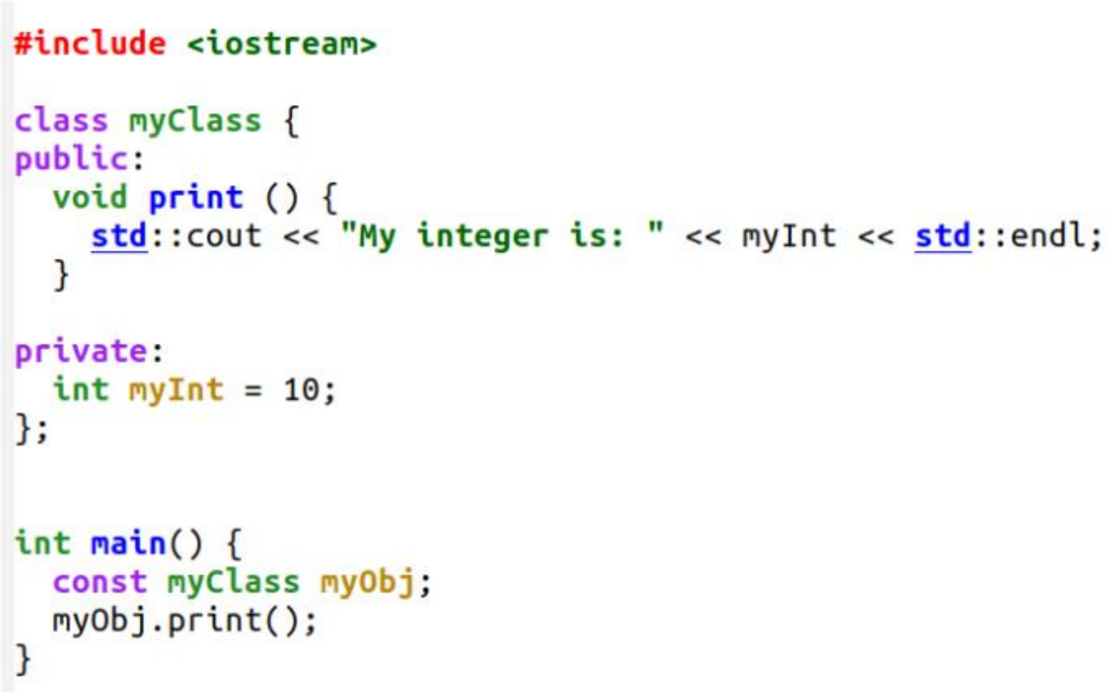
```
class Rectangle{
        const float width;
        const float length;
        const float area;
public:
        Rectangle(float w, float l, float a):
                        width(w), length(l), area(a)
        {}
};
```

Constructor: const data member

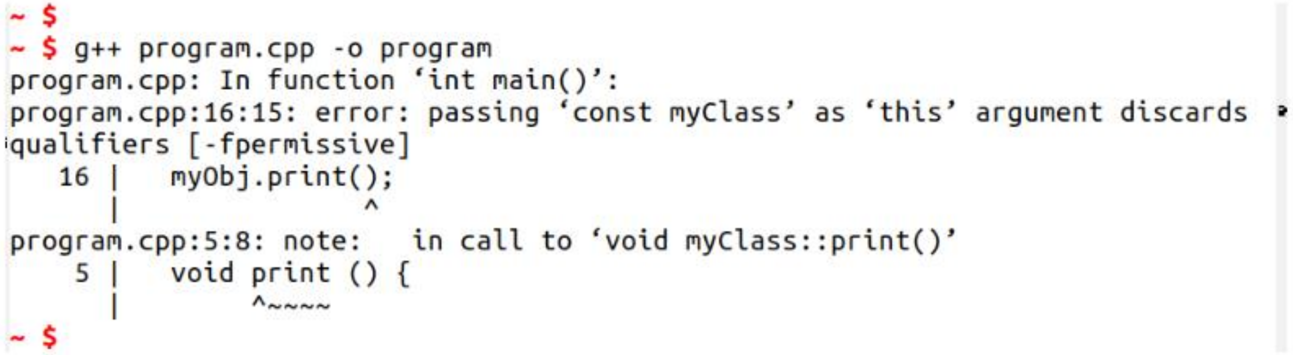
```
class Rectangle{
        const float width;
        const float length;
        const float area;
public:
        Rectangle(float w, float l, float a)     {
          width=w;
          length = l;
          area = a;
   }
};
```

Compilation error ✗

Exercise: Explain the error

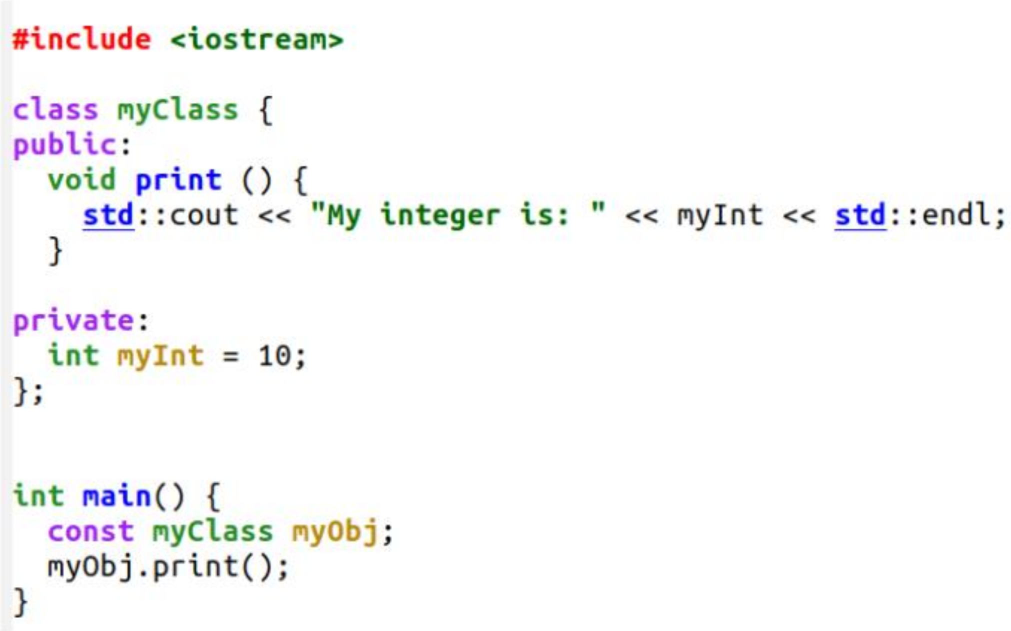


```cpp
#include <iostream>

class myClass {
public:
  void print () {
    std::cout << "My integer is: " << myInt << std::endl;
  }

private:
  int myInt = 10;
};


int main() {
  const myClass myObj;
  myObj.print();
}
```

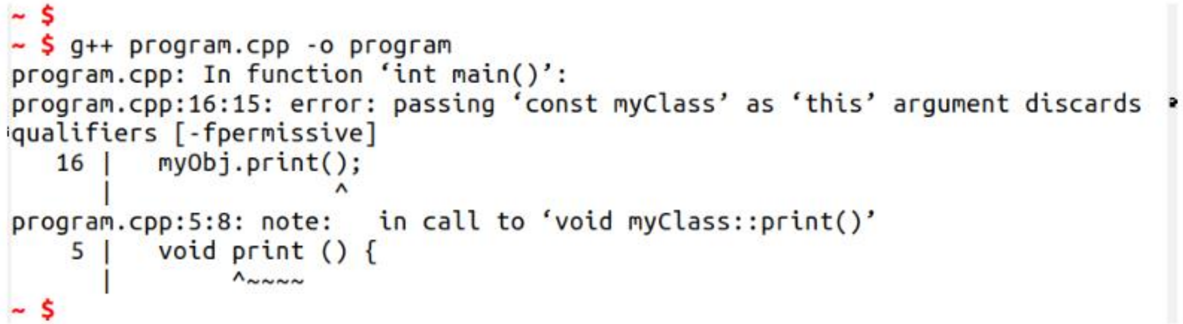


```
~ $
~ $ g++ program.cpp -o program
program.cpp: In function ‘int main()’:
program.cpp:16:15: error: passing ‘const myClass’ as ‘this’ argument discards
qualifiers [-fpermissive]
   16 |   myObj.print();
      |               ^
program.cpp:5:8: note:   in call to ‘void myClass::print()’
    5 |   void print () {
      |        ^~~~~
~ $
```

Exercise: Explain the error



```cpp
#include <iostream>

class myClass {
public:
  void print () {
    std::cout << "My integer is: " << myInt << std::endl;
  }

private:
  int myInt = 10;
};


int main() {
  const myClass myObj;
  myObj.print();
}
```



```
~ $
~ $ g++ program.cpp -o program
program.cpp: In function ‘int main()’:
program.cpp:16:15: error: passing ‘const myClass’ as ‘this’ argument discards
qualifiers [-fpermissive]
   16 |   myObj.print();
      |               ^
program.cpp:5:8: note:   in call to ‘void myClass::print()’
    5 |   void print () {
      |        ^~~~~
~ $
```

- Print function can potentially change the state of a myClass Object, so it cannot be called on a const object
- To assert that print cannot change object state, change it to void print () const {}

References and readings

- A Tour of C++, Bjarne Stroustrup. Chapter 1.4, 1.8, 2.3.
- C++ fundamental types: https://en.cppreference.com/w/cpp/language/types#void
- Reference: https://en.cppreference.com/w/cpp/language/reference
- Array and pointer arithmetic:

  https://faculty.cs.niu.edu/~mcmahon/CS241/Notes/arrays_and_pointers.html
- C++ class: https://en.cppreference.com/w/cpp/language/classes
- C++ header files: https://learn.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-170
- Effective C++, Meyers, Scott. Chapter 2. Constructors, Destructors, and Assignment Operators