# C++ Programming

# Week 8: Advanced Topics

*Dr. Owen Chen*
*Cary Chinese School*
*Director of Math and Computer Science*

2023 Summer

# Week 8: Agenda

- **Review Week 7 – Classes – Part II**
  - Class Constructor/Destructor
  - Class Hierarchy/Class Inheritance
  - Class Keywords
  - **Review Homework 7)**
- **Week 8: Advanced Topics**
  - Initialization
  - Pointers and References
  - Read from and write to a file
  - Heap and Stack

# Review on Week 7 - C++ Classes

- **Classes** are the most fundamental feature in C++. Classes let us define new types for our applications, making our programs shorter and easier to modify.
- **Data abstraction**—the ability to define both **data** and **function** members.
- **Encapsulate** a class by defining its implementation members as **private**.
- Classes may grant access to their nonpublic member by designating another class or function as a **friend**.
- Classes may define **constructors**, which are special member functions that control how objects are initialized. Constructors may be **overloaded**.
- Classes may define a single **destructor**, which is a special member function that releases memory when an object is destroyed.

# Class Constructor

**Constructor [ctor]**

A **constructor** is a *special* member function of a class that is executed when a new instance of that class is created

Goals: *initialization* and *resource acquisition*

Syntax: `T(...)` same named of the class and no return type

- A *constructor* is supposed to initialize *all* data members
- We can define *multiple constructors* with different signatures

# Copy Constructor

## Copy Constructor

A **copy constructor** T(const T&) creates a new object as a *deep copy* of an existing object

```
class A {
    A()         {} // default constructor
    A(int)      {} // non-default constructor
    A(const A&) {} // copy constructor
}
```

- Every class always defines an *implicit* or *explicit* copy constructor
- Even the copy constructor implicitly calls the *default* Base class constructor
- Even the copy constructor is considered a non-default constructor

# Class Destructor

### Destructor

A **destructor** is a special member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed. Destructor release memory space occupied by the objects created by the constructor.

Goals: *resources releasing*

Syntax: $\sim\mathbf{T()}$ same name of the class and no return type

- Any object has exactly one *destructor*, which is always *implictly* or *explicitly* Declared
- If a destructor is not defined for a class, compiler will automatically create a default one.

# Class Hierarchy

1/3

### Child/Derived Class or Subclass

A new class that inheriting variables and functions from another class is called a **derived** or **child** class

### Parent/Base Class

The *closest* class providing variables and functions of a derived class is called **parent** or **base** class

**Extend** a *base class* refers to creating a new class which retains characteristics of the base class and *on top it can add* (and never remove) its own members

**Syntax:**

```
class DerivedClass : [<inheritance attribute>] BaseClass {
```

# this Keyword

## this

Every object has access to its own address through the const pointer this

Explicit usage is not mandatory (and not suggested)

this is necessary when:

- The name of a local variable is equal to some member name
- Return reference to the calling object

```
class A {
    int x;
    void f(int x) {
        this->x = x; // without "this" has no effect
    }
    const A& g() {
        return *this;
    }
};
```

# const Keyword 1/2

## Const member functions

**Const member functions** (**inspectors** or **observer**) are functions marked with `const` that are not allowed to change the object state

Member functions without a `const` suffix are called *non-const member functions* or **mutators**. The compiler prevents from inadvertently mutating/changing the data members of *observer* functions

```
class A {
    int x = 3;

    int get() const {
     // x = 2;    // compile error class variables cannot be modified
        return x;
    }
};
```

# friend Keyword 1/3

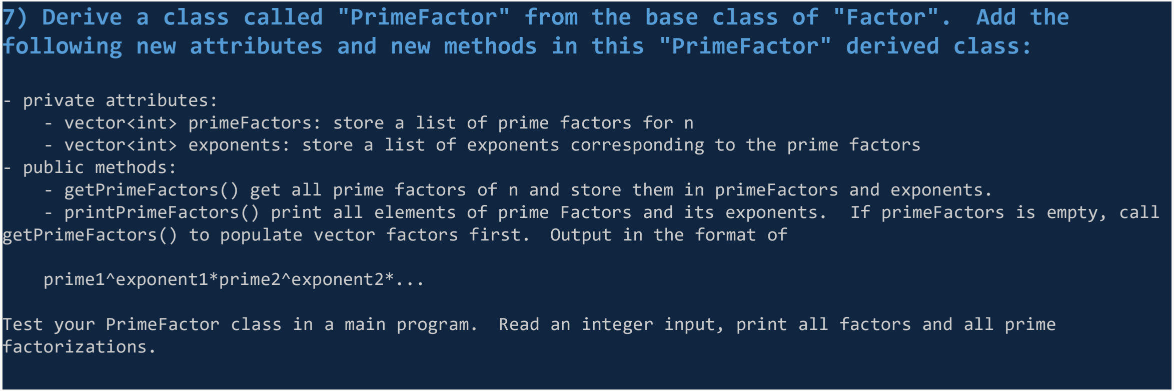
### friend Class

A friend class can access the private and protected members of the class in which it is declared as a friend

Friendship properties:

- **Not Symmetric**: if class A is a friend of class B, class B is not automatically a friend of class A
- **Not Transitive**: if class A is a friend of class B, and class B is a friend of class C, class A is not automatically a friend of class C
- **Not Inherited**: if class Base is a friend of class X, subclass Derived is not automatically a friend of class X; and if class X is a friend of class Base, class X is not automatically a friend of subclass Derived

# Homework 7



**7) Derive a class called "PrimeFactor" from the base class of "Factor". Add the following new attributes and new methods in this "PrimeFactor" derived class:**

- private attributes:
    - vector<int> primeFactors: store a list of prime factors for n
    - vector<int> exponents: store a list of exponents corresponding to the prime factors
- public methods:
    - getPrimeFactors() get all prime factors of n and store them in primeFactors and exponents.
    - printPrimeFactors() print all elements of prime Factors and its exponents. If primeFactors is empty, call getPrimeFactors() to populate vector factors first. Output in the format of

```
    prime1^exponent1*prime2^exponent2*...
```

Test your PrimeFactor class in a main program. Read an integer input, print all factors and all prime factorizations.

# Homework 7 - Solution

- **Source Code:**
- **https://github.com/owenjchen/cpp/tree/main/week7/homework**

- **factor.h**
- **factor.cpp**
- **primeFactor.h**
- **primeFactor.cpp**
- **primeFactor_main.cpp**
- **Makefile**

# Initialization

# Variable Initialization

C++03:

```
int a1;             // default initialization (undefined value)

int a2(2);          // direct (or value) initialization
int a3(0);          // direct (or value) initialization (zero-initialization)
// int a4();        // a4 is a function

int a5 = 2;         // copy initialization
int a6 = 2u;        // copy initialization (+ implicit conversion)
int a7 = int(2);    // copy initialization
int a8 = int();     // copy initialization (zero-initialization)

int a9 = {2};       // copy list initialization
```

# Uniform Initialization

C++11 **Uniform Initialization** syntax, also called *brace-initialization* or *braced-init-list*, allows to initialize different entities (variables, objects, structures, etc.) in a consistent way:

```
int b1{2};          // direct list (or value) initialization
int b2{};           // direct list (or value) initialization (zero-initialization)

int b3 = int{};  // copy initialization (zero-initialization)
int b4 = int{4}; // copy initialization

int b5 = {};        // copy list initialization (zero-initialization)
```

# Brace Initialization Advantages

The **uniform initialization** can be also used to *safely* convert arithmetic types, preventing implicit *narrowing*, i.e potential value loss. The syntax is also more concise than modern casts

```
int         b4 = -1;  // ok
int         b5{-1};   // ok
unsigned    b6 = -1;  // ok
//unsigned b7{-1};    // compile error

float    f1{10e30};   // ok
float    f2 = 10e40;  // ok, "inf" value
//float  f3{10e40};   // compile error
```

# Fixed-Size Array Initialization

One dimension:

```
int  a[3] = {1, 2, 3}; // explicit size
int  b[]  = {1, 2, 3}; // implicit size
char c[]  = "abcd";    // implicit size
int  d[3] = {1, 2};    // d[2] = 0 -> zero/default value

int  e[4] = {0};       // all values are initialized to 0
int  f[3] = {};        // all values are initialized to 0 (C++11)
int  g[3] {};          // all values are initialized to 0 (C++11)
```

Two dimensions:

```
int a[][2] = { {1,2}, {3,4}, {5,6} }; // ok
int b[][2] = { 1, 2, 3, 4 };          // ok
// the type of "a" and "b" is an array of type int[]
// int c[][]  = ...;                  // compile error
// int d[2][] = ...;                  // compile error
```

# Structure Initialization - C++03 1/4

```
struct S {
    unsigned x;
    unsigned y;
};

S s1;          // default initialization, x,y undefined values
S s2 = {};     // copy list initialization, x,y zero/default-initialization
S s3 = {1, 2}; // copy list initialization, x=1, y=2
S s4 = {1};    // copy list initialization, x=1, y zero/default-initialization
//S s5(3, 5);  // compiler error, constructor not found

S f() {
    S s6 = {1, 2}; // verbose
    return s6;
}
```

# Structure Initialization - C++11 2/4

```cpp
struct S {
    unsigned x;
    unsigned y;
    void*    ptr;
};

S s1{};         // direct list (or value) initialization
                //    x,y,ptr zero/default-initialization

S s2{1, 2};     // direct list (or value) initialization
                //    x=1, y=2, ptr zero/default-initialization

// S s3{1, -2}; // compile error, narrowing conversion

S f() { return {3, 2}; } // non-verbose
```

# Structure Initialization - Brace or Equal Initialization 3/4

**Non-Static Data Member Initialization** (NSDMI), also called *brace or equal initialization*:

```
struct S {
    unsigned x = 3; // equal initialization
    unsigned y = 2; // equal initialization
};
struct S1 {
    unsigned x {3}; // brace initialization
};
//----------------------------------------------------------------------------------
S s1;       // call default constructor (x=3, y=2)
S s2{};     // call default constructor (x=3, y=2)
S s3{1, 4}; // set x=1, y=4
```

# Structure Initialization - Designated Initializer List 4/4

C++20 introduces *designated initializer list*

```
struct A {
    int x, y, z;
};
A a1{1, 2, 3};                // is the same of
A a2{.x = 1, .y = 2, .z = 3}; // designated initializer list
```

*Designated initializer list* can be very useful for improving code readability

```
void f1(bool a, bool b, bool c, bool d, bool e) {}
// long list of the same data type -> error prone

struct B {
    bool a, b, c, d, e;
};                           // f2(B b)
f2({.a = true, .c = true});  // b, d, e = false
```

# Structure Binding

*Structure Binding* declaration C++17 binds the specified names to elements of initializer:

```
struct A {
    int x = 1;
    int y = 2;
} a;

A f() { return A{4, 5}; }
// Case (1): struct
auto [x1, y1] = a;      // x1=1, y1=2
auto [x2, y2] = f();    // x2=4, y2=5
// Case (2): raw arrays
int  b[2]     = {1,2};
auto [x3, y3] = b;      // x3=1, y3=2
// Case (3): tuples
auto [x4, y4] = std::tuple<float, int>{3.0f, 2};
```

# Dynamic Memory Initialization

C++03:

```
int* a1 = new int;            // undefined
int* a2 = new int();          // zero-initialization, call "= int()"
int* a3 = new int(4);         // allocate a single value equal to 4
int* a4 = new int[4];         // allocate 4 elements with undefined values
int* a5 = new int[4]();       // allocate 4 elements zero-initialized, call "= int()"
// int* a6 = new int[4](3); // not valid
```

C++11:

```
int* b1 = new int[4]{};       // allocate 4 elements zero-initialized, call "= int{}"
int* b2 = new int[4]{1, 2}; // set first, second, zero-initialized
```

# Pointers and References

# Pointers and Pointer Operations 1/3

## Pointer

A **pointer** $T^*$ is a value referring to a location in memory

## Pointer Dereferencing

Pointer **dereferencing** ($^*ptr$) means obtaining the value stored in at the location refereed to the pointer

## Subscript Operator []

The subscript operator ($ptr[]$) allows accessing to the pointer element at a given position

# Pointers and Pointer Operations 2/3

The **type of a pointer** (e.g. `void*`) is an *unsigned* integer of 32-bit/64-bit depending on the underlying architecture

- It only supports the operators `+, -, ++, --`, comparisons `==, !=, <, <=, >, >=`, subscript `[]`, and deferencing `*`
- A pointer can be *explicitly* converted to an integer type

```
void*  x;
size_t y = (size_t) x; // ok (explicit conversion)
// size_t y = x;        // compile error (implicit conversion)
```

# Pointer Conversion

- Any pointer type can be implicitly converted to `void*`
- Non-`void` pointers must be explicitly converted
- `static_cast`† is not allowed for pointer conversion for safety reasons, except for `void*`

```
int*  ptr1 = ...;
void* ptr2 = ptr1;         // int* -> void*, implicit conversion

void* ptr3 = ...;
int*  ptr4 = (int*) ptr3;  // void* -> int, explicit conversion required
                           // static_cast allowed
int*  ptr5 = ...;
char* ptr6 = (char*) ptr5; // int* -> char*, explicit conversion required,
                           // static_cast not allowed, dangerous
```

† see next lectures for `static_cast` details

# Pointers and Pointer Operations 3/3

Deferencing:

```
int* ptr1 = new int;
*ptr1     = 4;     // deferencing (assignment)
int a     = *ptr1; // deferencing (get value)
```

Array subscript:

```
int* ptr2 = new int[10];
ptr2[2]   = 3;
int var   = ptr2[4];
```

Common error:

```
int *ptr1, ptr2;  // one pointer and one integer!!
int *ptr1, *ptr2; // ok, two pointers
```

# 1 + 1 ≠ 2 : Pointer Arithmetic 1/2

## Subscript operator meaning:

`ptr[i]` is equal to `*(ptr + i)`

Note: subscript operator accepts also negative values

## Pointer arithmetic rule:

```
address(ptr + i) = address(ptr) + (sizeof(T) * i)
```

where T is the type of elements pointed by ptr

```
int array[4] = {1, 2, 3, 4};
cout << array[1];    // print 2
cout << *(array + 1);// print 2
cout << array;          // print 0xFFFAFFF2
cout << array + 1;      // print 0xFFFAFFF6!!
int* ptr = array + 2;
cout << ptr[-1];        // print 2
```

# $1 + 1 \neq 2$ : Pointer Arithmetic 2/2

```
char arr[4] = "abc"
```

| value | address | |
|---|---|---|
| ’a’ | 0x0 | ←arr[0] |
| ’b’ | 0x1 | ←arr[1] |
| ’c’ | 0x2 | ←arr[2] |
| ’\0’ | 0x3 | ←arr[3] |

```
int arr[3] = {4,5,6}
```

| value | address | |
|---|---|---|
| 4 | 0x0 | ←arr[0] |
| | 0x1 | |
| | 0x2 | |
| | 0x3 | |
| 5 | 0x4 | ←arr[1] |
| | 0x5 | |
| | 0x6 | |
| | 0x7 | |
| 6 | 0x8 | ←arr[2] |
| | 0x9 | |
| | 0x10 | |
| | 0x11 | |

# Address-of operator &

The **address-of operator** (&) returns the address of a variable

```
int  a = 3;
int* b = &a; // address-of operator,
             // 'b' is equal to the address of 'a'
a++;
cout << *b;  // print 4;
```

To not confuse with **Reference syntax:** `T&var=`

# Wild and Dangling Pointers

## Wild pointer:

```
int main() {
    int* ptr;   // wild pointer: Where will this pointer points?
    ...         // solution: always initialize a pointer
}
```

## Dangling pointer:

```
int main() {
    int* array = new int[10];
    delete[] array; // ok -> "array" now is a dangling pointer
    delete[] array; // double free or corruption!!
    // program aborted, the value of "array" is not null
}
```

## note:

```
int* array = newint[10];
delete[] array; // ok -> "array" now is a dangling pointer
array = nullptr; // no more dagling pointer
delete[] array; // ok, no side effect
```

# `void` Pointer - Generic Pointer

Instead of declaring different types of pointer variable it is possible to declare single pointer variable which can act as any pointer types

- `void*` can be compared
- Any pointer type can be implicitly converted to `void*`
- Other operations are unsafe because the compiler does not know what kind of object is really pointed to

```
cout << (sizeof(void*) == sizeof(int*)); // print true

int array[] = { 2, 3, 4 };
void* ptr   = array; // implicit conversion
cout << *array;      // print 2
// *ptr;             // compile error
// ptr + 2;          // compile error
```

# Reference 1/2

## Reference

A variable **reference** T& is an **alias**, namely another name for an already existing variable. Both variable and variable reference can be applied to refer the value of the variable

- A pointer has its own memory address and size on the stack, reference shares the **same memory address** (with the original variable)
- The compiler can internally implement references as *pointers*, but treats them in a very different way

# Reference 2/2

**References are safer than pointers**:

- References **cannot have NULL** value. You must always be able to assume that a reference is connected to a legitimate storage
- References **cannot be changed**. Once a reference is initialized to an object, it cannot be changed to refer to another object
  (Pointers can be pointed to another object at any time)
- References must be **initialized** when they are created
  (Pointers can be initialized at any time)

# Reference - Examples

**Reference syntax:** `T& var = ...`

```
//int& a;         // compile error no initilization
//int& b = 3; // compile error "3" is not a variable
int  c = 2;
int& d = c;      // reference. ok valid initialization
int& e = d;      // ok. the reference of a reference is a reference
d++;             // increment
e++;             // increment
cout << c;       // print 4
```

```
int  a = 3;
int* b = &a; // pointer
int* c = &a; // pointer
b++;         // change the value of the
*c++;        pointer 'b'
int& d = a;  // change the value of 'a' (a = 4)
d++;         // reference
             // change the value of 'a' (a = 5)
```

# Reference - Function Arguments 1/2

Reference vs. pointer arguments:

```
void f(int* value) {} // value may be a nullptr

void g(int& value) {} // value is never a nullptr

int a = 3;
f(&a);     // ok
f(0);      // dangerous but it works!! (but not with other numbers)
//f(a);    // compile error "a" is not a pointer

g(a);      // ok
//g(3);    // compile error "3" is not a reference of something
//g(&a);   // compile error "&a" is not a reference
```

# Reference - Function Arguments 2/2

References can be use to indicate fixed size arrays:

```
void f(int (&array)[3]) { // accepts only arrays of size 3
    cout << sizeof(array);
}
void g(int array[]) {
    cout << sizeof(array); // any surprise?
}

int  A[3], B[4];
int* C = A;
//----------------------------------------------------------
f(A);     // ok
// f(B); // compile error B has size 4
// f(C); // compile error C is a pointer
g(A);    // ok
g(B);    // ok
g(C);    // ok
```

# Reference - Arrays$^\star$

```cpp
int A[4];
int (&B)[4] = A;      // ok, reference to array
int C[10][3];
int (&D)[10][3] = C; // ok, reference to 2D array

auto c = new int[3][4];  // type is int (*)[4]
// read as "pointer to arrays of 4 int"
// int (&d)[3][4] = c;    // compile error
// int (*e)[3]      = c;    // compile error
int (*f)[4] = c;           // ok
```

```cpp
int array[4];
// &array is a pointer to an array of size 4
int size1 = (&array)[1] - array;
int size2 = *(&array + 1) - array;
cout << size1; // print 4
cout << size2; // print 4
```

# Read from and Write to a File

# C++ Files: **fstream**

- The fstream library allows us to work with files.
- To use the fstream library, include both the standard <iostream> AND the <fstream> header file:

```
#include <iostream>
#include <fstream>
```

| Class | Description |
|---|---|
| **ofstream** | Creates and writes to files |
| **ifstream** | Reads from files |
| **fstream** | A combination of ofstream and ifstream: creates, reads, and writes to files |

# C++ Files: **fstream**

- These classes are derived directly or indirectly from the classes **istream** and **ostream**.
- We have already used objects whose types were these classes:
  - ○ **cin** is an object of class **istream**
  - ○ **cout** is an object of class **ostream**.
- Therefore, we have already been using classes that are related to our file streams. We can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.

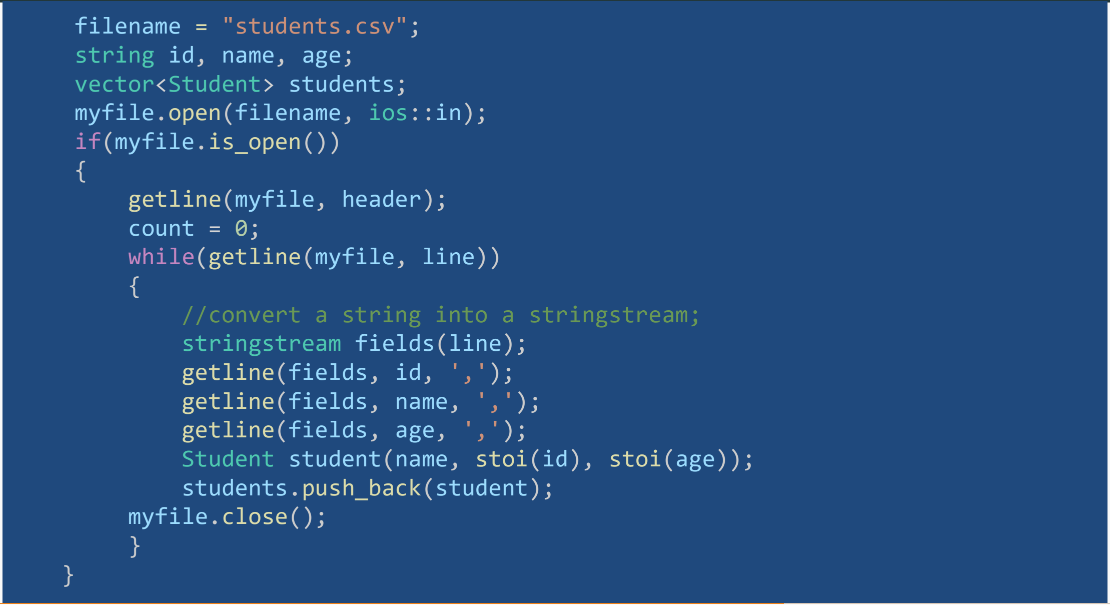
# Open a file to read

```
#include <fstream>
#include <string>
using namespace std;

int main() {
    string filename = "myfilein.txt";
    ifstream myfilein;
    myfilein.open(filename);
    …

}
```

# Read records from a file: getline()

```
int main() {
    string filename = “students.csv";
    string line;
    vector<string> mylines;
    ifstream myfile(filename);
    if (myfile.is_open()){
        while (getline(myfile,line))
        {
            mylines.push_back(line);
        }
        myfile.close();
        }
}
```

# Split one line into multiple fields by a delimter



```
     filename = "students.csv";
     string id, name, age;
     vector<Student> students;
     myfile.open(filename, ios::in);
     if(myfile.is_open())
     {
         getline(myfile, header);
         count = 0;
         while(getline(myfile, line))
         {
             //convert a string into a stringstream;
             stringstream fields(line);
             getline(fields, id, ',');
             getline(fields, name, ',');
             getline(fields, age, ',');
             Student student(name, stoi(id), stoi(age));
             students.push_back(student);
         myfile.close();
         }
    }
```

# Open a file to write

```
#include <fstream>
#include <string>
using namespace std;

int main() {
    string filename = "myfileout.txt";
    ofstream myfileout;
    myfileout.open(filename);
    …

}
```

# Write to ofsteam: <<

```
#include <fstream>
#include <string>
using namespace std;
int main() {
    //Write to a file
    filename = "students.out";
    ofstream myout(filename);
    if (myout.is_open()) {
        myout << "#id, name, age" << endl;
        for(auto st: students){
            myout << st.getStudentInfo() << endl;
        }
        myfile.close();
    }
}
```

# Heap and Stack

## Parenthesis and Brackets

{ } **braces**, informally “curly brackets”

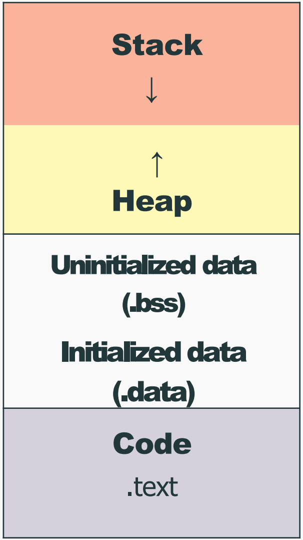
[ ] **brackets**, informally “square brackets”

( ) **parenthesis**, informally “round brackets”

< > **angle brackets**

# C++ Memmory Allocation

higher memory
addresses
0x00FFFFFF



Stack
↓

↑
Heap

Uninitialized data
(.bss)

Initialized data
(.data)

Code
.text

lower memory
addresses
0x00FF0000

**stack memory**

`int data[10]`

**dynamic memory**

`new int[10]`
`malloc(40)`

**Static/Global data**

`int data[10]`
(global scope)

bss = block started by symbol

# Data and BSS Segment

```
int data[]             = {1, 2}; // DATA segment memory
int big_data[1000000] = {};      // BSS segment memory
// (zero-initialized)

int main() {
    int A[] = {1, 2, 3}; // stack memory
}
```

Data/BSS (Block Started by Symbol) segments are larger than stack memory (max ≈ 1GB in general) but slower

# Stack and Heap Memory Overview

| | Stack | Heap |
|---|---|---|
| **Memory Organization** | Contiguous (LIFO) | Contiguous within an allocation, Fragmented between allocations (relies on virtual memory) |
| **Max size** | Small (8MB on Linux, 1MB on Windows) | Whole system memory |
| **If exceed** | Program crash at function entry (hard to debug) | Exception or $\mathrm{nullptr}$ |
| **Allocation** | Compile-time | Run-time |
| **Locality** | High | Low |
| **Thread View** | Each thread has its own stack | Shared among threads |

# Stack Memory

A local variable is either in the stack memory or CPU registers

```
int x = 3; // not on the stack (data segment)

struct A {
    int k; // depends on where the instance of A is
};
int main() {
    int   y    = 3;          // on stack
    char  z[]  = "abc";      // on stack
    A     a;                 // on stack (also k)
    void* ptr  = malloc(4); // variable "ptr" is on the stack
}
```

**The organization of the stack memory enables much higher performance. On the other hand, this memory space is limited!!**

# Stack Memory Data

**Types of data stored in the stack:**

*Local variables* Variable in a local scope

*Function arguments* Data passed from caller to a function

*Return addresses* Data passed from a function to a caller

*Compiler temporaries* Compiler specific instructions

*Interrupt contexts*

# Heap Memory - new, delete Keywords

**new, delete**

new/new[] and delete/delete[] are C++ *keywords* that perform dynamic memory allocation/deallocation, and object construction/destruction at runtime

`malloc` and `free` are C functions and they only allocate and free *memory blocks* (expressed in bytes)

# Dynamic Memory Allocation

- **Allocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
int* value = new int;                    // C++
```

- **Allocate *N* elements**

```
int* array = (int*) malloc(N * sizeof(int)); // C
int* array = new int[N];                     // C++
```

- **Allocate *N* structures**

```
MyStruct* array = (int*) malloc(N * sizeof(MyStruct)); // C
MyStruct* array = new MyStruct[N];                     // C++
```

- **Allocate and zero-initialize *N* elements**

```
int* array = (int*) calloc(N, sizeof(int)); // C
int* array = new int[N]();                  // C++
```

# Dynamic Memory Deallocation

- **Deallocate a single element**

```
int* value = (int*) malloc(sizeof(int)); // C
free(value);

int* value = new int;                    // C++
delete value;
```

- **Deallocate *N* elements**

```
int* value = (int*) malloc(N * sizeof(int)); // C
free(value);

int* value = new int[N];                     // C++
delete[] value;
```

# Allocation/Deallocation Properties

**Fundamental rules**:

- Each object allocated with `malloc()` must be deallocated with `free()`
- Each object allocated with `new` must be deallocated with `delete`
- Each object allocated with `new[]` must be deallocated with `delete[]`
- `malloc()`, `new`, `new[]` never produce `NULL` pointer in the *success* case, except for zero-size allocations (implementation-defined)
- `free()`, `delete`, and `delete[]` applied to `NULL`/ `nullptr` pointers do not produce errors

Mixing `new`, `new[]`, `malloc` with something different from their counterparts leads to *undefined behavior*