CS101A (H) Week1

C++ Introduction

Contents

Four languages

C, C++, Java, Python

C: The Foundation of Manual Control

As a foundational and relatively low-level language, C places the onus of initialization squarely on the programmer. Local variables must be manually initialized; otherwise, their values are unknown.

```c
#include <stdio.h>
struct Point { int x; int y;};
int global_var; // Global variables, if not explicitly initialized, will be automatically
// initialized to 0
int main() {
    int local_var = 100; // Local variables must be explicitly initialized
    int uninitialized_local; // If not initialized, this variable will contain an
    // uncertain "junk value"
    // printf("Uninitialized local: %d\n", uninitialized_local); // warning, output a random value

    int numbers[] = {1, 2, 3, 4, 5};
    // Initialize the structure using the specified initializer of C99
    struct Point p1 = {.y = 20, .x = 10}; // It doesn't have to be in any order
    return 0;
}
```

C++: Building on C with Abstraction and Safety

C++ inherits C's initialization syntax but vastly expands upon it with features.

- A Multitude of Initialization Forms
  - **Aggregate initialization:** Similar to C for arrays and simple structs.
  - **Copy-initialization:** Using the `=` operator.
  - **Direct-initialization:** Using parentheses `()`.
  - **List-initialization (since C++11):** Using curly braces `{}`, which is often recommended for its ability to prevent certain types of narrowing conversions, making it a safer option.

C++: Building on C with Abstraction and Safety

- **Constructors:** A significant difference from C is the concept of a constructor. When an object of a class is created, a constructor is called to initialize its members. C++ supports multiple constructors (overloading), default constructors (called when no arguments are provided), copy constructors (for creating a new object from an existing one), and move constructors (for efficiently transferring resources).
- **Member Initializer Lists:** C++ constructors can use a member initializer list to initialize class members before the constructor's body is executed. This is the only way to initialize `const` and reference members.

```cpp
#include <iostream>
#include <string>
class Rectangle {
private:
    int width;
    int height;
    std::string name;
public:
    // The constructor uses members to initialize the list, which is the recommended way
    Rectangle(int w, int h, std::string n) : width(w), height(h), name(n) {
        std::cout << "Rectangle '" << name << "' constructed." << std::endl; }
};

int main() {
    Rectangle r1 = Rectangle(10, 5, "R1"); // Copy-initialization
    Rectangle r2(12, 6, "R2"); // Direct-initialization
    Rectangle r3{14, 7, "R3"}; // List-initialization
    int a = 1;      // copy-initialization
    int b(2);     // direct-initialization
    int c{3};     // list-initialization
    return 0;
}
```

Java: Safety and Simplicity through the JVM

Java prioritizes safety and simplicity, and its initialization rules reflect this by eliminating some of the "dangers" present in C and C++.

- **Default Values:** In Java, all instance variables (class members) are automatically initialized to a default value if not explicitly initialized by the programmer (0 for numeric types, `false` for booleans, and `null` for object references). This prevents the use of uninitialized variables at the class level.
- **Mandatory Local Variable Initialization:** Conversely, Java's compiler enforces a strict rule that all local variables must be explicitly initialized before they are used. This compile-time check eliminates a common source of errors.

Java

- **Constructors and the `new` Keyword:** Similar to C++, objects in Java are initialized via constructors. The `new` keyword is used to allocate memory for a new object and invoke its constructor. Java also supports constructor overloading.
- **No Multiple Initialization Forms:** Unlike C++, Java does not have different syntactic forms of initialization for objects. The `new` keyword is the standard way.

```java
public class Box {
    // Instance variables will be assigned default values if they are not initialized
    private int width;        //  0
    private String label;     // null
    // Constructor, used for initializing objects
    public Box(int width, String label) {
        this.width = width;
        this.label = label;
    }
    public Box() {}

    public static void main(String[] args) {
        Box box1 = new Box(10, "MyBox");
        Box defaultBox = new Box(); // Label: null, Height: 0
        // Local variables must be explicitly initialized before use
        // int localValue;
        // System.out.println(localValue); // This line will cause a compilation error!
        int localValue = 200; // A value must be assigned first
        System.out.println("Initialized local variable: " + localValue);
    }
}
```

Python: Dynamic and Flexible Initialization

Python's dynamic nature leads to a more flexible and conceptually simpler model of initialization.

- **Dynamic Typing:** In Python, variables are not declared with a type. A variable is created when a value is first assigned to it. The type of the variable is the type of the object it currently refers to.
- **The `__init__` Method:** In the context of classes, the `__init__` method serves as the constructor. It is automatically called when a new instance of a class is created. The first argument to `__init__` is conventionally named `self` and refers to the instance being created. Programmers use `__init__` to set the initial state (attributes) of an object.
- **No Formal "Declaration":** There is no separate declaration step for variables in Python. Initialization happens at the point of assignment.

```python
# Variables are created and initialized when assigned values
# There is no need to declare types
message = "Hello, Python!"
count = 100

class Car:
    def __init__(self, make, model, year):
        self.make = make
        ...
# Create a Car object
# This will automatically call the __init__ method of the Car class
my_car = Car("Tesla", "Model S", 2023)
```

Overview

What do **embedded systems**, **game development**, **high frequency trading**, and **particle accelerators** have in common? - C++, of course!

Effective C++ Item 1 (by Scott Meyers): **View C++ as a federation of languages.**

> The easiest way is to view C++ not as a single language but as a federation of related languages ... Fortunately, there are only four:
>
> - C.
> - Object-Oriented C++.
> - Template C++.
> - The STL.

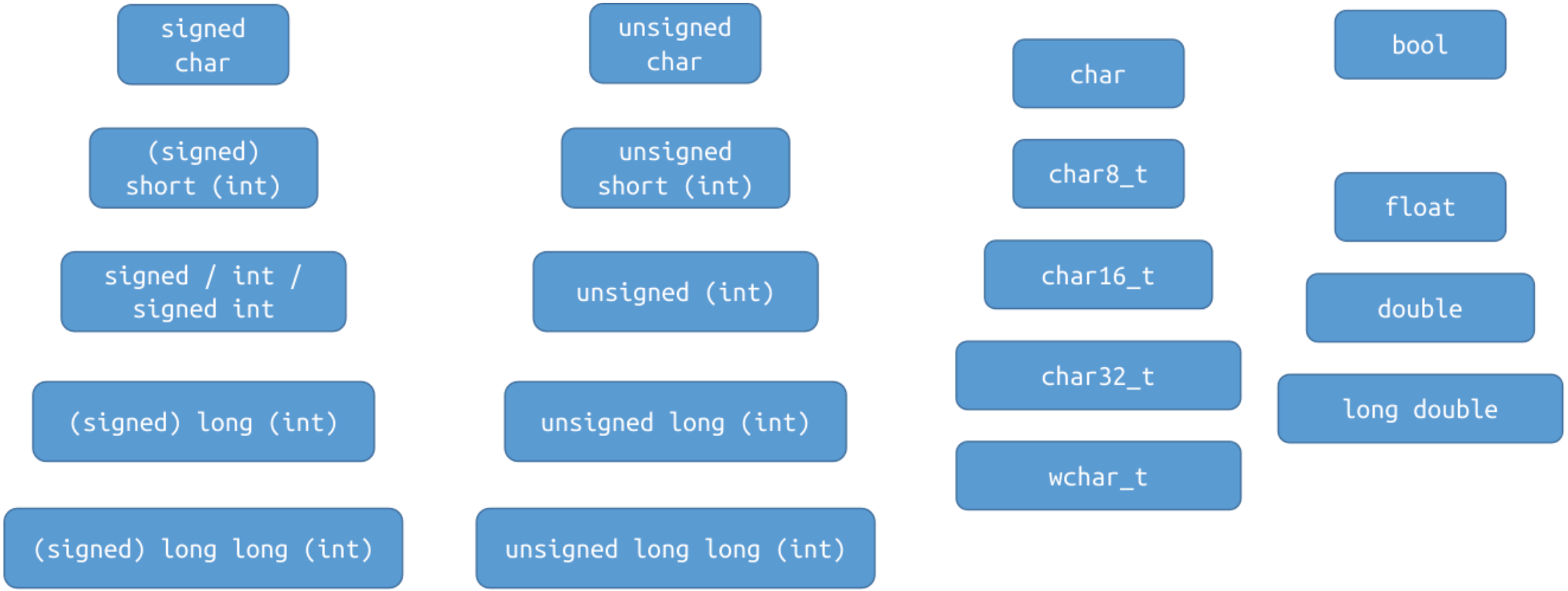
Standardization of C and C++

Standardization: **C++98**, **C++11**, C++14, C++17, **C++20**, C++23, C++26, ...

- C++98: The first ISO standard in 1998.
- C++11: Marks the beginning of **modern C++**.
- C++14/17: Some slight fixes and improvements of C++11.
- C++20: The first standard that delivers on virtually all the features that Bjarne Stroustrup dreamed of in *The Design and Evolution of C++* in 1994.
    - "D&E Complete"

CS101A is based on **C++17**.

Arithmetic types



Arithmetic types

- `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
- `sizeof(signed T) == sizeof(unsigned T)` for every `T` $\in \{$ `char`, `short`, `int`, `long`, `long long` $\}$
- Range of signed types: $\left[-2^{N-1}, 2^{N-1}-1\right]$. Range of unsigned types: $\left[0, 2^{N}-1\right]$
- Whether `char` is signed or not is **implementation-defined**. (An implementation-defined behavior depends on the compiler and the standard library, and is often also related to the hosted environment (e.g. the operating system).)
- Signed integer overflow is **undefined behavior**.
- Unsigned arithmetic **never overflows**: It is performed modulo 2^{N}, where N is the number of bits of that type.

Arithmetic types

type	width (at least)	width (usually)
`short`	16 bits	16 bits
`int`	16 bits	32 bits
`long`	32 bits	32 or 64 bits
`long long`	64 bits	64 bits

Arithmetic types

type	min	max
`short`	min=-32768	max=32767
`unsigned short`	min=0	max=65535

```
// the largest unsigned int
unsigned int u = std::numeric_limits<unsigned int>::max();
unsigned int u2 = u + 1; // u2 = 0
```

Pointer types

`PointeeType *`

- For `T` $\neq$ `U`, `T *` and `U *` are **different types**. (For example, `int *` and `double *` are different types.)
- **Null pointer**: The pointer holding the **null pointer value**, which is a special value indicating that the pointer is "pointing nowhere".
  - A null pointer can be obtained from `nullptr`.
  - Global or local static pointers will be zero-initialized to null.

```
int* p_null = nullptr;
```

Pointer types

- The value of a pointer of type `T *` is **the address of** an object of type `T`.
- `&var` returns the address of `var`. The return type is pointer to the type of `var`.
- Only when a pointer is actually pointing to an object is it **dereferenceable**, otherwise `*ptr` is **undefined behavior**.

```
int x = 42;
int* px = &x;
std::cout << "*px = " << *px << "\n"; // *px = 42
std::cout << "px = " << px << "\n"; // px = 0x6283bffb24
std::cout << "&x = " << &x << "\n"; // &x = 0x6283bffb24
// the return type of &x is int *
```

Pointer types

- `const T* p`
  - You cannot modify the object through p: `*p = value` is ill-formed.
  - The pointer value itself (where p points) can change: `p = other_ptr` is allowed.
- `T* const p`
  - The pointer itself is const : `p = other_ptr` is wrong.
  - The pointee is modifiable via p: `*p = value` is allowed.
- `const T* const p`
  - Neither can change: cannot reseat p, cannot modify the object through p.

Array types

`ElemType [N]`

- `T [N]`, `U [N]` and `T [M]` are **different types** for `T` $\neq$ `U` and `N` $\neq$ `M`. (For example, `int [5]` and `double [5]` are different types, `int [5]` and `int [10]` are different types.)
- Array sizes must be compile-time constants. Code like `int a[n]`, is ill-formed if `n` is not a constant expression. When the size is only known at run time, use dynamic containers such as `std::vector<Type>`, or other dynamic allocation mechanisms.
- In C (since C99), a VLA is an array whose size is determined at run time `int a[n]`.

Array types

- Valid index range: $[0, N)$. Subscript out of range is **undefined behavior**.

```
int a[5] = {10, 20, 30, 40, 50};
std::size_t N = sizeof(a) / sizeof(a[0]);
// valid subscript [0, N)
std::cout << "a[0] = " << a[0] << ", a[N-1] = " << a[N-1] << "\n";
// a[N] or a[-1] are undefined behavior
```

- Decay: `a` $\rightarrow$ `&a[0]`, `T [N]` $\rightarrow$ `T *`.

```
// The array name decays into a pointer to the first element
int* p = a;             // equal to &a[0], sizeof(p) = 8 bytes (pointer size)
std::cout << "*p = " << *p ;// *p = 10
// However, what is obtained by using the address symbol &a is a "pointer to
//the entire array", which is of a different type
int (*pa)[5] = &a;      // T (*)[N], sizeof(*pa) = 20 bytes (array of 5 ints)
std::cout << "(*pa)[2] = " << (*pa)[2] << "\n"; //(*pa)[2] = 30
```

Array types

- Decay: `a` → `&a[0]`, `T [N]` → `T *`.

```
void takes_pointer(int* p, std::size_t n) {
// 'p' points to the first element of the array argument (after decay)
    std::cout << "sizeof(p) = " << sizeof(p) << "\n";
}
int main() {
  int a[5] = {10, 20, 30, 40, 50};
  takes_pointer(a, 5); // T[N] decays when passing to a function, T[N] to T*
}
```

the output is: sizeof(p) = 8 bytes. (bytes of a pointer)

`struct` types

A special data type consisting of a sequence of **members**.

- The type name is `struct StructName`.
- $\texttt{sizeof(struct X)} \geqslant \sum_{\texttt{member} \in \texttt{X}} \texttt{sizeof(member)}$

  Alignment and padding make sizeof(Mixed) typically bigger than or equal to the sum of the members' sizeof values; the exact size depends on member order and alignment rules.

```
struct Mixed {
    char  c;     // 1 byte
    int   i;     // 4 bytes
    short s;     // 2 bytes
}; // There may be padding bytes to satisfy the maximum alignment requirement
std::cout << sizeof(Mixed) << "\n"; // sizeof(Mixed) = 12 (c=1, i=4, s=2)
std::cout << alignof(Mixed) << "\n"; // alignof(Mixed) = 4
```

Variables

Declare a variable: `Type varName`

- `ElemType varName[N]` for array type `ElemType[N]`.
- `T (*varName)[N]` for pointer to array type `T (*)[N]`.

```
int a[2][4];          // 2 rows, each row is int[4]
// Pointer to “array of 4 int”
int (*row)[4] = &a[0];   // row points to the first row (type: int (*)[4])
(*row)[1] = 42;             // assign a[0][1] = 42
row = row + 1;              // advance by one whole row (skips 4 ints)
(*row)[3] = 7;              // assign a[1][3] = 7
```

Variables

Initialize a variable: `= initializer`

- Designators for arrays: `= {[3] = 5, [7] = 4}`
- Designators for `struct` s: `= {.mem1 = x, .mem2 = y}`.
- Brace-enclosed list initializer for arrays and `struct` s: `= { ... }`.

```
int a[5] = {1, 2, 3};        // remaining elements zero-initialized: {1,2,3,0,0}
int b[10]{ [3] = 5, [7] = 4 }; // C++20

struct Point { int mem1, mem2; };
Point p = {10, 20};          // aggregate initialization
Point s{ .mem1 = 7, .mem2 = 9 };  // C++20
```

Initialization

If a variable is declared without explicit initializer:

- For global or local `static` variables, they are **empty-initialized**:
  - `0` for integer types,
  - `+0.0` for floating-point types,
  - null pointer value for pointer types.
- For local non-`static` variables, they are **uninitialized**, holding indeterminate values.

These rules apply recursively to the elements of arrays and the members of `struct`s.
Any use of the value of an uninitialized variable is **undefined behavior**.

Initialization

```
int g_int;              // zero-initialized -> 0
double g_dbl;           // +0.0
int* g_ptr;             // null pointer
int g_arr[3];           // {0,0,0}
struct S { int x; double y; };
S g_s;                  // {0, +0.0}

void func() {
    static int s_int;         // 0
    static double s_dbl;      // +0.0
    static int* s_ptr;        // nullptr
    static int s_arr[2];      // {0,0}
    static S s;               // {0, +0.0}

    int l_int;          // uninitialized
    double l_dbl;       // uninitialized
    int* l_ptr;         // uninitialized
    int l_arr[2];       // elements uninitialized
    S ls;               // members uninitialized
}
```

Expressions

Expressions = operators + operands.

- Operator precedence, associativity, and evaluation order of operands
  - `f() + g() * h()`, `f() - g() + h()`
- The only four operators whose operands have deterministic evaluation order:
  - `&&` and `||`: short-circuit evaluation
  - `?:`
  - `,` (not in a function call or in an initializer list)

Expressions

- If the evaluation order of `A` and `B` is unspecified, and if
  - both `A` and `B` contain a write to an object, or
  - one of them contains a write to an object, and the other one contains a read to that object

  then **the behavior is undefined**.

Arithmetic operators

`+`, `-`, `*`, `/`, `%`

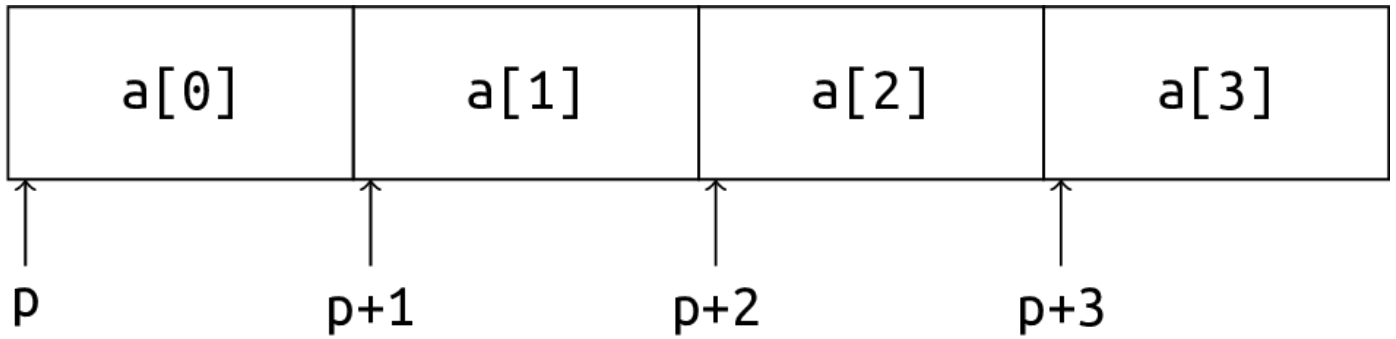
- Division: truncated towards zero.
- Remainder: `(a / b) * b + (a % b) == a` always holds.
- For `+`, `-`, `*` and `/`, the operands undergo a series of type conversions to a common type.

Bitwise operators: `~`, `&`, `|`, `^`, `<<`, `>>`

Compound assignment operators: `a op= b` is equivalent to `a = a op b`.

Be careful with signed overflows.

Pointer arithmetic



- Pointer arithmetic: `p++`, `++p`, `p--`, `--p`, `p + i`, `i + p`, `p - i`, `p += i`, `p -= i`, `p1 - p2`.
- Pointer arithmetic uses the units of the pointed-to type.
  - `p + i == (char *)p + i * sizeof(*p)`
- Pointer arithmetic must be performed within an array (including its past-the-end position), otherwise **the behavior is undefined**.

Operators

`++`, `--`

- `++a` and `--a` returns the value of `a` after incrementation/decrementation.
- `a++` and `a--` returns the original value of `a`.

`<`, `<=`, `>`, `>=`, `==`, `!=`

- The operands undergo a series of type conversions to a common type before comparison.

Operators

Member access: `obj.member`.

Member access through pointer: `ptr->member`, which is equivalent to `(*ptr).member`.

- `.` has higher precedence than `*`, so the parentheses around `*ptr` are necessary.

Control flow

- `if (cond) stmt`
- `if (cond) stmt1 else stmt2`
- `for (init_expr; cond; inc_expr) stmt`
- `while (cond) stmt`
- `do stmt while (cond);`
- `switch (integral_expr) { ... }`
- `break` and `continue`

Functions

Function declaration: `RetType funcName(Parameters);`

- Parameter names are not necessary, but types are required.
- A function can be declared multiple times.

Function definition: `RetType funcName(Parameters) { functionBody }`

- A function can be defined only once.

Functions

- Argument passing:
  - Use the argument to initialize the parameter.
  - The semantic is **copy**.
  - **Decay** always happens: One can never declare an array parameter.

Namespace `std`

`std::cin` and `std::cout`: names from the standard library.

C++ has a large standard library with a lot of names declared.

To avoid **name collisions**, all the names from the standard library are placed in a **namespace** named `std`.

- Example of name collisions in C: Suppose we want to write our own quick-sort:

```
#include <stdlib.h>
void qsort(int *a, int n) { // Ooops! stdlib already has one named `qsort`.
  // ...
}
```

Namespace `std`

`std::cin` and `std::cout`: names from the standard library.

C++ has a large standard library with a lot of names declared.

To avoid **name collisions**, all the names from the standard library are placed in a **namespace** named `std`.

- You can write `using std::cin;` to introduce `std::cin` into **the current scope**, so that `cin` can be used without `std::`.
- You may write `using namespace std;` to introduce **all the names in `std`** into the current scope, but **you will be at the risk of name collisions again.**

`std::string`

- Automatic memory management.
- `s.size()` returns the length. `s.empty()` returns whether `s` is empty.
- Use `+` and `+=` for concatenation. Use `<`, `<=`, `>`, `>=`, `==`, `!=` for comparison. Use `=` for copying.
- Use `>>` and `<<` for IO, as well as `std::getline`.
- Use `s[i]` to access the elements.
- Use range-based `for` loops to traverse a string.
- Use `std::to_string` and `std::stoi`, `std::stol`, ... for numeric conversions.
- Full list of functions related to `std::string`: https://en.cppreference.com/w/cpp/string/basic_string

Pointers and Arrays

Pointers

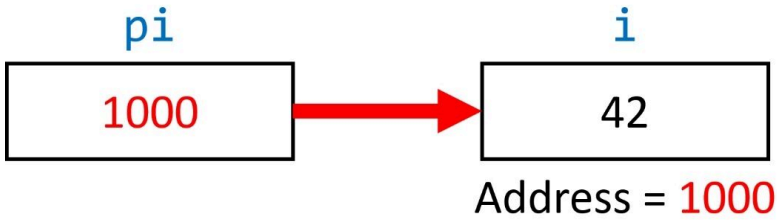
A pointer *points to* a variable. The **value** of a pointer is the address of the variable that it points to.

```
int i = 42;
int* pi = &i;
printf("%d\n", *pi);
```

- `int* pi;` declares a pointer named `pi`.
  - The type of `pi` is `int*`.
  - The type of the variable that `pi` points to ("pointee") is `int`.
- `&` is the **address-of operator**, used for taking the address of a variable.
- `*` in the expression `*pi` is the **indirection (dereference) operator**, used for obtaining the variable that a pointer points to.

Pointers

A pointer *points to* a variable.



We can access and modify a variable through its address (or a pointer pointing to it).

```
int num = 3;
int* ptr = &num;
printf("%d\n", *ptr);  // 3
*ptr = 10;
printf("%d\n", num);   // 10
++num;
printf("%d\n", *ptr);  // 11
```

Declare a pointer

To declare a pointer: `PointeeType* ptr;`

- The type of `ptr` is `PointeeType*`.
  - Pointer types with different pointee types are **different types**: `int*` and `double*` are different.
- The asterisk `*` can be placed near either `PointeeType` or `ptr`:
  - `PointeeType* ptr;` and `PointeeType *ptr;` are the same declaration.
  - `PointeeType * ptr;`, `PointeeType  *  ptr;` and `PointeeType*ptr;` are also correct.

Declare a pointer

The asterisk `*` can be placed near either `PointeeType` or `ptr`:

- `PointeeType* ptr;` may be more intuitive?

However, when declaring more than one pointers in one declaration statement, an asterisk is needed **for every identifier**:

```
int* p1, p2, p3;   // `p1` is of type `int*`, but `p2` and `p3` are ints.
int *q1, *q2, *q3; // `q1`, `q2` and `q3` all have the type `int*`.
int* r1, r2, *r3;  // `r1` and `r3` are of the type `int*`,
                   // while `r2` is an int.
```

[Best practice] Either `PointeeType *ptr` or `PointeeType* ptr` is ok. Choose one style and stick to it. But if you choose the second one, never declare more than one pointers in one declaration statement.

`&` and `*`

`&var` returns the address of the variable `var`.

- The result type is `Type *`, where `Type` is the type of `var`.
- `var` must be an object that has an identity (an *lvalue*): `&42` or `&(a + b)` are not allowed.

`*expr` returns **the variable** whose address is the value of `expr`.

- `expr` must have a pointer type `PointeeType *`. The result type is `PointeeType`.
- **The variable** is returned, not only its value. This means that we can modify the returned variable: `++*ptr` is allowed.

`*`

In a **declaration** `PointeeType *ptr`, `*` is a part of the pointer type `PointeeType *`.

In an **expression** like `*ptr`, `*` is the **indirection (dereference) operator** used to obtain the variable whose address is the value of `ptr`.

Do not mix them up!

The null pointer

The **null pointer value** is the "zero" value for pointer types.

A null pointer compares unequal to any pointer pointing to an object.

It is used for representing a pointer that "points nowhere".

Dereferencing a null pointer is undefined behavior, and often causes severe runtime errors!

- Because it is not pointing to an object.

```
int *ptr = nullptr; // `ptr` is a null pointer.
printf("%d\n", *ptr); // undefined behavior
*ptr = 42; // undefined behavior
```

Implicit initialization of pointers

If a pointer is not explicitly initialized:

- Global or local `static`: Initialized to the null pointer value.
- Local non-`static`: Initialized to indeterminate values, or in other words, **uninitialized**.
    - Uninitialized pointers are often called **wild pointers**.

A wild pointer do not point to a specific object, and is not a null pointer either.

Dereferencing a wild pointer is undefined behavior, and often causes severe runtime errors.

[Best practice] Avoid wild pointers.

Pointers that are not dereferenceable

Dereferencing such a pointer is undefined behavior, and usually causes severe runtime errors.

```
if (ptr != nullptr && *ptr == 42) { /* ... */ }
```

When `ptr` is a null pointer, the right-hand side operand `*ptr == 42` won't be evaluated, so `ptr` is not dereferenced.

Arrays

An array is a sequence of `N` objects of an *element type* `ElemType` stored **contiguously** in memory, where `N` $\in \mathbb{Z}_+$ is the *length* of it.

```
ElemType arr[N];
```

`N` must be a **constant expression** whose value is known at compile-time.

```
int a1[10];       // OK. A literal is a constant expression.
#define MAXN 10
int a2[MAXN];     // OK. `MAXN` is replaced with `10` by the preprocessor.
int n; scanf("%d", &n);
int a[n];         // A C99 VLA (Variable-Length Array), whose length is
                  // determined at runtime.
```

Array type

An array is a sequence of `N` objects of an *element type* `ElemType` stored **contiguously** in memory, where `N` $\in \mathbb{Z}_+$ is the *length* of it.

```
ElemType arr[N]; // The type of `arr` is `ElemType [N]`.
```

The type of an array consists of two parts:

1. the element type `ElemType`, and
2. the length of the array `[N]`.

Array initialization

If an array is declared without explicit initialization:

- Global or local `static`: Empty-initialization $\Rightarrow$ Every element is empty-initialized.
- Local non-`static`: Every element is initialized to indeterminate values (uninitialized).

Arrays can be initialized from brace-enclosed lists:

- Initialize the beginning few elements:

```
int a[10] = {2, 3, 5, 7}; // Correct: Initializes a[0], a[1], a[2], a[3]
int b[2] = {2, 3, 5};     // Error: Too many initializers
int c[] = {2, 3, 5};      // Correct: 'c' has type int[3].
int d[100] = {};          // Correct.
```

Array initialization

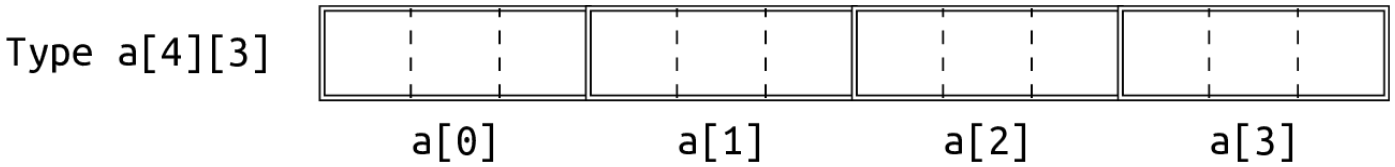
If an array is explicitly initialized, all the elements that are not explicitly initialized are **empty-initialized**.

```
int main(void) {
  int a[10] = {1, 2, 3}; // a[3], a[4], ... are all initialized to zero.
  int b[100] = {0};      // All elements of b are initialized to zero.
  int c[100] = {1};      // c[0] is initialized to 1,
                         // and the rest are initialized to zero.
}
```

`= {x}` is not initializing all elements to `x`!

Nested arrays

Nested arrays can be achieved as **arrays of arrays**:

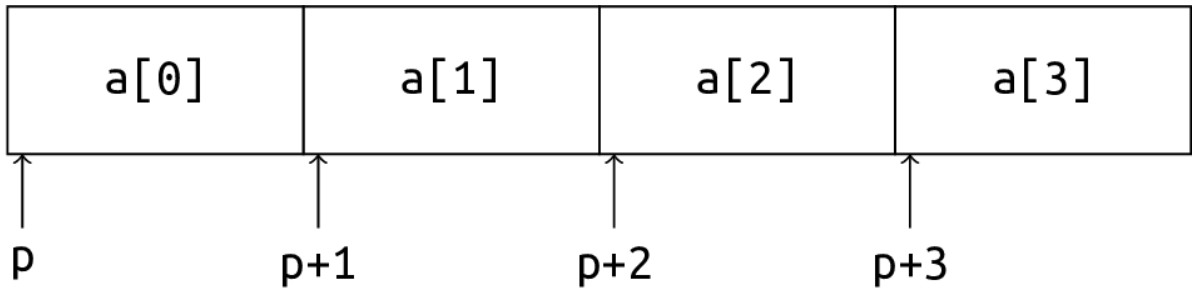



```
int a[4][3] = { // array of 4 arrays of 3 ints each (4x3 matrix)
    { 1 },      // row 0 initialized to {1, 0, 0}
    { 0, 1 },   // row 1 initialized to {0, 1, 0}
    { [2]=1 },  // row 2 initialized to {0, 0, 1}
};              // row 3 initialized to {0, 0, 0}
int b[4][3] = {             // row 0 initialized to {1, 3, 5}
  1, 3, 5, 2, 4, 6, 3, 5, 7 // row 1 initialized to {2, 4, 6}
};                          // row 2 initialized to {3, 5, 7}
                            // row 3 initialized to {0, 0, 0}
int y[4][3] = {[0][0]=1, [1][1]=1, [2][0]=1};  // row 0 initialized to {1, 0, 0}
                                               // row 1 initialized to {0, 1, 0}
                                               // row 2 initialized to {1, 0, 0}
                                               // row 3 initialized to {0, 0, 0}
```

Pointer arithmetic

Let `p` be a pointer of type `T *` and let `i` be an integer.

- `p + i` returns the address equal to the value of `(char *)p + i * sizeof(T)`. In other words, pointer arithmetic uses the unit of the pointed-to type.
- If we let `p = &a[0]` (where `a` is an array of type `T [N]`), then
  - `p + i` is equivalent to `&a[i]`, and
  - `*(p + i)` is equivalent to `a[i]`.

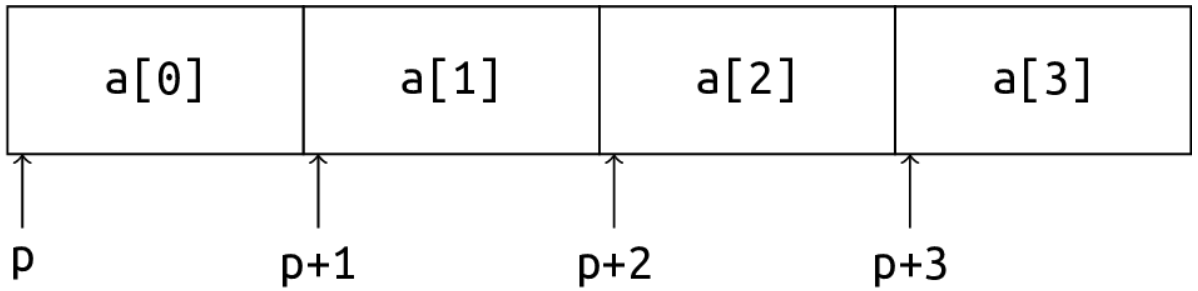


- Arithmetic operations `i + p`, `p += i`, `p - i`, `p -= i`, `++p`, `p++`, `--p`, `p--` are defined in the same way.

Array-to-pointer conversion

If we let `p = &a[0]` (where `a` is an array of type `T [N]`), then

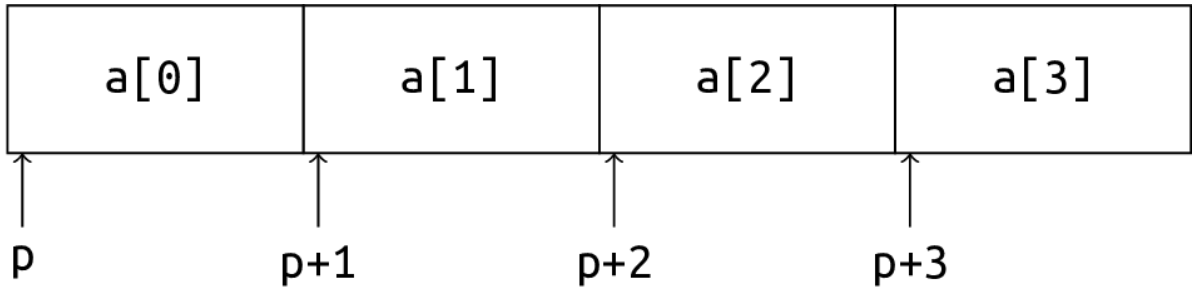
- `p + i` is equivalent to `&a[i]`, and
- `*(p + i)` is equivalent to `a[i]`.



Considering the close relationship between arrays and pointers, an array can be **implicitly converted** to a pointer to the first element: `a` $\longrightarrow$ `&a[0]`, `T [N]` $\longrightarrow$ `T *`.

- `p = &a[0]` can be written as `p = a` directly.
- `*a` is equivalent to `a[0]`.

Subtraction of pointers



Let `a` be an array of length `N`. If `p1 == a + i` and `p2 == a + j` (where `i` and `j` are nonnegative integers), the expression `p1 - p2`

- has the value equal to `i - j`, and
- has the type `ptrdiff_t`, which is a **signed** integer type declared in `<stddef.h>`.
  - The size of `ptrdiff_t` is implementation-defined. For example, it might be 64-bit on a 64-bit machine, and 32-bit on a 32-bit machine.
- Here `i`, `j` $\in [0, N]$ (closed interval), i.e. `p1` or `p2` may point to the "*past-the-end*" position of `a`.

Pointer arithmetic

Pointer arithmetic can only happen within the range of an array and its "past-the-end" position (indexed $[0, N]$). For other cases, **the behavior is undefined**.

Examples of undefined behaviors:

- `p1 - p2`, where `p1` and `p2` point to the positions of two different arrays.
- `p + 2 * N`, where `p` points to some element in an array of length `N`.
- `p - 1`, where `p` points to the first element `a[0]` of some array `a`.

Note that the evaluation of the innocent-looking expression `p - 1`, without dereferencing it, is still undefined behavior and may fail on some platforms.

Pass an array to a function

The only way of passing an array to a function is to **pass the address of its first element**.

The following declarations are equivalent:

```
void fun(int *a);
void fun(int a[]);
void fun(int a[10]);
void fun(int a[2]);
```

In all these declarations, the type of the parameter `a` is `int *`.

- How do you verify that?

Pass an array to a function

```
void fun(int a[100]);
```

The type of the parameter `a` is `int *`. How do you verify that?

```
void fun(int a[100]) {
  printf("%d\n", (int)sizeof(a));
}
```

Output: (On 64-bit Ubuntu 22.04, GCC 13)

```
8
```

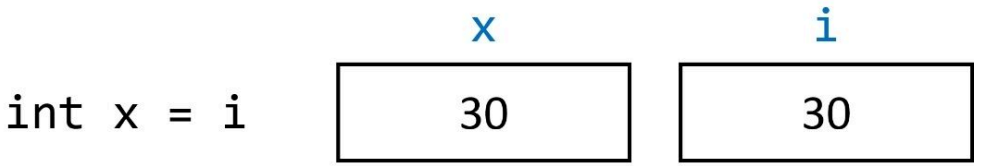
- If the type of `a` is `int[100]` as declared, the output should be `400` (assuming `int` is 32-bit).

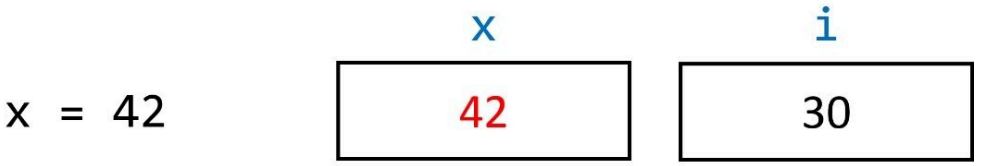
Argument passing

What is the output? Is the value of `i` changed to `42`?

```c
void fun(int x) {
  x = 42;
}
int main(void) {
  int i = 30;
  fun(i);
  printf("%d\n", i);
}
```

Argument passing



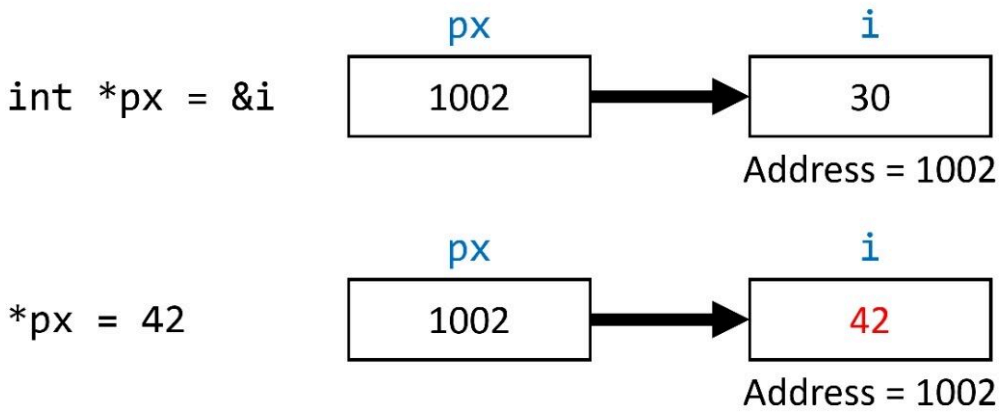


The output is still `30`. `i` is not changed.

- The parameter `x` is initialized as if `int x = i;`, thus obtaining the **value** of `i`.
  - `x` and `i` are two independent variables.
- Modification on `x` does not influence `i`.

Argument passing

To make the modification happen on a variable outside, pass its address into the function.




```c
void fun(int *px) {*px = 42;}
int main(void) {
  int i = 30;
  fun(&i); // int *px = &i
  printf("%d\n", i); // 42
}
```

Pass an array to a function

Even if you declare the parameter as an array (either `T a[N]` or `T a[]`), its type is still a pointer `T*`: **You are allowed to pass anything of type `T*` to it.**

- Array of element type `T` with any length is allowed to be passed to it.

```
void print(int a[10]) {
  for (int i = 0; i < 10; ++i)
    printf("%d\n", *(a + i));
}
int main(void) {
  int x[20] = {0}, y[10] = {0}, z[5] = {0}, w = 42;
  print(x);  // OK
  print(y);  // OK
  print(z);  // Allowed by the compiler, but undefined behavior!
  print(&w); // Still allowed by the compiler, also undefined behavior!
}
```

Pass an array to a function

Even if you declare the parameter as an array (either `T a[N]` or `T a[]`), its type is still a pointer `T*`: **You are allowed to pass anything of type `T*` to it.**

- Array of element type `T` with any length is allowed to be passed to it.

The length `n` of the array is often passed explicitly as another argument, so that the function can know how long the array is.

```
void print(int *a, int n) {
  for (int i = 0; i < n; ++i)
    printf("%d\n", *(a + i));
}
```

Subscript on pointers

```
void print(int *a, int n) {
  for (int i = 0; i < n; ++i)
    printf("%d\n", a[i]); // Look at this!
}
```

Subscript on pointers is also allowed! `a[i]` is equivalent to `*(a + i)`.

Return an array?

There is no way of returning an array from the function.

Returning the address of its first element is ok, **but be careful**:

This is OK:

```
int a[10];
int *foo(void) {  return a;}
```

This returns an **invalid address** —— it **returns the address of a local variable**.

```
int *foo(void) {
  int a[10] = {0};
  return a;
}
```

Pass a nested array to a function

When passing an array to a function, we make use of the **array-to-pointer conversion**:

- `Type [N]` will be implicitly converted to `Type *`.

A "2d-array" is an "array of array":

- `Type [N][M]` is an array of `N` elements, where each element is of type `Type [M]`.
- `Type [N][M]` should be implicitly converted to a "pointer to `Type[M]`".

What is a "pointer to `Type[M]`"?

Pointer to array

A pointer to an array of `N` `int`s:

```
int (*parr)[N];
```

An array of `N` pointers (pointing to `int`):

```
int *arrp[N];
```

Too confusing! How can I remember them?

Pointer to array

Too confusing! How can I remember them?

- `int (*parr)[N]` has a pair of parentheses around `*` and `parr`, so
  - `parr` is a pointer (`*`), and
  - points to something of type `int[N]`.
- Then the other one is different:
  - `arrp` is an array, and
  - stores `N` pointers, with pointee type `int`.

Pass a nested array to a function

The following declarations are equivalent: The parameter is of type `int (*)[N]`, which is a pointer to `int[N]`.

```
void fun(int (*a)[N]);
void fun(int a[][N]);
void fun(int a[2][N]);
void fun(int a[10][N]);
```

We can pass an array of type `int[K][N]` to `fun`, where `K` is arbitrary.

- The size for the second dimension must be `N`.
    - `T[10]` and `T[20]` are different types, so the pointer types `T(*)[10]` and `T(*)[20]` are not compatible.

Pass a nested array to a function

```c
void print(int (*a)[5], int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 5; ++j)
      printf("%d ", a[i][j]);
    printf("\n");
  }
}
int main(void) {
  int a[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
  int b[3][5] = {0};
  print(a, 2); // OK
  print(b, 3); // OK
}
```

Pass a nested array to a function

In each of the following declarations, what is the type of `a`? Does it accept an argument of type `int[N][M]`?

1. `void fun(int a[N][M])`
2. `void fun(int (*a)[M])`
3. `void fun(int (*a)[N])`
4. `void fun(int **a)`
5. `void fun(int *a[])`
6. `void fun(int *a[N])`
7. `void fun(int a[100][M])`
8. `void fun(int a[N][100])`

Pass a nested array to a function

In each of the following declarations, what is the type of `a`? Does it accept an argument of type `int[N][M]`?

1. `void fun(int a[N][M])`: A pointer to `int[M]`. Yes.
2. `void fun(int (*a)[M])`: Same as 1.
3. `void fun(int (*a)[N])`: A pointer to `int[N]`. **Yes iff `N == M`.**
4. `void fun(int **a)`: A pointer to `int *`. **No.**
5. `void fun(int *a[])`: Same as 4.
6. `void fun(int *a[N])`: Same as 4.
7. `void fun(int a[100][M])`: Same as 1.
8. `void fun(int a[N][100])`: A pointer to `int[100]`. Yes iff `M == 100`.

Notes

In fact, you can pass the address of an array:

```
void print_array_10(int (*pa)[10]) {
  for (int i = 0; i < 10; ++i)
    printf("%d\n", (*pa)[i]);
}
int main(void) {
  int a[10], b[100], c = 42;
  print_array_10(&a); // OK
  print_array_10(&b); // Error
  print_array_10(&c); // Error
}
```

In the function `print_array_10` above, the parameter type is `int (*)[10]`, a pointer to an array of 10 `int`s. The pointee type must be `int[10]`. Passing the address of anything else to it would not work.

References

Declare a reference

A **reference** defines an **alternative name** for an object ("refers to" that object).

Similar to pointers, the type of a reference is `ReferredType &`, which consists of two things:

- `ReferredType` is the type of the object that it refers to, and
- `&` is the symbol indicating that it is a reference.

Example:

```
int ival = 42;
int &ri = ival; // `ri` refers to `ival`.
                // In other words, `ri` is an alternative name for `ival`.
std::cout << ri << '\n'; // prints the value of `ival`, which is `42`.
++ri;           // Same effect as `++ival;`.
```

Declare a reference

```
int ival = 42;
int x = ival;                 // `x` is another variable.
++x;                          // This has nothing to do with `ival`.
std::cout << ival << '\n'; // 42
int &ri = ival;               // `ri` is a reference that refers to `ival`.
++ri;                         // This modification is performed on `ival`.
std::cout << ival << '\n'; // 43
```

Ordinarily, when we initialize a variable, the value of the initializer is **copied** into the object we are creating.

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

A reference is an alias

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

```
int ival = 42;
int &ri = ival;
++ri;             // Same as `++ival;`.
ri = 50;          // Same as `ival = 50;`.
int a = ri + 1; // Same as `int a = ival + 1;`.
```

After a reference has been defined, **all** operations on that reference are actually operations on the object to which the reference is bound.

```
ri = a;
```

What is the meaning of this?

A reference is an alias

```
int ival = 42;
int &ri = ival;
++ri;             // Same as `++ival;`.
ri = 50;          // Same as `ival = 50;`.
int a = ri + 1; // Same as `int a = ival + 1;`.
```

When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer.

After a reference has been defined, **all** operations on that reference are actually operations on the object to which the reference is bound.

```
ri = a;
```

- This is the same as `ival = a;`. **It is not rebinding `ri` to refer to `a`.**

A reference must be initialized

```
ri = a;
```

- This is the same as `ival = a;`. **It is not rebinding `ri` to refer to `a`.**

Once initialized, a reference remains bound to its initial object. **There is no way to rebind a reference to refer to a different object.**

Therefore, **references must be initialized.**

References must be bound to *existing objects* ("lvalues")

It is not allowed to bind a reference to temporary objects or literals:

```
int &r1 = 42;     // Error: binding a reference to a literal
int &r2 = 2 + 3; // Error: binding a reference to a temporary object
int a = 10, b = 15;
int &r3 = a + b; // Error: binding a reference to a temporary object
```

In fact, the references we learn today are "lvalue references", which must be bound to *lvalues*. We will talk about *value categories* in later lectures.

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Therefore, there are no "references to references".

```
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int & &rr = ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?

```
int &ri2 = ri; // Same as `int &ri2 = ival;`.
```

- `ri2` is a reference that is bound to `ival`.
- **Any use of a reference is actually using the object that it is bound to!**

References are not objects

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Pointers must also point to objects. Therefore, there are no "pointers to references".

```
int ival = 42;
int &ri = ival; // binding `ri` to `ival`.
int &*pr = ri; // Error! No such thing!
```

What is the meaning of this code? Does it compile?

```
int *pi = &ri; // Same as `int *pi = &ival;`.
```

Reference declaration

Similar to pointers, the ampersand `&` only applies to one identifier.

```
int ival = 42, &ri = ival, *pi = &ival;
// `ri` is a reference of type `int &`, which is bound to `ival`.
// `pi` is a pointer of type `int *`, which points to `ival`.
```

Placing the ampersand near the referred type does not make a difference:

```
int& x = ival, y = ival, z = ival;
// Only `x` is a reference. `y` and `z` are of type `int`.
```

`*` and `&`

Both symbols have many identities!

- In a **declaration** like `Type *x = expr`, `*` is **a part of the pointer type** `Type *`.
- In a **declaration** like `Type &r = expr`, `&` is **a part of the reference type** `Type &`.
- In an **expression** like `*opnd` where there is only one operand, `*` is the **dereference operator**.
- In an **expression** like `&opnd` where there is only one operand, `&` is the **address-of operator**.
- In an **expression** like `a * b` where there are two operands, `*` is the **multiplication operator**.
- In an **expression** like `a & b` where there are two operands, `&` is the **bitwise-and operator**.

Example: Use references in range-`for`

```
std::string str;
std::cin >> str;
int lower_cnt = 0;
for (char c : str)
  if (std::islower(c))
    ++lower_cnt;
std::cout << "There are " << lower_cnt << " lowercase letters in total.\n";
```

The range-`for` loop in the code above traverses the string, and declares and initializes the variable `c` in each iteration as if

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i]; // Look at this!
  if (std::islower(c))
    ++lower_cnt;}
```

Example: Use references in range-`for`

```
for (char c : str)
  // ...
```

The range-`for` loop in the code above traverses the string, and declares and initializes the variable `c` in each iteration as if

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i];
  // ...
}
```

Here `c` is a copy of `str[i]`. Therefore, modification on `c` does not affect the contents in `str`.

Example: Use references in range-`for`

What if we want to change all lowercase letters to their uppercase forms?

```
for (char c : str)
  c = std::toupper(c); // This has no effect.
```

We need to declare `c` as a reference.

```
for (char &c : str)
  c = std::toupper(c);
```

This is the same as

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char &c = str[i];
  c = std::toupper(c); // Same as `str[i] = std::toupper(str[i]);`.
}
```

Example: Pass by reference-to-`const`

Write a function that accepts a string and returns the number of lowercase letters in it:

```
int count_lowercase(std::string str) {
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

To call this function:

```
int result = count_lowercase(my_string);
```

Example: Pass by reference-to-`const`

```
int count_lowercase(std::string str) {
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```
std::string str = my_string;
```

The contents of the entire string `my_string` are copied!

Example: Pass by reference-to-`const`

```
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```
std::string str = my_string;
```

The contents of the entire string `my_string` are copied! This copy is unnecessary, because `count_lowercase` is a read-only operation on `str`.

How can we avoid this copy?

Example: Pass by reference-to-`const`

```
int count_lowercase(std::string &str) { // `str` is a reference.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```
int result = count_lowercase(my_string);
```

When passing `my_string` to `count_lowercase`, the parameter `str` is initialized as if

```
std::string &str = my_string;
```

Which is just a reference initialization. No copy is performed.

Example: Pass by reference-to-`const`

```
int count_lowercase(std::string &str) { // `str` is a reference.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

However, this has a problem:

```
std::string s1 = something(), s2 = some_other_thing();
int result = count_lowercase(s1 + s2); // Error: binding reference to
                                       // a temporary object.
```

`a + b` is a temporary object, which `str` cannot be bound to.

Example: Pass by reference-to-`const`

References must be bound to existing objects, not literals or temporaries.

There is an exception to this rule: References-to-`const` can be bound to anything.

```
const int &rci = 42; // OK.
const std::string &rcs = a + b; // OK.
```

`rcs` is bound to the temporary object returned by `a + b` as if

```
std::string tmp = a + b;
const std::string &rcs = tmp;
```

⇒ We will talk more about references-to-`const` in recitations.

Example: Pass by reference-to-`const`

The answer:

```
int count_lowercase(const std::string &str) { // `str` is a reference-to-`const`.
  int cnt = 0;
  for (char c : str)
    if (std::islower(c))
      ++cnt;
  return cnt;
}
```

```
std::string a = something(), b = some_other_thing();
int res1 = count_lowercase(a);       // OK.
int res2 = count_lowercase(a + b);   // OK.
int res3 = count_lowercase("hello"); // OK.
```

Benefits of passing by reference-to-`const`

Apart from the fact that it avoids copy, declaring the parameter as a reference-to-`const` also prevents some potential mistakes:

```
int some_kind_of_counting(const std::string &str, char value) {
  int cnt = 0;
  for (std::size_t i = 0; i != str.size(); ++i) {
    if (str[i] = value) // Ooops! It should be `==`.
      ++cnt;
    else {
      // do something ...
      // ...
    }
  }
  return cnt;
}
```

`str[i] = value` will trigger a compile-error, because `str` is a reference-to-`const`.

Benefits of passing by reference-to-`const`

1. Avoids copy.
2. Accepts temporaries and literals (*rvalues*).
3. The `const` qualification prevents accidental modifications to it.

[Best practice] Pass by reference-to-`const` if copy is not necessary and the parameter should not be modified.

References vs pointers

A reference

- is not itself an object. It is an alias of the object that it is bound to.
- cannot be rebound to another object after initialization.
- has no "default" or "zero" value. It must be bound to an object.

A pointer

- is an object that stores the address of the object it points to.
- can switch to point to another object at any time.
- can be set to a null pointer value `nullptr`.

Both a reference and a pointer can be used to refer to an object, but references are more convenient - no need to write the annoying `*` and `&`.

`std::vector`

Defined in the standard library file `<vector>`.

A "dynamic array".

Class template

`std::vector` is a **class template**.

Class templates are not themselves classes. Instead, they can be thought of as instructions to the compiler for *generating* classes.

- The process that the compiler uses to create classes from the templates is called **instantiation**.

For `std::vector`, what kind of class is generated depends on the type of elements we want to store, often called **value type**. We supply this information inside a pair of angle brackets following the template's name:

```
std::vector<int> v; // `v` is of type `std::vector<int>`
```

Create a `std::vector`

`std::vector` **is not a type itself**. It must be combined with some `<T>` to form a type.

```
std::vector v;                  // Error: missing template argument.
std::vector<int> vi;            // An empty vector of `int`s.
std::vector<std::string> vs; // An empty vector of strings.
std::vector<double> vd;         // An empty vector of `double`s.
std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s.
                                   // "2-d" vector.
```

What are the types of `vi`, `vs` and `vvi`?

Create a `std::vector`

`std::vector` **is not a type itself**. It must be combined with some `<T>` to form a type.

```
std::vector v;                // Error: missing template argument.
std::vector<int> vi;          // An empty vector of `int`s.
std::vector<std::string> vs; // An empty vector of strings.
std::vector<double> vd;       // An empty vector of `double`s.
std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s.
                                   // "2-d" vector.
```

What are the types of `vi`, `vs` and `vvi`?

- `std::vector<int>`, `std::vector<std::string>`, `std::vector<std::vector<int>>`.

Create a `std::vector`

There are several common ways of creating a `std::vector`:

```
std::vector<int> v{2, 3, 5, 7};      // A vector of `int`s,
                                     // whose elements are {2, 3, 5, 7}.
std::vector<int> v2 = {2, 3, 5, 7}; // Equivalent to ↑

std::vector<std::string> vs{"hello", "world"}; // A vector of strings,
                                               // whose elements are {"hello", "world"}.
std::vector<std::string> vs2 = {"hello", "world"}; // Equivalent to ↑

std::vector<int> v3(10);      // A vector of ten `int`s, all initialized to 0.
std::vector<int> v4(10, 42); // A vector of ten `int`s, all initialized to 42.
```

Note that all the elements in `v3` are initialized to `0`.

- We hate uninitialized values, so does the standard library.

Create a `std::vector`

Create a `std::vector` as a copy of another one: **No need to write a loop!**

```
std::vector<int> v{2, 3, 5, 7};
std::vector<int> v2 = v; // `v2`` is a copy of `v`
std::vector<int> v3(v);  // Equivalent
std::vector<int> v4{v};  // Equivalent
```

Copy assignment is also enabled:

```
std::vector<int> v1 = something(), v2 = something_else();
v1 = v2;
```

- Element-wise copy is performed automatically.
- Memory is allocated automatically. The memory used to store the old data of `v1` is deallocated automatically.

C++17 CTAD

"**C**lass **T**emplate **A**rgument **D**eduction": As long as enough information is supplied in the initializer, **the value type can be deduced automatically by the compiler**.

```
std::vector v1{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42);     // vector<int>, deduced from 42 (int)
std::vector v4(10);         // Error: cannot deduce template argument type
```

Size of a `std::vector`

`v.size()` and `v.empty()`: same as those on `std::string`.

```
std::vector v{2, 3, 5, 7};
std::cout << v.size() << '\n';
if (v.empty()) {
  // ...
}
```

`v.clear()`: Remove all the elements.

Append an element to the end of a `std::vector`

`v.push_back(x)`

```
int n;
std::cin >> n;
std::vector<int> v;
for (int i = 0; i != n; ++i) {
  int x;
  std::cin >> x;
  v.push_back(x);
}
std::cout << v.size() << '\n'; // n
```

Remove the last element of a `std::vector`

`v.pop_back()`

Exercise: Given `v` of type `std::vector<int>`, remove all the consecutive even numbers in the end.

```
while (!v.empty() && v.back() % 2 == 0)
  v.pop_back();
```

`v.back()`: returns the ***reference*** to the last element.

- How is it different from "returning the *value* of the last element"?

`v.back()` and `v.front()`

Return the references to the last and the first elements, respectively.

It is a **reference**, through which we can modify the corresponding element.

```
v.front() = 42;
++v.back();
```

For `v.back()`, `v.front()` and `v.pop_back()`, **the behavior is undefined** if `v` is empty. They do not perform any bounds checking.

Range-based `for` loops

A `std::vector` can also be traversed using a **range-based `for` loop**.

```
std::vector<int> vi = some_values();
for (int x : vi)
  std::cout << x << std::endl;
std::vector<std::string> vs = some_strings();
for (const std::string &s : vs) // use reference-to-const to avoid copy
  std::cout << s << std::endl;
```

Access through subscripts

`v[i]` returns the **reference** to the element indexed `i`.

- `i` $\in [0, N)$, where $N =$ `v.size()`.
- Subscript out of range is **undefined behavior**. `v[i]` performs no bounds checking.
  - In pursuit of efficiency, most operations on standard library containers do not perform bounds checking.
- A kind of "subscript" that has bounds checking: `v.at(i)`.
  - If `i` is out of range, *a* `std::out_of_range` *exception is thrown*.

Feel the style of STL

Basic and low-level operations are performed automatically:

- Default initialization of `std::string` and `std::vector` results in an empty string / container, not indeterminate values.
- Copy of `std::string` and `std::vector` is done automatically, which performs member-wise copy.
- Memory management is done automatically.

Interfaces are consistent:

- `std::string` also has member functions like `.push_back(x)`, `.pop_back()`, `.at(i)`, `.size()`, `.clear()`, etc. which do the same things as on `std::vector`.
- Both can be traversed by range-`for`.