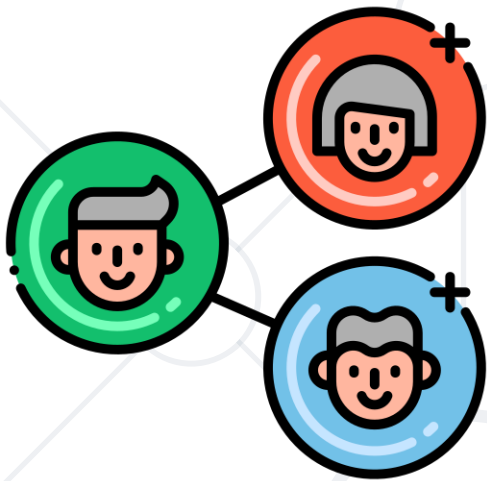


Computer Memory, Pointers and References

References, Computer Memory, Pointers, Pointer Arithmetic



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg/>



sli.do

#cpp-advanced

1. References
2. Computer Memory
3. Pointers
 - Referencing and Dereferencing
 - The NULL Pointer
4. Pointers and **const**
5. Pointer Arithmetics and Arrays





References

Creation, Usages, Limitations

- Identifiers assigned to the same memory as other identifiers

- **Type& name**

- Sometimes called "pseudonyms"

```
int original = 42;  
int& reference = original;  
original++; // original == 43; reference == 43  
reference++; // original == 44; reference == 44
```

- Assigned on declaration with a **variable** of the **same type**

```
int& reference; // compilation error  
int original = 42;  
double& reference = original; // compilation error
```

- Re-assigning caller variables

```
void swap(int& a, int& b)
{
    int oldA = a;
    a = b;
    b = oldA;
}
```

```
int main()
{
    int x = 13, y = 42;
    swap(x, y); // x == 42, y == 13
    return 0;
}
```

- Providing additional "return" values

```
int minValue(vector<int> numbers, int& foundAtIndex)
{
    foundAtIndex = 0;
    for (int i = 1; i < numbers.size(); i++)
    {
        if (numbers[foundAtIndex] > numbers[i])
        {
            foundAtIndex = i;
        }
    }
    return numbers[foundAtIndex];
    // the second parameter now contains the min index
}
```

Common Reference Usages

- Modifying caller's objects
 - Changing the object's fields (not re-assigning it)



```
void removeNegative(std::list<int>& numbers)
{
    auto i = numbers.begin();
    while (i != numbers.end())
    {
        if (*i < 0)
        {
            i = numbers.erase(i);
        }
        else i++;
    }
}
```


- **const** references can only be read, not written

- **const Type& name**

```
int original = 42;  
const int& reference = original;  
original++; // original == 43; reference == 43  
reference++; // compilation error
```

- Used to improve performance for object parameters:
 - Using a reference avoids copying the entire object
 - Using **const** prevents function from modifying the original

const Reference Parameters – Example

- Using reference prevents copying the vector

```
void printZeroIndices(const std::vector<int>& numbers)
{
    for (int i = 0; i < numbers.size(); i++)
    {
        if (numbers[i] == 0)
        {
            std::cout << i << " ";
        }
    }
}
```

const Reference Parameters – Example

- Marking it const prevents accidental editing

```
void printZeroIndices(const std::vector<int>& numbers)
{
    ...
    if (numbers[i] = 0)
    {
        // accidental "=" gives compilation error
    }
    ...
}
```

Reference Limitations

- If original variable goes out of scope, **reference is undefined**
- **Can't change** to reference other variable
- Initialized on creation – in class, must be set in **initializer list**





Computer Memory

Memory Structure, Variables in Memory

What Do We Call Memory?

- In computer science, memory usually is
 - A continuous, numbered (addressed) sequence of bytes
 - Storage for variables and functions created in programs
 - Random-access – equally fast accessing any byte
 - Addresses numbered in hexadecimal, prefixed with **0x**



Address	0x0	0x1	0x2	...	0x6afe4c	...
Byte	00001101	00101010	01000101	...	00000011	...

- A primitive data type takes up a sequence of bytes
 - **char** is 1 byte, 1 address – often used for reading byte by byte

```
char alpha = 'A'; // Let's assume alpha is at address 0x6afe4c
```

Address	...	0x6afe4b	0x6afe4c
Byte	01000001

- Other types and arrays use consecutive bytes

```
int year = 2018; // Let's assume year is at address 0x6afe4c
```

Address	...	0x6afe4b	0x6afe4c	0x6afe4d	0x6afe4e	0x6afe4f	...
Byte	11100010	00000111	00000000	00000000	...

- Prefix **operator&** returns a variable's address
 - Functions also have addresses – location in the memory

```
void f() {}

int main()
{
    int x = 42;
    auto addressX = &x;
    cout << x << " at " << addressX << endl;
    cout << "f()" << " code at " << &f << endl;
    return 0;
}
```


Array Address Values

- Array – a Type, a start address and a length

- Index **i** is at address: **start + i * sizeof(Type)**

We can store an address
in **size_t** position

```
int arr[] = { 2018, 310 }; // assume &arr[0] == 0x6afe4c
```

Address	...	0x6afe4b	0x6afe4c...0x6afe4f				0x6afe50...0x6afe53				0x6afe554
Byte	11100010	00000111	00000000	00000000	00110110	00000001	00000000	00000000	...
Value	2018				310				...

- array, it's address, and first element address are the same

```
cout << arr << " " << &arr << " " << &arr[0]; // 006AFE4C 006AFE4C 006AFE4C  
cout << &arr[1]; // 006AFE50
```



Pointers

Using and Representing Memory Addresses

Pointers

- A Memory-Address Type – store and can access a memory address

- **Type*** **name**

- **Type** – the type of value the pointer "points to"

```
char a = 'A';  
char* addressA = &a;
```

```
int x = 42;  
int* addressX = &x;
```

- A pointer to memory is what an index is to an array



- **Referencing** – setting what a pointer points to

```
int a = 42, b = 13; // let's assume &b == 0x69fef4
int* ptr = &a;      // points to a
ptr = &b;           // points to b
```

- **Dereferencing – operator*** – accesses memory, not a pointer

```
int a = 42; int* ptr = &a;
*ptr = 7      // a is now 7
cout << *ptr; // prints 7
```

- **operator->** – access member of pointed object

```
string s = "world"; string* ptr = &s;
ptr->insert(0, "hello "); // makes s == "hello world"
```

The NULL Pointer

- Special pointer value of:
 - `0`
 - `NULL`
 - `nullptr`
- Indicates a lack of value
- `nullptr` requires C++11 or greater, otherwise the code **won't compile**



```
int* findFirstNegativePtr(int numbers[], int length)
{
    for (int i = 0; i < length; i++)
    {
        if (numbers[i] < 0)
        {
            return &numbers[i];
        }
    }
    return nullptr;
}
```

NULL

"find" functions
returning **nullptr**
when no result
found




Pointers and const

Constant Pointers and Constant Data

Pointers and **const**

- Two things can change for a pointer:
 - Where it is pointing at
 - The data of the address



Pointer	Memory editable?	Address editable?
Type * ptr	YES	YES
const Type * ptr	NO	YES
Type * const ptr	YES	NO
const Type * const ptr	NO	NO

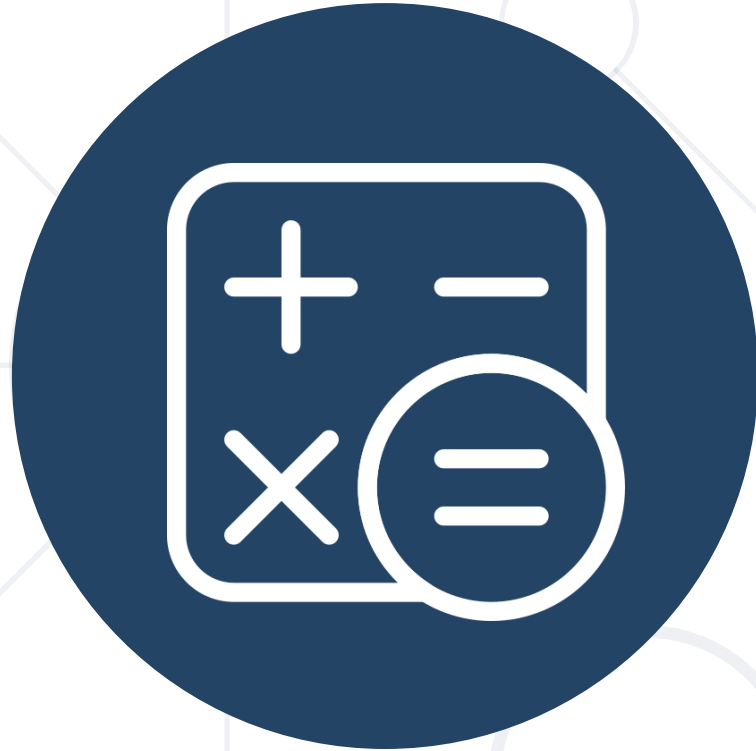
- Used similarly to **const** references
 - Pointer usage avoids object copy – only the address is copied
 - **const** on the **Type** prevents changing the pointed data

```
void printZeroIndices(const std::vector<int>* numbers) {  
    for (int i = 0; i < numbers->size(); i++) {  
        if (numbers->at(i) == 0) { std::cout << i << " "; }  
    }  
}  
  
int main() {  
    vector<int> numbers{ 1, 0, -2, 7, 0, 10, -100, 42 };  
    printZeroIndices(&numbers);  
    return 0;  
}
```

Example: Pointers to const Data

```
void printZeroIndices(const std::vector<int>* numbers)
{
    for (int i = 0; i < numbers->size(); i++)
    {
        if (numbers->at(i) == 0)
        {
            std::cout << i << " ";
        }
    }
}

int main()
{
    vector<int> numbers { 1, 0, -2, 7, 0, 10, -100, 42 };
    printZeroIndices(&numbers);
    return 0;
}
```



Pointer Arithmetic and Arrays

Type-Defined Pointer Calculations

- Pointer operations are based on their **Type**
 - Reading accesses exactly **sizeof(Type)** bytes
 - Writing sets exactly **sizeof(Type)** bytes

```
int year = 2018; // let's assume year is at address 0x6afe4c  
int* intPtr = &year;  
char* charPtr = (char*)&year;
```

Address	...	0x6afe4b	0x6afe4c	0x6afe4d	0x6afe4e	0x6afe4f	...
Byte	11100010	00000111	00000000	00000000	...

charPtr

intPtr

Pointer Arithmetic with Integers


- Typed pointers support integer addition/subtraction
- For a **Type* pointer** with address **x**
 - **pointer + value** calculates **x + sizeof(Type) * value**
 - **pointer - value** calculates **x - sizeof(Type) * value**

```
int number = 42; // assume &number == 0x6afe4c
int * intPtr = &number; char * charPtr = (char*)&number;

// NOTE: casting the char* to int* to avoid printing as a string
cout << intPtr << " " << (int*)charPtr << endl; // 0x6afe4c 0x6afe4c
intPtr++;
charPtr++;
cout << intPtr << " " << (int*)charPtr << endl; // 0x6afe50 0x6afe4d
```

Pointers as Arrays

- Array **operator[]** is actually defined with pointer arithmetic
- **arr[i]** compiles to ***(arr + i)**



```
int arr[3]{ 13, 42, 69 };  
int* p = arr;  
p[1] = -42;  
cout << arr[1];    // -42  
cout << *(p + 1);  // -42  
cout << p[1];      // -42
```

- Array parameters in functions "degenerate" into pointers

```
void f(int arr[], int length)  
void f(int* arr, int length)
```

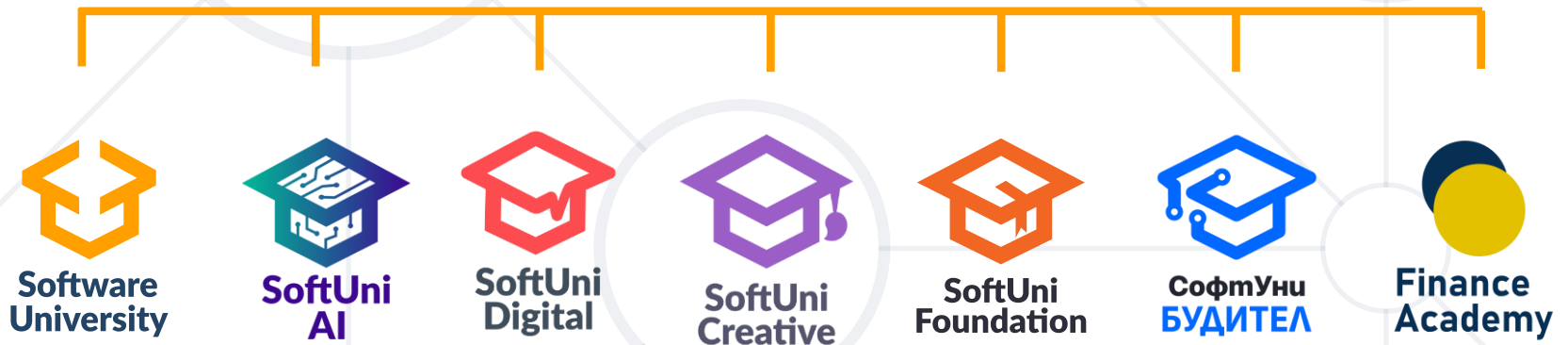
- **References** allow setting new identifiers for existing variables
- **Computer memory** is essentially an **array** of **bytes**
- Variables occupy consecutive bytes of memory
- **Pointers** are to memory what indices are to arrays
 - Used to read / write memory
 - Can change to point to other memory
- Pointer arithmetic allows pointers to work like arrays



Questions?



SoftUni



SoftUni Diamond Partners



**SUPER
HOSTING
.BG**



INDEAVR
Serving the high achievers



THE CROWN IS YOURS

VIVACOM

- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

- Software University Forums

- forum.softuni.bg



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

