

The background is a dense grid of small, semi-transparent circles in various colors including green, blue, purple, brown, orange, and red. Overlaid on this grid are faint, light-gray geometric shapes, including squares and circles, some of which contain thin white lines or patterns.

Algorithms

Pointers & Memory

Introduction

Pointers are one of the most powerful aspects of C++ programming, but also one of the most complex to master. **Pointers allow you to manipulate the computer's memory efficiently.**

Two concepts are fundamental to understanding how pointers work:

- The **size of all variables** and their **position in memory**.
- **All data is stored from a memory address**. This address can also be obtained and manipulated as another data.



All data has a memory address

All data is stored starting from a memory address and using as many bytes as necessary.

For example, consider the following data definition and its corresponding memory representation assuming starting at address **100**.

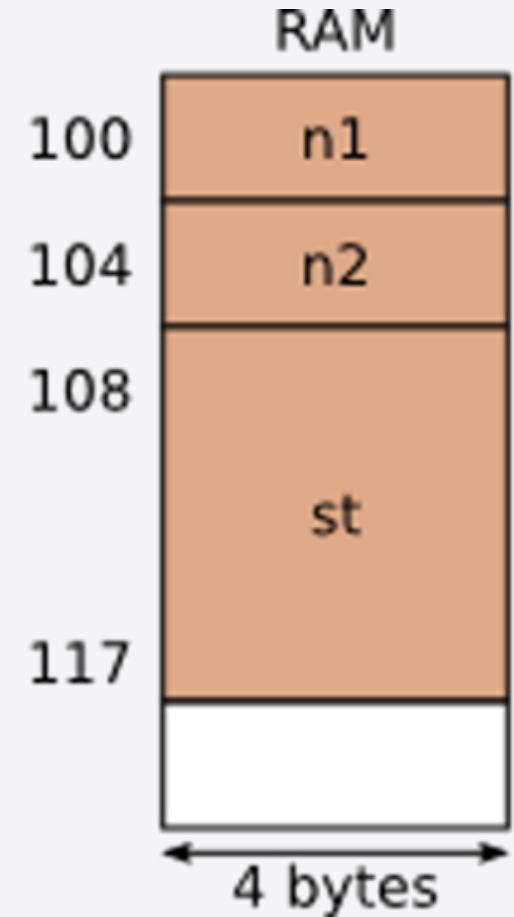
```
int n1;  
float n2;  
char st[10];
```

Assuming that

Char - 1 byte (8 bits)

Int - 4 bytes (32 bits)

Float - 4 bytes (32 bits)



*Have you ever wondered why there are **32-bit** and **64-bit** operating systems?*

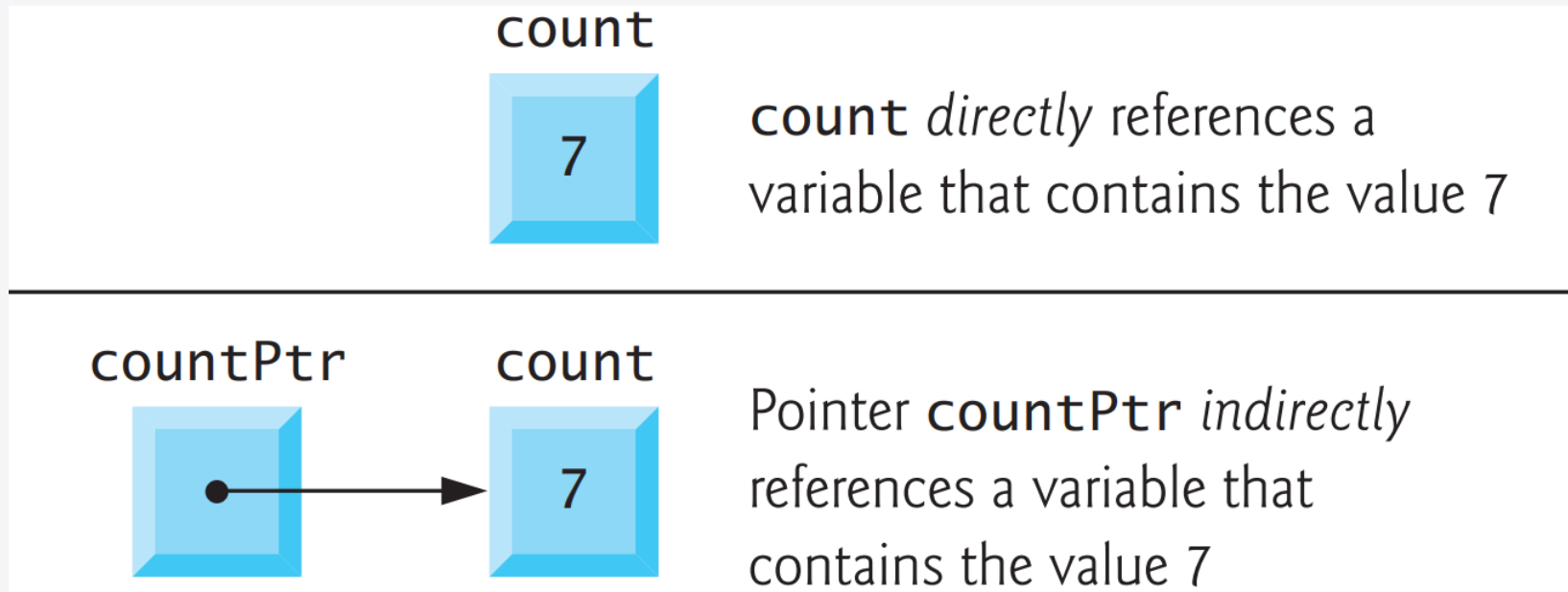


Pointer Variable Declaration & Initialization

Pointer variables contain **memory addresses** as their **values**.

Normally, a variable directly contains a **specific value**.

In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value**.





Declaring Pointers

Pointers, like any other variables, must be declared before they can be used. The standard way of declaring a pointer is

```
<type> * <identifier>;
```

For example, for the pointer **p**

```
int *p
```

declares the variable **p** to be of **type int*** (i.e., a pointer to an **int** value) and is read (right to left), "**p** is a pointer to **int**".

Each variable being declared as a pointer must be preceded by an **asterisk (*)**. **Pointers can be declared to point to objects of any data type.**



Another example is in the declaration

```
int *Ptr, p;
```

Here, the variable **Ptr** is declared to be a pointer to an **int** value.

Variable **p** in the preceding declaration is declared to be an **int**, but **not a pointer to an int**.

The ***** in the declaration applies only to the first variable.



Initializing Pointers

Pointers should be initialized to **nullptr** (added in C++11) / **NULL** (prior to C++11) or to a **memory address** either when they're declared or in an assignment.

A pointer with the value **nullptr** / **NULL** "**points to nothing**" and is known as a **null pointer**.

```
int * Ptr = NULL  
int * Ptr = nullptr
```

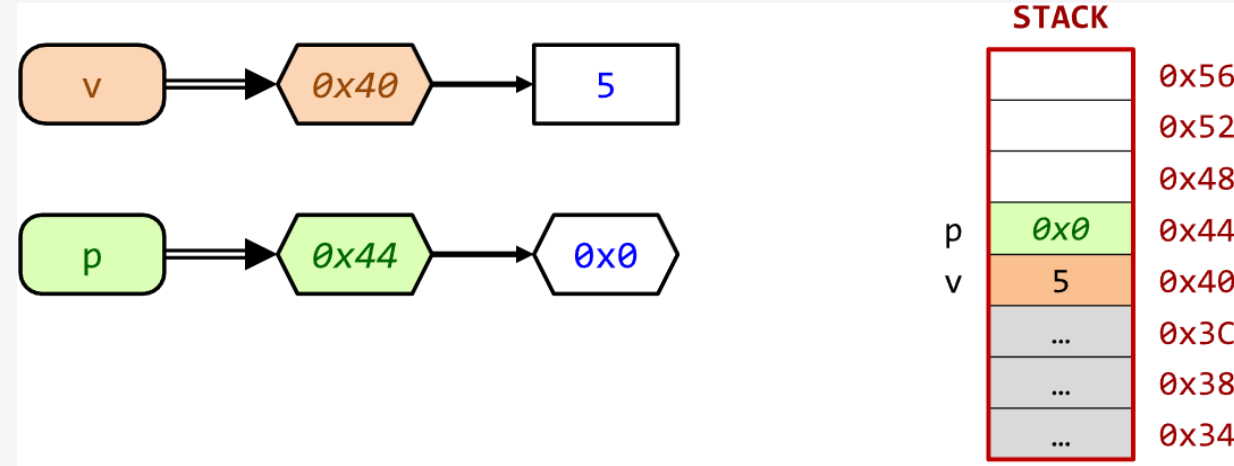
In earlier versions of C++, **NULL** is defined in several standard library headers to represent the value **0**.



Address (&) Operator

Given that Pointer only contain memory addresses, you can use the **address operator** (&) to obtain the **memory address** of its operand (variable / object). For example,

```
int v = 5; //value of v is 5
int* p = NULL; //value of p is NULL
```





Address (&) Operator

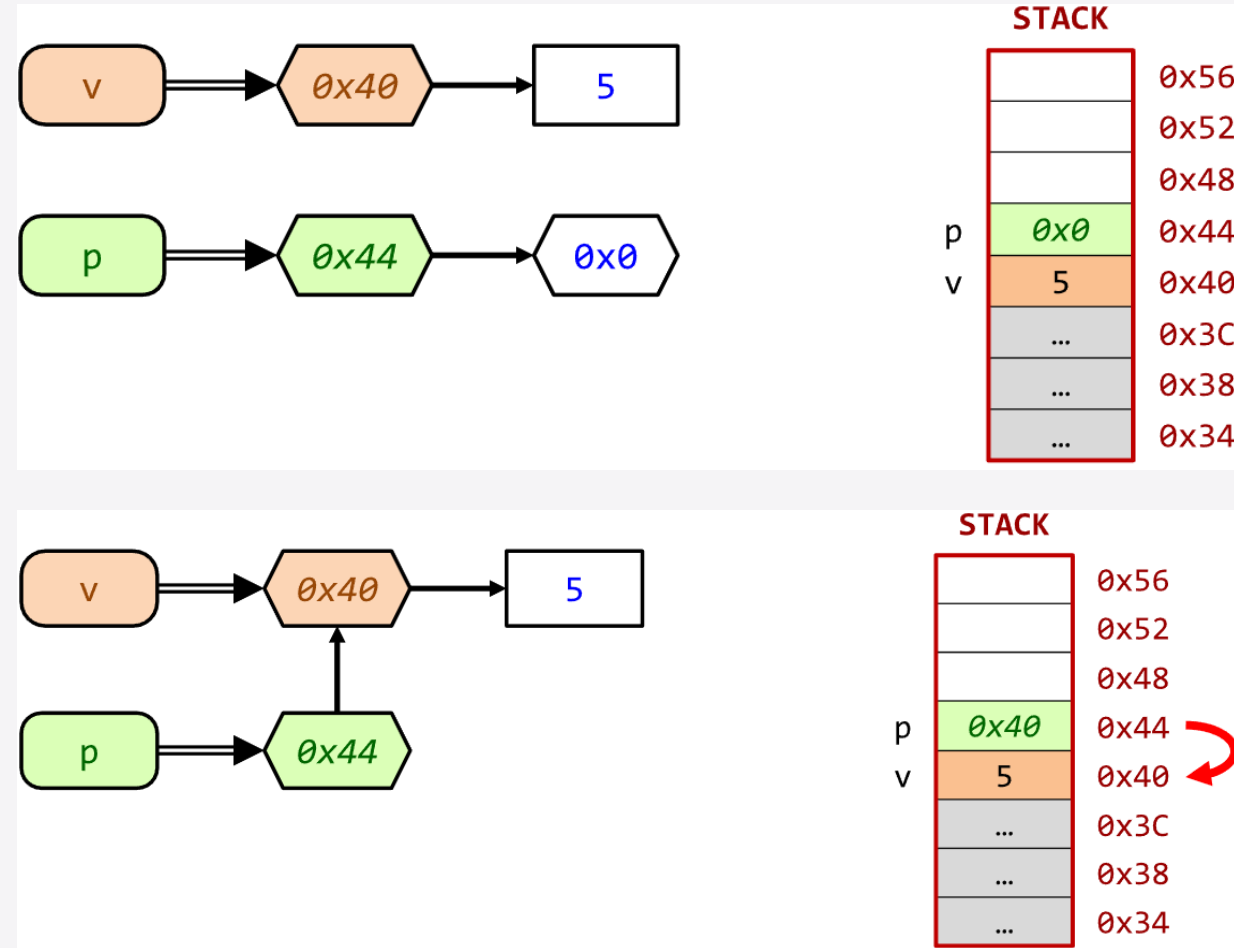
Given that Pointer only contain memory addresses, you can use the **address operator** (**&**) to *obtain the **memory address** of its operand* (variable / object). For example,

```
int v = 5; //value of v is 5
int* p = NULL; //value of p is NULL

p = &v; //take address of v
```

The address of the variable **v** (**position 40**) is assigned to the pointer **p**, stored at **position 44**.

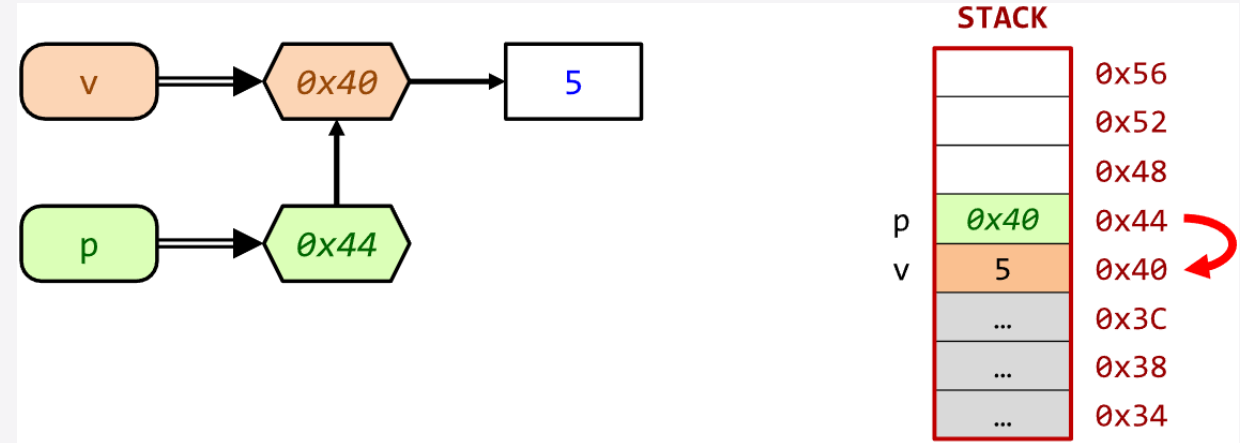
Then variable **p** is said to **"point to"** **v**. Now, **p** indirectly references variable **v**'s value (**5**).





```
int v = 5; //value of v is 5
int* p = NULL; //value of p is NULL
```

```
p = &v; //take address of v
```



The use of the **&** in the preceding statement is **not the same as its use in a *pass-by-reference declaration***, where it's always preceded by a **data-type name**. When declaring a reference, the **&** is part of the type.

In an expression like **&*v***, the **&** is the **address operator**.



The "pointer to" data type

Tipo T	Tamaño (bytes) [a]	Puntero a T	Tamaño (bytes)	Ejemplo de uso
int	4	int *	4	int *a, *b, *c;
unsigned int	4	unsigned int *	4	unsigned int *d, *e, *f;
short int	2	short int *	4	short int *g, *h, *i;
unsigned short int	2	unsigned short int *	4	unsigned short int *j, *k, *l;
long int	4	long int *	4	long int *m, *n, *o;
unsigned long int	4	unsigned long int *	4	unsigned long int *p, *q, *r;
char	1	char *	4	char *s, *t;
unsigned char	1	unsigned char *	4	unsigned char *u, *v;
float	4	float *	4	float *w, *x;
double	8	double *	4	double *y, *z;
long double	8	long double *	4	long double *a1, *a2;

**Reference values, may change depending on the platform.*

The **size of the pointers is always the same** regardless of the data they point to because they all store a memory address.



sizeof()

The **sizeof()** is an operator that evaluates the **size** of **data type, constants, variable**. It can also determine the size of **classes, structures**, and **unions**.

The size, which is calculated by the **sizeof()** operator, is the amount of RAM occupied in the computer (**bytes**).

```
int main()
{
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;

    return 0;
}
```

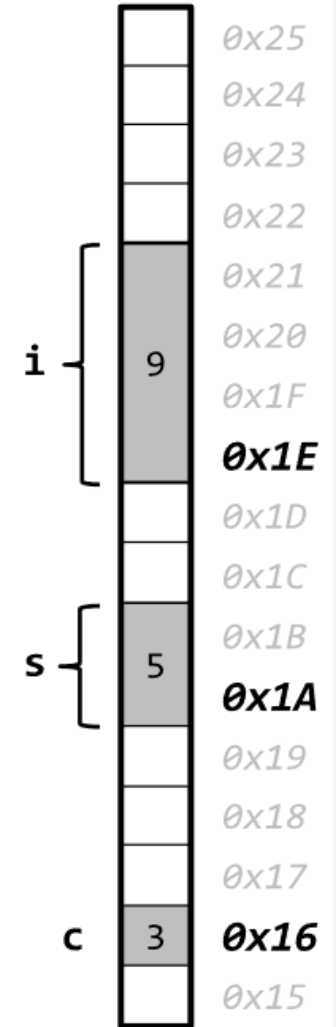
```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
```



Pointers Property (1/5)

Data types have different memory sizes

```
char  c = 3;  
short s = 5;  
int   i = 9;
```



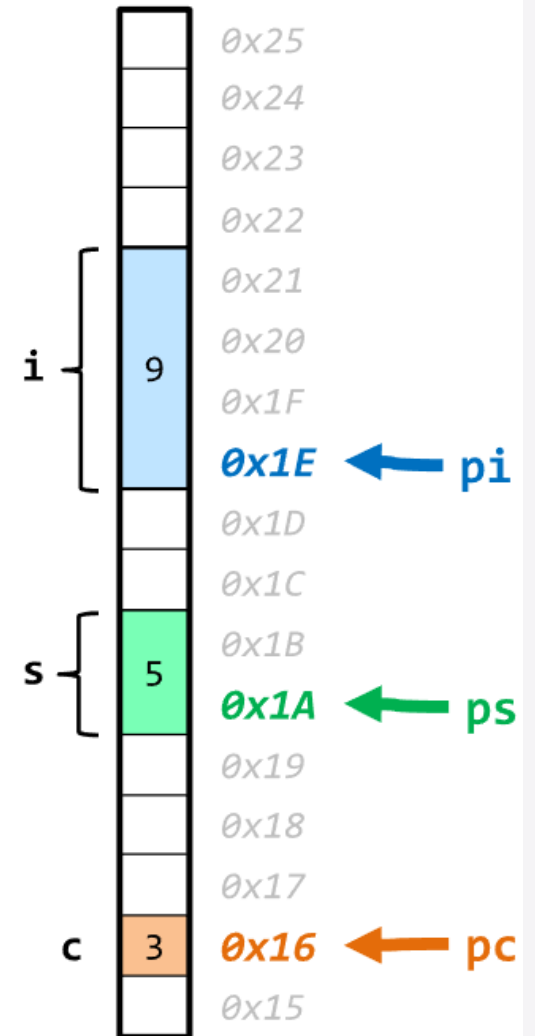


Pointers Property (2/5)

Pointers store **one** address only

```
char  c = 3;  
short s = 5;  
int   i = 9;
```

```
char* pc = &c;  
short* ps = &s;  
int* pi = &i;
```





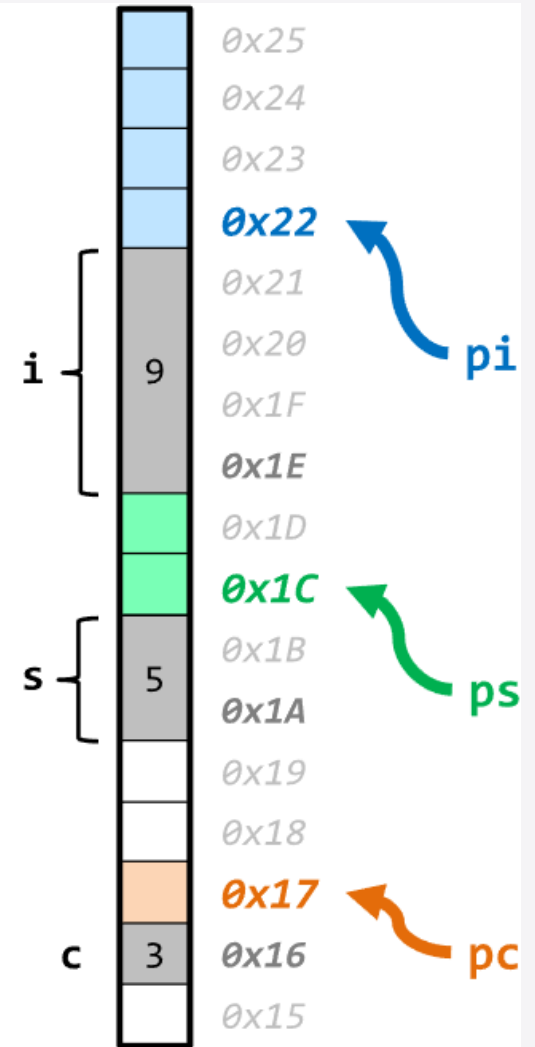
Pointers Arithmetic (3/5)

Increment by **1**.

```
char  c = 3;  
short s = 5;  
int   i = 9;
```

```
char* pc = &c;  
short* ps = &s;  
int* pi = &i;
```

```
pc++;  
ps++;  
pi++;
```





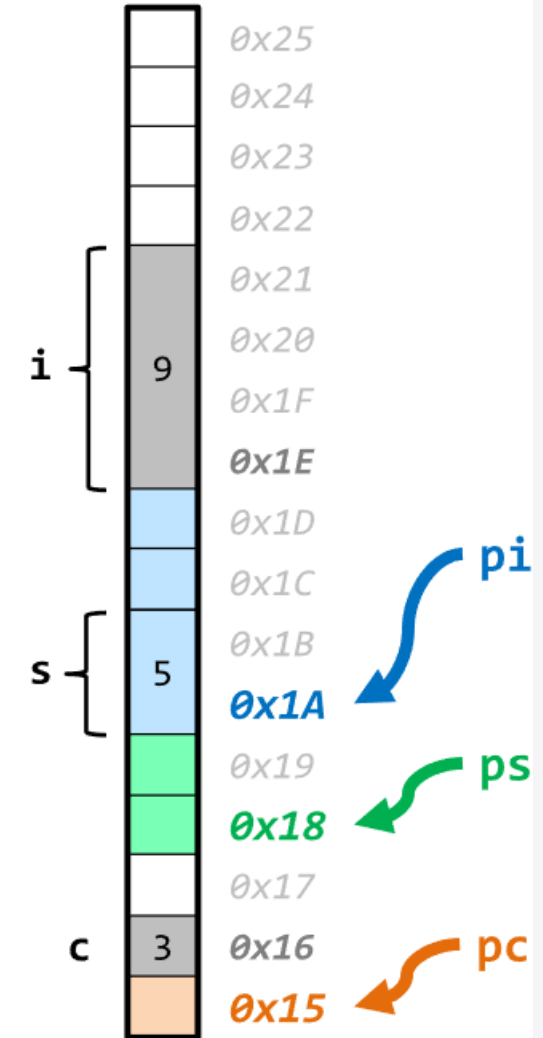
Pointers Arithmetic (4/5)

Decrement by **1**.

```
char  c = 3;  
short s = 5;  
int   i = 9;
```

```
char* pc = &c;  
short* ps = &s;  
int* pi = &i;
```

```
pc--;  
ps--;  
pi--;
```



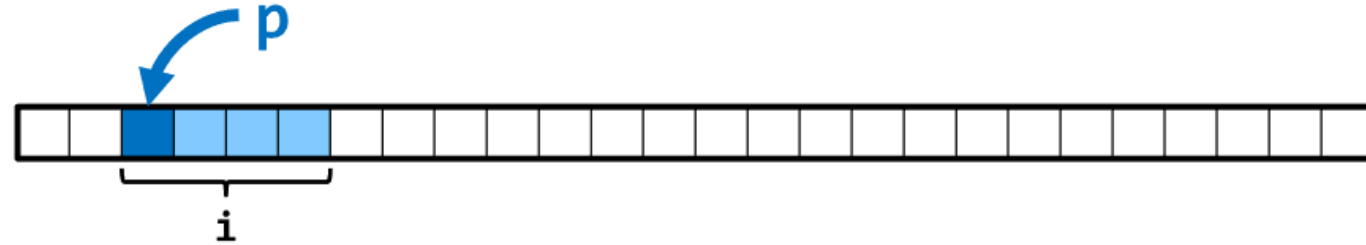


Pointers Arithmetic (5/5)

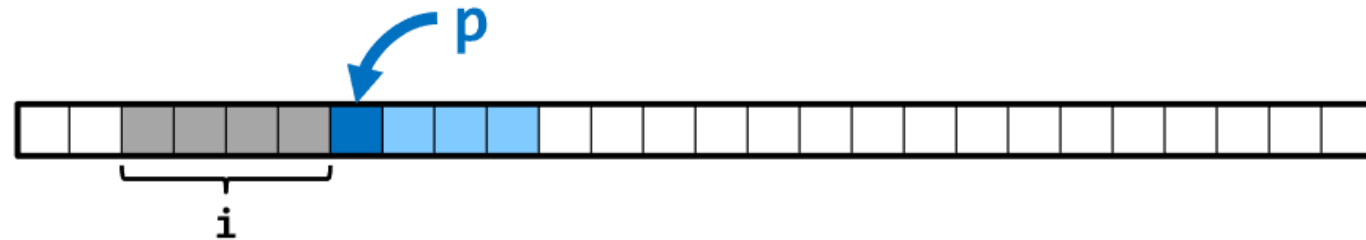
Here:

sizeof(int) = 4 bytes

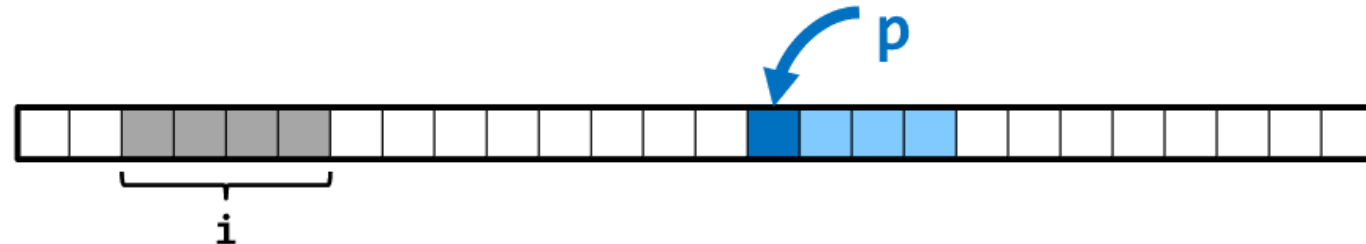
```
int i = 5;  
int* p = &i;
```



```
p = &i + 1;
```



```
p += 2;
```





Indirection (*) Operator

The unary ***** operator, referred to as the **indirection** / **dereferencing operator**, returns a **lvalue** representing **the value/object contained in the address pointed to by a pointer**. For example,

```
int v = 5; //value of v is 5
int *p = &v; //take address of v

cout << *p << endl;
```

displays the value **contained in** variable **v address**, namely, **5**, just as print **v** directly.

```
cout << v << endl;
```




Using ***** in this manner is called **dereferencing a pointer**. A **dereferenced pointer** may also be used as a **lvalue** on the left side of an **assignment statement**, as in

```
*p = 9;
```

which would assign the value **9** to **v** in an indirect form through the pointer, where ***p** is an alias for **v**.

The **dereferenced pointer** may also be used to receive an input value as in

```
cin >> *p;
```

which places the input value in **v**.

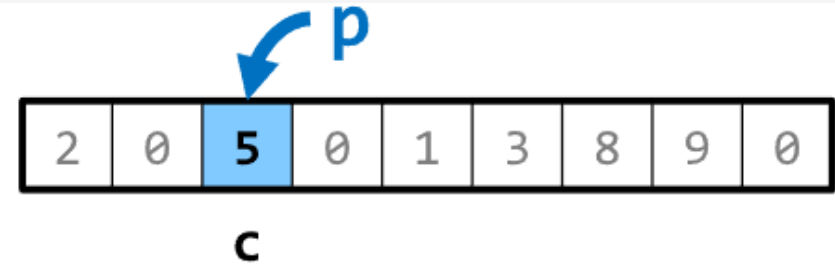


Pointers Arithmetic

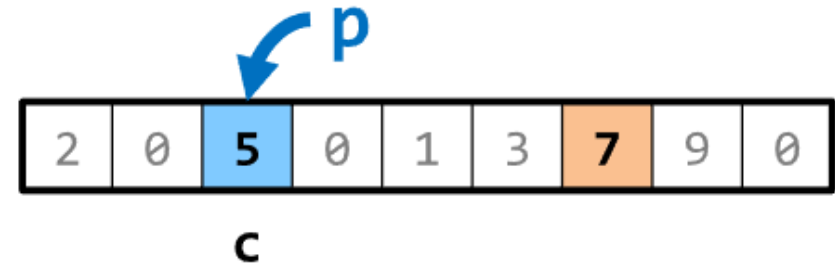
The **subscript operator** []

P[n] access value at (**address in pointer + n places**)

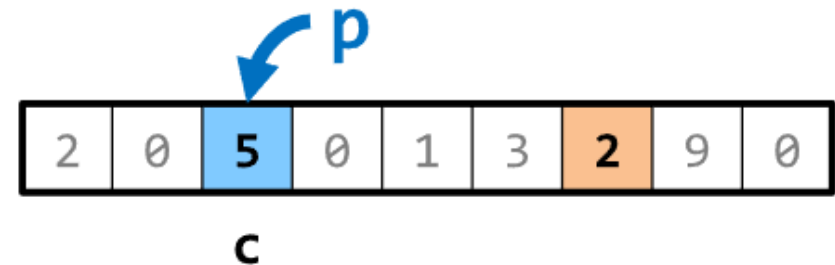
```
char c = 5;  
char* p = &c;
```



```
*(p + 4) = 7;
```



```
p[4] = 2;
```



Using the address (&) and Indirection (*) Operators #1

As seen in the program, memory locations are output as **hexadecimal** (i.e., **base-16**) integers.

The **address** of **a** (line 11) and the **value** of **aPtr** (line 12) are identical in the output, confirming that the **address** of **a** is indeed assigned to the pointer variable **aPtr**.

The outputs from *lines 13-14* confirm that ***aPtr** has the same **value** as **a**.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 7; // initialize a with 7
7
8      // initialize aPtr with the address of int variable a
9      int* aPtr = &a;
10
11     cout << "The address of a is " << &a;
12     cout << "\nThe value of aPtr is " << aPtr;
13     cout << "\n\nThe value of a is " << a;
14     cout << "\nThe value of *aPtr is " << *aPtr << endl;
15 }
```

```
The address of a is 0x61ff08
The value of aPtr is 0x61ff08
```

```
The value of a is 7
The value of *aPtr is 7
```

Using the address (&) and Indirection (*) Operators #2

Here, another example. In this program:

1. Line 15 assigns the **value 30** to the memory address stored in **ptr1**. As the pointer has the address of **num1** (line 9) this line is analogous to **num1 = 30**.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int num1, num2;
7      int *ptr1, *ptr2;
8
9      ptr1 = &num1;
10     ptr2 = &num2;
11
12     num1 = 10;
13     num2 = 20;
14
15     *ptr1 = 30;
16     *ptr2 = 40;
17
18     cout << num1 << "/n";
19     cout << num2 << "/n/n";
20
21     *ptr2 = *ptr1;
22
23     cout << num1 << "/n";
24     cout << num2 << "/n";
25
26     return 0;
27 }
```

Using the address (&) and Indirection (*) Operators #2

Here, another example. In this program:

1. Line 15 assigns the **value 30** to the memory address stored in **ptr1**. As the pointer has the address of **num1** (line 9) this line is analogous to **num1 = 30**.
2. Similarly, line 16 assigns the **value 40** to the address pointed to by **ptr2**. It is equivalent to **num2 = 40**.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int num1, num2;
7      int *ptr1, *ptr2;
8
9      ptr1 = &num1;
10     ptr2 = &num2;
11
12     num1 = 10;
13     num2 = 20;
14
15     *ptr1 = 30;
16     *ptr2 = 40;
17
18     cout << num1 << "/n";
19     cout << num2 << "/n/n";
20
21     *ptr2 = *ptr1;
22
23     cout << num1 << "/n";
24     cout << num2 << "/n";
25
26     return 0;
27 }
```


Using the address (&) and Indirection (*) Operators #2

Here, another example. In this program:

1. Line 15 assigns the **value 30** to the memory address stored in **ptr1**. As the pointer has the address of **num1** (line 9) this line is analogous to **num1 = 30**.
2. Similarly, line 16 assigns the **value 40** to the address pointed to by **ptr2**. It is equivalent to **num2 = 40**.
3. Line 21 contains two **indirections**. The expression to the right of the equal sign gets **the data at the address position** stored in **ptr1** (**num1**) and **it is stored at the position** stored in **ptr2** (**num2**).

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int num1, num2;
7      int *ptr1, *ptr2;
8
9      ptr1 = &num1;
10     ptr2 = &num2;
11
12     num1 = 10;
13     num2 = 20;
14
15     *ptr1 = 30;
16     *ptr2 = 40;
17
18     cout << num1 << "\n";
19     cout << num2 << "\n/n";
20
21     *ptr2 = *ptr1;
22
23     cout << num1 << "\n";
24     cout << num2 << "\n";
25
26     return 0;
27 }
```

Using the address (&) and Indirection (*) Operators #2

Here, another example. In this program:

1. Line 15 assigns the **value 30** to the memory address stored in **ptr1**. As the pointer has the address of **num1** (line 9) this line is analogous to **num1 = 30**.
2. Similarly, line 16 assigns the **value 40** to the address pointed to by **ptr2**. It is equivalent to **num2 = 40**.
3. Line 21 contains two **indirections**. The expression to the right of the equal sign gets **the data at the address position** stored in **ptr1** (**num1**) and **it is stored at the position** stored in **ptr2** (**num2**).

At the end of the program, the variable **num2** contains the **value 30** even though it has not been directly assigned.

30
40
30
30

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int num1, num2;
7      int *ptr1, *ptr2;
8
9      ptr1 = &num1;
10     ptr2 = &num2;
11
12     num1 = 10;
13     num2 = 20;
14
15     *ptr1 = 30;
16     *ptr2 = 40;
17
18     cout << num1 << "/n";
19     cout << num2 << "/n/n";
20
21     *ptr2 = *ptr1;
22
23     cout << num1 << "/n";
24     cout << num2 << "/n";
25
26     return 0;
27 }
```



```
4  int main()  
5  {  
6      int num1, num2;  
7      int *ptr1, *ptr2;  
8  
9      ptr1 = &num1;  
10     ptr2 = &num2;  
11  
12     num1 = 10;  
13     num2 = 20;  
14  
15     *ptr1 = 30;  
16     *ptr2 = 40;  
17  
18     *ptr2 = *ptr1;  
19  
20     return 0;  
21 }
```

Memoria		
100	????	num1
104	????	num2
108	100	ptr1
112	104	ptr2

After executing
line 10

Memoria		
100	10	num1
104	20	num2
108	100	ptr1
112	104	ptr2

After executing
line 13

Memoria		
100	30	num1
104	40	num2
108	100	ptr1
112	104	ptr2

After executing
line 16

Memoria		
100	30	num1
104	30	num2
108	100	ptr1
112	104	ptr2

After executing
line 18



Pass-by-Reference with Pointers

There are three ways in C++ to pass arguments to a function:

- pass-by-value
- pass-by-reference with a reference argument
- **pass-by-reference with a pointer argument.**

You have learned that arguments can be passed to a function using **reference parameters**, which **enable the called function to modify the original values of the arguments.**

Reference parameters also enable programs to **pass large data objects** to a function without the overhead of **pass-by-value** which, of course, **copies the object.**

Lesson review

The program passes variable number by value (line 22) to function **cubeByValue** (lines 4-7), which cubes its argument and passes the result back to **main** using a **return** statement (line 6). The new value is assigned to **num1** (line 22) in **main**.

We could have stored the result of **cubeByValue** in another variable and then return it.

```
The original value of num1 is 5
The original value of num2 is 3

The new value of num1 is 125
The new value of num2 is 27
```

```
4  int cubeByValue(int n)
5  {
6      return n*n*n;
7  }
8
9  int cubeByReference(int &n)
10 {
11     n = n*n*n;
12 }
13
14 int main()
15 {
16     int num1 = 5;
17     int num2 = 3;
18
19     cout << "The original value of num1 is " << num1;
20     cout << "\nThe original value of num2 is " << num2;
21
22     num1 = cubeByValue(num1);
23     cubeByReference(num2);
24
25     cout << "\n\nThe new value of num1 is " << num1;
26     cout << "\nThe new value of num2 is " << num2;
27
28     return 0;
29 }
```


An example of Pass-By-Reference with Pointers

The program passes the variable `number` to function `cubeByReferencePtr` using **pass-by-reference** with a pointer argument (line 16), the address of `number` is passed to the function.

The function uses the **dereferenced** pointer, `*nPtr`, an alias for `number` in `main`, to cube the value to which `nPtr` points (line 7). This directly changes the value of `number` in `main` (line 12).

```
4  int cubeByReferencePtr(int *nPtr)
5  {
6      // cube *nPtr
7      *nPtr = (*nPtr) * (*nPtr) * (*nPtr);
8  }
9
10 int main()
11 {
12     int number = 5;
13
14     cout << "The original value of number is " << number;
15
16     cubeByReferencePtr(&number);
17
18     cout << "\nThe new value of number is " << number;
19
20     return 0;
21 }
```

The original value of number is 5
The new value of number is 125

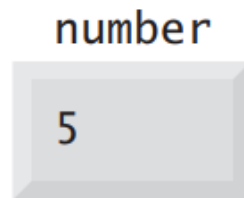


Pass-By-Reference with a Pointer Actually Passes the Pointer By Value

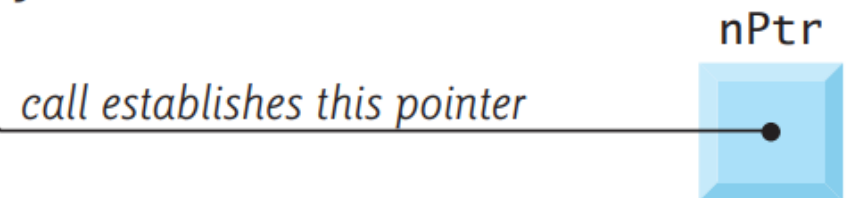
Passing a variable by reference with a pointer **does not actually pass anything by reference**, a pointer to that variable is **passed by value** and is copied into the function's corresponding pointer parameter.

The called function can then access that variable in the caller simply by **dereferencing** the pointer, thus accomplishing **pass-by-reference**.

```
int main() {  
    int number{5};  
    cubeByReference(&number);  
}
```



```
void cubeByReference( int* nPtr ) {  
    *nPtr = *nPtr * *nPtr * *nPtr;  
}
```





Actual Passing Reference to a Pointer (1/2)

Passing If a **pointer is passed to a function** as a parameter and **tried to be modified** then the **changes made to the pointer does not reflects back outside that function**. This is because only a copy of the pointer is passed to the function.

Instead of modifying the variable, you are only **modifying a copy of the pointer** and the original pointer remains unmodified.

Before: 23
After: 23

```
int global_Var = 42;

void changePointerValue(int* pp)
{
    pp = &global_Var;
}

int main()
{
    int var = 23;
    int* ptr_to_var = &var;

    cout << "Before: " << *ptr_to_var << endl;

    changePointerValue(ptr_to_var);

    cout << "After: " << *ptr_to_var << endl;

    return 0;
}
```



Actual Passing Reference to a Pointer (2/2)

The previous problem can be resolved by passing the **address of the pointer &** to the function instead of a copy.

program shows how to pass a **"Reference to a pointer"** to a function:

```
Before: 23
After: 42
```

```
int global_Var = 42;

void changePointerValue(int* &pp)
{
    pp = &global_Var;
}

int main()
{
    int var = 23;
    int* ptr_to_var = &var;

    cout << "Before: " << *ptr_to_var << endl;

    changePointerValue(ptr_to_var);

    cout << "After: " << *ptr_to_var << endl;

    return 0;
}
```



Pointers to an Array

As a principle the **name of an array** is a **constant pointer** to the **first element of the array**. That is, in

```
int sArr[5] = {1, 2, 3, 4, 5};
```

sArr is a pointer to **&sArr[0]** which is the address of the first element of the array. Therefore, using a pointer to manipulate an array makes sense.

```
int *sp;
```

```
sp = sArr; or sp = &sArr[0];
```

Since arrays occupy consecutive memory spaces, we can access all the data in the array.



Pointer / Offset Notation

If we would like to point to element `sArr[3]` in the array. The expression

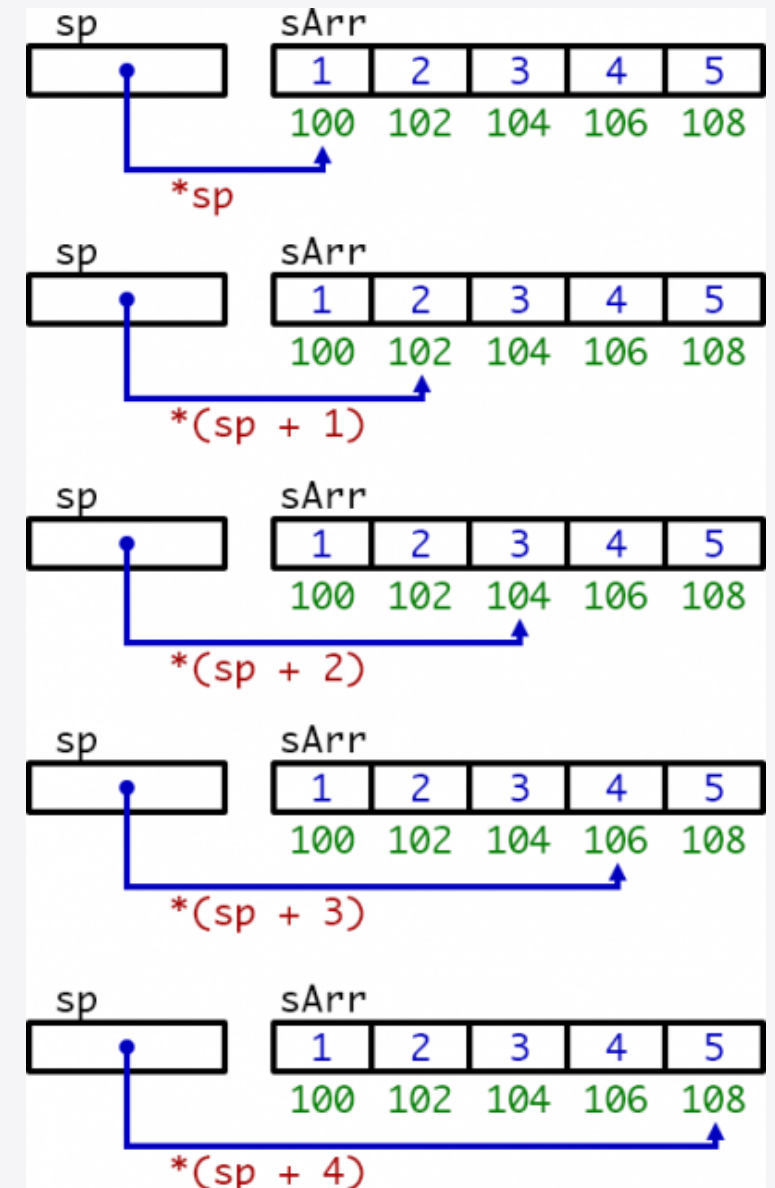
```
sArr += 3;
```

causes a **compilation error**, because it attempts to modify the built-in **array's constant pointer**. Alternatively, we use a pointer expression

```
*(sp + 3)
```

The **3** in the preceding expression is the **offset** to the pointer.

Since **arrays occupy consecutive memory spaces**, we can access all the data in the array.





Pointers to Pointer

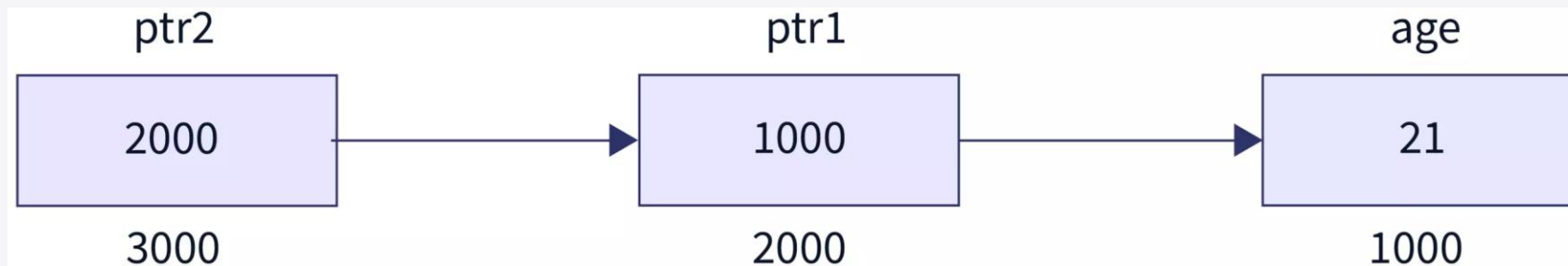
Declaring a Pointer to a Pointer in C++ or **double-pointer** is very similar to declaring a single pointer; the only difference is that an extra ***** is added.

The general syntax is

<data type> **<identifier name>;

Here **data type** is the datatype of the single pointer we want to point.

For example, if you want to store the address of an **int** variable, then the datatype of the **single pointer** would be **int**, and the double-pointer's data type would also be **int**.

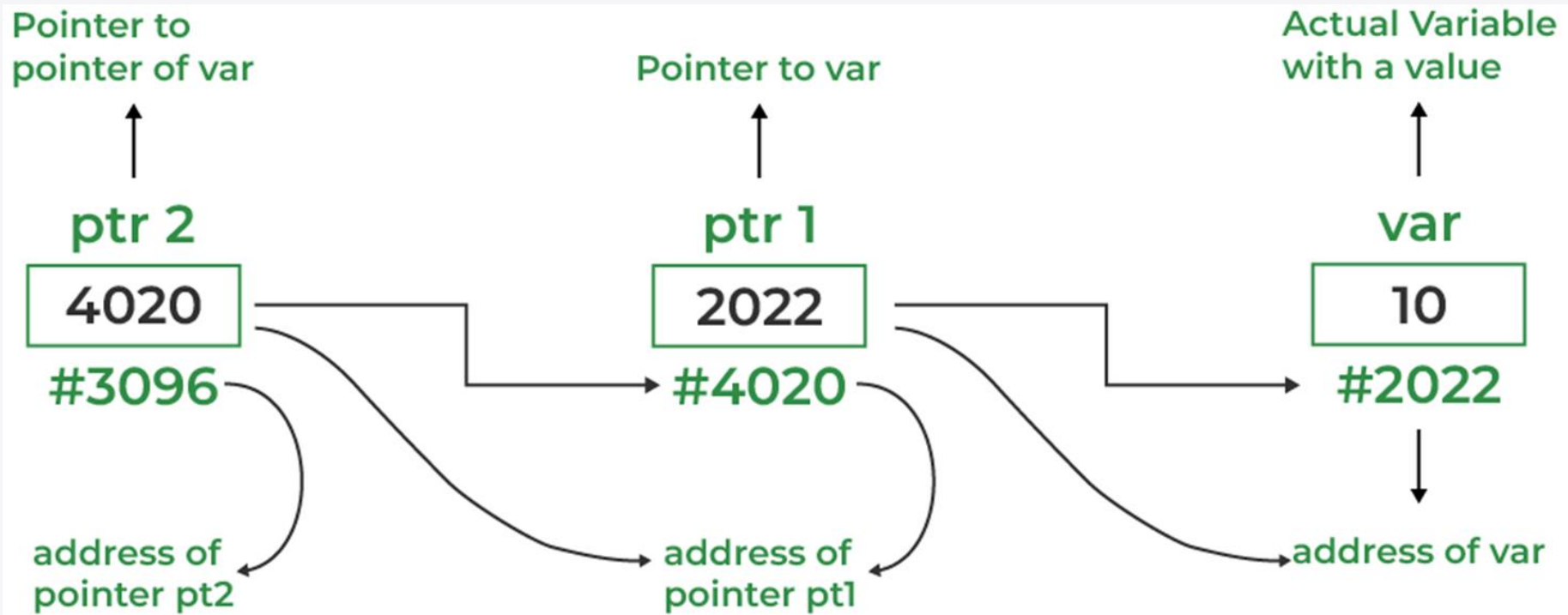


Pointer to Pointer - Example #1

```
int main()
{
    int val = 169;

    int *ptr1 = &val; // storing address of val to pointer ptr.

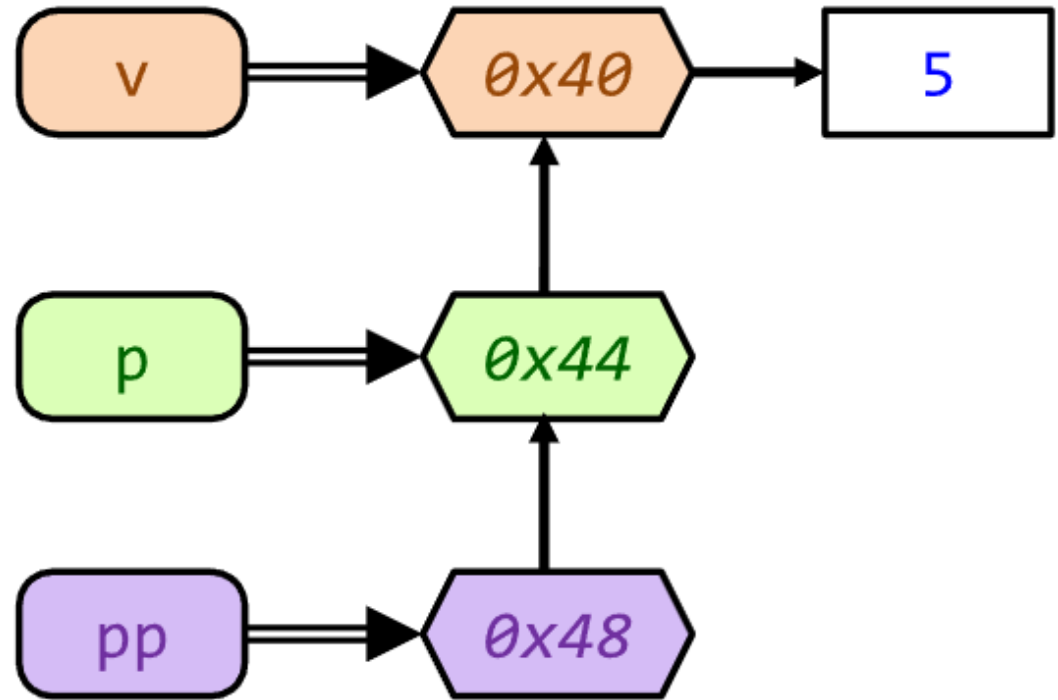
    // pointer to a pointer declared which is pointing to an integer.
    int **ptr2 = &ptr1;
}
```



Pointer to Pointer - Example #2

```
int    v = 5;  
int*   p = &v;  
int**  pp = &p;
```

```
cout << v;           // 5  
cout << p;           // 0x40  
cout << pp;          // 0x44  
  
cout << &v;          // 0x40 (= p)  
cout << &p;           // 0x44 (= pp)  
cout << &pp;          // 0x48  
  
cout << *p;           // 5  
cout << *pp;          // 0x40 (= p)  
cout << **pp;         // 5
```



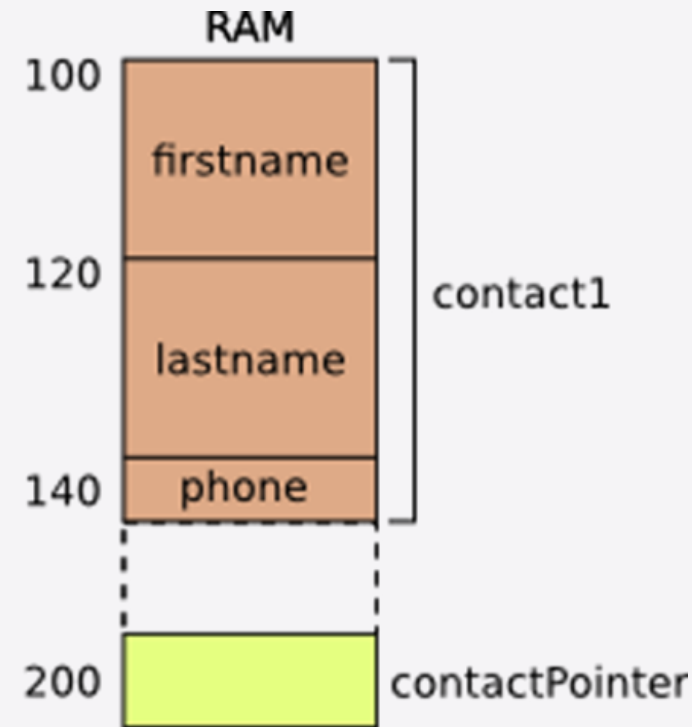


Pointers to data structures

For **data structures**, the rule applies identically. The following example show the declaration of a variable of **type structure** and another of type pointer to that structure:

```
struct contact
{
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact contact1;
struct contact *contactPointer;
```



The **contact1** variable is of **type structure** and occupies **44 bytes** (20+20+4), while **contactPointer** is of **type pointer** and occupies **4 bytes**.



Selection operators (., ->)

When a pointer **indirection** is performed, it is only to access the variables of the structure. This operation requires two operators, the **indirection** (operator **"*"**) and the **access to the variable** of a structure (operator **"."**). The syntax is:

```
(*pointer).x = 10;
```

```
(*pointer).c = 'l';
```

These two operations are grouped in the **operator "->"** which is placed between the pointer and the variable name of the structure. The equivalent notation, therefore, is:

```
pointer->x = 10;
```

```
pointer->c = 'l';
```

```
struct s
{
    int x;
    char c;
};

struct s element;
struct s *Ptr;

Ptr = &element;
```



The **operator** "**->**" is always written after a pointer pointing to a structure, and precedes the name of one of the **variables** of that structure.

The following program shows how data can be copied from one structure to another using this operator.

3.5
5.5
10.5
3.5
5.5
10.5

```
struct coordinates
{
    float x, y, z;
};

int main()
{
    struct coordinates loc1, loc2;
    struct coordinates *Ptr1, *Ptr2;

    Ptr1 = &loc1;
    Ptr2 = &loc2;


    Ptr1->x = 3.5;
    Ptr1->y = 5.5;
    Ptr1->z = 10.5;

    cout << loc1.x << endl;
    cout << loc1.y << endl;
    cout << loc1.z << endl << endl;

    Ptr2->x = Ptr1->x;
    Ptr2->y = Ptr1->y;
    Ptr2->z = Ptr1->z;

    cout << loc2.x << endl;
    cout << loc2.y << endl;
    cout << loc2.z << endl;


    return 0;
}
```



Find the error in each of the following program segments. Assume the following declarations and statements:

```
int *zPtr; // zPtr will reference built-in array z
void *sPtr = nullptr;
int number;
int z[ 6 ] = { 10, 20, 30, 40, 50, 60 };
```

- a) zPtr = zPtr + 2;
- b) array number = zPtr + 2; // use pointer to get second value of a built-in
- c) number = *zPtr(3); // assign built-in array element 3 (the value 40) to number
- d) // display entire built-in array z in reverse order
 for (size_t i = 6; i > 0; --i)
 cout << zPtr[i] << endl;
- e) z[0] = *sPtr; // assign the value pointed to by sPtr to first element of z
- f) z++;



a) **Error:** zPtr has not been initialized. **Correction:** Initialize zPtr with `zPtr = z;`
(Parts b-e depend on this correction.)

b) **Error:** The pointer is not dereferenced. **Correction:** Change the statement to `number = *zPtr;`

c) **Error:** `zPtr[2]` is not a pointer and should not be dereferenced. **Correction:** Change `*zPtr[2]` to `zPtr[2]`.

d) **Error:** Referring to an out-of-bounds built-in array element with pointer subscripting. **Correction:** To prevent this, change the relational operator in the for statement to `<` or change the 5 to a 4.

e) **Error:** Trying to modify a built-in array's name with pointer arithmetic. **Correction:** Use a pointer variable instead of the built-in array's name to accomplish pointer arithmetic, or subscript the built-in array's name to refer to a specific element.



Dynamic Memory

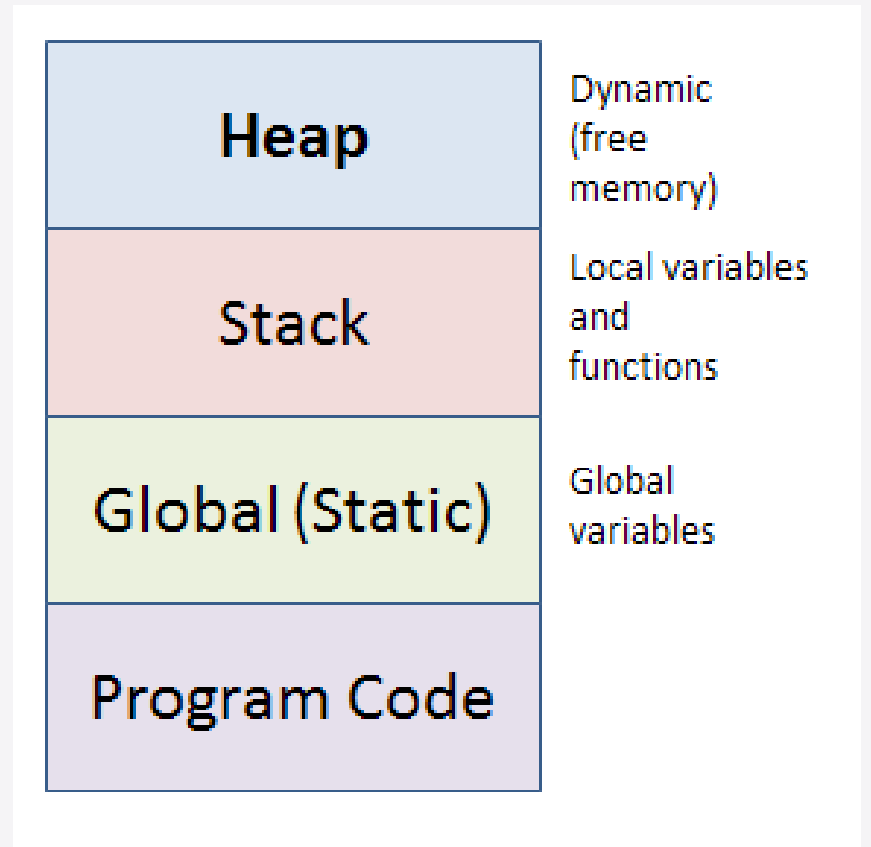


Memory management

Memory is classified into many categories. The operating system reserves one area of memory for program **code or instructions** and another for **local variables** that are used during runtime(**stack**).

The rest of the memory that is not used by any program is known as **Heap**.

Our program can make use of the **heap** to vary the size of our application at runtime, that is why it is called **dynamic memory**.

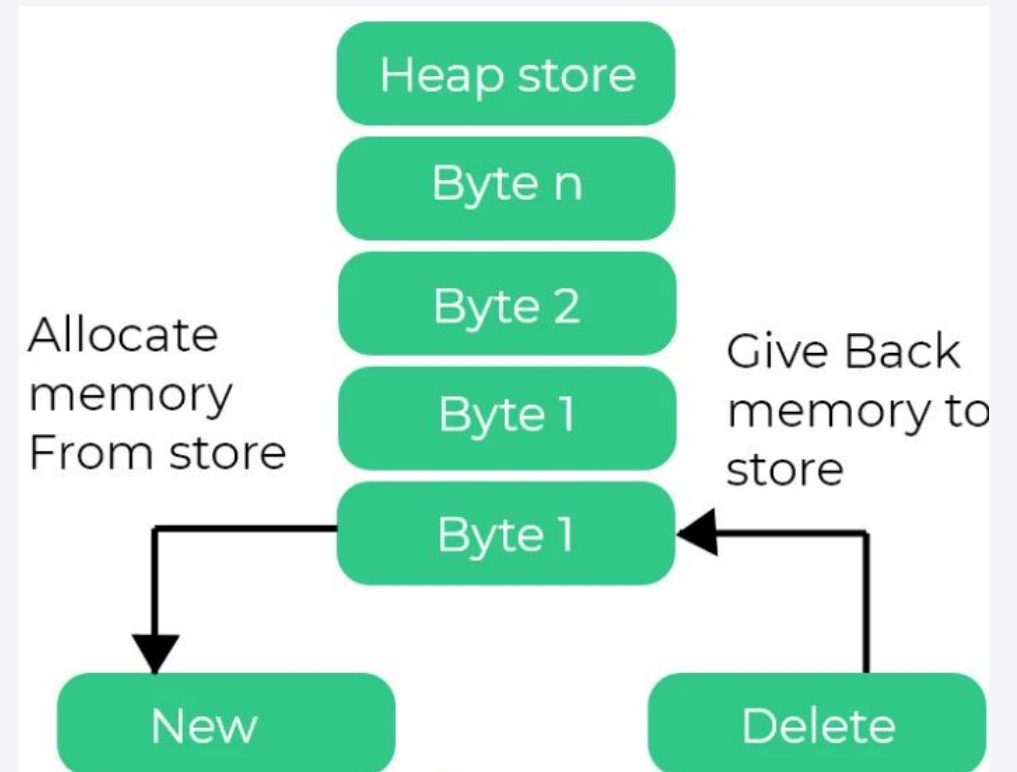


Operator "new" & "delete"

C++ has two operators to **reserve** and **free dynamic memory**, they are "**new**" and "**delete**". All reserved memory must be freed before exiting the program.

The **malloc()** function from C, still exists in C++, but **it is recommended to avoid it**.

The main advantage of **new** over **malloc()** is that doesn't just **allocate memory**, it **constructs objects** (initialize the class constructors).





"new" Operator

The generic syntax to use **new** operator to allocate memory dynamically for any data-type.

pointer = **new** <data type>;

or

pointer = **new** <data type> [number_of_elements];

Here, **data-type** could be any built-in data type including an **array** or any user defined data types include **class** or **structure**.

Example. we request memory to allocate a **float** variable at run time.

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double;   // Request memory for the variable
```



"delete" Operators

At any point, when you feel a variable that has been dynamically allocated is not anymore required, **you can free up the memory that it occupies in the free store** with the "**delete**" operator as follows:

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double;    // Request memory for the variable
```

```
delete pvalue;          // Release memory pointed to by pvalue
```

Example

```
#include <iostream>
using namespace std;

int main () {
    double* pvalue = NULL; // Pointer initialized with null
    pvalue = new double;    // Request memory for the variable

    *pvalue = 29494.99;     // Store value at allocated address
    cout << "Value of pvalue : " << *pvalue << endl;

    delete pvalue;         // free up the memory.

    return 0;
}
```

If we compile and run above code, this will produce the following result

```
Value of pvalue : 29495
```



Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an **array of characters**, e.g., **20 characters**.

```
char* pvalue = NULL;           // Pointer initialized with null
pvalue = new char[20];         // Request memory for the variable
```

To **remove** the array that we have just created the statement would look like this

```
delete [] pvalue;              // Delete array pointed to by pvalue
```



Following the similar generic syntax of **new** operator, you can allocate for a **fixed size multi-dimensional array** as follows

```
double (*pvalue)[4] = new double[3][4];
```

or

```
auto pvalue = new double[3][4];
```

or

```
double *pvalue = new double[3 * 4];
```

However, the syntax to release the memory for multi-dimensional array will remain same as previously:

```
delete []pvalue;
```



However, if you had a **dynamic size multi-dimensional array** as follows

```
//create an array of pointers that points to more arrays
int **matrix = new int*[5];
for(int i=0; i<5; i++)
{
    matrix[i] = new int[5];
    for(int j=0; j<5; j++)
    {
        matrix[i][j] = i*5 + j;
    }
}
```

The recommendation to release the memory will be

```
//Free each sub-array
for(int i=0; i<5; i++)
{
    delete[] matrix[i];
}
//Free the array of pointers
delete[] matrix;
```



Dynamic Memory Allocation for Objects

Objects are no different from simpler data types.

Remember that **new** allocate memory and **initialize the class constructors**.

Consider the following code where we are going to use an **array of objects**:

```
#include <iostream>
using namespace std;

class Box {
public:
    Box() {
        cout << "Constructor called!" <<endl;
    }
    ~Box() {
        cout << "Destructor called!" <<endl;
    }
};

int main() {
    Box* myBoxArray = new Box[4];
    delete [] myBoxArray; // Delete array

    return 0;
}
```




If you were to allocate an array of **4 Box objects**, the **constructor** would be called **4 times** and similarly while deleting these objects, **destructor** will also be called same number of times.

If we compile and run above code, this will produce the following result:

```
Constructor called!  
Constructor called!  
Constructor called!  
Constructor called!  
Destructor called!  
Destructor called!  
Destructor called!  
Destructor called!
```

```
#include <iostream>  
using namespace std;  
  
class Box {  
    public:  
        Box() {  
            cout << "Constructor called!" <<endl;  
        }  
        ~Box() {  
            cout << "Destructor called!" <<endl;  
        }  
};  
  
int main() {  
    Box* myBoxArray = new Box[4];  
    delete [] myBoxArray; // Delete array  
  
    return 0;  
}
```