Dynamic Memory management

The size of a static array data structure is fixed at compilation time at some constant value. Sometime it is useful to determine the array size dynamically at run time.

C++ allows dynamically allocating and deallocating memory for any object at run time using the operators **new** and **delete**. The **new** operator is used to dynamically allocate (reserve) the exact amount of memory required to hold an object or array.

The memory allocated using the **new** operator is reserved on the free store (the heap). The heap is a region of memory assigned to each program.

Once memory is allocated, we can access it using the pointer returned by the new operator.

When we no longer need the memory, we can return it to the free store using the **delete** operator.

Allocating dynamic memory

The syntax of allocating memory for an object of data type "type" is:

type * p = new type;

this will reserve sizeof(type) bytes in the heap and returns a pointer to that object.

int * p = new int;

will allocate sizeof(int) bytes in the heap and assigns its address to the pointer p.

Initializing Dynamic Memory

We can initialize a dynamically allocated variable once it is allocated. For example,

double * ptr = new double(2.5);

initializes a newly created double to 2.5 and assigns its address to the pointer ptr.

Releasing dynamic memory:

To free a dynamically allocated memory for an object we use the operator delete.

delete typePtr;

this call will de-allocate the memory associated with object pointed to by typePtr.

```
double *p  =  new double(6.7);
delete p;
```

Dynamically allocating array:

The **new** operator can be used to allocate contiguous block of memory for an array.

int * arrptr = new int[100];

dynamically allocates 100 -- array elements of integers and assigns the address of its first element to the pointer arrptr.

Releasing dynamically allocated array

To free a dynamically allocated memory for an array we use the operator delete []

```
short * ptr = new short[1000];
delete [] ptr;
```

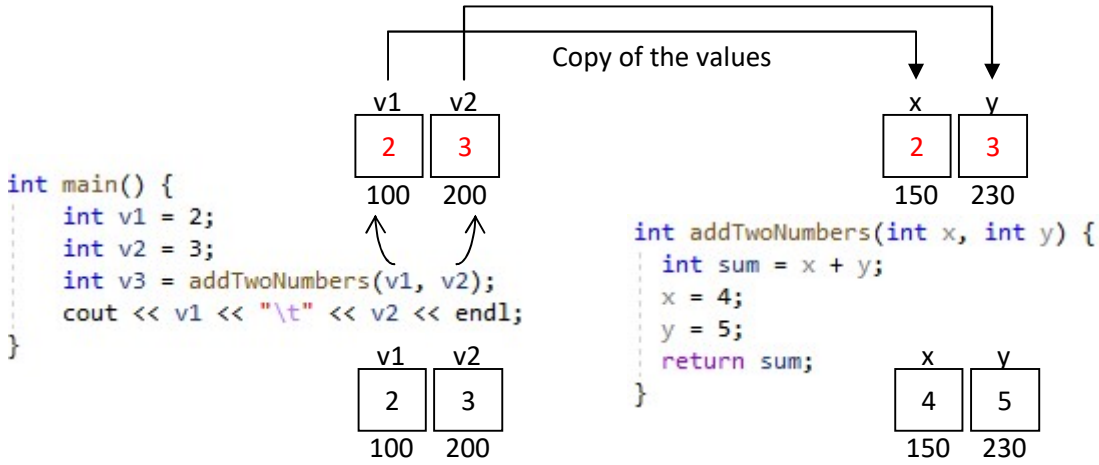
Passing parameters to function

There are three ways that can be used to pass parameters to/from functions. But first let's introduce some terminology.

- If function A() called function B(), then A is called **the caller** and function B is called **the callee**
- **Formal Parameter:** The variables and their data types as they appear in the prototype of the function.
- **Actual Parameter:** The values corresponding to the formal parameters that appear in the caller function.

Passing by value:

When passing by value, changes made by the callee function do not alter the actual values in the caller. The modifications made by the callee change a copy of the actual parameters.



Pass by value: a copy of the actual parameters is used in the formal parameters

- Passing by value has the following advantages:
  1) Arguments in the caller can be constants, variables, or expressions

```
int addOne(int x) {
	return x + 1;
}

int main() {
	int y = 4;
	addOne(y);
	addOne(3);
	addOne(2*y+3);
	return 0;
}
```

  2) The original data is protected, the function can modify the data without affecting the original data in the caller

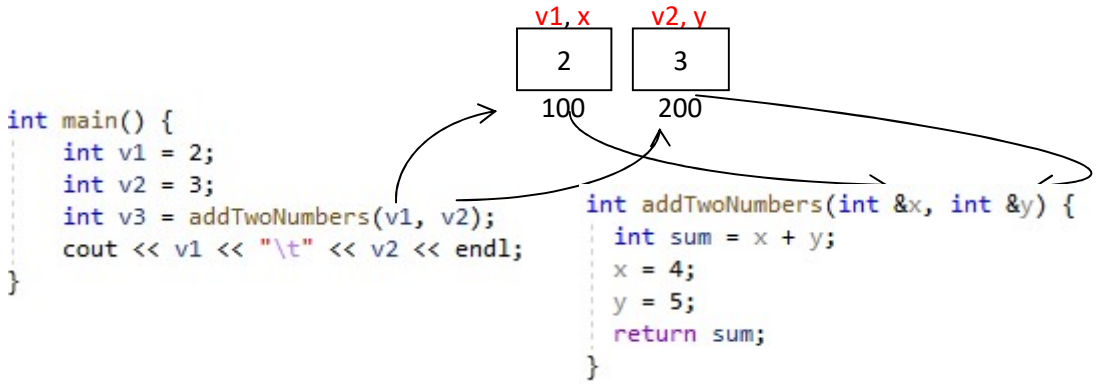
  Example:

```
void changer(int x) {
	x = x + 1;
}

int main() {
	int y = 4;
	cout << "Before the call: " << y << endl; // output: 4
	changer(y);
	cout << "After the call: "  << y << endl; // output: 4
	return 0;
}
```

- However, this way of passing is not efficient when passing large object waste of memory and performance of making the copies

Passing by reference:

In this case there is only one copy of the variables, the formal parameters are just aliases to the actual parameters. If the callee makes any modifications to the parameters, the actual values in the caller will be changed as well.



Pass by reference: only one copy of the actual parameters is used. The formal parameters are aliases (notice the & symbole)

Example

```
void changer(int &x) {
	x = x + 1;
}

int main() {
	int y = 4;
	cout << "Before the call: " << y << endl; // output: 4
	changer(y);
	cout << "After the call: "  << y << endl; // output: 5
	return 0;
}
```

The main advantages of passing by reference are:

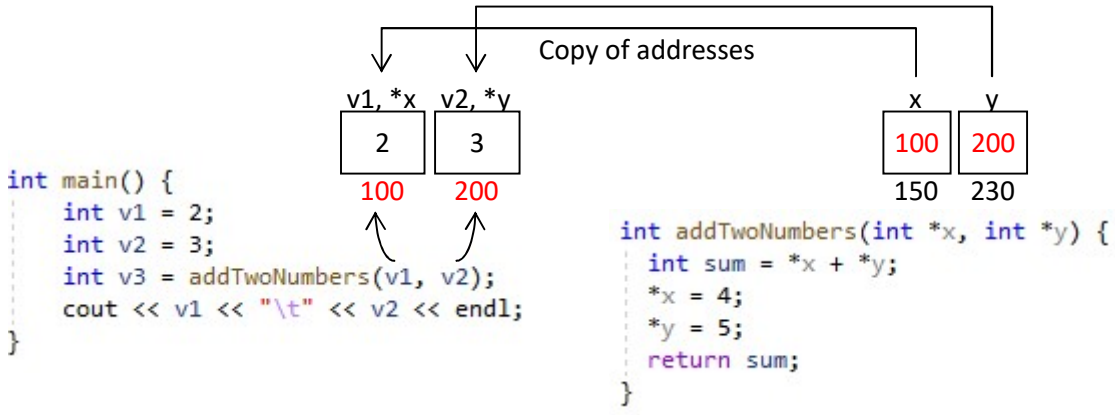
1) We can return more values from the function to the caller

```
void plusMinusOne(int x, int &p, int &m) {
    p = x + 1;
    m = x - 1;
}

int main() {
    int p, m;
    plusMinusOne(10, p, m);
    cout << p << endl; // output: 11
    cout << m << endl; // output: 9
    return 0;
}
```

2) More efficient in terms of memory and speed when passing large objects

Passing by address:

In passing by address, a copy of the addresses (not the values) of the actual parameters is passed to the function. The formal parameters are pointers to the actual parameters. The actual variables can be accessed through these pointers (*x and *y) from the callee.



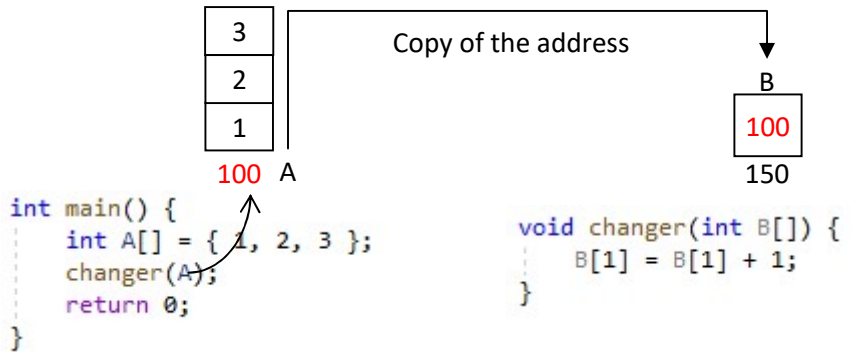
Passing by address: a copy of the addresses of the actual parameters is passed to the callee. The formal parameters are just pointers to the actual parameters

Example

```
void changer(int *x) {
	*x = *x + 1;
}

int main() {
	int y = 4;
	cout << "Before the call: " << y << endl; // output: 4

	changer(y);    // error: can pass value where address is expected
	changer(&y);   // correct we bass a copy of the address of y

	cout << "After the call: "  << y << endl; // output: 5
	return 0;
}
```

Passing 1D array to a function:

Arrays are passed by **address** only whether the formal parameters are declared as (int A[]) or (int *A) . This means only **a copy of the address of the first element** is passed to the function. So, if the function modifies the array elements, that modification will change the values of the array in the caller as well. But if the callee changes the pointer itself, that change will not affect the array in the caller.



Passing arrays: always by address, a copy of the address of the first element of the array is passed by value .

Example

```
void changer(int B[]) {
	cout << "B: " << (uintptr_t) B << endl; // prints the address of the incoming
	                                        // array which is saved in B
	cout << "B: " << (uintptr_t) &B << endl; // prints the address of the pointer B
	int x = 7;
	B = &x;  // this is ok since B is a pointer
}

int main() {
	int A[] = { 1, 2, 3 };
	cout << "A: " << (uintptr_t) &A << endl;  // the address of the first element of A
	cout << "A: " << (uintptr_t) A << endl;  // the address of the first element of A
	int x = 3;
	changer(A);
	A = &x;           // error: A is not an lvalue (it is not a pointer)

	return 0;
}
```

Example

```
void changer(int *B) { // same as int B[]
	B[1] = B[1] * 10;
	B = NULL;  // B points to null now
}

int main() {
	int A[] = { 1, 2, 3 };
	cout << "A[1]: " <<  A[1] << endl;  // output: 2
	changer(A);
	cout << "A[1]: " <<  A[1] << endl;  // output: 20

	return 0;
}
```

Using const with pointers

The const qualifier tells the compiler that the value of a variable can't be modified after initialization. This is useful for type safety when passing parameters by reference or by address. When passing by value, it is quarantine that the function can't modify the callers' variables, but it is inefficient. Therefore, for performance, we typically pass by reference or address combined with using the const qualifier. The const tells the compiler that the function shouldn't modify the variable and if it does the compiler will generate an error. Another, advantage of the const qualifier is that provide some documentation to the other programmers reading the code. It tells them that the function will not change these parameters.

- A non-constant pointer to a non-constant data: allow modifying both the data and the pointer

  char *p = &ch;
  void func(char *p);

- A non-constant pointer to a const data: allow modifying the pointer but not the data.

  const char *p = &ch;
  void func(const char *p);

- A constant pointer to non-constant data: allow modifying the data, but not the pointer

  char * const p = &ch;
  void func(char * const p);

- A constant pointer to a constant data: both the pointer and the data can't be modified

  const char * const p = & ch;
  void func(const char * const p);

Example:

```
void changerA(const int * B) {
	B[1] = B[1] + 1;  // compilation error>: can't change const data
	int x = 3;
	B = &x;           // Ok, can change non-const pointer B
}

void changerB(int * const B) {
	B[1] = B[1] + 1;  // ok, can change non-const data
	int x = 3;
	B = &x;           // error, can't change the const pointer
}

int main() {
	int A[] = { 1, 2, 3 };
	int x = 3;
	changer(A);
	return 0;
}
```