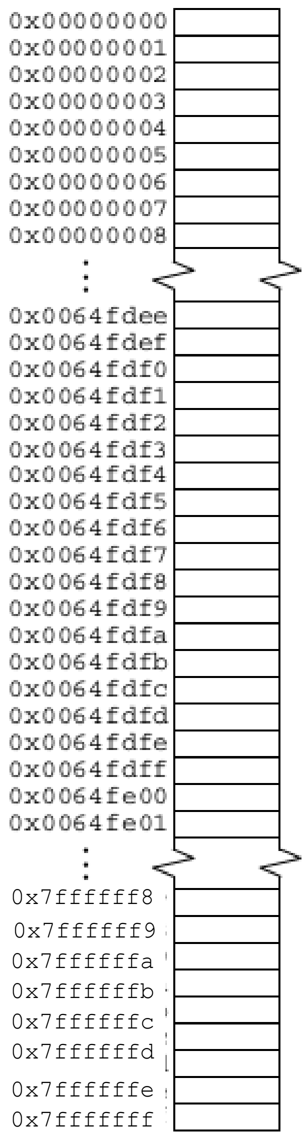
Pointers and References

The reference operator

- Computer memory can be imagined as a very large array of bytes. For example, a computer with 2 GB of RAM contains an array of 2.147.483.648 (2^{31}) bytes.
  - As an array, these bytes are indexed from 0 to 2.147.483.647.
  - The index of each byte is its memory address from 0x00000000 to 9x7FFFFFFF in hexadecimal

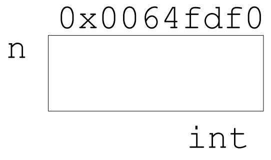


The reference operator

- A variable declaration associates three fundamental attributes to the variable: its name, its type, and its memory address. For example, the declaration

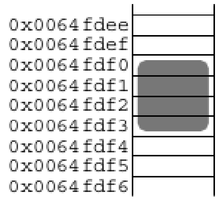
  int n;

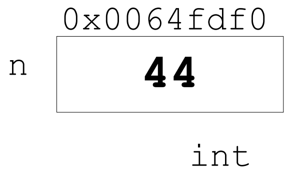
  associates the name n , the type int , and the address of some location in memory where the value of n is stored. Suppose that address is 0x0064fdf0 .
- Then we can visualize n like this:

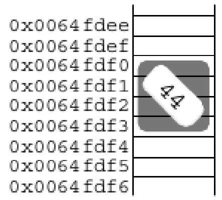


- Variables of type int occupy 4 bytes in memory → the variable n shown would occupy 4-byte block of memory
  - Note that the address of the object is the address of the first byte in the block of memory where the object is stored.
- If the variable is initialized, like this:

  int n=44;
- then the two representations look like this:







- The variable's value 44 is stored in the four bytes allocated to it.

The reference operator

- In C++, you can obtain the address of a variable by using the reference operator **&** , also called the *address operator*. The expression **&n** evaluates to the address of the variable n.

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

int main(){
    int n=44;
    cout << "n = " << n << endl;
    // prints the value of n
    cout << "&n = " << &n << endl; // prints the address of n
}
```

```
n = 44
&n = 0x7fffeac4f0a4
```

- The output shows that the address of n is 0x0064fdf0 . You can tell that the output `0x7fffeac4f0a4` must be an address because it is given in hexadecimal form, identified by its `0x` prefix.
- Displaying the address of a variable this way is not very useful. The reference operator **&** has other more important uses.
  - We already saw one use in the previous lesson : designating reference parameters in a function declaration That use is closely tied to another: declaring reference variables.

FirstPointer.cpp

References

- A reference is an *alias* or *synonym* for another variable. It is declared by the syntax

  `type& ref-name = var-name;`

  where `type` is the variable's type, `ref-name` is the name of the reference, and `var-name` is the name of the variable.
- For example, in the declaration

  `int& rn=n;`

  `rn` is declared to be a reference to the variable `n` , which must already have been declared.

```
int main(){
  int n=44;
  int& rn=n; // r is a synonym for n
  cout << "n = " << n << ", rn = " << rn << endl;
  --n;
  cout << "n = " << n << ", rn = " << rn << endl;
  rn *= 2;
  cout << "n = " << n << ", rn = " << rn << endl;
}
```

```
n = 44, rn = 44
n = 43, rn = 43
n = 86, rn = 86
```

The two identifiers `n` and `rn` are different names for the same variable; they always have the same value. Decrementing n changes both n and nr to 32. Doubling rn increases both n and rn to 64.

SecondPointer.cpp

References

- Like constants, references must be initialized when they are declared. But unlike a constant, a reference must be initialized to a variable, not a literal:

```
int& rn=44; // ERROR: 44 is not a variable!
```

  - Some compilers may allow this, issuing a warning that a temporary variable had to be created to allocate memory to which the reference `rn` can refer.
- Although a reference must be initialized to a variable, references are not variables:
  - A variable is an object; i.e., a block of contiguous bytes in memory used to store accessible information.

References Are Not Separate Variables

```
#include <iostream>
using namespace std;
int main()
{
  int n=44;
  int& rn=n; // r is a synonym for n
  cout << " &n = " << &n << ", &rn = " << &rn << endl;
  int& rn2=n;
  // r is another synonym for n
  int& rn3=rn; // r is another synonym for n
  cout << "&rn2 = " << &rn2 << ", &rn3 = " << &rn3 << endl;
}
```

```
 &n = 0x7fff375564ec, &rn = 0x7fff375564ec
&rn2 = 0x7fff375564ec, &rn3 = 0x7fff375564ec
```

The first line of output shows that n and rn have the same address: `0x7fff375564ec`. Thus they are merely different names for the same object.

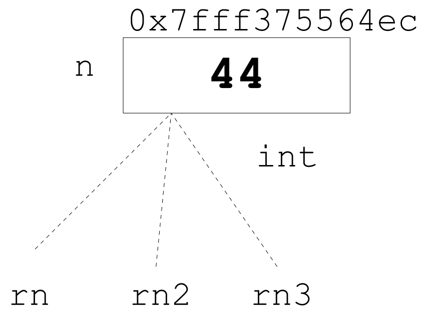
RefAndVariable.cpp

References Are Not Separate Variables

```
#include <iostream>
using namespace std;
int main()
{
  int n=44;
  int& rn=n; // r is a synonym for n
  cout << " &n = " << &n << ", &rn = " << &rn << endl;
  int& rn2=n;
  // r is another synonym for n
  int& rn3=rn; // r is another synonym for n
  cout << "&rn2 = " << &rn2 << ", &rn3 = " << &rn3 << endl;
}
```

```
 &n = 0x7fff375564ec, &rn = 0x7fff375564ec
&rn2 = 0x7fff375564ec, &rn3 = 0x7fff375564ec
```

The first line of output shows that n and rn have the same address: `0x7fff375564ec` . Thus they are merely different names for the same object.
The second line of output shows that an object can have several references, and that a reference to a reference is the same as a reference to the object to which it refers.
In this program, rn, rn2 and rn3 there is only one object: an int named n with address `0x7fff375564ec` .
The names `rn` , `rn2` , and `rn3` are all references to that same object.



RefAndVariable.cpp

Pointers

- The reference operator `&` returns the memory address of the variable to which it is applied.
- It is also store the *address* in another variable.
- The type of the variable that stores an address is called a **pointer**.
- Pointer variables have the derived type “pointer to T ”, where T is the type of the object to which the pointer points. For example, the address of an int variable can be stored in a pointer variable of type `int*` .

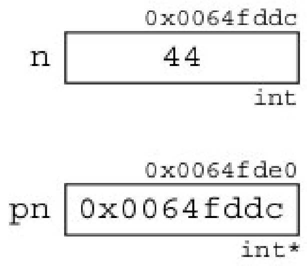
Pointers

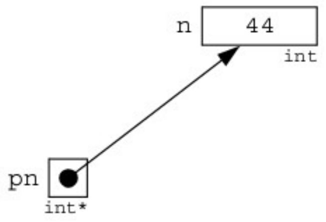
```
int main()
{
  int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n; // pn holds the address of n
  cout << "pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;

  return 0;
}
```

```
n = 44, &n = 0x7ffdd685155c
pn = 0x7ffdd685155c
&pn = 0x7ffdd6851560
```

The variable n is initialized to 44. Its address is `0x7ffdd685155c` . The variable `pn` is initialized to `&n` which is the address of n , so the value of pn is `0x7ffdd685155c`, as the second line of output shows. But `pn` is a separate object, as the third line of output shows: it has the distinct address `0x7ffdd6851560`.





Pointer1.cpp

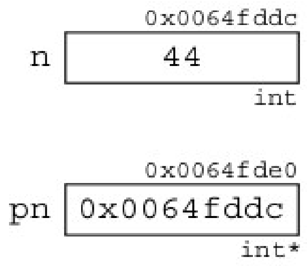
Pointers

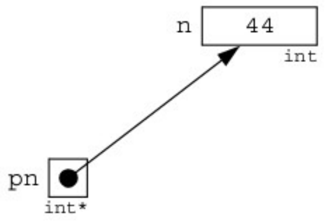
```
int main()
{
  int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n; // pn holds the address of n
  cout << "pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;

  return 0;
}
```

```
n = 44, &n = 0x7ffdd685155c
pn = 0x7ffdd685155c
&pn = 0x7ffdd6851560
```

The variable n is initialized to 44. Its address is `0x7ffdd685155c` . The variable `pn` is initialized to `&n` which is the address of n , so the value of pn is `0x7ffdd685155c`, as the second line of output shows. But `pn` is a separate object, as the third line of output shows: it has the distinct address `0x7ffdd6851560`.





Pointer1.cpp

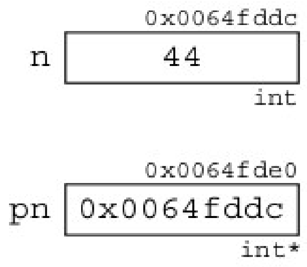
Pointers

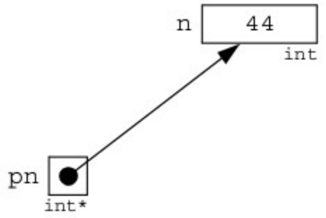
```
int main()
{
  int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n; // pn holds the address of n
  cout << "pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;

  return 0;
}
```

```
n = 44, &n = 0x7ffdd685155c
pn = 0x7ffdd685155c
&pn = 0x7ffdd6851560
```

The variable n is initialized to 44. Its address is `0x7ffdd685155c` . The variable `pn` is initialized to `&n` which is the address of n , so the value of pn is `0x7ffdd685155c`, as the second line of output shows. But `pn` is a separate object, as the third line of output shows: it has the distinct address `0x7ffdd6851560`.





Pointer1.cpp

The deference operator

If `pn` points to `n`, we can obtain the value of `n` directly from `p`; the expression `*pn` evaluates to the value of `n`. This evaluation is called “dereferencing the pointer” `pn`, and the symbol ***** is called the dereference operator.

```
int main()
{
  int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n; // pn holds the address of n
  cout << "pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;
  cout << "*pn = " << *pn << endl;
  return 0;
}
```

```
n = 44, &n = 0x7ffdd685155c
pn = 0x7ffdd685155c
&pn = 0x7ffdd6851560
*pn = 44
```

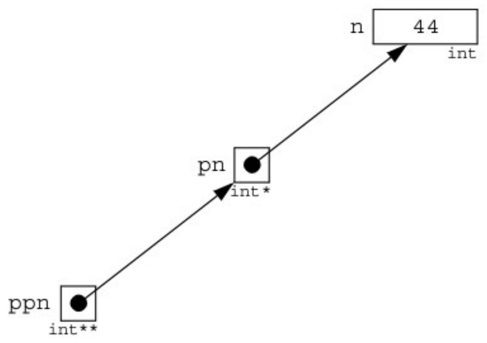
This shows that `*pn` is an alias for `n` : they both have the value 44.

Pointers to Pointers

```
int main()
{
  int n=44;

  cout << "n = " << n << endl;
  cout << "&n = " << &n << endl;
  int* pn=&n; // pn holds the address of n
  cout << "pn = " << pn << endl;
  cout << " &pn = " << &pn << endl;
  cout << " *pn = " << *pn << endl;
  int** ppn=&pn; // ppn holds the address of pn
  cout << " ppn = " << ppn << endl;
  cout << " &ppn = " << &ppn << endl;
  cout << " *ppn = " << *ppn << endl;
  cout << "**ppn = " << **ppn << endl;

  return 0;
}
```



```
n = 44
&n = 0x7ffc5b8ea2a4
pn = 0x7ffc5b8ea2a4
 &pn = 0x7ffc5b8ea2a8
 *pn = 44
 ppn = 0x7ffc5b8ea2a8
 &ppn = 0x7ffc5b8ea2b0
 *ppn = 0x7ffc5b8ea2a4
**ppn = 44
```

Derived types

- Like the reference operator `&` , the dereference operator `*` is used for two distinct purposes:
  - When applied as a prefix to a pointer to an object, it forms an expression that evaluates to that object's value.
  - When applied as a suffix to a type `T` , it names the derived type "pointer to T ". For example, `int*` is the type "pointer to int "
- In C++ there are five kinds of derived types:

```
const int C = 33;            //const int
int& rn = n;                 //reference to int
int* pn = &n;                //pointer to int
int a[] = { 33, 66 };        //array of int
int f() = { return 33; };    //function returning int
```

- A derived type can derive from any other type. So many combinations are possible:

```
int* const Pn=44;             // constant pointer to an int
const int* pN=&N;             // pointer to a constant int
const int* const PN=&N;       // constant pointer to a constant int
float& ar[] = { x, y };       // array of 2 references to floats
float* ap[] = { &x, &y };     // array of 2 pointers to floats
long& r() { return n; }       // function returning reference to long
long* p() { return &n; }      // function returning pointer to long
long (*pf)() { return 44; } // pointer to function returning long
```

- Some derived types require the assistance of `typedef` s:

```
typedef char Word[255];     // type array of 255 chars
Word& pa=a;                 // reference to an array of 255 chars
Word* pa=&a;                // pointer to an array of 255 chars
```

Objects and lvalues

- From *The Annotated C++ Reference Manual*: "An *object* is a region of storage. An *lvalue* is an expression referring to an object or function."
- Originally, the terms "lvalue" and "rvalue" referred to things that appeared on the left and right sides of assignments, but now "lvalue" is more general.
- The simplest examples of lvalues are names of objects, i.e., variables:

```
int n;
n = 44; // n is a lvalue
```

- The simplest examples of things that are not lvalues are literals:

```
44 = n; // ERROR: 44 is not an lvalue
```

- But symbolic constants are lvalues:

```
const int MAX = 65535; // MAX is an lvalue
```

- even though they cannot appear on the left side of an assignment:

```
MAX = 21024; // ERROR: MAX is constant
```

- lvalues that can appear on the left side of an assignment are called *mutable lvalues*; those that cannot are called *immutable lvalues*: a variable is a mutable lvalue; a constant is an immutable lvalue.

Objects and lvalues

- Other examples of mutable lvalues include subscripted variables and dereferenced pointers:

```
int a[8];
a[5] = 22; // a[5] is a mutable lvalue
int* p = &n;
*p = 77; // *p is a mutable lvalue
```

- Other examples of immutable lvalues include arrays, functions, and references.
- In general, an lvalue is anything whose address is accessible. Since an address is what a reference variable needs when it is declared, the C++ syntax requirement for such a declaration specifies an lvalue:

```
type& refname = lvalue;
```

- For example, this is a legal declaration of a reference:

```
int& r = n; // OK: n is a lvalue
```

- but these are illegal:

```
int& r = 44;        //ERROR: 44 is not a lvalue
int& r = n++;       //ERROR: n++ is not a lvalue
int& r = cube(n);   //ERROR: cube(n) is not a lvalue
```

Returning a reference

- A function’s return type may be a reference provided that the value returned is an lvalue which is not local to the function.
- This restriction means that the returned value is actually a reference to an lvalue that exists after the function terminates. Consequently that returned lvalue may be used like any other lvalue; for example, on the left side of an assignment.

```
int& max(int& m, int& n)  // return type is reference to int
{
  return (m > n ? m : n); // return type is reference to int
}

int main()
{
  int m = 44, n = 22;
  cout << m << ", " << n << ", " << max(m,n) << endl;
  max(m,n) = 55;    // changes the value of m from 44 to 55
  cout << m << ", " << n << ", " << max(m,n) << endl;
  return 0;
}
```

```
44, 22, 44
55, 22, 55
```

The `max()` function returns a reference to the larger of the two variables passed to it. Since the `return` value is a <u>reference</u>, the expression `max(m,n)` acts like a reference to `m` (since `m` is larger than `n`). So assigning 55 to the expression `max(m,n)` is equivalent to assigning it to `m` itself.

Arrays and pointers

- Although pointer types are not integer types, some integer arithmetic operators can be applied to pointers.
- The affect of this arithmetic is to cause the pointer to point to another memory location.
- The actual change in address depends upon the size of the fundamental type to which the pointer points.
- Pointers can be incremented and decremented like integers. However, the increase or decrease in the pointer's value is equal to the size of the object to which it points.

```
int main()
{ const int SIZE = 3;
  short a[SIZE] = {22, 33, 44};
  cout << "a = " << a << endl;
  cout << "sizeof(short) = " << sizeof(short) << endl;
  short* end = a + SIZE;  // converts SIZE to offset 6
  short sum = 0;
  for (short* p = a; p < end; p++)
    { sum += *p;
      cout << "\t p = " << p;
      cout << "\t *p = " << *p;
      cout << "\t sum = " << sum << endl;
    }
  cout << "end = " << end << endl;
  return 0;
}
```

```
a = 0x7fff3e149052
sizeof(short) = 2
      p = 0x7fff3e149052     *p = 22   sum = 22
      p = 0x7fff3e149054     *p = 33   sum = 55
      p = 0x7fff3e149056     *p = 44   sum = 99
end = 0x7fff3e149058
```

Arrays and pointers

- Although pointer types are not integer types, some integer arithmetic operators can be applied to pointers.
- The affect of this arithmetic is to cause the pointer to point to another memory location.
- The actual change in address depends upon the size of the fundamental type to which the pointer points.
- Pointers can be incremented and decremented like integers. However, the increase or decrease in the pointer's value is equal to the size of the object to which it points.

```
int main()
{ const int SIZE = 3;
  short a[SIZE] = {22, 33, 44};
  cout << "a = " << a << endl;
  cout << "sizeof(short) = " << sizeof(short) << endl;
  short* end = a + SIZE;  // converts SIZE to offset 6
  short sum = 0;
  for (short* p = a; p < end; p++)
    { sum += *p;
      cout << "\t p = " << p;
      cout << "\t *p = " << *p;
      cout << "\t sum = " << sum << endl;
    }
  cout << "end = " << end << endl;
  return 0;
}
```

```
a = 0x7fff3e149052
sizeof(short) = 2
      p = 0x7fff3e149052     *p = 22   sum = 22
      p = 0x7fff3e149054     *p = 33   sum = 55
      p = 0x7fff3e149056     *p = 44   sum = 99
end = 0x7fff3e149058
```

On this machine short integers occupy 2 bytes; since p is a pointer to short, each time it is incremented it advances 2 bytes to the next short integer in the array. That way, sum += *p accumulates their sum of the integers. If p were a pointer to double and sizeof(double) were 8 bytes, then each time p is incremented it would advance 8 bytes.

Arrays and pointers

- Although pointer types are not integer types, some integer arithmetic operators can be applied to pointers.
- The affect of this arithmetic is to cause the pointer to point to another memory location.
- The actual change in address depends upon the size of the fundamental type to which the pointer points.
- Pointers can be incremented and decremented like integers. However, the increase or decrease in the pointer's value is equal to the size of the object to which it points.

```
int main()
{ const int SIZE = 3;
  short a[SIZE] = {22, 33, 44};
  cout << "a = " << a << endl;
  cout << "sizeof(short) = " << sizeof(short) << endl;
  short* end = a + SIZE;  // converts SIZE to offset 6
  short sum = 0;
  for (short* p = a; p < end; p++)
    { sum += *p;
      cout << "\t p = " << p;
      cout << "\t *p = " << *p;
      cout << "\t sum = " << sum << endl;
    }
  cout << "end = " << end << endl;
  return 0;
}
```

```
a = 0x7fff3e149052
sizeof(short) = 2
      p = 0x7fff3e149052    *p = 22   sum = 22
      p = 0x7fff3e149054    *p = 33   sum = 55
      p = 0x7fff3e149056    *p = 44   sum = 99
end = 0x7fff3e149058
```

On this machine short integers occupy 2 bytes; since p is a pointer to short, each time it is incremented it advances 2 bytes to the next short integer in the array. That way, sum += *p accumulates their sum of the integers. If p were a pointer to double and sizeof(double) were 8 bytes, then each time p is incremented it would advance 8 bytes.

The `new` operator

- When a pointer is declared like this:

```
float* p; //p is a pointer to a float
```

- it only allocates memory for the pointer itself. The value of the pointer will be some memory address, but the memory at that address is not yet allocated.
- This means that storage could already be in use by some other variable. In this case, p is uninitialized: it is not pointing to any allocated memory. Any attempt to access the memory to which it points will be an error:

```
*p = 3.14159; // ERROR: no storage has been allocated for *p
```

- A good way to avoid this problem is to initialize pointers when they are declared:

```
float x = 3.14159;   // x contains the value 3.14159
float* p = &x;       // p contains the address of x
cout << *p;          // OK: *p has been allocated
```

- In this case, accessing `*p` is not a problem because the memory needed to store the float 3.14159 was automatically allocated when x was declared; p points to the same allocated memory.
- Another way to avoid the problem of a dangling pointer is to allocate memory explicitly for the pointer itself. This is done with the new operator:

```
float* q;
q = new float; // allocates storage for 1 float
*q = 3.14159;  // OK: *q has been allocated
```

The **new** operator

- The new operator returns the address of a block of s unallocated bytes in memory, where s is the size of a float. (Typically, sizeof(float) is 4 bytes.) Assigning that address to q guarantees that *q is not currently in use by any other variables.
- The first two of these lines can be combined, thereby initializing q as it is declared:

```
float* q = new float;
```

- Note that using the `new` operator to initialize `q` only initializes the pointer itself, not the memory to which it points.
- It is possible to do both in the same statement that declares the pointer:

```
float* q = new float(3.14159);
cout << *q; // ok: both q and *q have been initialized
```

- In the unlikely event that there is not enough free memory to allocate a block of the required size, the new operator will return 0 (the NULL pointer):

```
double* p = new double;
if (p == 0) abort(); // allocator failed: insufficient memory
else *p = 3.141592658979324;
```

  - This prudent code calls an abort() function to prevent dereferencing the NULL pointer.
- Consider again the two alternatives to allocating memory:

```
float x = 3.14159;            // allocates named memory
float* p = new float(3.14159); // allocates unnamed memory
```

- In the first case, memory is allocated at compile time to the named variable x . In the second case, memory is allocated at run time to an unnamed object that is accessible through *p .

The `delete` operator

- The delete operator reverses the action of the new operator, returning allocated memory to the free store. It should only be applied to pointers that have been allocated explicitly by the new operator:

```
float* q = new float(3.14159);
delete q;      // deallocates q
*q = 2.71828; // ERROR: q has been deallocated
```

- Deallocating `q` returns the block of `sizeof(float)` bytes to the free store, making it available for allocation to other objects.
- Once q has been deallocated, it should not be used again until after it has been reallocated. A deallocated pointer, also called a dangling pointer, is like an uninitialized pointer: it doesn't point to anything.
- A pointer to a constant cannot be deleted:

```
const int * p = new int;
delete p; // ERROR: cannot delete pointer to const
```

  - This restriction is consistent with the general principle that constants cannot be changed.
- Using the `delete` operator for fundamental types (`char`, `int`, `float`, `double` , etc.) is generally <u>not recommended</u> because little is gained at the risk of a potentially disastrous error:

```
float x = 3.14159; // x contains the value 3.14159
float* p = &x; // p contains the address of x
delete p;      // RISKY: p was not allocated by new
```

- This would deallocate the variable `x` , a mistake that can be very difficult to debug.

Dynamic arrays

- An **array name** is really **just a constant pointer** that is allocated at compile time:

```
float a[20]; // a is a const pointer to a block of 20 floats
float* const p = new float[20]; // so is p
```

- Both `a` and `p` are constant pointers to blocks of 20 floats. The declaration of `a` is called *static binding* because it is allocated at compile time; the symbol `a` is bound to the allocated memory even if the array is never used while the program is running.
- In contrast, a non-constant pointer can be used to postpone the allocation of memory until the program is running. This is generally called *run-time binding* or *dynamic binding*:

```
float* p = new float[20];
```

  - An array that is declared this way is called a dynamic array.
- Compare the two ways of defining an array:

```
float a[20]; // static array
float* p = new float[20]; // dynamic array
```

- The static array `a` is created at compile time; its memory remains allocated throughout the run of the program. The dynamic array p is created at run time; its memory allocated only when its declaration executes. Furthermore, the memory allocated to the array `p` is deallocated as soon as the delete operator is invoked on it:

```
delete [] p; // deallocates the array p
```

- Note that the subscript operator `[]` must be included this way, because `p` is an array.

Using dynamic arrays

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

void get(double*& a, int& n)
{ cout << "Enter number of items: "; cin >> n;
  a = new double[n];
  cout << "Enter " << n << " items, one per line:\n";
  for (int i = 0; i < n; i++)
    { cout << "\t" << i+1 << ": ";
      cin >> a[i];
    }
}
//----------------------------------------
void print(double* a, int n)
{
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}
//-----------------------------------------
int main()
{
  double* a; // a is simply an unallocated pointer
  int n;
  get(a,n); // now a is an array of n doubles
  print(a,n);
  delete [] a; // now a is simply an unallocated pointer again
  get(a,n); // now a is an array of n doubles
  print(a,n);
  return 0;
}
```

```
Enter number of items: 4
Enter 4 items, one per line:
        1: 6.78
        2: 5.77
        3: 8.12
        4: 9.03
6.78 5.77 8.12 9.03
Enter number of items: 2
Enter 2 items, one per line:
        1: 5.7
        2: 8.1
5.7 8.1
```

Using dynamic arrays

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

void get(double*& a, int& n)
{ cout << "Enter number of items: "; cin >> n;
  a = new double[n];
  cout << "Enter " << n << " items, one per line:\n";
  for (int i = 0; i < n; i++)
    { cout << "\t" << i+1 << ": ";
      cin >> a[i];
    }
}
//---------------------------------------
void print(double* a, int n)
{
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}
//---------------------------------------
int main()
{
  double* a; // a is simply an unallocated pointer
  int n;
  get(a,n); // now a is an array of n doubles
  print(a,n);
  delete [] a; // now a is simply an unallocated pointer again
  get(a,n); // now a is an array of n doubles
  print(a,n);
  return 0;
}
```

```
Enter number of items: 4
Enter 4 items, one per line:
        1: 6.78
        2: 5.77
        3: 8.12
        4: 9.03
6.78 5.77 8.12 9.03
Enter number of items: 2
Enter 2 items, one per line:
        1: 5.7
        2: 8.1
5.7 8.1
```

The new operator allocates storage for n double s after the value of n is obtained interactively. The array is created "on the fly" while the program is running.

Using dynamic arrays

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

void get(double*& a, int& n)
{ cout << "Enter number of items: "; cin >> n;
  a = new double[n];
  cout << "Enter " << n << " items, one per line:\n";
  for (int i = 0; i < n; i++)
    { cout << "\t" << i+1 << ": ";
      cin >> a[i];
    }
}
//----------------------------------------
void print(double* a, int n)
{
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}
//-----------------------------------------
int main()
{
  double* a; // a is simply an unallocated pointer
  int n;
  get(a,n); // now a is an array of n doubles
  print(a,n);
  delete [] a; // now a is simply an unallocated pointer again
  get(a,n); // now a is an array of n doubles
  print(a,n);
  return 0;
}
```

```
Enter number of items: 4
Enter 4 items, one per line:
     1: 6.78
     2: 5.77
     3: 8.12
     4: 9.03
6.78 5.77 8.12 9.03
Enter number of items: 2
Enter 2 items, one per line:
     1: 5.7
     2: 8.1
5.7 8.1
```

The new operator allocates storage for n double s after the value of n is obtained interactively. The array is created “on the fly” while the program is running.

Before get() is used to create another array the current array has to be deallocated with the delete operator. Note that the subscript operator [] must be specified when deleting an array.

Using dynamic arrays

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

void get(double*& a, int& n)
{ cout << "Enter number of items: "; cin >> n;
  a = new double[n];
  cout << "Enter " << n << " items, one per line:\n";
  for (int i = 0; i < n; i++)
    { cout << "\t" << i+1 << ": ";
      cin >> a[i];
    }
}
//----------------------------------------
void print(double* a, int n)
{
  for (int i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
}
//-----------------------------------------
int main()
{
  double* a; // a is simply an unallocated pointer
  int n;
  get(a,n); // now a is an array of n doubles
  print(a,n);
  delete [] a; // now a is simply an unalloc
  get(a,n); // now a is an array of n double
  print(a,n);
  return 0;
}
```

```
Enter number of items: 4
Enter 4 items, one per line:
     1: 6.78
     2: 5.77
     3: 8.12
     4: 9.03
6.78 5.77 8.12 9.03
Enter number of items: 2
Enter 2 items, one per line:
     1: 5.7
     2: 8.1
5.7 8.1
```

The new operator allocates storage for n double s after the value of n is obtained interactively. The array is created “on the fly” while the program is running.

Before get() is used to create another array the current array has to be deallocated with the delete operator. Note that the subscript operator [] must be specified when deleting an array.

The array parameter a is a pointer that is passed by reference:
void get(double*& a, int& n)
This is necessary because the new operator will change the value of a which is the address of the first element of the newly allocated array.

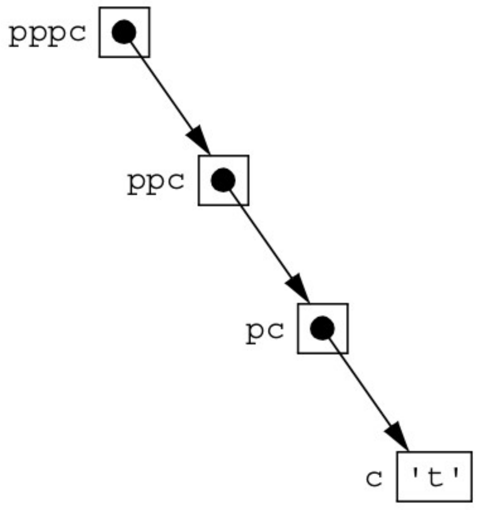
Pointer5.cpp

Pointers to pointers

- A pointer may point to another pointer. For example,

```
char c = 't';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'w'; // changes value of c to 'w'
```

- We can visualize these variables like this:
  - The assignment ***pppc = 'w' refers to the contents of the address pc that is pointed to by the address ppc that is pointed to by the address pppc .



Pointers to functions

- Like an array name, a function name is actually a constant pointer.
- We can think of its value as the address of the code that implements the function. A pointer to a function is simply a pointer whose value is the address of the function name.
- Since that name is itself a pointer, a pointer to a function is just a pointer to a constant pointer.
- For example:

```
int f(int);          // declares function f
int (*pf)(int); // declares function pointer pf
pf = &f;            // assigns address of f to pf
```

- We can visualize the function pointer like this: The value of function pointers is that they allow us to define functions of functions. This is done by passing a function pointer as a parameter to another function.

Example : The Sum of a Function

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

int sum(int (*)(int), int);
int square(int);
int cube(int);

int main()
{
  cout << sum(square,4) << endl;
  // 1 + 4 + 9 + 16
  cout << sum(cube,4) << endl;
  // 1 + 8 + 27 + 64
}

int sum(int (*pf)(int k), int n)
{ // returns the sum f(0) + f(1) + f(2) + . . . + f(n-1):
  int s = 0;
  for (int i = 1; i <= n; i++)
    s += (*pf)(i);
  return s;
}
int square(int k)
{
  return k*k;
}
int cube(int k)
{
  return k*k*k;
}
```

```
30
100
```

The sum() function evaluates the function to which pf points, at each of the integers 1 through n , and returns the sum of these n values.

Note that the declaration of the function pointer parameter pf in the sum() function's parameter list requires the dummy variable k .

NUL, NULL, and `void`

- The constant 0 (zero) has type int. Nevertheless, this symbol can be assigned to all the fundamental types:

```
char c = 0;          //initializes c to the char '\0'
short d = 0;     //initializes d to the short int 0
int n = 0;    //initializes n to the int 0
unsigned u = 0;    //initializes u to the unsigned int 0
float x = 0;     //initializes x to the float 0.0
double z = 0;    //initializes z to the double 0.0
```

- In each case, the object is initialized to the number 0. In the case of type char , the character c becomes the null character; denoted by '\0' or NUL , it is the character whose ASCII code is 0.
- The values of pointers are memory addresses. These addresses must remain within that part of memory allocated to the executing process, with the exception of the address 0x0. This is called the NULL pointer. The same constant applies to pointers derived from any type:

```
char* pc = 0;       // initializes pc to NULL
short* pd = 0;      // initializes pd to NULL
int* pn = 0;        // initializes pn to NULL
unsigned* pu = 0; // initializes pu to NULL
float* px = 0;      // initializes px to NULL
double* pz = 0;     // initializes pz to NULL
```

NUL, NULL, and `void`

- The NULL pointer cannot be dereferenced. This is a common but fatal error:

```
int* p = 0;
*p = 22; // ERROR: cannot dereference the NULL pointer
```

- A reasonable precaution is to test a pointer before attempting to dereference it:

```
if (p) *p = 22; // ok
```

- This tests the condition (p != NULL) because that condition is true precisely when p is nonzero.
- The name void denotes a special fundamental type. Unlike all the other fundamental types,void can only be used in a derived type:

```
void x; // ERROR: no object can have type void
void* p; // OK
```

- The most common use of the type void is to specify that a function does not return a value:

```
void swap(double&, double&);
```

- Another, different use of void is to declare a pointer to an object of unknown type:

```
void* p = q;
```

- This use is most common in low-level C programs designed to manipulate hardware resources.

Esercitazione 6

- Exercise 1 (pointerEs1.cpp)
  Write a program that:

- declares a variable double a and a double * aPtr
- assign to aPtr the address of a
- assign to a value of 5 using aPtr (i.e. it is forbidden to write a = 5)
- print a and aPtr
- Multiply by 2 using aPtr (i.e. it is forbidden to write a = a * 2.)
- print a and aPtr

- Exercise 2 (pointerEs2.cpp)

Write a program that:

- create an array of integers of size n
- assigns to the elements of this array the values 0, 1, 2, 3, ..., n using the pointer arithmetic
- print the array

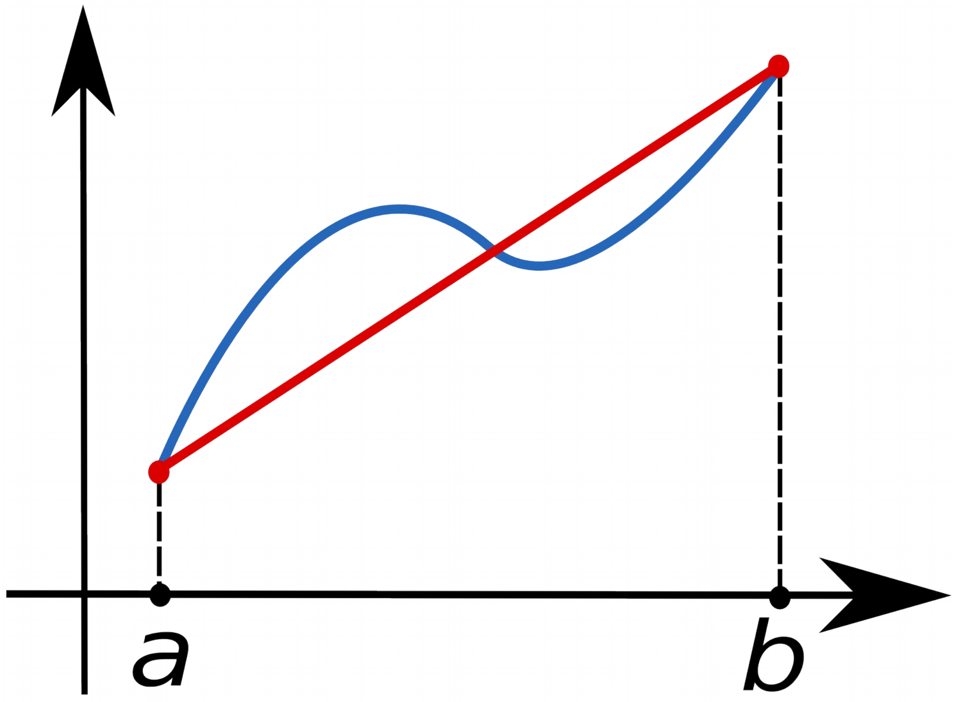
```
./pointerEs2
 a [0] = 0; aPtr = 0xbfffe110; * aPtr = 0
 a [1] = 1; aPtr = 0xbfffe118; * aPtr = 1
 a [2] = 2; aPtr = 0xbfffe120; * aPtr = 2
 a [3] = 3; aPtr = 0xbfffe128; * aPtr = 3
 a [4] = 4; aPtr = 0xbfffe130; * aPtr = 4
 a [5] = 5; aPtr = 0xbfffe138; * aPtr = 5
 a [6] = 6; aPtr = 0xbfffe140; * aPtr = 6
 a [7] = 7; aPtr = 0xbfffe148; * aPtr = 7
 a [8] = 8; aPtr = 0xbfffe150; * aPtr = 8
 a [9] = 9; aPtr = 0xbfffe158; * aPtr = 9
```

Interval : Numerical integration

Trapezoidal rule

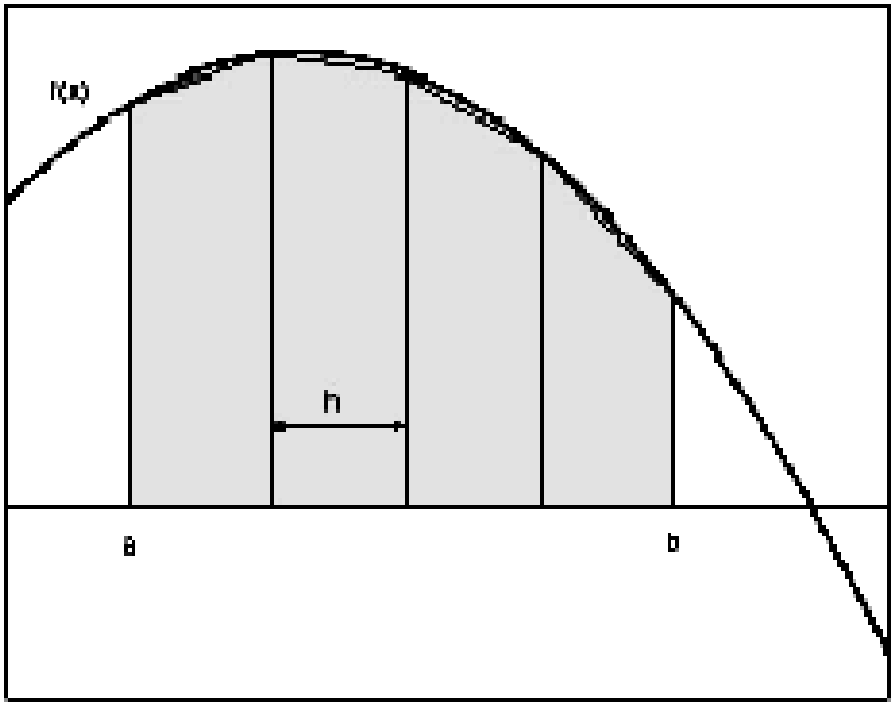
- The integral of a function is approximated with the area of a trapezium with vertexes: (a,f(a)), (b,f(b)), (b,0) e (a,0).
- This approximation is valid only if in the function in the considered interval is ~ flat.
- If this is not valid, the full range can be divided into N subintervals.

$$\int_a^b f(x)\,dx \approx (b-a)\frac{f(a)-f(b)}{2}$$



Trapezoidal rule

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2}+\sum_{k=1}^{n-1} f\left(a+k\frac{b-a}{n}\right)\right)$$



Esercitazione 6

- **Exercise 3**

Calculate the integral of a function y = f (x) with the trapezoidal method:

area = DeltaX * ((y (0) + y (n)) / 2 + (y (1) + y (2) + ... + y (n-1)))

where n is the number of sub-intervals in which the integration domain and DeltaX is the amplitude of each sub-interval.

The function which has been implemented in the example is log10(x). You can implement as **external** function the one that you like the most.

y (i-1) and y (i) are the values assumed by the function at the lower end and at the top of the i-th interval.

The user can specify the integration range and the number of sub-intervals during program execution.

The program must consist of:

*a main program,

*a function that calculates the values assumed by the integrand function at the ends of the sub-intervals and the integral with the trapezoidal rule.

– Suggestion: use an array to store the values of the function at the ends of each sub-intervals: double func[n];

(main: useTrapezioidalntegration.cpp

Function: TrapezioidalIntegration.{cpp,h})

```
Execution example:
./useTrapeziodalIntegration
Calculation of an integral with the Trapezium
method
Low value of the integration interval: 1
High value of the integration interval: 2
Number of sub-intervals: 20
The integral of the function in the interval 1
2is :0.16772
```

Esercitazione 6

Exercise 4

Using the function that calculates the integral with the trapezoidal method developed for the previous exercise, write a program so that the user can specify the desired accuracy.

Tips:

- the user gives the epsilon parameter from the keyboard. The integral must be calculated in an iterative way, doubling the subintervals at each iteration, until (abs (area- oldArea) < epsilon * abs (area)).

 area is the value of the integral in the current iteration, oldArea is the value in the previous iteration. abs is the absolute value (i.e. you have to include cmath).

- make sure that there are at least 3-4 iterations to avoid accidental convergences

- end the iterative process even in the absence of convergence after a maximum number of pre-set iterations

Execution example:

```
./useTrapezoidalIntegration2
Calculation of an integral with the Trapeziums method
Low value of the integration interval: 1
High value of the integration interval: 2
Precision: 0.01
The integral is: 0.167748
```

Esercitazione 6

Exercise 5 (Derivative.{cpp,h}, UseDerivative.cpp)

Write a function that returns the numerical derivative of a given function at a given point x, using a given tolerance h. Use the formula

$$f'(x) = (f(x+h)-f(x-h))/(2h)$$

This derivative() function has three arguments: a pointer to the function f, the x value, and the tolerance h.

In this exercise you have to implement and use the **`cube()`** function.

```
./derivative
Derivative example
x: 1
Tolerance: 0.001
The derivative of cube function in x=1 is 3
```

Characters and Strings

C-string

- A C-string (also called a character string) is a sequence of contiguous characters in memory terminated by the `NUL` character `'\0'` .
- C-strings are accessed by variables of type `char*` (pointer to char).
- For example, if s has type char* , then

  ```
  cout << s << endl;
  ```

  will print all the characters stored in memory beginning at the address s and ending with the first occurrence of the NUL character.
- The C header file **`<cstring>`** provides a wealth of special functions for manipulating C-strings.
- For example, the call `strlen(s)` will return the number of characters in the C-string `s`, not counting its terminating NUL character. These functions all declare their C-string parameters as pointers to char.

Fundamentals of Characters and Strings

- String assignment
  - Character array
    - **`char color[] = "blue";`**
    - Creates 5 element **`char`** array **`color`**
      - last element is **`'\0'`**
  - Variable of type **`char *`**
    - **`char *colorPtr = "blue";`**
      - Creates pointer **`colorPtr`** to letter **`b`** in string **`“blue”`**
        - **`“blue”`** somewhere in memory
  - Alternative for character array
    - **`char color[] = { ‘b’, ‘l’, ‘u’, ‘e’, ‘\0’ };`**

Fundamentals of Characters and Strings

- Reading strings
  - Assign input to character array **`word[ 20 ]`**

```
cin >> word
```

  - Reads characters until whitespace or EOF
  - String could exceed array size

```
cin >> setw( 20 ) >> word;
```

  - Reads 19 characters (space reserved for **`'\0'`**)

Some `cin` member functions

- The input stream object cin includes the input functions: `cin.getline()` , `cin.get()`,`cin.ignore()`, `cin.putback()`, and `cin.peek()` .
- Each of these function names includes the prefix "`cin.`" because they are "member functions" of the cin object.
- **`cin.getline(str,n)`** reads up to n characters into str and ignores the rest.
- **`cin.get()`** is used for reading input character-by-character. The call `cin.get(ch)` copies the next character from the input stream cin into the variable ch and returns 1, unless the end of file is detected in which case it returns 0.
- **`cout.put()`** is the opposite of get is put . The function is used for writing to the output stream cout character-by-character.
- **`cin.putback()`** function restores the last character read by a `cin.get()` back to the input stream cin .
- **`cin.ignore()`** function reads past one or more characters in the input stream cin without processing them.
- **`cin.peek()`** function can be used in place of the combination cin.get() and cin.putback() functions. The call ch = cin.peek() copies the next character of the input stream cin into the char variable ch without removing that character from the input stream.
- The header file **`<ctype.h>`** declares the function `toupper(ch)` which returns the uppercase equivalent of ch if ch is a lowercase letter.

String Manipulation Functions of the String-handling Library

`char *strcpy( char *s1, const char *s2 );`	Copies the string **`s2`** into the character array **`s1`**. The value of **`s1`** is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most **`n`** characters of the string **`s2`** into the character array **`s1`**. The value of **`s1`** is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string **`s2`** to the string **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most **`n`** characters of string **`s2`** to string **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string **`s1`** with the string **`s2`**. The function returns a value of zero, less than zero or greater than zero if **`s1`** is equal to, less than or greater than **`s2`**, respectively.

String Manipulation Functions of the String-handling Library

`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to **`n`** characters of the string **`s1`** with the string **`s2`**. The function returns zero, less than zero or greater than zero if **`s1`** is equal to, less than or greater than **`s2`**, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **`strtok`** breaks string **`s1`** into “tokens”—logical pieces such as words in a line of text—delimited by characters contained in string **`s2`**. The first call contains **`s1`** as the first argument, and subsequent calls to continue tokenizing the same string contain **`NULL`** as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, **`NULL`** is returned.
`size_t strlen( const char *s );`	Determines the length of string **`s`**. The number of characters preceding the terminating null character is returned.

String Manipulation Functions of the String-handling Library

- Copying strings
  - **`char *strcpy( char *s1, const char *s2 )`**
    - Copies second argument into first argument
      - First argument must be large enough to store string and terminating null character
  - **`char *strncpy( char *s1, const char *s2, size_t n )`**
    - Specifies number of characters to be copied from string into array
    - Does not necessarily copy terminating null character

The standard C++ `string` type

- Standard C++ defines its string type in the **`<string>`** header file. Objects of type string can be declared and initialized in several ways:

```
string s1;                // s1 contains 0 characters
string s2 = "New York";  // s2 contains 8 characters
string s3(60, '*');       // s3 contains 60 asterisks
string s4 = s3;           // s4 contains 60 asterisks
string s5(s2, 4, 2);      // s5 is the 2-character string
"Yo"
```

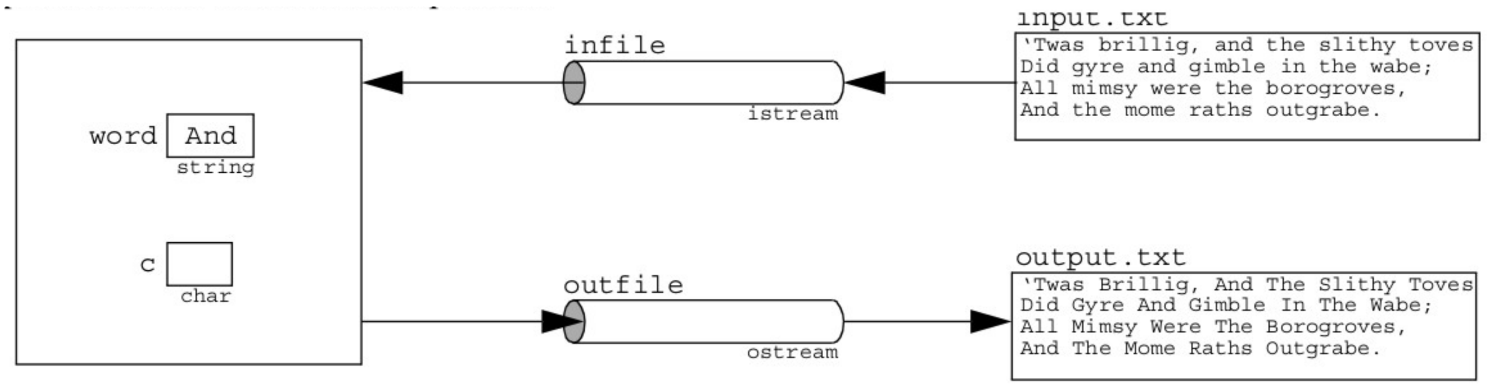
- If the string is not initialized, like s1 here, then it represents the empty string containing 0 characters.
- A string can be initialized the same way a C-string is, like s2
- A string can be initialized to hold a given number of the same character, like s3 here which holds 60 stars.
- Unlike a C-string, C++ string objects can be initialized with a copy of another existing string object, like s4, or with a substring of an existing string, like s5 .
- Note that the standard substring designator has three parts: the parent string (s2 , here), the starting character (s2[4] , here), and the length of the substring (2 , here).

Files

- File processing in C++ is very similar to ordinary interactive input and output because the same kind of stream objects are used.
- Input from a file is managed by an **`ifstream`** object the same way that input from the keyboard is managed by the **`istream`** object **`cin`**
- Similarly, output to a file is managed by an **`ofstream`** object the same way that output to the monitor or printer is managed by the **`ostream`** object **`cout`** .
- The only difference is that **`ifstream`** and **`ofstream`** objects have to be declared explicitly and initialized with the external name of the file which they manage.
- You also have to **`#include`** the **`<fstream>`** header file (or <fstream.h> in pre-Standard C++) that defines these classes.

EXAMPLE: Capitalizing All the Words in a Text File

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
  ifstream infile("input.txt");
  ofstream outfile("output.txt");
  string word;
  char c;
  while (infile >> word)
    { if (word[0] >= 'a' && word[0] <= 'z') word[0] += 'A' - 'a';
      outfile << word;
      infile.get(c);
      outfile.put(c);
    }
  return 0;
}
```



```
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogroves,
And the mome raths outgrabe.
```

```
'Twas Brillig, And The Slithy Toves
Did Gyre And Gimble In The Wabe;
All Mimsy Were The Borogroves,
And The Mome Raths Outgrabe.
```

Esercitazione 6- Esericizio Facoltativo

Implement and test the following function:

```
bool is_palindrome(string s);
// Returns true iff s is a palindrome
// EXAMPLES: is_palindrome("RADAR") returns true,
// is_palindrome("ABCD") returns false
```

IsPalindrome.cpp

Example on how to manipulate strings

String Manipulation Functions of the String-handling Library

```cpp
 // Fig. 5.28: fig05_28.cpp
 // Using strcpy and strncpy.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <cstring>    // prototypes for strcpy and strncpy

int main()
{
   char x[] = "Happy Birthday to You";
   char y[ 25 ];
   char z[ 15 ];

   strcpy( y, x );  // copy contents of x into y

   cout << "The string in array x is: " << x
        << "\nThe string in array y is: " << y

   // copy first 14 characters of x into z
   strncpy( z, x, 14 );  // does not copy null characte
   z[ 14 ] = '\0';       // append '\0' to z's contents

   cout << "The string in array z is: " << z << endl;

   return 0;  // indicates successful termination

} // end main
```

<cstring> contains prototypes for **strcpy** and **strncpy**

Copy entire string in array **x** into array **y**

Copy first 14 characters of array **x** into array **z**. Note that this does not write terminating null character

Append terminating null character

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

String to copy.

Copied string using **strcpy**.

Copied first 14 characters using **strncpy**

String Manipulation Functions of the String-handling Library

- Concatenating strings
  - **`char *strcat( char *s1, const char *s2 )`**
    - Appends second argument to first argument
    - First character of second argument replaces null character terminating first argument
    - Ensure first argument large enough to store concatenated result and null character
  - **`char *strncat( char *s1, const char *s2, size_t n )`**
    - Appends specified number of characters from second argument to first argument
    - Appends terminating null character to result

String Manipulation Functions of the String-handling Library

```
 // Fig. 5.29: fig05_29.cpp
 // Using strcat and strncat.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <cstring>   // prototypes for strcat and strncat

int main()
{
   char s1[ 20 ] = "Happy ";
   char s2[] = "New Year ";
   char s3[ 40 ] = "";

   cout << "s1 = " << s1 << "\ns2 = " << s2;

   strcat( s1, s2 );  // concatenate s2 to s1

   cout << "\n\nAfter strcat(s1,
        << "\ns2 = " << s2;

   // concatenate first 6 characters of s1 to s3
   strncat( s3, s1, 6 );  // places '\0' after last character

```

<cstring> contains prototypes for **strcat** and **strncat**

Append **s2** to **s1**

Append first 6 characters of **s1** to **s3**

String Manipulation Functions of the String-handling Library

```
      cout << "\n\nAfter strncat                  " << s1
           << "\ns3 = " << s3;

      strcat( s3, s1 );  // concatenate s1 to s3
      cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
           << "\ns3 = " << s3 << endl;

      return 0;  // indicates successful termination

   } // end main
```

Append **s1** to **s3**

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

String Manipulation Functions of the String-handling Library

- Comparing strings
  - Characters represented as numeric codes
    - Strings compared using numeric codes
  - Character codes / character sets
    - ASCII
      - “American Standard Code for Information Interchage”
    - EBCDIC
      - “Extended Binary Coded Decimal Interchange Code”

String Manipulation Functions of the String-handling Library

- Comparing strings
  - **`int strcmp( const char *s1, const char *s2 )`**
    - Compares character by character
    - Returns
      - Zero if strings equal
      - Negative value if first string less than second string
      - Positive value if first string greater than second string
  - **`int strncmp( const char *s1,`**
    `const char *s2, size_t n )`
    - Compares up to specified number of characters
    - Stops comparing if reaches null character in one of arguments

String Manipulation Functions of the String-handling Library

```
 // Fig. 5.30: fig05_30.cpp
 // Using strcmp and strncmp.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

#include <cstring>  // prototypes for strcmp and strncmp

int main()
{
   char *s1 = "Happy New Year";
   char *s2 = "Happy New Year";
   char *s3 = "Happy Holidays";

   cout << "s1 = " << s1 << "\ns2 = " << s
        << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2)
        << setw( 2 ) << strcmp( s1, s2 )
        << "\nstrcmp(s1, s3) = " << setw( 2 )
        << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
        << setw( 2 ) << strcmp( s3, s1 );
```

<cstring> contains prototypes for **strcmp** and **strncmp**.

Compare **s1** and **s2**.

Compare **s1** and **s3**.

Compare **s3** and **s1**.

String Manipulation Functions of the String-handling Library

```

   cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
        << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
        << setw( 2 ) << strncmp( s1, s3, 7 )
        << "\nstrncmp(s3, s1, 7) = "
        << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;

   return 0;  // indicates successful termination

} // end main
```

Compare up to 6 characters of **`s1`** and **`s3`**

Compare up to 7 characters of **`s1`** and **`s3`**

Compare up to 7 characters of **`s3`** and **`s1`**

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

String Manipulation Functions of the String-handling Library

- Tokenizing
  - Breaking strings into tokens, separated by delimiting characters
  - Tokens usually logical units, such as words (separated by spaces)
  - **`"This is my string"`** has 4 word tokens (separated by spaces)
  - **`char *strtok( char *s1, const char *s2 )`**
    - Multiple calls required
      - First call contains two arguments, string to be tokenized and string containing delimiting characters
        - Finds next delimiting character and replaces with null character
    - Subsequent calls continue tokenizing
      - Call with first argument **`NULL`**

String Manipulation Functions of the String-handling Library

```
 // Fig. 5.31: fig05_31.cpp
 // Using strtok.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <cstring>  // prototype for strtok

int main()
{
   char sentence[] = "This is a sentence with 7 tokens";
   char *tokenPtr;

   cout << "The string to be tokenized
        << "\n\nThe tokens are:\n\n";

   // begin tokenization of sentence
   tokenPtr = strtok( sentence, " " );

   // continue tokenizing sentence until tokenPtr becomes NULL
   while ( tokenPtr != NULL ) {
      cout << tokenPtr << '\n';
      tokenPtr = strtok( NULL, " " );  // get next token

   } // end while

   cout << "\nAfter strtok, sentence

   return 0;  // indicates successful

} // end main
```

`<cstring>` contains prototype for **`strtok`**

First call to **`strtok`** begins tokenization

Subsequent calls to **`strtok`** with **`NULL`** as first argument to indicate continuation

String Manipulation Functions of the String-handling Library

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

String Manipulation Functions of the String-handling Library

- Determining string lengths
  - **`size_t strlen( const char *s )`**
    - Returns number of characters in string
      - Terminating null character not included in length

String Manipulation Functions of the String-handling Library

```
// Fig. 5.32: fig05_32.cpp
// Using strlen.
#include <iostream>

using std::cout;
using std::endl;

#include <cstring>  // prototype for strlen

int main()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   cout << "The length of \"" << string1
        << "\" is " << strlen( string1 )
        << "\nThe length of \"" << string2
        << "\" is " << strlen( string2 )
        << "\nThe length of \"" << string3
        << "\" is " << strlen( string3 ) << endl;

   return 0;  // indicates successful termination

} // end main
```

<cstring> contains prototype for **strlen**.

Using **strlen** to determine length of strings

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fundamentals of Characters and Strings

- Character constant
  - Integer value represented as character in single quotes
  - **`'z'`** is integer value of **`z`**

    `122` in ASCII
- String
  - Series of characters treated as single unit
  - Can include letters, digits, special characters **`+`**, **`-`**, **`*`** ...
  - String literal (string constants)

    Enclosed in double quotes, for example:

    `"I like C++"`
  - Array of characters, ends with null character **`'\0'`**
  - String is constant pointer
    - Pointer to string’s first character
    - Like arrays